

# Cyber-Security Foundations

## Part I: Applied Cryptography

©Amir Herzberg

Department of Computer Science and Engineering  
University of Connecticut

March 17, 2020

# Contents

Contents	ii
<b>1 Introduction</b>	<b>1</b>
1.1 About this textbook . . . . .	1
1.2 Brief History of Computers, Cryptology and Cyber . . . . .	4
1.2.1 From classical to modern Cryptology . . . . .	6
1.2.2 Cryptology is not just about secrecy! . . . . .	8
1.3 Cybersecurity Goals and Attack Models . . . . .	9
1.4 Notations . . . . .	11
<b>2 Encryption and Pseudo-Randomness</b>	<b>12</b>
2.1 From Ancient Ciphers to Kerckhoffs' Principle . . . . .	14
2.1.1 Ancient Keyless Monoalphabetic Ciphers . . . . .	15
2.1.2 Shift Cipher: a Keyed Variant of the Caesar Cipher . .	18
2.1.3 Mono-alphabetic Substitution Ciphers . . . . .	19
2.1.4 Kerckhoffs' known-design principle . . . . .	21
2.2 Cryptanalysis Attack Models: CTO, KPA, CPA and CCA . .	22
2.3 Generic attacks and the Key-Length Principle . . . . .	23
2.3.1 Exhaustive search . . . . .	24
2.3.2 Table Look-up and Time-memory tradeoff attacks . .	25
2.3.3 The sufficient effective key length principle . . . . .	26
2.4 Unconditional security and One Time Pad . . . . .	27
2.5 Stream Ciphers and Pseudo-Randomness . . . . .	28
2.5.1 Stream Ciphers . . . . .	28
2.5.2 Pseudo-Random Generators (PRGs) . . . . .	29
2.5.3 Secure PRG Constructions . . . . .	33
2.5.4 Pseudo-random functions (PRFs) . . . . .	35
2.5.5 PRF: Constructions and Robust Combiners . . . . .	39
2.5.6 The key-separation principle and application of PRF .	40
2.6 Defining secure encryption . . . . .	41
2.6.1 Attacker model . . . . .	42
2.6.2 Success criteria . . . . .	43
2.7 The Cryptographic Building Blocks Principle . . . . .	47
2.8 Block Ciphers and Pseudo-Random Permutations (PRPs) . .	49
2.8.1 Constructing PRP from PRF: the Feistel Construction .	50

2.9	Secure Encryption Modes of Operation . . . . .	52
2.9.1	The Electronic Code Book mode (ECB) mode . . . . .	53
2.9.2	The Per-Block Random Mode (PBR) . . . . .	55
2.9.3	The Output-Feedback (OFB) Mode . . . . .	57
2.9.4	The Cipher Feedback (CFB) Mode . . . . .	60
2.9.5	The Cipher-Block Chaining (CBC) mode . . . . .	61
2.9.6	Ensuring CCA Security . . . . .	63
2.10	Case study: the (in)security of WEP . . . . .	63
2.10.1	CRC-then-XOR does not ensure integrity . . . . .	64
2.11	Encryption: Final Words . . . . .	65
2.12	Encryption and Pseudo-Randomness: Additional exercises . . . . .	67
<b>3</b>	<b>Message Authentication Code (MAC) Schemes</b>	<b>81</b>
3.1	Encryption for Authentication? . . . . .	81
3.2	Message Authentication Code (MAC) schemes . . . . .	82
3.3	MAC: definitions and usage . . . . .	83
3.3.1	Usage of MAC . . . . .	84
3.4	Approaches to Constructing MAC . . . . .	86
3.4.1	MAC design ‘from scratch’ . . . . .	86
3.4.2	Robust combiners for MAC . . . . .	87
3.4.3	MAC constructions from cryptographic hash functions .	88
3.5	CBC-MAC: $ln$ -bit Fixed-Input-Length MAC from $l$ -bit PRF/PRP	88
3.5.1	Defining generalized-MAC and FIL-MAC . . . . .	88
3.5.2	CBC-MAC . . . . .	90
3.5.3	Constructing Secure VIL MAC . . . . .	92
3.6	Combining Authentication and Encryption . . . . .	93
3.6.1	Authenticated Encryption (AE) and AEAD schemes .	93
3.6.2	EDC-then-Encrypt Schemes . . . . .	94
3.6.3	Generic Authenticated Encryption Constructions .	95
3.6.4	Single-Key Generic Authenticated-Encryption . . . . .	99
3.7	Message Authentication: Additional exercises . . . . .	100
3.8	Message Authentication: Additional exercises . . . . .	104
<b>4</b>	<b>Cryptographic Hash Functions</b>	<b>108</b>
4.1	Introducing Crypto-hash functions, their goals and applications	108
4.1.1	Warmup: hashing for efficiency . . . . .	108
4.1.2	Goals and requirements for crypto-hashing . . . . .	111
4.1.3	Applications of crypto-hash functions . . . . .	113
4.1.4	Standard cryptographic hash functions . . . . .	114
4.2	Collision Resistant Hash Function (CRHF) . . . . .	115
4.2.1	Birthday attack on collision resistance . . . . .	117
4.2.2	CRHF Applications (1): Integrity and Hash-Block .	118
4.2.3	CRHF Applications (2): Blockchain . . . . .	121
4.2.4	CRHF Applications (3): The Hash-then-Sign (HtS) paradigm	123
4.2.5	Keyless CRHF do not exist - yet are useful ?! . . . . .	125
4.3	Second-preimage resistance (SPR) . . . . .	125

4.3.1	The Chosen-Prefix Vulnerability and its HtS exploit . . . . .	127
4.4	One-Way Functions, aka Preimage Resistance . . . . .	129
4.4.1	Using OWF for One-Time Passwords (OTP) and OTP-chain . . . . .	130
4.4.2	Using OWF for One-Time Signatures . . . . .	131
4.5	Randomness extraction . . . . .	132
4.6	The Random Oracle Methodology . . . . .	135
4.6.1	Proof-of-Work (PoW) and Bitcoin . . . . .	137
4.7	Constructing crypto-hash: from FIL to VIL . . . . .	137
4.7.1	The Merkle hash tree construction . . . . .	137
4.7.2	The Merkle-Damgård Construction . . . . .	140
4.7.3	Some concerns with Merkle-Damgård construction . . . . .	144
4.8	The HMAC and NMAC constructions . . . . .	144
4.9	Crypto-hashing Robust combiners . . . . .	144
4.10	Cryptographic hash functions: additional exercises . . . . .	144
<b>5</b>	<b>Shared-Key Protocols</b>	<b>149</b>
5.1	Introduction . . . . .	149
5.1.1	Inputs and outputs signals . . . . .	150
5.1.2	Focus: two-party shared-key handshake and session protocols . . . . .	151
5.1.3	Adversary Model . . . . .	151
5.1.4	Secure Session Protocols . . . . .	152
5.2	Shared-key Entity-Authenticating Handshake Protocols . . . . .	156
5.2.1	Shared-key entity-authenticating handshake protocols: signals and requirements . . . . .	156
5.2.2	The (insecure) SNA entity-authenticating handshake protocol . . . . .	159
5.2.3	2PP: Three-Flows authenticating handshake protocol . . . . .	161
5.3	Session-Authenticating Handshake Protocols . . . . .	162
5.3.1	Session-authenticating handshake: signals, requirements and variants . . . . .	163
5.3.2	Session-authenticating 2PP . . . . .	164
5.3.3	Nonce-based request-response authenticating handshake protocol . . . . .	165
5.3.4	Two-Flows Request-Response Authenticating Handshake, assuming Synchronized State . . . . .	166
5.4	Key-Setup Handshake . . . . .	167
5.4.1	Key-Setup Handshake: Signals and Requirements . . . . .	168
5.4.2	Key-setup 2PP extension . . . . .	169
5.4.3	Key-Setup: Deriving Per-Goal Keys . . . . .	169
5.5	Key Distribution Protocols and GSM . . . . .	170
5.5.1	Case study: the GSM Key Distribution Protocol . . . . .	172
5.5.2	Replay attacks on GSM . . . . .	174
5.5.3	Cipher-agility and Downgrade Attacks . . . . .	176

5.6	Resiliency to key exposure: forward secrecy, recover secrecy and beyond . . . . .	180
5.6.1	Forward secrecy handshake . . . . .	180
5.6.2	Recover-Security Handshake Protocol . . . . .	182
5.6.3	Stronger notions of resiliency to key exposure . . . . .	183
5.6.4	Per-goal Keys Separation. . . . .	184
5.6.5	Resiliency to exposures: summary . . . . .	185
5.7	Shared-Key Session Protocols: Additional Exercises . . . . .	188
<b>6</b>	<b>Public Key Cryptology</b>	<b>193</b>
6.1	Introduction to PKC . . . . .	193
6.1.1	Public key cryptosystems . . . . .	194
6.1.2	Digital signature schemes . . . . .	195
6.1.3	Key exchange protocols . . . . .	195
6.1.4	Advantages of Public Key Cryptology (PKC) . . . . .	196
6.1.5	The price of PKC: assumptions, computation costs and key-length . . . . .	197
6.1.6	Hybrid Encryption . . . . .	201
6.1.7	The Factoring and Discrete Logarithm Hard Problems .	202
6.2	The DH Key Exchange Protocol . . . . .	205
6.2.1	Physical key exchange . . . . .	205
6.2.2	Key exchange protocol: prototypes . . . . .	207
6.2.3	The Diffie-Hellman Key Exchange Protocol and Hardness Assumptions . . . . .	210
6.3	Key Derivation Functions (KDF) and the Extract-then-Expand paradigm . . . . .	212
6.4	Using DH for Resiliency to Exposures: PFS and PRS . . . . .	214
6.4.1	Authenticated DH: Perfect Forward Secrecy (PFS) . . .	215
6.4.2	The Synchronous-DH-Ratchet protocol: Perfect Forward Secrecy (PFS) and Perfect Recover Security (PRS) . . .	216
6.4.3	The Asynchronous-DH-Ratchet protocol . . . . .	217
6.4.4	The Double Ratchet Key-Exchange protocol . . . . .	218
6.5	Security for Public Key Cryptosystems . . . . .	219
6.6	Discrete-Log based public key cryptosystems: DH and El-Gamal	220
6.6.1	The DH PKC . . . . .	220
6.6.2	The El-Gamal PKC . . . . .	221
6.6.3	Homomorphic encryption and re-encryption. . . . .	221
6.7	The RSA Public-Key Cryptosystem . . . . .	223
6.7.1	RSA key generation. . . . .	223
6.7.2	Textbook RSA: encryption, decryption and correctness.	223
6.7.3	The RSA assumption and security . . . . .	225
6.7.4	RSA with OAEP (Optimal Asymmetric Encryption Padding)	227
6.8	Public key signature schemes . . . . .	227
6.8.1	RSA-based signatures . . . . .	230
6.8.2	Discrete-Log based signatures . . . . .	232
6.9	Additional Exercises . . . . .	233

<b>7 The TLS/SSL protocols for web-security and beyond</b>	<b>241</b>
7.1 Introducing TLS/SSL . . . . .	242
7.1.1 TLS/SSL: High-level Overview . . . . .	242
7.1.2 TLS/SSL: security goals . . . . .	243
7.1.3 SSL/TLS: Engineering goals . . . . .	245
7.1.4 TLS/SSL and the TCP/IP Protocol Stack . . . . .	246
7.1.5 The SSL/TLS record protocol . . . . .	246
7.2 The beginning: the handshake protocol of SSLv2 . . . . .	248
7.2.1 SSLv2: the ‘basic’ handshake . . . . .	249
7.2.2 SSLv2: <i>ID</i> -based Session Resumption . . . . .	251
7.2.3 SSLv2: ciphersuite negotiation and downgrade attack .	252
7.2.4 Client authentication in SSLv2 . . . . .	253
7.3 The Handshake Protocol: from SSLv3 to TLSv1.2 . . . . .	253
7.3.1 SSLv3 to TLSv1.2: improved derivation of keys . . . . .	255
7.3.2 Crypto-agility, backwards compatibility and downgrade attacks . . . . .	257
7.3.3 Secure extensibility principle and TLS extensions . . . . .	261
7.3.4 SSLv3 to TLSv1.2: DH-based key exchange . . . . .	263
7.3.5 SSLv3 to TLSv1.2: session resumption . . . . .	265
7.3.6 Client authentication . . . . .	268
7.4 State-of-Art: TLS 1.3 . . . . .	269
7.5 TLS/SSL: Additional Exercises . . . . .	271
<b>8 Public Key Infrastructure (PKI)</b>	<b>276</b>
8.1 Public Key Certificates and Infrastructure . . . . .	276
8.2 Basic PKI Concepts from the X.509 Standard . . . . .	278
8.2.1 The X.500 Global Directory Standard . . . . .	279
8.2.2 The X.500 Distinguished Name . . . . .	279
8.2.3 X.509 Public Key Certificates . . . . .	283
8.2.4 The X.509 Extensions Mechanism . . . . .	286
8.3 Certificate Validation and Standard Extensions . . . . .	288
8.3.1 Trust-Anchor-signed Certificate Validation . . . . .	289
8.3.2 Standard Alternative-Naming Extensions . . . . .	290
8.3.3 Standard key-usage and policy extensions . . . . .	291
8.3.4 Certificate path validation . . . . .	292
8.3.5 The certificate path constraints extensions . . . . .	293
8.3.6 The basic constraints extension . . . . .	296
8.3.7 The naming constraints extension . . . . .	297
8.3.8 The policy constraints extension . . . . .	298
8.4 Certificate Revocation . . . . .	299
8.4.1 Certificate Revocation List (CRL) . . . . .	300
8.4.2 Online Certificate Status Protocol (OCSP) . . . . .	301
8.4.3 Optimized variants of OCSP . . . . .	306
8.4.4 OCSP Stapling: query by subject (website) . . . . .	310
8.5 X.509/PKIX Web-PKI CA Failures and Defenses . . . . .	310
8.5.1 Weaknesses of X.509/PKIX Web-PKI . . . . .	311

8.5.2	Defenses against Corrupt/Negligent CAs . . . . .	312
8.6	Certificate Transparency (CT): Detecting CA failures . . . . .	314
8.6.1	Issuing CT certificates and the Signed Certificate Timestamp (SCT) . . . . .	316
8.6.2	Monitoring: simplified by assuming Trustworthy Loggers	316
8.6.3	Auditing to detect rogue loggers . . . . .	317
8.6.4	Monitoring in CT, using Merkle tree of certificates . . .	317
8.7	PKI: Additional Exercises . . . . .	317
<b>9</b>	<b>Usable Security and User Authentication</b>	<b>324</b>
9.1	Password-based Login . . . . .	324
9.1.1	Hashed password file . . . . .	324
9.1.2	One-time Passwords with Hash-Chain . . . . .	324
9.2	Phishing . . . . .	324
9.3	Usable Defenses against Phishing . . . . .	324
9.4	Usable End-to-End Security . . . . .	324
9.5	Usable Security and Authentication: Additional Exercises . . .	324
<b>10</b>	<b>Review exercises and solutions</b>	<b>327</b>
10.1	Review exercises . . . . .	327
10.2	Solutions to selected exercises . . . . .	328
<b>Index</b>		<b>333</b>
<b>Bibliography</b>		<b>338</b>

## Acknowledgments

While this is still a work-in-progress, I realize that I should better begin acknowledging the people who help me, to minimize the unavoidable errors of omission where I forget to thank and acknowledge help from friends and colleagues. So I am beginning now to write down these acknowledgments; please let me know if you notice any omissions, and accept my apologies in advance.

I received a lot of help from my teaching assistants (TAs) in the University of Connecticut (UConn): Justin Furuness (2018) and Sam Markelon (2019). I am also grateful for my TAs for the similar course I gave earlier in Bar Ilan University: Hemi Leibowitz, Haya Shulman and Yigal Yoffe.

I am also grateful to the many students, mainly in the University of Connecticut, who have been tolerant of my innumerable mistakes and tackled this hard-to-read text, helping me to improve it with invaluable feedback and suggestions. Similar thanks to peer professors who used the text and offered comments, suggestions or even text, exercises and figures; this is all highly appreciated. In particular, thanks to Shaanan Cohney for sending the LaTeX source code for Figure 2.29, taken from [32].

# Chapter 1

## Introduction

*Cybersecurity* refers to security aspects of systems involving communication and computation mechanisms. Cybersecurity is a wide area, which involves the security of many different applications and types of infrastructure.

Security is a tricky area, where intuition may often mislead as attacks often exploit subtle vulnerabilities, which our intuition fails to consider. This makes this area challenging, and interesting; in particular, in this area, *intuition is a dangerous guide*, and careful, adversarial thinking is crucial. Indeed, for such an applied, systems area, *precise definitions and proofs* are surprisingly important. In particular, in many areas of engineering, designs are evaluated under *typical, expected scenarios and failures*. When this approach is adopted to evaluate security solutions, the designers often evaluate the system under what they consider as *expected adversarial attacks*. However, this is a mistake: security systems should be evaluated against *arbitrary adversarial strategies*, as much as possible. Of course, this does not mean that we should assume an omnipotent adversary, against whom every reasonable attack would fail; but our defenses should be designed assuming only limitations on the *capabilities* of the adversary, not on the adversarial *strategy*.

### 1.1 About this textbook

The goal of this textbook is to provide *solid foundations* to the area of applied, technical Cybersecurity, accessible to students learning this topic for the first time, and possibly also helpful for experts in the area. The textbook can be used either as a companion to a course with lectures, or for self-study; in both cases, a set of presentations is available and may be useful for lecturers and students. The textbook also contains a significant number of exercises and examples, although adding even more is clearly desirable.

Like most technical textbooks, we cover and explain different techniques and topics. Beyond these, however, we will also try to help readers develop the *adversarial thinking* that allows the cybersecurity expert to avoid and identify vulnerabilities - even subtle ones - in practical systems. To achieve this, we

combine discussion of informal design principles and of practical systems and their vulnerabilities, with presentation of key theoretical tools and definitions.

There is a wide variety of different tools and techniques used to ensure cybersecurity, and obviously, we will not be able to cover all of them. The current plan is to focus on two volume and central topics. In this, first, volume, we focus on *applied cryptography*. In the second volume, we focus on *Internet and Web Security*. Some of the other important topics include *software security*, including malware, secure coding, and more; *privacy and anonymity*, *operating systems and computer security*, *security of cyber-physical systems* and *Internet-of-Things (IoT)*, *usable security and social engineering*, *machine learning security*, *information flow analysis*, and more.

Why is the first volume introducing applied cryptography? One reason is that cryptography is important and applicable to most areas of cybersecurity; furthermore, cryptography is an ancient, mature yet still exciting and ‘hot’ area, many of whose principles and approaches apply also to other areas of cybersecurity - and to develop the ‘adversarial thinking’ which is so important for the cyber-security practitioner.

Focusing on cryptography also allows this part to be mostly self-contained, without minimal reliance on prior knowledge in discrete mathematics, which is usually studied in a course in the first (freshman) year of study of computer science and related areas. In particular, no prior knowledge is assumed in cryptography, networking, operating-systems, system programming, theory of computer science or advanced math. This allows learning the contents of this volume, typically in a course, before learning other courses and topics in cybersecurity and cryptography.

This part serves as textbook for the introductory course on cybersecurity, CSE3400, in the University of Connecticut (UConn).

### **Contents of this volume.**

Cybersecurity and cryptography are vast, fascinating fields. This part was carefully chosen, to be self-contained and limited to reasonable scope, yet provide sufficient background in cryptography for cybersecurity practitioners.

We cover important applied and basic cryptographic schemes, including shared and public key cryptosystems, digital signatures, pseudo-randomness, message authentication codes, cryptographic hash functions, authentication and key distribution protocols, and more. We also discuss several applications of cryptography for other areas of cybersecurity, including public key infrastructure, web and transport-layer security, privacy and anonymity, and user authentication, including secure usability and social engineering.

### **Learning more cybersecurity and cryptography**

Other aspects of cyber-security are covered by other UConn courses - and many excellent books and online sources.

In particular, *network security* would be the topic of the second volume of this manuscript, and covered in the follow-up UConn course CSE 4502; a draft version of the lectures (presentations) is available from the author, and would be posted upon revision (or if requested).

For a different approach to applied aspects of cryptography, and for coverage of many topics in network-security, we recommend the book by Perlman, Kaufman and Speciner [91].

More in depth coverage of cryptography is provided by UConn courses 4702, 5852 and 5854. System-security is covered by UConn course 4400.

These notes gives a very minimal glimpse of the beautiful and extensive *theory of cryptography* - only very few, basic definitions and proofs. We included these, since we believe this helps to understand the important goal of *security against arbitrary attacker strategies* - a goal which is central to the area of cybersecurity. Readers and students are very encouraged to take a course in cryptography and to read relevant textbooks. In particular, some good options include:

- One of the good books providing easy-to-read introduction to the basic issues of cryptography is [106].
- A more updated and more in-depth coverage is provided in [2].
- An excellent, in depth coverage of the fascinating *theory of cryptography* is in [57, 58].
- Several good books, e.g., [82], cover more advanced cryptographic tools, based on mathematics, such as Elliptic curves, key distribution and pre-distribution schemes, authentication codes and more.

In this introduction, we begin with a brief history of computers, cryptology and cyber, including some of the key terms such as ‘cybersecurity’. We then introduce one of the basic principles of cybersecurity, the *attack model and security requirements principle*. Finally, we present a table of the notations used in this manuscript.

*Online resources - lecture notes and lectures (presentations) - and request for feedback and contributions.* These notes has a corresponding set of power-point presentations. These presentations, like the manuscript itself, are available for download from the ResearchGate website. If some presentation is not available or appears outdated, or if these notes appear outdated, please alert me by email or by a ResearchGate message - I may have forgotten to update. I would also appreciate feedback, esp. corrections and other suggestions for improvements and places where clarifications are needed; don’t hesitate, if you didn’t understand, it is my fault, not yours, tell me and I will improve. Finally, I welcome also contributions, esp. additional exercises and/or solutions. Thanks!

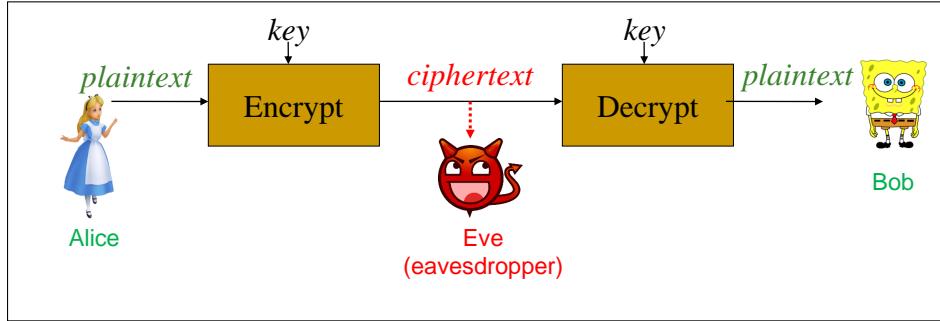


Figure 1.1: Encryption: terms and typical use. Alice needs to send sensitive information (*plaintext*) to Bob, so that the information will reach Bob - but remain confidential from Eve, who can *eavesdrop* on the communication between Alice and Bob. To do this, Alice *encrypts* the plaintext; the encrypted form is called *ciphertext*, and Eve cannot learn anything from it (except its size). However, Bob can *decrypt* the ciphertext, which recovers the plaintext.

## 1.2 Brief History of Computers, Cryptology and Cyber

Computers have quickly become a basic component of life, with huge impact on society and economy. However, their beginning was slow and modest. Babbage's concept of a computer was in 1833, but the first working computer was Konrad Zuse's Z1, completed in 1938 - full 105 years later. Furthermore, Z1 was an unreliable, slow mechanical computer, and did not have any useful applications or use, except as proof of feasibility. In particular, special-purpose calculating devices were way better in performing applied calculations.

Really useful computers appeared only during World War II - as a result of massive investment by the armies of both sides. The British effort was by far more important and more successful, since they identified a very important military application - for which programmable computers had large advantage over special-purpose devices: *cryptanalysis*.

Cryptology, which literally means the 'science of secrets', has been applied to protect sensitive communication since ancient times, and is therefore much more ancient than computers. Originally, cryptology focused on protecting *confidentiality*, is provided by *encryption*; see Figure 1.1. Encryption transforms sensitive information, referred to as *plaintext*, into a form called *ciphertext*, which allows the intended recipients to *decrypt* it back into the plaintext; the ciphertext should not expose any information to an attacker. The focus on encryption and confidentiality is evident in the term *cryptography*, i.e., 'secret writing', which is often used as a synonym for cryptology.

Cryptology, and in particular cryptanalysis, i.e., 'breaking' the security of cryptographic schemes, played a key role throughout history. Possibly most notably - and most well known - is the role cryptanalysis played during the second world war (WW II). Both sides used multiple types of encryption de-

vices, where the most well known are the Enigma and Lorenz devices used by the Germans; and both sides invested in cryptanalysis - with much greater successes to the allies, luckily.

The Enigma machine was used throughout the war, and early machines, captured and smuggled from Poland to Germany, were used to construct complex, special-purpose devices called *Bombe*, that helped cryptanalysis of Enigma traffic.

Lorenz encryption devices were introduced later during the war, and no complete device was available to cryptanalysts. This motivated the design of Colossus, the first fully functional and useful computer, by Tommy Flowers. Since Colossus was programmable, it was possible to test many possible attacks and to successfully cryptanalyze (different versions of) Lorenz, and some other cryptosystems.

One critical difference between the Colossus and more modern computers, is that *the Colossus did not read a program from storage*. Instead, setting up the program for the Colossus involved manual setting of switches and jack-panel connections. This is much less convenient, of course; but it was acceptable for the Colossus, since there were only few such machines and few, simple programs, and the simplicity of design and manufacture was more important than making it easier to change programs. Even this crude form of ‘programming’ was incomparably easier than changing the basic functionality of the machine, as required in special-purpose devices - including the Bombe used to decipher the Enigma.

The *idea* of a stored program was proposed already in 1936/1937 - by two independent efforts. Konrad Zuse briefly mentioned such design in a patent on floating-point calculations published at 1936 [116], and Alan Turing defined and studied a formal model for stored-program computers - now referred to as the *Turing machine model* - as part of his seminal paper, ‘On Computable Numbers’ [107]. However, practical implementations of stored-program computers appeared only *after* the WW II. Stored-program computers were much easier to use, and allowed larger and more sophisticated programs as well as the use of the same hardware for multiple purposes (and programs). Hence, stored-program computers quickly became the norm - to the extent some people argue that earlier devices were not ‘real computers’.

Stored-program computers also created a vast market for programs. It now became feasible for programs to be created in one location, and then shipped to and installed in a remote computer. For many years, this was done by physically shipping the programs, stored in media such as tapes, discs and others. Once computer networks became available, program distribution is often, in fact usually, done by sending the program over the network.

Easier distribution of software, meant also that the same program could be used by many computers; indeed, today we have programs that runs on billions of computers. The ability of a program to run on many computers, created an incentive to develop more programs; and the availability of a growing number of programs, increased the demand for computers - and the impact on society and economy.

The impact of computers further dramatically increased, when computer networks facilitated inter-computer communication. The introduction of personal computers (1977-1982), and later of the Internet, web, smartphones and IoT devices, caused further dramatic increase in the impact of computers and ‘smart devices’ - which are also computers, in the sense of running programs.

This growing importance of cyberspace also resulted in growing interest in the potential social implications, and a growing number of science-fiction works focused on these aspects. One of these was the novel ‘Burning Chrome’, published by William Gibson at 1982. This novel introduced the term *cyberspace*, to refer to these interconnected networks, computers, devices and humans. This term is now widely used, often focusing on the impact on people and society. The *cyber* part of the term is taken from *Cybernetics*, a term introduced in [114] for the study communication and control systems in machines, humans and animals.

There was also increased awareness of the risks of abuse and attacks on different components of cyberspace, mainly computers, software, networks, and the data and information carried and stored in them. This awareness significantly increased as attacks on computer systems became wide-spread, esp. attacks exploiting software vulnerabilities, and/or involving malicious software, i.e., *malware*. This awareness resulted in the study, research and development of threats and corresponding security mechanisms: *computer security*, *software security*, *network security* and *data/information security*.

The awareness of security risks also resulted in important works of fiction. One of these was the 1983 novel *cyberpunk*, by Bruce Bethke. Bethke coined this term for individuals which are socially-inept yet technologically-savvy.

Originally a derogatory term, cyberpunk was later adopted as a name of a movement; e.g., with several manifestos, e.g., [73]. In these manifestos, as well as in works of fiction, cyberpunks and hackers are still mostly presented as socially-inept yet technology-savvy, indeed, they are often presented as possessing incredible abilities to penetrate systems. However, all this is often presented in positive, even glamorous ways, e.g., as saving human society from oppressive, corrupt regimes, governments and agencies.

Indeed, much of the success of the Internet is due to its decentralized nature, and many of the important privacy tools, such as the Tor anonymous communication system [42], are inherently decentralized, which may be hoped to defend against potentially malicious government. Furthermore, some of these tools, such as the PGP encryption suite [54], were developed in spite of significant resistance by much of the establishment.

### 1.2.1 From classical to modern Cryptology

Cryptology has been studied and used for ages; we discuss some really ancient schemes in § 2.1. In this section, we discuss few basic facts about the long and fascinating history of cryptology; much more details are provided in several excellent books, including [69, 81, 104].

Literally, *cryptology* means ‘the study of secrets’. The word ‘secret’ can be interpreted here in two ways: *how to protect secrets* and *how to use secrets*. The oldest branch of cryptology is *cryptography*<sup>1</sup> or *encryption*, whose goal is *confidentiality*, i.e., to *hide the meaning of sensitive information* from attackers, *when the information is visible to the attacker*. Encryption is the topic of the following chapter; in particular, we present there the *Kerckhoff’s principle* [71], which states that the security of encryption should always depend on the secrecy of a *key*, and not on the secrecy of the encryption method. Kerckhoff’s principle had fundamental impact on cryptology, and is now universally accepted. One implication of it is that *security should not rely on obscurity*, i.e., on making the security or cryptographic mechanism itself secret. Note that applies not just to cryptology, but also to other aspects of cybersecurity.

Kerckhoff publication [71], at 1883, marks the beginning of a revolution in cryptology. Until then, the design of cryptosystems was kept secret; Kerckhoff realized that it is better to assume - realistically, too - that the attacker may be able to capture encryption devices and reverse-engineer them to learn the algorithm. This holds much more for modern cryptosystems, often implemented in software, or in chips accessible to potential attackers. It is also important that Kerckhoff *published* his work; there were very few previous published works in cryptology, again due to the belief that cryptology is better served by secrecy.

However, even after Kerckhoff’s publication, for many years, there was limited published research in cryptology, or even development, except by intelligence and defense organizations - in secret. The design and cryptanalysis of the Enigma, discussed above, are a good example; although the cryptanalysis work involved prominent researchers like Turing, and had significant impact on development of computers, it was kept classified for many years.

This changed quite dramatically in the 1970s, with the beginning of what we now call *modern cryptology*, which involves extensive academic research, publication, products and standards, and has many important applications.

Two important landmarks seem to mark the beginning of modern cryptology. The first was the development and publication of the *Data Encryption Standard (DES)* [105]. The second is the publication of the seminal paper *new directions in cryptography* [40], by Diffie and Hellman. This paper introduced the radical, innovative concept of *Public Key Cryptology (PKC)*, where the key to encrypt messages may be *public*, and only the corresponding decryption key must be kept private, allowing easy distribution of encryption keys. The paper also presented the *Diffie-Hellman Key Exchange* protocol; we discuss both of these in chapter 6.

In [40], Diffie and Hellman did not yet present an implementation of public key cryptography. The first publication of a public-key cryptosystem was RSA by Rivest, Shamir and Adelment in [96], and this is still one of the most popular public-key cryptosystems. In fact, the same design was discovered already few years earlier, by the GCHQ British intelligence organization. The CGHQ kept

---

<sup>1</sup> *Cryptography* is often used as a synonym to cryptology, although the literal meaning of cryptography is ‘secret writing’, which is arguably related mostly to encryption.

this achievement secret until 1997, long after the corresponding designs were re-discovered independently and published in [96]. See [81] for more details about the fascinating history of the discovery - and re-discovery - of public key cryptology.

This repeated discovery of public-key cryptology illustrates the dramatic change between ‘classical’ cryptology, studied only in secrecy and with impact mostly limited to the intelligence and defense areas, and modern cryptology, with extensive published research and extensive impact on society and economy.

### 1.2.2 Cryptology is not just about secrecy!

Our discussion so far focused on the use of cryptology to ensure secrecy of information, typically by encryption. However, modern cryptology is not limited to encryption and the goal of confidentiality; it covers other threats and goals related to information and communication. This includes goals such as *authenticity* and *integrity*, which deal with detection of tampering of information by the attacker (integrity), or with detection of impersonation by the attacker (authentication).

In particular, one of the most important mechanisms of modern cryptography, crucial to many of its applications and much of its impact, is the design of *digital signature* schemes. We discuss signature schemes in chapter 6; let us provide here a grossly simplified discussion.

Consider, first, ‘classical’, handwritten signatures. Everyone should be able to validate a signed document by comparing the signature on it to a sample signature, known to be of that signer; and yet, nobody but the signer should be able to sign a document using her signature, even if the other person has seen the sample signature, as well as signatures over other (even similar) documents.

Intuitively, digital signatures provide similar functionality to handwritten signatures; but, of course, the documents and the signatures are strings (files) rather than ink on paper, and the processes of signing and validating are done by applying appropriate functions (or algorithms), as illustrated in Figure 1.2. Specifically, the *signing* algorithm  $Sign$  uses a private signing key  $s$  known only to the signer, and the *validating* algorithm  $Ver$  uses a corresponding public validation key  $v$ . If  $\sigma = S_s(m)$ , i.e.,  $\sigma$  is a signature of message  $m$  using signing key  $s$ , then  $V_v(m, \sigma) = Ok$ ; and for an attacker who does not know  $s$ , it should be infeasible to find any unsigned message  $m'$  and any ‘signature’  $\sigma'$  s.t.  $V_v(m', \sigma') = Ok$ . The use of digital signatures allows computer-based signing and validation of digital documents, which is crucial to secure communication and transactions over the Internet.

We discuss digital signatures extensively, in § 6.8, and their application to the SSL/TLS protocol in chapter 7, and to Public Key Infrastructure (PKI) in chapter 8.

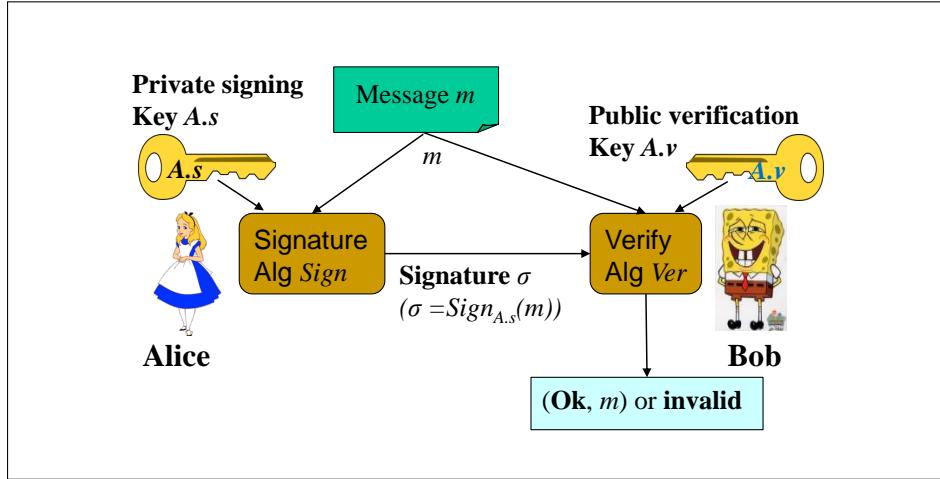


Figure 1.2: Digital signature and verification algorithms: terms and typical use. Alice *signs* message  $m$ , by applying the signing algorithm  $\text{Sign}$  using her private signing key  $A.s$ , resulting in the *signature*  $\sigma = \text{Sign}_{A.s}(m)$ . Alice sends the signature  $\sigma$  and the message  $m$  to Bob, who *verifies* the signature by applying the verification algorithm  $\text{Ver}$  using Alice's public verification key  $A.v$ , namely, computes  $\text{Ver}_{A.v}(m, \sigma)$ . If the result is *Invalid*, this implies that  $\sigma$  is not a valid signature of  $m$ ; if the result is *Ok*, then  $\sigma$  is valid signature, i.e.,  $m$  was indeed signed by Alice.

### 1.3 Cybersecurity Goals and Attack Models

In a broad sense, secure systems should protect ‘good’ parties using the systems legitimately, from damages due to *attackers* (also known as *adversaries*). Attacker could be ‘bad’ users or ‘outsiders’ with some access to the systems or their communication. Note that attackers may often also control a (corrupted) device which they don’t ‘own’.

There are two basic ways of protecting against attackers, *prevention* and *deterrence*:

**Prevention** is a proactive approach: we design and implement the system so that the attacker cannot cause damage (or can cause reduced damage). *Encryption* is an example of a cryptographic means to prevent attacks, as it is usually used to *prevent* an attacker from disclosing sensitive information.

**Deterrence** is a reactive approach: we design mechanisms that will cause damages to the attacker, if she causes harm, or even when we detect an attempt to cause harm. Effective deterrence requires the ability to *detect* the attack, to *attribute* the attack to the attacker, and to *penalize* the attacker sufficiently. Furthermore, deterrence can only be effective against

a rational adversary; no penalty is guaranteed to suffice to deter an irrational adversary, e.g., a terrorist. The use of *digital signatures* is one important deterrence mechanism. Signatures are used to deter attacks in several ways; in particular, a signature verified using the attacker's well-known public key, over a given message, provides evidence that the attacker signed that message. Such evidence can be used to punish or penalize the attacker in different ways - an important deterrent. Signatures may also be provided by users, as in reviews - to deter bad service or product, to motivate the provision of good services and products, and to allow users to choose a good service/product based on evaluations by peer users.

Note that deterrence is only effective if the adversary is *rational*, and would refrain from attacking if her expected profit (from attack) would be less than the expected penalty.

An obvious challenge in designing and evaluating security, is that we must 'expect the unexpected'; attackers are bound to behave in unexpected ways. As a result, it is critical to properly define the system, and identify and analyze any *risks*. In practice, deployment of security mechanisms has costs, and risk analysis would consider these costs against the risks, taking into account probabilities and costs of different attacks and their potential damages; however, we do not consider these aspects, and only focus on ensuring specific security goals, against specific, expected kinds of attackers.

**Cybersecurity goals.** Cybersecurity is often associated with three high-level goals, i.e., ensuring *Confidentiality, Integrity/authenticity and Availability (CIA)*. Note that integrity/authenticity and availability are separate from confidentiality, and often do not involve encryption; however, they often involve other cryptographic mechanisms, such as digital signatures, as we discussed above (subsection 1.2.2). Furthermore, note that these three goals are very broad, as they apply to most cybersecurity systems; when we study the security of any given system, we should first define specific security goals for that particular system, which will usually elaborate on these three high-level goals.

One of the fundamentals of modern cryptology, which already appears in [40], is an attempt to understand and define a clear model of the attacker capabilities, and a clear goal/requirements for the scheme/system. We believe that not only in cryptology, but in general in security, the articulation of the *attack model* and of the *security requirements* is fundamental to design and analysis of security. Indeed, we consider this the *first principle of cybersecurity*. This principles applies also in areas of cybersecurity where it may not be feasible to have completely rigorous models and proofs. Yet, precise articulation of attacker model and capabilities is very important, and helps identify and avoid vulnerabilities.

A well-articulated description of the attacker model and capabilities, and of the security requirements and assumptions, is necessary to evaluate and ensure

Table 1.1: Notations used in this manuscript.

$S = \{a, b, c\}$	A set $S$ with three elements - $a, b$ and $c$ .
$\Pi_{x \in S} V_x$	Multiplication (like $\Sigma$ is addition), e.g., $\Pi_{x \in \{a, b, c\}} V_x = V_a \cdot V_b \cdot V_c$ .
$C \cup B$	union of sets $C$ and $B$
$\bar{x}$	The inverse of bit $x$ , or bit-wise inverse of binary string $x$ .
$\{a, b\}^l$	The set of strings of length $l$ over the alphabet $\{a, b\}$ .
$a^l$	String consisting of $l$ times $a$ ; e.g., $1^4 = 1111$ .
$\{a, b\}^*$	The set of strings of any length, over from the alphabet $\{a, b\}$ .
$\parallel$	Concatenation; $abc \parallel def = abcdef$ .
$a[i]$	The $i^{\text{th}}$ bit of string $a$
$a[i \dots j]$ or $a_{i \dots j}$	String containing $a[i] \parallel \dots \parallel a[j]$ .
$a^R$	The ‘reverse’ of string $a$ , e.g., $abcde^R = edcba$ .
$\oplus$	Bit-wise exclusive OR (XOR); $0111 \oplus 1010 = 1101$ .
$\otimes$	Another notation for XOR - should always be changed for $\oplus$ .
$x \xleftarrow{\$} X$	Select element $x$ from set $X$ with uniform distribution.
$\Pr_{x \xleftarrow{\$} X}(F(x))$	The probability of $F(x)$ to occur, when $x$ is selected uniformly from set $X$ .
$A^{B_k(\cdot)}$	Algorithm $A$ with <i>oracle access</i> to algorithm $B$ , instantiated with key $k$ . Namely, algorithm $A$ may ‘invoke’ $B_k$ on input $x$ , and receive $B_k(x)$ , without algorithm $A$ ‘knowing’ key $k$ .
PPT	The set of Probabilistic Polynomial Time algorithms. An algorithm $A$ is in PPT if there is some constant $c$ s.t. given any input $x$ , algorithm $A$ terminates in at most $P( x )$ steps.
$NEGL(n)$	The set of ‘negligible functions’: for any constant $c$ and sufficiently large $n$ holds: $f(n) < n^c$ . See Def. 2.3.

security, for *arbitrary* interactions with the adversary. The adversary is limited in its *capabilities*, not in its *strategy*.

**Principle 1** (Security Goals and Attack Model). *Design and evaluation of system security, should include a clear, well defined model of the attacker capabilities (attack model) and of the exact criteria for a system, function or algorithm to be considered secure vs. vulnerable (security requirements).*

## 1.4 Notations

Notations are essential for precise, efficient technical communication, but it can be frustrating to understand text which uses unfamiliar or forgotten notations. Our goal is to help the reader to follow the notations in this notes, Table 1.1 presents notations which we use extensively. Please refer to it whenever you see some unclear notation, and let me know of any missing notation.

## Chapter 2

# Encryption and Pseudo-Randomness

Encryption deals with protecting the confidentiality of sensitive information, which we refer to as *plaintext message*  $m$ , by encoding (encrypting) it into *ciphertext*  $c$ . The ciphertext  $c$  should hide the contents of  $m$  from the adversary, yet allow recovery of the original information by legitimate parties, using a decoding process called *decryption*. Encryption is one of the oldest applied sciences; some basic encryption techniques have been used thousands of years ago.

One result of the longevity of encryption is the use of different terms. The cryptographic encoding operation is referred to as either *encryption* or *encipherment*, and the decoding operation is referred to as *decryption* or *decipherment*. Encryption schemes are often referred to as *cryptosystems* or as *ciphers*; in particular we will discuss two specific types of cryptosystems referred to as *block ciphers*<sup>1</sup> and *stream ciphers*. We use the terms ‘encryption scheme’ and ‘cryptosystem’ interchangeably.

Fig. 2.1 shows the encryption of *plaintext message*  $m$  (top of figure) and decryption of *ciphertext*  $c$  (middle of figure), both using the same *shared key*  $k$ . The message is from some *message-space*  $M$ , and the key is from some *key-space*  $K$ ; both  $M$  and  $K$  are typically sets of binary strings, e.g., all strings or all strings of specific length. We define a shared-key cryptosystem in Def. 2.1.

**Definition 2.1** (Shared-key cryptosystem). *A shared-key cryptosystem is a pair of keyed algorithms,  $\langle E, D \rangle$ , ensuring correctness, i.e., for every message  $m \in M$  and key  $k \in K$  holds:  $\text{Decrypt}_{\text{key}}(\text{Encrypt}_{\text{key}}(m)) = m$ , as illustrated in the bottom of Fig. 2.1.*

Definition 2.1 is quite general. In particular, it does not restrict the encryption and decryption algorithms, e.g., they may be randomized; and it allows arbitrary message space  $M$  and key space  $K$ . In the rest of this chapter, we will

---

<sup>1</sup>A *block* is a term for a string of bits of fixed length, the *block length*.

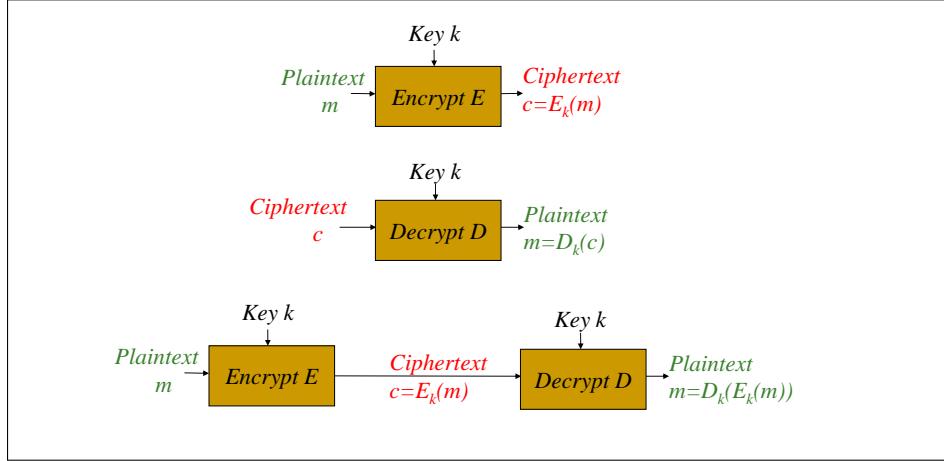


Figure 2.1: High-level shared-key encryption process. *Top*: encryption of plaintext message  $m$  using key  $k$ , resulting in ciphertext  $c = E_k(m)$ . *Middle*: decryption of ciphertext  $c$  using key  $k$ , resulting in plaintext message  $m = D_k(c)$ . *Bottom*: the correctness property, i.e., decryption of ciphertext  $c = E_k(m)$  returns back the original plaintext  $m = D_k(c) = D_k(E_k(m))$ .

see a variety of shared-key cryptosystems, some deterministic, some randomized, and with different message and key spaces. Some schemes even require a further generalization, namely, the use of *state*, e.g., a counter; we discuss stateful encryption schemes later in this chapter.

Shared-key cryptosystems are also referred to as *symmetric cryptosystems*, referring to their use of the same key  $k$  for encryption and decryption. This is in contrast to *public-key cryptosystems*, also referred to as *asymmetric cryptosystems*, which use separate keys for encryption (e.g., denoted  $e$ ) and for decryption ( $d$ ). The two keys are related, to allow the use of  $d$  to decrypt messages encrypted using  $e$ ; but it should be infeasible to derive the decryption key  $d$  given the encryption key  $e$ , hence, the encryption key can be made public (not secret). We illustrate a public-key cryptosystems in Fig.2.2, and discuss them in chapter 6.

Note also that Definition 2.1 focuses on *correctness*; it does not require the encryption scheme to ensure any *security* property. The reason for that is that defining ‘security’ is more complex than one may initially expect. Intuitively, there is a common goal: confidentiality, in a strong sense, against powerful adversaries; however, there are subtle issues, as well as multiple variants which differ in their exact requirements and assumptions about the adversary capabilities. Later on, we discuss the security requirements and present a definition.

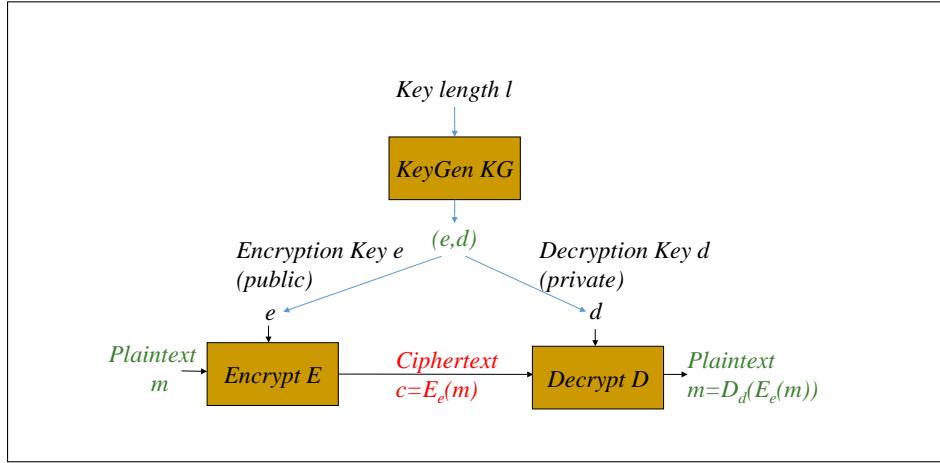


Figure 2.2: Public-Key Cryptosystem ( $KG, E, D$ ). The Key-Generation algorithm  $KG$  outputs a (public,private) keypair  $(e, d)$  of given length  $l$ . The encryption algorithm  $E$  uses the public encryption key  $e$  to compute the ciphertext  $c = E_e(m)$ , given plaintext message  $m$ . The decryption algorithm  $D$  uses the private decryption key  $d$  to compute the plaintext; correctness holds if for every pair of matching keys  $(e, d) = KG(l)$  and every message  $m$ , holds  $m = D_d(E_e(m))$ .

## 2.1 From Ancient Ciphers to Kerckhoffs' Principle

Cryptology is one of the most ancient sciences. We begin our discussion of encryption schemes by discussing few ancient ciphers, and some simple variants. An important property that one has to keep in mind is that the *design* of these ciphers was usually kept as a secret; even when using a published design, users typically kept their choice secret. Indeed, it *is* harder to cryptanalyze a scheme which is not even known; see discussion in subsection 2.1.4, where we present the Kerckhoffs' principle, which essentially says that security of a cipher should not depend on the secrecy of its design.

Since the ancient ciphers were considered secret, some of the ancient designs did not use secret keys at all; we discuss such *keyless ciphers* in subsection 2.1.1. Besides the historical perspective, discussing these simple, ancient ciphers helps us introduce some of the basic ideas and challenges of cryptography and cryptanalysis. Readers interested in more knowledge about the fascinating history of cryptology should consult some of the excellent manuscripts such as [69,104].

The very ancient ciphers were *mono-alphabetic substitution ciphers*. Monoalphabetic substitution ciphers use a fixed mapping from each plaintext character, to a corresponding ciphertext character (or some other symbol). Namely, these ciphers are stateless and deterministic, and defined by a permutation from the plaintext alphabet to a set of ciphertext characters or symbols. We further discuss mono-alphabetic substitution ciphers in subsection 2.1.3.

### 2.1.1 Ancient Keyless Monoalphabetic Ciphers

In this section, we discuss several well-known, simple, weak and ancient ciphers. These ciphers also share two important additional properties: they do not utilize a secret key, i.e., are ‘keyless’; and they are *monoalphabetic*. A cipher is monoalphabetic if they are defined by a single, fixed mapping from plaintext letter to ciphertext letter or symbol.

**The At-Bash cipher** The At-Bash cipher may be the earliest cipher whose use is documented; specifically, it is believed to be used, three times, in the book of Jeremiah belonging to the old testament. The cipher maps each of the letters in the Hebrew alphabet, to a different letter. Specifically, the letters are mapped in ‘reverse order’: first letter to the last letter, second letter to the one-before-last letter, and so on. See illustrated in Fig. 2.3; while you may not be familiar with the letters of the Hebrew alphabet, the mapping can still be identified by the visual appearance. If not, that’s Ok; we next describe an adaptation of the At-Bash cipher to the Latin alphabet.

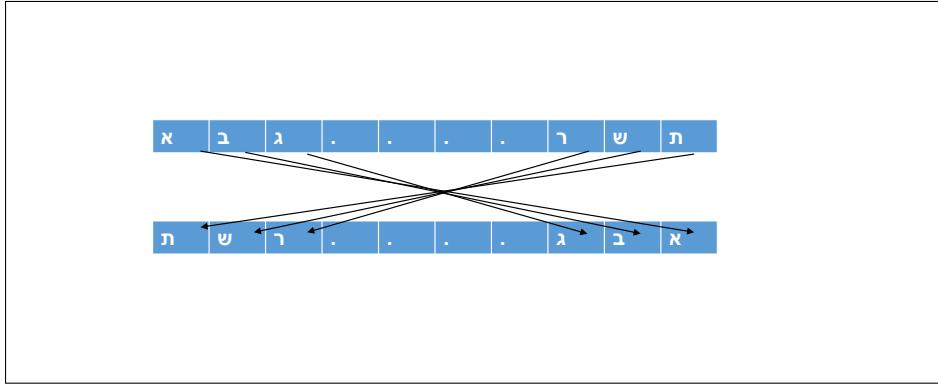


Figure 2.3: The At-BaSh Cipher.

**The Az-By cipher** The Az-By cipher is the same as the At-Bash cipher, except using the Latin alphabet. It is convenient to define the Az-By cipher, the At-Bash cipher, and similar ciphers, by a formula. For that purpose, we define the cipher as a function of the input letter, where each letter is represented by its distance from the beginning of the alphabet; i.e., we represent the letter ‘A’ by the number 0, the letter ‘B’ by 1, and so on, till letter ‘Z’, represented by 25. The Az-By cipher is now defined by the following encryption function:

$$E_{Az-By}(m) = 25 - m \quad (2.1)$$

We illustrate the Az-By cipher in the top part of Fig. 2.4; below it, we present two similar ancient ciphers, which we next discuss - the Caesar and the ROT13 ciphers.

AzBy														
A	B	C	D	E	F	G	H	I	J	K	L	M		
Z	Y	X	W	V	U	T	S	R	Q	P	O	N		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
M	L	K	J	I	H	G	F	E	D	C	B	A		

Caesar														
A	B	C	D	E	F	G	H	I	J	K	L	M		
D	E	F	G	H	I	J	K	L	M	N	O	P		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
Q	R	S	T	U	V	W	X	Y	Z	A	B	C		

ROT13														
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		
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	M		

Figure 2.4: The AzBy, Caesar and ROT13 Ciphers.

**The Caesar cipher.** We next present the well-known Caesar cipher. The Caesar cipher has been used, as the name implies, by Julius Caesar. It is also a mono-alphabetic cipher, operating on the set of the 26 Latin letters, from A to Z. In Caesar cipher, the encryption of plaintext letter  $p$  is found by simply ‘shifting’ to the fourth letter after it, when letters are organized in a circle (with A following Z). The Caesar cipher is illustrated by the middle row in Figure 2.4.

We next write the formula for the Caesar cipher. As above, we represent each letter by its distance from the beginning of the alphabet; i.e., we represent the letter ‘A’ by the number 0, and so on; ‘Z’ is represented by 25. This can be conveniently written as a formular, using modular arithmetic notations; see a brief recap in Note 2.1. Namely, the Caesar encryption of  $p$  is given by:

$$E_{\text{Caesar}}(m) = m + 3 \pmod{26} \quad (2.2)$$

**Exercise 2.1.** Write the formulas for the decryption of the Caesar and Az-By ciphers.

**Exercise 2.2.** Use Equations (2.4) and (2.6) to prove that a number is divided by 3 if and only if the sum of its decimal digits is divided by 3; prove the similar rule for division by 9.

### Note 2.1: Basic modular arithmetic

Modular arithmetic is based on the *modulo* operation, denoted  $\mod$ , and defined as the *residue* in integer division between its two operands. Namely,  $a \mod n$  is an integer  $b$  s.t.  $0 \leq b < n$  and for some integer  $i$  holds  $a = b + i \cdot n$ . Note that  $a$  and  $i$  may be negative, but the value of  $a \mod n$  is unique.

The  $\mod$  operation is applied after all ‘regular’ arithmetic operations such as addition and multiplication; i.e.,  $(a + b \mod n) = [(a + b) \mod n]$ . On the other hand, we can apply the  $\mod$  operation also to the operands of addition and multiplication, often simplifying the computation of the final result. Namely, the following useful rules are easy to derive, for any integers  $a, b, c, d, n, r$  s.t.  $0 \leq r < n$ :

$$r \mod n = r \quad (2.3)$$

$$[(a \mod n) + (b \mod n)] \mod n = (a + b) \mod n \quad (2.4)$$

$$[(a \mod n) - (b \mod n)] \mod n = (a - b) \mod n \quad (2.5)$$

$$[(a \mod n) \cdot (b \mod n)] \mod n = a \cdot b \mod n \quad (2.6)$$

$$a^b \mod n = (a \mod n)^b \mod n \quad (2.7)$$

$$[(a + c) \mod n = (b + c) \mod n] \Leftrightarrow [a \mod n = b \mod n] \quad (2.8)$$

$$[a \cdot r \mod n = b \cdot r \mod n] \Leftrightarrow [a \mod n = b \mod n] \quad (2.9)$$

For simplicity, we often write the  $\mod n$  operation only at the right hand side of an equation, but it still applies to both sides. To avoid confusion, in such situations, it is preferable to use the congruence symbol  $\equiv$ , instead of the ‘regular’ equality symbol  $=$ , e.g.,  $a + b \equiv c \cdot d \mod n$ .

We apply - and discuss - more advanced modular arithmetic, in chapter 6, where we discuss public key cryptography.

**Exercise 2.3.** Use Eq. (2.7) to show that for any integers  $a, b$  holds  $a^b \mod (a - 1) = 1$ .

**The ROT13 cipher** ROT13 is popular variant of the Caesar cipher, with the minor difference that the ‘shift’ is of 13 rather than of 3, i.e.,  $E_{ROT13}(p) = p + 13 \mod 26$ . Note that in this special case, encryption and decryption are exactly the same operation:  $E_{ROT13}(E_{ROT13}(p)) = p$ . The ROT13 cipher is illustrated by the bottom row in Figure 2.4. Instead of writing down the formula for ROT13 encryption, as done above for Az-By and Caesar ciphers, we encourage the readers themselves to write it down.

**The Masonic cipher.** A final example of a historic, keyless, monoalphabetic cipher, is the Masonic cipher. The Masonic cipher is from the 18<sup>th</sup> century, and is illustrated in Fig. 2.5. This cipher used a ‘key’ to map from plaintext to ciphertext and back, but the key is only meant to assist in the mapping, since it has regular structure and considered part of cipher.

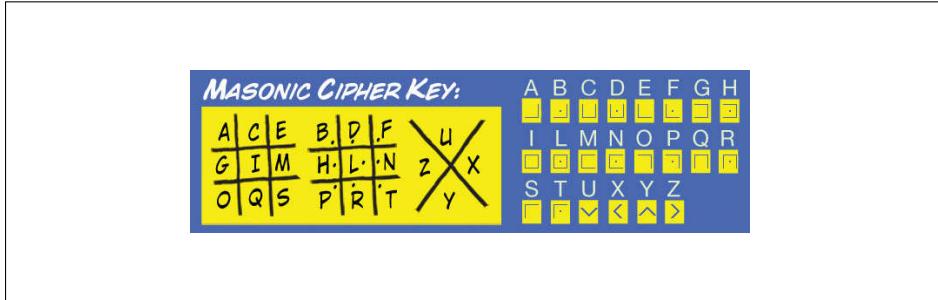


Figure 2.5: The Masonic Cipher, written graphically and as a mapping from the Latin alphabet to graphic shapes.

### 2.1.2 Shift Cipher: a Keyed Variant of the Caesar Cipher

Keyless ciphers have limited utility; in particular, the design of the cipher becomes a critical secret, whose exposure completely breaks security. Therefore, every modern cipher, and even most historical ciphers, use secret keys.

Readers who are interested in these historical (yet keyed) ciphers should consult manuscripts on the history of cryptology, e.g. [69, 104]. We merely focus on the *shift cipher*, which is a simple keyed variant of the Caesar cipher,  $E(p) = p+3 \bmod 26$ . We find the shift cipher helpful in introducing important concepts and principles, later in this chapter.

The first variant of the shift cipher encrypts a single Latin character at a time, just like the Caesar cipher; it simply uses an arbitrary shift rather than the fixed shift of three as in the original Caesar cipher. This variable shift can be considered as a key and denoted  $k$ . Namely, this shift cipher is defined by  $E_k(p) = p + k \bmod 26$ .

Obviously, this variant of the shift cipher is about as insecure as the original Caesar cipher; the attacker only needs to exhaustively search the 26 possible key values. To prevent this simple attack, one can further extend Caesar to use longer blocks of plaintext and keys, allowing the use of many shift values rather than just one of 26 possible shifts.

For convenience, assume that the input is now encoded as a binary string. The  $l$ -bits *shift cipher* operates on plaintexts of  $l$  bits and also uses  $l$  bit key. To encrypt plaintext block  $p \in \{0,1\}^l$ , using key  $k \in \{0,1\}^l$ , we compute  $E_k(p) = p + k \bmod 2^l$ .

By using sufficiently large  $l$  (length of keys and blocks), exhaustive search become impractical. However, this cipher is still insecure. For example, as the following exercise shows, it is easily broken by an attacker who has access to one *known plaintext pair*, i.e., a pair of (plaintext  $m$ , ciphertext  $E_k(m)$ ). An attack exploiting such pairs is called a *known plaintext attack (KPA)*.

**Exercise 2.4** (Known plaintext attack (KPA) on shift cipher). *Consider the  $l$ -bits shift cipher  $E_k(p) = p + k \bmod 2^l$ , where  $p, k \in \{0,1\}^l$ . Suppose attacker*

can get encryption  $E_k(p)$  of one known block  $p \in \{0, 1\}^l$ . Show that this allows the attacker to decrypt an arbitrary ciphertext  $c'$ , i.e., find  $p'$  s.t.  $c' = E_k(p')$ .

### 2.1.3 Mono-alphabetic Substitution Ciphers

The Mason, At-Bash, Caesar and shift ciphers are all *mono-alphabetic substitution ciphers*. Monoalphabetic substitution ciphers are deterministic, state-less mappings from plaintext characters to ciphertext characters or symbols; indeed, the use of any other set of symbols instead of letters does not make any real change in the security of such ciphers, hence we will assume permutation from characters into characters. For the 26 letters Latin alphabet, there are  $26! > 2^{88}$  such permutations, clearly ruling out exhaustive search for the key.

Of course, specific mono-alphabetic substitution ciphers may use only a small subset of these permutations. This is surely the case for all of the naive ciphers we discussed above - the Mason, At-Bash, Caesar and shift ciphers.

However, what about using a general permutation over the alphabet - where the permutation itself becomes a key? We refer to this cipher as the *general mono-alphabetic substitution cipher*. The key for this cipher may be written as a simple two-rows table, with one row containing the plaintext letters and the other row containing the corresponding ciphertext letter (or symbol). For example, see such a code in Figure 2.6, where the plaintext alphabet is the Latin alphabet plus four special characters (space, dot, comma and exclamation mark), for total of 30 characters. The reader may very well recall the use of similar ‘key tables’ from mono-alphabetic ciphers often used by kids.

A	B	C	D	E	F	G	H	I	J
Y	*	X	K	L	D	B	C	Z	F
K	L	M	N	O	P	Q	R	S	T
G	E	R	U	+	J	W	!	H	M
U	V	W	X	Y	Z	.	,	!	
P	N	I	O	Q	S	-	V	T	A

Figure 2.6: Example of a possible key for a general mono-alphabetic substitution cipher, for alphabet of 30 characters.

As we already concluded, exhaustive search for the key is impractical - too many keys. However, even when we select the permutation completely at random, this cipher is vulnerable to a simple attack, which does not even require encryption of known plaintext. Instead, this attack requires some knowledge

about the possible plaintexts. In particular, assume that the attacker knows that the plaintext is text in the English language. An attack of this type, that (only) assumes some known properties about the distribution of the input plaintext, is called a *cipher-text only attack* (CTO).

In our example, the CTO attacker makes use of known, measurable facts about English texts. The most basic fact is the distribution of letters used in English texts, as shown in Figure 2.7. Using this alone, it is usually quite easy to identify the letter E, as it is considerably more common than any other letter. Further cryptanalysis can use the guess for E to identify other letters, possibly using also the distributions of two letters and three letters; if the plaintext also contains spaces and punctuation marks, this can help the attacker significantly. This type of ciphertext-only attack (CTO) is called the *letter-frequency attack*.

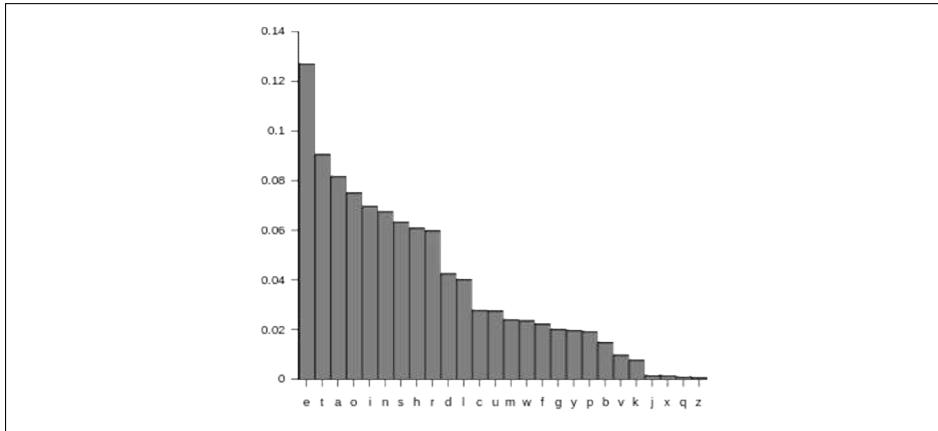


Figure 2.7: Distribution of letters in typical English texts.

Note that for the letter-frequency and other CTO attacks to be effective and reliable, the attacker requires a significant amount of ciphertext.

**Exercise 2.5.** Write a program that computes the distribution of letters in given texts, and run it over random texts of different sizes. Compare the resulting distributions to Fig. 2.7 and to each other (for consistency). You will see that longer texts tend to be much more consistent (and closer to Fig. 2.7).

In fact, this phenomena exists for other attacks too; cryptanalysis often require significant amount of ciphertext encrypted using the same encryption key. This motivates limiting the use of each cryptographic key to a limited amount of plaintext (and ciphertext), with the hope of making cryptanalysis harder or, ideally, infeasible.

**Principle 2** (Limit usage of each key). *Systems deploying ciphers/cryptosystems, should limit the amount of usage of each key, changing keys as necessary, to foil cryptanalysis attacks.*

Identifier	Cipher	Ciphertext	Plaintext	Time
A	Caesar	JUHDW SDUWB		
B	AzBy	ILFMW GZYOV		
C	ROT13	NYBAR NTNVA		
D		BLFMT OLEVI		
E		FZNEG UBHFR		
F		EUHDN UXOHV		

Table 2.1: Ciphertexts for Exercise 2.6. All plaintexts are pairs of two simple five-letter words. The three upper examples have the cipher spelled out, the three lower examples hide it ('obscurity'). It does not make them secure, but decryption may take a bit longer.

An extreme example of this is the One-Time Pad cipher, which we discuss later. The one-time pad is essentially the a one-bit substitution cipher - but with a different random mapping for each bit. This turns the insecure substitution cipher, into a *provably secure* cipher!

#### 2.1.4 Kerckhoffs' known-design principle

The confidentiality of keyless ciphers is completely based on the secrecy of the scheme itself, since it is enough to know the decryption process in order to decrypt - no key is required. However, even for keyed cryptosystems, it seems harder to attack without knowing the design; see Exercise 2.6. Therefore, in 'classical' cryptography, cryptosystems were kept secret and not published, to make cryptanalysis harder.

**Exercise 2.6.** *Table 2.1 shows six ciphertexts, all using very simple substitution ciphers. The ciphers used to encrypt the top three ciphertexts are indicated, but the ciphers used to encrypt the bottom three ciphertexts are not indicated. Decipher, in random order, all six ciphertexts and measure the time it took you to decipher each of them. Fill in the blanks in the table: the plaintexts, the time it took you to decipher each message, and the ciphers used for ciphertexts D, E and F. Did the knowledge of the cipher significantly ease the cryptanalysis process?*

One of the recent and quite famous examples of this policy are the encryption algorithms in the GSM network, which were kept secret - until eventually leaked. Indeed, soon after this leakage, multiple attacks were published; possibly the most important and interesting being a practical ciphertext only (CTO) attack [6]. One may conclude from this that, indeed, ciphers should remain secret; however, most experts believe that the opposite is true, i.e., that GSM designers should have used a published cryptosystem. In fact, newer cellular networks indeed use cryptosystems with published specifications.

The idea that ciphers should be designed for security even when known to attacker was presented already in 1883, by the Dutch cryptographer Auguste Kerckhoffs formulated. This is now known as the *Kerckhoffs' principle*, and considered as one of the basic principles of cryptography:

**Principle 3** (Kerckhoffs' known-design principle). *When designing or evaluating the security of (cryptographic) systems, assume adversary knows the design – everything except the secret keys.*

We intentionally put the word ‘cryptographic’ in parenthesis; this is since the principle is mostly accepted today also with regard to non-cryptographic security system such as operating systems and network security devices.

There are several reasons to adopt Kerckhoffs' principle. Kerckhoffs' original motivation was apparently the realization that cryptographic devices are likely to be captured by the enemy, and if secrecy of the design is assumed, this renders them inoperable - exactly in conflict situations, when they are most needed. The GSM scenario, as described above, fits this motivation; indeed, GSM designers did not even plan a proper ‘migration plan’ for changing from the exposed ciphers to new, hopefully secure ciphers.

Indeed, it appears that one reason to adopt Kerckhoffs' principle when designing a system is simply that this makes the designers more aware of possible attacks - and usually, result in more secure systems.

However, the best argument in favor of the Kerckhoffs' principle is that it allows public, published, standard designs, used for multiple applications. Such standardization and multiple applications have the obvious advantage of efficiency of production and support. However, in the case of cryptographic systems, there is an even greater advantage. Namely, a public, published design facilitates evaluation and cryptanalysis by many experts, which is best-possible guarantee for security and cryptographic designs - except for provably-secure designs, of course. In fact, even ‘probably-secure’ designs were often found to have vulnerabilities due to careful review by experts, due to a mistake in the proof or to some modeling or other assumption, often made implicitly.

Sometimes it is feasible to combine the benefits of open design (following Kerckhoffs' principle) and of secret design (placing another challenge on attackers), by combining two candidate schemes, one of each type, using a *robust combiner* construction, which ensures security as long as *one* of the two schemes is not broken [65]. For example, see Lemma 2.2 below.

## 2.2 Cryptanalysis Attack Models: CTO, KPA, CPA and CCA

As discussed in § 1.3 and in particular Principle 1, security should be defined and analyzed with respect to a clear, well defined model of the attacker capabilities, which we refer to as the *attack model*. In this section, we introduce for *cryptanalysis attack models*: *CTO*, *KPA*, *CPA* and *CCA*. Namely, these define capabilities of attackers trying to ‘break’ an encryption scheme.

We discussed above the exhaustive search and letter-frequency attacks, both requiring only the ability to identify correctly-decrypted plaintext (with significant probability); since these attacks do not require any pairs of plaintext and ciphertext messages, we refer to it as a ciphertext-only (CTO) attack.

To facilitate CTO attacks, the attacker must have some knowledge about the distribution of plaintexts. In practice, such knowledge is typically due to the specific application or scenario, e.g., when it is known that the message is in English, and hence the attacker can apply known statistics such as the letter-distribution histogram Figure 2.7. For a formal, precise definition, we normally allow the adversary to *pick* the plaintext distribution, but define security carefully to avoid ‘trivial’ attacks which simply exploit this ability of the adversary.

We also discussed the table look-up and time-memory tradeoff attacks; in both of these generic attacks, the adversary must be able to obtain encryption of one or few specific plaintext messages - the messages used to create the pre-computed table. We refer to such attacks as *chosen-plaintext attacks (CPA)*. We say that the CPA attack model is *stronger* than the CTO attack model, meaning that every cryptosystem vulnerable to CTO attack, is also vulnerable to CPA.

There are two additional common attack models for encryption schemes, which represent two different variants of the CPA attack. The first is the *known-plaintext attack (KPA) model*, where the attacker receives one or multiple pairs of plaintext and the corresponding ciphertext, but the plaintext being chosen randomly.

**Exercise 2.7** ( $CPA > KPA > CTO$ ). *Explain (informally) why every cryptosystem vulnerable to CTO attack, is also vulnerable to KPA, and every cryptosystem vulnerable to KPA, is also vulnerable to CPA. We say that the KPA model is stronger than the CTO model and weaker than the CPA mode.*

Finally, in the *chosen-ciphertext attack (CCA)* attack model, the attacker has the ability to receive the decryption of arbitrary ciphertexts, chosen by the attacker. We adopt the common definition, where CCA-attackers are also allowed to perform CPA attacks, i.e., attacker can obtain the encryptions of attacker-chosen plaintext messages. With this definition, trivially,  $CCA > CPA$ . Combining this with the previous exercise, we have the complete ordering:  $CCA > CPA > KPA > CTO$ .

### 2.3 Generic attacks and the Key-Length Principle

In this section we first discuss two *generic attacks*, i.e., attacks which work for all (or many) schemes, without depending on the specific design of the attacked scheme, but only on general properties, such as key length. In subsection 2.3.1 we discuss the *exhaustive search* generic attack, and in subsection 2.3.2 we discuss the *table look-up* and *time-memory tradeoff* generic attacks.

Exhaustive search is a ciphertext-only (CTO) attack, like the letter-frequency attack presented earlier. In contrast, the table look-up and time-memory tradeoff attacks, require some *known-plaintext*; attacks which require known-plaintext are called *known-plaintext attacks (KPA)*. In § 2.2 we introduce two additional attack models, the *chosen-plaintext attack (CPA)* and the *chosen-ciphertext attack (CCA)*.

Finally, in subsection 2.3.3 we present the *key-length principle*, which essentially says that security requires the use of cryptographic keys which are ‘long enough’ to foil generic attacks - and other known attacks.

### 2.3.1 Exhaustive search

Following the Kerckhoffs’ principle, we assume henceforth that all designs are known to the attacker; defense is only provided via the secret keys. Therefore, one simple attack strategy, is to try to decrypt ciphertext messages using all possible keys, and detect the key - or (hopefully few) keys - where the decryption seems to result in possible plaintext, and discard keys which result in clearly-incorrect plaintext. For example, if we know that the plaintext is text in English, we can test candidate plaintexts, e.g., by being reasonably-close to the distribution of letters in English (Figure 2.7). This attack is called *exhaustive key search*, *exhaustive cryptanalysis* or *brute force attack*; we will use the term exhaustive search.

Like other CTO attacks, exhaustive search has some requirements on the plaintext space; for exhaustive search, it should be possible to efficiently identify valid plaintext. In addition, exhaustive search may not work for stateful ciphers, since decryption depends on the state and not only on the key. Finally, decoding of an arbitrary plaintext with an incorrect key should usually result in clearly-invalid plaintext. Exhaustive search can be applied whenever these properties hold, i.e., it is a generic CTO attack.

Exhaustive search is practical only when the key space is not too large. For now we focus on symmetric ciphers, where the key is usually an arbitrary binary string of given length, namely the key space is  $2^l$  where  $l$  is the key length, i.e., the problem is the use of insufficiently-long keys. Surprisingly, designers have repeatedly underestimated the risk of exhaustive search, and used ciphers with insufficiently long keys, i.e., insufficiently large key space.

Let  $T_S$  be the *sensitivity period*, i.e., the duration required for maintaining secrecy, and  $T_D$  be the time it takes to test each potential key, by performing one or more decryptions. Hence, attacker can test  $T_S/T_D$  keys out of the keyspace containing  $2^l$  keys. If  $T_S/T_D > 2^l$ , then the attacker can test all keys, and find the key for certain (probability 1); otherwise, the attacker succeeds with probability  $\frac{T_S}{T_D \cdot 2^l}$ . By selecting sufficient key length, we can ensure that the success probability is as low as desired.

For example, consider the conservative assumption of testing a billion keys per second, i.e.,  $T_D = 10^{-9}$ , and requiring the security for a three thousand years, i.e.,  $T_S = 10^8$ , with probability of attack succeeding at most 0.1%. We find that to ensure security with these parameters against brute force attack,

we need keys of length  $l \geq \log_2 \left( \frac{T_S}{T_D} \right) = \log_2(10^{17}) < 60$ . This is close to the length of the Data Encryption Standard (DES), published in 1977.

The above calculation assumed a minimal time to test each key. Of course, attackers will often be able to test many keys in parallel, by using multiple computer and/or parallel processing, using hardware implementations, parallel computing, often using widely-available GPUs, or multiple computers. Such methods were used during 1994-1999 in multiple demonstrations of the vulnerability of the *Data Encryption Standard (DES)* to different attacks. The final demonstration was exhaustive-search completing in 22 hours, testing many keys in parallel using a 250,000\$ dedicated-hardware machine ('deep crack') together with distributed.net, a network of computers contributing their idle time.

However, the impact of such parallel testing, as well as improvements in processing time, is easily addressed by reasonable extension of key length. Assume that an attacker is able to test 100 million keys in parallel during the same  $10^{-9}$  second, i.e.,  $T_D = 10^{-17}$ . With the same goals and calculation as above we find that we need keys of length  $l \geq \log_2 \left( \frac{T_S}{T_D} \right) = \log_2(10^{26}) < 100$ . This is far below even the minimal key length of 128 bits supported by the Advanced Encryption Standard (AES). Therefore, exhaustive search is not a viable attack against AES or other ciphers with over 100 bits.

**Testing candidate keys.** Recall that we assume that decrypting arbitrary ciphertext with an incorrect key should usually result in clearly-invalid plaintext. Notice our use of the term 'usually'; surely there is *some* probability that decryption with the wrong key, will result in seemingly-valid plaintext. Hence, exhaustive search may often not return only the correct secret key. Instead, quite often, exhaustive search may often return *multiple* candidate keys, which all resulted in seemingly-valid decryption. In such cases, the attacker must now test each of these candidate keys, by trying to decrypt additional ciphertexts, and discarding a key when its decryption of some ciphertext appears to result in invalid plaintext.

### 2.3.2 Table Look-up and Time-memory tradeoff attacks

Exhaustive search is very computation-intensive; it finds the key, on the average, after testing half of the keyspace. On the other hand, its storage requirements are very modest, and almost<sup>2</sup> independent of the key space.

In contrast, the *table look-up attack*, which we next explain, uses  $O(2^l)$  memory, where  $l$  is the key length, but only table-lookup time. However, this requires ciphertext of some pre-defined plaintext message, which we denote  $p$ .

In the table look-up attack, the attacker first *pre-computes*  $T(k) = E_k(p)$  for every key  $k$ . Later, the attacker asks for the encryption  $c^* = E_{k^*}(p)$  of the same plaintext  $p$ , using the unknown secret key  $k^*$ . The attacker looks up

---

<sup>2</sup>Exhaustive search needs storage for the key guesses.

the table  $T$  and identifies all the keys  $k$  such that  $c^* = T(k)$ . The number of matching keys is usually one or very small, allowing the attacker to quickly rule out the incorrect keys (by decrypting some additional ciphertext messages).

The table look-up attack requires  $O(2^l)$  storage, to ensure  $O(1)$  computation, while the exhaustive search attack uses  $O(1)$  storage and  $O(2^l)$  computations. Several more advanced generic attacks allow different *tradeoffs* between the computing time and the storage requirements of different attacks; the first and most well known such attack was published by Martin Hellman [64]. These tradeoffs are based on the use of cryptographic hash functions, which we discuss chapter 4.

### 2.3.3 The sufficient effective key length principle

The use of longer keys, e.g. 128 bits, is *not* a sufficient condition for security. For example, we saw the the general mono-alphabetic substitution cipher (subsection 2.1.3) is insecure, although its key space is relatively large; increasing its key length, e.g., by adding symbols, will not improve security significantly.

Exhaustive search is often used as a measure to compare against; we say that a cipher using  $k$  bit keys has *effective key length*  $l$  if it the best attack known against it takes about  $2^l$  operations, where  $k \geq l$ . We expect the effective key length of good symmetric ciphers to be close to their real key length, i.e.,  $l$  should not be ‘too smaller’ cf. to  $k$ . For important symmetric ciphers, any attack which increases the gap between  $k$  and  $l$  would be of great interest, and as the gap grows, there will be increasing concern with using the cipher. The use of key lengths which are 128 bit and more leaves a ‘safety margin’ against the potential better future attacks, and gives time to change to a new cipher as better attacks will be found.

Note that for asymmetric cryptosystems, there is often a large gap between the real key length  $l$  and the effective key length  $k$ . This is considered acceptable, since the design of asymmetric cryptosystems is challenging, and it seems reasonable to expect attacks with performance much better than exhaustive search; in particular, in most public key systems, the secret key is not an arbitrary, random binary string. In particular, currently, the use of the RSA cryptosystem with keys below 2000 bits is considered insecure.

Normally, we select sufficient key length to ensure security against any conceivable adversary, e.g., leaving a reasonable margin above effective key length of say 100 bits; a larger margin is required when the *sensitivity period* of the plaintext is longer. The the cost of using slightly longer key is usually well justified, considering the damages of loss of security and of having to change in a hurry to a cipher with longer effective key length.

In some scenarios, however, the use of longer keys may have significant costs; for example, doubling the key length in the RSA cryptosystem, increases the computational costs by about six. We therefore may also consider the risk from exposure, as well as the resources that a (rational) attacker may deploy to break the system. This is summarized by the following principle.

**Principle 4** (Sufficient effective key length). *Deployed cryptosystems should have sufficient effective key length to foil feasible attacks, considering the maximal expected adversary resources, as well as cryptanalysis and speed improvements expected over the sensitivity period of the plaintext.*

Experts and different standardization and security organizations, publish estimates of the required key length of different cryptosystems (and other cryptographic schemes); we present few recommendations in Table 6.1, which we present later on, in subsection 6.1.5.

## 2.4 Unconditional security and One Time Pad

The exhaustive search and table look-up attacks are generic - they do not depend on the specific design of the cipher: their complexity is merely a function of key length. This raises the natural question: is every cipher breakable, given enough resources? Or, can encryption be secure *unconditionally* - even against attacker with unbounded resources (time, computation speed, storage)?

We next present such unconditionally secure cipher, the *One Time Pad (OTP)*. The one time pad is often attributed to an 1919 patent by Gilbert Vernam [111], although some of the critical aspects may have been due to Mauborgne [20], and in fact, the idea was already proposed by Frank Miller at 1882 [19]; we again refer readers to the many excellent references on history of encryption, e.g., [69, 104].

The one time pad is not just unconditionally secure - it is also an exceedingly simple and computationally efficient cipher. Specifically:

**Encryption:** To encrypt message, compute its bitwise XOR with the key. Namely, the encryption of each plaintext bit, say  $m_i$ , is one ciphertext bit,  $c_i$ , computed as:  $c_i = m_i \oplus k_i$ , where  $k_i$  is the  $i^{\text{th}}$  bit of the key.

**Decryption:** Decryption simply reverses the encryption, i.e., the  $i^{\text{th}}$  decrypted bit would be  $c_i \oplus k_i$ .

**Key:** The key  $k = \{k_1, k_2, \dots\}$  should consist of independently drawn fair coins, and its length must be at least as long as that of the plaintext.

See illustration in Figure 2.8.

To see that decryption recovers the plaintext, observe that given  $c_i = m_i \oplus k_i$ , the corresponding decrypted bit is  $c_i \oplus k_i = (m_i \oplus k_i) \oplus k_i = m_i$ , as required.

The unconditional secrecy of OTP was recognized early on, and established rigorously in a seminal paper published in 1949 by Claude Shannon [102]. In that paper, Shannon also proved that *every unconditionally-secure cipher must have keys as long as the plaintext*, namely, that as long as unconditional secrecy is required, this aspect cannot be improved. For proofs and details, please consult a textbook on cryptology, e.g., [106].

The cryptographic literature has many beautiful results on unconditional security. However, it is rarely practical to use such long keys, and in practice,

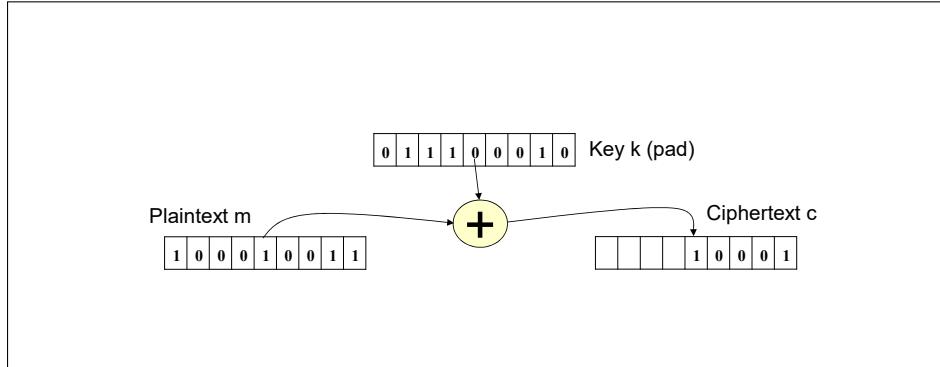


Figure 2.8: The One Time Pad (OTP) cipher.

adversaries - like everyone else - have limited computational abilities. Therefore, in this course, we focus on computationally-bounded adversaries.

While the key required by OTP makes it rarely practical, we next demonstrate how OTP can be used as a core component of practical schemes, which use shorter keys (and are ‘only’ secure against computationally-bounded attackers). In particular, we next show a computationally-secure variant of OTP, where the key can be much smaller than the plaintext.

The attentive reader may have noticed that OTP is not a shared-key cryptosystem as defined in Definition 2.1, since that definition did not allow the encryption process to be stateful. We therefore extend the definition so that the encryption and decryption operations may have another input and another output. The additional input is the current state, and the additional output is the next state; both are from a set  $S$  of possible states.

**Definition 2.2** (Shared-key cryptosystem - allowing state). *A shared-key cryptosystem is a pair of keyed algorithms,  $\langle E, D \rangle$ , each with two inputs and two outputs. We say that  $\langle E, D \rangle$  ensures correctness if for every message  $m \in M$ , key  $k \in K$  and state  $s \in S$  holds that if  $(c, s') = E_k(m, s)$  then  $(m, s') = D_k(c, s)$ .*

**Exercise 2.8.** Define the (stateful) encryption and decryption functions  $\langle E, D \rangle$  for the OTP cipher.

Hint: use the number of bits produced so far as the state. Or see solution in chapter 10.  $\square$

## 2.5 Stream Ciphers and Pseudo-Randomness

### 2.5.1 Stream Ciphers

In contrast to the ancient ciphers of § 2.1, the One Time Pad is a *stream cipher*, i.e., it is stateful: the decryption of a plaintext bit depends on its

location within the plaintext string (or stream). Stream ciphers are often used in applied cryptography: rarely, OTP or another unconditionally-secure cipher, but much more commonly, with a bounded-length key, providing computational security.

One simple and common way to implement such a bounded-key-length stream cipher, is to use the key as the input to a *pseudo-random generator (PRG)*, which we discuss in subsection 2.5.2. Given a ‘short’ random string, say of length  $l$ , the PRG outputs a longer string which is *pseudo-random* - i.e., cannot be distinguished from a true random string (in an efficient way). Actually it suffices for the input to be pseudo-random, to ensure pseudo-random output. Some PRGs may produce output which is only slightly longer than their inputs, but since the input can be pseudo-random, we can invoke the PRG repeatedly, to expand a short random input string to much longer output. We then XOR each plaintext bit with the corresponding pseudo-random bit, as shown in Figure 2.9.

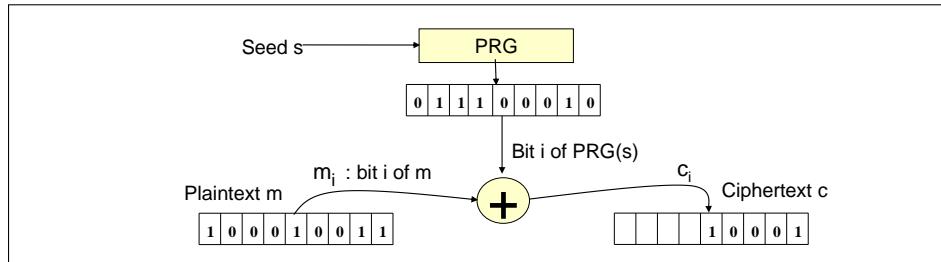


Figure 2.9: Pseudo-Random Generator Stream Cipher. The input to the PRG is usually called either key or seed; if the input is random, or pseudo-random, then the (longer) output string is pseudo-random.

The pseudo-random generator stream cipher is very similar to the OTP; the only difference is that instead of using a truly random sequence of bits to XOR the plaintext bits, we use the output of a *pseudo-random generator (PRG)*. If we denote the  $i^{th}$  output bit of  $PRG(s)$  by  $PRG_i(s)$ , we have that the  $c_i$ , the  $i^{th}$  ciphertext bit, is defined as:  $c_i = m_i \oplus PRG_i(s)$ . This is a stateful shared-key cryptosystem, quite similar to the OTP; specifically, the state is the index of the bit  $i$ , encryption  $E_s(m_i, i)$  returns  $(m_i \oplus PRG_i(s))$ , and decryption  $D_s(c_i, i)$  returns  $(c_i \oplus PRG_i(s))$ .

In subsection 2.5.2 below, we discuss PRGs. Following that, in subsection 2.5.4, we discuss Pseudo-Random Functions (PRFs), which allow *stateless stream ciphers*.

### 2.5.2 Pseudo-Random Generators (PRGs)

A PRG is an *efficient* algorithm, whose input is a binary string  $s$ , called *seed* (or sometimes *key*); if the input is random, or pseudo-random, then the (longer) output string is pseudo-random.

The concept of a PRG is our first well-defined cryptographic mechanism; it is based on the ingenious idea of *indistinguishability tests*, which is beautiful - but not trivial to understand. This concept was first proposed by Alan Turing, already at 1950, in a famous paper which explored what should be considered an *intelligent machine*; this paper is one of the foundations of artificial intelligence. Turing proposed the test illustrated in Figure 2.10, as a possible definition of an *intelligent machine*.

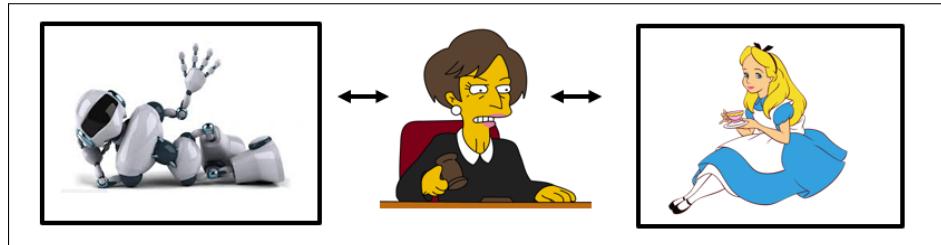


Figure 2.10: The Turing Indistinguishability Test. A machine is considered *intelligent*, if a *distinguisher* (judge) cannot determine in which box is the machine and in which is a human. Turing stipulated that communication between the distinguisher and the boxes will only be in printed form, to avoid what he considered as ‘technical’ challenges such as voice recognition.

Many cryptographic mechanisms are defined using indistinguishability tests, which are similar, in their basic concept, to the Turing indistinguishability test. We now present the *pseudorandom generator indistinguishability test*, used to define a (secure) pseudo-random generator.

The PRG indistinguishability test is illustrated in Figure 2.11. Consider function  $f$  from  $n$ -bits strings to  $m$ -bit strings, where  $m > n$  (the output is longer than the input). Let  $seed$  be a random string in the domain, i.e.,  $seed \xleftarrow{\$} \{0, 1\}^n$ , and  $rand$  be a random string in the range, i.e.,  $rand \xleftarrow{\$} \{0, 1\}^m$ . We say that function  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$  is a (secure) pseudo-random generator (PRG), if  $m > n$ , and a distinguisher can't efficiently distinguish between  $rand \xleftarrow{\$} \{0, 1\}^m$  and  $f(seed)$ , for  $seed \xleftarrow{\$} \{0, 1\}^n$ , for any efficient distinguisher algorithm (deterministic or randomized).

Like many other cryptographic schemes, a PRG may be defined for a fixed or variable input length. For both *fixed input length (FIL)* and *variable input length (VIL)* PRGs, the output is also a binary string, which is always longer than the input:  $|PRG(s)| > |s|$ . Furthermore, for any inputs of the same length  $|s| = |s'|$ , the outputs are *longer* binary strings - also of the same length:  $|PRG(s)| = |PRG(s')| > |s| = |s'|$ . Typically the output length is a simple function of the input length, e.g.,  $|PRG(s)| = 2 \cdot |s|$  or  $|PRG(s)| = |s| + 1$ .

The goal of a PRG is to ensure that its output bits are *pseudorandom*. Intuitively, this implies that these bits are ‘indistinguishable’ from truly random bits. Namely, select a random bit  $b$  fairly, i.e., select  $b = 1$  with probability  $\frac{1}{2}$

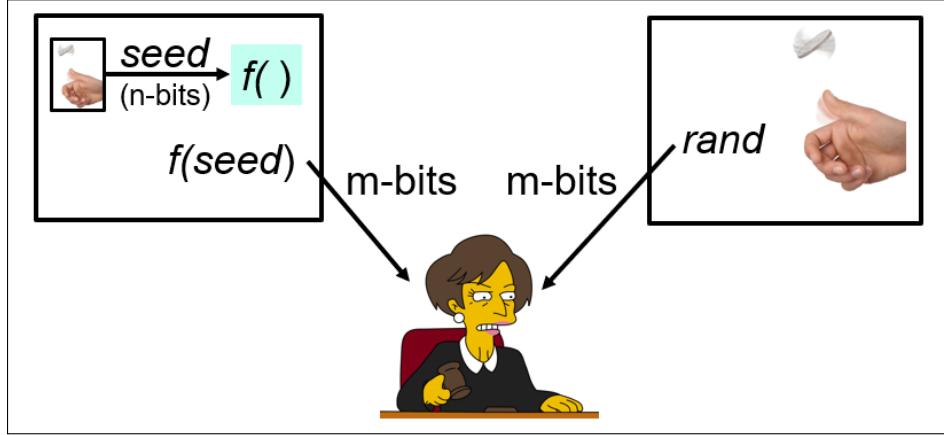


Figure 2.11: The Pseudo-Random Generator (PRG) Indistinguishability Test. We say that function  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$  is a (secure) pseudo-random generator (PRG), if  $m > n$ , and a distinguisher can't efficiently distinguish between  $rand \xleftarrow{\$} \{0, 1\}^m$  and  $f(seed)$ , for  $seed \xleftarrow{\$} \{0, 1\}^n$ .

(and  $b = 0$  also with probability  $\frac{1}{2}$ . If  $b = 1$ , the adversary is given the output of the PRG, and if  $b = 0$ , the adversary is given a truly random string of the same length. The PRG is secure, if the adversary cannot guess if  $b = 0$  or  $b = 1$ , any better than the random guess (being correct with probability  $\frac{1}{2}$ ).

We usually refer to the algorithm used by the attacker to determine if a given string is truly random or pseudorandom as a *distinguisher*  $D$ . Intuitively, the goal of the distinguisher  $D$  is to output 1 (true) if it is given a pseudo-random string (output of the PRG), and 0 (false) if it is given a truly random string - i.e., to ‘identify the pseudo-random string’. We therefore define the ‘advantage’ of  $D$  as the probability that  $D$  is outputs 1 when given the output of  $PRG$ , minus the probability that  $D$  is outputs 1 when given a truly random string  $r$ .

Equation (2.10) defines the *advantage* of a given (attacking) *distinguishing* algorithm  $D$ , for a given pseudorandom generator  $PRG$ , input length  $n$  and output length  $l(n)$ .

$$ADV_{PRG,D}(n) \equiv \Pr_{s \xleftarrow{\$} \{0,1\}^n} [D(PRG(s))] - \Pr_{r \xleftarrow{\$} \{0,1\}^{l(n)}} [D(r)] \quad (2.10)$$

The probabilities in Eq. (2.10) are computed for uniformly-random  $n$ -bit binary string  $s$  (seed), and uniformly-random  $|PRG(1^n)|$ -bit binary string  $r$ . The probability is also taken over the random coins of the distinguisher  $D$ ; the  $PRG$  algorithm is deterministic.

Intuitively, a PRG is secure if there is no ‘*effective distinguisher*’ for it, i.e., an distinguisher  $D$  that obtains ‘significant advantage’ within reasonable investment of resources (computation time). In practice, we use fixed input

length (FIL) PRGs, which we consider as secure, after sufficient efforts are invested in vain in attempts to design an effective distinguisher  $D$ .

However, it is not clear how to define security for a fixed-input-length PRG; instead, we define security only for variable input length (VIL) PRGs. A distinguisher  $D$  is successful if it obtains *non-negligible* advantage - i.e., advantage which exceeds some (positive) polynomial in the input length  $n$ .

We first define the notion of a *negligible function*. Intuitively, this is a function  $f(n)$  which is bounded by some polynomial  $p(n)$ , for ‘sufficiently large’  $n$ . The following definition states the same thing, just a bit more rigorously.

**Definition 2.3** (Negligible function). *We say that a function  $f : \mathbb{N} \rightarrow \mathbb{R}$  (from integers to real numbers) is negligible, if for every constant  $c$ , there is some number  $n' \in \mathbb{N}$  s.t. for all  $n > n'$  holds  $|f(n)| < n^c$ . We denote the set of negligible functions by  $\text{NEGL}$ . If necessary, we explicitly identify the parameter, as in  $\text{NEGL}(n)$ .*

**Exercise 2.9.** *which of the following functions are negligible? Prove your responses:* (a)  $f_a(n) = 10^{-8} \cdot n^{-10}$ , (b)  $f_b(n) = 2^{n/2}$ , (c)  $f_c(n) = \frac{1}{n!}$ .

Working with negligible functions is a useful simplification; here is one convenient property, which shows that if an algorithm has negligible probability to ‘succeed’, then running it a polynomial number of time, will not help - the probability to succeed will remain negligible.

**Lemma 2.1.** *Consider negligible function  $f : \mathbb{N} \leftarrow \mathbb{R}$ , i.e.,  $f(n) \in \text{NEGL}$ . Then for any polynomial  $p(n)$ , the function  $g(n) = f(p(n))$  is also negligible, i.e.,  $g(n) = f(p(n)) \in \text{NEGL}$ .*

We can now finally define a secure PRG. The definition assumes both the PRG and the distinguisher  $D$  are *polynomial-time*, meaning that their running time is bounded by a polynomial in the input length. Polynomial-time algorithms are often referred to simply by the term *efficient* algorithms. Applied cryptography, and most of the theoretical cryptography, only deal with polynomial-time (efficient) algorithms, schemes and adversaries. Note that the PRG must be deterministic, but the distinguisher  $D$  may be probabilistic (randomized).

**Definition 2.4** (Secure Pseudo-Random Generator (PRG)). *A deterministic polynomial time algorithm PRG is a Secure Pseudo-Random Generator (PRG), if for every Probabilistic Polynomial Time (PPT) distinguisher  $D$ , the advantage of  $D$  for PRG is negligible, i.e.,  $\text{ADV}_{\text{PRG}, D}(n) \in \text{NEGL}$ .*

The term ‘secure’ is often omitted; i.e., when we say that algorithm  $f$  is a pseudo-random generator (PRG), this implies that it is a secure PRG.

### 2.5.3 Secure PRG Constructions

Note that we did not present a *construction* of a secure PRG. In fact, if we could present a provably-secure construction of a secure PRG, satisfying Def. 2.4, this would immediately prove that  $P \neq NP$ , solving the most well-known open problems in the theory of complexity. Put differently, if  $P = NP$ , then there cannot be any secure PRG algorithm (satisfying Def. 2.4). The same holds for most of the cryptographic mechanisms we will learn in this course, including secure encryption, if we allow messages to be longer than the key. Of course, the One Time Pad (OTP) em is an example of secure encryption, but there, the key is as long as the plaintext.

What *is* possible, is to present a conditionally-secure construction of a PRG, namely, a PRG which is secure if some assumption holds. Such constructions are proven secure by a *provable reduction* to the underlying assumption, showing that if the construction fails to satisfy Def. 2.4, then the underlying assumption is false. Courses and books on cryptography are full of such reduction proofs, e.g., see [57, 58, 106]. Exercise 2.10 and Fig. 2.12 present an example of such reduction. We present some relative-simple examples, mostly in the exercises at the end of the chapter, which should hopefully suffice to build intuition on identifying insecure designs and constructing secure designs; e.g., Exercise 2.31.

**Exercise 2.10.** Let  $G : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$  be a secure PRG. Is  $G'(r||s) = r||G(s)$ , where  $r, s \in \{0, 1\}^n$ , also a secure PRG?

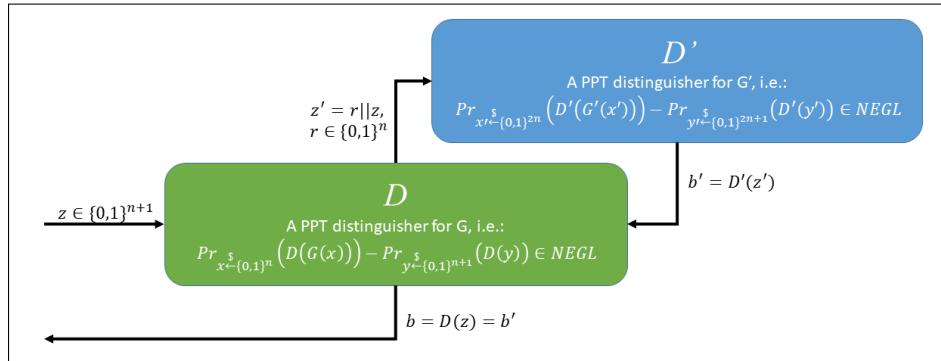


Figure 2.12: Reduction of Ex. 2.10: given an ‘effective distinguisher’  $D'$  for  $G'$ , we use it for an ‘effective distinguisher’ for  $G$ ; hence, if  $G$  is a secure PRG (no ‘effective distinguisher’), then  $G'$  must also be secure PRG.

*Solution:* Yes, if  $G$  is a PRG then  $G'$  is also a PRG. Assume, to the contrary, that there is a ‘good PPT algorithm  $D'$  that distinguishes, with non-negligible probability, between the output of  $G'$  and a random string of same length ( $2n + 1$  bits). Namely, for random strings  $x, y$  of length  $2n, 2n + 1$  respectively,

holds  $\Pr(D'(G'(x))) - \Pr(D'(y))$  is significant. Let  $D(z) = D'(r||z)$ , where  $r \xleftarrow{\$} \{0,1\}^n$  and  $z \in \{0,1\}^n$ . Now,

$$\Pr_{x \xleftarrow{\$} \{0,1\}^n} (D(G(x))) = \Pr_{r \xleftarrow{\$} \{0,1\}^n} \left[ \Pr_{x \xleftarrow{\$} \{0,1\}^n} (D'(r||G(x))) \right] = \Pr_{x' \xleftarrow{\$} \{0,1\}^{2n}} (D'(G'(x'))) \quad \text{and}$$

$$\Pr_{y \xleftarrow{\$} \{0,1\}^{n+1}} (D(y)) = \Pr_{r \xleftarrow{\$} \{0,1\}^n} \left[ \Pr_{y \xleftarrow{\$} \{0,1\}^{n+1}} (D'(r||z)) \right] = \Pr_{y' \xleftarrow{\$} \{0,1\}^{2n+1}} (D'(y'))$$

From Eq. (EQ:ADV:PRG),

$$\begin{aligned} ADV_{G,D}(n) &= \Pr_{x \xleftarrow{\$} \{0,1\}^n} (D(G(x))) - \Pr_{y \xleftarrow{\$} \{0,1\}^{n+1}} (D(y)) \\ &= \Pr_{x' \xleftarrow{\$} \{0,1\}^{2n}} (D'(G'(x'))) - \Pr_{y' \xleftarrow{\$} \{0,1\}^{2n+1}} (D'(y')) \\ &= ADV_{G',D'}(2n) \end{aligned}$$

The claim follows from Lemma 2.1.  $\square$

There are many proposed designs for PRGs. Many of these are based on a Feedback Shift Registers (FSRs), with a known linear or non-linear feedback function  $f$ , as illustrated in Fig. 2.13. For Linear Feedback Shift Registers (LFSR), the feedback function  $f$  is simply XOR of some of the bits of the register. Given the value of the initial bits  $r_1, r_2, \dots, r_l$  of an FSR, the value of the next bit  $r_{l+1}$  is defined as:  $r_{l+1} = f(r_1, \dots, r_l)$ ; and following bits are defined similarly:  $(\forall i > l) r_i = f(r_{i-l}, \dots, r_{i-1})$ .

FSRs are well studied with many desirable properties. However, by definition, their state is part of their output. Hence, they cannot directly be used as cryptographic PRGs. Instead, there are different designs of PRGs, often combining multiple FSRs (often LFSRs) in different ways. For example, the A5/1 and A5/2 stream ciphers defined in the GSM standard, combined three linear shift registers.

Feedback shift registers are convenient for hardware implementations. There are other PRG designs which are designed for software implementations, such as the RC4 stream cipher.

There are many cryptanalysis attacks on different stream ciphers, including the three mentioned above (RC4 and the two GSM ciphers, A5/1 and A5/2); presenting details of these ‘classical’ ciphers and attacks on them is beyond our scope, see, e.g., [6] for GSM’s A5/1 and A5/2, and [74, 84] for RC4.

Instead, we next give an example of a practical attack against MS-Word 2002, which exploits a vulnerability in the *usage* of RC4 rather than its in design. Namely, this attack could have been carried out if *any* PRG was used incorrectly as in MS-Word 2002.

**Example 2.1.** *MS-Word 2002 used RC4 for document encryption, in the following way. The user provided password for the document; that password was*

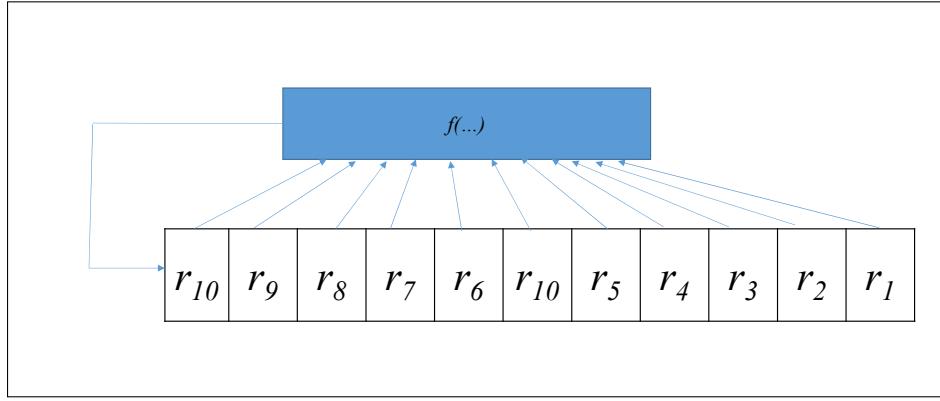


Figure 2.13: Feedback Shift Register, with (linear or non-linear) feedback function  $f()$ .

*used as a key to the RC4 PRG, producing a long pseudo-random string which is referred to as Pad, i.e.,  $\text{Pad} = \text{RC4}(\text{password})$ . When the document is saved or restored from storage, it is XORed with Pad. This design is vulnerable; can you spot why?*

*The vulnerability is not specific in any way to the choice of RC4; the problem is in how it is used. Namely, this design re-uses the same pad whenever the document is modified - a ‘multi-times pad’ rather than OTP. The plaintext Word documents contain sufficient redundancy, to allow decryption. See details in [85].*

This vulnerability is not unusual; in fact, cryptanalysis is a rather rare tool in cyber attacks, which usually use ‘system vulnerabilities’, often due to *incorrect usage* of cryptography rather than with the design of the cryptographic schemes themselves.

#### 2.5.4 Pseudo-random functions (PRFs)

One practical drawback of stream ciphers is the fact that they require state, to remember how many bits (or bytes) were already output. What happens if state is lost? Can we reduce the use and dependency on state, to allow recovery from loss of state, or to avoid the use of state when encryption is not used, e.g., between one message and the next?

We next introduce a new pseudo-random primitive, called *pseudo-random functions (PRFs)*, which allows to provide such ‘stateless stream cipher’. More precisely, a PRF allows a cryptosystem which uses stream-cipher for each message or as long as state is kept, but can re-initialize the stream cipher when state is lost (e.g., upon beginning a new message).

**Random functions** Before we discuss pseudo-random functions, let us briefly discuss their ‘ideal’ - ‘truly’ random functions. Both random functions and PRFs are defined with respect to a given domain  $D$  and range  $R$ ; in a typical case, both  $D$  and  $R$  are binary strings, i.e., for some integers  $n, m$ , we have  $R = \{0, 1\}^n$ ,  $D = \{0, 1\}^m$ .

A random function is a randomly-chosen mapping from domain  $D$  to range  $R$  (or a subset of  $R$ ). Namely, it is the result of selecting random values from the range  $R$ , to each value in the domain  $D$ .

In principle, we can select a random function. To select a random function from  $R = \{0, 1\}^n$  to  $D = \{0, 1\}^m$  requires  $n \cdot 2^m$  coin flips; this can be easily done for small domain and range, as in Exercise 2.11.

**Exercise 2.11.** *Using a coin, select randomly a function:*

1.  $f_1 : \{0, 1\} \rightarrow \{0, 1\}$
2.  $f_2 : \{0, 1\} \rightarrow \{0, 1\}^3$
3.  $f_3 : \{0, 1\}^3 \rightarrow \{0, 1\}$

*What was the amount of coin flips requires for each function? Compare to  $n \cdot 2^m$ .*

In the typical case where the domain is large, the choice of a random function requires excessive storage and coin-flip operations. Selecting and storing it is difficult. Furthermore, in cryptography, it is useful to have a secret random function which is shared between two or more parties, e.g., Alice and Bob. Securely communicating such a huge table can be infeasible.

The construction of a ‘stateless stream cipher’ is one application which could be solved easily if the parties share a secret, randomly-chosen function. Specifically, whenever the state is lost, one party can generate a random string  $r$  and *send it, in the clear* to the other party. Both parties now compute  $f(r)$ , and use it as the seed to a PRG. If the domain is large enough, with high probability the same seed was not used previously.

The problem with this design, is that the domain must be huge, to ensure negligible probability of choosing the same seed. However, using such a huge domain requires generating and securely communicating a huge table containing the function, which is often impractical.

**Pseudo-Random Functions (PRFs)** Pseudo-random functions are an efficient substitute to random functions, which ensures similar properties, while requiring the generation and sharing of only short keys. The main limitation is that PRFs are secure only against computationally bounded adversaries.

A PRF scheme has two inputs: a *secret key*  $k$  and a ‘message’  $m$ ; we denote it as  $PRF_k(m)$ . Once  $k$  is fixed, the PRF becomes only a function of the message. The basic property of PRF is that this function ( $PRF_k(\cdot)$ ) is *indistinguishable* from a truly random function. Intuitively, this means that a PPT adversary cannot tell if it is interacting with  $PRF_k(\cdot)$  with domain  $D$

and range  $R$ , or with a random function  $f$  from  $D$  to  $R$ . Hence, PRFs can be used in many applications, providing an efficient, easily-deployable to the impractical use of truly random functions.

For example, PRFs can be used to construct a ‘stateless’ stream cipher; in fact, there are three natural methods to do it. The first method is to simply replace the use of random function in the construction described in subsection 2.5.1, which uses a PRG. The other method is similar, except that it includes the PRG functionality within it, i.e., it only requires the PRF. See Fig. 2.14.

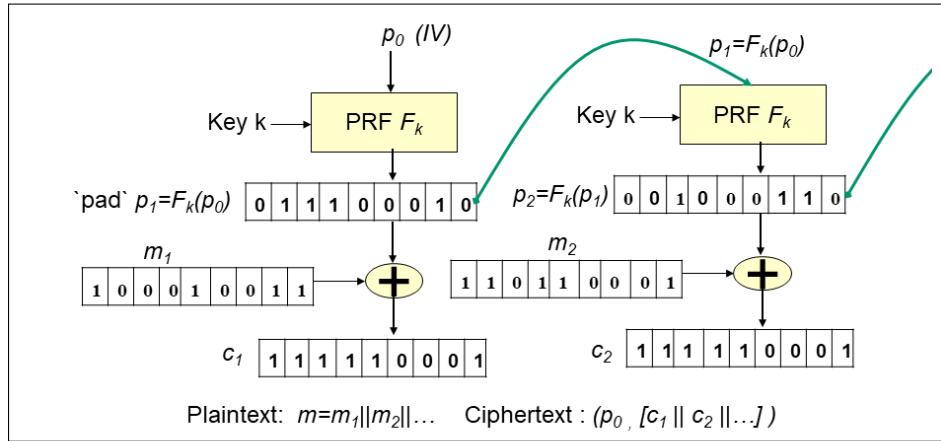


Figure 2.14: Pseudo-Random Function ‘Stateless’ Stream Cipher. The input  $p_0$  should be non-repeating, e.g., randomly, from sufficiently large domain; it is often referred to as *IV* (*Initialization Vector*). A new value for  $p_0$  is chosen for every message, or whenever there is no prior state.

Before we define a PRF, we first define the *oracle access notation*. Oracle access is a central concept in complexity theory and cryptography; see [57]. Basically, it means that  $ADV$  may provide input to the corresponding oracle and receive back the responses, without knowing which implementation of the oracle it interacts with (in this case, the truly-random function or the PRF).

**Definition 2.5** (Oracle notation). *Let  $F$  by a function (or an algorithm implementing a function). We use the notation  $A^F$  to denote an algorithm  $A$ , which can, as part of its operation, provide inputs to  $F$  and receive the corresponding outputs. We refer to  $F$  as an oracle and to  $A^F$  as algorithm  $A$  with oracle  $F$ . We may specify some of the inputs to the oracle, allowing  $A$  to specify only the others, e.g.,  $A_k^F(\cdot, b)$ , which implies that  $A$  may receive the results of  $F_k(a, b)$  for values  $a$  chosen by  $A$ , but only for the specified values of  $k, b$ .*

We next present the precise definition for a PRF. In this definition, the adversary  $ADV$  has *oracle access* to one of two functions: a random function from domain  $D$  to range  $R$ , i.e.,  $f \xleftarrow{\$} \{D \leftarrow R\}$ , or the PRF keyed with a

random  $n$ -bit key,  $F_k$  with  $k$  being a random  $n$  bit string, i.e.,  $k \xleftarrow{\$} \{0, 1\}^n$ . We denote these two cases by  $ADV^f$  and  $ADV^{F_k}$ , respectively. The idea of the definition is illustrated in 2.15.

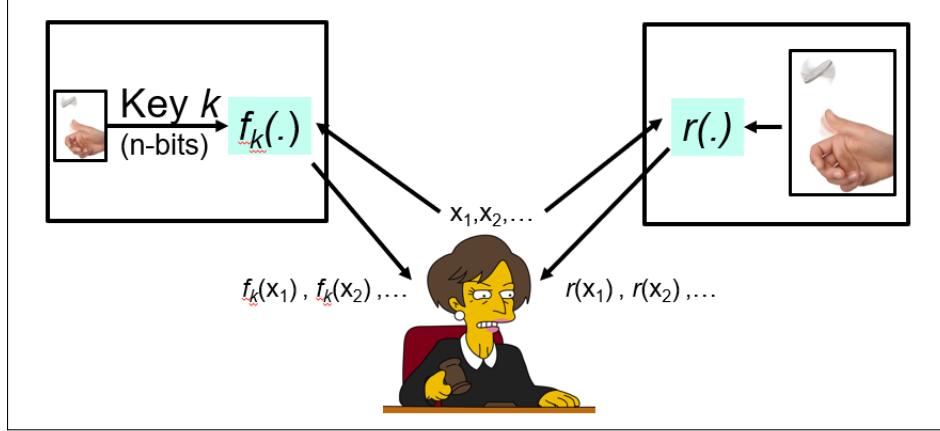


Figure 2.15: The Pseudo-Random Function (PRF) Indistinguishability Test. We say that function  $f_k(x) : \{0, 1\}^K \times \{0, 1\}^n \rightarrow \{0, 1\}^m$  is a (secure) pseudo-random generator (PRG), if a distinguisher can't efficiently distinguish between  $f_k(\cdot)$  and a random function  $r$  from  $\{0, 1\}^n$  to  $\{0, 1\}^m$ .

**Definition 2.6.** A pseudo-random function (PRF) is a polynomial-time computable function  $F : \{0, 1\}^* \times D \rightarrow R$  s.t. for all PPT algorithms  $ADV$  holds:

$$\Pr [ADV^{F_k}(1^n) = 'Rand'] - \Pr [ADV^f(1^n) = 'Rand'] \in NEGL(n)$$

Where the probabilities are taken over random coin tosses of  $ADV$ ,  $k \xleftarrow{\$} \{0, 1\}^n$ , and  $f \xleftarrow{\$} \{D \rightarrow R\}$ .

The basic idea of this definition is similar to the definition of a secure PRG (definition 2.4). Namely, a PRF ( $F_k$ ) is secure if every PPT algorithm  $ADV$  cannot have significant probability of distinguishing between  $F_k$  and a random function  $f$  (over same domain  $D$  and range  $R$ ), where  $n$  is the security parameter.

The  $ADV^{F_k}$  and  $ADV^f$  notations mean that  $ADV$  is given ‘oracle’ access to the respective function ( $F_k()$  or  $f()$ , respectively). Oracle access means that adversary can give any input  $x$  and get back that function applied to  $x$ , i.e.,  $F_k(x)$  or  $f(x)$ , respectively; other terms for this are ‘black box access’ or ‘subroutine’.

See Table 2.2 for a summary and comparison of random function/permuation, PRG, PRF and PRP (defined later).

Function	Key	Input	Property
Random function $r(x)$	None	Any	For each input $x$ in domain, let $r(x)$ be a random value from range
Random permutation $\pi(x)$	None	Any	For each input $x$ in domain, let $\pi(x)$ be a random value from domain, not assigned as $\pi(y)$ for any $y \neq x$ .
PRG $f(x)$	None	Secret, random	Output indistinguishable from random string of same length
PRF $f_k(x)$	Secret, random	Any	Can't distinguish btw $f_k(x)$ and a random function $r(x)$ (same domain, range)
PRP $f_k(x)$	Secret, random	Any	Can't distinguish btw $f_k(x)$ and a random permutation $\pi(x)$ (same domain)

Table 2.2: Comparison between random function, random permutation, PRG, PRF, and PRP

### 2.5.5 PRF: Constructions and Robust Combiners

The concept of PRFs was proposed in a seminal paper by Goldreich, Goldwasser and Micali [59]; the paper also presents a provably-secure construction of PRF, given a PRG. That is, if there is a successful attack on the constructed PRF, this attack can be used as a ‘subroutine’ to construct a successful attack on the underlying PRG. However, the construction requires many applications of the PRG for a single application of the PRF. Therefore, this construction is not applied in practice.

Instead, practical designs use simpler and more efficient constructions of PRFs. In particular, there are efficient constructions of PRFs from other cryptographic schemes (‘building blocks’), in particular, from *block ciphers* and from *cryptographic hash functions*. We will discuss these constructions after presenting the corresponding schemes. There are also provably-secure constructions of PRFs from very cryptographic schemes, such a *one-way functions*; these are more of theoretical interest, see [57].

Another option is to construct candidate pseudo-random functions *directly*, without assuming and using any other ‘secure’ cryptographic function, basing their security on failure to ‘break’ them using known techniques and efforts by expert cryptanalysts. In fact, pseudo-random functions are among the cryptographic functions that seem good candidate for such ‘ad-hoc’ constructions; it is relatively easy to come up with a reasonable candidate PRF, which will not be trivial to attack. See Exercise 2.33.

Finally, it is not difficult to *combine* two *candidate PRFs*  $F', F''$ , over the same domain and range, into a combined PRF  $F$  which is secure as long as either  $F'$  or  $F''$  is a secure PRF. We refer to such construction as a *robust combiner*. Robust combiners constructions are known for many cryptographic primitives. The following lemma, from [65], presents a trivial yet efficient robust combiner for PRFs, and also allows us to give a simple example of the

typical cryptographic proof of security based on reduction to the security of underlying modules.

**Lemma 2.2** (Robust combiner for PRFs). *Let  $F', F'' : \{0,1\}^* \times D \rightarrow R$  be two polynomial-time computable functions, and let:*

$$F_{(k',k'')}(x) \equiv F'_{k'}(x) \oplus F''_{k''}(x) \quad (2.11)$$

*If either  $F'$  or  $F''$  is a PRF, then  $F$  is a PRF. Namely, this construction is a robust combiner for PRFs.*

*Proof.* WLOG assume  $F'$  is a PRF, and we show that  $F$  is also a PRF. Suppose to the contrary, i.e., exists some adversary  $ADV$  that can efficiently distinguish between  $F_{k',k''}(\cdot)$  and a random function  $f \xleftarrow{\$} \{D \rightarrow R\}$ .

We use  $ADV$  as a subroutine in the design of  $ADV'$ , that will distinguish between  $F'_{k'}(\cdot)$  and a random function  $f' \xleftarrow{\$} \{D \rightarrow R\}$ , in contradiction to the assumption that  $F'$  is a PRF.

$ADV'$  first generates a random key  $k''$  (for  $ADV''$ ). It then runs  $ADV$ . Whenever  $ADV$  asks to compute  $F_{k',k''}(x)$ , then  $ADV'$  uses the oracle to receive  $y' = F'_{k'}(x)$ , uses  $k''$  to compute  $y'' = F''_{k''}(x)$  and returns  $y' \oplus y''$ , which is obviously the value of  $F_{k',k''}(x)$ . When  $ADV$  concludes with a guess of ‘R’ or otherwise, then  $ADV'$  returns the same guess, succeeding whenever  $ADV$  succeeds.  $\square$

Considering that it is quite easy to design candidate PRFs (see Exercise 2.33), and that it is very easy to robustly-combine candidate PRFs (Lemma 2.2), it follows that PRFs are a good basis for more complex cryptographic schemes. In particular, later in this section we show how to use a PRF to construct secure encryption.

### 2.5.6 The key-separation principle and application of PRF

In the PRF robust combiner (Eq. 2.11), we used *separate* keys for the two candidate-PRF functions  $F', F''$ . In fact, this is *necessary*, as the following exercise shows.

**Exercise 2.12** (Independent keys are required for PRF robust combiner). *Let  $F', F'' : \{0,1\}^* \times D \rightarrow R$  be two polynomial-time computable functions, and let  $F_k(x) = F'_k(x) \oplus F''_k(x)$  and  $G_{(k',k'')}(x) = F'_{k'}(F''_{k''}(x))$ . Demonstrate that the fact that one of  $F', F''$  is a PRF, may not suffice to ensure that  $F$  and  $G$  would be PRFs.*

This is an example for the general *key-separation principle* below. In fact, the study of robust combiners often helps to better understand the properties of cryptographic schemes and to learn how to do cryptographic proofs.

**Principle 5** (Key-separation). *Use separate, independently-pseudorandom keys for each different cryptographic scheme, as well as for different types/sources of plaintext and different periods.*

The principle combines three main motivations to the use of separate, independently-pseudorandom keys:

**Per-goal keys:** separate keys for different cryptographic schemes. A system may use multiple different cryptographic functions or schemes, often for different goals, e.g., encryption vs. authentication. In this case, security may fail if the same or related keys are used for multiple different functions. Exercise 2.12 above is an example.

**Limit information for cryptanalysis.** By using separate, independently-pseudorandom keys, we reduce the amount of information available to the attacker (ciphertext, for example).

**Limit the impact of key exposure.** Namely, ensure that exposure of some of the keys, will not jeopardize the secrecy of communication encrypted with the other keys.

Pseudo-random Functions (PRFs) have many applications in cryptography, including encryption, authentication, key management and more. One important application is *derivation of multiple separate keys* from a single shared secret key  $k$ . Namely, a PRF, say  $f$ , is handy whenever two parties share one secret key  $k$ , and need multiple *separate, independently pseudorandom* keys from  $k$ . This is achieved by applying the PRF  $f$  with the key  $k$ , to any separate inputs, one for each separate derived key; for example, you can use  $k_i = f_k(i)$  to derive separate, independently pseudorandom keys  $k_i$  for goals  $i = 1, 2, \dots$

As another example, system designers often want to limit the impact of key exposure, due to cryptanalysis or to system attacks. One way to reduce the damage from key exposures is to change the keys periodically, e.g., use key  $k_d$  for day number  $d$ :

**Example 2.2** (Using PRF for independent per-period keys.). *Assume that Alice and Bob share one master key  $k_M$ . They may derive a shared secret key for day  $d$  as  $k_d = \text{PRF}_{k_M}(d)$ . Even if all the daily keys are exposed, except of one day  $\hat{d}$ , the key for  $\hat{d}$  remains secure as long as  $k_M$  is kept secret.*

## 2.6 Defining secure encryption

So far, we presented two definitions of security: for PRG and for PRF (defs. 2.4 and 2.6, respectively). However, what about the encryption schemes? This is the subject of this section.

We dedicate a section to the definition of secure encryption, since this is a quite tricky issue. In fact, people have been designing - and attacking - cryptosystems for millennia, without a precise definition of their security! This only changed with the seminal paper of Goldwasser and Micali [60] which defined secure encryption and presented a provably-secure design; this paper is one of cornerstones of the theory of modern cryptography. The following exercise may help you understand some of the difficulties involved - esp. as you later read the rest of this section.

Attack type	Cryptanalyst capabilities
Ciphertext Only (CTO)	Plaintext distribution or predicate
Known Plaintext Attack (KPA)	Set of (ciphertext, plaintext) pairs
Chosen Plaintext Attack (CPA)	Attacker Ciphertext for arbitrary plaintext chosen by attacker
Chosen Ciphertext Attack (CCA)	Plaintext for arbitrary ciphertext chosen by attacker

Table 2.3: Types of Cryptanalysis Attacks. In *all* attack types, the cryptanalyst knows the cipher design and a body of ciphertext.

**Exercise 2.13** (Defining secure encryption). *Define secure symmetric encryption, as illustrated in Figure 2.1. Refer separately to the two aspects of security definitions: (1) the attacker model, i.e., the capabilities of the attacker, and (2) the success criteria, i.e., what constitutes a successful attack and what constitutes a secure encryption scheme.*

### 2.6.1 Attacker model

The first aspect of a security definition, is a precise *attacker model*, defining the maximal expected capabilities of the attacker. We discussed already some of these capabilities. In particular, we already discussed the *computational limitations* of the attacker: in § 2.4 we discussed the *unconditional security* model, where attackers have unbounded computational resources, and from subsection 2.5.2 we focus on *Probabilistic Polynomial Time (PPT)* adversaries, whose computation time is bounded by some polynomial in their input size.

Attacker capabilities include also their possible interactions with the attacked scheme and the environment; in the case of encryption schemes, we refer to these are *types of cryptanalysis attack*. We mentioned above some of these, mainly cipher-text only (CTO) and known-plaintext attack (KPA).

Two other attack types are the *chosen plaintext attack (CPA)* and the *chosen ciphertext attack (CCA)*. In a *chosen-plaintext attack*, the adversary can choose plaintext and receive the corresponding ciphertext (encryption of that plaintext). In *chosen-ciphertext attack*, the adversary can chose *ciphertext* and receive the corresponding plaintext (its decryption), or error message if the ciphertext does not correspond to well-encrypted plaintext.

We summarize these four basic types of cryptanalysis in Table 2.3. You will find more refined variants of these basic attack, as well as additional types of cryptanalysis attacks, in cryptography courses and books, e.g., [58, 106].

It is desirable to allow for attackers with maximal capabilities. Therefore, when we evaluate cryptosystems, we are interested in their resistance to all types of attacks, and esp. the stronger ones - CCA and CPA. On the other hand, when we design systems using a cipher, we try to limit that attacker's capabilities; e.g., a standard method to foil CCA attacks is to ensure that all plaintext messages contain sufficient redundancy, to make it infeasible for

the attacker to generate validly-decrypting ciphertext, even given other valid ciphertexts.

### 2.6.2 Success criteria

The other aspect of a security definition of a scheme or a system is the *success criteria*, i.e., precise definition of what is a successful attack - and what constitutes a secure scheme or system. Note that we used general terms, which can be applied to *any* secure system and not limited to cryptographic schemes; indeed, the precise definition of security, including both attacker model and success criteria for attack and scheme/system, is fundamental to security, and not just to cryptography.

Intuitively, the security goal of encryption is *confidentiality*: to transform plaintext into ciphertext, in such way that will allow specific parties ('recipients') - and only them - to perform decryption, transforming the ciphertext back to the original plaintext. However, the goal as stated may be interpreted to only forbid recovery of the exact, complete plaintext; but what about recovery of partial plaintext?

For example, suppose an eavesdropper can decipher half of the characters from the plaintext - is this secure? We believe most readers would not agree. What if she can decipher less, say one character? In some applications, this may be acceptable; in others, even exposure of one character may have significant consequences.

Intuitively, we require that an adversary cannot learn *anything* given the ciphertext. This may be viewed as extreme; for example, in many applications the plaintext includes known fields, and their exposure may not be a concern. However, it is best to minimize assumptions and use definitions and schemes which are secure for a wide range of applications.

Indeed, in general, when we design a security system, cryptographic or otherwise, it is important to clearly define both aspects of security: the attacker model (e.g., types of attacks 'allow' and any computational limitations), as well as the success criteria (e.g., ability to get merchandise without paying for it). Furthermore, it is difficult to predict the actual environment ways in which a system would be used; this motivates the *conservative design principle*, as follows.

**Principle 6** (Conservative design). *Cyber-security mechanisms, and in particular cryptographic schemes, should be specified and designed with minimal assumptions and for maximal range of applications and for maximal, well-defined attacker capabilities. On the other hand, systems should be designed to minimize the attackers' abilities, in particular, limiting the attacker's ability to attack the cryptographic schemes in use.*

Okay, so hopefully, you now agree that it would be best to require that an adversary cannot learn *anything* from the ciphertext. But how do we even define this? This is not so easy, which may explain the importance of that seminal paper by Goldwasser and Micali [60], which present a precise definition for this notion, which they refer to as *semantic security*. We will present, however,

a different notion: *indistinguishability*, which is a bit easier to understand and use - and was shown in [60] to be equivalent to semantic security.

Intuitively, an encryption scheme ensures indistinguishability, if an attacker cannot distinguish between encryption of any two given messages. But, again, turning this into a ‘correct’ and precise definition, requires care.

The concept of indistinguishability is reminiscent of disguises; it may help to consider the properties we can open to find in an ‘ideal disguise service’:

**Any two disguised persons are indistinguishable:** cannot distinguish between *any* two persons, when in disguised. Yes, even beautiful Rachel from ugly Leah!

**Except, the two persons should have the ‘same size’:** assuming that a disguise is of ‘reasonable size’ (overhead), a giant can’t be disguised to be indistinguishable from a dwarf!

**Re-disguises should be different:** if we see Rachel in disguise, and then she disappears and we see a new disguise, we should not be able to tell if it is Rachel again, in new disguise - or any other disguised person! This means that disguises must be randomized or stateful, i.e., every two disguises of same person (Rachel) will be different.

We will present corresponding properties for indistinguishable encryption:

**Encryptions of any two messages are indistinguishable.** To allow arbitrary applications, we allow the attacker to choose the two messages. However, there is one restriction: *the two messages should be same length*.

**Re-encryptions should be different:** attacker should not be able to distinguish encryptions, based on previous encryptions of the same messages. This means that encryption must be randomized or stateful, so that two encryptions of same message will be different. (A weaker notion of ‘deterministic encryption’ allows detection of re-encryption of a message, and is sometimes used for scenarios where state and randomization are to be avoided.)

We are finally ready to formally present the indistinguishability-based definition of secure encryption. For simplicity, we first present a definition for stateless encryption (Definition 2.1), and only for chosen-plaintext (CPA) attacks.

**Definition 2.7** (IND-CPA for stateless encryption). *A stateless shared-key cryptosystem  $\langle E, D \rangle$  is CPA-indistinguishable (IND-CPA), if for every PPT adversary  $ADV$  holds:*

$$\Pr[b = IndCPA_{ADV, \langle E, D \rangle}(b, k, n)] \leq \frac{1}{2} + NEGL(n)$$

Where the probability is over coin tosses of  $ADV$  and  $E$ , the random choices  $b \xleftarrow{\$} \{0, 1\}$ ,  $k \xleftarrow{\$} \{0, 1\}^n$ , and where  $IndCPA_{ADV, \langle E, D \rangle}(b, k, n)$  is:

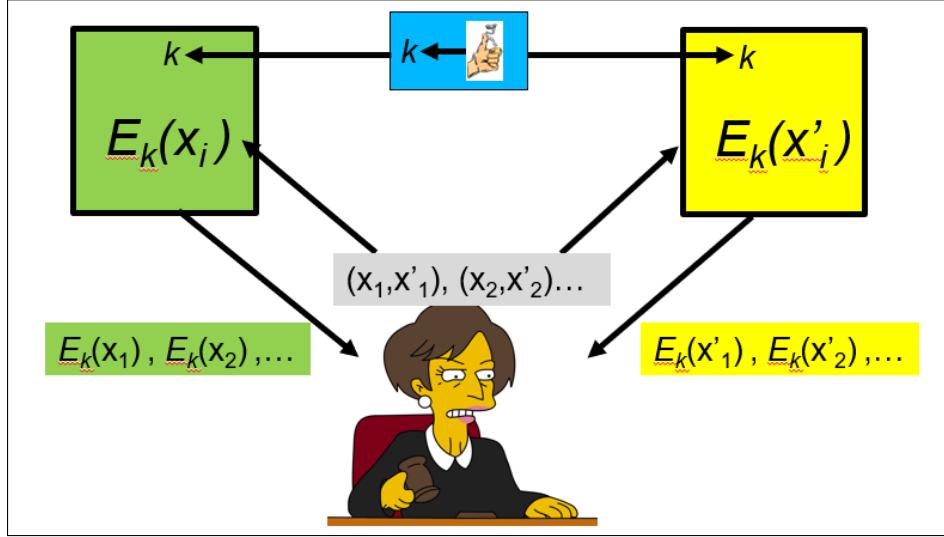


Figure 2.16: The CPA-Indistinguishable Encryption Test. We use two encryption boxes, both given the same randomly chosen key  $k$ : a green box and a yellow box, placed randomly as shown (with green on left) or reverse (with yellow on left). The distinguisher can choose input pairs  $(x_i, x'_i)$  given to both boxes; the inputs must be of same length, i.e.,  $|x_i| = |x'_i|$ . The green box outputs the encryption of  $x_i$ , i.e.,  $E_k(x_i)$ , and the yellow box outputs  $E_k(x'_i)$ . The distinguisher should determine where is the green box; the encryption is considered CPA-Indistinguishable, if the distinguisher cannot efficiently determine correctly, significantly better than a random guess (with 50% chance).

```
{
   $(m_0, m_1, s_{ADV}) \leftarrow ADV^{E_k(\cdot)}(\text{`Choose'}, 1^n)$  s.t.  $|m_0| = |m_1|$ 
   $c^* \leftarrow E_k(m_b)$ 
   $b^* = ADV^{E_k(\cdot)}(\text{`Guess'}, s_{ADV}, c^*)$ 
  Return  $b^*$ 
}
```

The definition allows the adversary to choose the two challenge messages, between whose decryption it should distinguish, to ensure that security is sufficient for any application. For extra safety, we allow the adversary to see encryptions (ciphertexts) of chosen plaintexts, already in this initial ‘choose challenges’ phase. We also allow the adversary to pass arbitrary state  $s$  from this ‘choose’ phase to the ‘guess’ phase where it should guess the value of  $b$ , i.e., if  $m^*$  is encryption of  $m_0$  or of  $m_1$ .

Note that in  $ADV^{E_k(\cdot)}$ , we use the oracle notation introduced before Def. 2.6. Namely,  $ADV^{E_k(\cdot)}$  denotes calling the  $ADV$  algorithm, with ‘oracle access’ to the (keyed) PPT algorithm  $E_k(\cdot)$ , i.e.,  $ADV$  can provide arbitrary plaintext string  $m$  and receive  $E_k(m)$ .

We next extend the definition to cover also stateful encryption (Definition 2.2). We only need to change the  $IndCPA$  experiment, and the changes are minor. First, obviously, whenever the encryption algorithm is invoked, by the adversary or directly by  $IndCPA$ , the state is provided as additional input, and the output also contains the new state. The adversary is allowed to completely control the state in the input, and observe it in the output; we use  $s^*$  to denote the state specified by the adversary to be used by  $IndCPA$  to compute  $c^*$ . Second, before returning  $b^*$ , the adversary's 'guess' for the value of the bit, the  $IndCPA$  experiment validates that the adversary never asked for encryption with the same state  $s^*$ .

**Definition 2.8** (IND-CPA for stateful encryption). *A stateful shared-key cryptosystem  $\langle E, D \rangle$  is CPA-indistinguishable (IND-CPA), under same conditions as for stateless cryptosystem (Definition 2.7), and where  $IndCPA_{ADV, \langle E, D \rangle}(b, k, n)$  is modified to:*

```
{
   $(m_0, m_1, s_{ADV}, s^*) \leftarrow ADV^{E_k(\cdot, \cdot)}(\text{Choose}, 1^n)$  s.t.  $|m_0| = |m_1|$ 
   $c^* \leftarrow E_k(m_b, s^*)$ 
  Let  $b^* = ADV^{E_k(\cdot, \cdot)}(\text{Guess}, s_{ADV}, c^*)$ 
  If  $ADV$  'cheated', i.e., made an encryption query with state  $s^*$ , then return  $\perp$ 
  Else, return  $b^*$ 
}
```

Finally, we notice that encryption must either be randomized or stateful - or both - in order to ensure indistinguishability. The reason is simple: the adversary is allowed to make queries for arbitrary messages - including the 'challenges'  $m_0, m_1$ . If the encryption scheme is deterministic - and stateless - then all encryptions of a message, e.g.  $m_0$ , will return a fixed ciphertext; this will allow the attacker to trivially 'win' in the IND-CPA experiment.

**Defining indistinguishability for CTO, KPA and CCA attack models**  
 Definition 2.8 focuses on Chosen-Plaintext Attack (CPA) model. Modifying this definition for the case of chosen-ciphertext (CCA) attacks requires a rather simple change and extension, and hence left as an exercise.

**Exercise 2.14.** Define a chosen ciphertext indistinguishable (IND-CCA) encryption scheme, as a modification of Definition 2.8.

However, modifying the definition for Cipher-Text-Only (CTO) attack and Known-Plaintext Attack (KPA) is more challenging. For KPA, the obvious question is *which* plaintext-ciphertext messages are known; this may be solved by using random plaintext messages, however, in reality, the known-plaintext is often quite specific.

It is similarly challenging to modify the definition so it covers CTO attacks, where the attacker must know *some* information about the plaintext distribution. This information may be related to the specific application, e.g., when

the plaintext is English. In other cases, information about the plaintext distribution may be derived from system design, e.g., text is often encoded using ASCII, where one of the bits in every character is the parity of the other bits. An even more extreme example is in GSM, where the plaintext is the result of application of Error-Correcting Code (ECC), providing significant redundancy which even allows a CTO attack on GSM's A5/1 and A5/2 ciphers [6]. In such case, the amount of redundancy in the plaintext can be compared to that provided by a KPA attack. We consider it a CTO attack, as long as the attack does not require knowledge of all or much of the plaintext corresponding to the given ciphertext messages.

Some systems, including GSM, allow the attacker to guess all or much of the plaintext for some of the ciphertext messages, e.g., when sending a predictable message at specific time. Such systems violate the Conservative Design Principle (principle 6), since a KPA-vulnerability of the cipher renders the system vulnerable. A better system design would limit the adversary's knowledge about the distribution of plaintexts, requiring a CTO vulnerability to attack the system.

## 2.7 The Cryptographic Building Blocks Principle

We next discuss the *design of secure symmetric encryption schemes*. Ideally, we would use schemes which are proven to be secure, e.g., IND-CPA (Definition 2.7), without any assumptions such as about security or computational-hardness properties of functions. However, IND-CPA implies that there is no efficient (PPT) algorithm, that can distinguish between encryption of two messages. Of course, this refers to algorithms for distinguishing, which are only given the ciphertext; efficient distinguishing is easy, given the decryption key. In terms of the theory of computational complexity, this implies that distinguishing cannot be done efficiently (i.e., in polynomial time) - unless given a specific 'hint' (the decryption key). This implies that the complexity class  $P$ , which contains all problems which can be solved efficiently, is *strictly smaller* than the class  $NP$ , of problems which can be solved efficiently given some string ('hint'). Namely, this would imply that  $P \neq NP$  - which will be a solution to the most fundamental question in the theory of computational complexity.

It is not practical, to require the encryption algorithm to have a property, whose existence implies a solution to such a basic, well-studied open question. Therefore, both theoretical and applied cryptography consider designs whose security relies on failed attempts in cryptanalysis. The big question is: should we rely on failed cryptanalysis of the scheme itself, or on failed cryptanalysis of underlying components of the scheme?

It may seem that the importance of encryption schemes should motivate the first approach, i.e., relying of failed attempts to cryptanalyze the scheme. Surely this was the approach in ancient and 'classical' cryptology.

However, in modern applied cryptography, it is much more common to use the second approach, i.e., to construct encryption using 'simpler' underlying

primitives, and to base the security of the cryptosystem on the security of these component modules. We summarize this approach in the following principle, and then give some justifications.

**Principle 7** (Cryptographic Building Blocks). *The security of cryptographic systems should only depend on the security of few basic building blocks. These blocks should be simple and with well-defined and easy to test security properties. More complex schemes should be proven secure by reduction to the security of the underlying blocks.*

The advantages of following the cryptographic building blocks principle include:

**Efficient cryptanalysis:** by focusing cryptanalysis effort on few schemes, we obtain much better validation of their security. The fact that the building blocks are simple, and are selected to be easy to test, make cryptanalysis even more effective.

**Replacement and upgrade:** by using simple, well-defined modules, we can replace them for improved efficiency - or to improve security, in particular after being broken or when doubts arise.

**Flexibility and variations:** complex systems and schemes naturally involve many options, tradeoffs and variants; it is better to build all such variants using the same basic building block.

**Robust combiners:** there are known, efficient robust-combiners designs for the basic cryptographic building blocks [65]. If desired, use these as the basic blocks for improved security.

The cryptographic building blocks principle is key to both applied and theoretical modern cryptography. From the theoretical perspective, it is important to understand which schemes can be implemented, given another scheme. There are many results exploring such relationships between different cryptographic schemes and functions, with many positive results (constructions), few negative results (proofs that efficient constructions are impossible or improbable), and very few challenging open questions.

In modern applied cryptography, the principle implies the need to define a very small number of basic building blocks, which would be very efficient, simple functions - and convenient for many applications. In fact, most of applied cryptography is based on four such primitives: the shared-key *block cipher*, which we already discuss next; the keyless *cryptographic hash functions* (chapter 4); and the public encryption and signature schemes (chapter 6). Cryptographic hash functions and block ciphers are much more efficient than the public key schemes (see Table 6.1), hence, preferred, and used in most practical systems - when public-key operations may be avoided.

## 2.8 Block Ciphers and Pseudo-Random Permutations (PRPs)

Modern symmetric encryption schemes are built in modular fashion, using a basic building block - the *block cipher*. A block cipher is defined by a pair of keyed functions,  $E_k, D_k$ , such that the domain and the range of both  $E_k$  and  $D_k$  are  $\{0, 1\}^n$ , i.e., binary strings of fixed length  $n$ ; for simplicity, we (mostly) use  $n$  for the length of both keys and blocks, as well as the security parameter, although in some ciphers, these are different numbers.

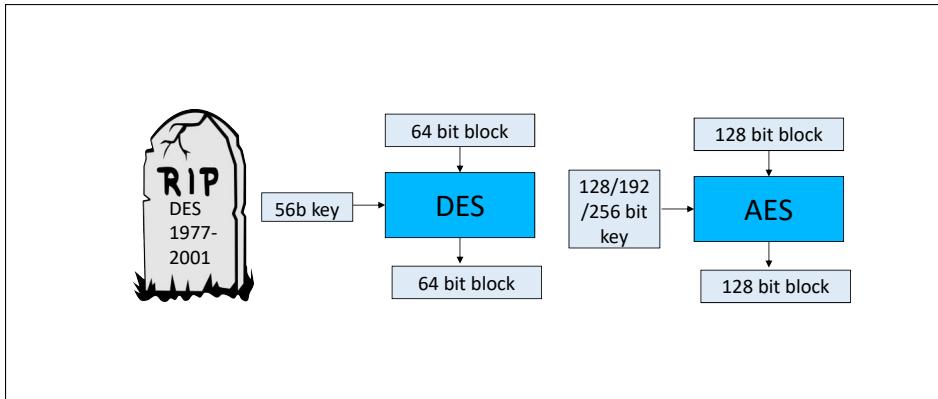


Figure 2.17: High-Level view of the NIST standard block ciphers: AES (current) and DES (obsolete).

Block ciphers may be the most important basic cryptographic building blocks. Block ciphers are in wide use in many practical systems and constructions, and two of them were standardized by NIST - the DES (1977-2001) [105], the first standardized cryptographic scheme, and the AES (2002-????) [35], its successor. See illustration of the NIST standard block ciphers in Fig. 2.17.

Unfortunately, there is no universally-accepted definition of the exact cryptographic requirements from block ciphers. Hence, their required security properties are not very well defined, which makes it hard to decide if a given cipher is ‘secure’. We will adopt here the view of a secure block cipher as a reversible Pseudo-Random Permutations (PRPs). We next discuss the definition of a PRP and a reversible PRP (block cipher).

[AH: add here discussion of permutation and random permutation.]

A *Pseudo-Random Permutations (PRP)*  $PRP_k(\cdot)$  is an efficient algorithm, similar to a PRF, except that for any given key  $k$ , the outcome  $PRP_k(\cdot)$  is a permutation, indistinguishable from a random permutation over  $\{0, 1\}^n$ . A *block cipher*, also called a *reversible PRP*, is a pair of PRPs  $E_k, D_k$  over  $\{0, 1\}^n$ , s.t. for every message  $m$  in the domain, holds  $m = D_k(E_k(m))$ . Namely, a block cipher is indistinguishable from a random pair of a permutation and its inverse permutation. The fact that  $m = D_k(E_k(m))$  for every  $k, m \in \{0, 1\}^n$

is called the *correctness property* - note that this is essentially the same as presented in Def. 2.1 for encryption schemes.

**Exercise 2.15.** *Precisely define a PRP and a reversible PRP (block cipher), following Def. 2.6.*

Note that block ciphers, in particular DES and AES, are often referred to as encryption schemes, although they do not satisfy the requirements of most definitions of encryption, e.g., the IND-CPA test of Def. 2.8.

**Exercise 2.16.** *Explain why PRPs and reversible PRPs (block ciphers) fail the IND-CPA test (Def. 2.8).*

On the other hand, we will soon see multiple constructions of secure encryption based on block ciphers; these constructions are often referred to as *modes of operation*. Indeed, block ciphers are widely used as cryptographic building blocks, as they satisfy much of the requirements of the Cryptographic Building Blocks principle. They are simple, deterministic functions with Fixed Input Length (FIL), which is furthermore identical to their output length. This should be contrasted with ‘full fledged encryption schemes’, which are randomized (or stateful) and have Variable Input Length (VIL). Indeed, block ciphers are even easier to evaluate than PRFs, since PRFs may have different input and output lengths.

Another desirable property of block ciphers is that they have a simple *robust combiner*, i.e., a method to combine two or more candidate block ciphers, into one ‘combined’ function which is a secure block cipher provided one or more of the candidates is a secure block cipher. This is shown in [65] and the following exercise.

**Exercise 2.17.** *Design a robust combiner for PRPs and for reversible PRPs (block ciphers).*

*Hint:* cf. Lemma 2.2, and/or read [65]. □

### 2.8.1 Constructing PRP from PRF: the Feistel Construction

In Ex. 2.33 we show that it is not too difficult to construct a reasonable candidate PRF. However, constructing candidate PRP seems harder. This motivates constructing a PRP, given a PRF.

Let us first consider a simple yet useful case: a PRF whose range is identical to its domain  $D$  - as is the case for a PRP or reversible PRP (block cipher). An obvious difference between such PRF and a PRP, is that, for a given key  $k$ , a PRF  $F_k$  is likely to have some ‘collisions’  $F_k(x) = F_k(y)$ , while a PRP or block cipher never have such collisions; namely, the PRF is not a PRP. However, as the following Lemma shows, a PRP over a domain  $D$  is also a PRF over (domain and range)  $D$ ; and, on the other hand, no computationally-bounded (PPT) adversary can distinguish between a PRP and a PRF (over domain  $D$ ).

**Lemma 2.3.** Every PRP over domain  $D$  is also a PRF (with domain and range  $D$ ), and every PRF over domain  $D$  and range  $D$ , is indistinguishable from a PRP over  $D$ .

*Proof:* In a polynomial set of queries of a random function, there is negligible probability of having two values which will map to the same value. Hence, it is impossible to efficiently distinguish between a random function and a random permutation. The proof follows since a PRF (PRP) is indistinguishable from a random function (resp., permutation).  $\square$

Constructing a PRP from a PRF is not trivial; see the following two exercises.

**Exercise 2.18.** Let  $f$  be a PRF from  $n$ -bit strings to  $n$ -bit strings. Show that  $g_{k_L, k_R}(m_L || m_R) = f_{k_L}(m_L) || f_{k_R}(m_R)$  is not a PRP (over  $2n$ -bit strings).

*Hint:* given a black box containing  $g$  or a random permutation over  $2n$ -bit strings, design a distinguishing adversary  $D$  as follows.  $D$  makes two queries, one with input  $0^{2n}$  and the other with input  $0^n || 1^n$ . From the output,  $D$  can easily infer if the box contained  $g$  or a random permutation.  $\square$

The next exercise presents a slightly more elaborate scheme, which is essentially a reduced version of the Feistel construction (presented next).

**Exercise 2.19.** Let  $f$  be a PRF from  $n$ -bit strings to  $n$ -bit strings. Show that  $g_{k_L, k_R}(m_L || m_R) = m_L \oplus f_{k_R}(m_R) || m_R \oplus f_{k_L}(m_L)$  is not a PRP (over  $2n$ -bit strings).

We next present the Feistel construction, the most well known and simplest construction of PRP - in fact, a reversible PRP (block cipher) - from PRF. As shown in Fig. 2.18, the Feistel cipher transforms an  $n$ -bit PRF into a  $2n$ -bit (invertible) PRP.

Formally, given a function  $y = f_k(x)$  with  $n$ -bit keys inputs and outputs, the three-rounds Feistel is  $g_k(m)$  defined as:

$$\begin{aligned} L_k(m) &= m_{0,\dots,n-1} \oplus F_k(m_{n,\dots,2n-1}) \\ R_k(m) &= F_k(L_k(m)) \oplus m_{n,\dots,2n-1} \\ g_k(m) &= L_k(m) \oplus F_k(R_k(m)) || R_k(m) \end{aligned}$$

Note that we consider only a ‘three rounds’ Feistel cipher, and use the same underlying function  $F_k$  in all three rounds, but neither aspect is mandatory. In fact, the Feistel cipher is used in the design of DES and several other block ciphers, typically using more rounds (e.g., 16 in DES), and often using different functions at different rounds.

Luby and Rackoff [83] proved that by a Feistel cipher of three or more ‘rounds’, using a PRF as  $F_k(\cdot)$ , results in a reversible PRP, i.e., in a block cipher.

One may ask, why use the Feistel design rather than directly design a reversible PRP? Indeed, this is done in AES, which does not follow the Feistel

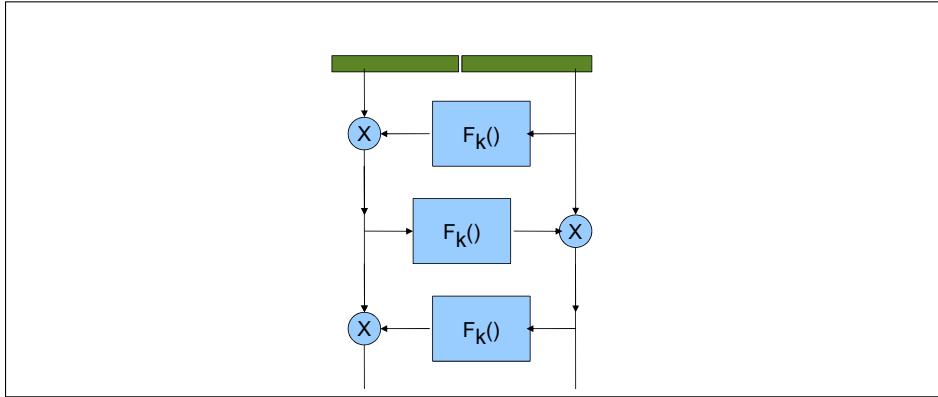


Figure 2.18: Three ‘rounds’ of the Feistel Cipher, constructing a block cipher (reversible PRP) from a PRF  $F_k(\cdot)$ . The Feistel cipher is used in DES (but not in AES). Note: this figure should be improved; in particular it uses the notation  $\otimes$  for bitwise  $XOR$ , instead of  $\oplus$ ; and add  $L_k(m), R_k(m)$ .

cipher design. An advantage of using the Feistel design is that it allows the designer to focus on the pseudo-randomness requirements when designing the PRF, without having simultaneously to make sure that the design is also an invertible permutation. Try to design a PRP, let alone an reversible PRP, and compare it to using the Feistel cipher!

## 2.9 Secure Encryption Modes of Operation

Finally we get to design symmetric encryption schemes; following the Cryptographic Building Blocks principle, the designs are based on a the much simpler block ciphers.

The term *modes of operation* is used for constructions of more complex cryptographic mechanisms from block ciphers, for different purposes. See the DES specifications [105], which described ‘standard modes of operation for DES’, including four or the five modes we will discuss here, as summarized in Table 2.4.

**Exercise 2.20.** Table 2.4 specifies only the encryption process. Write the decryption process for each mode and show that correctness is satisfied, i.e.,  $m = D_k(E_k(m))$  for every  $k, m \in \{0, 1\}^*$ .

The ‘modes of operations’ in Table 2.4 are designed to turn block ciphers into more complete cryptosystems, handling aspects like:

**VIL:** allow encryption of arbitrary, variable input length messages.

**Randomization/state:** Most modes used for encryption, use randomness and/or state to ensure independence between two encryptions of the same

Mode	Encryption	Comments
Electronic code book (ECB)	$c_i = E_k(m_i)$	Insecure
Per-Block Counter (PBC)	$c_i = m_i \oplus E_k(i)$	Nonstandard, stateful
Per-Block Random (PBR)	$r_i \xleftarrow{\$} \{0,1\}^n, c_i = (r_i, m_i \oplus E_k(r_i))$	Nonstandard, stateless
Output Feedback (OFB)	$r_0 \xleftarrow{\$} \{0,1\}^n, r_i = E_k(r_{i-1}), c_0 \leftarrow r_0, c_i \leftarrow r_i \oplus m_i$	Stateless; fast online encryption
Cipher Feedback (CFB)	$c_0 \xleftarrow{\$} \{0,1\}^n, c_i \leftarrow m_i \oplus E_k(c_{i-1})$	Parallel
Cipher-Block Chaining (CBC)	$c_0 \xleftarrow{\$} \{0,1\}^n, c_i \leftarrow E_k(m_i \oplus c_{i-1})$	Integrity
Counter (CTR)		TBD

Table 2.4: Important Modes of Operation for Encryption using  $n$ -bit block cipher. The plaintext is given as a set of  $n$ -bit blocks  $m_1||m_2||\dots$ , where each block has  $n$  bits, i.e.,  $m_i \in \{0,1\}^n$ . Similarly the ciphertext is produced as a set of  $n$ -bits blocks  $c_0||c_1||\dots \in \{0,1\}^n$ , where  $c_i \in \{0,1\}^n$  (except ECB and PBC, which do not output  $c_0$ , and PBR, where ciphertext blocks are  $2b$ -bits long).

(or of related) messages, as required for indistinguishability-based security definitions.

**Key length:** Some modes are designed to allow the use of non-standard length of keys. *Longer keys* are used, e.g., in Triple-DES, to try to improve security - at least against exhaustive search. *Shorter keys* are used to create intentionally-weakened versions, e.g. to meet export regulations, or to make sure that the best way to attack the scheme is via exhaustive search.

There are several other important modes. Some are designed to ensure different goals, mainly efficiency; in particular, the simple *counter (CTR) mode* [41], which features random-access decryption. Some other modes ensure other properties besides or in addition to confidentiality. This includes the *CBC-MAC mode*, which provides authentication; and also the *Counter and CBC-MAC (CCM) mode* and the (more efficient) *Galois/Counter Mode (GCM) mode*, which ensure *authenticated encryption*, i.e., both confidentiality and encryption ‘at the same time’. We discuss authentication and authenticated-encryption in the next chapter.

### 2.9.1 The Electronic Code Book mode (ECB) mode

ECB is a naïve mode, which isn’t really a proper ‘mode’: it simply applies the block cipher separately to each block of the plaintext. Namely, to decrypt the plaintext string  $m = m_1||m_2||\dots$ , where each  $m_i$  is a block (i.e.,  $|m_i| = n$ ), we simply compute  $c_i = E_k(m_i)$ . Decryption is equally trivial:  $m_i = D_k(c_i)$ ,

and correctness of encryption, i.e.,  $m = D_k(E_k(m))$  for every  $k, m \in \{0, 1\}^*$ , follows immediately from the correctness of the block cipher  $E_k(\cdot)$ .

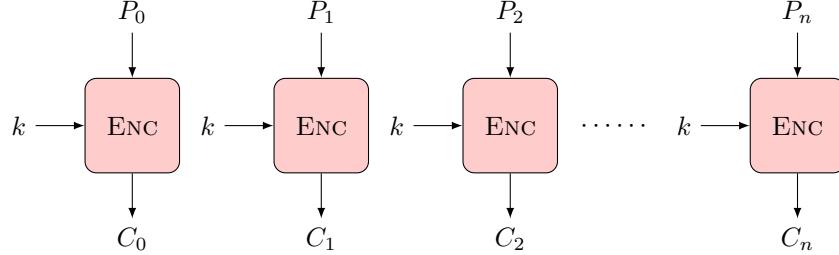


Figure 2.19: Electronic Code Book (ECB) mode encryption. Adapted from [68].

Note: notations are not exactly consistent with text, should be fixed.

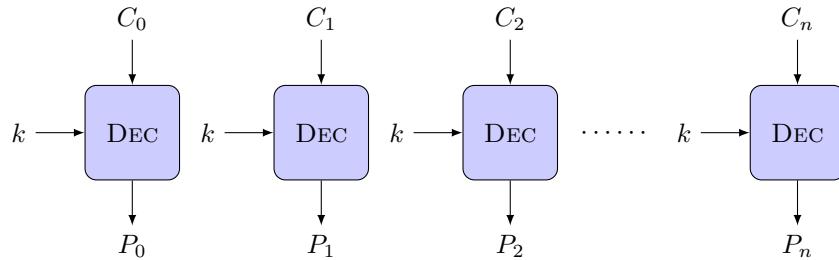


Figure 2.20: Electronic Code Book (ECB) mode decryption. Adapted from [68]. Note: notations are not exactly consistent with text, should be fixed.

The reader may have already noticed that ECB is simply a mono-alphabetic substitution cipher, as discussed in subsection 2.1.3. The ‘alphabet’ here is indeed large: each ‘letter’ is a whole  $n$ -bit block. For typical block ciphers, the block size is significant, e.g.,  $b_{DES} = 64$  bits for DES and  $b_{AES} = 128$  bits; this definitely improves security, and may make it challenging to decrypt ECB-mode messages in many scenarios.

However, obviously this means that ECB may expose some information about plaintext, in particular, all encryptions of the same plaintext block will result in the same ciphertext block. Even with relatively long blocks of 64 or 128 bits, such repeating blocks are quite likely in practical applications and scenarios, since *inputs are not random strings*. Essentially, this is a generalization of the letter-frequency attack of subsection 2.1.3 (see Fig. 2.7).

This weakness of ECB is often illustrated graphically by the example illustrated in Fig. 2.21, using the ‘Linux Penguin’ image [50, 115].

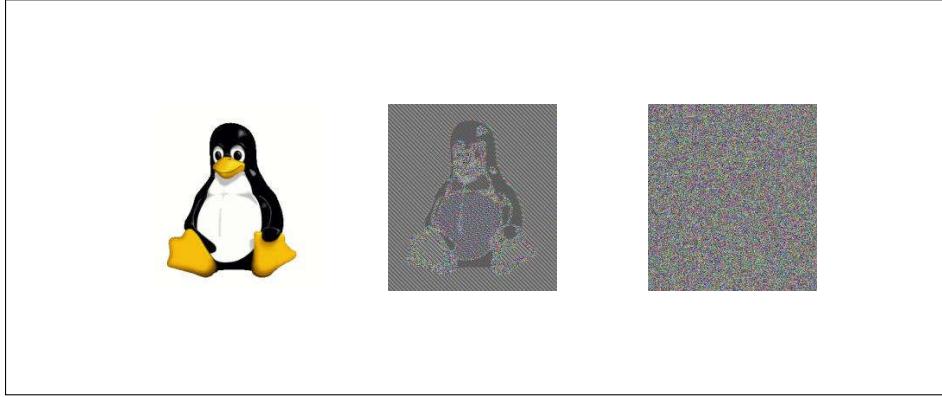


Figure 2.21: The classical visual demonstration of the weakness of the ECB mode. The middle and the right ‘boxes’ are encryptions of the bitmap image shown on the left; can you identify which of the two is ‘encrypted’ using ECB, and which is encrypted with one of the secure encryption modes?

### 2.9.2 The Per-Block Random Mode (PBR)

We next present the Per-Block Random Mode (PBR). PBR is a non-standard and inefficient mode; we find it worth discussing, since it is a simple way to construct a secure encryption scheme, from a PRF, PRP or block cipher.

Let  $m = m_1||m_2||\dots||m_M$  be a plaintext message, where each  $m_i \in \{0, 1\}^n$  is one block. PBR-mode encryption of  $m$  is denoted  $c = E^{PBR}(m; r)$ , where  $r = r_1||r_2||\dots||r_M$  is a corresponding string of  $n$ -bit random blocks, and computed as follows:

$$c = E_k^{PBR}(m; r) = c_1||\dots||c_M \text{ WHERE } c_i = (r_i, m_i \oplus E_k(r_i)) \quad (2.12)$$

Namely, encrypt each message block  $m_i$  with the corresponding random block  $r_i$ . Note that we can encode each  $c_i$  simply as  $c_i = r_i||m_i$ , i.e., as a string of  $2n$  bits; the pairwise notation is equivalent, and a bit easier to work with. Use either notation to write the corresponding decryption  $D^{PBR}(c)$  in Exercise 2.20.

Note that PBR mode does not use at all the ‘decryption’ function  $D$  of the underlying block cipher - for either encryption or decryption. This is the reason that PBR can be deployed using a PRF or PRP, and not just using a block cipher. As can be seen from Table 2.4 and Exercise 2.20, this also holds for OFB and CFB modes.

PBR is *not* a standard mode, and rightfully so, since it is wasteful: it requires the use of one block of random bits per each block of the plaintext, and all these random blocks also become part of the ciphertext and used for decryption, i.e., the length of the ciphertext is double the length of the plaintext. However, PBR is *secure* - allowing us to discuss a simple provably-secure con-

struction of a symmetric cryptosystem, based on the security of the underlying block cipher (reversible PRP).

**Theorem 2.1.** *If  $E$  is a PRF or PRP, or  $(E, D)$  is a block cipher (reversible PRP), then  $(E^{PBR}, D^{PBR})$  is a CPA-indistinguishable symmetric encryption.*

*Proof.* We present the proof when  $E$  is a PRF; the other cases are similar. We also focus, for simplicity, on encryption of a single-block message,  $m = m_1 \in \{0, 1\}^n$ .

Denote by  $(\hat{E}^{PBR}, \hat{D}^{PBR})$  the same construction, except using, instead of  $E$ , a ‘truly’ random function  $f \xleftarrow{\$} \{0, 1\}^n \rightarrow \{0, 1\}^n$ . In this case, for any pair of plaintext messages  $m_0, m_1$  selected by the adversary, the probability of  $c^* = f(m_0)$  is exactly the same as the probability of  $c^* = f(m_1)$ , from symmetry of the random choice of  $f$ . Hence, the attackers’ success probability, when ‘playing’ the IND-CPA game (Def. 2.1) ‘against’  $(\hat{E}^{PBR}, \hat{D}^{PBR})$  is exactly half. Note that this holds even for computationally-unbounded adversary.

Assume, to the contrary, that there is some PPT adversary  $ADV$ , that is able to gain a non-negligible advantage  $c^n$  against  $(E^{PBR}, D^{PBD})$ . Recall that this holds, even assuming  $E$  is a PRF - however, as argued above,  $ADV$  succeeds with probability exactly half, i.e., with exactly zero advantage, against  $(\hat{E}^{PBR}, \hat{D}^{PBR})$ , i.e., if  $E$  was a truly random function.

We can use  $ADV$  to distinguish between  $E_k(\cdot)$  and a random function, with significant probability, contradicting the assumption that  $E_k$  is a PRF; see Def. 2.6. Namely, we run  $ADV$  against the PBR construction instantiated with either a true random function or  $E_k(\cdot)$ , resulting in either  $(\hat{E}^{PBR}, \hat{D}^{PBR})$  or  $(E^{PBR}, D^{PBR})$ , correspondingly. Since  $ADV$  wins with significant advantage against  $(E^{PBR}, D^{PBR})$ , and with no advantage against  $(\hat{E}^{PBR}, \hat{D}^{PBR})$ , this allows distinguishing, proving the contradiction.  $\square$

**Error propagation, integrity and CCA security** Since PBR mode encrypts the plaintext by bitwise XOR, i.e.,  $c_i = (r_i || m_i \oplus E_k(r_i))$ , flipping a bit in the second part result in flipping of the corresponding bit in the decrypted plaintext, with no other change in the plaintext. We say that such bit errors are *perfectly localized* or have no error propagation. On the other hand, bit errors in the random pad part  $r_i$  corrupt the entire corresponding plaintext block, i.e., are propagated to the entire block. In any case, errors are somewhat localized - other plaintext blocks are decrypted intact.

This property may seem useful, to limit the damage of such errors, but that value is very limited. On the other hand, this property has two negative security implications. The first is obvious: an attacker can flip specific bits in the plaintext, i.e., PBR provides no integrity protection. Of course, we did not require cryptosystems to ensure integrity; in particular, the situation is identical for other bitwise-XOR ciphers such as OTP.

The other security drawback is that PBR is not IND-CCA secure. This directly results from the error localization property. Since all the modes we

show in this chapter localize errors (perfectly, or to a single or two blocks), it follows that *none* of these modes is IND-CCA secure.

**Exercise 2.21** (Error localization conflicts with IND-CCA security). *Show that every cryptosystem where errors in one ciphertext block are localized to one or two corresponding blocks, is not IND-CCA secure.*

*Hint:* Consider the case of three-block plaintexts; one block must not be corrupted. This suffices for attacker to succeed in the IND-CCA game.

### 2.9.3 The Output-Feedback (OFB) Mode

We now proceed to discuss standard modes, which provably-ensure secure encryption, with randomization, for multiple-block messages - yet are more efficient cf. to the PRB mode.

We begin with the simple Output-Feedback (OFB) Mode. In spite of its simplicity, this mode ensures provably-secure encryption - and requires the generation and exchange of only a *single block of random bits*, cf. to one block of random bits per each plaintext block, as in PRB.

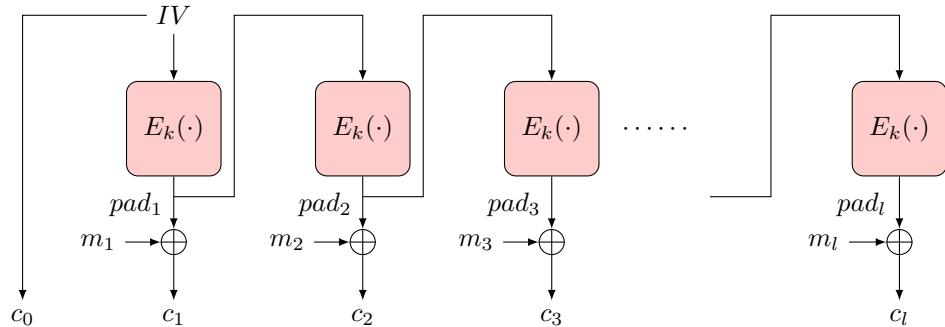


Figure 2.22: Output Feedback (OFB) mode encryption. Adapted from [68].

The OFB mode is illustrated in Figs. 2.22 (encryption) and 2.23 (decryption). OFB is a variant on the PRF-based stream cipher discussed in subsection 2.5.1 and illustrated in Fig. 2.14, and, like it, operates on input which consists of  $l$  blocks of  $n$  bits each. The difference is that OFB uses a PRP (block cipher)  $E_k$  instead of the PRF  $PRF_k$ .

We use a random *Initialization Vector (IV)* as a ‘seed’ to generate a long sequence of pseudo-random  $n$ -bit *pad blocks*,  $pad_1, \dots, pad_l$ , to encrypt plaintext blocks  $m_1, \dots, m_l$ . We next do bitwise XOR of the pad blocks  $pad_1, \dots, pad_l$ , with the corresponding plaintext blocks  $m_1, \dots, m_l$ , resulting in the ciphertext which consists of the random IV  $c_0$  and the results of the XOR operation, i.e.  $c_1 = m_1 \oplus pad_1$ ,  $c_2 = m_2 \oplus pad_2, \dots$

Let us now define  $OFB - E_k(m)$ , the OFB mode for a given block cipher  $(E, D)$ . For simplicity we define  $OFB - E_k(m)$  for messages  $m$  which consist of

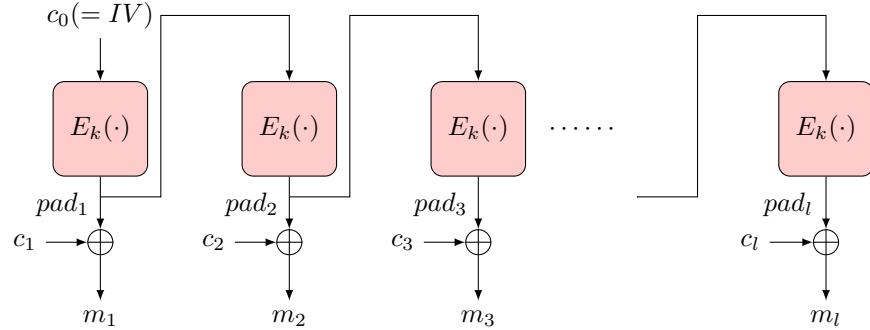


Figure 2.23: Output Feedback (OFB) mode decryption. Adapted from [68].

some number  $l$  of  $n$ -bit blocks, i.e.,  $m = m_1 || \dots || m_l$ , where  $(\forall i \leq l) |m_i| = n$ . Then  $OFB - E_k(m)$  is defined as:

$$OFB - E_k(m_1 || \dots || m_l) = (c_0 || c_1 || \dots || c_l) \quad (2.13)$$

where:

$$pad_0 \xleftarrow{\$} \{0, 1\}^n \quad (2.14)$$

$$pad_i \leftarrow E_k(pad_{i-1}) \quad (2.15)$$

$$c_0 \leftarrow pad_0 \quad (2.16)$$

$$c_i \leftarrow pad_i \oplus m_i \quad (2.17)$$

**Offline pad pre-computation** The OFB mode allows both encryption and decryption process to pre-compute the pad, ‘offline’ - i.e., before the plaintext and ciphertext, respectively, are available. Offline pad pre-computation is possible, since the pad does not depend on the plaintext (or ciphertext). This can be important, e.g., when limited computation speed CPU needs to support a limited number of ‘short bursts’, without adding latency. Once the plaintext/ciphertext is available, we only need one XOR operation per block.

**Parallelism.** The pad is computed sequentially; there does not appear to be way to speed-up its computation using parallelism.

**Error propagation, correction and integrity** Another important property for encryption schemes is their error propagation properties, namely, what is the impact of a modification of a single bit or block. Since OFB operates as a bit-wise stream cipher, then bit errors are *perfectly localized*: a change in any ciphertext bit simply causes a change in the corresponding plaintext bit - and no other bit. The ‘perfect bit error localization’ property is rather unique; in particular it does not hold for any of the other standard modes we discuss

(ECB, CFB and CBC), although it does hold also for the (non-standard) PBR mode.

The ‘perfect bit error localization’ property implies that *error correction works* equally well, if applied to the plaintext (before encryption, with correction applied to plaintext after decryption), or applied to ciphertext. Without localization, a single bit error in the ciphertext could translate to many bit errors in the plaintext, essentially implying that error correction can only be effectively applied to the ciphertext.

This motivated some designers to use OFB or a similar XOR-based stream cipher, so they can apply error correction on the plaintext. This was a bad idea, as it results in structured redundancy in the plaintext, which may make attacks easier. For example, assume that the attacker does not have any knowledge on the plaintext domain; this means that even exhaustive search cannot be applied, simply since the attacker cannot identify that a particular key is incorrect, by observing that it results in corrupt plaintext - since there is no way to distinguish between the real plaintext and corrupt plaintext.

A more important example is the *Ciphertext-Only CTO attack* on the A5/1 and A5/2 stream ciphers [6], both used in the GSM protocol. This attack exploits the known relationship between ciphertext bits, due to the fact that Error Correction Code was applied to the plaintext. This redundancy suffices to attack the ciphers, using techniques that normally can be applied only in Known Plaintext attacks. For details, see [6].

One may wonder, why would these designers prefer to apply error correction to the plaintext rather than to the ciphertext. One motivation may be the hope that this may make cryptanalysis harder, e.g., corrupt some plaintext statistics such as letter frequencies.

Another motivation may be the hope that applying error correction to the plaintext may provide integrity, since attackers changing the ciphertext will end up corrupting the plaintext. Due to the perfect bit error localization, an attacker can easily flip a specific plaintext bit - by flipping the corresponding ciphertext bit. However, since the attacker can flip multiple ciphertext bits, thereby flipping the corresponding plaintext bits, there are many cases where the attacker can modify the ciphertext in such a way to flip specific bits in the plaintext - while also ‘fixing’ the error detection/correction code, to make the message appear correct. We conclude the following principle.

**Principle 8** (Minimize plaintext redundancy). *Plaintext should preferably have minimal redundancy. In particular, plaintext should preferably not contain Error Correction or Detection codes.*

Namely, applying error correction to plaintext is a bad idea - certainly when using stream-cipher design such as OFB. This raises the obvious question: can an encryption mode of a block cipher, also protect the integrity of the decrypted plaintext? Both of the following modes, CFB and CBC, provide some defense of integrity - by *ensuring errors do propagate*.

**Provable security of OFB.** The above discussed weaknesses are due to incorrect *deployments* of OFB; correctly used, OFB is secure. Proving the security of OFB follows along similar lines to Theorem 2.1, except that in order to deal with multi-block message, we will need to use a more elaborate proof technique called ‘hybrid proof’; we leave that for courses and books focusing on Cryptology, e.g., [58, 106].

#### 2.9.4 The Cipher Feedback (CFB) Mode

We now present the Cipher Feedback (CFB) Mode. Like most standard modes (except ECB), it also uses a random first block (‘initialization vector’, IV). In fact, CFB further resembles OFB; it uses the IV to generate a first pseudo-random pad  $pad_1 = E_k(IV)$ , and it computes ciphertext blocks by bitwise XOR between the pad blocks and the corresponding plaintext blocks,  $c_i = pad_i \oplus m_i$ .

The difference between CFB and OFB is just in the ‘feedback’ mechanism, namely, the computation of the pads  $p_i$  (for  $i > 1$ ). In CFB mode, this is done using the ciphertext rather than the previous pad, i.e.,  $pad_i = E_k(c_{i-1}) = E_k(pad_{i-1} \oplus m_{i-1})$ . See Fig. 2.24.

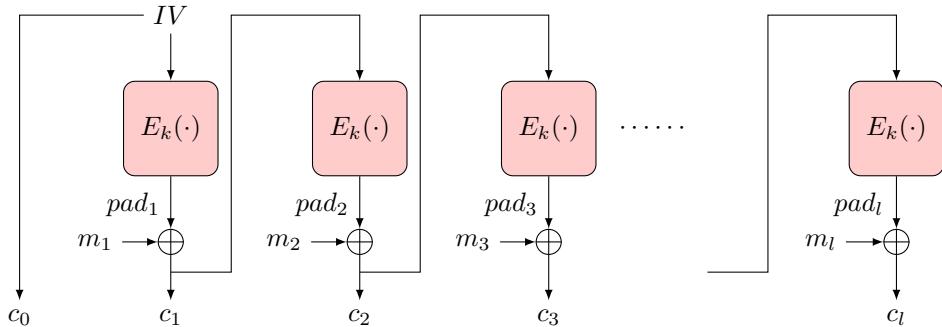


Figure 2.24: Cipher Feedback (CFB) mode encryption. Adapted from [68].

**Optimizing implementations: parallel decryption, but no precomputation** Unlike OFB, the CFB mode does not support offline pre-computation of the pad, since the pad depends on the ciphertext (of previous block).

One optimization that is possible is to parallelize the decryption operation. Namely, decryption may be performed for all blocks in parallel, since the decryption  $m_i$  of block  $i$  is  $m_i = c_i \oplus p_i = c_i \oplus E_k(c_{i-1})$ , i.e., can be computed based on the ciphertexts of this block and of the previous block.

**Error localization and integrity** Error localization in CFB is not perfect; a single bit error in one ciphertext block, completely corrupts the following plaintext block.

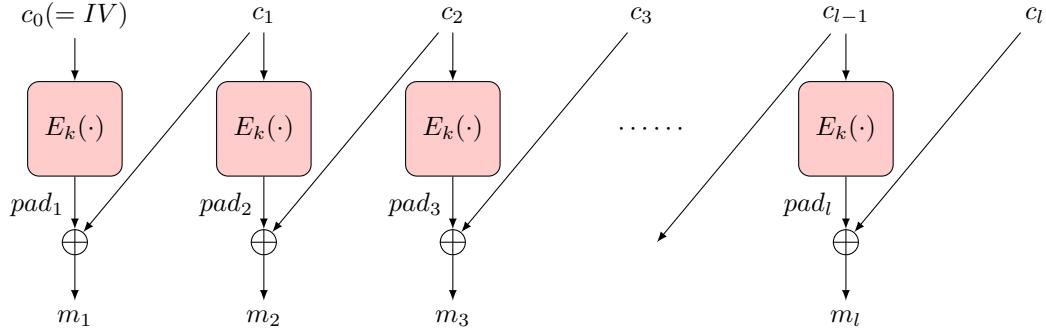


Figure 2.25: Cipher Feedback (CFB) mode decryption. Adapted from [68].

As we discussed for OFB, this reduction in error localization may be viewed as an advantage in ensuring integrity. Like OFB mode, the CFB mode allows the attacker to flip specific bits in the decrypted plaintext, by flipping corresponding bits in the ciphertext. However, as a result of such bit flipping, say in block  $i$ , the decrypted plaintext of the following block is completely corrupted. Intuitively, this implies that applying an error-detection code to the plaintext, would allow detection of such changes, in contrast to the situation with OFB mode.

However, this dependency on the error detection code applied to the plaintext may cause some concerns. First, it is an assumption about the way that OFB is used; can we provide some defense for integrity that will not depend on such additional mechanisms as an error detection code? Second, it seems challenging to prove that the above intuition is really correct, and this is likely to depend on the specifics of the error detection code used. Finally, adding error detection code to the plaintext increases its redundancy, in contradiction to Principle 8. We next present the CBC mode, which provides a different defense for integrity, which addresses these concerns.

### 2.9.5 The Cipher-Block Chaining (CBC) mode

We now present one last encryption mode, the Cipher-Block Chaining (CBC) mode. CBC is a very popular mode for encryption of multiple-block messages; like CFB, it allows parallel decryption but not offline pad precomputation.

The CBC mode, like the OFB and CFB modes, uses a random Initialization Vector (IV) which is used as the first block of the ciphertext,  $c_0 \xleftarrow{\$} \{0, 1\}^n$ . However, in contrast to OFB and CFB, to encrypt the  $i^{\text{th}}$  plaintext block  $m_i$ , it XOREs it with the previous ciphertext block  $c_{i-1}$ , and *then* applies the block cipher. Namely,  $c_i = E_k(c_{i-1} \oplus m_i)$ . See Fig. 2.26.

Note: notations are not exactly consistent with text, should be fixed.

More precisely, let  $(E, D)$  be a block cipher, and let  $m = m[1] \parallel \dots \parallel m[n]$  be a message (broken into blocks). Then the CBC encryption of  $m$  using key  $k$

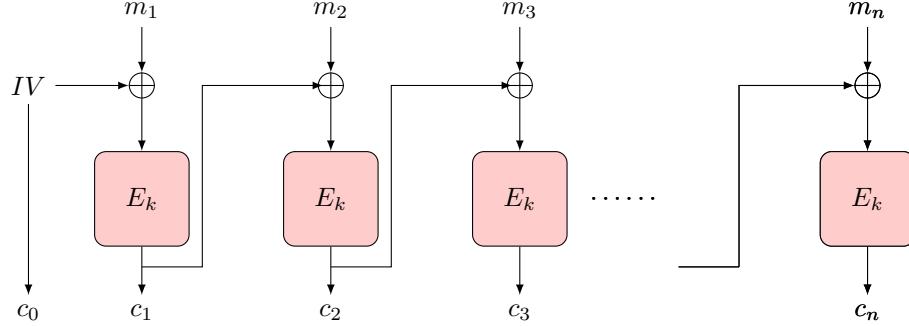


Figure 2.26: Cipher Block Chaining (CBC) mode encryption. Adapted from [68].

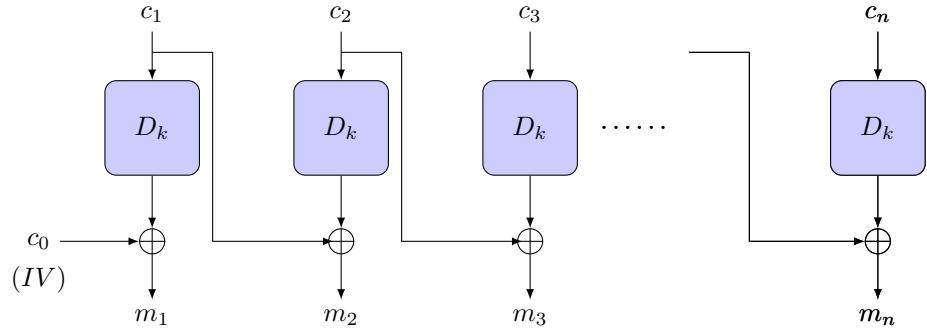


Figure 2.27: Cipher Block Chaining (CBC) mode decryption. Adapted from [68]. Note: notations are not exactly consistent with text, should be fixed.

and initialization vector  $IV \in \{0, 1\}^l$  is defined as:

$$CBC_k^E(m; IV) = c[0] || c[1] || \dots || c[n] | c[0] = IV, \quad (i > 0) \quad c[i] = E_k(c[i-1] \oplus m[i]) \quad (2.18)$$

The CBC mode, like the other modes (except ECB), ensures IND-CPA, i.e., security against CPA attacks, provided that the underlying block cipher is a secure PRP; in fact, a PRF suffices. However, it is not secure against CCA attacks.

**Exercise 2.22.** Demonstrate that CBC mode does not ensure security against CCA attacks.

**Error propagation and integrity** Any change in the CBC ciphertext, even of one bit, results in unpredictable output from the block-cipher's ‘decryption’ operation, and hence with unpredictable decryption. Namely, flipping a bit in

the ciphertext block  $i$  does not flip the corresponding bit in plaintext block  $i$ , as it did in the OFB and CFB modes.

However, flipping of a bit in the ciphertext block  $c_{i-1}$ , without change to block  $c_i$ , results in flipping of the corresponding bit in the  $i^{\text{th}}$  decrypted plaintext block. Namely, bit-flipping is still possible in CBC, it is just a bit different - and it ‘corrupts’ the previous block, rather than the following block. Indeed, this kind of tampering is used in several attacks on systems deploying CBC, such as the Poodle attack [88].

#### 2.9.6 Ensuring CCA Security

We already observed in Exercise 2.21, that any cryptosystem (and mode) that ensures error localization to some extent, cannot be IND-CCA secure. This implies that none of the modes we discussed is IND-CCA secure. Such failure can occur even for the much weaker - and more common - case of Feedback-only CCA attacks, where the attacker does not receive the decrypted plaintext, but only indication is the plaintext was ‘valid’ or not.

How can we ensure security against CCA attacks? One intuitive defense is to avoid giving any feedback on failures. However, this is harder than it may seem. In particular, feedback may involve subtle timing channels. Furthermore, preventing of feedback on errors will make it harder to resolve problems.

A better approach may be to *detect the chosen-ciphertext queries*. One simple way to do this, is by *authenticating* the ciphertext, i.e., appending to the ciphertext an *authentication tag*, which allows secure detection of any modification in the ciphertext. Such authentication is the subject of the next chapter.

### 2.10 Case study: the (in)security of WEP

We conclude this chapter, and further motivate the next, by discussing a case study: vulnerabilities of the *Wired Equivalency Privacy (WEP)* standard [33]. Notice that these critical vulnerabilities were discovered long ago, mostly in [28], relatively soon after the standard was published; yet, products and networks supporting WEP still exist. This is an example of the fact that once a standard is published and adopted, it is often very difficult to fix security. Hence, it is important to carefully evaluate security in advance, in an open process that encourages researchers to find vulnerabilities, and, where possible, with proofs of security. To address these vulnerabilities, WEP was replaced - possibly in too much haste - with a new standard, the (*WPA*); vulnerabilities were also found in WPA, see ??.

WEP stands for *Wired Equivalency Privacy*; it was developed as part of the IEEE 802.11b standard, to provide some protection of data over wireless local area networks (also known as WiFi networks). As the name implies, the original goals aimed at a limited level of privacy (meaning confidentiality), which was deemed ‘equivalent’ to the (physically limited) security offered by a wired connection. WEP includes encryption for confidentiality, CRC-32

error-detection code ('checksum') for integrity, and authentication to prevent injection attacks.

WEP assumes a symmetric key between the mobile devices and an *access point*. Many networks share the same key with all mobiles, which obviously allow each device to eavesdrop to all communication. However, other vulnerabilities exist even if a network uses good key-management mechanisms to share a separate key with each mobile.

Confidentiality in WEP is protected using the RC4 stream-cipher, used as a PRG-based stream cipher as in subsection 2.5.1. The PRG is initiated with a secret shared key, which is specified to be only 40 bits long. This short key size was chosen intentionally, to allow export of the hardware, since when the standard was drafted there were still export limitations on longer-key cryptography. Some implementations also support longer, 104-bit keys.

The PRG is also initiated with 24-bit per-packet random *Initialization Vector (IV)*. We use  $RC4_{IV,k}$  to denote the random string output by RC4 when initialized using given  $IV, k$  pair.

WEP packets use the CRC-32 error detection/correction code, computed over the plaintext message  $m$ ; we denote the result by  $CRC(m)$ . The CRC-32 is a popular family of error detection/correction code, which can detect most corruption of the message, and even allow correction of messages with up to 3 or 5 corrupted bits, depending on the specific code [75].

We use  $WEP_k(m; IV)$  to denote the output packet, which consists of the IV and the bitwise XOR of the pad generated by RC4 with the message and the CRC, i.e.:  $WEP_k(m; IV) = [IV, RC4_{IV,k} \oplus (m||CRC(m))]$ . It is often easier to focus on the second part,  $WEP'_k(m; IV) = RC4_{IV,k} \oplus (m||CRC(m))$ ; namely,  $WEP_k(m; IV) = [IV, WEP'_k(m; IV)]$ .

### 2.10.1 CRC-then-XOR does not ensure integrity

CRC-32 is a quite good error detection/correction code. By encrypting the output of CRC, specifically by XORing with the pseudo-random pad generated by RC4, the WEP designers hoped to protect message integrity. However, like many other error correcting codes, CRC-32 is *linear*, namely:

$$CRC(m \oplus m') = CRC(m) \oplus CRC(m') \quad (2.19)$$

Hence, an attacker can change the message  $m$  sent in a WEP packet, by flipping any desired bits, and appropriately adjust the CRC field.

Specifically, let  $\Delta$  represent the string of length  $|m|$  containing 1 for bit locations that the attacker wishes to flip. Given  $WEP'_k(m; IV)$ , the attacker can compute a *valid*  $WEP'_k(m \oplus \Delta; IV) = RC4_{IV,k} \oplus (m||CRC(m))$  as follows:

$$\begin{aligned} WEP'_k(m \oplus \Delta; IV) &= RC4_{IV,k} \oplus (m \oplus \Delta||CRC(m \oplus \Delta)) \\ &= RC4_{IV,k} \oplus (m \oplus \Delta||CRC(m) \oplus CRC(\Delta)) \\ &= RC4_{IV,k} \oplus (m||CRC(m)) \oplus (\Delta||CRC(\Delta)) \\ &= WEP'_k(m; IV) \oplus WEP'_k(\Delta; IV) \end{aligned}$$

Namely, the CRC mechanism, XOR-encrypted, does not provide any meaningful integrity protection.

**WEP authentication-based vulnerabilities** WEP defines two modes of authentication, although one of them, called *open-system authentication*, simply means that there is no authentication.

The other mode is called *shared-key authentication*. It works very simply: the access point sends a random challenge  $R$ ; and the mobile sends back  $WEP_k(R; IV)$ , i.e., a proper WEP packet containing the ‘message’  $R$ .

This authentication mode is currently rarely used, since it allows attacks on the encryption mechanism. First, notice it provides a trivial way for the attacker to obtain ‘cribs’ (known plaintext - ciphertext pairs). Of course, encryption systems *should* be protected against known-plaintext attacks; however, following the *conservative design principle* (principle 6), system designers should try to make it difficult for attackers to obtain cribs. In the common, standard case of 40-bit WEP implementations, a crib is deadly - attacker can now do a trivial exhaustive search to find the key.

Even when using longer keys (104 bits), the shared-key authentication opens WEP to simple cryptanalysis attack. Specifically, since  $R$  is known, the attacker learns  $RC4_{IV,k}$  for a given, random  $IV$ . Since the length of the  $IV$  is just 24 bits, it is feasible to obtain a collection of most  $IV$  values and the corresponding  $RC4_{IV,k}$  pads, allowing decryption of most messages.

As a result of these concerns, most WEP systems use only open-systems authentication mode, i.e., do not provide any authentication.

**Further WEP encryption vulnerabilities** We briefly mention two further vulnerabilities of the WEP encryption mechanism.

The first vulnerability exploits the integrity vulnerability discussed earlier. As explained there, the attacker can flip arbitrary bits in the WEP payload message. WEP is a link-layer protocol; the payload is usually an Internet Protocol (IP) packet, whose header contains, in known position, the destination address. An attacker can change the destination address, causing forwarding of the packet directly to the attacker!

The second vulnerability is the fact that WEP uses ‘plain’ RC4, which has been shown in [84] to be vulnerable.

## 2.11 Encryption: Final Words

Confidentiality, as provided by encryption, is the oldest goal of cryptology, and is still critical to the entire area of cybersecurity. Encryption has been studied for millennium, but for many years, the design of cryptosystems was kept secret, in the hope of improving security. Kerckhoffs’ principle, however, has been widely adopted and caused cryptography to be widely studied and deployed, in industry and academia.

Cryptography was further revolutionized by the introduction of precise definitions and proofs of security by reduction, based on the theory of complexity. In particular, modern study of applied cryptography makes extensive use of provable security, and is mostly studied on *computational security*, i.e., ensuring security properties with high probability, against probabilistic polynomial time (PPT) adversaries. We have seen a small taste of such definitions and proofs in this chapter; we will see a bit more later on, but for a real introduction to the theory of cryptography, see appropriate textbooks, such as [57, 58].

## 2.12 Encryption and Pseudo-Randomness: Additional exercises

**Exercise 2.23.** *ConCrypt Inc. announces a new symmetric encryption scheme, CES. ConCrypt announces that CES uses a 128-bit keys and is five times faster than AES, and is the first practical cipher to be secure against computationally-unbounded attackers. Is there any method, process or experiment to validate or invalidate these claims? Describe or explain why not.*

**Exercise 2.24.** *ConCrypt Inc. announces a new symmetric encryption scheme, CES512. ConCrypt announces that CES512 uses a 512-bit keys, and as a result, is proven to be much more secure than AES. Can you point out any concerns with using CES512 instead of AES?*

**Exercise 2.25.** *Compare the following pairs of attack models. For each pair (A, B), state whether every cryptosystem secure under attack model A is also secure under attack model B and vice versa. Prove (if you fail to prove, at least give compelling argument) your answers. The pairs are:*

1. (Cipher-text only, Known plain text)
2. (Known plain text, Chosen plain text)
3. (Known plain text, Chosen cipher text)
4. (Chosen plain text, Chosen cipher text)

**Exercise 2.26.** *Alice is communicating using the GSM cellular standard, which encrypts all calls between her phone and the access tower. Identify the attacker model corresponding to each of the following cryptanalysis attack scenarios:*

1. *Assume that Alice and the tower use a different shared key for each call, and that Eve knows that specific, known message is sent from Bob to Alice at given times.*
2. *Assume (only) that Alice and the tower use a different shared key for each call.*
3. *Assume all calls are encrypted using a (fixed) secret key  $k_A$  shared between Alice's phone and the tower, and that Eve knows that specific, known control messages are sent, encrypted, at given times.*
4. *Assume (only) that all calls are encrypted using a (fixed) secret key  $k_A$  shared between Alice's phone and the tower*

**Exercise 2.27.** *We covered several encryption schemes in this chapter, including At-Bash (AzBy), Caesar, Shift-cipher, general monoalphabetic substitution, OTP, PRG-based stream cipher, RC4, block ciphers, and the ‘modes’ in Table 2.4. Which of these is: (1) stateful, (2) randomized, (3) FIL, (4) polynomial-time?*

**Exercise 2.28.** Consider use of AES with key length of 256 bits and block length of 128 bit, for two different 128 bit messages, A and B (i.e., one block each). Bound, or compute precisely if possible, the probability that the encryption of A will be identical to the encryption of B, in each of the following scenarios:

1. Both messages are encrypted with the same randomly-chosen key, using ECB mode.
2. Both messages are encrypted with two keys, each of which is chosen randomly and independently, and using ECB mode.
3. Both messages are encrypted with the same randomly-chosen key, using CBC mode.
4. Compute now the probability the the same message is encrypted to the same ciphertext, using a randomly-chosen key and CBC mode.

**Exercise 2.29** (Key dependent message security). Several works design cryptographic schemes such as encryption schemes, which are secure against a ‘key dependent message attack’, where the attacker specifies a function  $f$  and receives encryption  $E_k(f(k))$ , i.e., encryption of the message  $f(k)$  where  $k$  is the secret key. See [23].

1. Extend the definition of secure pseudo-random function for security against key-dependent message attacks.
2. Suppose that  $F$  is secure pseudo-random function. Show a (‘weird’) function  $F'$  which is also a secure pseudo-random function, but not secure against key-dependent message attacks.

**Exercise 2.30** (PRG constructions). Let  $G : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$  be a secure PRG. Is  $G'$ , as defined in each of the following sections, a secure PRG? Prove.

1.  $G'(s) = G(s^R)$ , where  $s^R$  means the reverse of  $s$ .
2.  $G'(r||s) = r||G(s)$ , where  $r, s \in \{0, 1\}^n$ .
3.  $G'(s) = G(s \oplus G(s)_{1\dots n})$ , where  $G(s)_{1\dots n}$  are the  $n$  most-significant bits of  $G(s)$ .
4.  $G'(s) = G(\pi(s))$  where  $\pi$  is a (fixed) permutation.
5.  $G'(s) = G(s + 1)$ .
6. (harder!)  $G'(s) = G(s \oplus s^R)$ .
  - A. Solution to  $G'(r||s) = r||G(s)$ :
  - B. Solution to  $G'(s) = G(s \oplus G(s)_{1\dots n})$ : may not be a PRG. For example, let  $g$  be a PRG from any number  $m$  bits to  $m+1$  bits, i.e., output is pseudorandom string just one bit longer than the input. Assume even  $n$ ; for every  $x \in \{0, 1\}^{n/2}$

and  $y \in \{0, 1\}^{n/2} \cup \{0, 1\}^{1+n/2}$ , let  $G(x||y) = x||g(y)$ . If  $g$  is a PRG, then  $G$  is also a PRG (why?). However, when used in the above construction:

$$\begin{aligned} G'(x||y) &= G[(x||y) \oplus G(x||y)] \\ &= G[(x||y) \oplus (x||g(y))] \\ &= G[(x \oplus x)||y \oplus g(y))] \\ &= G[0^{n/2}||y) \oplus (x||g(y))] \\ &= 0^{n/2}||y \oplus g(y) \end{aligned}$$

As this output begins with  $n/2$  zero bits, it can be trivially distinguished from random. Hence  $G'$  is clearly not a PRG.  $\square$

**Exercise 2.31.** Let  $G_1, G_2 : \{0, 1\}^n \rightarrow \{0, 1\}^{2n}$  be two different candidate PRGs (over the same domain and range). Consider the function  $G$  defined in each of the following sections. Is it a secure PRG - assuming both  $G_1$  and  $G_2$  are secure PRGs, or assuming only that one of them is secure PRG? Prove.

1.  $G(s) = G_1(s) \oplus G_2(s)$ .
2.  $G(s) = G_1(s) \oplus G_2(s \oplus 1^{|s|})$ .
3.  $G(s) = G_1(s) \oplus G_2(0^{|s|})$ .

**Exercise 2.32.** Let  $G : \{0, 1\}^n \rightarrow \{0, 1\}^m$  be a secure PRG, where  $m > n$ .

1. Let  $m = n + 1$ . Use  $G$  to construct a secure PRG  $G' : \{0, 1\}^n \rightarrow \{0, 1\}^{2n}$ .
2. Let  $m = 2n$ , and consider  $G'(x) = G(x)||G(x+1)$ . Is  $G'$  a secure PRG?
3. Let  $m = 2n$ . Use  $G$  to construct a secure PRG  $G' : \{0, 1\}^n \rightarrow \{0, 1\}^{4n}$ .
4. Let  $m = 4n$ . Use  $G$  to construct a secure PRG  $\hat{G} : \{0, 1\}^n \rightarrow \{0, 1\}^{64n}$ .

**Exercise 2.33** (Ad-Hoc PRF competition project). In this exercise, you will experiment in trying to build directly a cryptographic scheme - in this case, a PRF - as well as in trying to ‘break’ (cryptanalyze) it. Do this exercise with others, in multiple groups (each containing one or multiple persons).

1. In the first phase, each group will design a PRF, whose input, key and output are all 64 bits long. The PRF should be written in Python (or some other agreed programming language), and only use the basic mathematical operations: module addition/subtraction/multiplication/division/remainder, XOR, max and min. You may also use comparisons and conditional code. The length of your program should not exceed 400 characters, and it must be readable. You will also provide (separate) documentation.

2. All groups will be given the documentation and code of the PRFs of all other groups, and try to design programs to distinguish these PRFs from a random function (over same input and output domains). A distinguisher is considered successful if it is able to distinguish in more than 1% of the runs.
3. Each group, say  $G$ , gets one point for every PRF that  $G$  succeeded to distinguish, and one point for every group that failed to distinguish  $G$ 's PRF from random function. The group with the maximal number of points wins.

**Exercise 2.34.** Let  $f$  be a secure Pseudo-Random Function (PRF) with  $n$  bit keys, domain and range, and let  $k$  be a secret, random  $n$  bit key. Derive from  $k$ , using  $f$ , two pseudorandom keys  $k_1, k_2$ , e.g., one for encryption and one for authentication. Each of the derived keys  $k_1, k_2$  should be  $2n$ -bits long, i.e., twice the length of  $k$ . Note: the two keys should be independent, i.e., each of them (e.g.,  $k_1$ ) should be pseudorandom, even if the adversary is given the other (e.g.,  $k_2$ ).

$$1. \ k_1 = \underline{\hspace{2cm}}$$

$$2. \ k_2 = \underline{\hspace{2cm}}$$

**Exercise 2.35** (PRF constructions). Let  $F^{n,b,l} : \{0,1\}^n \times \{0,1\}^b \rightarrow \{0,1\}^l$  be a secure PRF; for brevity, we write simply  $F_k(x)$  for  $F_k^{n,b,l}(x)$ . Is  $F'$ , as defined in each of the following sections, a secure PRF? Prove.

1.  $\hat{F}_k(m) = F_k(m \oplus 1)$ .
2.  $\hat{F}_k(m) = F_m(k)$ .
3.  $\hat{F}_k(m) = F_k(m^R)$ , where  $m^R$  means the reverse of  $m$ .
4.  $\hat{F}_k(m_L || m_R) = F_k(m_L) || F_k(m_R)$ .
5.  $\hat{F}_k(m_L || m_R) = (m_L \oplus F_k(m_R)) || (F_k(m_L) \oplus m_R)$ .
6. (harder!)  $F'_k(m_L || m_R) = F_{F_k(1^b)}(m_L) || F_{F_k(0^b)}(m_R)$ . Assume  $l = n$ .
7.  $\hat{F}_k(m) = LSb(F_k(m))$ , where  $LSb$  returns the least-significant bit of the input.
8. (harder!)  $\hat{F}_k(m_L || m_R) = F_{F_k(1^b)}(m_L) || F_{F_k(0^b)}(m_R \oplus F_k(m_L))$ . Assume  $l = n = b$ .

*Solution of  $\hat{F}_k(m) = F_k(m \oplus 1)$ :* yes, if  $F_k(m)$  is a secure PRF, then  $\hat{F}_k(m) = F_k(m \oplus 1)$  is also a secure PRF. Assume, to the contrary, that there is a PPT algorithm  $\hat{A}$  that ‘breaks’  $\hat{F}$ , i.e., there is some (strictly-positive) polynomial  $p(n)$  s.t. for sufficiently large  $n$  holds:

$$\Pr(\hat{A}^{\hat{F}_k(\cdot)}(1^n) = 1) - \Pr(\hat{A}^{\hat{f}(\cdot)}(1^n) = 1) > p(n) \quad (2.20)$$

where the probabilities are computed over uniformly-random coin tosses by  $A$  and uniformly-random choice of  $k \in \{0, 1\}^n$  and of function  $\hat{f} : \{0, 1\}^b \rightarrow \{0, 1\}^l$ .

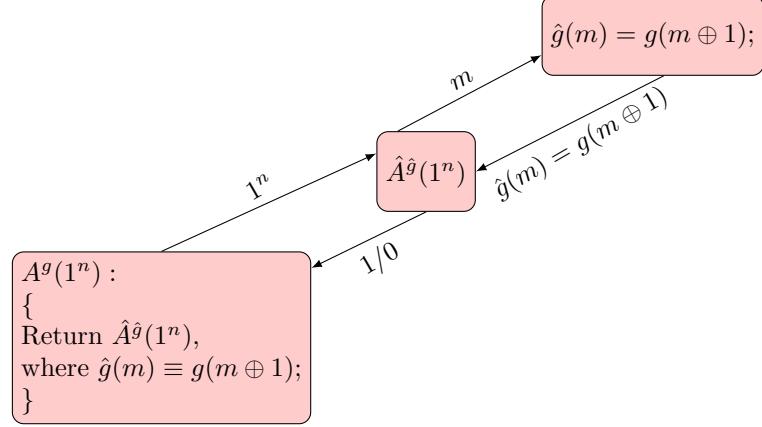


Figure 2.28: Design of adversary  $A$  for the solution of item 1 of Exercise 2.35.

We use the adversary  $\hat{A}$  as a ‘subroutine’ to implement a PPT algorithm  $A$ , as illustrated in 2.28. Namely, the value of  $A^g(1^n)$ , i.e., the output of  $A$ , given oracle to an unknown function  $g$ , and applied to security parameter  $1^n$ , is defined as:

$$A^{g(m)}(1^n) \equiv \hat{A}^{g(m \oplus 1)}(1^n) \quad (2.21)$$

Which implies trivially that:

$$\hat{A}^{g(m)}(1^n) \equiv A^{g(m \oplus 1)}(1^n) \quad (2.22)$$

By applying this equation together with the fact that, by design,  $\hat{F}_k(m) = F_k(m \oplus 1)$ , we have:

$$\hat{A}^{\hat{F}_k(m)}(1^n) = A^{F_k(m \oplus 1 \oplus 1)}(1^n) = A^{F_k(m)}(1^n) \quad (2.23)$$

Namely, we can rewrite Eq. (2.20) as:

$$\Pr(A^{F_k(m)}(1^n) = 1) - \Pr(A^{\hat{f}(m \oplus 1)}(1^n) = 1) > p(n) \quad (2.24)$$

where the probabilities are computed over uniformly-random coin tosses by  $A$  and uniformly-random choice of  $k \in \{0, 1\}^n$  and of function  $\hat{f} : \{0, 1\}^b \rightarrow \{0, 1\}^l$ . Let  $f(m) \equiv \hat{f}(m \oplus 1)$ ; a uniform choice of  $\hat{f}$  implies a uniform choice of  $f$  (since the two sets of functions are permutations). Hence we have:

$$\Pr(A^{F_k(m)}(1^n) = 1) - \Pr(A^{f(m)}(1^n) = 1) > p(n) \quad (2.25)$$

However, Eq. (2.25) implies that  $F$  is not a secure PRF, which contradicts the assumption was  $F$  is a secure PRF, proving that that assumption was wrong, i.e.,  $\hat{F}$  is a secure PRF.  $\square$

**Exercise 2.36** (ANSI X9.31 PRG and the DUHK attack). *The ANSI X9.31 is a well-known PRG design, illustrated in Fig. 2.29. In this question we investigate a weakness in it, presented in [70]; it was recently shown to be still relevant for some devices using this standard, in the so-called DUHK attack [32]. Our presentation is a slight simplification of the X9.31 design but retains the important aspects of the attack. The design uses a PRF (or block cipher)  $F : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ , with a randomly-chosen and then fixed key  $k$ ; the attacks we discuss assume that the key is then known to the attacker. Let  $f(x) = F_k(x)$ .*

1. *Let  $g(x) = f(x)||f(f(x))||\dots$ . Is  $g$  a secure PRG?*
2. *Let  $g(x) = g_1(x)||g_2(x)||\dots$ , where  $g_1(x) = f(x \oplus f(t))$ ,  $g_2(x) = f(f(t) \oplus f(g_1(x) \oplus f(t)))$ , and  $t$  is a known value (representing the time). Is  $g$  a secure PRG?*

Hint: Solution to first part is almost trivial; indeed this part is mostly there to aid you in solving the second part (which may not be much harder, after solving the first part).

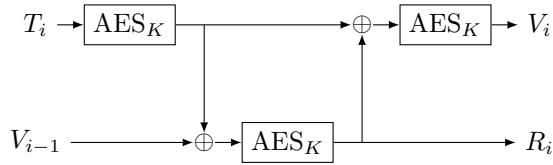


Figure 2.29: A single round of the ANSI X9.31 generator, instantiated  $F_k(x)$  by  $AES_k(x)$  (i.e., using AES as the block cipher or PRF).

**Exercise 2.37.** *A message  $m$  of length 256 bytes is encrypted using a 128-bit block cipher, resulting in ciphertext  $c$ . During transmission, the 200<sup>th</sup> bit was flipped due to noise. Let  $c'$  denote  $c$  with the 200<sup>th</sup> bit flipped, and  $m'$  denote the result of decryption of  $c'$ .*

1. *Which bits in  $m'$  would be identical to the bits in  $m$ , assuming the use of each of the following modes: (1) ECB, (2) CBC, (3) OFB, (4) CFB? Explain (preferably, with diagram).*
2. *For each of the modes, specify which bits is predictable as a function of the bits of  $m$  and the known fact that the 200<sup>th</sup> bit flipped.*

**Exercise 2.38.** *Hackme Bank protects money-transfer orders digitally sent between branches, by encrypting them using a block cipher. Money transfer orders have the following structure:  $m = f||r||t||x||y||p$ , where  $f, r$  are each 20-bits representing the payer (from) and the payee (recipient),  $t$  is a 32-bit field encoding the time,  $x$  is a 24 bit field representing the amount,  $y$  is a 128-bit comment field defined by the payer and  $p$  is 32-bit parity fields, computed as*

the bitwise-XOR of the preceding 32-bit words. Orders with incorrect parity, outdated or repeating time field, or unknown payer/payee are ignored.

Mal captures ciphertext message  $x$  containing money-transfer order of 1\\$ from Alice to his account. You may assume that Mal can ‘trick’ Alice into including a comment field  $y$  selected by Mal. Assume 64-bit block cipher. Can Mal cause transfer of larger amount to his account, and how, assuming use of the following modes:

1. ECB
2. CBC
3. OFB
4. CFB

*Solution:* The first block contains  $f, r$  (10 bits each), and top 24 bits of the time  $t$ , the second block contains 8 more bits of the time,  $x$  (24 bits) and 32 bits of the comment; block three contains 64 bits of comments, and block four contains 32 bits of comment and 32 bits of parity. Denote these four plaintext blocks by  $m_1||m_2||m_3||m_4$ .

Denote the ciphertext blocks captured by Mal as  $c_0||c_1||c_2||c_3||c_4$ , where  $c_0$  is the IV.

1. ECB: attacker select the third block (completely comment) to be identical to the second block, except for containing the maximal value in the 24 bits from bit 8 to bit 31. The attacker then switches between the third and fourth block before giving to the bank. Parity bits do not change.
2. CBC: Attacker chooses  $y$  s.t.  $m_3 = m_2$ . Now attacker sends to the bank the manipulated message  $z_0||c_1||c_2||c_3||c_4$ , where  $z_0 = m_1 \oplus m_3 \oplus c_2$ . As a result, decryption of the first block retrieves  $m_1$  correctly (as  $m_1 = z_0 \oplus m_3 \oplus c_2$ ), and decryption of the last block similarly retrieves  $m_4$  correctly (no change in  $c_3, c_4$ ). However, both the second and the third block, decrypt to the value  $(c_3 \oplus c_2 \oplus m_3)$ . Hence, the 32 bit XOR of the message does not change. The decryption of the second block (to  $c_3 \oplus c_2 \oplus m_3$ ) is likely to leave the time value valid - and to increase the amount considerably.
3. OFB: the solution is trivial since Mal can flip arbitrary bits in the decrypted plaintext (by flipping corresponding bits in the ciphertext).
4. CFB: as in CBC, attacker chooses  $y$  s.t.  $m_3 = m_2$ . Attacker sends to the bank the manipulated message  $c_0||c_1||c_1||c_1||z_4$  where  $z_4 = p_4 \oplus c_2 \oplus p_2$ .

**Exercise 2.39** (Affine block cipher). Hackme Inc. proposes the following highly-efficient block cipher, using two 64-bit keys  $k_1, k_2$ , for 64-bit blocks:  $E_{k_1, k_2}(m) = (m \oplus k_1) + k_2 \pmod{2^{64}}$ .

1. Show that  $E_{k_1, k_2}$  is an invertible permutation (for any  $k_1, k_2$ ), and the inverse permutation  $D_{k_1, k_2}$ .
  2. Show that  $(E, D)$  is not a secure block cipher (invertible PRP).
  3. Show that encryption using  $(E, D)$  is not CPA-IND, when used in the following modes: (1) ECB, (2) CBC, (3) OFB, (4) CFB.

**Exercise 2.40** (How not to build PRP from PRF). Suppose  $F$  is a secure PRF with input, output and keyspace all of length  $n$  bits. For  $x_L, x_R \in \{0, 1\}^n$ , let  $F'_k(x_L || x_R) = F_k(x_L) || F_k(x_R)$  and  $F''_k(x_L || x_R) = F_k(x_L \oplus x_R) || F_k(x_L \oplus F_k(x_L \oplus x_R))$ . Prove that neither  $F'_k$  nor  $F''_k$  are a PRP.

**Exercise 2.41** (Building PRP from a PRF). Suppose you are given a secure PRF  $F$ , with input, output and keyspace all of length  $n$  bits. Show how to use  $F$  to construct:

1. A PRP, with input and output length  $2n$  bit and key length  $n$  bits,
  2. A PRP, with input, output and key all of length  $n$  bits.

**Exercise 2.42** (Indistinguishability definition). Let  $(E, D)$  be a stateless shared-key encryption scheme, and let  $p_1, p_2$  be two plaintexts. Let  $x$  be 1 if the most significant bits of  $p_1, p_2$  are identical and 0 otherwise, i.e.,  $x = \{1 \text{ if } MSb(p_1) = MSb(p_2), \text{ else } 0\}$ . Assume that there exists an efficient algorithm  $X$  that computes  $x$  given the ciphertexts, i.e.,  $x = X(E_k(p_1), E_k(p_2))$ . Show that this implies that  $(E, D)$  is not IND-CPA secure, i.e., there is an efficient algorithm  $ADV$  which achieves significant advantage in the IND-CPA experiment. Present the implementation of  $ADV$  by filling in the missing code below:

$$\left. \begin{array}{l} \{ ADV^{E_k}(\text{'Choose'}, 1^n) : \{ \dots \} \\ \quad ADV^{E_k}(\text{'Guess'}, s, c^*) : \{ \dots \} \end{array} \right\}$$

**Exercise 2.43** (BEAST vulnerability). Versions of SSL/TLS before TLS1.1, use CBC encryption in the following way. They select the IV randomly only for the first message  $m_0$  in a connection; for subsequent messages, say  $m_i$ , the IV is simply the last ciphertext block of the previous message. This creates a vulnerability exploited, e.g., by the BEAST attack and few earlier works [5, 45]. In this question we explore a simplified version of these attacks. For simplicity, assume that the attacker always knows the next IV to be used in encryption, and can specify plaintext message and receive its CBC encryption using that IV. Assume known block length, e.g., 16 bytes.

1. Assume the attacker sees ciphertext  $(c_0, c_1)$  resulting from CBC encryption with  $c_0$  being the IV, of a single-block message  $m$ , which can have only two known values:  $m \in \{m_0, m_1\}$ . To find if  $m$  was  $m_0$  or  $m_1$ , the adversary uses fact that it knows the next IV to be used, which we denote

$c'_0$ , and asks for CBC encryption of a specially-crafted single-block message  $m'$ ; denote the returned ciphertext by the pair  $(c'_0, c'_1)$ , where  $c'_0$  is the (previously known) IV, as indicated earlier. The adversary can now compute  $m'$  from  $c'_1$ :

a) What is the value of  $m'$  that the adversary will ask to encrypt?

b) Fill the missing parts in the solution of the adversary:

$$m = \begin{cases} m_0 & \text{if } \underline{\hspace{2cm}} \\ m_1 & \text{if } \underline{\hspace{2cm}} \end{cases}$$

2. Show pseudo-code for the attacker algorithm used in the previous item.
3. Show pseudo-code for an attack that finds the last byte of message  $m$ .  
Hint: use the previous solution as a routine in your code.
4. Assume now that the attacker tries to find a long secret plaintext string  $x$  of length  $l$  bytes. Assume attacker can ask for encryption of messages  $m = p||x$ , where  $p$  is a plaintext string chosen by the attacker. Show pseudo-code for an attack that finds  $x$ . Hint: use previous solution as routine; it may help to begin considering fixed-length  $x$ , e.g., four bytes.

**Sketch of solution to second part (to be updated):** Attacker makes query for encryption of some one-block message  $y$ , receives  $\alpha_0, \alpha_1$  where  $\alpha_1 = E_k(\alpha_0 \oplus y)$ . Suppose now attacker knows next message will be encrypted with IV  $I$ . Attacker picks  $m_0 = I \oplus y \oplus \alpha_0$ , and  $m_1$  some random message. If game pick bit  $b = 0$  then attacker receives encryption of  $m_0$  which is  $I$  and  $E_k(I \oplus m_0) = E_k(I \oplus I \oplus y \oplus \alpha_0) = E_k(y \oplus \alpha_0) = \alpha_1$ ; otherwise, it receives some other string.

**Sketch of solution to third part:** solution to previous part allowed attacker to check if the plaintext was a given string; we now simply repeat this for the 256 different strings corresponding to all possible values of last byte of  $m_2$ .  $\square$

**Exercise 2.44** (Robust combiner for PRG). 1. Given two candidate PRGs, say  $G_1$  and  $G_2$ , design a robust combiner, i.e., a ‘combined’ function  $G$  which is a secure PRG if either  $G_1$  or  $G_2$  is a secure PRG.

2. In the design of the SSL protocol, there were two candidate PRGs, one (say  $G_1$ ) based on the MD5 hash function and the other (say  $G_2$ ) based on the SHA-1 hash function. The group decided to combine the two; a simplified version of the combined PRG is  $G(s) = G_2(s||G_1(s))$ . Is this a robust-combiner, i.e., a secure PRG provided that either  $G_1$  or  $G_2$  is a secure PRG?

Hint: Compare to Lemma 2.2. You may read on hash functions in chapter 4, but the exercise does not require any knowledge of that; you should simply

consider the construction  $G(s) = G_2(s||G_1(s))$  for arbitrary functions  $G_1, G_2$ .  $\square$

**Exercise 2.45** (Using PRG for independent keys). *In Example 2.2, we saw how to use a PRF to derive multiple pseudo-random keys from a single pseudo-random key, using a PRF.*

1. *Show how to derive two pseudo-random keys, using a PRG, say from  $n$  bits to  $2n$  bits.*
2. *Show how to extend your design to derive four keys from the same PRG, or any fixed number of pseudo-random keys.*

**Exercise 2.46.** Let  $(E, D)$  be a block cipher which operates on 20 byte blocks; suppose that each computation of  $E$  or  $D$  takes  $10^{-6}$  seconds (one microsecond), on given chips. Using  $(E, D)$  you are asked to implement a secure high-speed encrypting/decrypting gateway. The gateway receives packets at line speed of  $10^8$  bytes/second, but with maximum of  $10^4$  bytes received at any given second. The goal is to have minimal latency, using minimal number of chips. Present an appropriate design, argue why it achieves the minimal latency and why it is secure.

**Exercise 2.47.** Consider the AES block cipher, with 256 bit key and 128 bit blocks, and two random one-block (128 bit) messages,  $m_1$  and  $m_2$ , and two random (256-bit) keys,  $k_1$  and  $k_2$ . Calculate (or approximate/bound) the probability that  $E_{k_1}(m_1) = E_{k_2}(m_2)$ .

**Exercise 2.48** (PRF→PRG). *Present a simple and secure construction of a PRG, given a secure PRF.*

**Exercise 2.49** (Independent PRGs). *Often, a designer has one random or pseudo-random ‘seed/key’ binary string  $k \in \{0, 1\}^*$ , from which it needs to generate two or more independently pseudorandom strings  $k_0, k_1 \in \{0, 1\}^*$ ; i.e., each of these is pseudorandom, even if the other is given to the (PPT) adversary. Let PRG be a pseudo-random generator, which on input of arbitrary length  $l$  bits, produces  $4l$  output pseudorandom bits. For each of the following designs, prove its security (if secure) or its insecurity (is insecure).*

1. *For  $b \in \{0, 1\}$ , let  $k_b = \text{PRG}(b||k)$ .*
2. *For  $b \in \{0, 1\}$ , let  $k_b = \text{PRG}(k)[(b \cdot 2 \cdot |k|) \dots ((2 + b) \cdot |k| - 1)]$ .*

*Solution:*

1. Insecure, since it is possible for a secure PRG to ignore the first bit, i.e.,  $\text{PRG}(b||s) = \text{PRG}(\bar{b}||s)$ , resulting in  $k_0 = \text{PRG}(0||k) = \text{PRG}(1||k) = k_1$ . We skip the (simple) proof that such a PRG may be secure.
2. Secure, since each of these is a (non-overlapping) subset of the output of the PRG.

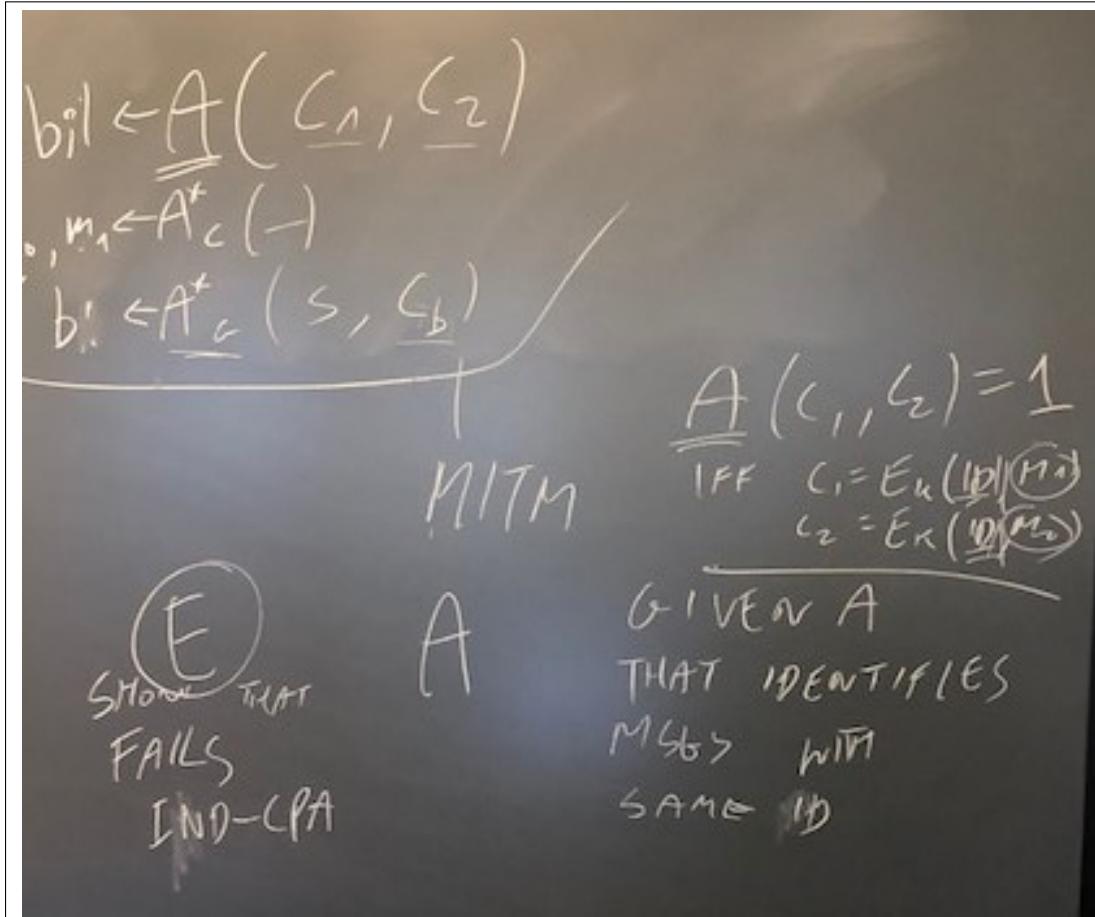


Figure 2.30: Figure for Exercise 2.50 (to be done).

**Exercise 2.50** (Indistinguishability hides partial information). *In this exercise we provide an example to the fact that a cryptosystem that ensures indistinguishability (IND-CPA), is guaranteed not to leak partial information about plaintext, including relationships between the plaintext corresponding to different ciphertexts. Let  $(E, D)$  be an encryption scheme, which leaks some information about the plaintexts; specifically we assume that there exists an efficient adversary  $A$  s.t. for two ciphertexts  $c_1, c_2$  of  $E$ , holds  $A(c_1, c_2) = 1$  if and only if the plaintexts share a common prefix, e.g.,  $c_1 = E_k(ID||m_1)$  and  $c_2 = E_k(ID||m_2)$  (same prefix,  $ID$ ). Show that this implies that  $(E, D)$  is not IND-CPA secure. See the question illustrated in Fig. 2.30.*

*Solution:* see sketch in Fig. 2.31.

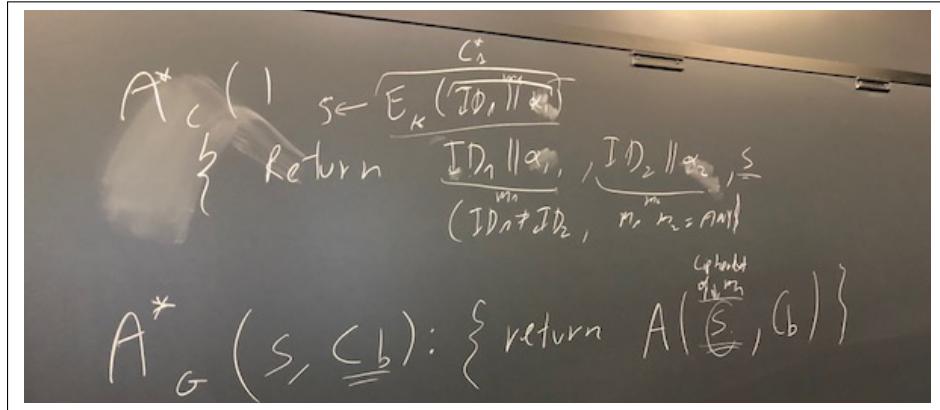


Figure 2.31: Figure for solution of Exercise 2.50 (to be done).

**Exercise 2.51** (Encrypted cloud storage). Consider a set  $P$  of  $n$  sensitive (plaintext) records  $P = \{p_1, \dots, p_n\}$  belonging to Alice, where  $n < 10^6$ . Each record  $p_i$  is  $l > 64$  bits long ( $(\forall i)(p_i \in \{0, 1\}^l)$ ). Alice has very limited memory, therefore, she wants to store an encrypted version of her records in an insecure/untrusted cloud storage server  $S$ ; denote these ciphertext records by  $C = \{c_1, \dots, c_n\}$ . Alice can later retrieve the  $i^{\text{th}}$  record, by sending  $i$  to  $S$ , who sends back  $c_i$ , and then decrypting it back to  $p_i$ .

1. Alice uses some secure shared key encryption scheme  $(E, D)$ , with  $l$  bit keys, to encrypt the plaintext records into the ciphertext records. The goal of this part is to allow Alice to encrypt and decrypt each record  $i$  using a unique key  $k_i$ , but maintain only a single ‘master’ key  $k$ , from which it can easily compute  $k_i$  for any desired record  $i$ . One motivation for this is to allow Alice to give keys to specific record(s)  $k_i$  to some other users (Bob, Charlie,...), allowing decryption of only the corresponding ciphertext  $c_i$ , i.e.,  $p_i = D_{k_i}(c_i)$ . Design how Alice can compute the key  $k_i$  for each record  $(i)$ , using only the key  $k$  and a secure block cipher (PRP)  $(F, F^{-1})$ , with key and block sizes both  $l$  bits. Your design should be as efficient and simple as possible. Note: do not design how Alice gives  $k_i$  to relevant users - e.g., she may do this manually; and do not design  $(E, D)$ .

Solution:  $k_i = \underline{\hspace{2cm}}$

2. Design now the encryption scheme to be used by Alice (and possibly by other users to whom Alice gave keys  $k_i$ ). You may use the block cipher  $(F, F^{-1})$ , but not other cryptographic functions. You may use different encryption scheme  $(E^i, D^i)$  for each record  $i$ . Ensure confidentiality of the plaintext records from the cloud, from users (not given the key for that record), and from eavesdroppers on the communication. Your design should be as efficient as possible, in terms of the length of the ciphertext

(in bits), and in terms of number of applications of the secure block cipher (PRP)  $(F, F^{-1})$  for each encryption and decryption operation. In this part, assume that Alice stores  $P$  only once, i.e., never modifies records  $p_i$ . Your solution may include a new choice of  $k_i$ , or simply use the same as in the previous part.

Solution:  $k_i = \underline{\hspace{2cm}}$ ,  
 $E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$ ,  
 $D_{k_i}^i(c_i) = \underline{\hspace{2cm}}$ .

3. Repeat, when Alice may modify each record  $p_i$  few times (say, up to 15 times); let  $n_i$  denote number of modifications of  $p_i$ . The solution should allow Alice to give (only) her key  $k$ , and then Bob can decrypt all records, using only the key  $k$  and the corresponding ciphertexts from the server. Note: if your solution is the same as before, this may imply that your solution to the previous part is not optimal.

Solution:  $k_i = \underline{\hspace{2cm}}$ ,  
 $E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$ ,  
 $D_{k_i}^i(c_i) = \underline{\hspace{2cm}}$ .

4. Design an efficient way for Alice to validate the integrity of records retrieved from the cloud server  $S$ . This may include storing additional information  $A_i$  to help validate record  $i$ , and/or changes to the encryption/decryption scheme or keys as designed in previous parts. As in previous parts, your design should only use the block cipher  $(F, F^{-1})$ .

Solution:  $k_i = \underline{\hspace{2cm}}$ ,  
 $E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$ ,  
 $D_{k_i}^i(c_i) = \underline{\hspace{2cm}}$ ,  
 $A_i = \underline{\hspace{2cm}}$ .

5. Extend the keying scheme from the first part, to allow Alice to also compute keys  $k_{i,j}$ , for integers  $i, j \geq 0$  s.t.  $1 \leq i \cdot 2^j + 1, (i+1) \cdot 2^j \leq n$ , where  $k_{i,j}$  would allow (efficient) decryption of ciphertext records  $c_{i \cdot 2^j + 1}, \dots, c_{(i+1) \cdot 2^j}$ . For example,  $k_{0,3}$  allows decryption of records  $c_1, \dots, c_8$ , and  $k_{3,2}$  allows decryption of records  $c_{13}, \dots, c_{16}$ . If necessary, you may also change the encryption scheme  $(E^i, D^i)$  for each record  $i$ .

Solution:  $k_{i,j} = \underline{\hspace{2cm}}$ ,  
 $E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$ ,  
 $D_{k_i}^i(p_i) = \underline{\hspace{2cm}}$ .

**Exercise 2.52** (Modes vs. attack models.). For every mode of encryption we learned (see Table 2.4):

1. Is this mode always secure against any of the attack models we discussed (CTO, KPA, CPA, CCA)?
2. Assume this mode is secure against KPA. Is it then also secure against CTO? CPA? CCA?

3. Assume this mode is secure against CPA. Is it then also secure against CTO? KPA? CCA?

Justify your answers.

**Exercise 2.53.** Recall that WEP encryption is defined as:  $WEP_k(m; IV) = [IV, RC4_{IV,k} \oplus (m || CRC(m))]$ , where  $IV$  is a random 24-bit initialization window, and that  $CRC$  is a error-detection code which is linear, i.e.,  $CRC(m \oplus m') = CRC(m) \oplus CRC(m')$ . Also recall that WEP supports shared-key authentication mode, where the access point sends random challenge  $r$ , and the mobile response with  $WEP_k(r; IV)$ . Finally, recall that many WEP implementations use 40-bit key.

1. Explain how an attacker may efficiently find the 40-bit WEP key, by eavesdropping on the shared-key authentication messages between the mobile and the access point.
2. Present a hypothetical scenario where WEP would have used a fixed value of  $IV$  to respond to all shared-key authentication requests, say  $IV=0$ . Show another attack, that also finds the key using the shared-key authentication mechanism, but requires less time per attack. Hint: the attack may use (reasonable) precomputation process, as well as storage resources; and the attacker may send a ‘spoofed’ challenge which the client believes was sent by the access point.
3. Identify the attack models exploited in the two previous items: CTO, KPA, CPA or CCA?
4. Suppose now that WEP is deployed with a long key (typically 104 bits). Show another attack which will allow the attacker to decipher (at least part) of the encrypted traffic.

## Chapter 3

# Message Authentication Code (MAC) Schemes

Modern cryptography addresses different goals related to threats to information and communication accessible to an attacker. In this chapter, we focus on the goal of *authentication* of information and communication. Specifically, we discuss cryptographic schemes and protocols to detect when an attacker impersonates as somebody else, or modifies information from another agent.

Our discussion has two main parts. In the first part, we discuss *message authentication code (MAC) schemes*, which ensure that information was created by a known entity - without any modification. People often expect that encryption will ensure this property; we discuss the use of encryption for authentication, and show that this is quite tricky, although possible if done correctly. We also discuss how to combine MAC and encryption, to ensure both confidentiality and authenticity, and to improve system security.

In the second part, we discuss *entity authentication protocols*, which allow secure identification of communicating peer, usually by deploying MAC schemes. Authentication protocols are deceptively simple, and there have been many insecure designs - we will discuss some examples.

### 3.1 Encryption for Authentication?

As we discussed in previous chapter, encryption schemes ensure *confidentiality*, i.e., an attacker observing an encrypted message (ciphertext) cannot learn anything about the plaintext (except its length). Sometimes, people expect encryption to be *non-malleable*; intuitively, a non-malleable encryption scheme prevents the attacker from *modifying* the message in a ‘meaningful way’. See definition and secure constructions of non-malleable encryption schemes in [43]. However, be warned: achieving, and even defining, non-malleability is not as easy as it may seem!

In fact, many ciphers are *malleable*; often, an attacker can easily modify a known ciphertext  $c$ , to  $c' \neq c$  s.t.  $m' = D_k(c') \neq m$  (and also  $m' \neq \text{ERROR}$ ).

Furthermore, often the attacker can ensure useful relations between  $m'$  and  $m$ . An obvious example is when using the (unconditionally-secure) one-time-pad (OTP), as well as using Output-Feedback (OFB) mode.

**Example 3.1.** Suppose an attacker, Mal, eavesdrop on ciphertext  $c$  sent from Alice to her bank, where  $c$  is OTP-encryption of the plaintext message  $m = \text{'Transfer 10\$ to Bob. From: Alice. Password: 'IluvBob'.}'$ , encoded in ASCII. Show how Mal can modify the message so the bank will transfer money to his account rather than to Bob. How much money can Mal steal by sending the message?

Explain why your solution also works when using OFB mode encryption - or any PRG-based stream cipher.

We conclude that encryption schemes may not suffice to ensure authentication. This motivates us to introduce, in the next section, another symmetric-key cryptographic scheme, which is designed explicitly to ensure authentication and integrity: the *Message Authentication Code (MAC)*.

### 3.2 Message Authentication Code (MAC) schemes

Message Authentication Code (MAC) schemes are a simple, symmetric key cryptographic functions, designed to verify the authenticity and integrity of information (message). A MAC function  $MAC_k(m)$  has two inputs, a (secret)  $n$ -bit *secret (symmetric) key*  $k$ , and a message  $m$ ; note that there is no random inputs, i.e., MAC schemes are deterministic.

The output  $MAC_k(m)$  is often referred to as the *tag*. MAC functions are used to detect (unauthorized) messages or changes to messages. Intuitively, given  $m$ ,  $MAC_k(m)$  for a secret, random key  $k$ , it is hard for a (computationally-bounded) attacker to find another message  $m' \neq m$  together with the value of  $MAC_k(m')$ .

Typically, as shown in Fig. 3.1, a secret, symmetric MAC key  $k$  is shared between two (or more) parties. Each party can use the key to authenticate a message  $m$ , by computing an *authentication tag*  $MAC_k(m)$ . Given a message  $m$  together with a previously-computed tag  $T$ , a party verifies the authenticity of the message  $m$  by re-computing  $MAC_k(m)$  and comparing it to the tag  $T$ ; if equal, the message is valid, i.e., the tag must have been previously computed by the same party or another party, using the same secret key  $k$ .

In a typical use, one party, say Alice, sends a message  $m$  to a peer, say Bob, authenticating  $m$  by computing and attaching the tag  $T = MAC_k(m)$ . Bob confirms that  $T = MAC_k(m)$ , thereby validating that Alice sent the message, since he shares  $k$  only with Alice. See Fig. 3.1.

**Sender identification** Consider two - or more - parties, that use the same MAC key to send authenticated messages among them. By validating the tag received, recipients know that the tag was computed by one of the key holders - but not which key holder computed the MAC. Adding the identity of the

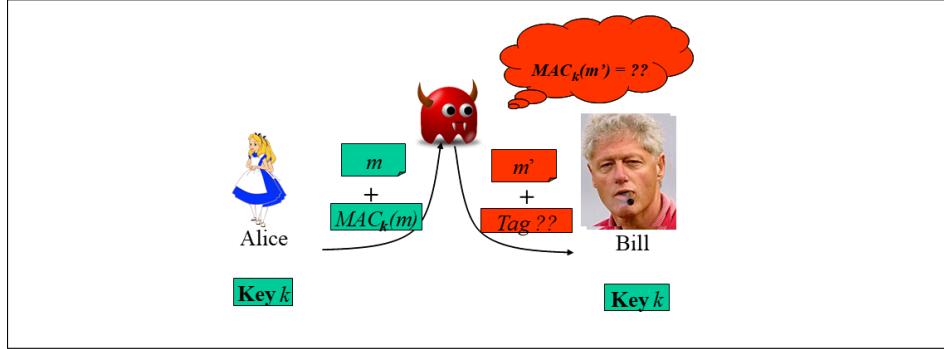


Figure 3.1: Message Authentication Code (MAC).

sender to the input to the MAC, in addition to the message itself, ensures correct identification of the sender, if all the parties are trusted to add their identity.

**Repudiation/deniability** To validate that a given tag  $T$  correctly validates a message  $m$ , i.e.,  $T = MAC_k(m)$ , requires the ability to compute  $MAC_k(\cdot)$ , i.e., knowledge of the secret key  $k$ . However, this implies the ability to compute (valid) tags from any other message. This allows the entity that computed the tag to later deny having done it, since it could have been computed also by other entities. We later discuss *digital signature schemes*, which use a secret key to compute the signature (tag), and a public key to validate it, which can be used to prevent senders from denying/reputating messages.

### 3.3 MAC: definitions and usage

A secure MAC scheme prevents an attacker from obtaining a correct MAC value  $F_k(m)$  for any message  $m$ , except by randomly guessing, or if the value was computed by a party knowing the key. Similar to the definitions of chosen-plaintext attack and of pseudo-random functions and permutations, we allow the adversary to obtain the MAC values for *any* other message. The formal definition follows; notice the  $ADV^{F_k(\cdot) \text{ except } m}$  notation, implying that the adversary can provide an arbitrary query message  $q$  and receive its MAC value,  $F_k(q)$ , as long as  $q \neq m$ .

**Definition 3.1** (MAC secure against forgery). *An  $n$ -bit Message Authentication Code (MAC) is a function*

$$F : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^n.$$

We say that a MAC  $F$  is secure (against forgery), if for all PPT algorithms  $ADV$  holds:

$$\Pr \left[ (m, F_k(m)) \leftarrow ADV^{F_k(\cdot) \text{ except } m} \right] < NEGL(n) \quad (3.1)$$

Where the probability is taken over the random coin tosses of  $ADV$  and the choice  $k \xleftarrow{\$} \{0,1\}^n$ .

Note that the definition assumed a fixed output length  $n$ , which implied that the adversary could simply guess the MAC value randomly, winning with probability  $2^{-n}$ , as reflected in Eq. 3.1. Usually, the output length is chosen to be long enough to make the probability of such guess negligible. The definition may be modified to allow output length  $l < n$ , by adding a  $2^{-l}$  element to the right-hand-side of Eq. (3.1).

### 3.3.1 Usage of MAC

MAC is a simple cryptographic mechanism, which is quite easy to use; however, it should be applied correctly - understanding its properties and not expecting it to provide other properties. We now discuss few aspects of the usage of MAC schemes, and give few examples of common mistakes.

**Confidentiality** We first note that MAC is a great tool to ensure integrity and authenticity - but does *not ensure confidentiality*. Namely,  $MAC_k(m)$  may expose information about the message  $m$ . This is sometimes overlooked by system designers; for example, early versions of the SSH protocol, used so-called ‘Encrypt and Authenticate’ method, where to protect message  $m$  the system sent  $E_k(m) \parallel MAC_k(m)$ ; one problem with this design is that  $MAC_k(m)$  may expose information about  $m$ .

Notice that while obviously confidentiality is not a goal of MAC schemes, one may hope that it is derived from the authentication property. To refute such false hopes, it is best to construct a counterexample - a very useful technique to prove that claims about cryptographic schemes are incorrect. The counter-examples are often very simple - and often involve ‘stupid’ or ‘strange’ designs, which are especially designed to meet the requirements of the cryptographic definitions - while demonstrating the falseness of the false assumptions. Here is an example showing that MAC schemes may expose the message.

**Example 3.2** (MAC does not ensure confidentiality.). *To show that MAC may not ensure confidentiality, we construct such a non-confidential MAC function  $F^{NM}$ . Our construction uses an arbitrary secure MAC scheme  $F$  (which may or may not ensure confidentiality). Specifically:*

$$F_k^{NM}(m) = F_k(m) \parallel LSb(m)$$

where  $LSb(m)$  is the least-significant bit of  $m$ . Surely,  $F^{NM}$  does not ensure confidentiality, since it exposes a bit of the message (we could have obviously exposed more bits - even all bits!).

On the other hand, we now show that  $F^{NM}$  is a secure MAC. Assume to the contrary, that there is some attacker  $A^{NM}$  that succeeds (with significant probability) against  $F^{NM}$ . We use  $A^{NM}$  to construct an attacker  $A$  that succeeds with the same probability against  $F$ . Attacker  $A$  works as follows:

1. When  $A^{NM}$  makes a query  $q$  to  $F^{NM}$ , then  $A$  makes the same query to  $F$ , receiving  $F_k(q)$ ; it then returns  $F_k^{NM}(q) = F_k(q) \parallel LSb(q)$ , as expected by  $A^{NM}$ .
2. When  $A^{NM}$  outputs its guess  $m, T$ , where  $T$  is its guess for  $MAC_k(m)$  and  $m$  was not used in any of  $A^{NM}$ 's queries, then  $A$  outputs the same guess.

*It follows that  $F^{NM}$  is a secure MAC if and only if  $F$  is a secure MAC.*  $\square$

We note that MAC functions are often constructed from block ciphers, which are also used for confidentiality - see constructions later on. This is one reason that people are sometimes misled into thinking MAC also ensures confidentiality. Similarly, it is sometimes incorrectly assumed that MAC ensures randomization. Such assumptions may hold for a particular MAC construction - but are not valid in general, and hence best avoided - unless necessary in a particular application, and then, designers should make sure that the extra properties hold for their particular choice. The following exercise is very similar to the example above, this time showing that MAC is not necessarily a PRF.

**Exercise 3.1** (Non-PRF MAC). *Show that a MAC function is not necessarily a Pseudo-Random Function (PRF).*

**Key separation** Another problem with the SSH ‘Encrypt and Authenticate’ design,  $E_k(m) \parallel MAC_k(m)$ , is the fact that the same key is used for both encryption and MAC. This can cause further vulnerability; an example is shown in the following simple exercise.

**Exercise 3.2.** *Show that the use of the same key for encryption and MAC in  $E_k(m) \parallel MAC_k(m)$ , can allow an attacker to succeed in forgery of messages - in addition to the potential loss of confidentiality shown above.*

In fact, this is a good example to the *principle of key separation*.

**Principle 9** (Key Separation). *Keys used for different purposes and cryptographic schemes, should be independent from each other - ideally, each chosen randomly; if necessary, pseudo-random derivation is also Ok.*

**Freshness and sender authentication** A valid MAC received with a message, shows that the message was properly authenticated by an entity holding the secret key. In many applications involving authentication, it is necessary to ensure further properties. We already commented above, that MAC does not ensure *sender authentication*, unless the design ensures that only the specific sender will compute MAC using the specific key over the given message. This is usually ensured by including the sender identity as part of the payload being signed, although another way to ensure this is for each sender to use its own authentication key.

Another important property is *freshness*, namely, ensuring that the message was not already handled previously. Again, to ensure freshness, the sender should include appropriate indication in the message, or use a different key. This is usually achieved by including, in the message, a timestamp, a counter or a random number ('nonce') selected by the party validating freshness. Each of these options has its drawbacks: need for synchronized clocks, need to keep a state, or need for the sender to receive the nonce from the recipient (additional interaction).

### 3.4 Approaches to Constructing MAC

In the next section, we discuss constructions of secure MAC from block ciphers and Pseudorandom Functions (PRFs). Before doing that, we dedicate this section to discuss other approaches to construct MAC schemes: direct design 'from scratch', using a robust combiner to combine candidate MAC functions, and constructions based on cryptographic hash functions.

#### 3.4.1 MAC design 'from scratch'

In this approach, we design a candidate MAC function from non-cryptographic operations, possibly using some problems which are considered computationally-hard. The security of such design is based on the failure of significant cryptanalysis efforts. However, following the cryptographic building block principle (principle 7), MAC functions are rarely designed 'from scratch'. We give an example of vulnerabilities in EDC-based MAC designs, and then discuss robust-combiners for MAC - a possible way to combine weakly-trusted designs.

**A (failed) attempt to construct MAC from EDC** Let us consider a specific design, which, intuitively, may look promising: constructing a MAC from (good) Error Detection Code (EDC), such as one of the good CRC schemes. EDC schemes are designed to ensure integrity - albeit, their design model assumes random errors, rather than intentional modifications. However, can we extend them, using a secret key, to provide also authentication?

Notice that in § 2.10 we showed that encryption of CRC may not suffice to ensure authentication. Still, this does not rule out their use for authentication by using a secret key in a different way, in particular, unrelated to encryption. In the following exercise, we show that two specific, natural constructions,  $MAC_k(m) = EDC(k||m)$  and  $MAC'_k(m) = EDC(m||k)$ , are insecure.

**Exercise 3.3** (Insecure EDC-based MACs). *Show that both  $MAC_k(m) = EDC(k||m)$  and  $MAC'_k(m) = EDC(m||k)$  are insecure, even when using a 'strong' EDC such as CRC.*

*Solution:* We first note that the insecurity is obvious, for simple EDCs such as bit-wise XOR of all bits of the message, and appending the result as an EDC. This weak EDC detect single bit errors, but fails to detect any error

involving an even number of bits. This holds equally well with and without a secret key, concatenated before or after the message.

Let us now show that these designs are insecure, also when using a ‘strong EDC’ such as CRC. Specifically, consider  $CRC-MAC_k(m) = CRC(k||m)$ ; we next show this is not a secure MAC.

Recall that the CRC function is linear, namely  $CRC(m \oplus m') = CRC(m) \oplus CRC(m')$ . Hence, for any message  $m$ ,

$$CRC-MAC_k(m) = CRC(k||m) \quad (3.2)$$

$$= CRC\left((0^{|k|}||m) \oplus (k||0^{|m|})\right) \quad (3.3)$$

$$= CRC(0^{|k|}||m) \oplus CRC(k||0^{|m|}) \quad (3.4)$$

$$= CRC-MAC_{0^{|k|}}(m) \oplus CRC-MAC_k(0^{|m|}) \quad (3.5)$$

Namely, to forge the MAC for any message  $m$ , the attacker makes a query for  $q = 0^{|m|}$ , and receives  $CRC-MAC_k(0^{|m|})$ . Adversary now computes:

$CRC-MAC_{0^{|k|}}(m) = CRC(0^{|k|}||m)$ , and finally computes  $CRC-MAC_k(m) = CRC-MAC_{0^{|k|}}(m) \oplus CRC-MAC_k(0^{|m|})$ .  $\square$

We conclude that ECD-based MACs are - as expected - insecure.

Note that in Exercise 3.3 above, the attack assumes that the attacker can obtain the MAC for the specific message (query)  $q = 0^{|m|}$ . Obtaining MAC for this specific (‘chosen’) message, may be infeasible in many scenarios, i.e., the attack may appear impractical. However, as the following exercise shows, it is quite easy to modify the attack, so that it works for *any* (‘known’) message for which the attacker can receive the MAC value.

**Exercise 3.4** (Realistic attack on CRC-MAC). *Show how an attacker can calculate  $CRC-MAC_k(m)$  for any message  $m$ , given the value  $CRC-MAC_k(m')$  for any message  $m'$  s.t.  $|m'| = |m|$ .*

*Guidance:* The attack is a slight modification of the one in Exercise 3.3, exploiting the linearity very much like in Eq. 3.2, except for using query  $q$  instead of the special message  $0^{|m|}$ , and in particular, computing  $CRC-MAC_{0^{|k|}}(q)$ . The remaining challenge is to select the query  $q$  so that  $CRC-MAC_k(m) = CRC-MAC_{0^{|k|}}(q) \oplus CRC-MAC_k(m')$ ; you can find the required value of  $q$  by essentially solving this equation, which, using the linearity of  $CRC$ , is actually a simple linear equation.  $\square$

### 3.4.2 Robust combiners for MAC

A robust combiner for MAC combines two (or more) candidate MAC functions, to create a new composite function, which is proven secure provided that one (or a sufficient number) of the underlying functions is secure. There is actually a very simple robust combiner for MAC schemes: *concatenation*, as we show in the following exercise (from [65]).

**Exercise 3.5.** *Show that concatenation is a robust combiner for MAC functions.*

*Solution:* Let  $F', F''$  be two candidate MAC schemes, and define  $F_{k',k''}(m) = F'_{k'}(m) \parallel F''_{k''}(m)$ . We should show that it suffices for either  $F'$  or  $F''$  is a secure MAC, for  $F$  to be a secure MAC scheme as well. Without loss of generality, assume  $F'$  is secure; and assume, to the contrary, that  $F$  is not a secure MAC. Namely, assume an attacker  $ADV$  that can output a pair  $m, F_{k',k''}$  (without making a query to receive MAC of  $m$ ). We use  $ADV$  to construct an adversary  $ADV'$  against  $F'$ .

Adversary  $ADV'$  operates by running  $ADV$ , as well as running selecting a key  $k''$  and running  $F''_{k''}(\cdot)$ . Whenever  $ADV$  makes a query  $q$ , then  $ADV$  makes the same query to the  $F'_{k'}(\cdot)$  oracle, to receive  $F'_{k'}(q)$ , and computes by itself  $F''_{k''}(q)$ , together resulting with the required response  $(F'_{k'}(q), F''_{k''}(q))$ .

When  $ADV$  finally returns with the pair  $(m, F_{k',k''}(m)) = (m, F'_{k'}(m) \parallel F''_{k''}(m))$ , then  $ADV'$  simply returns the pair  $(m, F'_{k'}(m))$ .  $\square$

However, concatenation is a rather inefficient construction for robust combiner of MAC schemes, since it results in duplication of the length of the output. The following exercise shows that exclusive-or is also a robust combiner for MAC - and since the output length is the same as of the component MAC schemes, it is efficient.

**Exercise 3.6.** Show that exclusive-or is a robust combiner for MAC functions. Namely, that  $MAC_{(k',k'')}(x) = MAC'_{k'}(x) \oplus MAC''_{k''}(x)$  is a secure MAC, if one or both of  $\{MAC', MAC''\}$  is a secure MAC.

*Guidance:* Use similar analysis to the one in Lemma 2.2.  $\square$

### 3.4.3 MAC constructions from cryptographic hash functions

Some of the most popular MAC constructions are based on *cryptographic hash functions*. The most important is probably the HMAC construction [10, 11], defined as:

$$HMAC_k(m) = h(k \oplus OPAD \parallel h(k \oplus IPAD \parallel m)) \quad (3.6)$$

Where OPAD, IPAD are fixed constant strings. We will discuss cryptographic hash functions only later, and then we will also discuss HMAC and some other MAC constructions based on cryptographic hash functions. At this point, we will only mention that some cryptographic hash functions are extremely efficient, and this efficiency can be mostly inherited by HMAC. For example, the Blake2b [3] cryptographic hash functions achieves speeds of over  $10^9$  bytes/second, using rather standard CPU (intel I5-6600 with 3310MHz clock).

## 3.5 CBC-MAC: $ln$ -bit Fixed-Input-Length MAC from $l$ -bit PRF/PRP

### 3.5.1 Defining generalized-MAC and FIL-MAC

Our discussion focuses on constructions of MAC functions from block ciphers. In fact, these construction do not require a block cipher - any pseudo-random

function or permutation suffices; however, following the cryptographic building blocks principle, block ciphers are normally used as the basic building block, and hence our discussion will mostly use the term ‘block cipher’ for the underlying scheme.

Before presenting the construction of a ‘regular, full’ MAC from a block cipher, we begin by defining and presenting a construction for the simpler case of a Fixed-Input-Length (FIL) MAC.

Let us first present a *generalization* of the definition of MAC presented earlier (Def. 3.1). The more general definition below, allows the MAC to be defined for an arbitrary input domain  $D$ , not necessarily the set  $\{0, 1\}^*$  of all binary strings.

**Definition 3.2** (Generalized definition of MAC). *Let  $D$  be a domain (set). An  $n$ -bit Message Authentication Code (MAC) over  $D$  is a function*

$$F : \{0, 1\}^* \times D \rightarrow \{0, 1\}^l$$

s.t. for all PPT algorithms  $ADV$  holds:

$$\Pr \left[ (m, F_k(m)) \leftarrow ADV^{F_k(\cdot | \text{except } m)} \right] < \text{NEGL}(n) \quad (3.7)$$

Where the probability is taken over the random coin tosses of  $ADV$  and the choice  $k \xleftarrow{\$} \{0, 1\}^n$ .

A  $n$ -bit Fixed-Input-Length (FIL) MAC is a MAC over domain  $\{0, 1\}^n$  for some input length  $n$ .

Note: we often refer simply to FIL MAC or to  $n$ -bit MAC, as shorter terms for  $n$ -bit FIL MAC.

**Every  $n$ -bit block cipher is an  $n$ -bit FIL MAC** The next lemma shows that every  $n$ -bit PRF or block cipher is also a FIL MAC for domain and range  $\{0, 1\}^n$ . The Theorem holds also for any PRP (and block cipher).

**Lemma 3.1** (Every  $n$ -bit PRF is a  $n$ -bit FIL MAC). *Every  $n$ -bit block cipher is also a FIL MAC over domain and range  $\{0, 1\}^n$ . Similarly, all PRFs and PRPs with domain and range  $\{0, 1\}^n$  are also FIL MAC (over domain and range  $\{0, 1\}^n$ ).*

*Proof:* Consider a randomly chosen function  $f$  over domain and range  $\{0, 1\}^n$ , namely  $f \xleftarrow{\$} \{\{0, 1\}^n \rightarrow \{0, 1\}^n\}$ . Recall from subsection 2.5.4 that the mapping  $f(m)$  is chosen uniformly from  $\{0, 1\}^n$ , independently of any other mappings  $\pi(q)|q \neq m$ . Hence,

$$\Pr_{f \xleftarrow{\$} \{\{0, 1\}^n \rightarrow \{0, 1\}^n\}} \left[ (m, \pi(m)) \leftarrow ADV^{f(\cdot | \text{except } m)} \right] \leq 2^{-n}$$

Note that we did not even limit  $ADV$  computationally.

Assume now that Eq. 3.7 does not hold for some PPT adversary  $\text{ADV}$  and some PRF  $F_k(\cdot)$ , i.e.,  $F$  is *not* a secure MAC. Namely, when running with oracle to  $F_k$ ,  $\text{ADV}$  would ‘win’ with probability significantly larger than  $2^{-n}$ , while when running with oracle to the random function  $f$  as above, it would ‘win’ with probability at most  $2^{-n}$ . Hence, given oracle access to either a random function  $f(\cdot)$  or to the pseudorandom function  $F_k(\cdot)$ , we can test for success of  $\text{ADV}$  and if it is significantly over  $2^{-n}$ , conclude that we were given the PRF, contradicting the assumption that  $F_k$  is a secure PRF.

The claim for PRP and block ciphers follows since in a polynomial set of queries of a random permutation, there is negligible probability of finding two values which map to the same value, hence it is impossible to efficiently distinguish between a random function and a random permutation. See Lemma 2.3.  $\square$

### 3.5.2 CBC-MAC

Lemma 3.1 shows that every  $l$ -bit block cipher (or PRF, PRP) is also an  $l$ -bit MAC; however, this is limited to  $l$ -bit inputs, i.e., where the input domain is a single block - i.e., the same domain as of the block cipher. How can we deal with longer messages? We first consider the (easier) case, where the input domain is a specific number of blocks, i.e., an  $(\eta \cdot n)$ -bit MAC, where  $\eta$  is a fixed integer.

Let us first consider some *insecure* constructions. First, consider performing MAC to each block independently, similar to the ECB-mode (§ 2.9). This would result in a long MAC - and, worse, will not prevent an attacker from obtaining MAC for a different message, by re-ordering or duplicating blocks!

Next, consider adding a counter to the input. This prevents the trivial attack - but not simple variants, as shown in the following exercise. For simplicity, the exercise is given for  $\eta = 2$  - and also has the disadvantage of a longer output tag.

**Exercise 3.7** (CTR-mode MAC is insecure). *Let  $E$  be a secure  $(n + 1)$ -bit block cipher, and define the following  $2n$ -bit domain function:  $F_k(m_0||m_1) = E_k(0||m_0)||E_k(1||m_1)$ . Present a counterexample showing that  $F$  is not a secure  $2n$ -bit MAC.*

**The CBC-MAC construction and its security** We next present the *CBC-MAC mode*, which is a widely used, standard construction of an  $(\eta \cdot n)$ -bit MAC, from an  $n$ -bit block cipher. The CBC-MAC mode, illustrated in Fig. 3.2, is a variant of the CBC mode used for encryption, see § 2.9. Given a block-cipher  $E$ , we define  $CBC - MAC^E$  as in Eq. 3.8; see illustrated in Fig. 3.2.

$$CBC - MAC_k^E(m_1||m_2||\dots||m_\eta) = \{c_0 \leftarrow 0^n; (i = 1 \dots \eta) c_i = E_k(m_i \oplus c_{i-1}); \text{output } c_\eta\} \quad (3.8)$$

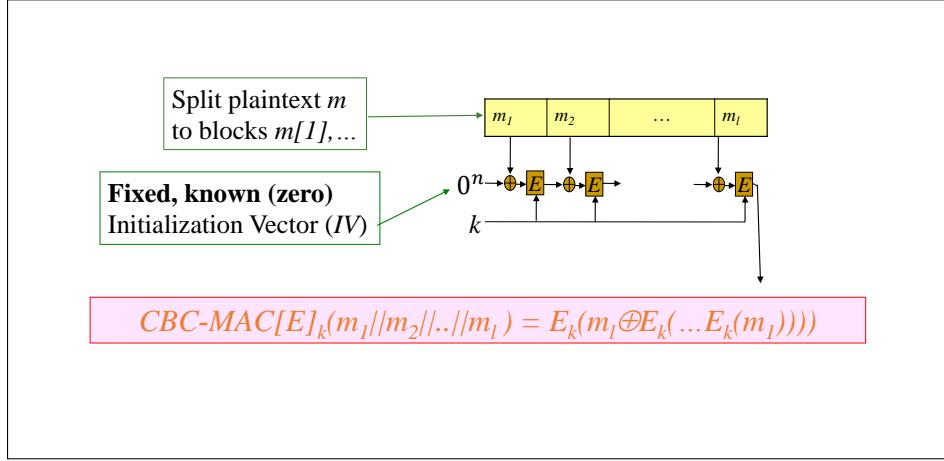


Figure 3.2: CBC-MAC: construction of MAC from block cipher.

When  $E$  is obvious we may simply write  $CBC - MAC_k(\cdot)$ .

Note: for simplicity, the construction is defined for input which is an integral number of blocks. Padding allows CBC-MAC to apply to input of arbitrary bit length.

There are other constructions of secure MAC from block ciphers, including more efficient constructions, e.g., supporting parallel computation. However, CBC-MAC is the most widely used MAC based on block ciphers, as also possibly the simplest, hence we focus on it.

We next present Lemma 3.2 which shows that CBC-MAC constructs a secure MAC, provided that the underlying ‘encryption’ function  $E$  is indeed a PRF. In fact, we observe that if  $E$  is a PRF, then CBC-MAC using  $E$  is also a PRF - therefore, from Lemma 3.1, CBC-MAC is also a secure MAC. Note that from Lemma 2.3, it follows that this holds also when  $E$  is a block cipher (or PRP).

**Lemma 3.2.** *If  $E$  is an  $n$ -bit PRF, PRP or block cipher, then  $CBC - MAC_k^E(\cdot)$  is a secure  $n \cdot \eta$ -bit PRF and MAC, for any constant integer  $\eta > 0$ .*

*Proof:* see in [13].

**CBC-MAC is *not* a VIL-MAC** The following simple exercise shows that while the CBC-MAC construction is a  $\eta \cdot n$ -bit FIL MAC, for any  $\eta$ , it is still not a  $\{0, 1\}^*$ -MAC, i.e., a VIL MAC.

**Exercise 3.8** (CBC-MAC is not VIL MAC). *Show that CBC-MAC is not a  $\{0, 1\}^*$ -MAC.*

*Solution:* Let  $f_k(\cdot) = CBC - MAC_k^E(\cdot)$  be the CBC-MAC using an underlying  $n$ -bit block cipher  $E_k$ . Namely, for a single-block message  $a \in \{0, 1\}^n$ , we

have  $f_k(a) = E_k(a)$ ; and for a two block message  $a||b$ , where  $a, b \in \{0, 1\}^n$ , we have  $f_k(a||b) = E_k(b \oplus E_k(a))$ .

We present a simple adversary  $A^{f_k}$ , with oracle access to  $f_k$ , i.e.,  $A$  is able to make arbitrary query  $x \in \{0, 1\}^*$  to  $f_k$  and receive the result  $f_k(x)$ . Let  $X$  denote all the queries made by  $A$  during its run. We show that  $A^{f_k}$  generates a pair  $x, f_k(x)$ , where  $x \notin X$ , which shows that  $f_k$  (i.e., CBC-MAC) is not a  $\{0, 1\}^*$ -MAC (i.e., VIL MAC).

Specifically, the adversary  $A$  first make an arbitrary single-block query, for arbitrary  $a \in \{0, 1\}^n$ . Let  $c$  denote the result, i.e.,  $c = f_k(a) = E_k(a)$ . Then,  $A$  computes  $b = a \oplus c$  and outputs the pair of message  $a||b$  and tag  $c$ .

Note that  $c = f_k(a||b)$ , since  $f_k(a||b) = E_k(b \oplus E_k(a)) = E_k((a \oplus c) \oplus c) = E_k(a) = c$ . Namely,  $c$  is indeed the correct tag for  $a||b$ . Obviously,  $A$  did not make a query to receive  $f_k(a||b)$ . Hence,  $A$  succeeds in the VIL MAC game against CBC-MAC.  $\square$

### 3.5.3 Constructing Secure VIL MAC

Lemma 3.2 shows that CBC-MAC is a secure  $\eta n$ -bit FIL PRF (and MAC); however, Exercise 3.8 shows that it is not a VIL PRF/MAC. The crux of the example was that we used the CBC-MAC of a one-block string, and presented it as the MAC of a 2-block string. This motivates a minor change to the construction, where we prepend the length of the input to the input before applying CBC-MAC. Lemma 3.3 shows that a small variant of this construction is indeed a secure VIL MAC. We refer to this variant as *length-prepended CBC-MAC*.

**Lemma 3.3.** *Let  $f_k(m) = \text{CBC-MAC}_k^E(\pi(m)||m)$ , where  $\pi(m)$  is a prefix-free encoding of  $|m|$  (the length of  $m$ ). Then  $f_k(\cdot)$  is a VIL PRF (and MAC).*

*Proof:* See [13].  $\square$

In practice,  $\pi(m)$  is usually implemented by simply prepending the length of  $m$ , encoded by a fixed-length field, e.g., 64 bits. Note that this encoding allows ‘only’ strings  $m$  of length up to  $2^{64}$ ; of course, this is long enough for any practical purpose. (To support even longer strings, use a prefix-free encoding of the length.)

The next exercise notes that some variants of this construction are not secure.

[AH: fix exercise below, and/or give solution, it isn’t reasonable as is. (readers: ignore it)]

**Exercise 3.9.** *Show that the following are not secure VIL PRF / MAC constructions.*

1.  $f_k(m) = \text{CBC-MAC}_k^E(m||\pi(m))$ , where  $\pi(m)$  is a prefix-free encoding of the length of  $m$ .

2.  $f_k(m) = \text{CBC} - \text{MAC}_k^E(\psi(m)||m)$  where  $\psi(m) = |m|$ . Note here you need to use the fact that  $|m|$  is not prefix free. Note: this is a tricky question.

Prepending the length to secure the CBC-MAC construction has the drawback that the length must be known in advance. A slightly more complex variant of CBC-MAC, called CMAC, avoids this requirement, and is defined as a NIST standard [47].

### 3.6 Combining Authentication and Encryption

Message authentication combines authentication (sender identification) and integrity (detection of modification). However, when transmitting messages, we often have additional goals. These include security goals such as confidentiality, as well as fault-tolerance goals such as error-detection/correction, and even efficiency goals such as compression. In this section, we focus on the combination of the two basic security goals: *encryption* and *authentication*. Later on, in subsection 5.1.4, we discuss the complete *secure session transmission* protocol, which addresses additional goals involving security, reliability and efficiency, for a session (connection) between two parties.

There are two main options for ensuring the confidentiality and authentication/integrity requirements together: (1) by correctly combining an encryption scheme with a MAC scheme, or (2) by using a combined *authenticated encryption* scheme. In the first subsection below, we discuss authenticated encryption schemes, including the security requirements and adversary model for the combination of confidentiality (encryption) and authentication. In the following subsections, we discuss specific generic constructions, combining arbitrary MAC and encryption schemes.

#### 3.6.1 Authenticated Encryption (AE) and AEAD schemes

Since the combination of confidentiality and authenticity is often required, there are also constructions of combined *Authenticated Encryption (AE)* schemes. AE schemes, like encryption schemes, consist of two functions: *encrypt-and-authenticate* and *decrypt-and-verify*. The decrypt-and-verify returns ERROR if the ciphertext is found not-authentic; similar verification property can be implemented by a MAC scheme, by comparing the ‘tag’ received with a message, to the result of computing the MAC on the message. AE schemes may also have a *key-generation function*; in particular, this is necessary when the keys are not uniformly random.

The use of such combined scheme allows simpler, less error-prone implementations, with calls to only one function (encrypt-and-authenticate or decrypt-and-verify) instead of requiring the correct use of both encryption/decryption and MAC functions. Many constructions are *generic*, i.e., built by combining arbitrary implementation of cryptographic schemes, following the ‘cryptographic building blocks’ principle. The combinations of encryption scheme and

MAC scheme that we study later in this subsection, are good examples for such generic constructions.

Other constructions are ‘ad-hoc’, i.e., they are designed using specific functions. Such ad-hoc construction may have better performance than generic constructions, however, that may come at the cost of requiring more complex or less well-tested security assumptions, in contrary to the Cryptographic Building Blocks principle.

In many applications, some of the data to be authenticated, should not be used also by agents which do not have the secret (decryption) key; for example, the identity of the destination. Such data is often referred to as *associated data*, and authenticated encryption schemes supporting it are referred to as *AEAD* (*Authenticated Encryption with Associated Data*) schemes [98]. AEAD schemes have the same three functions (key-generation, encrypt-and-authenticate, decrypt-and-verify); the change is merely in adding an optional ‘associated-data’ field as input to the encrypt-and-authenticate function and as output of the decrypt-and-verify function.

**Authenticated encryption: attack model and success/fail criteria**  
 We now briefly discuss the attack model (attacker capabilities) and the goals (success/fail criteria) for the combination of authentication and confidentiality (encryption), as is essential for any security evaluation (principle 1). Essentially, this combines the corresponding attack model and goals of encryption schemes (indistinguishability test) and of message authentication code (MAC) schemes (forgery test).

As in our definitions for encryption and MAC, we consider a computationally-limited (PPT) adversary. We also allow the attacker to have similar capabilities as in the definitions of secure encryption / MAC. In particular, we allow *chosen plaintext* queries, where the attacker provides input messages (plaintext) and receives their authenticated-encryption, as in the chosen-plaintext attack (CPA) we defined for encryption. We also allow *chosen ciphertext* queries, where the attacker provides ciphertext and receives back the decrypted plaintext, or error indication in case the ciphertext is found ‘invalid’; this combines the requirement of resistance to forgery from the MAC definition, with the security of encryption against chosen-ciphertext attack. We may also consider the weaker *feedback-only CCA* attack model, where the adversary is only given only the error indication (or no-error indication), but not given the actual decryption.

**Exercise 3.10.** Present precise definitions for IND-CPA and security against forgery for AE and AEAD schemes.

### 3.6.2 EDC-then-Encrypt Schemes

Several practical secure communication systems first apply an Error-Detecting-Code (EDC) to the message, and then encrypt it, i.e.:  $c = E_k(m||EDC(m))$ .

We believe that the motivation for this design is the hope to ensure authentication as well as confidentiality, i.e., the designers were (intuitively) trying to develop an authenticated-encryption scheme. Unfortunately, such designs are often insecure; in fact, often, the application of EDC/ECC before encryption, allows attacks on the confidentiality of the design. We saw one example, for WEP, in § 2.10. Another example of such vulnerability is in the design of GSM, which employs not just an Error Detecting Code but even an Error Correcting Code, with very high redundancy. In both WEP and GSM, the encryption was performed by XORing the plaintext (after EDC/ECC) with the keystream (output of PRG).

However, EDC-then-Encrypt schemes are often vulnerable, also when using other encryption schemes. For example, the following exercise shows such vulnerability, albeit against the authentication property, when using CBC-mode encryption.

**Exercise 3.11** (EDC-then-CBC does not ensure authentication). *Let  $E$  be a secure block cipher and let  $CBC_k^E(m; IV)$  be the CBC-mode encryption of plaintext message  $m$ , using underlying block cipher  $E$ , key  $k$  and initialization vector  $IV$ , as in Eq. (2.18). Furthermore, let  $EDCtCBC_k^E(m; IV) = CBC_k^E(m||h(m); IV)$  where  $h$  is a function outputting one block (error detecting code). Show that  $EDCtCBC^E$  is not a secure authenticated encryption; specifically, that authentication fails.*

*Hint:* attacker asks for  $EDCtCBC^E$  encryption of the message  $m' = m||h(m)$ ; the output gives also the encryption of  $m$ .  $\square$

**Hash-then-CBC is also insecure** Note that the above exercise applies equally to the case where  $h$  is a (cryptographic) hash function; actually, no property of  $h$  is used (except that its output is one block).

### 3.6.3 Generic Authenticated Encryption Constructions

We now discuss ‘generic’ constructions, combining arbitrary MAC and encryption schemes, to ensure both confidentiality or authentication/integrity. As discussed above, these constructions can be used to construct a single, combined ‘authenticated encryption’ scheme, or to ensure both goals (confidentiality and authenticity) in a system.

Different generic constructions were proposed - but not all are secure. Let us consider three constructions, all applied in important, standard applications. For each of the designs, we present the process of authenticating and encrypting a message  $m$ , using two keys -  $k'$  used for encryption, and  $k''$  used for authentication.

**Authenticate and Encrypt (A&E)** , e.g., used in early versions of the SSH protocol:  $C = Enc_{k'}(m)$ ,  $A = MAC_{k''}(m)$ ; send  $(C, A)$ .

**Authenticate then Encrypt (AtE)** , e.g., used in the SSL and TLS standards:  $A = MAC_{k''}(m)$ ,  $C = Enc_{k'}(m, A)$ ; send  $C$ .

**Encrypt then Authenticate (EtA)** , e.g., used by the IPsec standard:  $C = Enc_{k'}(m)$ ,  $A = MAC_{k''}(C)$ ; send  $(C, A)$ .

**Exercise 3.12** (Generic AE and AEAD schemes). *Above we described only the ‘encrypt-and-authenticate’ function of the authenticated-encryption schemes for the three generic constructions, and even that, we described informally, without the explicit implementation. Complete the description by writing explicitly, for each of the three generic constructions above, the implementation for the three functions: encrypt-and-authenticate (EnA), decrypt-and-verify (DnV) and key-generation (KG). Present also the AEAD (Authenticated Encryption with Associated Data) version.*

*Partial solution:* we present only the solution for the A&E construction. The AE implementations are:

$$\begin{aligned} A\&E.\text{EnA}_{(k', k'')}(m; r) &= (Enc_{k'}(m; r), MAC_{k''}(m)) \\ A\&E.\text{DnV}_{(k', k'')}(c, a) &= \begin{cases} \text{ERROR} & \text{if } Dec_{k'}(c) = \text{ERROR or } a \neq MAC_{k''}(c) \\ Dec_{k'}(c) & \text{otherwise} \end{cases} \\ A\&E.\text{KG}(r) &= \left( r \left[ 1 \dots \left\lfloor \frac{|r|}{2} \right\rfloor \right], r \left[ 1 + \left\lfloor \frac{|r|}{2} \right\rfloor, |r| \right] \right) \end{aligned}$$

The AEAD implementations are very similar, except also with Associated Data (*wAD*); we present only the EnA function:

$$A\&E.\text{EnAwAD}_{(k', k'')}(m, d; r) = (Enc_{k'}(m; r), d, MAC_{k''}(m||d))$$

□

Some of these three generic constructions are insecure, as we demonstrate below for particular pairs of encryption and MAC functions. Can you identify - or guess - which? The answers were given, almost concurrently, by two beautiful papers [14, 76]; the main points are in the following exercises.

Exercise 3.13 shows that A&E is insecure; this is quite straightforward, and hence readers should try to solve it alone before reading the solution.

**Exercise 3.13** (Authenticate and Encrypt (A&E) is insecure). *Show that a pair of secure encryption scheme  $\text{Enc}$  and secure MAC scheme  $\text{MAC}$  may be both secure, yet their combination using the A&E construction would be insecure.*

*Solution:* given any secure MAC scheme  $\text{MAC}$ , let

$$MAC'_{k''}(m) = MAC_{k''}(m)||m[1]$$

where  $m[1]$  is the first bit of  $m$ .

If  $MAC$  is a secure MAC then  $MAC'$  is also a secure MAC. However,  $MAC'$  exposes a bit of its input; hence, its use in  $A\&E$  would allow the adversary to distinguish between encryptions of two messages, i.e., the resulting, combined scheme is not IND-CPA secure - even when the underlying encryption scheme  $E$  is secure.  $\square$

Exercise 3.14 shows that AtE is also insecure. The argument is more elaborate than the A&E argument from Exercise 3.13, and it may not be completely necessary to understand it for a first reading; however, it is a nice example of a cryptographic counterexample, so it may be worth investing the effort. Readers may also consult [76] for more details.

**Exercise 3.14** (Authenticate then Encrypt (AtE) is insecure). *Show that a pair of secure encryption scheme  $Enc$  and secure MAC scheme  $MAC$  may be both secure, yet their combination using the AtE construction would be insecure.*

*Solution:* Consider the following simplified version of the Per-Block Random (PBR) mode presented in subsection 2.9.2, defined for single block messages:  $Enc_k(m; r) = m \oplus E_k(r) || r$ , where  $E$  is a block cipher; notice that this is also essentially OFB and CFB mode encryption, applied to single block messages. When the random bits are not relevant, i.e., simply selected uniformly, then we do not explicitly write them and use the simplified notation  $Enc_k(m)$ .

As shown in Theorem 2.1, if  $E$  is a secure block cipher (or even merely a PRF or PRP), then  $Enc$  is IND-CPA secure encryption scheme. Denote the block length by  $4n$ , i.e., assume it is a multiple of 4. Hence, the output of  $Enc$  is  $8n$ -bits long.

We next define a randomized transform  $Split : \{0,1\} \rightarrow \{0,1\}^2$ , i.e., from one bit to a pair of bits. The transform always maps 0 to 00, and randomly transforms 1 to  $\{01, 10, 11\}$  with the corresponding probabilities  $\{49.9\%, 50\%, 0.1\%\}$ . We extend the definition of  $Split$  to  $2b$ -bit long strings, by applying  $Split$  to each input block, i.e., given  $2n$ -bit input message  $m = m_1 || \dots || m_{2n}$ , where each  $m_i$  is a bit, let  $Split(m) = Split(m_1) || \dots || Split(m_{2n})$ .

We use  $Split$  to define a ‘weird’ variant of  $Enc$ , which we denote  $Enc'$ , defined as:  $Enc'_k(m) = Enc_k(Split(m))$ . The reader should confirm that, assuming  $E$  is a secure block cipher, then  $Enc'$  is IND-CPA secure encryption scheme (for  $2n$  bits long plaintexts).

Consider now  $AtE_{k,k'}(m) = Enc'_k(m || MAC_{k'}(m)) = Enc_k(Split(m || MAC_{k'}(m)))$ , where  $m$  is an  $n$ -bits long string, and where  $MAC$  has input and outputs of  $n$ -bits long strings. Hence, the input to  $Enc'$  is  $2n$ -bits long, and hence, the input to  $Enc$  is  $4n$ -bits long - as we defined above.

However,  $AtE$  is *not* a secure authenticated-encryption scheme. In fact, given  $c = AtE_{k,k'}(m)$ , we can decipher  $m$ , using merely feedback-only CCA queries.

Let us demonstrate how we find the first bit  $m_1$  of  $m$ . Denote the  $8n$  bits of  $c$  as  $c = c_1 || \dots || c_{8n}$ . Perform the query  $c' = \bar{c}_1 || \bar{c}_2 || c_3 || c_4 || c_5 || \dots || c_{8n}$ , i.e., inverting the first two bits of  $c$ . Recall that  $c = AtE_{k,k'}(m) = Enc_k(Split(m || MAC_{k'}(m)))$  and that  $Enc_k(m; r) = m \oplus E_k(r) || r$ . Hence, by inverting  $c_1, c_2$ , we invert the two bits of  $Split(m_1)$  upon decryption.

The impact depends on the value of  $m_1$ . If  $m_1 = 0$ , then  $\text{Split}(m_1) = 00$ ; by inverting them, we get 11, whose ‘unsplit’ transform returns 1 instead of 0, causing the MAC validation to fail, providing the attacker with an ‘ERROR’ feedback. However, if  $m_1 = 1$ , then  $\text{Split}(m_1)$  is either 01 or 10 (with probability 99.9%), and inverting both bits does not impact the ‘unsplit’ result, so that the MAC validation does not fail. This allows the attacker to determine the first bit  $m_1$ , with very small (0.1%) probability of error (in the rare case where  $\text{Split}(m_1)$  returned 11).  $\square$

Note that the AtE construction *is* secure - for specific encryption and MAC schemes. However, it is *not* secure for arbitrary secure encryption and MAC schemes, i.e., as a generic construction. Namely, Encrypt-then-Authenticate (EtA) is the only remaining candidate generic construction. Fortunately, *EtA is secure*, for any secure encryption and MAC scheme, as the following lemma states.

**Lemma 3.4** (EtA is secure [76]). *Given a CPA-IND encryption scheme  $\text{Enc}$  and a secure MAC scheme  $\text{MAC}$ , their EtA construction ensures both CPA-IND and secure MAC.*

*Proof sketch:* We first show that the IND-CPA property holds. Suppose, to the contrary, that there is an efficient (PPT) adversary  $ADV$  that ‘wins’ against EtA in the IND-CPA game, with significant probability. We construct adversary  $ADV'$  that ‘wins’ against the encryption scheme  $\text{Enc}$ , employed as part of the EtA scheme. Specifically,  $ADV'$  generates a key  $k''$  for the MAC function, and runs  $ADV$ . Whenever  $ADV$  asks for authenticated-encryption of some message  $m$ , then  $ADV'$  uses its oracle to compute  $c = \text{Enc}_{k'}(m)$ , and uses the key  $k''$  it generated to compute  $a = \text{MAC}_{k''}(c)$ .  $ADV'$  then returns the pair  $(c, a)$ , which is exactly the required  $\text{EtA}.\text{EnA}_{k', k''}(m)$ . When  $ADV$  guesses a bit  $b$ , then  $ADV'$  guesses the same bit. If  $ADV$  ‘wins’, i.e., correctly guesses, then  $ADV'$  also ‘wins’. It follows that there is no efficient (PPT) adversary  $ADV$  that ‘wins’ against EtA in the IND-CPA game.

We next show that EtA also ensures security against forgery, as in Def. 3.1, adjusted to for AE / AEAD schemes, as in Ex. 3.10. Suppose there is an efficient (PPT) adversary  $ADV$  that succeeds in forgery of the EtA scheme, with significant probability. Namely,  $ADV$  produces a message  $c$  and tag  $a$  s.t.  $m = \text{EtA}.\text{DnV}_{k', k''}(c, a)$ , for some message  $m$ , without making a query to  $\text{EtA}.\text{EnA}_{k', k''}(m)$ . By construction, this implies that  $a = \text{MAC}_{k''}(c)$ .

However, from the definition of encryption (Def. 2.1), specifically the correctness property, there is no other message  $m' \neq m$  whose encryption would result in same ciphertext  $c$ . Hence,  $ADV$  did not make a query for  $\text{MAC}_{k''}(c)$  - yet obtained it - in contradiction to the assumed security of MAC.  $\square$

*Additional properties of EtA: efficiency and foil DoS, CCA* Not only EtA is secure given any secure Encryption and MAC scheme, it also has three additional desirable properties:

**Efficiency:** Any corruption of the ciphertext, intentional or benign, is detected immediately by the verification process (comparing the received tag to

the MAC of the ciphertext). This is much more efficient than encryption.

**Foil DoS:** This improved efficiency, implies that it is much harder, and rarely feasible, to exhaust the resources of the recipient by sending corrupted messages (ciphertext).

**Foil CCA:** By validating the ciphertext before decrypting it, EtA schemes prevent CCA attacks, where the attacker sends ciphertext messages and uses the resulting plaintext and/or feedback to attack the underlying encryption scheme.

### 3.6.4 Single-Key Generic Authenticated-Encryption

All three constructions above used two separate keys:  $k'$  for encryption and  $k''$  for authentication. Sharing two separate keys may be harder than sharing a single key. Can we use a single key  $k$  for both the encryption and the MAC functions used in the generic authenticated encryption constructions (or, specifically, in the EtA construction, since it is always secure)? Note that this excludes the obvious naive ‘solution’ of using ‘double-length’ key, split to an encryption key and a MAC key. The following exercise shows that such ‘key re-use’ is insecure.

**Exercise 3.15** (Key re-use is insecure). *Let  $E'$ ,  $MAC'$  be secure encryption and MAC schemes. Show (contrived) examples of secure encryption and MAC schemes, built using  $E'$ ,  $MAC'$ , demonstrating vulnerabilities for each of the three generic constructions, when using the same key for authentication and for encryption.*

*Partial solution:*

**A&E:** Let  $E_{k',k''}(m) = E'_{k'}(m) || k''$  and  $MAC_{k',k''}(m) = k' || MAC'_{k''}(m)$ . Obviously, when combined using the A&E construction, the result is completely insecure - both authentication and confidentiality are completely lost.

**AtE:** To demonstrate loss of authenticity, let  $E_{k',k''}(m) = E'_{k'}(m) || k''$  as above.

**EtA:** To demonstrate loss of confidentiality, let  $MAC_{k',k''}(m) = k' || MAC'_{k''}(m)$  as above. To demonstrate loss of authentication, with a hint to one elegant solution: combine  $E_{k',k''}(m) = E'_{k'}(m) || k''$  as above, with a (simple) extension of Example 3.2.

The reader is encouraged to complete missing details, and in particular, to show that the all the encryption and MAC schemes used in the solution are secure (albeit contrived) - only their combined use, in the three generic constructions, is insecure.  $\square$

Since we know that we cannot re-use the same key for both encryption and MAC, the next question is - can we use two separate keys,  $k'$ ,  $k''$  from a single key  $k$ , and if so, how? We leave this as a (not too difficult) exercise.

**Exercise 3.16** (Generating two keys from one key). *Given a secure  $n$ -bit-key shared-key encryption scheme  $(E, D)$  and a secure  $n$ -bit-key MAC scheme  $MAC$ , and a single random, secret  $n$ -bit key  $k$ , show how we can derive two keys  $(k', k'')$  from  $k$ , s.t. EtA construction is secure, when using  $k'$  for encryption and  $k''$  for MAC, given:*

1. A secure  $n$ -bit-key PRF  $f$ .
2. A secure  $n$ -bit-key block cipher  $(\hat{E}, \hat{D})$ .
3. A secure PRG from  $n$  bits to  $2n$  bits.

### 3.7 Message Authentication: Additional exercises

**Exercise 3.17.** *Mal intercepts a message sent from Alice to her bank, and instructing the bank to transfer 10\$ to Bob. Assume that the communication is protected by OTP encryption, using a random key shared between Alice and her bank, and by including Alice's password as part of the plaintext, validated by the bank. Assume Mal knows that the message is an ASCII encoding of the exact string Transfer 10\$ to Bob. From: Alice, PW: xxxxx, except that xxx is replaced by Alice's password (unknown to Mal). Show how Mal can change the message so that upon receiving it, the bank will, instead, transfer 99\$ to Mal.*

**Exercise 3.18.** *Hackme Inc. proposes the following highly-efficient MAC, using two 64-bit keys  $k_1, k_2$ , for 64-bit blocks:  $MAC_{k_1, k_2}(m) = (m \oplus k_1) + k_2 (\text{mod } 2^{64})$ . Show that this is not a secure MAC.*

*Hint:* Compare to Exercise 2.39.

**Exercise 3.19.** *Let  $F : \{0, 1\}^n \rightarrow \{0, 1\}^l$  be a secure PRF, from  $n$  bit strings to  $l < n$  bit strings. Define  $F' : \{0, 1\}^n \rightarrow \{0, 1\}^l$  as:  $F'_k(m) = F_k(m) \parallel F_k(\bar{m})$ , i.e., concatenate the results of  $F_k$  applied to  $m$  and to the inverse of  $m$ . Present an efficient algorithm  $ADV^{F'_k}$  which demonstrates that  $F'$  is not a secure MAC, i.e., outputs tuple  $(x, t)$  s.t.  $x \in \{0, 1\}^n$  and  $t = F'_k(x)$ . Algorithm  $ADV^{F'_k}$  may provide input  $m \in \{0, 1\}^n$  and receive  $F'_k(m)$ , as long as  $x \neq m$ . You can present  $ADV^{F'_k}$  by 'filling in the blanks' in the 'template' below, modifying and/or extending the template if desired, or simply write your own code if you like.*

$ADV^{F'_k} : \{t' = F'_k(\underline{\hspace{1cm}})\};$

*Return*  $(\underline{\hspace{1cm}}); \}$

**Exercise 3.20.** *Consider CFB – MAC, defined below, similarly to the definition of CBC – MAC (Eq. (3.8)):*

$$CFB-MAC_k^E(m_1 \parallel m_2 \parallel \dots \parallel m_\eta) = \{c_0 \leftarrow 0^l; (i = 1 \dots \eta) c_i = m_i \oplus E_k(c_{i-1}); outputc_\eta\} \quad (3.9)$$

1. Show an attack demonstrating that  $CFB - MAC_k^E$  is not a secure  $l \cdot \eta$ -bit MAC, even when  $E$  is a secure  $l$ -bit block cipher (PRP). Your attack should consist of:
  - a) Up to three ‘queries’, i.e., messages  $m = \underline{\hspace{1cm}}$ ,  $m' = \underline{\hspace{1cm}}$  and  $m'' = \underline{\hspace{1cm}}$ , each of one or more blocks, to which the attacker receives  $CFB - MAC_k^E(m)$ ,  $CFB - MAC_k^E(m')$  and  $CFB - MAC_k^E(m'')$ . Note: actually, one query suffices.
  - b) A forgery, i.e., a pair of a message  $m^F = \underline{\hspace{1cm}}$  and its authenticator  $a = \underline{\hspace{1cm}}$  such that  $m^F \notin \{m, m', m''\}$  and  $a = CFB - MAC_k^E(m^F)$ .
2. Would your attack also work against the ‘improved’ variant  $ICFB - MAC_k^E(m) = E_K(CFB - MAC_k^E(m))$ ? If not, present an attack against  $ICFB - MAC_k^E(m)$ :  $m = \underline{\hspace{1cm}}$ ,  $m' = \underline{\hspace{1cm}}$ ,  $m'' = \underline{\hspace{1cm}}$ ,  $m^F = \underline{\hspace{1cm}}$  and  $a = \underline{\hspace{1cm}}$ .

**Exercise 3.21.** 1. Alice sends to Bob the 16-byte message ‘I love you Bobby’, where each character is encoded using one-byte (8 bits) ASCII encoding. Assume that the message is encrypted using the (64-bit) DES block cipher, using OFB mode. Show how an attacker can modify the ciphertext message to result with the encryption of ‘I hate you Bobby’.

2. Can you repeat for CFB mode? Show or explain why not.
3. Can you repeat for CBC mode? Show or explain why not.
4. Repeat previous items, if we append to the message its CRC, and verify it upon decryption.

**Exercise 3.22.** 1. Extend the definition of FIL MAC (in Definition 3.2), to allow MAC with minimal and maximal input length.

2. Show that CBC-MAC does not satisfy this definition, even if the input must be at least two blocks in length.

**Exercise 3.23.** 1. Our definition of FIL CBC-MAC assumed that the input is a complete number of blocks. Extend the construction to allow input of arbitrary length, and prove its security.

2. Repeat, for VIL CBC-MAC.

**Exercise 3.24.** Consider a variant of CBC-MAC, where the value of the IV is not a constant, but instead the value of the last plaintext block, i.e.:

$$CBC - MAC_k^E(m_1 || m_2 || \dots || m_\eta) = \{c_0 \leftarrow m_\eta; (i=1 \dots \eta) c_i = E_k(m_i \oplus c_{i-1}); output c_\eta\} \quad (3.10)$$

Is this a secure MAC? Prove or present convincing argument.

**Exercise 3.25.** Let  $E$  be a secure PRF. Show that the following are not secure MAC schemes.

1. ECB-encryption of the message.
2. The XOR of the output blocks of ECB-encryption of the message.

**Exercise 3.26** (MAC from a PRF). In Exercise 2.33 you were supposed to construct a PRF, with input, output and keyspace all of 64 bits. Show how to use such (candidate) PRF to construct a VIL MAC scheme.

**Exercise 3.27.** This question discuss a (slightly simplified) vulnerability in a recently proposed standard. The goal of the standard is to allow a server  $S$  to verify that a given input message was ‘approved’ by a series of filters,  $F_1, F_2, \dots, F_f$  (each filter validates certain aspects of the message). The server  $S$  shares a secret  $k_i$  with each filter  $F_i$ . To facilitate this verification, each message  $m$  is attached with a tag; the initial value of the tag is denoted  $T_0$  and each filter  $F_i$  receives the pair  $(m, T_{i-1})$  and, if it approves of the message, outputs the next tag  $T_i$ . The server  $s$  will receive the final pair  $(m, T_f)$  and use  $T_f$  to validate that the message was approved by all filters (in the given order).

A proposed implementation is as follows. The length of the tag would be the same as of the message and of all secrets  $k_i$ , and that the initial tag  $T_0$  would be set to the message  $m$ . Each filter  $F_i$  signals approval by setting  $T_i = T_{i-1} \oplus k_i$ . To validate, the server receives  $(m, T_f)$  and computes  $m' = T_f \oplus k_1 \oplus k_2 \oplus \dots \oplus k_f$ . The message is considered valid if  $m' = m$ .

1. Show that in the proposed implementation if the tag  $T_f$  is computed as planned (i.e. as described above), then the message is considered valid if and only if all filters approved of it.
2. Show that the proposed implementation is insecure.
3. Present a simple, efficient and secure alternative design for the validation process.
4. Present an improvement to your method, with much improved, good performance even when messages are very long (and having tag as long as the message is impractical).

Note: you may combine the solutions to the two last items; but separating the two is recommended, to avoid errors and minimize the impact of errors.

**Exercise 3.28** (Single-block authenticated encryption?). Let  $E$  be a block cipher (or PRP or PRF), for input domain  $\{0, 1\}^l$ , and let  $l' < l$ . For input domain  $m \in \{0, 1\}^{l-l'}$ , let  $f_k(m) = E_k(m||0^{l'})$ .

1. Prove or present counterexample:  $f$  is a secure MAC scheme.
2. Prove or present counterexample:  $f$  is an IND-CPA symmetric encryption scheme.

**Exercise 3.29.** Let  $F : \{0, 1\}^\kappa \times \{0, 1\}^{l+1} \rightarrow \{0, 1\}^{l+1}$  be a secure PRF, where  $\kappa$  is the key length, and both inputs and outputs are  $l + 1$  bits long. Let  $F' : \{0, 1\}^\kappa \times \{0, 1\}^{2l} \rightarrow \{0, 1\}^{2l+2}$  be defined as:  $F'_k(m_0 || m_1) = F_k(0 || m_0) || F_k(1 || m_1)$ , where  $|m_0| = |m_1| = l$ .

1. Explain why it is possible that  $F'$  would not be a secure  $2l$ -bit MAC.
2. Present an adversary and/or counter-example, showing  $F'$  is not a secure  $2l$ -bit MAC.
3. Assume that, indeed, it is possible for  $F'$  not to be a secure MAC. Could  $F'$  then be a secure PRF? Present a clear argument.

**Exercise 3.30.** Given a keyed function  $f_k(x)$ , show that if there is an efficient operation ADD such that  $f_k(x + y) = \text{ADD}(f_k(x), f_k(y))$ , then  $f$  is not a secure MAC scheme. Note: a special case is when  $\text{ADD}(a, b) = a + b$ .

**Exercise 3.31** (MAC from other block cipher modes). In § 3.5 we have seen given an  $n$ -bit block cipher  $(E, D)$ , the CBC-MAC, as defined in Eq. (3.8), is a secure  $n \cdot \eta$ -bit PRF and MAC, for any integer  $\eta > 0$ ; and in Ex. 3.7 we have seen this does not hold for CTR-mode MAC. Does this property hold for...

**ECB-MAC**, defined as:  $\text{ECB-MAC}_k^E(m_1 || \dots || m_\eta) = E_k(m_1) || \dots || E_k(m_\eta)$

**PBC-MAC**, defined as:  $\text{PBC-MAC}_k^E(m_1 || \dots || m_\eta) = m_1 \oplus E_k(1) || \dots || m_\eta \oplus E_k(\eta)$

**OFB-MAC**, defined as:  $\text{OFB-MAC}_k^E(m_1 || \dots || m_\eta) = \text{pad}_0, m_1 \oplus E_k(\text{pad}_0) || \dots || m_\eta \oplus E_k(\text{pad}_{\eta-1})$  where  $\text{pad}_0$  is random.

**CFB-MAC**, defined as:  $\text{CFB-MAC}_k^E(m_1 || \dots || m_\eta) = c_0, c_1, \dots, c_\eta$  where  $c_0$  is random and  $c_i = m_i \oplus E_k(c_{i-1})$  for  $i \geq 1$ .

**XOR-MAC**, defined as:  $\text{XOR-MAC}_k^E(m_1 || \dots || m_\eta) = \bigoplus E_k(i \oplus E_k(m_i))$

Justify your answers, by presenting counterexample (for incorrect claims) or by showing how an adversary against the MAC function, you construct an adversary against the block cipher.

### 3.8 Message Authentication: Additional exercises

**Exercise 3.32.** *Mal intercepts a message sent from Alice to her bank, and instructing the bank to transfer 10\$ to Bob. Assume that the communication is protected by OTP encryption, using a random key shared between Alice and her bank, and by including Alice's password as part of the plaintext, validated by the bank. Assume Mal knows that the message is an ASCII encoding of the exact string Transfer 10\$ to Bob. From: Alice, PW: xxxxx, except that xxx is replaced by Alice's password (unknown to Mal). Show how Mal can change the message so that upon receiving it, the bank will, instead, transfer 99\$ to Mal.*

**Exercise 3.33 (TBD).**

**Exercise 3.34.** *Let  $F : \{0, 1\}^n \rightarrow \{0, 1\}^l$  be a secure PRF, from  $n$  bit strings to  $l < n$  bit strings. Define  $F' : \{0, 1\}^n \rightarrow \{0, 1\}^l$  as:  $F'_k(m) = F_k(m) \parallel F_k(\bar{m})$ , i.e., concatenate the results of  $F_k$  applied to  $m$  and to the inverse of  $m$ . Present an efficient algorithm  $ADV^{F'_k}$  which demonstrates that  $F'$  is not a secure MAC, i.e., outputs tuple  $(x, t)$  s.t.  $x \in \{0, 1\}^n$  and  $t = F'_k(x)$ . Algorithm  $ADV^{F'_k}$  may provide input  $m \in \{0, 1\}^n$  and receive  $F'_k(m)$ , as long as  $x \neq m$ . You can present  $ADV^{F'_k}$  by 'filling in the blanks' in the 'template' below, modifying and/or extending the template if desired, or simply write your own code if you like.*

$ADV^{F'_k} : \{t' = F'_k(\underline{\hspace{10cm}})\};$

Return (                ); }

**Exercise 3.35.** *Consider CFB – MAC, defined below, similarly to the definition of CBC – MAC (Eq. (3.8)):*

$$CFB-MAC_k^E(m_1 \parallel m_2 \parallel \dots \parallel m_\eta) = \{c_0 \leftarrow 0^l; (i = 1 \dots \eta) c_i = m_i \oplus E_k(c_{i-1}); output c_\eta\} \quad (3.11)$$

1. *Show an attack demonstrating that  $CFB - MAC_k^E$  is not a secure  $l \cdot \eta$ -bit MAC, even when  $E$  is a secure  $l$ -bit block cipher (PRP). Your attack should consist of:*

- a) *Up to three 'queries', i.e., messages  $m = \underline{\hspace{10cm}}$ ,  $m' = \underline{\hspace{10cm}}$  and  $m'' = \underline{\hspace{10cm}}$ , each of one or more blocks, to which the attacker receives  $CFB - MAC_k^E(m)$ ,  $CFB - MAC_k^E(m')$  and  $CFB - MAC_k^E(m'')$ . Note: actually, one query suffices.*
- b) *A forgery, i.e., a pair of a message  $m^F = \underline{\hspace{10cm}}$  and its authenticator  $a = \underline{\hspace{10cm}}$  such that  $m^F \notin \{m, m', m''\}$  and  $a = CFB - MAC_k^E(m^F)$ .*

2. *Would your attack also work against the 'improved' variant  $ICFB - MAC_k^E(m) = E_K(CFB - MAC_k^E(m))$ ? If not, present an attack against  $ICFB - MAC_k^E(m)$ :  $m = \underline{\hspace{10cm}}$ ,  $m' = \underline{\hspace{10cm}}$ ,  $m'' = \underline{\hspace{10cm}}$ ,  $m^F = \underline{\hspace{10cm}}$  and  $a = \underline{\hspace{10cm}}$ .*

**Exercise 3.36.** 1. Alice sends to Bob the 16-byte message ‘I love you Bobby’, where each character is encoded using one-byte (8 bits) ASCII encoding. Assume that the message is encrypted using the (64-bit) DES block cipher, using OFB mode. Show how an attacker can modify the ciphertext message to result with the encryption of ‘I hate you Bobby’.

2. Can you repeat for CFB mode? Show or explain why not.
3. Can you repeat for CBC mode? Show or explain why not.
4. Repeat previous items, if we append to the message its CRC, and verify it upon decryption.

**Exercise 3.37.** 1. Extend the definition of FIL MAC (in Definition 3.2), to allow MAC with minimal and maximal input length.

2. Show that CBC-MAC does not satisfy this definition, even if the input must be at least two blocks in length.

**Exercise 3.38.** 1. Our definition of FIL CBC-MAC assumed that the input is a complete number of blocks. Extend the construction to allow input of arbitrary length, and prove its security.

2. Repeat, for VIL CBC-MAC.

**Exercise 3.39.** Consider a variant of CBC-MAC, where the value of the IV is not a constant, but instead the value of the last plaintext block, i.e.:

$$CBC-MAC_k^E(m_1||m_2||\dots||m_\eta) = \{c_0 \leftarrow m_\eta; (i=1\dots\eta)c_i = E_k(m_i \oplus c_{i-1}); output c_\eta\} \quad (3.12)$$

Is this a secure MAC? Prove or present convincing argument.

**Exercise 3.40.** Let  $E$  be a secure PRF. Show that the following are not secure MAC schemes.

1. ECB-encryption of the message.
2. The XOR of the output blocks of ECB-encryption of the message.

**Exercise 3.41** (MAC from a PRF). In Exercise 2.33 you were supposed to construct a PRF, with input, output and keyspace all of 64 bits. Show how to use such (candidate) PRF to construct a VIL MAC scheme.

**Exercise 3.42.** This question discuss a (slightly simplified) vulnerability in a recently proposed standard. The goal of the standard is to allow a server  $S$  to verify that a given input message was ‘approved’ by a series of filters,  $F_1, F_2, \dots, F_f$  (each filter validates certain aspects of the message). The server  $S$  shares a secret  $k_i$  with each filter  $F_i$ . To facilitate this verification, each message  $m$  is attached with a tag; the initial value of the tag is denoted  $T_0$  and each filter  $F_i$  receives the pair  $(m, T_{i-1})$  and, if it approves of the message,

outputs the next tag  $T_i$ . The server  $s$  will receive the final pair  $(m, T_f)$  and use  $T_f$  to validate that the message was approved by all filters (in the given order).

A proposed implementation is as follows. The length of the tag would be the same as of the message and of all secrets  $k_i$ , and that the initial tag  $T_0$  would be set to the message  $m$ . Each filter  $F_i$  signals approval by setting  $T_i = T_{i-1} \oplus k_i$ . To validate, the server receives  $(m, T_f)$  and computes  $m' = T_f \oplus k_1 \oplus k_2 \oplus \dots \oplus k_f$ . The message is considered valid if  $m' = m$ .

1. Show that in the proposed implementation if the tag  $T_f$  is computed as planned (i.e. as described above), then the message is considered valid if and only if all filters approved of it.
2. Show that the proposed implementation is insecure.
3. Present a simple, efficient and secure alternative design for the validation process.
4. Present an improvement to your method, with much improved, good performance even when messages are very long (and having tag as long as the message is impractical).

Note: you may combine the solutions to the two last items; but separating the two is recommended, to avoid errors and minimize the impact of errors.

**Exercise 3.43** (Single-block authenticated encryption?). Let  $E$  be a block cipher (or PRP or PRF), for input domain  $\{0,1\}^l$ , and let  $l' < l$ . For input domain  $m \in \{0,1\}^{l-l'}$ , let  $f_k(m) = E_k(m||0^{l'})$ .

1. Prove or present counterexample:  $f$  is a secure MAC scheme.
2. Prove or present counterexample:  $f$  is an IND-CPA symmetric encryption scheme.

**Exercise 3.44.** Let  $F : \{0,1\}^\kappa \times \{0,1\}^{l+1} \rightarrow \{0,1\}^{l+1}$  be a secure PRF, where  $\kappa$  is the key length, and both inputs and outputs are  $l+1$  bits long. Let  $F' : \{0,1\}^\kappa \times \{0,1\}^{2l} \rightarrow \{0,1\}^{2l+2}$  be defined as:  $F'_k(m_0||m_1) = F_k(0||m_0)||F_k(1||m_1)$ , where  $|m_0| = |m_1| = l$ .

1. Explain why it is possible that  $F'$  would not be a secure  $2l$ -bit MAC.
2. Present an adversary and/or counter-example, showing  $F'$  is not a secure  $2l$ -bit MAC.
3. Assume that, indeed, it is possible for  $F'$  not to be a secure MAC. Could  $F'$  then be a secure PRF? Present a clear argument.

**Exercise 3.45.** Given a keyed function  $f_k(x)$ , show that if there is an efficient operation ADD such that  $f_k(x + y) = \text{ADD}(f_k(x), f_k(y))$ , then  $f$  is not a secure MAC scheme. Note: a special case is when  $\text{ADD}(a, b) = a + b$ .

**Exercise 3.46** (MAC from other block cipher modes). *In § 3.5 we have seen given an  $n$ -bit block cipher  $(E, D)$ , the CBC-MAC, as defined in Eq. (3.8), is a secure  $n \cdot \eta$ -bit PRF and MAC, for any integer  $\eta > 0$ ; and in Ex. 3.7 we have seen this does not hold for CTR-mode MAC. Does this property hold for...*

**ECB-MAC**, defined as:  $ECB-MAC_k^E(m_1||\dots||m_\eta) = E_k(m_1)||\dots||E_k(m_\eta)$

**PBC-MAC**, defined as:  $PBC-MAC_k^E(m_1||\dots||m_\eta) = m_1 \oplus E_k(1)||\dots||m_\eta \oplus E_k(\eta)$

**OFB-MAC**, defined as:  $OFB-MAC_k^E(m_1||\dots||m_\eta) = pad_0, m_1 \oplus E_k(pad_0)||\dots||m_\eta \oplus E_k(pad_{\eta-1})$  where  $pad_0$  is random.

**CFB-MAC**, defined as:  $CFB-MAC_k^E(m_1||\dots||m_\eta) = c_0, c_1, \dots, c_\eta$  where  $c_0$  is random and  $c_i = m_i \oplus E_k(c_{i-1})$  for  $i \geq 1$ .

**XOR-MAC**, defined as:  $XOR-MAC_k^E(m_1||\dots||m_\eta) = \bigoplus E_k(i \oplus E_k(m_i))$

Justify your answers, by presenting counterexample (for incorrect claims) or by showing how an adversary against the MAC function, you construct an adversary against the block cipher.

## Chapter 4

# Cryptographic Hash Functions

*This chapter was recently moved to its current location (after MAC). Previously, it was located after public-key cryptosystems. There may still be some ‘bugs’ due to this re-ordering, I hope to find and fix these ‘real soon now’, and would appreciate if you point out any such ‘bug’ or any provide any other feedback/suggestion/correction.*

In this chapter, we discuss cryptographic hash functions. Cryptographic hash functions are of the most widely-used cryptographic schemes, with many diverse properties, uses and applications; in fact, in the previous chapter we mentioned one application: construction of a MAC scheme from crypto-hash function.

### 4.1 Introducing Crypto-hash functions, their goals and applications

A hash function  $h$  has an input which is called the *message* and denoted  $m$ , and its output  $h(m)$  is often referred to as the *digest*. Most applied hash functions are unkeyed, i.e., the digest is simply  $h(m)$ , as shown just now; but we also consider keyed hash functions,  $h_k(m)$ , where the key  $k$  is a random string, which is (usually) non-secret.

The basic property of hash functions is *compression*, i.e., the digest is shorter than the message. We adopt the common approach where the message is variable-length (VIL) binary string and the digest is a fixed length binary string, i.e., for unkeyed hash,  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ . For keyed hash, assume, for simplicity, that the key is of the same length  $n$  as the digest. Keyed hash is then defined as  $h : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ , with  $|h_k(m)| = |k|$ .

#### 4.1.1 Warmup: hashing for efficiency

Before we focus on crypto-hash functions, we first discuss briefly the use of hash functions for randomly mapping data, as used (also) for load-balancing

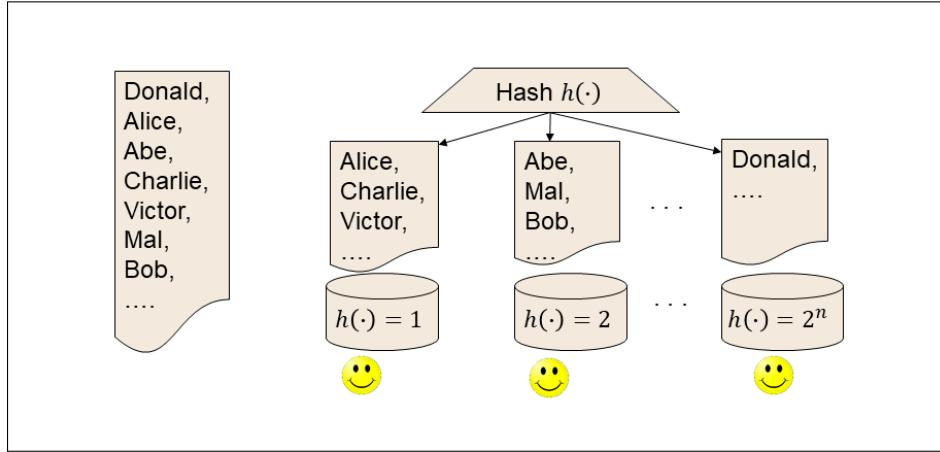


Figure 4.1: Typical load-balancing application of non-cryptographic hash function  $h$

and other ‘classical’, non-adversarial scenarios. Our goal is to provide intuition for the required security properties - and awareness of some of the challenges.

A common application of hash functions, including non-cryptographic hash functions, is to map the inputs into the  $2^n$  possible digest values (‘bins’) in a ‘random’ manner, i.e., with a roughly equal probability of assignment to each bin (digest value). This property is used in many algorithms and data structures, to improve efficiency and fairness. For non-security applications, the exact properties required from this ‘random’ mapping are often vague, or defined in terms of statistical tests that the function should pass.

A typical application of non-cryptographic hash functions is illustrated in Fig. 4.1. Here, a hash function  $h$  maps from the set of names (given as unbounded-length strings), to a smaller set, say the set of  $n$  bit binary strings. The goal of applying a hash function here, is *load balancing* of the number of entries assigned to each bin, i.e., that each ‘bin’ will be assigned roughly the same number of names; in particular, if the number of names mapped is much less than the number of bins ( $2^n$ ), we can expect very few or no *collisions*, i.e., two names mapped to the same bin. Such load-balancing is important for efficiency and fairness.

Of course, in cryptography, and cybersecurity in general, we mainly consider adversarial settings. In the context of load-balancing applications as shown in Fig. 4.1, this refers to an adversary who can manipulate some of the input names, and whose goal is to cause imbalanced allocation of names to bins, i.e., many collisions - which can cause bad performance.

Consider an attacker whose goal is to degrade the performance for particular name, say Bob. This attacker may provide to the system, a list of deviously-crafted names  $x_1, x_2, \dots$ , especially selected such that all of them ‘collide’ - i.e., are mapped to the same bin as Bob, i.e.,  $h(\text{Bob}) = h(x_1) = h(x_2) = \dots$ . The

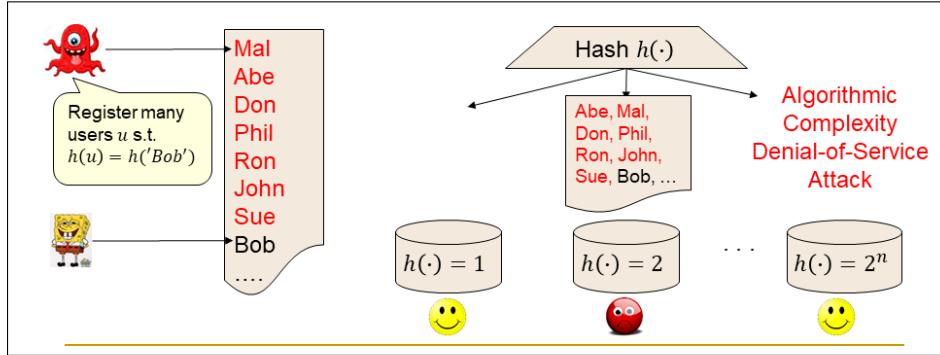


Figure 4.2: Algorithmic Complexity Denial-of-Service Attack exploiting insecure hash function  $h$ , to cause many collisions

attack is illustrated in Fig. 4.2. This method is sufficient to significantly impair the performance of many algorithms and systems, in a so-called *Algorithmic Complexity Denial-of-Service Attacks*. One way in which attackers may exploit such attack, is to cause excessive overhead for network security devices, such as malware/virus scanners, intrusion-detection systems (IDS) and intrusion-preventions systems (IPS), causing these systems to ‘give up’ and allowing attacks to penetrate undetected. We will discuss this and other denial-of-service attacks in the next part of these lecture notes, which focuses on network security.

Such vulnerability of ‘classical’ hash functions, which were not designed for security, is not surprising. Instead, it is indicative of the need for well-defined security requirements for (cryptographic) hash functions, following the *attack model and security requirements principle* (Principle 1). In particular, using a secure cryptographic hash functions, with the correct security property, should foil the Algorithmic Complexity Denial-of-Service Attack of Fig. 4.2. Specifically, each name that the adversary chooses, is mapped to a ‘random’ bin; only one in roughly  $2^n$  names will match the ‘target’, i.e.  $h(x_i) = h(\text{Bob})$ .

Note that some hash functions may seem to provide sufficiently-randomized mapping when the inputs are ‘natural’, but may still allow an attacker to easily select inputs that will hash to a specific bin (e.g., the one Bob is mapped to). See the following exercise.

**Exercise 4.1.** Given an alphabetic string  $x$ , let  $\text{num}(x, i)$  be the alphabetical-position of the  $i^{\text{th}}$  letter in  $x$ , e.g., if  $x = \text{'abcdef'}$ , then  $\text{num}(x, i) = i$ . Consider hash function  $h(x) = \sum_{i=1}^{|x|} \text{num}(x, i) \bmod 26$ , i.e., sum of all the letters (mod 26). Show how an attacker may easily generate a set of  $n$  strings,  $\{x_i\}_{i=1}^n$  s.t.  $(\forall i = 1, \dots, n) h(x_i) = h(\text{'bob'})$ , i.e., all these strings are mapped to the same bin  $h(\text{'bob'})$ . The attacker should not need to compute the hash value for many different strings. Give 3 examples of such strings. Note: the strings do not have to be ‘real names’ - any alphabetic string is allowed.

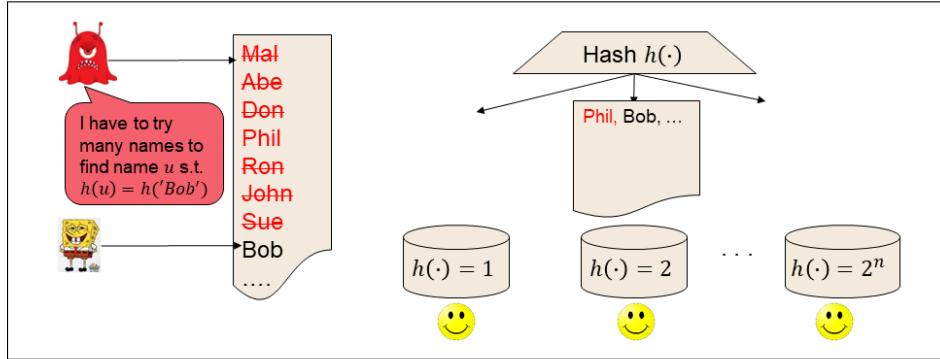


Figure 4.3: With a secure cryptographic hash function, the only way for the attacker to has a probability of  $2^{-n}$  to is unable to find inputs that will collide with legitimate input ‘Bob’

When the number of bins  $2^n$  is large enough, finding such a *collision* (match), with the given value ‘Bob’, becomes infeasible - assuming that the adversary can only randomly select values and test each of them by computing the hash. More precisely, given a random preimage of the hash function, in this case ‘Bob’, it should be infeasible for the attacker to find a *second preimage* (collision)  $m$  s.t.  $h(m) = h(\text{Bob})$ . This property is called *second preimage resistance (SPR)*. See Fig. 4.3.

However, relying on SPR requires large digest length  $n$ , so that  $2^n$  (number of bins) would very large; what can we do when the number of bins isn’t that large? We need to make it harder to *compute* the mapping  $h$ . A simple solution is to use a secret key  $k$ ; in this case, we essentially use a *pseudo-random function (PRF)*, or, equivalently, require the (keyed) hash function to (also) be a PRF.

When a secret key cannot be used, and computation of  $h$  must be possible to everyone, we can offer two solutions. First, we may restrict the choice of a name, i.e., allow only ‘authorized user names’ and restrict their distribution. Second, we may make it harder to compute  $h$  - using a *Proof-of-Work (PoW)* construction; see §4.6.

#### 4.1.2 Goals and requirements for crypto-hashing

The usefulness of hash functions stems from their diverse security properties. Roughly, these properties fall into three broad goals: *integrity*, i.e., ensuring uniqueness of the message; *confidentiality*, i.e., ‘hiding’ the contents of the message; and *randomness*, i.e., ensuring that unknown values (of input or output) are pseudorandom.

These goals are broad, and motivate different security requirements. Defining the security requirements for cryptographic hash functions, is quite tricky; in particular, there are several distinct requirements, required and motivated by the many different applications for cryptographic hash functions. Some

of the definitions may appear similar on first sight - but the differences are meaningful and critical.

Goal	Requirement	Abridged description
Integrity	Collision resistance (CRHF; Definition 4.1)	Can't find collision $(m, m')$ , i.e., $m \neq m'$ yet $h(m) = h(m')$ .
Integrity	Second-preimage resistance (SPR; Definition 4.4)	Can't find collision to random $m$ : $m' \neq m$ yet $h(m) = h(m')$ .
Confidentiality	One-way function (OWF; Definition 4.6)	Given $h(m)$ for random $m$ , can't find $m$ .
Randomness	Randomness extracting (Definition 4.7)	Choose $m$ except $n$ random bits; then output is pseudorandom.
All	Random oracle methodology (ROM; §4.6)	Consider $h$ as random function

Table 4.1: Goals and Requirements for unkeyed crypto-hash  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ . For keyed hash  $h : (\{0, 1\}^*)^2 \rightarrow \{0, 1\}^n$ , use  $h_k$  instead of  $h$ , with random key  $k$ . In SPR and OWF, the message  $m$  is random.

We discuss four security requirements, which we consider to be most important for applied cryptography: *collision resistance*, *one-way function* (also referred to as *preimage resistance*), *second-preimage resistance*, and *randomness extraction*. Table 4.1 maps the three goals to the corresponding requirements, with a brief description of each requirement (and reference to the relevant section). We also include the random oracle methodology (§ 4.6), where we analyze the security of a system using crypto-hash functions, as if we use a randomly-chosen function instead; the ROM is often adopted to allow analysis of designs using cryptographic hash functions, without identifying a feasible security requirement which ensures security of the design.

Here is an exercise which will strengthen your (still intuitive) understanding of the different security requirements in Table 4.1. A solution is provided for part (a); if you have difficulties solving part (b), try again after reading the following sections, which discuss each of these requirements in details.

**Exercise 4.2** (Insecure hash example). *Show that the following hash functions fail to provide any of the security properties defined above: (a)  $h(x) = x \bmod 2^n$ , (b)  $h'(x) = x^2 \bmod 2^n$ .*

*Solution for item (a):* We first show that  $h$  is not SPR (and hence surely not collision resistant). Namely, assume we are given a random input  $x$ ; let  $x' = x + 2^n$ . Clearly  $x' \neq x$ , and yet  $h(x') = (x + 2^n) \bmod 2^n = x \bmod 2^n = h(x)$ , namely,  $x'$  is a collision (second preimage) with  $x$ .

We next show that  $h$  is not one-way function (OWF). Specifically, given  $h(x)$  for any preimage  $x$ , let  $x' = h(x)$ ; clearly:

$$\begin{aligned} h(x') &= x' \bmod 2^n = h(x) \bmod 2^n \\ &= (x \bmod 2^n) \bmod 2^n = x \bmod 2^n = h(x) \end{aligned}$$

Application	Sufficient requirements
Integrity, hash-block, blockchain (§4.2)	Collision resistance
Hash-then-sign (§4.2.4)	Collision resistance
One-time-password (§4.4.1)	OWF
Password hashing, OTP-chain (9.1)	ROM
One-time signatures (§4.4.2)	OWF (for VIL also CRHF)
Proof-of-Work (§4.6)	ROM
Key/random generation (§4.5)	Randomness extracting; $\geq n$ random input bits
Random map (§4.1.1)	SPR (for large $n$ )

Table 4.2: Applications of crypto-hashing, and the corresponding requirements.

Namely,  $x'$  is a preimage of  $h(x)$ , and hence  $h$  is not a OWF.

Finally we show that  $h$  is not a randomness extracting hash function. Specifically, let  $x = r||0^n$ , where  $r$  is a random  $n$  bit string. Then  $h(x) = (r||0^n) \bmod 2^n = 0^n$ , which is obviously *not* a random string.  $\square$

#### 4.1.3 Applications of crypto-hash functions

The broad security requirements of cryptographic hash functions, facilitate their use in many systems and for an extensive variety of applications. These different applications and systems rely on different security requirements. As in any security system, it is important to identify the exact security requirements and assumptions; however, published designs and even standards, do not always identify the requirements, or state them imprecisely. Important applications of cryptographic hash functions, which we map to the corresponding necessary requirements in Table 4.2, include:

**Integrity, hash-block and blockchain** : the (short) digest  $h(m)$  allows validation of the integrity of the (long) message (or file)  $m$ . *hash-block* allows efficient digest of multiple messages (or files), with efficient validation of only specific message(s)/file(s), and with privacy - disclose messages only when required. *Blockchain* further extends, to ensure integrity for a sequence of multiple blocks; it is often combined with *Timestamping*, to allow validation of the creation time.

**Hash-then-Sign** : possibly the most well-known use of hash functions: facilitate signatures over long (VIL) documents. This uses the *hash-then-sign* paradigm (subsection 4.2.4), i.e., applying the RSA or other FIL signing function, to a digest  $h(m)$  of the message being signed  $m$ . See subsection 4.2.4.

**Login when server file may be compromised** : Hash functions are used to improve the security of password-based login authentication, in several ways. The most widely deployed method is using *hashed password file*,

which makes exposure of the server's password file less risky - since it contains only the hashed passwords. Another approach is to use hash-based *one-time password*, which is a random number allowing the server to authenticate user, with drawbacks of single-use and having to remember or have this random number. One-time passwords are improved into *OTP-chain* (aka *hash-chain*), to allow multiple login sessions with the same credential. See §9.1 and §4.4.1.

**Proof-of-Work** : cryptographic hash functions are often used to provide *Proof-of-Work (PoW)*, i.e., to prove that an entity performed considerable amount of computations. This is used by Bitcoin and other cryptocurrencies, and for other applications. See § 4.6.

**Key derivation and randomness generation** : hash functions are used to extract random, or pseudorandom, bits, given input with 'sufficient randomness'. In particular, this is used to derive secret shared keys. See §4.5.

**Random map** : as discussed above (§4.1.1), hash functions are often used, usually for efficiency, for 'random' mapping of inputs into 'bins', i.e., the  $2^n$  possible digests.

#### 4.1.4 Standard cryptographic hash functions

Due to their efficiency, simplicity and wide applicability, cryptographic hash functions are probably the most commonly-used 'cryptographic building blocks', as discussed in the cryptographic building blocks principle (Principle 7). This implies the importance of defining and adopting standard functions, which can be widely evaluated for security - mainly by cryptanalysis - and the need for definitions of security.

There have been many proposed cryptographic hash functions; however, since security is based on failed efforts for cryptanalysis, designers usually avoid less-well-known (and hence less tested) designs. These include the MD4 and MD5 functions proposed by RSA Inc., the SHA 1, 2 and 3 functions standardized by NIST, BLAKE2, RIPEMD and RIPEMD-160. Several of these, however, are rarely used in new designs, since they were shown to fail some of the requirements (discussed next). In particular, *collisions* were found for MD4, MD5, SHA-1 and RIPEMD.

Note that existing standards define only *keyless* cryptographic hash functions. However, as we later explain, there are strong motivations to use *keyed cryptographic hash functions*, which use a random, public key (without a private key). In particular, one of the security requirements, collision-resistance, cannot be achieved by any keyless function. We later discuss constructions of keyed hash functions from (standard) keyless hash functions.

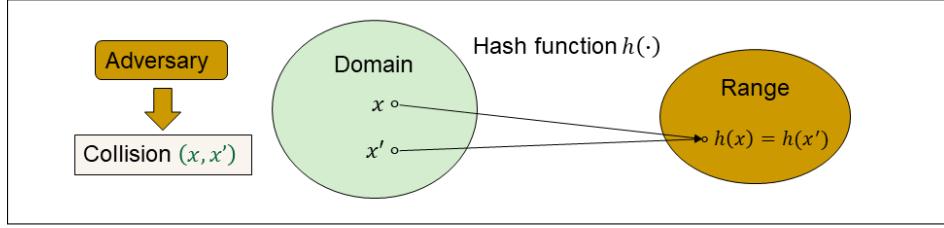


Figure 4.4: Keyless collision resistance: hard to find a collision, i.e., a pair of inputs  $x, x' \in \text{Domain}$ , which are mapped by hash function  $h$  to the same output,  $h(x) = h(x')$ .

## 4.2 Collision Resistant Hash Function (CRHF)

A hash function  $h(m)$  maps unbounded length binary strings  $m \in \{0, 1\}^*$ , to  $n$ -bit binary strings  $h(m) \in \{0, 1\}^n$ ; we often refer to  $h(m)$  as the *digest* of  $m$ . Hence, there are infinitely many *collisions*, i.e., messages  $m \neq m'$  s.t.  $h(m) = h(m')$ . However, *finding* such collisions may not be easy, when the domain of digests is large enough, i.e., for large  $n$ . Intuitively, we say that a hash function is *collision resistant*, if it is computationally-hard to find *any* collision, as illustrated in Figure 4.4. Definition follows.

**Definition 4.1** (Keyless Collision Resistant Hash Function (CRHF)). *A keyless hash function  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$  is collision-resistant if for every efficient (PPT) algorithm  $A$ , for sufficiently large  $n$ , the probability that  $A$ , on input  $1^n$ , will output a collision, is negligible (in  $n$ ). As a formula;*

$$\Pr [A(1^n) = (m, m') | m \neq m' \text{ yet } h(m) = h(m')] \in \text{NEGL}(n)$$

where the probability is over the coin tosses of the adversary  $A$ .

The definition for *keyed* CRHF seems very similar; the only difference is that the probability is also taken over the key, and the key is provided as input to the adversary (and the hash). Recall that, for simplicity,  $n$  is length of both the digest and the key. See Figure 4.5.

**Definition 4.2** (Keyed Collision Resistant Hash Function (CRHF)). *A keyed hash function  $h : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$  is collision-resistant if for every efficient (PPT) algorithm  $A$ , for sufficiently large  $n$ , the probability that  $A$ , given a random  $n$  bit key  $k \in \{0, 1\}^n$ , will output a collision ( $m \neq m'$ ) ( $h_k(m) = h_k(m')$ , is negligible (in  $n$ ). As a formula;*

$$\Pr [A(k) = (m, m') | m \neq m' \text{ yet } h_k(m) = h_k(m')] \in \text{NEGL}(n)$$

where the probability is over the coin tosses of the adversary  $A$  and the random choice of  $k$ .

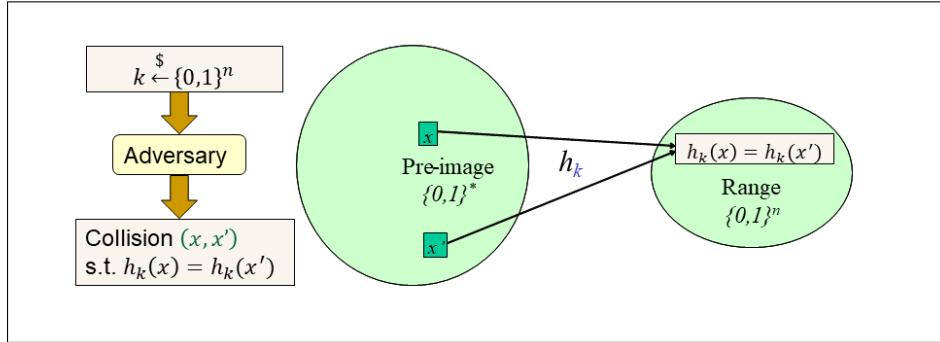


Figure 4.5: Keyed collision resistance hash function (CRHF): given random key  $k$ , it is hard to find a collision, i.e., a pair of inputs  $x, x' \in \text{Domain}$ , s.t.  $h_k(x) = h_k(x')$ .

Note that the order of the ‘event’ in the definition is important; if we first (randomly) select the key  $k$  and only then consider an arbitrary algorithm  $A$ , then the choice of  $A$  could depend on  $k$ . By selecting  $A$  first, we measure the performance of each adversary against a random key  $k$ .

### Using $n$ for digest and key length.

We defined unkeyed hash functions for a fixed digest length  $n$ ; but, published standards for hash functions, are for specific digest lengths, e.g., 160 bits. However, in Definition 4.1 - and the other security definitions we present later - we allow the adversary to run in time bounded by polynomial in  $n$ , and require the probability of success to be negligible - i.e., smaller than any polynomial in  $n$ .

Formally, the digest length  $n$  is a *parameter*, and the notation should be  $h^{(n)}(m)$ , so we can specify the digest length  $n$ ; we write  $h(m)$  instead, for simplicity, and since standard hash functions are defined for specific digest length (e.g., 160 bits). The implicit assumption is that the lengths used in practice, are ‘long enough’ to make attacks impractical; this is much like the use of fixed key length of cryptosystems such as AES and RSA, although the security definitions are stated with respect to schemes which support arbitrary-long keys.

For simplicity, when discussing keyed hash functions, we use  $n$  to denote the length of both the digest and the key. In reality, one may use keyed hash with different input and key length; however, for design reasons, it is actually quite common to use the same length for the key and for the digest.

## Why does the definitions limit run-time, and allows (negligible) probability of collision?

Both definition 4.2 and definition 4.1, may appear to be overly-permissive, in two ways: we restrict the adversary to run in polynomial time (PPT) in the length  $n$  of the digest, and we allow the adversary to have a negligible probability of finding a collision. However, without these two ‘relaxations’, the definition would not be feasible.

Let us first argue, that an adversary which can run in time exponential in  $n$ , would be able to find a collision. Consider a hash function  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ , and a set  $X$  containing  $2^n + 1$  distinct input binary strings. The output of  $h$  is the set of  $n$ -bits strings, which contains  $2^n$  elements; hence, there must be at least two elements  $x \neq x'$  in the set  $X$ , which collide, i.e.,  $h(x) = h(x')$ . An adversary that runs in time exponential in  $n$  can surely compute  $h(x)$  for every element in  $X$  and find this collision. Hence, the definitions restrict the adversary to run in time polynomial in  $n$ .

We next extend the argument and show, for any hash function  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ , a PPT algorithm (i.e., probabilistic algorithm that runs in time polynomial in  $n$ ) with non-zero probability to find a collision. Consider the same set  $X$  and before, and an algorithm that selects two random elements in  $X$ ; with small probability, this algorithm would output the collision  $x \neq x'$  s.t.  $h(x) = h(x')$ . Therefore, the definitions allow the adversary to have negligible probability of finding a collision.

We expressed the arguments above for a keyless hash  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ , but they obviously hold also for keyed hash.

We next show, that finding collision actually requires only  ${}^1O(\sqrt{2^n})$  attempts; this is due to the *birthday paradox*.

### 4.2.1 Birthday attack on collision resistance

The above argument presented an algorithm that finds a collision, by computing at most  $2^n + 1$  hash values. However, the expected number of hash-computations required to find a collision is only  $O(2^{n/2}) = O(\sqrt{2^n})$ , not  $O(2^n)$ . This is due to the *birthday paradox*: in a room containing 23 persons, the probability of a *collision*, i.e., two parties having birthday on the same date, is about half - much more than intuitively expected.

More precisely, the expected number  $q$  of messages  $\{m_1, m_2, \dots, m_q\}$  which should be hashed until finding a collision  $h(m_i) = h(m_j)$  is approximately:

$$q \lesssim 2^{n/2} \cdot \sqrt{\frac{\pi}{2}} \lesssim 1.254 \cdot 2^{n/2} \quad (4.1)$$

Hence, to ensure collision-resistance against adversary who can do  $2^q$  computations, e.g.,  $q = 80$  hash calculations, we need the digest length  $n$  to be roughly twice that size, e.g., 160 bits. Namely, the *effective key length* of a

---

<sup>1</sup>

CRHF is only  $q = n/2$ . This motivates the fact that hash functions often have digest length twice the key length of shared-key cryptosystems used in the same system. Of course, using longer digest length and/or longer key length should not harm security; and in fact, as mentioned above, designs of keyed hash functions often use the same length  $n$  for both key and digest length.

#### 4.2.2 CRHF Applications (1): Integrity and Hash-Block

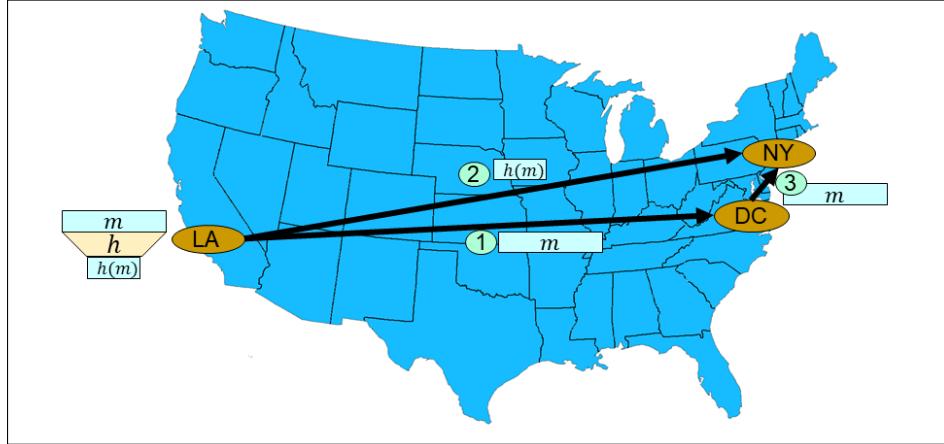


Figure 4.6: Example of use of hash function  $h$  to validate integrity of file  $m$  downloaded by a user in NY, from an untrusted repository in DC. To validate integrity, the user downloads the (short) digest directly from the website of the producer, in LA. This reduces network overhead - and load on the producer's website - compared to downloading the entire file from the producer's website.

Collision resistance is a great tool for ensuring integrity. One common application is to distribute a (large) object  $m$ , e.g., a file containing the executable code of a program. Suppose the file  $m$  is distributed from its producer in LA, to a user or repository in Washington DC (step 1 in Fig. 4.6). Next, a user in NY is downloading the file from the repository (or peer user) in DC (step 3), and, to validate the integrity, also the digest  $h(m)$  of the file, directly from the producer in LA (step 2). By downloading the large file  $m$  from (nearby) DC, the transmission costs are reduced; by checking integrity using the digest  $h(m)$ , we avoid the concern that the file was modified in DC or in transit between LA, DC and NY.

A potential remaining concern is modification of the digest  $h(m)$  received directly from producer in LA, by a *Monster-in-the-Middle (MitM)*<sup>2</sup> attacker. This may be addressed in different ways, including the use of secure web connection for retrieving  $h(m)$ , as discussed in the next chapter, and/or receiving the digest from multiple independent sources.

---

<sup>2</sup>Also called Man-in-the-Middle

This method is usually deployed manually, by savvy users. It may also be deployed automatically, by having the download and integrity-check done automatically by a script, received, with the hash, from the producer's site; see [55].

### Hash block.

In practice, large objects often consist of multiple smaller objects, which we call *files* or *messages*. For example, a typical software distribution may consist of multiple files, e.g.,  $(m_1, m_2, m_3, m_4)$ , such as when using large libraries. In such cases, it is often desirable to allow recipients to validate integrity only of specific files that they need, for efficiency, and possibly for privacy too. CRHF allow this to be done conveniently, by performing two levels of the hash operation: once over each file, and then, hash over all the resulting digests.

We refer to the large object composed of multiple files/messages as a *block*, and to the construction as a *hash-block*. More precisely, consider a 'block'  $B = (m_1, m_2, m_3, m_4)$  and a hash function  $h$ . The  $h$ -hash-block of  $B$  using  $h$ , denoted  $HB_h(B)$ , is the hash of the concatenation of the hashes of each file. For example, if  $B = (m_1, m_2, m_3, m_4)$ , then  $HB_h(B) = HB_h(m_1, m_2, m_3, m_4) = h[h(m_1)||h(m_2)||h(m_3)||h(m_4)]$ . See Figure 4.7.

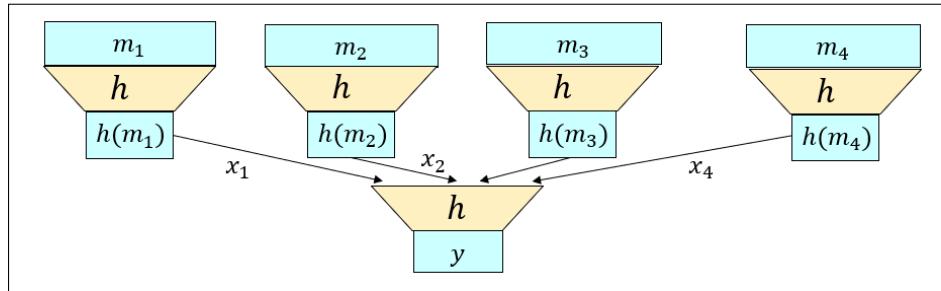


Figure 4.7: The  $h$ -hash-block  $HB_h(B)$  of a four-message block  $B = (m_1, m_2, m_3, m_4)$ . Let  $x_i = h(m_i)$  denote the hash of a single message  $m_i$  (for  $i \in \{1, 2, 3, 4\}$ ). The hash block  $HB_h(B)$  is computed as the hash of the concatenation of the hashes of each message, i.e.,  $HB_h(B) = HB_h(m_1, m_2, m_3, m_4) = h(x_1||x_2||x_3||x_4) = h[h(m_1)||h(m_2)||h(m_3)||h(m_4)]$ . The hash-block allows efficient validation, as illustrated in Figure 4.8.

The hash-block construction allows efficient validation of only one or few files, given the  $HB_h(B)$ , the digest of the entire block. For example, as illustrated in Figure 4.8, to validate  $m_2$ , it suffices to provide  $HB_h(B)$ ,  $m_2$  and the digests of the other files:  $h(m_1)$ ,  $h(m_3)$  and  $h(m_4)$ . Namely, to validate  $m_2$ , we compute  $h(m_2)$  and then validate that  $HB_b(B) = h[h(m_1)||h(m_2)||h(m_3)||h(m_4)]$ . Note that the digests of the other messages  $h(m_i)$  may be received from the (untrusted) repository; the CRHF property ensures we detect any incorrect values for  $m_2$  or any of the digests  $(h(m_1), h(m_3)$  and  $h(m_4))$ .

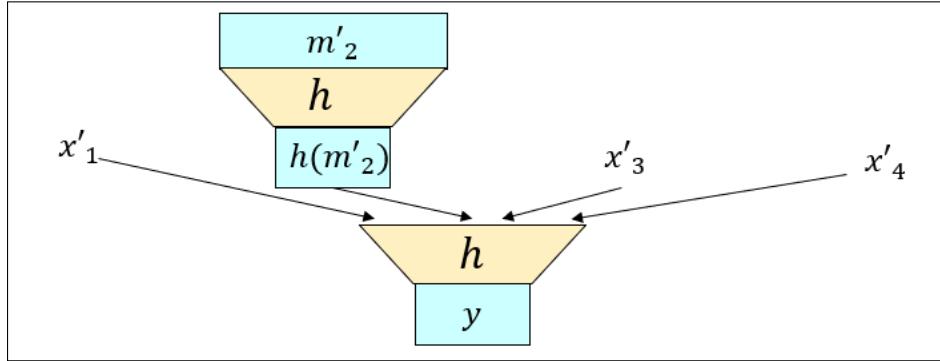


Figure 4.8: Hash-block: efficient validation of a specific message being part of the set of input messages, given the digest of the hash-block,  $y = HB_h(B)$ . For example, given  $m'_2$ , purported to be the same as the second message in the hashed block  $m_2$ , we validate it as follows. The validation requires the ‘helper values’  $x'_1, x'_3$  and  $x'_4$ , which are purported to be the hashes of the other messages in the block ( $m_1, m_3$  and  $m_4$ , respectively). The given  $m'_2$  is *valid* (i.e., equal to the original  $m_2$ ), if  $y = h(x'_1||h(m'_2)||x'_3||x'_4)$ . The validation process is often referred to as *Proof of Inclusion* of  $m'_2$ .

Note that while we present the design, above and in Figure 4.7, only for keyless CRHF, it works equally well - with minor, obvious modifications - also for a keyed CRHF.

**Exercise 4.3.** Define and illustrate the use of keyed CRHF  $h$  for hash-block.

The hash-block design allows better efficiency - no need to transmit, hash and store irrelevant files/messages. It is also useful for privacy, when some recipients should have access only to some files, e.g., if each file  $m_i$  contains data which is private to user  $i$ . Note, however, that the CRHF does not ensure confidentiality, i.e., the collision-resistance does not ensure that the value of  $h(m)$  will not expose some information about  $m$ . Namely, for such privacy property, the hash function  $h$  should also have additional properties, such as other properties we discuss later on for general-purpose cryptographic hash functions.

**Exercise 4.4.** Let  $h$  be a (keyed or keyless) CRHF. Use  $h$  to design another hash function  $g$ , s.t. (1)  $g$  is also a CRHF, yet (2)  $g$  exposes one or more bits of its input.

The hash-block can be viewed as a ‘single-level’ tree, with the leaves computing the digest (hash) of each file/message, and the root computing the digest of the entire block, by computing the hash of the digests of each file. In §4.7.1 we present the *Merkle hash-tree*, an extension of the hash-block construction, which provides even more efficient validation of a block of many messages, by

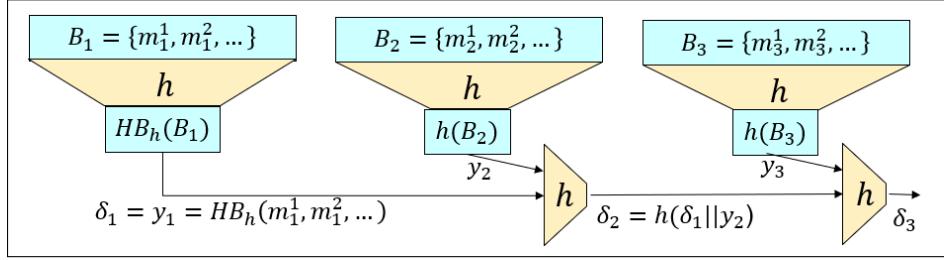


Figure 4.9: A blockchain: sequence of digests  $\delta_1, \delta_2, \dots$  of a sequence of blocks  $B_1 = \{m_1^1, m_1^2, \dots\}, B_2 = \{m_2^1, \dots\}, \dots$

using a tree with multiple levels, to reduce the amount of communication and storage for hashes of messages. But let us first, in the following subsection, present *blockchains*, an important cryptographic construction which can be applied either to hash-blocks or to Merkle hash trees.

#### 4.2.3 CRHF Applications (2): Blockchain

A (simplified) *blockchain* is an extension of the hash-block construction, in which we use a CRHF to hash *multiple blocks*, each containing multiple entries (files, transactions, ...), as illustrated in Figure 4.9. Namely, the blocks are:

$$B_1 = \{m_1^1, m_1^2, \dots\}, B_2 = \{m_2^1, m_2^2, \dots\}, B_3 = \{m_3^1, m_3^2, \dots\} \quad (4.2)$$

The blockchain is constructed incrementally, i.e., one block at a time. Let  $\delta_i = BC_h(B_1, \dots, B_i)$  denote the output of the blockchain, using hash function  $h$ , applied to the sequence of blocks  $B_1, \dots, B_i$ . This is computed incrementally, as illustrated in Figure 4.9, by applying the hash-block function to each block, i.e.:

$$\delta_i = BC_h(B_1, \dots, B_i) = \begin{cases} HB_h(B_1) & \text{if } i = 1 \\ h(HB_h(B_i) || \delta_{i-1}) & \text{if } i > 1 \end{cases} \quad (4.3)$$

Note that to compute a digest, we only need the previous digest and the current block ; we do not need previous blocks. More importantly, this also holds for validation; to validate any block  $B_j$ , we need only  $B_j$ , the previous digest  $\delta_{j-1}$  and the hash of the following blocks,  $\delta_{j+1}, \dots, \delta_i$ , for some  $i > j$ . In the typical case, we only need to validate that a specific message/file, say  $m_j^2$ , is included in (a particular block in) the blockchain. In this case, we don't even need all of  $B_j$ ; it suffices to provide  $m_j^2$  itself, and the digest of the other entries:  $h(m_j^1), h(m_j^3), h(m_j^4), \dots$ . See Figure 4.10.

Another typical validation is that a new digest for the blockchain, say  $\delta_3$ , is a ‘valid continuation’, i.e., is *consistent*, with the last known digest of the blockchain, say  $\delta_1$ . In this case, the validation only requires the digests of the blocks added the last known digest (e.g.,  $\delta_1$ ) and until the new digest (e.g.  $\delta_3$ ). In this example, these hash-block digests would be  $y_2 = HB_h(B_2)$  and

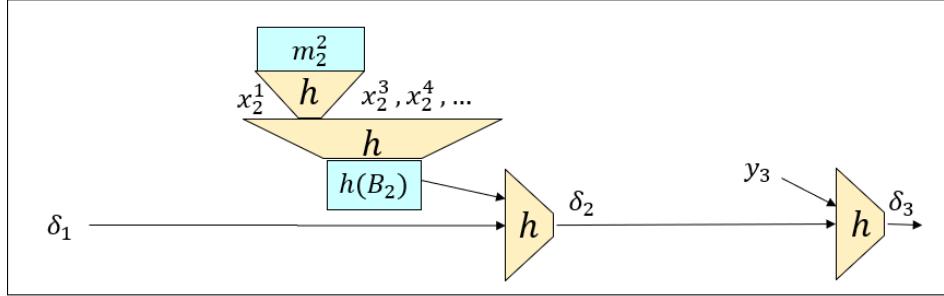


Figure 4.10: Proof-of-Inclusion of message  $m_2^2$  in a blockchain digest  $\delta_3$ .

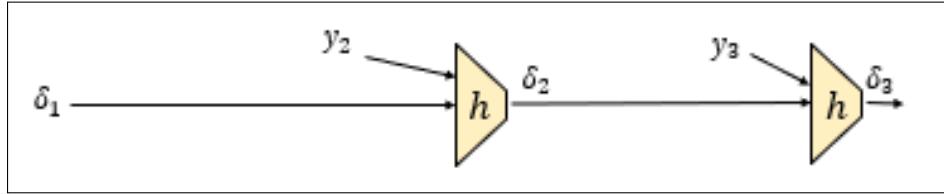


Figure 4.11: Proof-of-Consistency of a new blockchain digest,  $\delta_3$ , based on known digest,  $\delta_1$ , and the digests of blocks added after  $\delta_1$ , i.e.,  $y_2 = HB_h(B_2)$  and  $y_3 = HB_h(B_3)$ .

$y_3 = HB_h(B_3)$ . We often refer to such validation as a *Proof-of-Consistency* (*PoC*); see this example illustrated in ??.

Blockchains have many important and well-known applications requiring efficient authentication of a series of ‘blocks’ of multiple entries (messages, files, transactions...). The most notable application is Bitcoin and other cryptocurrencies. Most of these applications use *extensions* of the simplified blockchains presented above; we next discuss, briefly, the two main types of extensions: *Merkle-tree blockchains* (for better efficiency), and *controlled blockchains* (for crypto-currencies and other applications).

**Merkle-Tree Blockchains.** For simplicity, our description of blockchains above assumed that the digest  $y_i$  of each block  $B_i$  is computed using a hash-block function  $HB_h(B_i)$ . However, the same design may be applied to other methods of computing verifiable digest of a block of messages; indeed, in practice, blockchains often use other types of digests. Probably the most common design uses the Merkle-tree construction to compute the digest of each block; see §4.7.1.

**Controlled blockchains.** Our discussion of blockchains so far, did not involve any restrictions or controls over the blocks added to a blockchain. However, many applications, e.g. Bitcoin and other crypto-currencies, require a mechanism that controls the addition of new blocks to the blockchain; in fact, when people refer to ‘blockchains’, they usually a control mechanism. There is a variety of control mechanisms, but they broadly fall into one of the following

two categories:

**Permissioned blockchains**, where only specific, authorized parties can add a block to the blockchain. In a typical permissioned blockchain, every blockchain digest must be signed using the private signing key of (one or more) authorized parties.

**Permissionless blockchains**, where *any* party may, in principle, add a block to the blockchain; however, typically, there still exists some control mechanism that limits the issuing of new blocks. Bitcoin is the most well-known permissionless blockchain; in Bitcoin, issuing a new block requires a solution to a difficult computational problem, namely, a *Proof-of-Work* (*PoW*). Proof-of-work are often implemented by cryptographic hash functions. For more details of PoW and Bitcoin, see §4.6.1.

#### 4.2.4 CRHF Applications (3): The Hash-then-Sign (HtS) paradigm

Collision-resistance is a powerful property; in particular, it facilitates one of the most important applications of cryptographic hash functions - the *hash-then-sign (HtS) paradigm*. The hash-then-sign paradigm is essential for efficient deployment of public-key digital signatures, which we briefly introduced in chapter 1 (see Figure 1.2). We present constructions for signature schemes, based on cryptographic hash functions, later in this chapter, and other constructions in § 6.8; both use the hash-then-sign paradigm for efficiency.

Given signature scheme  $(S, V)$  whose inputs are of length  $l$  bits, and keyless hash  $h : \{0, 1\}^* \rightarrow \{0, 1\}^l$ , the hash-then-sign signature of any message  $m$  is defined as  $S_s^h(m) \equiv S_s(h(m))$ , where we use  $s$  for the private signing key of  $S$  (and of  $S_s^h$ ).

We now discuss how to sign arbitrary-length messages, by (efficiently) using a Fixed Input Length (FIL) public key signature scheme, say with  $n$  bits input, and a keyless CRHF. The solution is the *hash-then-sign (HtS) paradigm*, defined below and illustrated in Figure 4.12.

**Definition 4.3** (Hash-then-Sign for keyless hash function  $h$ ). *Let  $(S, V)$  be a (FIL) public key signature scheme, defined for messages of length  $l$  bits, and let  $h : \{0, 1\}^* \rightarrow \{0, 1\}^l$ . Let:*

$$\begin{aligned} S_s^h(m) &\equiv (S_s(h(m)), m) \\ V_v^h(\sigma, m) &\equiv \{m \text{ if } h(m) = V_v(\sigma, h(m)), \text{ error otherwise}\} \end{aligned}$$

*We say that  $(S^h, V^h)$  is the hash-then-sign signature scheme, constructed from  $(S, V)$  and  $h$ .*

The hash-then-sign efficiently uses FIL signatures, with very short input length (rarely over 250 bytes), to supports long, variable-length (VIL) messages. We next show that provided that  $h$  is a Collision-Resistant Hash Function (CRHF), and that the FIL signature scheme  $(S, V)$  is secure, then  $(S^h, V^h)$  is secure, too.

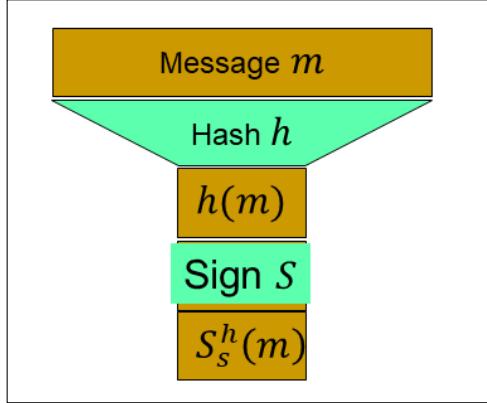


Figure 4.12: The Hash-then-Sign paradigm: given signature scheme  $(S, V)$ , where inputs are  $n$ -bit strings  $m \in \{0, 1\}^n$ , and a keyless CRHF  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ , the *hash-then-sign* signature function is  $S_s^h(m) = S_s(h(m))$ , where  $s$  is the signature key.

We have:

**Theorem 4.1** (Keyless Hash-then-Sign is secure). *Let  $(S, V)$  be a secure signature scheme with inputs of length  $l$ , and let  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$  be a keyless CRHF function. Let  $(S^h, V^h)$  be their hash-then-sign signature, with signature defined as:  $S_s^h(m) \equiv S_s(h(m))$ . Then  $(S^h, V^h)$  is a secure signature scheme.*

*Proof:* [TBD]

In practice, hash-then-sign is often used as in Theorem 4.1, i.e., with keyless hash  $S_s^h(m) \equiv S_s(h(m))$ . However, this design has an obvious drawback: we know for certain, that no keyless hash function can satisfy the CRHF requirement - as we show in the next subsection. One way to address this is to use the hash-then-sign paradigm, but with a *keyed* CRHF.

**Theorem 4.2** (Keyed Hash-then-Sign is secure). *Let  $(S, V)$  be a secure signature scheme with inputs of length  $n$ , and  $h : \{0, 1\}^* \times \{0, 1\}^n \rightarrow \{0, 1\}^n$  be a keyed CRHF function. Let  $(S^h, V^h)$  be their hash-then-sign signature, with signature defined as:*

$$S_s^h(m) \equiv \left\{ k \xleftarrow{\$} \{0, 1\}^n ; \text{ Output } (k, S_s(h_k(m))) \right\} \quad (4.4)$$

*Then  $(S^h, V^h)$  is a secure signature scheme.*

*Proof:* [TBD]

**Exercise 4.5.** *A system uses keyed Hash-then-Sign signatures. However, for efficiency, a specific single key for the hash function is selected by each party, initially, and provided with the signature public key of the user.*

*Show a keyed hash function where such design would be vulnerable, and an attack exploiting this vulnerability to cause the system to sign a collision, i.e., the signature would validate two different messages.*

#### 4.2.5 Keyless CRHF do not exist - yet are useful ?!

We see that collision resistance, as defined above, is a powerful property for hash functions, and, in particular, allows secure use of hash functions, e.g., in the Hash-then-Sign paradigm. However, when considering keyless hash functions, collision resistance has a critical drawback: *it cannot be realized: there exists no secure keyless CRHF*, according to the definition above. In fact, for any given keyless hash function  $h$ , there is a trivial, efficient adversarial algorithm  $A$ , that outputs a collisions for  $h$ , i.e., a pair  $(m, m')$  s.t.  $m \neq m'$  but  $h(m) = h(m')$ . Let us explain.

We first note that, since the domain of  $h$  is unbounded while the range is bounded, there are collisions, i.e., a pair of messages  $m \neq m'$  s.t.  $h(m) = h(m')$ . We define  $A$  as the following simple algorithm:  $A(1^n) = \{\text{print } m, m'\}$ . Namely,  $A$  simply prints out the collision. Yes, it isn't a 'real' algorithm, but hopefully, it serves to explain the issue: a keyless function must have some fixed set of collisions, therefore, it cannot be hard to find collisions.

If this concern seems 'only theoretical' and not convincing enough, consider this: most attacks against well-known, widely used (keyless) hash functions, were demonstrations of collisions, i.e., failure to achieve keyless collision resistance; in particular, these are still the only published attacks against MD5, SHA-1 and RIPE.

On the other hand, the simplicity of keyless hash functions still makes them convenient for initial design of protocols. Furthermore, we can still hope that, in practice, no such collision would be found for newer hash functions such as SHA-3 - at least, for many years. Hence, when analyzing security of an existing system, or in the (rare) case where there is a compelling performance reason to avoid a key for the hash, one may still use a keyless hash in some applications - with awareness of the potential vulnerability. In fact, many cryptographic designs use a much stronger simplification - the *random oracle model (ROM)*, see §4.6.

However, when it is possible, it is preferable to use a *keyed CRHF*, for which this argument does not hold, and which are believed to exist.

Another alternative is to design the application to only require the weaker *second-preimage resistance (SPR)* property, which we discuss next. However, care must be taken that SPR is really sufficient, as it seems quite common that people believe that the SPR property suffices - where it is insufficient, creating vulnerabilities.

### 4.3 Second-preimage resistance (SPR)

The second property we introduce is *second-preimage resistance (SPR)*, illustrated in Figure 4.13. Here, the adversary is given a specific, randomly-chosen

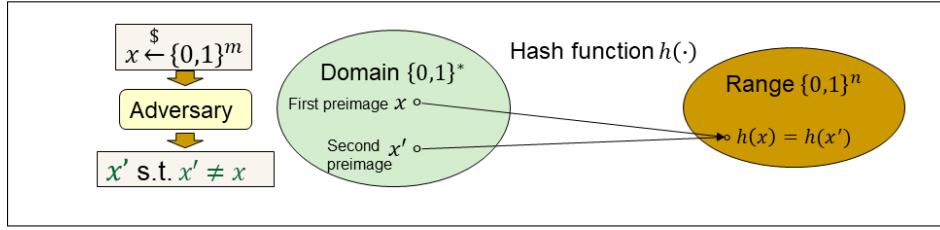


Figure 4.13: Second-preimage resistance (SPR): for sufficiently long input length  $m$ , given a random *first preimage*  $x \in \{0,1\}^m$ , it is hard to find a collision with  $x$ , i.e., a *second preimage*  $x' \in \{0,1\}^*$  s.t.  $x' \neq x$  yet  $h(x) = h(x')$ .

*first preimage*  $x \in \{0,1\}^m$ , for sufficiently-long length  $m$ , and ‘wins’ if it outputs a colliding *second preimage*, i.e.,  $x' \in \{0,1\}^*$  s.t.  $x' \neq x$  yet  $h(x') = h(x)$ . Note that in order to be able to select the preimage  $x$  uniformly, it has to be taken from a finite set; this is the reason for selecting it from the set of strings of sufficient length  $m$ .

**Definition 4.4** (Second-preimage resistance (SPR) Hash Function). A (keyless) hash function  $h : \{0,1\}^* \rightarrow \{0,1\}^n$  is second-preimage resistant (SPR) if for every efficient algorithm  $A \in PPT$ , for sufficiently long input length  $m$  holds:

$$\Pr_{x \xleftarrow{\$} \{0,1\}^m} [x' \leftarrow A(x); (x' \neq x) \wedge (h(x') = h(x))] \in \text{NEGL}(n) \quad (4.5)$$

SPR is sometimes referred to as *weak collision resistance*, and indeed, almost trivially, every CRHF is also a SPR. However, the reverse is not true, and in particular, it is widely believed that (keyless) SPR hash functions exist, while, as argued above, keyless CRHF cannot exist; and, in practice, collision attacks are known against SHA1 and MD5, but not second-preimage attacks. See the next exercise.

**Exercise 4.6.** Let  $h$  be a hash function. Assuming that  $h$  is a CRHF, show that it is also SPR. Next, show also another function,  $h'$ , derived from  $h$ , such that  $h'$  is also SPR, but not CRHF.

We can use a SPR for the ‘random mapping’ application, discussed in subsection 4.1.1. In this application, the goal is to prevent attacker from selecting many inputs to collide with an incoming message or name (e.g., with ‘Bob’). Of course, the attacker can try different values, and for each guess, has probability  $2^{-n}$  of being a collision; but this attack fails if we select sufficiently large digest size  $n$ . Hence, for this case, SPR is a sufficient requirement.

There are other applications which require only the SPR property and not the stronger collision-resistance property, e.g., see the following exercise. However, we usually prefer to use only keyless cryptographic hash functions for whom no collisions have been found, i.e., where CRHF ‘seems’ to hold (recall

we know that actually, no function can satisfy CRHF goal). In particular, this protects against the common case, where a designer incorrectly believes that an SPR hash suffices, while the system is actually vulnerable to non-SPR collision attacks. Importantly, SPR is not sufficient for Hash-and-Sign applications, as we discuss next.

**Exercise 4.7.** Identify an additional application, discussed earlier in this chapter, for which a SPR hash function suffices for security, and explain why the SPR property suffices for security of that application.

#### 4.3.1 The Chosen-Prefix Vulnerability and its HtS exploit

Theorem 4.1 shows that Hash-then-Sign is secure, when used with a CRHF. But would the weaker SPR property suffice? Even if the attacker can find *some* collision  $h(m) = h(m')$ , this is for some ‘random’ strings  $m, m'$  - how would the attacker convince the signer to sign  $m$ , and why should the alternative message  $m'$  be of (significant) value to the attacker? In short: is there a *realistic* attack, which may be possible against an SPR hash  $h$  (although it must fail against a CRHF)? We next show that this is indeed the case - by presenting such attack, exploiting the *chosen-prefix vulnerability*.

**Definition 4.5** (Chosen-prefix vulnerability). *Hash function  $h$  is said to suffer from the chosen-prefix vulnerability, if there is an efficient collision-finding algorithm  $CF$ , s.t. given any (prefix) string  $p \in \{0, 1\}^*$ , the algorithm  $CF$  efficiently output a (collision) pair of strings  $x, x' \in \{0, 1\}^*$ , s.t. for any (suffix) string  $s \in \{0, 1\}^*$  holds  $h(p||x||s) = h(p||x'||s)$ . Namely,*

$$(\exists CF \in PPT) (\forall p \in \{0, 1\}^*) [(x, x') \leftarrow CF(p)] (\forall s) h(p||x||s) = h(p||x'||s) \quad (4.6)$$

**Chosen-prefix attacks are practical.** The chosen-prefix vulnerability is a realistic concern; in fact, such vulnerabilities were found for widely-used standard hash functions such as MD5 and SHA1, which is a major reason to avoid these and use other cryptographic hash functions. We next show how the vulnerability facilitates a realistic attack on the Hash-then-Sign paradigm, allowing attacker to trick users into signing what appears to third party to be a statement (e.g., money transfer) that the user never intended to sign. A more elaborate attack allows also forgery of public key certificates signed using the MD5 hash function .

#### Chosen-prefix attack on Hash-then-Sign: simplified version

We begin by presenting a simplified version of the chosen-prefix attack on hash-then-sign paradigm. In this version, a Monster-in-the-Middle (MitM) attacker, say Mel, finds a collision  $(x, x')$  for the prefix string  $p = \text{'Pay'}$ . Assume that  $x$  is significantly less than  $x'$ , namely  $x << x'$ ; the attack will result in illegitimate profit of  $x' - x$  to Mel.

Mel should now convince a victim user, say Alice, to pay  $x\$$  to Mel, e.g., by selling it an item for this price. Alice generates the corresponding payment order, e.g.,  $m = \text{'Pay } x\$ \text{ to Mel'}$ . Next, Alice computes the hash  $y = h(m) = h(\text{'Pay } x\$ \text{ to Mel'})$ , signs it  $\sigma = S_{A.s}(y)$  and sends  $(m, \sigma)$  to the bank.

Recall that Mel has MitM capabilities, i.e., he is able to capture the message sent by Alice to her bank. Mel now modifies this message, so that the bank believes it received from Alice an order to pay to Mel a larger amount,  $x' >> x$  dollars. Specifically, Mel forwards to the bank  $(m', \sigma)$ , where  $m'$  is the modified message  $m' = \text{'Pay } x'\$ \text{ to Mel'}$ , and  $\sigma$  is Alice's signature  $\sigma = S_{A.s}(y)$  as computed and sent by Alice.

Note that  $m$  and  $m'$  share the prefix  $p = \text{'Pay '}$ , and the suffix 'to Mel'. Hence, by Definition 4.5, holds that  $y = h(m) = h(m')$ . Namely, when the bank verifies this incoming (modified) message  $m'$  which the bank received, using Alice's public verification key  $A.v$ , the validation would be fine: the  $\sigma$  is a valid signature of the digest  $y = h(m')$ . The bank therefore transfers  $x'$  dollars in to Mel's account.

However, this attack is yet simplified and not really realistic. In particular, the attacker has to find a collision with a significant difference between the two points,  $x << x'$ ; and, more significantly, people do not sign plain-text (ASCII) messages, but more readable versions such as PDF documents. We next discuss a more realistic variant of the attack, which works for PDF files.

### Realistic Chosen-prefix Attack: Signing PDF documents

We now improve the chosen-prefix attack, to allow forgery of signatures over documents, formatted in 'rich' markup languages like PDF, postscript, and HTML. The attacker, Mel, exploits the fact that these (and similar) languages, allow documents to contain conditional rendering statements, allowing the document to display different content depending on different conditions.

In the attack, Mel uses the conditional rendering capability, to create two documents  $D_1, D_M$  that have the same hash value,  $h(D_1) = h(D_M)$ , but when rendered by the correct viewer, e.g., PDF viewer, the two documents are rendered very differently. Namely, viewing  $D_1$ , the reader displays text  $t_1 = \text{'Pay 1\$ to Amazon'}$ , while viewing  $D_M$ , the reader displays text  $t_M = \text{'Pay one million dollars to Mel'}$ . The rest of the contents, and even the details of the markup language used, do not materially change the attack, so we ignore them.

Mel creates these two documents as follows. First, the documents share common prefix and suffix:  $D_1 = p||x_1||s$ ,  $D_M = p||x_M||s$ .

The prefix  $p$  consists of headers and preliminaries as required by the markup language, e.g., %PDF for PDF, or `<!DOCTYPE html>` for HTML, followed by the 'if' statement in the appropriate syntax. Simplifying, let's say that  $p = \text{if }$ .

Mel next applies the collision-finding algorithm  $CF$  (Definition 4.5), to find collision for prefix  $p$ , namely:  $(x_1, x_M) \leftarrow CF(p)$ . For every suffix  $s$  holds:  $h(p||x_1||s) = h(p||x_M||s)$ .

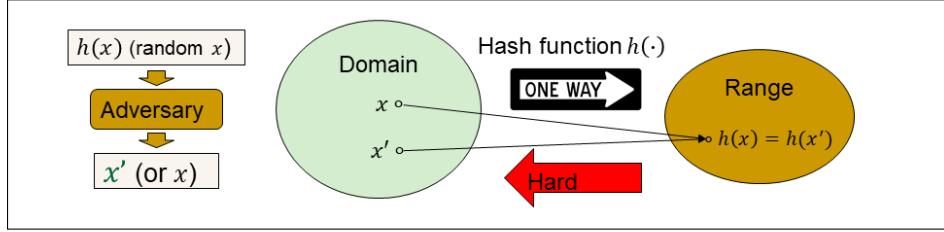


Figure 4.14: One-Way Function, aka Preimage-Resistant hash function: given a random preimage  $x$ , it is hard to find  $x$ , or any other preimage  $x'$  of  $h(x)$ , i.e., s.t..  $h(x') = h(x)$ .

To complete  $D_1, D_M$ , Mel sets the suffix  $s$  to:

$$s \leftarrow '=' || x_1 || \text{'then display'} || t_1 || ',', \text{else display'} || t_M$$

Mel is now ready to launch the attack on Alice, similarly to the simplified attack above. Namely, Mel first sends  $D_1$  to Alice, who views it, sees the rendering  $t_1$ . Let us assume that Alice agrees to pay 1\$ to Amazon, and hence signs  $D_1$ , i.e., computes  $\sigma = S_{A,s}(h(D_1))$  and sends  $(D_1, \sigma)$  to the bank.

Mel now uses his MitM capabilities, to capture  $(D_1, \sigma)$ , and forwards to the bank the modified message  $(D_M, \sigma)$ . The bank validates the signature, which would be Ok since  $h(D_1) = h(p||x_1||s) = h(p||x_M||s) = h(D_M)$ . The bank then views  $D_M$ , sees  $t_M = \text{'Pay one million dollars to Mel'}$ , and transfers one million dollars from Alice to Mel.

**Exercise 4.8.** Consider the hash function  $h(x_1||x_2||\dots||x_l) = \sum_{i=1}^l x_i \pmod p$ , where each  $x_i$  is 64 bits and  $p$  is a 64-bit prime. (a) Is  $h$  a SPR? CRHF? (b) Present a collision-finding algorithm  $CF$  for  $h$ . (c) Create two HTML files  $D_1, D_M$  as above, i.e., s.t.  $h(D_1) = h(D_M)$  yet they when viewed in browser, they display texts  $t_1, t_M$  as above.

#### 4.4 One-Way Functions, aka Preimage Resistance

The third security property we define is called *Preimage resistance* or *One-Way Function (OWF)*, and illustrated in Fig. 4.14. We prefer the term *one-way function*, that emphasizes the ‘one-way’ property: computing  $h(x)$  is easy, but ‘inverting’ it to find  $x$ , or a colliding preimage  $x'$  s.t.  $h(x') = h(x)$ , is hard.

**Definition 4.6** (Preimage resistance / One-Way Function). *An efficient function  $h : \{0,1\}^* \rightarrow \{0,1\}^*$  is called preimage resistant, or a one-way function, if for every efficient algorithm  $A \in PPT$ , and for sufficiently large  $m$  (length of input), holds:*

$$\Pr_{x \leftarrow \{0,1\}^m} [x' \leftarrow A(h(x)); (h(x') = h(x))] \in NEGL(m) \quad (4.7)$$

Note that the definition does not require one-way functions to be hash functions, i.e., we do not require bounded output length, or even that the output length be shorter than the input length. In fact, one particularly interesting case is *one-way permutation*, which is a OWF which is also a permutation.

The correct way to use a OWF is by ensuring that the input domain is random. We next present two related applications: *one-time password* and *one-time signature*.

#### 4.4.1 Using OWF for One-Time Passwords (OTP) and OTP-chain

A one-way function can be used to facilitate *one-time password* (*OTP*) authentication, allowing a user (say Alice) to prove to a server, say Bob, her identity, by Bob using a non-secret *validation token* for Alice.

To implement one-time passwords, Alice first selects a random string  $s$  to serve as the (secret) one-time password. Next, Alice computes the non-secret validation token as  $v = h(s)$ . Alice shares the validation token  $v$  with Bob (and possibly others -  $v$  is not secret!). Later, to prove her identity, or to make some other agreed-upon signal, Alice sends  $s$ . Bob can easily confirm that  $v = h(s)$ .

Unfortunately, the One-Way Function property - maybe due to the catchy name - is sometimes misunderstood. In particular, designs sometimes specify the use of a OWF - when, in fact, if the hash function is (only) a OWF, the design may be vulnerable.

In particular, consider *OTP-chain*, a widely-deployed generalization of one-time password authentication. In a OTP-chain, Alice pre-computes a whole ‘chain’ of values,  $x_i = h(x_{i-1})$ , beginning with a random value  $x_0$ , and till  $x_l$  for some  $l$  (the ‘chain length’). This is used to allow Alice to authenticate  $l$  times: in the  $i$ -th authentication, Alice sends  $x_{l-i}$ . Everyone who knows  $x_l$ , can easily validate, by applying  $h$ . OTP-Chain is often referred to as *hash-chain*, but we prefer the term OTP-chain, since the term hash-chain is easy to confuse with a blockchain (§4.2.3), which is a very different design.

Is OTP-chain authentication secure? Actually, if our only assumption is that  $h$  is a OWF, this is not correct. The problem is that for  $i > 0$ , the values  $x_i$  are not guaranteed to distribute uniformly; see example in next exercise. However, OTP-chain *is secure* - if the function is not only one-way, but also a permutation; see the following exercise.

**Exercise 4.9** (OTP-chain using OWF may be vulnerable). *Let  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ , and define  $g : \{0, 1\}^* \rightarrow \{0, 1\}^{2n}$  as:*

$$g(x) = \{0^{2n} \text{ if } x = 0 \pmod{2^n}, \text{ otherwise } h(x)||0^n\}$$

*Show that if  $h$  is a OWF, then  $g$  is also a OWF; yet, show that  $f(x) = g(g(x))$  is not a OWF, and in particular, that a OTP-chain using  $g$  is completely insecure.*

**Exercise 4.10.** *Show that if  $h$  is a OWF and a permutation over any given input length  $m$ , then OTP-chain using  $h$  is secure.*

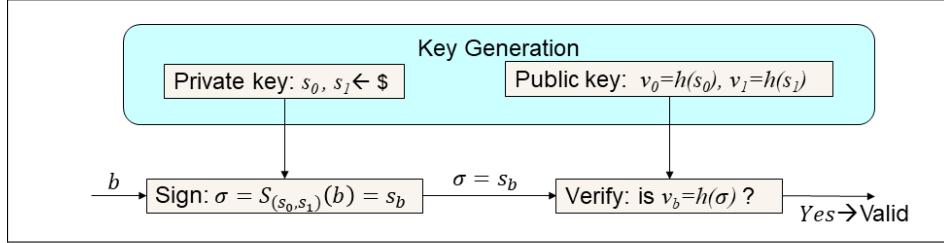


Figure 4.15: A one-time signature scheme, limited to a single bit (as well as to a single signature).

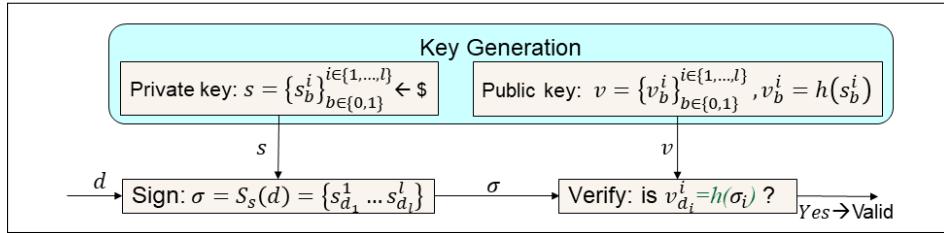


Figure 4.16: A one-time signature scheme, for  $l$ -bit string (denoted  $d$ ).

#### 4.4.2 Using OWF for One-Time Signatures

We next show how to use one-way functions to implement public key signatures - limited to signing only a single (one) time, i.e., a *one-time signature scheme*. In spite of their obvious limitation, one-time signatures are sometimes useful, due to their negligible computational overhead. Note, however, that they require rather long public keys and signatures.

We present the construction in three steps, each with a separate figure. Figure 4.15 presents a one-time signature scheme, which is defined only for the case of a single-bit message. This is a simple extension of one-time passwords. The private key  $s$  simply consists of two random strings  $s_0, s_1$ , and the public key - of the hash of these strings:  $v_0 = h(s_0), v_1 = h(s_1)$ . To sign a bit  $b$ , we simply send  $\sigma = s_b$ ; validate incoming bit  $b$  and signature  $\sigma$ , we validate that  $v_b = h(\sigma)$ . Note that  $v = (v_0, v_1)$  is not secret; only  $s = (s_0, s_1)$  is secret - and we can disclose  $s_b$  upon signing bit  $b$ .

We next extend the scheme, to allow one-time signature of an  $l$ -bit string  $d$ , as illustrated in Figure 4.16. This is simply application of the one-bit signature scheme of Figure 4.15, over each bit  $d_i$  of  $d$ , for  $i = 1, \dots, l$ .

Finally, we extend the scheme further, to efficiently sign arbitrary-length inputs (VIL) string  $m$ . This extension, illustrated in Figure 4.17, simply applies the *hash-then-sign* paradigm. Namely, we first use a CRHF, which we denote by  $g$  (since it does not have to be the same as  $h$ ), to compute the  $l$ -bit digest  $d$  of the message:  $d = g(m)$ . Then, we simply apply the scheme of Figure 4.16, to sign the digest  $d$ .

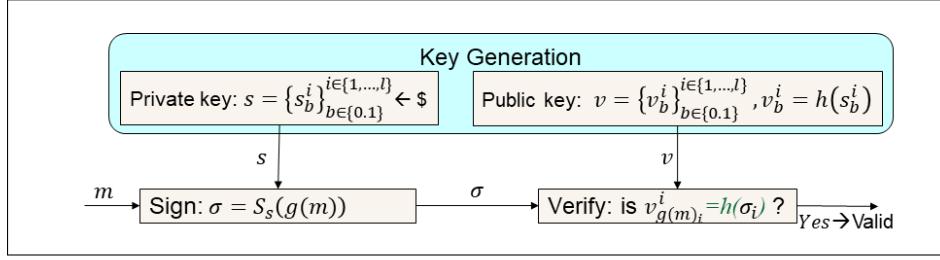


Figure 4.17: A one-time signature scheme, limited to a single bit (as well as to a single signature).

## 4.5 Randomness extraction

For the last security property we define, we use the term *randomness extraction*. Intuitively, a function is randomness extracting, if its output is uniformly random, provided that its input has ‘sufficient randomness’. This is important in security and cryptography, since randomness is necessary for many mechanisms; in particular, randomness is essential for both encryption and key-generation, as well as for challenge-response based authentication. However, cryptographic systems rarely have available sources of ‘true, perfect randomness’; existing sources of randomness, such as measurements of delays of different physical actions, are definitely not perfectly random. Another application of randomness extraction is for *key derivation* §6.3. Randomness extraction also received considerable attention in the research of the theory of cryptography.

In spite of its important applications and importance in theoretical research, randomness extraction is less well known to practitioners than the previous properties, and not listed among the requirements of standard hash functions. This may be since it is more subtle and harder to define; in particular, what does it mean for the input to have ‘sufficient randomness’? There are different possible formal definitions for this intuitive requirement. Unfortunately, the definitions used in the theoretical cryptography publications, appear to be too complex for our modest goals. Instead, we discuss two simple models for randomness extraction.

### Von Neuman’s Biased-Coin Model

We first discuss the classical *biased-coin* model proposed already at 1951 by Von Neuman [112], one of the pioneers of computer science. In the Von Neuman model, each of the input bits is the result of an independent tosses of a coin with fixed bias. Namely, for every bit generated, the value 1 is generated with probability  $0 < p < 1$  and the value 0 is generated with probability  $1 - p$  - with no relation to the value of other bits.

Von-Neuman proposed the following method to extract perfect randomness from these biased bits. First, arrange these sampled bits in pairs  $\{(x_i, y_i)\}$ .

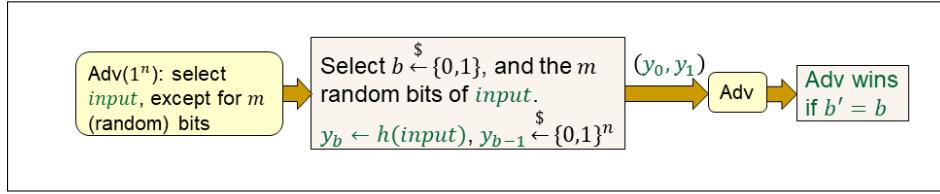


Figure 4.18: Bitwise-Randomness Extraction (simplified).

Then, remove pairs where both bits are identical, i.e., leave only pairs of the form  $\{(x_i, 1-x_i)\}$ . Finally, output the sequence  $\{x_i\}$ . This simple - if somewhat ‘wasteful’ in input bits - algorithm, outputs uniformly random bits.

**Exercise 4.11** (Von Neuman extractor). *Show that, if the input satisfies the assumption of the Von Neuman model, then the output is uniformly random, i.e., each bit  $x_i$  is 1 with probability exactly half - independently of all other bits.*

The Von Neuman extractor is simple, and the output is proven uniform without any computational assumption. However, the assumption is hard to justify, for most typical security applications of randomness-extraction. In particular, consider the goal of *key derivation*, in particular as applied to CDH problem in the DH protocol (§6.3); surely there is no justification to assume that every bit is result of a biased coin flip. We next discuss a different model, which seems to be applicable to more scenarios.

### Bitwise Randomness Extracting Function

We now present a different simple model for randomness extraction, which we call *bitwise randomness extracting*. We believe that bitwise randomness extraction is helpful for understand this important property of cryptographic hash functions.

Intuitively, a hash function is *bitwise-randomness extracting* if the output is pseudorandom, even if the adversary can select the input message - except for a ‘sufficient number’ of the bits of the message. This intuition is illustrated in Fig. 4.18, which defines a ‘game’ where an adversarial algorithm  $Adv$  tries to defeat the randomness extraction - by selecting some input message, except for some number  $m$  of random bits, and then distinguishing between the output and a random string (of the same length).

In Fig. 4.19, we further clarify how the adversary selects the input (except for  $m$  random bits). The adversary  $Adv$  first outputs *two* binary strings, denoted  $x$  and  $M$ , of the same length  $|x| = |M|$ . The string  $x$  is the input to the hash function, selected by the adversary - except for (at least)  $m$  of the bits in  $x$ , which are ‘randomized’, i.e., replaced with random bits.

The string  $M$  is a *bitmask*; each of its bits corresponds to one bit of  $x$ . When a bit in  $M$  is turned on (1), then the corresponding bit of  $x$  would be

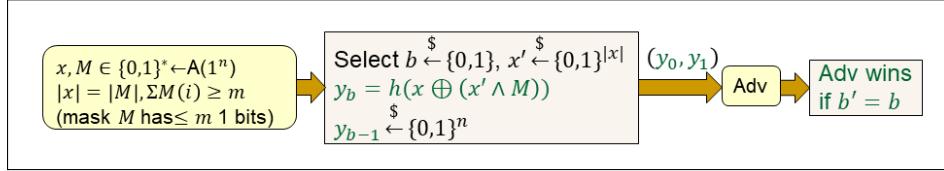


Figure 4.19: Bitwise-Randomness Extraction (more precise).

randomized. At least  $m$  bits in  $M$  contain 1, i.e.,  $m \leq \sum_i M[i]$ . If  $M[i] = 1$ , i.e., the  $i^{th}$  bit  $i$  of  $M$  is turned on, then this bit of  $x$  is randomized. Technically, we do this by selecting randomly a string  $x'$  of the same length as  $x$ , and then using  $x \oplus (x' \wedge M)$  as the input to the hash function  $h$ .

Finally, we perform an ‘indistinguishability test’, much like used to define (IND-CPA) encryption (Definition 2.7), or, even more, the (simpler) definition of pseudorandom generator (PRG, Definition 2.4). Namely, we select a random bit  $b$ , and let  $y_b = h(x \oplus (x' \wedge M))$  and  $y_{b-1} \stackrel{\$}{\leftarrow} \{0,1\}^n$ ; the adversary  $Adv$  ‘wins’ if it correctly guesses the value of  $b$ .

**Definition 4.7** (Bitwise-Randomness Extracting function). *An efficient hash function  $h : \{0,1\}^* \rightarrow \{0,1\}^n$  is called bitwise-randomness extracting if for every efficient algorithm  $A \in PPT$ , every negligible function  $NEGL$  and for sufficiently large  $m$ , holds:*

$$\Pr [b = ExtExp_{A,h}(b, m)] \leq \frac{1}{2} + NEGL(n)$$

where the probability is taken over  $b \stackrel{\$}{\leftarrow} \{0,1\}$  and over the coin tosses of  $A$ , and where the ‘bitwise-randomness extraction experiment’  $ExtExp_{A,h}(b, m)$  is shown in Algorithm 1.

---

**Algorithm 1** Bitwise-Randomness Extraction experiment  $ExtExp_{A,h}(b, m)$ .

---

```

(x, M) ← A(1m+n)
If |x| ≠ |M| or  $\sum_{i=1}^{|M|} M(i) < m$  return ⊥
x' ← {0, 1}|x|
yb ← h(x ⊕ (x' ∧ M)) (note: the range of h is {0, 1}n)
yb-1 ← {0, 1}n
return A(y0, y1, x, M)

```

---

It is interesting to note that it is easy to design a bitwise-randomness extracting function  $h : \{0,1\}^* \rightarrow \{0,1\}$ , i.e., for the very special case of  $n = 1$ , i.e., extracting a single bit; but extracting more bits is not as simple.

**Exercise 4.12.** Present a bitwise-randomness extracting function  $h$  whose output is a single bit, and prove that it satisfies the definition.

Randomness extraction is closely related to PRG; in particular, in the common case where we have a partially-random input, but need more than  $n$  (pseudo)random bits, we can use extractor to get  $n$  pseudorandom bits and then use a PRG  $f$  to expand it to a longer pseudorandom string. This called the *extract-then-expand* methodology. Note that  $f$  may expand its input by much more than one bit.

**Exercise 4.13** (Extract-then-expand). *Let  $h$  be a bitwise-randomness-extracting hash function and  $f$  be a secure PRG. Show that the function  $g(x) = f(h(x))$  is a randomness extracting hash function (with longer output length). Namely, for large enough  $m$ , if  $m$  (or more) bits in  $x$  are random, then  $g(x) = f(h(x))$  is pseudorandom.*

## 4.6 The Random Oracle Methodology

Often, designers use cryptographic hash functions, but without identifying a specific, well-defined requirement of the hash function, that suffices to ensure security. However, such constructions are often still ‘secure’ in the practical sense, that no practical attack against them is found - for many years, and often, in spite of considerable motivation for cryptanalysis and efforts.

Of course, there are also plenty of constructions which similarly use hash functions, without a well-defined assumption - but which have been found to be insecure. However, very often, the attacks are *generic*, i.e., do not exploit a weakness of a specific hash function, and apply when the design is implemented with an arbitrary hash function.

Therefore, even when we cannot identify the specific ‘generic’ property of hash function on which security relies, it is useful to distinguish between designs which are secure assuming *some* reasonable security property of the hash function, and designs which are vulnerable - even for an ‘ideal-security’ hash function. However, how can we define such ‘ideal-security’ hash function?

The *Random Oracle Methodology (ROM)*<sup>3</sup>, proposed by Bellare and Rogaway [16], is a common approach to this dilemma. Intuitively, *Random Oracle Methodology (ROM)* constructions and protocols are secure in the (impractical) case that the parties select  $h()$  as a truly random function (for the same domain and range), instead of using a concrete, specific hash function as  $h()$ . The function  $h()$  is still assumed to be public, i.e. known to the attacker.

A central component of this approach is the concept of a *random oracle*, which is a randomly-chosen function available to both system and adversary; in practice, we will instantiate this system using a specific hash function. Namely, we model an ideal hash function as a *random function* (over the same domain and range, i.e.,  $\{0, 1\}^* \rightarrow \{0, 1\}^n$ ). We next present a somewhat informal definition of this concept, with the hope that it will help clarify this important approach (at least to some readers; you may ignore this definition if you find it not helpful.).

---

<sup>3</sup>Often people use the term ‘model’ instead of ‘methodology’, but we find it a bit misleading, and prefer ‘methodology’ or ‘method’.

**Definition 4.8** (ROM-security). Let  $H$  be the set of all ‘hash functions’ i.e., functions from arbitrary-length binary strings to  $n$ -bit binary strings:  $H = \{h : \{0, 1\}^* \rightarrow \{0, 1\}^n\}$ . Consider a parametrized system  $\mathcal{S}^h$ , which can use any given hash function  $h$ ; namely, instantiated with  $\mathcal{S}^h$  a specific  $h \in H$ , then is a well-defined system. Similarly, consider a parametrized adversary  $\mathcal{A}^h$ , which can use any given hash function  $h$ ; namely, instantiated with a specific  $h \in H$ , then  $\mathcal{A}^h$  is a well-defined algorithm. Furthermore, assume both  $\mathcal{S}^h$  and  $\mathcal{A}^h$  operate in polynomial time (in their inputs), if the running time of calls to  $h$  is ignored.

Consider an arbitrary security definition  $\pi(\mathcal{S}^h, \mathcal{A}^h)$  for such systems and adversaries (both instantiated with the same specific hash function  $h \in H$ ), namely,  $\pi$  is a predicate which maps instantiated-systems and instantiated-adversaries to secure (system is secure for this adversary) or vulnerable.

We say that the (parametrized) system  $\mathcal{S}_h$  is **ROM-secure**, or secure using the Random Oracle Methodology, if for every parametrized adversary  $\mathcal{A}^h$ , when we select  $h$  as a random hash function, then  $\mathcal{S}^h$  satisfies security definition  $\pi$ , except with negligible probability. Namely,

$$\Pr_{h \xleftarrow{\$} H} (\pi(\mathcal{S}^h, \mathcal{A}^h) \neq \text{secure}) \in \text{NEGL}(n) \quad (4.8)$$

To ‘prove security under the ROM’, you would analyze the construction/protocol under as if the function  $h()$ , used by the protocol, was chosen randomly at the beginning of the execution and then made available to all parties - legitimate parties running the protocols or using the scheme/construction, as well as the adversary.

For example, in the particular case of Exercise 4.25, you would prove that if we instantiate the HMAC construction in this way (with  $h$  being a truly random function) then the resulting HMAC would satisfy the properties listed. As another example, you can prove that OTP-chain is ROM-secure.

Exercise 4.26 is the ‘complementary’ question: it asks you to show that assuming (‘just’) the  $h()$  is a CRHF, is NOT enough for the HMAC construction to securely fulfill the mentioned properties. For example, if you come up with a function  $h()$  which is collision resistant, yet using it in the HMAC construction would expose the key used, then surely using the HMAC construction with this (weird)  $h$  would not result in secure PRF or MAC.

Note that ROM-security does not necessarily imply that the design is ‘really’ secure, when implemented with any specific hash function; once the hash function is fixed, there may very well be an adversary that breaks the system. However, ROM-security is definitely a very good indication of security, since a vulnerability has to use some property of the specific hash function. Indeed, there are many designs which are only proven to be ROM-secure. In fact, ROM-security is so often used, that papers often use the term ‘secure in the standard model’ to emphasize that their results are ‘really’ proven secure, rather than only ROM-secure.

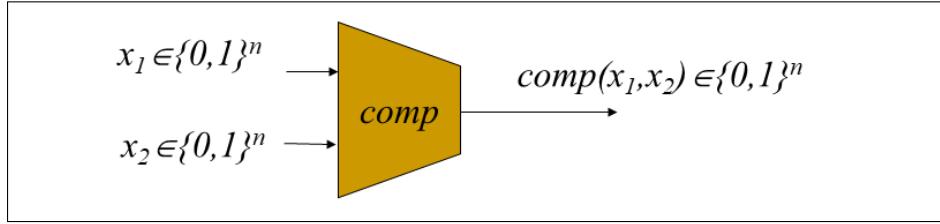


Figure 4.20: A compression function: fixed input length to fixed (and shorter) output length. For simplicity, assume input length is  $2n$ .

#### 4.6.1 Proof-of-Work (PoW) and Bitcoin

A *Proof of Work (PoW)* is a very different cryptographic mechanism, compared to the other mechanisms we discussed. Intuitively, a PoW allows one party, the *worker*, to solve a *challenge*, with *approximately known amount of computational work*, resulting in a *proof* of this success, which can be efficiently verified by anyone.

[TBD]

### 4.7 Constructing crypto-hash: from FIL to VIL

Recall the ‘cryptographic building blocks’ principle 7: *The security of cryptographic systems should only depend on the security of few basic building blocks. These blocks should be simple and with well-defined and easy to test security properties.*

Cryptographic hash functions are one of these few building blocks of applied cryptography, due to their simplicity and wide range of applications. However, they may be further simplified by restricting their domain to fixed-input length (FIL). In this section, we discuss two methods to construct a VIL crypto hash function, from a simple, FIL cryptographic hash function. Actually, the term *compression function* is commonly used for FIL cryptographic hash functions; it may not be the best terminology, but to avoid confusion, we adopt it. Therefore, in this section, we discuss constructions of a (VIL) cryptographic hash function  $h$ , from (FIL, cryptographic) compression function  $comp$ .

For simplicity, we consider a compression function  $comp$  which accepts two inputs of  $n$  bits, and output a single  $n$  bit string. See Figure 4.20.

#### 4.7.1 The Merkle hash tree construction

The Merkle hash tree construction extends the hash-block technique of §4.2.2, to construct a (VIL) cryptographic hash function, using a (FIL) cryptographic compression function. A simplified illustration of the scheme is presented in Fig. 4.21; this illustration assumes that the compression scheme has two  $n$  bit inputs and one  $n$  bit output, i.e.,  $comp : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ , as in

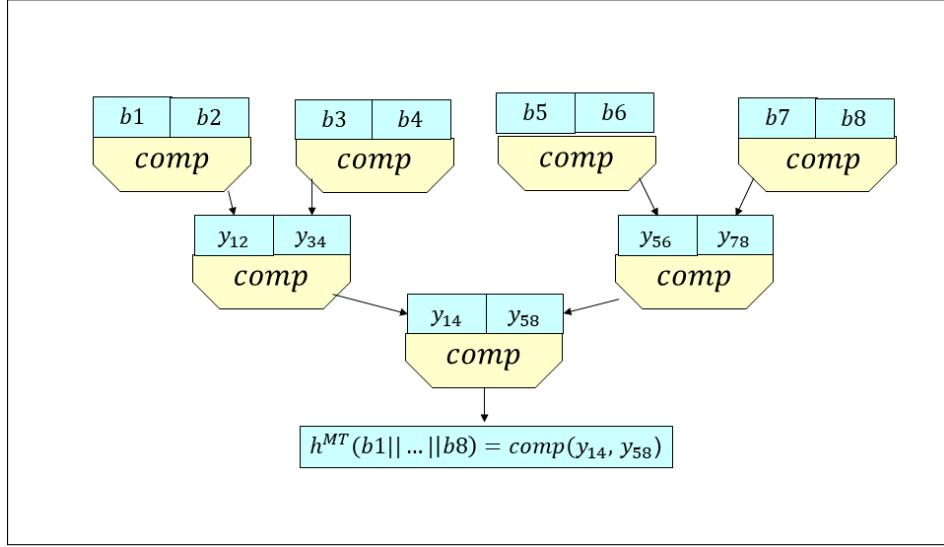


Figure 4.21: From compression (FIL-hash) to (VIL) hash function: example of the Merkle Hash Tree.

Figure 4.20. The illustration further assumes that the input consists of eight blocks of  $n$  bits,  $b_1, \dots, b_8$ .

As a formula, this simplified Merkle tree  $h$ , using compression function  $comp : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ , is defined by:

$$h^{MT}(m) = \left\{ \begin{array}{ll} \text{If } l = 1: & comp(b_1 || b_2) \\ \text{Else:} & comp(h^{MT}(m_{1,2^{l-1}}), h^{MT}(m_{2^{l-1}+1,2^l})) \end{array} \right\} \quad (4.9)$$

Where the input string is  $m = b_1 || \dots || b_{2^l}$  with each  $b_i$  being one ‘block’ (fixed number of bits, say  $2n$ ) and we use the notation  $m_{i,j} = b_i, \dots, b_j$  to refer to a substring of the message.

It is easy to see that if there is a collision for two inputs of the same length, i.e.,  $b_1 || \dots || b_{2^l}$  and a different  $b'_1 || \dots || b'_{2^l}$  s.t.  $h^{MT}(b_1 || \dots || b_{2^l}) = h^{MT}(b'_1 || \dots || b'_{2^l})$ , then there is also a collision for the compression function  $comp$ . Hence, if  $comp$  is a CRHF, then  $h$  is also a CRHF, at least, if the inputs to  $h$  are limited to be of fixed size (some number  $2^l$  of blocks).

For simplicity, we presented the construction with the restriction that the input is exactly  $2^l$  blocks of  $n$  bits each. To support arbitrary-length input strings, the construction is adjusted to allow the input to be any number  $m$  of blocks (and not necessarily  $m = 2^l$ ), and the input is padded to length of  $m$  blocks (of  $n$  bits each); we omit the details.

Similarly, for simplicity, we presented the construction assuming that the compression function  $comp$  is ‘symmetric’, i.e., both of its inputs are of length  $n$  bits. In practice, typical ‘building block’ compression functions are often asymmetric - one input is, e.g., 512 bits, and the other is shorter, e.g., 128

bits. A further simple adjustment to the construction allows the use of such ‘asymmetric’ compression function.

Even with these adjustments, this construction is rarely if ever deployed, in practice, to construct (VIL) hash from (FIL) compression functions. One reason are implementation and performance drawbacks, compared to alternatives such as the Merkle-Damgård construction, which we present next. In particular note that computation of the Merkle-tree hash, requires at least storage of one hash value for each layer of the tree, which is particularly inconvenient in hardware.

Furthermore, the Merkle tree construction also offers weaker security guarantees. Foremost, it does not fully ensure collision-resistance - even if *comp* is collision resistant. Of course, as argued above, if *comp* is collision resistant, then  $h^{MT}$  is collision resistant - when the inputs are restricted to the same number  $2^l$  of blocks (possibly after padding). However, it is not too difficult to present example of two inputs of *different length* which are mapped to the same output; see next exercise.

**Exercise 4.14.** Let *comp* be a collision-resistant compression function (i.e., CRHF, for some fixed length ‘blocks’), and let  $h^{MT}$  be the result of the Merkle tree construction using *comp*, as illustrated in Fig. 4.21. Show two strings  $x \neq x'$  s.t.  $h^{MT}(x) = h^{MT}(x')$ .

*Hint:* Consider an arbitrary string of some four blocks,  $x = b_1||b_2||b_3||b_4$ , i.e.,  $h^{MT}(x) = h^{MT}(b_1||b_2||b_3||b_4) = comp(y_{12}||y_{34})$ , where  $y_{12} = comp(b_1||b_2)$ ,  $y_{34} = comp(b_3||b_4)$ . From this, identify a string  $x'$  of two blocks,  $x' = b'_1||b'_2$ , s.t.  $h^{MT}(x) = h^{MT}(x')$ .  $\square$

Another security-disadvantage of the Merkle tree is that its security is ‘fragile’, in the sense that given *any* collision of the compression function *comp*, we can find collisions for the Merkle-tree hash  $h^{MT}$  - for *any* given prefix and suffix. This is illustrated in Figure 4.22.

### Using Merkle Tree to improve privacy and efficiency

In spite of these significant drawbacks, the Merkle tree construction is still widely used, but normally, with each node applying a hash function  $h$  instead of a compression function *comp*, with variable-length inputs - typically, messages or files.

Figure 4.23 illustrates this construction, using a (possibly VIL) hash function instead of a compression function (i.e., FIL hash function) as in Figure 4.21; note that we denote the inputs by  $m_1, \dots, m_8$ , to emphasize that these are messages (or files) and not necessarily fixed-length blocks.

The figure also illustrates the two main motivations for using such Merkle hash tree:

**Efficiency:** The same ‘fingerprint’, e.g.,  $h^{MD}(m_1||\dots||m_8)$  in the example in Figure 4.23, can be used to authenticate all of the messages - but also any subset, including the typical case of just one message, e.g.,  $m_5$ .

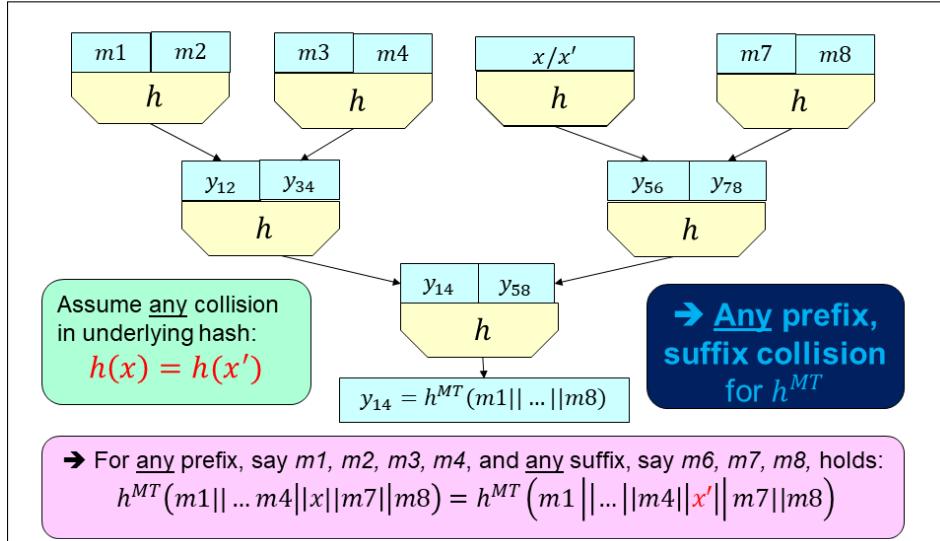


Figure 4.22: Any given collision  $h(x) = h(x')$  for  $h$ , there is a collision for any prefix  $p$  and suffix  $s$ . In this example,  $p$  can be  $m_1$ ,  $m_1||m_2$ ,  $m_1||\dots||m_3$  or  $m_1||\dots||m_4$ , and similarly  $s$  can be  $m_8$  or  $m_7||m_8$ .

Furthermore, to validate a single message such as  $m_5$ , we only need a limited number of additional values: (1)  $m_6$ , to compute  $y_{56} = h(m_5||m_6)$ , (2)  $y_{78}$ , to compute  $y_{58} = h(y_{56}||y_{78})$ , and (3)  $y_{14}$ , to compute  $y_{14} = h^{MD}(m_1||\dots||m_8) = h(y_{14}||y_{58})$ . This is important in many cases, e.g., to use only a single (computationally-intensive) public key signature operations, to authenticate multiple files or messages, allowing specific recipients to only validate relevant parts. Essentially, in this regards, the Merkle hash tree is an improved version of the ‘single-layer’ hash-block construction, allowing more efficient ‘proofs’.

**Privacy:** As we just explained, using hash-tree, we do not have to produce the contents of any messages except those requiring validation. This is handy when some of the messages should not be exposed to all parties - e.g., each party may only receive its own message, plus the necessary ‘internal nodes’ ( $y_{nn}$  values).

#### 4.7.2 The Merkle-Damgård Construction

We next discuss the Merkle-Damgård construction of a (VIL) cryptographic hash function  $h$ , from a (FIL) cryptographic compression function, denoted  $comp$  or simply  $c$ . This is an *iterative* design; the basic idea is to compress each block, with the result of the compression of the previous block, as illustrated in Figure 4.24.

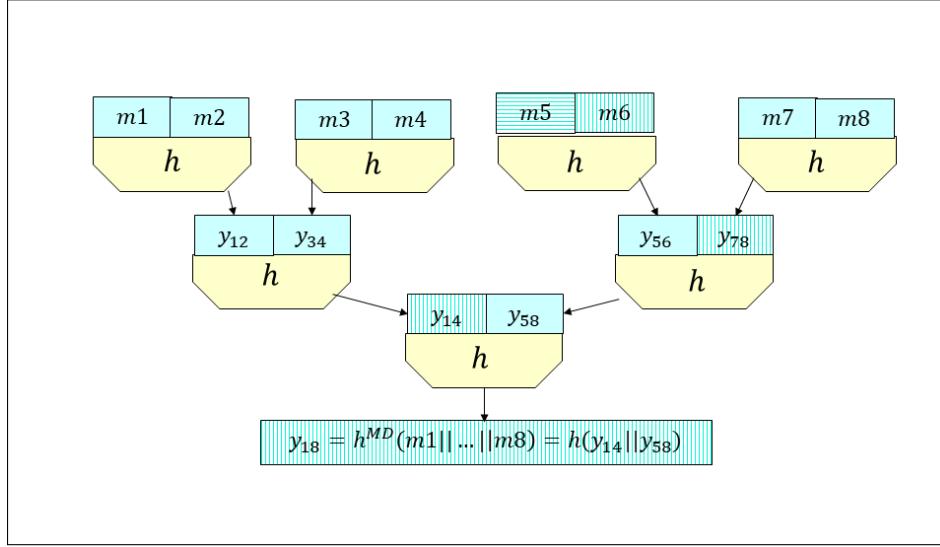


Figure 4.23: Typical use of Merkle Tree, to improve privacy and/or efficiency when validating many files/messages.

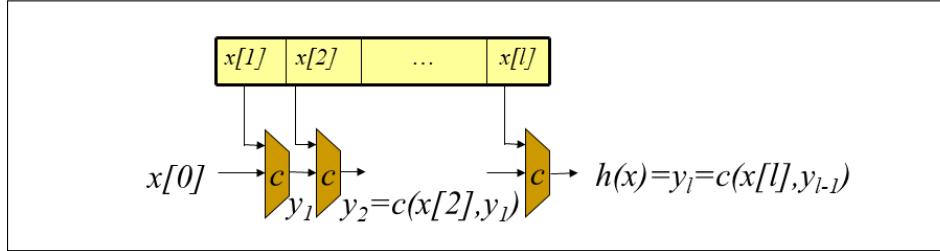


Figure 4.24: The ‘naive’ Merkle-Damgård Construction

We present this design in three phases. The first phase, nicknamed the ‘naive MD construction’, is illustrated in Figure 4.24. For simplicity, in Figure 4.24, and throughout this subsection, we assume a compression function  $c$  from  $2n$  bits to  $n$  bits, as in Figure 4.20, although the design trivially adapts to an asymmetric compression function, with input  $m + n$  bits and output  $n$  bits; with  $m > n$ , this requires less invocations of the compression function, and typical values in practice are  $m = 512$  and  $n = 128, 160$  or  $256$ .

With this simplification ( $m = n$ , i.e., input to compression function is of length  $2n$ ), and assuming that the input to the hash function consists exactly

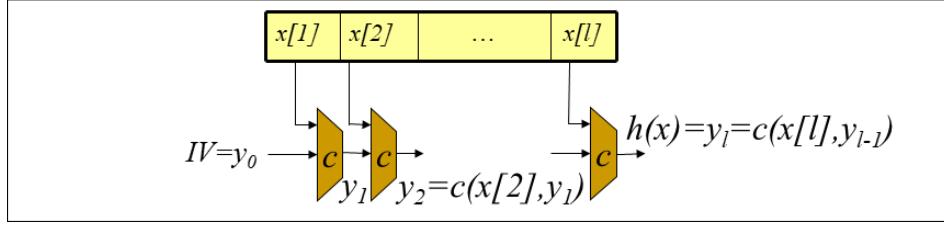


Figure 4.25: The Merkle-Damgård Construction with Initialization Vector (IV)

of an integral number of ‘blocks’, i.e.,  $(l + 1) \cdot n$  bits, we can define  $h$  as:

$$y_0 = x[0] \quad (4.10)$$

$$y_1 = c(x[0]||x[1]) \quad (4.11)$$

$$(\forall i : 1 \leq i \leq l) y_i = c(y_{i-1}||x[i]) \quad (4.12)$$

$$h(x) = y_l \quad (4.13)$$

However, this simplified construction is definitely *not* a CRHF. Specifically, note that:

$$h(x[0]||x[1]||x[2]) = h(y_1||x[2]) \quad (4.14)$$

Where  $y_1$  is defined as in Eq. (4.11).

### Merkle-Damgård with IV

We will present two ideas that are usually used to ensure that the resulting hash function is CRHF - assuming that the compression function is CRHF. The first idea is to use a specific, fixed *initialization value (IV)* for  $y_0$ , instead of using the first block  $x[0]$ . This idea is shown in Figure 4.25.

By adding the fixed IV, we definitely prevent the attack presented earlier (in Eq. (4.14)). We also provide a way to turn the construction into a *keyed* crypto-hash, by viewing the IV as a key. Furthermore, this change makes it much less likely, that a randomly discovered collision for  $c$ , would translate to an applicable collision for  $h$ .

**Exercise 4.15.** Assume that one specific collision  $(x_1, x_2)$  is found for the compression function, i.e.,  $c(x_1) = c(x_2)$ , where  $x_1, x_2 \in \{0, 1\}^{2n}$  and  $x_1 \neq x_2$ . Show a collision for the naive construction (Figure 4.24); and then show how this collision implies many other collisions. On the other hand, argue that discovery of such a specific collision for the compression function, is not very likely to result in discovery of collision for the MD construction with IV, Figure 4.25.

*Hint:* once you found one collision to the hash function, you can find longer collisions by extending it.  $\square$

This change is also sufficient to ensure that given a collision  $(x, x')$  of  $h$ , we can find a collision for the compression function  $c$  - however, only for same-length collisions, i.e.,  $|x| = |x'|$ .

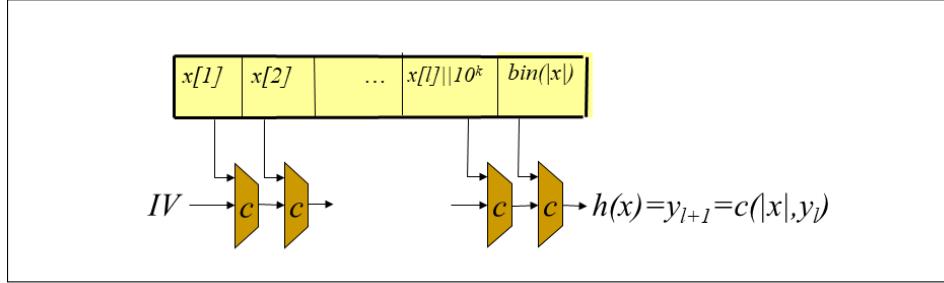


Figure 4.26: The Strengthened Merkle-Damgard Construction

**Lemma 4.1.** Any collision  $x \neq x'$  s.t.  $h(x) = h(x')$  for the MD construction with IV (Figure 4.25), where the two strings are of the same length  $l = |x| = |x'|$ , implies a collision in the compression function  $c$ .

*Proof:* Assume, to the contrary, that there is such collision. Let  $i$  be the maximal value s.t.  $x[i] \neq x'[i]$ ; there must be such value (otherwise,  $x = x'$ ). We denote  $h(x')$  as  $y'_l$  and similarly, we denote by a tag all the intermediate variables used when computing  $h(x')$ .

If  $y_i = y'_i$ , then we have a collision; otherwise, let  $j > i$  be the maximal value s.t.  $y_j \neq y'_j$ ; there must be such  $j < l$ , since, if  $j = l$ , then  $y'_l \neq y_l$  which contradicts the assumption of collision to  $h$ . Namely, for  $j < l$ , we have  $y_{j+1} = y'_{j+1}$ .

However,  $y_{j+1} = c(y_j || x[j+1])$ , and  $y'_{j+1} = c(y'_j || x'[j+1])$ ; i.e., since  $y_{j+1} = y'_{j+1}$  and  $y_j \neq y'_j$ , we have a collision for the compression function.  $\square$

What about collisions of two strings of different length? This may still be possible - even for when no collisions are known for the compression function. Let us present this as an exercise - although, since it may be a tricky one, we will also provide the solution.

**Exercise 4.16.** Show that there may be a collision-resistant compression function  $c$ , s.t. the MD construction with IV using  $c$ , denoted  $h$ , is not a CRHF.

*Solution:* We first observe that if there are collision-resistant compression functions, then there is a collision-resistant compression function such that  $c(IV || \xi) = IV$  for some value  $\xi$ ; the proof of this observation is technical and omitted. It follows that  $h(\xi) = h(\xi || \xi) = h(\xi || \xi || \xi) = \dots$ , i.e., there are collisions in  $h$ .  $\square$

### The Merkle-Damgard length-padding strengthening

We now present the second idea, which consists of padding the length of the input to the end of the input string; this method is usually called *Merkle-Damgard strengthening*. The construction is illustrated in Figure 4.26.

By appending the length, we prevent collisions using different-length strings.

**Lemma 4.2.** Any collision  $x \neq x'$  s.t.  $h(x) = h(x')$  for the strengthened MD construction (Figure 4.26), implies a collision in the compression function  $c$ .

*Proof:* There are two cases. In the first case,  $|x| = |x'|$ . This means that the lengths of  $x$  and  $x'$  with their lengths padded to the end are also equal. In this case, the collision is implied by the same argument as in Lemma 4.1. In the second case,  $|x| \neq |x'|$ . In this case, since  $h(x) = h(x')$  and the lengths are padded at the end of  $x$  and  $x'$ , then  $c(|x|, y_l) = c(|x'|, y'_l)$ , where  $|x| \neq |x'|$ , which is a collision in  $c$  (regardless of whether or not  $y_l$  and  $y'_l$  are equal).  $\square$

#### 4.7.3 Some concerns with Merkle-Damgard construction

[TBD]

### 4.8 The HMAC and NMAC constructions

[TBD]

### 4.9 Crypto-hashing Robust combiners

[TBD]

### 4.10 Cryptographic hash functions: additional exercises

**Exercise 4.17** (XOR-hash). Consider messages of  $n$  blocks of  $l$  bits each, denoted  $m_1 \dots m_n$ . Define hash function  $h$  for such  $l \cdot n$  bit messages, as:  $h(m_1 \dots m_n) = \bigoplus_{i=1}^n m_i$ . Show that  $h$  does not have each of the following properties, or present a convincing argument why it does:

1. Collision-resistance.
2. Second-preimage resistance.
3. One-wayness (preimage resistance)
4. Randomness extraction.
5. Secure MAC, when  $h$  is used in the HMAC construction.

*Solution to part 4 (randomness extraction):* consider even  $n$ , i.e.,  $n = 2\mu$  for some integer  $\mu$ , and random messages of the form  $m_1 || \dots || m_n$  where for every  $i = 1 \dots$  holds  $m_{2i-1} = m_{2i}$ . Clearly, this set of messages has lots of randomness (in fact  $m \cdot l$  random bits); however the output of  $h$  would be always zero, i.e., completely deterministic and predictable. Hence  $h$  is not randomness extracting.  $\square$

**Exercise 4.18.** Let  $h$  be a ‘compression function’, i.e., a cryptographic hash function whose input is of length  $2l$  and output is of length  $l$ . Let  $h' : \{0,1\}^{2l \cdot n} \rightarrow \{0,1\}^l$  extend  $h$  to inputs of length  $2l \cdot n$ , as follows:  $h'(m_1||\dots||m_n) = \bigoplus_{i=1}^n h(m_i)$ , where  $(\forall i = 1, \dots, n) |m_i| = 2l$ . For each of the following properties, assume  $h$  has the property, and show that  $h'$  may not have the same property. Or, if you believe  $h'$  does retain the property, argue why it does. The properties are:

1. Collision-resistance.
2. Second-preimage resistance.
3. One-wayness (preimage resistance)
4. Randomness extraction.

Would any of your answers change, if  $h$  and  $h'$  have a random public key as an additional input?

**Exercise 4.19.** Consider messages of  $2n$  blocks of  $l$  bits each, denoted  $m_1 \dots m_n$ , and let  $h_c$  be a secure compression function, i.e., a cryptographic hash function from  $2n$  bits to  $l$  bits. Define hash function  $h$  for such  $2n$  blocks of  $l$  bits messages, as:  $h(m_1 \dots m_{2n}) = \bigoplus_{i=1}^n h_c(m_{2i}, m_{2i-1})$ . Show that  $h$  does not have each of the following properties, although  $h_c$  has the corresponding property, or present a convincing argument why it does:

1. Collision-resistance.
2. Second-preimage resistance.
3. One-wayness (preimage resistance)
4. Randomness extraction.
5. Secure MAC, when  $h$  is used in the HMAC construction.

**Exercise 4.20.** It is proposed to combine two hash functions by cascade, i.e., given hash functions  $h_1, h_2$  we define  $h_{12}(m) = h_1(h_2(m))$  and  $h_{21}(m) = h_2(h_1(m))$ . Suppose collision are known for  $h_1$ ; what does this imply for collisions in  $h_{12}$  and  $h_{21}$ ?

**Exercise 4.21.** Recently, weaknesses were found in few cryptographic hash functions such as  $h_{MD5}$  and  $h_{SHA1}$ , and as a result, there were many proposals for new functions. Dr. Simpleton suggests to combine the two into a new function,  $h_c(m) = h_{SHA1}(h_{MD5}(m))$ , whose output length is 160 bits. Prof. Deville objects; she argued that hash functions should have longer outputs, and suggest a complex function,  $h_{666}$ , whose output size is 666 bits. A committee setup to decide between these two, proposes, instead, to XOR them into a new function:  $f_X(m) = [0^{506}||h_c(m)] \oplus h_{666}(m)$ .

1. Present counterexamples showing that each of these may not be collision-resistant.
2. Present a design where we can be sure that finding a collision is definitely not easier than finding one in  $h_{SHA1}$  and in  $h_c$ .
3. Repeat both parts, for randomness-extraction.

**Exercise 4.22.** Let  $h$  be the result of a Merkle hash tree, using a compression function  $\text{comp} : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ , and let  $(KG, S, V)$  be a secure (FIL) signature scheme. Let  $S_s^h(m) = S_s(h(m))$  follow the ‘hash then sign’ paradigm, to turn  $(KG, S, V)$  into a VIL signature scheme, i.e., allow signatures over messages of arbitrary number of blocks. Show that  $S_s^h$  is not a secure signature scheme, by presenting an efficient adversary (program) that outputs a forged signature.

**Exercise 4.23.** Consider the following slight simplification of the popular HMAC construction:  $h'_k(m) = h(k||h(k||m))$ , where  $h : \{0,1\}^* \rightarrow \{0,1\}^n$  is a hash function,  $k$  is a random, public  $n$ -bit key, and  $m \in \{0,1\}^*$  is a message.

1. Assume  $h$  is a CRHF. Is  $h'_k$  also a CRHF?
  - Yes. Suppose  $h'_k$  is not a CRHF, i.e., there is some adversary  $A'$  that finds a collision  $(m'_1, m'_2)$  for  $h'$ , i.e.,  $h'_k(m'_1) = h'_k(m'_2)$ . Then at least one of the following pairs of messages  $(m_{1,1}, m_{2,1})$ ,  $(m_{1,2}, m_{2,2})$  is a collision for  $h$ , i.e., either  $h(m_{1,1}) = h(m_{2,1})$  or  $h(m_{1,2}) = h(m_{2,2})$  (or both). The strings are:  $m_{1,1} = \underline{\hspace{1cm}}$ ,  $m_{1,2} = \underline{\hspace{1cm}}$ ,  $m_{2,1} = \underline{\hspace{1cm}}$ ,  $m_{2,2} = \underline{\hspace{1cm}}$ .
    - No. Let  $\hat{h}$  be some CRHF, and define  $h(m) = \underline{\hspace{1cm}}$ . Note that  $h$  is also a CRHF (you do not have to prove this, just to design  $h$  so this would be true and easy to see). Yet,  $h'_k$  is not a CRHF. Specifically, the following two messages  $m'_1 = \underline{\hspace{1cm}}$ ,  $m'_2 = \underline{\hspace{1cm}}$  are a collision for  $h'_k$ , i.e.,  $h'_k(m_1) = h'_k(m_2)$ .
2. Assume  $h$  is a SPR hash function. Is  $h'_k$  also SPR?
  - Yes. Suppose  $h'_k$  is not SPR, i.e., for some  $l$ , there is some algorithm  $A'$  which, given a (random, sufficiently-long) message  $m'$ , outputs a collision, i.e.,  $m'_1 \neq m'$  s.t.  $h'_k(m') = h'_k(m'_1)$ . Then we define algorithm  $A$  which, given a (random, sufficiently long) message  $m$ , outputs a collision, i.e.,  $m_1 \neq m$  s.t.  $h_k(m) = h_k(m_1)$ . The algorithm  $A$  is:  
 Algorithm  $A(m)$ :  
 {  
 Let  $m' = \underline{\hspace{1cm}}$   
 Let  $m'_1 = A'(m')$   
 Output  $\underline{\hspace{1cm}}$   
 }
  - No. Let  $\hat{h}$  be some SPR, and define  $h(m) = \underline{\hspace{1cm}}$ . Note that  $h$  is also a SPR (you do not have to prove this, just to design  $h$  so this would be true and easy to see). Yet,  $h'_k$  is not a SPR. Specifically,

given a random message  $m'$ , then  $m'_1 = \underline{\hspace{2cm}}$  is a collision, i.e.,  $m' \neq m'_1$  yet  $h'_k(m'_1) = h'_k(m'_2)$ .

3. Assume  $h$  is a OWF. Is  $h'_k$  also a OWF?

Yes. Suppose  $h'_k$  is not OWF, i.e., for some  $l$ , there is some algorithm  $A'$  which, given  $h'_k(m')$  for a (random, sufficiently-long) message  $m'$ , outputs a preimage, i.e.,  $m'_1 \neq m'$  s.t.  $h'_k(m') = h'_k(m'_1)$ . Then we define algorithm  $A$  which, given  $h(m)$  for a (random, sufficiently long) message  $m$ , outputs a preimage, i.e.,  $m_1$  s.t.  $h_k(m) = h_k(m_1)$ . The algorithm  $A$  is:

Algorithm  $A(m)$ :

{

Let  $m' = \underline{\hspace{2cm}}$

Let  $m'_1 = A'(m')$

Output  $\underline{\hspace{2cm}}$

}

No. Let  $\hat{h}$  be some SPR, and define  $h(m) = \underline{\hspace{2cm}}$ . Note that  $h$  is also a SPR (you do not have to prove this, just to design  $h$  so this would be true and easy to see). Yet,  $h'_k$  is not a SPR. Specifically, given a random message  $m'$ , then  $m'_1 = \underline{\hspace{2cm}}$  is a collision, i.e.,  $m' \neq m'_1$  yet  $h'_k(m'_1) = h'_k(m'_2)$ .

4. Repeat similarly for bitwise randomness extraction.

**Exercise 4.24.** Consider the following construction:  $h'_k(m) = h(k||m)$ , where  $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$  is a hash function,  $k$  is a secret  $n$ -bit key, and  $m \in \{0, 1\}^*$  is a message. Assume you are given some SPR hash function  $\hat{h} : \{0, 1\}^* \rightarrow \{0, 1\}^{\hat{n}}$ ; you can use  $\hat{n}$  which is smaller than  $n$ . Using  $\hat{h}$ , construct hash function  $h$ , so that (1) it is ‘obvious’ that  $h$  is also SPR (no need to prove), yet (2)  $h'_k(m) = h(k||m)$  is (trivially) not a secure MAC. Hint: design  $h$  s.t. it becomes trivial to find  $k$  from  $h'_k(m)$  (for any  $m$ ).

1.  $h(x) = \underline{\hspace{2cm}}$ .

2. (Justification)  $h$  is a SPR since  $\underline{\hspace{2cm}}$ .

3. (Justification)  $h'_k(m) = h(k||m)$  is not a secure MAC since  $\underline{\hspace{2cm}}$ .

**Exercise 4.25** (HMAC is secure under ROM). Show that the HMAC construction is secure under the Random Oracle Methodology (ROM), when used as a PRF, MAC and KDF.

**Exercise 4.26** (HMAC is insecure using CRHF). Show counterexamples showing that even if the underlying hash function  $h$  is collision-resistant, its (simplified) HMAC construction  $hmac_k(x) = h(k||h(k||m))$  is insecure when used as any of PRF, MAC and KDF.

**Exercise 4.27** (Hash-tree with efficient proof of non-inclusion). The Merkle hash-tree allows efficient proof of inclusion of a leaf (data item) in the tree.

*Present a variant of this tree which allows efficient proof of either inclusion or of non-inclusion of an item with given ‘key’ value (where each item consist of a key and data; there may be multiple items with the same key). Assume all data items (keys and values) are given together - no need to build the tree dynamically. Your solution may ‘expose’ the another (key, value) pair beyond the one queried, or of two other values, for a ‘proof of non-inclusion’. Note: try to provide solution which is efficient in number of hash operations required for verification (the number should be about one more than in the regular Merkle tree).*

## Chapter 5

# Shared-Key Protocols

### 5.1 Introduction

In the previous two chapters, we discussed cryptographic schemes, which consist of one or more functions, with security criteria. For example, MAC, PRF and PRG schemes consist of a single function, with criteria such as security against forgery (Def. 3.1). Similarly, encryption schemes consist of multiple functions (encryption, decryption and possible key generation), with criteria such as CPA-indistinguishability (CPA-IND, Def. 2.8).

Cryptographic schemes are very useful, but rarely as a single, stand-alone function. Usually, cryptographic schemes are used as a part of different *protocols*, which involve multiple exchanges of messages among a set  $P$  of *parties*, often only two, e.g.,  $P = \{Alice, Bob\}$ . There are many types of protocols, including many related to cryptography, and, more generally, security.

**What is a protocol?** A protocol is a set of algorithms, one *algorithm*  $\pi_p$  for each party  $p \in P$ . Each party  $p \in P$  is also associated with a *state*  $s_p$ , which may change over time, in discrete *events*; we denote the state of  $p \in P$  at time  $t$  by  $s_p(t)$ .

An event is triggered by some *input signal*  $x$ , and results in some *output signal*  $y$ . We use  $s_p(t^-)$  for the state of  $p$  just before an event at time  $t$ , and  $s_p(t^+)$  for the state of  $p$  after the event. The output signal and the new state the result of applying  $\pi_p$ , the algorithm of the protocol at party  $p$ , to the input signal and previous state, i.e.:

$$(y, s_p(t^+)) \leftarrow \pi_p(x, s_p(t^-)) \quad (5.1)$$

An *execution* of the protocol, is a sequence of events, each associated with a specific participant  $p \in P$  and specific time, where the time of events is strictly increasing - we assume events never occur exactly at the same time.

### Note 5.1: Alternatives to global, synchronous time

We focus on applied, simpler models; in particular, we use a simple, synchronous model for time and protocols. Specifically, to use the notation  $s_p(t)$  for the state at  $p$  and time  $t$ , we assume that events, in all parties, are ordered along some *time-line*. Furthermore, in some protocols, we also assume *synchronous clocks*, where each party has access to the current value of the time  $t$ . There are many academic works which study more complex and less intuitive models. In particular, in reality, clocks are not precisely synchronized; indeed, ensuring good clock synchronization is not trivial, and doing it securely, in spite of different attacks, is even more challenging. Even the assumption that all events can be mapped along the same ‘universal time-line’ is arguably incompatible with relativistic physics; there is extensive research on ‘local time’ models, where events in different locations are only partially ordered.

#### 5.1.1 Inputs and outputs signals

It is convenient to define a protocol operation for each *type* of input, e.g., an initialization *INIT* signal. Most input signals include an associated input *value*; for example,  $INIT(\kappa)$  may indicate initialization with shared secret  $\kappa$ . Another typical input signal is  $IN(\mu)$ , which signals receipt of message  $\mu$  from the network.

Similarly, it is often convenient to consider the output of the protocol as one or more (type, value) pairs of *output signals*. For example,  $OUT(\mu, d)$  signals that the protocols outputs message  $\mu$ , to be sent by the underlying network to destination  $d \in P$ .

Many protocols assume *synchronized clocks*. This is facilitated by a *WAKEUP* input signal, invoked periodically, or upon termination of a ‘sleep’ period, requested by the protocol in a previous *SLEEP(t)* output signal.

Specific protocols have different additional input and output signals. These are mostly used for the input and output for the services provided by  $\pi$ . For example, *handshake protocols* secure the setup and termination of sessions, using *OPEN* and *CLOSE* input signals to open/close a session, and providing *UP* and *DOWN* output signals to indicate the state of a session; see subsection 5.2.1. In contrast, *session protocols* secure the transmission of messages, and hence have a  $SEND(m, \sigma)$  input signal, invoked to request the protocol to send a message  $m$  over session  $\sigma$ , and a  $DELIVER(m, \sigma)$  output signal, to deliver a message  $m$  received over session  $\sigma$ .

*Conventions:* We use italics, e.g.,  $m$ , for protocol-specific inputs and outputs, such as messages sent and received *using* the protocol, and Greek font, e.g.,  $\mu$ , for ‘lower-layer’ messages, that are output *by* the protocol (in  $OUT(\mu, d)$  events) or that the protocol receives as input from the network (in  $IN(\mu)$  events). We write signal names in capital letters.

### 5.1.2 Focus: two-party shared-key handshake and session protocols

In this chapter we discuss shared-key protocols, i.e., protocols between two parties,  $P = \{Alice, Bob\}$ . Both parties are initialized with a shared symmetric key  $\kappa$ . In the following chapters, we discuss protocols using public keys, in addition or instead of shared keys; we also discuss protocols using passwords or other low-entropy secrets.

A session begins by executing a *handshake protocol*, which involves authentication (of one or both parties), and often also agree on a shared *session key*; handshake protocols are the main topic of this chapter. A successful handshake is usually followed by exchange of one or more messages between the parties, using a *session protocol*. We discuss session protocols in subsection 5.1.4.

Handshake and session protocols are usually quite simple and easy to understand - but this simplicity can be deceptive, and many attacks have been found on proposed, and even widely deployed, protocols. Most attacks are surprisingly simple in hindsight. Furthermore, most attacks - esp. on handshake protocols - do not involve exploiting vulnerabilities of the underlying cryptographic schemes such as encryption and MAC. Instead, these attacks exploit the fact that designers did not use the schemes correctly, e.g., used schemes for goals they are not designed for - e.g., assuming that encryption provides authentication. There have been many attacks on such insecure, vulnerable handshake protocols, in spite of their use of secure cryptographic primitives. Therefore, it is critical to clearly define the adversary model and conservative security goals, and to have proof of security under reasonable assumptions - or, at least, careful analysis.

### 5.1.3 Adversary Model

Most of the works on design and analysis of cryptographic protocols, and in particular of handshake and session protocols, adopts a *Monster in the Middle (MitM)* attack model. A MitM<sup>1</sup> attacker has *eavesdropper* capabilities, i.e., it is exposed to the contents of every message  $\mu$  sent over the network by any party in an  $OUT(\mu, d)$  signal, to any intended recipient  $d \in P$ .

In addition to the eavesdropper capabilities, a MitM attacker also *actively controls* the communication. Namely, the adversary has complete control over the occurrence and contents of  $IN(\mu)$  signals. This allows the attacker to *drop*, *duplicate*, *reorder* or *modify* messages, including causing receipt of messages  $\mu$  which were never sent.

In addition, we usually assume that the the attacker has complete or significant control over other input events, as well as receiving all or much of the output events. In fact, to define integrity-properties, such as message or handshake authentication, we usually allow the adversary complete control over all

---

<sup>1</sup>Many works refer to the MitM model as *Man in the Middle*, but we prefer the term *Monster in the Middle*. This allows us to use the pronoun ‘It’ to refer to the attacker, in contrast to Alice and Bob.

the input to the protocol (e.g., *OPEN* signals), and provide the adversary with all the outputs of the protocol. For example, see Definition 5.2, which defines mutual-authentication for handshake protocols.

In contrast, to define confidentiality properties, we slightly restrict the adversary's access to output events, and control over input events. For example, we define IND-CPA security for session protocols by allowing the attacker to choose arbitrary ‘plaintext’ messages  $m$ , as well as to select a pair of special ‘challenge’ messages  $m_0, m_1$ . We then select a random bit  $b \in \{0, 1\}$ . The attacker then observes the encoding of the chosen messages  $m$  as well as of the ‘challenge’ message  $m_b$ . Finally, the attacker guesses the value of  $b$ . This essentially extends the IND-CPA security definition for encryption schemes, Definition 2.8

Finally, in protocols involving multiple parties, we may also let the adversary control some of the parties running the protocols; this is less relevant to two-party protocols which are our focus in this chapter. We will, however, discuss scenarios where the attacker may have access to some of the storage, including keys, used by some of the parties.

#### 5.1.4 Secure Session Protocols

A *secure session transmission protocol* defines the processing of messages as they are sent and received between two parties, over an insecure connection; such protocols are often referred to simply as *session protocol* or as *record protocol* (since they specify handling of a single message, also referred to as record). Session transmission protocols use a secret key shared between the parties; in principle, one could also use public-private key pairs, however, public-key cryptography is much more computationally expensive, hence, the use of public-key is normally limited to the session-setup (handshake) protocols, used to initiate the session - and sometimes also periodically, to refresh the keys and begin new instance of the session transmission protocol.

Session transmission protocols are a basic component in any practical system for secure communication over insecure channels, such as Transport-Layer Security (TLS) and Secure Socket Layer (SSL) protocols, used to protect many Internet applications including web communication, and the IPsec (IP security) protocol, used to protect arbitrary communication over the Internet Protocol (IP).

A session consists of three phases: it is *opened*, then *active* and finally *closed*, with indication of normal or abnormal (error) termination. Messages are sent and received only when session is active. We use these concepts to (informally) define the main security goals of session protocols.

**Definition 5.1** (Goals of session protocols). **Confidentiality.** *The confidentiality goal is normally similar to the definition for encryption schemes. Namely, the adversary chooses two challenge messages  $m_0, m_1$ , and should not be able to identify which of the two was randomly chosen and sent, with probability significantly better than half - similarly to the CPA-indistinguishability*

*(CPA-IND, Def. 2.8) test. The main difference is that we allow a stateful protocol instead of stateless scheme.*

**Message authentication: universal forgery capabilities.** To test a protocol for ensuring message authentication, we would allow the adversary to ask parties to send arbitrary messages. The MitM attacker should not be able to cause receipt of a message that it never asked a party to send - similarly to the security against forgery (Def. 3.1) test.

**Session mapping.** There is a one-to-one mapping from every session where one party sent or received a message, to a session at the corresponding other party, s.t. the messages received in each of the two sessions session, are identical, in content and order, to the corresponding messages sent in the corresponding session.

**Detecting truncation.** If a session at Alice is mapped to a session at Bob, but Bob sent some messages that Alice did not receive, than the session at Alice did not yet terminate - or, terminated with failure indication.

The obvious and common way to ensure secure order is by authenticating a sequence number *Seq*, or some other mechanism to prevent re-ordering, typically, as part of the authentication of the message. Often, the underlying communication channel preserves order of messages, and hence the sender and recipient can maintain *Seq* and it is not necessary to include *Seq* explicitly in the messages; for example, this is done in SSL/TLS. In other cases, the sequence number is sent in the clear, i.e., authenticated but not encrypted; this allows ordering and identification of losses - even by intermediate agents who cannot decrypt. Other fields that are often sent authenticated but ‘in the clear’, i.e., not encrypted, include the sender and recipient addresses; usually, these fields are grouped together, and referred to as the *header*.

Note that ensuring secure order, by authentication of messages and sequence numbers, does not suffice to prevent *truncation*, i.e., loss of the very last messages from one party to the other. To prevent truncation, the session protocol usually terminates with a special ‘termination exchange’, allowing a party to detect when an attacker dropped some message from its peer.

In addition to the security goals, session-transmission protocols have the following reliability and efficiency goals:

**Error detection and/or correction,** , typically using Error Detection Code (EDC) such as Checksum, or Error Correction Code (ECC), such as Reed-Solomon codes. These mechanisms are designed against random errors, and are not secure against intentional, ‘malicious’ modifications. Note that secure message authentication, such as using MAC, also ensures error detection; however, as we explain below, it is often desirable to also use the (insecure) error detection codes.

**Compression,** to improve efficiency by reducing message length, usually provided by applying a compression code. As we explain below, this requirement may conflict with the confidentiality requirement.

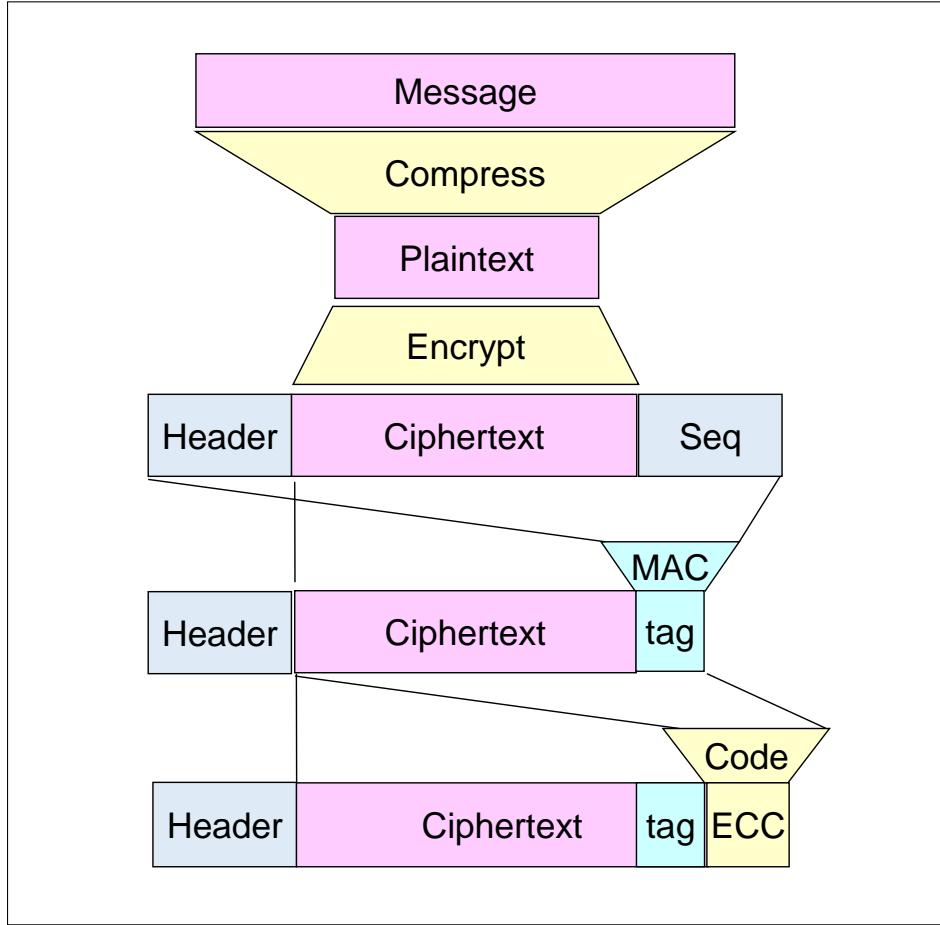


Figure 5.1: Secure Session Sending Process: Security, Reliability and Compression.

Fig. 5.1 presents the recommended process for combining these functions, justified as follows:

- Compression is only effective when applied to data with significant redundancy; plaintext is often redundant, in which case, applying compression to it could be effective. In contrast, ciphertext would normally not have redundancy. Hence, if compression is used, it must be applied before encryption. Note, however, that this may conflict with the confidentiality requirement, as we explain below; for better confidentiality, avoid compression.
- Encryption is applied next, before authentication (MAC), following the ‘Encrypt-then-Authenticate’ construction. Alternatively, we may use an authenticated-encryption with associated data (AEAD) scheme, to com-

bine the encryption and authentication functions. Notice that by applying authentication after encryption or using an AEAD scheme, we facilitate also authentication of sequence-number or similar field used to prevent re-play/re-order/omission, which is often known to recipient (and not sent explicitly); we can also authenticate ‘header’ fields such as destination address, which are not encrypted since they are used to process (e.g., route) the encrypted message; see Fig. 5.1. The Encrypt-then-Authenticate mode also allows prevention of chosen-ciphertext attacks and more efficient handling of corrupted messages.

- Finally, we apply error correction / detection code. This allows efficient handling of messages corrupted due to noise or other benign reasons. An important side-benefit is that authentication failures of messages to which errors were not detected, imply an intentional forgery attack - attacker made sure that the error-detecting code will be correct.

**Compress-then-Encrypt Vulnerability** Note that there is a subtle vulnerability in applying compression before encryption, since encryption does not hide the length of the plaintext, while the length of compressed messages depends on the contents. In particular, a message containing randomly-generated strings typically does not compress well (length after compression is roughly as long as before compression), while messages containing lots of redundancy, e.g., strings composed of only one character, compress well (length after compression is much shorter). This allows an attacker to distinguish between the encryptions of two compressed messages, based on the redundancy of the plaintexts; see next exercise, as well as Exercise 5.13. This vulnerability was exploited in several attacks on the TLS protocol including CRIME, TIME and BREACH.

**Exercise 5.1.** Let  $(Enc, Dec)$  be an IND-CPA secure encryption scheme, and let  $Enc'_k(m) = Enc_k(\text{Compress}(m))$ , where  $\text{Compress}$  is a compression function. Show that  $Enc'$  is not IND-CPA secure.

**Organization of the rest of this chapter** In § 5.2 we discuss *shared-key authentication-handshake protocols*, which provide *mutual entity authentication* of the two parties communicating - or only of one of them. In § 5.3 we discuss *session-authentication handshake* protocols, which provide message/session authentication in addition to mutual authentication, namely, authenticate messages exchanged as part of the handshake, in addition to authenticating the parties.

In § 5.4, we discuss *key-setup handshake protocols*, which provide, in addition to authentication, also a shared *session key*  $k$ . The session key  $k$  is later used by *session protocol* to protect the communication between the parties. In ?? we discuss the *Key-Reinstallation Attack (KRACK)*, an interesting attack which is effective against multiple key-setup handshake protocols, most notably against the *WPA2* handshake.

Table 5.1: Signals of shared-key entity-authenticating handshake protocols, for session  $\sigma$ . The first three signals are common to most protocols; the rest are specific to handshake protocols.

Signal	Type	Party	Meaning
$INIT(\kappa)$	Input	Both	Initialize with shared key $\kappa$
$IN(\mu)$	Input	Both	Message $\mu$ received via the network
$OUT(\mu, d)$	Output	Both	Send message $\mu$ to $d \in P$ via network
$OPEN(p_R)$	Input	Initiator	Open session to $p_R \in P$
$BIND(p, \sigma)$	Output	Both	Begin handshake to $p \in P$ ; identifier $\sigma$
$UP_\rho(\sigma)$	Output	$\rho$	Session $\sigma$ is up at $\rho \in \{I, R\}$ (Initiator, Responder)
$CLOSE(\sigma)$	Input	Both	Close session $\sigma$
$DOWN_\rho(\sigma, \xi)$	Output	$\rho$	Session $\sigma$ terminated; outcome $\xi \in \{OK, Fail\}$

In § 5.5 we briefly discuss *Key Distribution* protocols, where keys are setup with the help of a third party. We mostly focus on the *GSM security protocols*, which provide client-authentication and keying for confidentiality for the GSM mobile network. We discuss two serious vulnerabilities of the GSM handshake: *replay attacks*, facilitated by client-only authentication, and *downgrade attacks*, facilitated by insecure *ciphersuite negotiation*.

In § 5.6, we discuss variants designed to provide (limited) *resiliency to exposures* of secret information (keys).

## 5.2 Shared-key Entity-Authenticating Handshake Protocols

Shared-key Entity-Authenticating Handshake Protocols authenticate one party to another, or mutually-authenticate both parties; in both cases, authentication is done using a key  $\kappa$  shared between the parties (in  $INIT(\kappa)$  signal). The handshake is usually initiated at one party by a request from the user or application; we refer to this party as *initiator* ( $I$ ). In the other party, the *responder* ( $R$ ), the handshake usually begins upon receiving the first message  $\mu$ , via the network, from the initiator.

### 5.2.1 Shared-key entity-authenticating handshake protocols: signals and requirements

The application at the initiator initiates a new handshake session, by invoking the  $OPEN(p_R)$  input signal of the handshake protocol. The protocol at the initiator responds to the  $OPEN(p_R)$  request by a  $BIND(p_R, \sigma)$  output signal, where  $\sigma$  is a *session identifier* used to refer to this session.

The initiator also begins the exchange of protocol-messages with the responder, with the initiator  $p_I$  sending a message  $\mu_I$  using the  $OUT(\mu_I, p_R)$  output event, and the responder  $p_R$  sending back a message  $\mu_R$  using  $OUT(\mu_R, p_I)$ .

If the adversary allows the exchange, this results in  $IN(\mu_I)$  and  $IN(\mu_R)$  input events, at the responder  $p_R$  and initiator  $p_I$ , respectively.

If the messages are exchanged correctly, the responder  $p_R$  will also output a  $BIND(p_I, \sigma)$  signal, indicating the successful initiation of a handshake session by initiator  $p_I$ , where  $\sigma$  is the *same session identifier* as in the corresponding  $BIND(p_R, \sigma)$  output signal previously output by the initiator  $p_I$  to refer to this session.

The session identifier  $\sigma$  is used in following input and output signals to refer to this particular session, similarly to a socket/file handles in used in the socket API. For simplicity, assume that session identifiers are unique, i.e., the protocol never opens two sessions with the same identifier  $\sigma$ , not even consecutively.

Following the  $BIND$  signal, the protocol at both parties continues to exchange messages, using  $OUT$  signals, to perform the authentication. When the entity authentication is successful at a party  $\rho \in I, R$ , then the protocol invokes the  $UP_\rho(\sigma)$  output signal, indicating that the session is *active*. Messages are sent, usually using a *session protocol*, only while the session is active.  $OPEN(p_R)$ .

The protocol outputs signal  $DOWN_\rho(\sigma, \xi)$  to indicate termination of session  $\sigma$  at party  $\rho \in I, R$ . The  $\xi \in \{OK, Fail\}$  signals successful (OK) or unsuccessful (Fail) termination. Successful termination is always the result of a previous  $CLOSE(\sigma)$  input signal, requesting the protocol to terminate session  $\sigma$ ; unsuccessful (Fail) termination is due to failure, and does not require a previous  $CLOSE$  input signal.

**Security of shared-key authentication-handshake protocol.** We next define security of shared-key authentication-handshake protocols. A protocol ensures responder authentication , if every session which is active (UP) at initiator, is at least *OPENed* at the responder. Initiator authentication requires the ‘dual’ property for the responder. The following definition attempts to clarify these notions, hopefully without excessive formalization.

**Definition 5.2** (Mutually entity-authenticating shared-key handshake protocol). *An execution of handshake protocol is said to be responder-authenticating is whenever any initiator  $p_I \in P$  signals  $UP_I(\sigma)$ , there was a previous  $BIND(p_R, \sigma)$  output signal at some responder  $p_R \in P$ . Similarly, an execution is said to be initiator-authenticating if whenever some responder  $p_R \in P$  signals  $UP_R(\sigma)$ , there was a previous  $BIND(p_R, \sigma)$  signal at some initiator  $p_I \in P$ .*

*A two-party shared key handshake protocol ensures responder (initiator) authentication, if for every PPT adversary  $ADV$ , the probability of an execution not to be respnder (respectively, initiator) authenticating is negligible (as a function of the length of the shared key  $\kappa$ ).*

*A handshake protocol is mutually authenticating if it ensures both responder and initiator authentication.*

The rest of this section deals with secure and vulnerable designs of mutual-authentication handshake protocols. Achieving a secure design is not very

### Note 5.2: More rigorous treatment and sessions without explicit identifiers

There is a wealth of literature and research on the topics of authentication, key setup and key distribution protocol, which is far beyond our scope; some of the basic works include [12, 15, 18, 21, 22]. In particular, most of the formal works on secure handshake protocols avoid the assumption of explicit session identifiers. This is obviously more general, and also somewhat simplifies the definitions of the signals. However, practical protocols do use explicit session identifiers, which are ‘practically unique’; we find that requiring the protocol to produce such unique identifiers, simplifies the definitions and discussion of security.

difficult, however, it has to be done carefully, since it is also easy to come up with vulnerable design.

**Consecutive vs. concurrent authentication** Support for multiple concurrent handshakes is a critical aspect of handshake protocols. Some protocols only allow consecutive handshakes between each pair of initiator and responder. We refer to such protocol as *consecutive handshake protocols*, and to protocols that allow concurrent handshake as *concurrent handshake protocols*.

One can transform a concurrent handshake protocols into a consecutive handshake protocol, by simply blocking *OPEN* requests until the corresponding *DOWN* response. We say that a (potentially concurrent) handshake protocol *ensures consecutive mutual authentication*, if the consecutive version of the protocol ensures mutual authentication. Consecutive initiator and responder authentication are defined similarly.

It is easier to ensure consecutive authentication - there are significantly less ways to attack it. However, realistic protocols are usually designed and used with such restriction, i.e., allowing concurrent handshakes. This makes sense; applications sometimes use concurrent parallel sessions between the same two parties, and furthermore, concurrent handshakes are necessary to prevent ‘lock-out’ due to synchronization-errors (e.g., lost state by Initiator), or an intentional ‘lock-out’ by a malicious attacker, as part of a *denial-of-service attack*. In any case, the consecutive handshakes restriction is not really essential; therefore, the conservative design principle (Principle 6) indicates we should always assume concurrent handshakes are possible.

When designers neglect to consider the threats due to concurrent sessions, yet the protocol allows concurrent sessions, the result is often a vulnerable protocol. This is a typical example of the results of failures to articulate the requirements from the protocol and the adversary model. We next present an example of such vulnerability: the SNA handshake protocol. We believe that the designers of the SNA handshake protocol considered only sequential handshakes, although SNA implementations allows concurrent sessions.

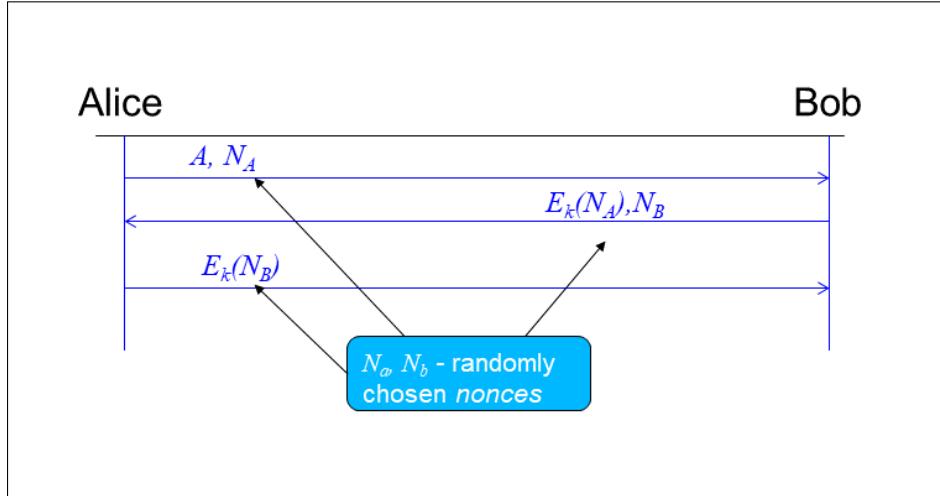


Figure 5.2: SNA two-party mutual authentication handshake protocol

### 5.2.2 The (insecure) SNA entity-authenticating handshake protocol

As a simple, yet realistic, example of a two-party, shared key mutual authentication handshake protocol, consider the *SNA handshake protocol*. IBM's SNA (Systems Network Architecture) was the primary networking technology from 1974 till the late 1980s, and is still in use by some 'legacy' applications. Security in SNA is based on a 'master key'  $k$  shared between the two communicating parties.

We describe the insecure version of the SNA entity-authenticating handshake protocol, and later its replacement - the 2PP entity-authenticating handshake protocol. Both protocols use a shared secret key  $k$ , to (only) authenticate two parties to each other, without deriving a session key. We first explain the protocol, illustrated in Fig. 5.2, and then discuss its security.

The SNA handshake protocol operates in three simple flows, as illustrated in Fig. 5.2. The protocol uses a block cipher  $E$ . The initiator, say Alice, sends to its peer, say Bob, three values: her identifier, which we denote  $A$ , the session identifier  $\sigma$ , and  $N_A$ , a random  $l$ -bit binary string which serves as a challenge (*nonce*). Here,  $l$  is the size of the inputs and outputs to a block cipher  $E$  used by the protocol.

The responder, say Bob, replies with a 'proof of participation'  $E_k(N_A)$ , using the pre-shared key  $k$ . Bob also sends his own random  $l$ -bit challenge (*nonce*)  $N_B$ , and  $\sigma$ , which allows Alice to match the response to the request, and  $N_B$ .

Upon receiving Bob's response, Alice validates that the response contains the correct function  $E_k(N_A)$  of the nonce that it previously selected and sent. If so, Alice concludes that it communicates indeed with Bob. Alice then completes

the handshake by sending its own ‘proof of participation’  $E_k(N_B)$ , with the session identifier  $\sigma$ .

Finally, Bob similarly validates that it received the expected function  $E_k(N_B)$  of its randomly selected nonce  $N_B$ , and concludes that this handshake was initiated by Alice.

Both parties, Alice and Bob, signal successful completion of the handshake, using  $UP(\sigma)$ , (only) upon receiving the expected response ( $E_k(N_A)$  for Alice and  $E_k(N_B)$  for Bob).

Readers are encouraged to transform the description above and in Fig. 5.2 to pseudocode.

**SNA handshake ensures (only) consecutive mutual authentication.** The simple SNA handshake of Fig. 5.2 is *secure if restricted to consecutive handshake* - but is *vulnerable allowing concurrent handshakes*, as shown in the following exercises.

Let us explain why the protocol ensures mutual authentication, when restricted to a single-concurrent-session. Suppose, first, that Alice completes the protocol successfully. Namely, Alice received the expected second flow,  $E_k(N_A)$ . Assume that this happened, *without* Bob previously receiving  $N_A$  as first flow from Alice (and sending  $E_K(N_A)$  back). Due to the consecutive restriction, Alice surely did not compute  $E_K(N_A)$ . Hence, the adversary must have computed  $E_K(N_A)$ , contradicting the PRP assumption for  $E$ . Note that an eavesdropping attacker may collect such pairs  $(N_A, E_k(N_A))$  or  $(N_B, E_k(N_B))$ , however, since  $N_A, N_B$  are quite long strings (e.g., 64 bits), the probability of such re-use of same  $N_A, N_B$  is negligible.

**Exercise 5.2** (SNA handshake fails to ensure concurrent mutual authentication). *Show that the SNA handshake protocol does not ensure concurrent mutual authentication.*

*Sketch of solution:* see Fig 5.3. □

**Exercise 5.3.** *Does the SNA handshake protocol ensure concurrent responder authentication? Explain, using a sequence diagram.*

*Hint for solution:* recall that the attacker is allowed to instruct an initiator to *OPEN* sessions. In the attack, it suffices to show how Eve causes Alice to think she had *two* sessions with Bob, while Bob was involved only in *one* session. Finding the right sequence to cause this is the main challenge in this question. □

Note that the SNA handshake fails even to ensure consecutive authentication, if  $E$  is a (shared key) encryption scheme rather than a block cipher.

**Exercise 5.4.** *Assuming that Alice and Bob each participate in no more than one session at a given time. Show that the SNA authentication protocol (Fig. 5.2) may not ensure (even) consecutive mutual authentication, if the function  $E$  is implemented using an (IND-CPA secure) symmetric encryption scheme  $E$ .*

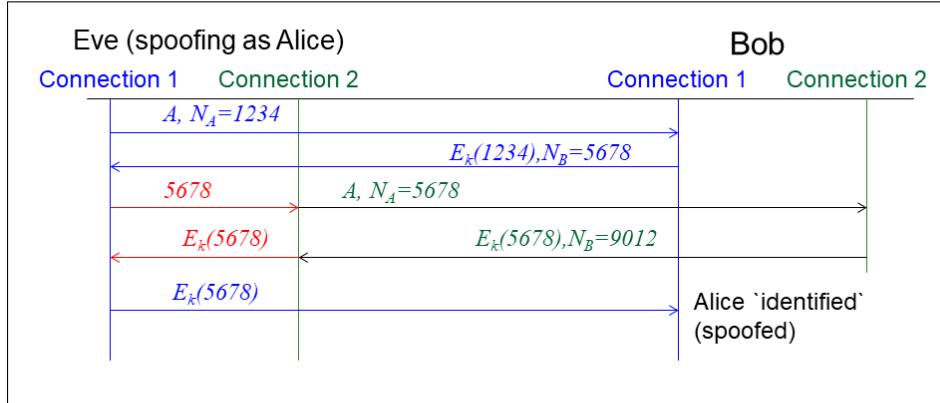


Figure 5.3: Example of SNA handshake failing to ensure concurrent initiator authentication. Eve pretends to be Alice, who is not present at all during the communication. To do this, Eve uses two connections.

*Hint:* Consider the different modes-of-operation construction of an encryption scheme from a block cipher, and encryption of a single-block message (such as  $N_A$  or  $N_B$ ). For one or more of these, the attacker could impersonate as Alice by sending  $N_A$ , receiving  $N_B$  and  $E_k(N_A)$ , and then *computing*  $E_k(N_B)$  by using only this information - *without* knowing the key  $k$ !  $\square$

Before we present secure protocols, it is useful to identify weaknesses of the SNA protocol, exploited in the attacks on it, and derive some *design principles*:

- Encryption and block-cipher (PRP) do not ensure authenticity. *To authenticate messages, use a MAC function!*
- The SNA protocol allowed *redirection*: giving to *Bob* a value from a message which was originally sent - in this case, by *Bob* himself - to a different entity (Alice). To prevent redirection, we should identify the party in the challenge, or even better, use separate keys for each direction.
- Prevent replay and reorder. The SNA attack sent to Bob a first flow, which contained part a message sent (by Bob) to Alice, during the second flow. To prevent this, the protocol should identify flow.

### 5.2.3 2PP: Three-Flows authenticating handshake protocol

We next present *2PP*, a secure two party shared-key authenticating handshake protocol; the name *2PP* simply stands for *two party protocol*. The 2PP protocol was a replacement to the SNA handshake protocol, proposed in [22].

Following the weaknesses identified above in the SNA handshake protocol, the 2PP handshake follows the following *design principles*:

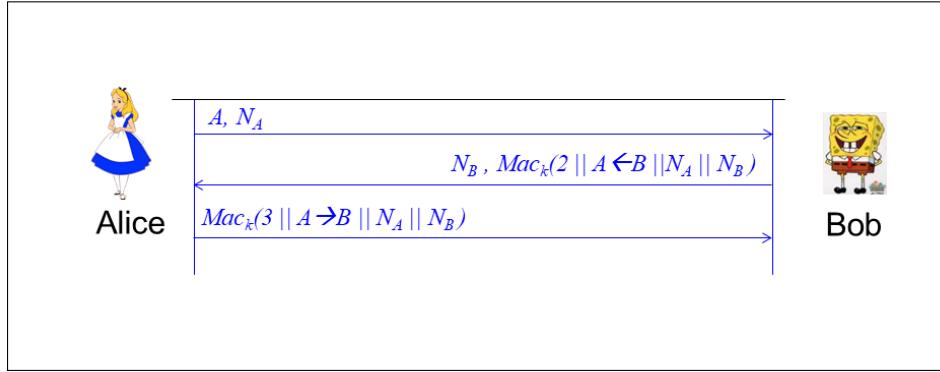


Figure 5.4: 2PP: Three-Flows Authenticating Handshake Protocol.

**Use the correct cryptographic mechanism needed.** Specifically, 2PP uses Message Authentication Code (MAC) for authentication, rather than relying on block ciphers or encryption, as done by the SNA authentication protocol.

**Prevent redirection:** include identities of the parties (A,B) as part of the input to the authentication function (MAC).

**Prevent reorder:** Separate 2nd and 3rd flows: 3 vs. 2 input blocks.

**Prevent replay and do not provide ‘oracle’ to attacker:** authenticated data (input to MAC) in a flow, always includes a random value (*nonce*) selected by the recipient in a previous flow.

The flows of the 2PP handshake are presented in Figure 5.4. The values  $N_A$  and  $N_B$  are called *nonce*; each is a  $l$ -bit string, where  $l$  is the length of the shared key  $\kappa$ , and often referred to as the *security parameter*. The nonces  $N_A, N_B$  are selected randomly, by Alice (initiator) and Bob (responder), respectively.

The 2PP handshake uses  $\sigma = N_A || N_B$  as the session identifier; this ensures the uniqueness of the session identifier  $\sigma$ . The protocol, at both ends, outputs *BIND* once it receives the first flow from the peer (so it can construct  $\sigma = N_A || N_B$ ), and *OPEN* upon receiving the (correct) last flow from the peer. By validating this last flow, 2PP ensures mutual authentication (Def. 5.2). If interested, see Lemma 5.1 and its proof, both in Note 5.3.

### 5.3 Session-Authenticating Handshake Protocols

Works on cryptographic protocols, including handshake and session protocols, usually adopt the Monster-in-the-Middle (MitM) adversary model, see subsection 5.1.3. In this case, *entity* authentication is often insufficient; the actual messages exchanged between the parties should be protected. In this section,

Note 5.3: Proof that 2PP ensures mutual authentication

**Lemma 5.1** (2PP ensures mutual authentication). *The 2PP protocol, as in Figure 5.4, is mutually authenticating.*

*Proof:* Recall that in 2PP,  $\sigma = N_A \parallel N_B$ . Supposed that there exist a PPT algorithm ADV that results, with significant probability, in executions which are not responder authenticating, i.e., where some initiator, e.g., Alice ( $A \in P$ ), which signals  $UP_A(N_A \parallel N_B)$ , without a previous  $BIND(Alice, N_A \parallel N_B)$  at responder Bob.

Alice signals  $UP_A(N_A \parallel N_B)$  only upon receiving  $MAC_\kappa(2 \parallel A \leftarrow B \parallel N_A \parallel N_B)$ . When does Bob send this? Only in runs in which Bob received the first flow  $A, N_A$ , supposedly from Alice, and then selected  $N_B$  randomly. In this case, Bob would also signal  $BIND(Alice, N_A \parallel N_B)$ ; but we assumed that (with significant probability) Bob did not signal  $BIND(Alice, N_A \parallel N_B)$ .

We conclude that (with significant probability), Bob didn't send the message  $2 \parallel A \leftarrow B \parallel N_A \parallel N_B$  (and  $MAC_\kappa(2 \parallel A \leftarrow B \parallel N_A \parallel N_B)$ ). Notice that Alice definitely has never computed  $MAC_\kappa$  over  $2 \parallel A \leftarrow B \parallel N_A \parallel N_B$ , simply since, in 2PP, Alice only computes  $MAC$  over messages containing  $A \rightarrow B$ , never on messages containing  $A \leftarrow B$ .

Yet, our assumption was that ADV, with significant probability, creates execution where Alice signals  $UP_A(N_A \parallel N_B)$ , implying that at some point during the execution, Alice received  $MAC_\kappa(2 \parallel A \leftarrow B \parallel N_A \parallel N_B)$ , although neither her nor Bob ever invoked  $MAC_\kappa$  over this message. Since ADV is a MitM, surely this message is available to ADV.

We can now use ADV as a routine of another PPT adversary  $ADV^{MAC}$ , which is able to output this  $MAC_\kappa$  value, without calling an oracle with this particular value, hence contradicting the assumption that  $MAC$  is secure against forgery (Definition 3.1). Adversary  $ADV^{MAC}$  simply runs ADV, until ADV outputs  $MAC_\kappa(2 \parallel A \leftarrow B \parallel N_A \parallel N_B)$  - with significant probability. The contradiction shows that our assumption of the existence of such ADV must be false.  $\square$

we study a minor extensions, where the handshake protocol does not only authenticate the entities, but also the *session*, including the messages exchanged between the parties.

### 5.3.1 Session-authenticating handshake: signals, requirements and variants

A session-authenticating handshake protocol includes additional signals, allowing initiator and responder to exchange one or few messages on a specific session  $\sigma$ . Specifically, the protocol at both parties has an additional input signal  $SEND(\sigma, m)$ , requesting to send message  $m$  to the peer over session  $\sigma$ , and the additional output signal  $RECEIVE(\sigma, m)$ , to deliver message  $m$  received from the peer over  $\sigma$ . For simplicity, the message-authenticating handshake protocols we discuss in this subsection send at most one message over each session per participant; session protocols, discussed later, extend this to transmission of multiple messages.

A handshake protocol *ensures session authentication* if it ensures mutual entity-authentication Definition 5.2, and furthermore, a message is received at one party, say Bob, at session  $\sigma$ , only if it was sent by the peer party, say Alice, in the same session  $\sigma$ , and after the session  $\sigma$  was *OPENed* at Bob. Furthermore, if a session  $\sigma$  terminates successfully at one party, e.g., with  $DOWN_I(\sigma, OK)$ , after a message was sent, i.e.,  $SEND(\sigma, m)$  input signal, then that message was previously successfully received by the peer in this session  $\sigma$ , i.e.,  $RECEIVE(\sigma, m)$  output signal occurred earlier.

Note that this definition builds on the mapping that exists between sessions at both ends, as required from mutual entity-authenticating protocol (Def. 5.2).

The session authentication property implies *message authentication*, i.e., messages received by one party were indeed sent by the other party. It also implies *freshness*, which means that messages are received *in FIFO order*, without any *reordering* or *duplications*; however, *omissions* may be permitted. Freshness also implies that when a message is received, say by Bob, in some session, that message was sent by Alice during the same session, after the session *OPENed* in Bob.

**Session-authenticating handshake: three variants.** We present three slightly-different session-authenticating handshake protocols. We begin with a session-authenticating variant of 2PP; like ‘regular’ 2PP, the session-authenticating 2PP variant also involves three flows. This protocol can authenticate one message from responder to initiator and one message from initiator to responder - however, the responder has to send its message *before* receiving the message from the initiator.

However, in the very common and important case of where a client device initiates a session and sends a request, and the server sends back a response, we need the *initiator (client)* to send the first message and the *responder (server)* to send back a response, which is not handled by message-authenticating 2PP. We present two other variants that support this; the first simply extends 2PP to an additional (fourth) flow, and the other assumes synchronized clocks, allowing the handshake to involve only one message from client to server and one response from server to client.

### 5.3.2 Session-authenticating 2PP

We first discuss the three-flows session-authenticating handshake protocol; this is a minor extension to the base 2PP, as shown in Figure 5.5. In fact, the only change is adding the messages ( $m_R$  from responder to initiator, and  $m_I$  from initiator to responder) to the second and third flows, respectively.

The three-flows session-authenticating 2PP has, however, a significant drawback, which makes it ill-suited for many applications. Specifically, in this protocol, the first message is sent by the *responder*, and only then the second message is sent, by the *initiator*. This does not fit the common ‘request-response’ interaction, where the initiator contacts the responder and sends to it some *request*, and the responder sends back a *response* to the request.

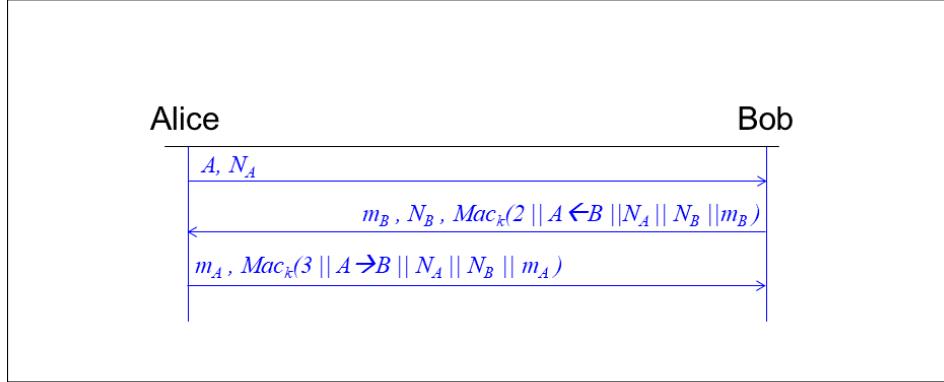


Figure 5.5: Three-flows session-authenticating 2PP handshake

In the next two subsections, we present two variants of message-authenticating handshake, that support sending a request by the initiator and receiving a response from the responder; we refer to such protocols as *request-response authenticating*. Note that each of these two variants have also a drawback, compared to the three-flows session-authenticating 2PP: either a fourth flow, or relying on synchronized time and/or state.

### 5.3.3 Nonce-based request-response authenticating handshake protocol

We now present another minor variant of 2PP, which also provides session authentication for a short message - a *request* message *req* from Alice to Bob, and a *response* message *resp* from Bob to Alice, using nonces. This protocol allows the initiator (Alice) to send its message - the *request* - first, and the responder sends its message - the *response* - only upon receiving the request. The protocol does not require the parties to use synchronized clocks, and while it does require them to remember each party to remember the (random) values of the nonces that this party picked, there is no need to maintain this state once the session ended. See Figure 5.6

The four-flows handshake involves two simple extensions of the basic 2PP protocol. The first extension is an additional, fourth flow, from responder back to initiator, which carries the response of the responder to the request from the initiator, which is sent as part of the third flow (which is the second flow from initiator to responder). The second extension is simply the inclusion of these messages - request and response - in the corresponding flows, and as inputs to the Message Authentication Code (MAC) applied and sent in both flows.

The disadvantage of this protocol is, obviously, the need for two additional flows, which also implied an additional ‘round trip’ delay until sending the request and response. We next present another request-response authenticating

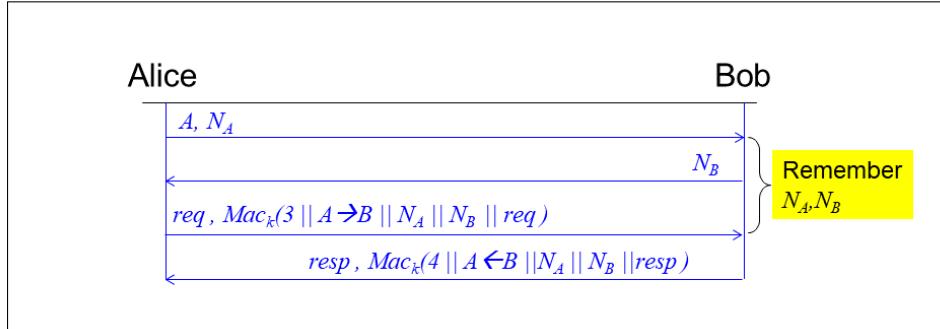


Figure 5.6: Four-flows, request-response authenticating handshake protocol

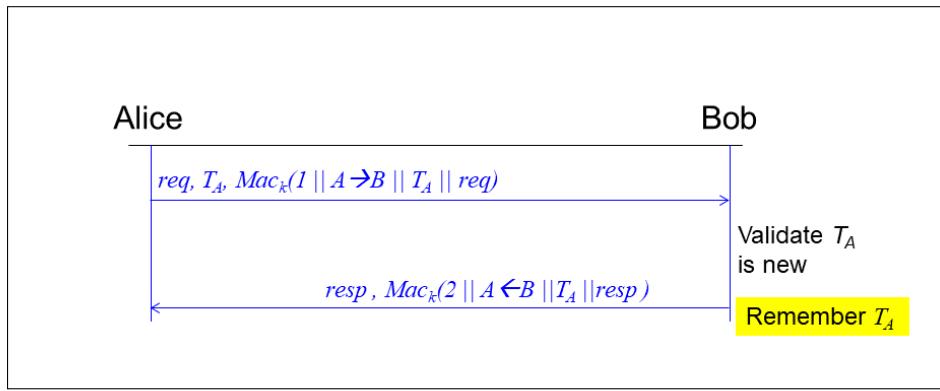


Figure 5.7: Timestamp-based Authenticated Handshake, assuming Synchronized Clocks

handshake, which requires only *two flows*, at the price of requiring synchronized clocks or state.

#### 5.3.4 Two-Flows Request-Response Authenticating Handshake, assuming Synchronized State

The 2PP protocol requires three flows, and the request-response authenticating variant presented above requires an additional flow. In contrast, Fig. 5.7 presents a simple alternative request-response authenticating handshake protocol, which requires only *two flows*.

The challenge in this protocol is for the responder to verify the *freshness* of the request, i.e., that the request is not a *replay* of a request already received in the past. Freshness also implies no *reordering*; for example, a responder  $p_R$  should reject request  $x$  from  $p_I$ , if  $p_R$  already received another request  $x'$  from  $p_I$ , where  $x'$  was sent after  $x$ . Freshness prevents an attacker from replaying information from previous exchanges. For example, consider the

request-response authentication of Figure 5.6; if  $N_B$  is removed (or fixed), then an eavesdropper to the flows between Alice and Bob in one request-response session, can copy these and cause the Bob to process the same request again. For some requests, e.g., *Transfer 100\$ from my account to Eve*, this can be a concern.

To ensure *freshness* without requiring the extra flows, one may use *Timestamps* instead of exchanging *nonces*. Notice that even if clocks are synchronized, there is some delay for messages, forcing recipients to allow messages which contain timestamps earlier than the time at which they were received. To deal with this, the recipient (e.g., Bob) typically remembers the last timestamp received, and processes only requests only with increasing timestamps. This memory can be erased after there has been no new request for enough time, allowing the recipient to confirm that a new request is not a replay. Still, when the delay is large, timestamps has the additional advantage that the recipient does not need to remember all ‘pending nonces’ (sent and not yet received), assuming synchronized clocks.

Timestamp-based authenticated handshake may operate as follows. The initiator includes with each request, say of session  $\sigma$ , the *timestamp* value  $T_A(\sigma)$ , which is guaranteed to *exceed* the value  $T_A(\sigma')$  sent with any previously-sent request sent from this initiator to this responder. One way to implement  $T_A$  is to set it, at time  $t$ , to  $T_A = \text{time}_A(t)$ , where  $\text{time}_A(t)$  gives the current value of a clock maintained by Alice at time  $t$ , where  $\text{time}_A(t)$  monotonously increases with time  $t$ . Another way for the initiator, e.g. Alice, to implement  $T_A$ , is using a counter maintained by Alice and incremented upon every *OPEN* or *SEND* event. However, often the requirement to maintain a persistent state, is harder than the requirement of keeping a clock.

The responder, e.g., Bob in Fig. 5.7, needs to maintain the last-received value of  $T_A$ , and ignore or reject requests which arrive with  $T_A$  values smaller than the last value it received. Even this state may be reduced if the responder also has a clock, synchronized - to some extent - with the initiator, as we show in the following exercise.

**Exercise 5.5.** Assume that the initiator (Alice) and responder (Bob) have synchronized clocks,  $\text{time}_A()$  and  $\text{time}_B()$ , whose values never differ by more than  $\Delta$  seconds, i.e. for every time  $t$  holds  $|\text{time}_A(t) - \text{time}_B(t)| \leq \Delta$ , and that the delay for message sent between Alice and Bob is at most  $D$ . Explain how to modify the protocol to reduce the persistent storage requirements, while maintaining security (and in particular, ensuring freshness).

## 5.4 Key-Setup Handshake

In this chapter, as in most of this course, and, in fact, in most of the work on cryptographic protocols, we adopt the MitM adversary model, as presented in subsection 5.1.3. A MitM adversary is able to eavesdrop on messages as well as to modify them. In a typical use of handshake protocol, the handshake is used only to securely setup a following exchange of messages - a *session*.

Therefore, if we consider a MitM adversary, we normally have to consider not just attacks on the handshake phase, but also following attacks on the session itself.

Later, in subsection 5.1.4, we discuss the session protocols, that protect the communication over a session from a MitM attacker; these protocols rely on a secret key shared between the participants in the session. In principle, the parties could use a fixed shared secret key for all sessions between them - e.g., one could simply use the shared initialization ('master') key  $\kappa$ . However, there are significant security advantages in limiting the use of each key, following Principle 2. Specifically:

- By changing the key periodically:, we reduce the amount of ciphertext using the same key available to the cryptanalyst, which may make cryptanalysis harder or infeasible.
- By changing session keys periodically, and making sure that each of the session keys remain secret (pseudo-random) even if all other session keys are exposed, we limit or reduce the damages due to exposure of some of the keys.
- The separation between session keys and master key, allows some or all security to be preserved even after attacks which expose the entire storage of a device. One way to achieve this is when the master key is confined to a separate *Hardware Security Module (HSM)*, protecting it even when the device storage is exposed. We later discuss solutions which achieve limited preservation of security following exposure, even without an HSM.

Handshake protocols are often used to setup a new, separate key for each session; when sessions are of limited length, and/or are limited in the amount of information carried over them, this provides the desired reduction in ciphertext associated with any single key. While such protocols are usually also ensuring mutual authentication, we now focus on the key-setup aspect.

#### 5.4.1 Key-Setup Handshake: Signals and Requirements

Key-setup handshake protocols require only a minor change to the  $UP$  signal, as described in subsection 5.2.1. Specifically, we only need to add the session key as part of the  $UP$  signal, as in:  $UP_\rho(\sigma, k)$ , to signal that session  $\sigma$ , at party  $\rho \in \{I, R\}$ , would use shared key  $k$ .

There are two security requirements from key-setup handshake protocols. The first requirement is that the *two parties agree on the same key*. For simplicity, we require this to hold in executions where the responder ( $p_R$ ) outputs  $UP_R(\sigma, k)$ ; this usually suffice, since in most protocols, e.g., 2PP, an  $UP_R$  event implies a previous  $UP_I$  event.

The second requirement from key-setup handshake is that *each session key would be secret*. More precisely, each session key should be pseudo-random, i.e., indistinguishable from a random string of same length, even if the adversary is given all the other session keys.

We say that a two-party shared-key handshake protocol *ensures secure key-setup*, if it ensures both requirements, i.e., it ensures synchronized key-setup as well as pseudo-random session keys.

#### 5.4.2 Key-setup 2PP extension

We next explain the *key-setup 2PP extension*, a simple extension to the 2PP protocol, that ensures secure key-setup. This is achieved by outputting, in the  $UP_I$  and  $UP_R$  signals, the session key as:

$$k = \text{PRF}_\kappa(N_A \parallel N_B) \quad (5.2)$$

In Eq. (5.2),  $\kappa$  denotes the long-term shared secret key (provided to both parties in the  $\text{INIT}(\kappa)$  input signal), and  $N_A, N_B$  are the values exchanged in the protocol.

Since both parties compute the key in the same way from  $N_A \parallel N_B$ , it follows that they will receive the same key, i.e., the key-setup 2PP extension ensures synchronized key-setup. Furthermore, since  $N_A$  and  $N_B$  are chosen randomly for each session, then each session identifier - the pair  $N_A \parallel N_B$  - is used, with high probability, in no more than a single session. Since the session keys are computed using a pseudo-random function,  $k = \text{PRF}_\kappa(N_A \parallel N_B)$ , it follows that the key of each session is pseudo-random (even given all other session keys). Namely, *the key-setup 2PP extension ensures secure key setup*.

#### 5.4.3 Key-Setup: Deriving Per-Goal Keys

Following the *key-separation principle* (principle 5), session protocols often use two separate keyed cryptographic functions, one for encryption and one for authentication (MAC); the key used for each of the two goals should be be pseudo-random, even given the key to the other goal. We refer to such keys are *per-goal keys*. The next exercise explains how we can use a single shared key, from the 2PP or another key-setup protocol, to derive such per-goal keys.

**Exercise 5.6** (Per-goal keys).

1. Show why it is necessary to use separately pseudorandom keys for encryption and for authentication (MAC), i.e., per-goal keys.
2. Show how to securely derive one key  $k_E$  for encryption and a key  $k_A$  for authentication, both from the same session key  $k$ , yet each key (e.g.,  $k_E$ ) is pseudo-random even given the other key (resp.,  $k_A$ ).
3. Show a modification of the key-setup 2PP extension, which also derives a secure, pseudo-random pair of keys  $k_E, k_A$ , but a bit more efficiently than by deriving both of them from  $k$ .

Explain the security of your solutions.

Hints for solution:

1. Assume you are given a secure encryption and MAC and ‘corrupt’ them into the required counter-example.
2. Let  $k_E = \text{PRF}_k('E')$ ,  $k_A = \text{PRF}_k('A')$ .
3. Instead of deriving  $k$  as in Eq. (5.2), derive  $k_E, k_A$  ‘directly’ using:

$$k_E = \text{PRF}_\kappa('E'||N_A||N_B) \quad (5.3)$$

$$k_A = \text{PRF}_\kappa('A'||N_A||N_B) \quad (5.4)$$

□

To further improve security of the session protocol, we may use *two separate pairs of per-goal keys*: one pair  $(k_E^{A \rightarrow B}, k_A^{A \rightarrow B})$  for (encryption, authentication) of messages from Alice to Bob, and another pair  $(k_E^{B \rightarrow A}, k_A^{B \rightarrow A})$  for (encryption, authentication) of messages from Bob to Alice.

- Exercise 5.7.**
1. How could the use of separate, pseudo-random pairs of per-goal keys for the two ‘directions’ improve security?
  2. Show how to securely derive all four keys (both pairs) from the same session key  $k$ .
  3. Show a modification of the key-setup 2PP extension, which securely derives all four keys (both pairs) ‘directly’, a bit more efficiently than by deriving them from  $k$ .

Explain the security of your solutions.

*Hints for solution of parts 2 and 3:* follow similar approach as for Ex. 5.6, but derive directly from the ‘master key’, saving a cryptographic operation. □

## 5.5 Key Distribution Protocols and GSM

In this section, we expand a bit beyond our focus on two party protocols, to briefly discuss shared-key, three-party *Key Distribution Protocols*. In general, key distribution protocols establish a shared key between two or more entities. We focus on Key Distribution Protocols which use only symmetric cryptography (shared keys), and involve only three parties: Alice, Bob - and a *trusted third party (TTP)*, often referred to as the *Key distribution Center (KDC)*, whose goal is to establish a shared key between the other parties. The KDC shares a key with each party:  $k_A$  with Alice and  $k_B$  with Bob; using these keys, the KDC helps Alice and Bob to share a symmetric key  $k_{AB}$  between them.

There are many types of Key Distribution Protocols. We present one typical, simple protocol in Figure 5.8. Later in this section, we will focus on a different key distribution protocol, which is used in the GSM cellular network standard - and which is notoriously insecure.

The process essentially consists of two exchanges. The first exchange is between Alice and the KDC. In this exchange, the KDC sends to Alice the key

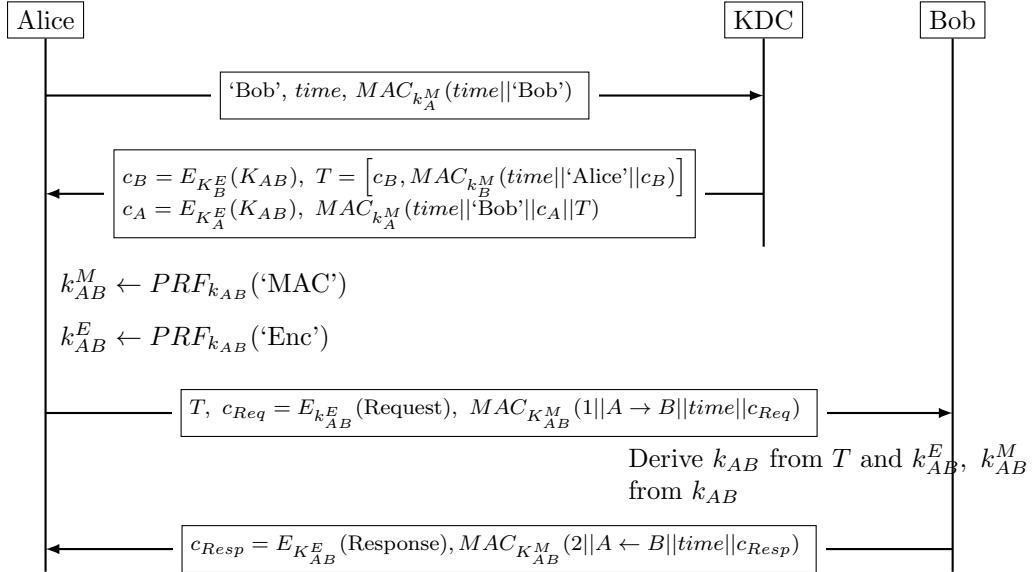


Figure 5.8: Key Distribution Center Protocol. The protocol assumes that the KDC shares two keys with each party; with Alice, it shares  $k_A^E$  for encryption and  $k_A^M$  for authentication (MAC), and with Bob, it shares  $k_B^E$  for encryption and  $k_B^M$  for authentication (MAC). In this protocol, KDC selects a shared-key  $k_{AB}$  to be used by Alice and Bob for the specific request-response. Alice and Bob use  $k_{AB}$  and a pseudo-random function  $PRF$  to derive two shared keys,  $k_{AB}^E = PRF_{K_{AB}}('Enc')$  (for encryption) and  $k_{AB}^M = PRF_{K_{AB}}('MAC')$  (for authentication, i.e., MAC).

$k_{AB}$  that will be shared between Alice and Bob. In addition, Alice receives a *ticket*  $T$ , which is, essentially, the key  $k_{AB}$ , encrypted and authenticated - for Bob.

In the second phase, Alice sends her request to Bob, together with the ticket  $T$ , allowing Bob to retrieve  $k_{AB}$ . Alice and Bob both derive from  $k_{AB}$  the shared encryption and authentication (MAC) keys,  $k_{AB}^E$  and  $k_{AB}^M$  respectively.

Note that in the above protocol, the KDC never initiates communication, but only *responds* to an incoming request; this communication pattern, where a server machine (in this case, the KDC) only responds to incoming request, is referred to as *client-server*. It is often preferred, since it relieves the server (e.g., KDC) from the need to maintain state for different clients, which makes it easier to implement an efficient service, esp. when clients may access different servers.

In many applications, the TTP has an additional role: *access control*. Namely, the TTP would control the ability of the client (Alice) to contact the service (Bob). In this case, the ticket does not only transport the key

but also becomes a *permit* for the use of the server. One important example for such a service is the *Kerberos* system [89], adopted in Windows and other systems.

This protocol requires and assumes *synchronized clocks (time)* between all parties. This allows Alice and Bob to validate that the key  $K_{AB}$  they receive is ‘fresh’ and not a replay; when tickets also serve as permits, this also allows Bob to validate that Alice was allowed access to the service (at the given time). If there are no synchronized clocks, then we must use a different flow than shown in Figure 5.8, in particular, Bob should be able to provide a nonce  $N_B$  to the KDC, and then to validate that that nonce  $N_B$  exists in the ticket that it receives. Such protocols were also studied and deployed, e.g., see [21].

**Exercise 5.8.** Extend the KDC protocol of Figure 5.8 to provide also access-control functionality by the KDC. Specifically, Alice should send her request also to the KDC, in authenticated and private manner; and the KDC would include a secure signal for Bob within the ticket  $T$ , letting Bob know that the KDC approves that request. Your solution should only use the flows in Figure 5.8, adding and/or modifying the information sent as necessary. The solution should avoid exposure of the request to the attacker (eavesdropper); the KDC, of course, should be aware of the request (to approve it).

*Hint:* notice that an extension to the protocol of Figure 5.8 is required, since in that protocol, the KDC does not even receive Alice’s request, and surely cannot ‘approve’ it.

**Exercise 5.9.** Present an alternative design for a KDC protocol, which avoids the assumption of synchronized clocks. Your solution should maintain client-server communication, i.e., the KDC (as a server) should only send responses to incoming requests, and never initiate communication with a client.

*Hint:* you may solve this by defining the contents of the flows in Fig. 5.9; and you may use ideas from the 2PP protocol. Or, a simpler solution may use additional state in the KDC, essentially, for counters.

The rest of this section focuses on a very specific, important case-study of a three-party key distribution protocol, namely, that of the GSM network, and some of its (very serious) vulnerabilities.

### 5.5.1 Case study: the GSM Key Distribution Protocol

We next discuss the *GSM security protocol*, another important-yet-vulnerable shared-key authentication and key-setup protocol. The GSM security protocol is performed at the beginning of each connection between a *Mobile* and a *Base*. The base is the cellular network provider used in this connection; the base typically does not maintain the mobile’s key  $k_i$ , which is known only to the mobile itself and to the *Home*, which is the cellular network provider to whom the mobile is subscribed. The home retrieves  $k_i$  from its *clients keys table CK*, but does not send it to the base. Instead, the home uses  $k_i$  to derive and send

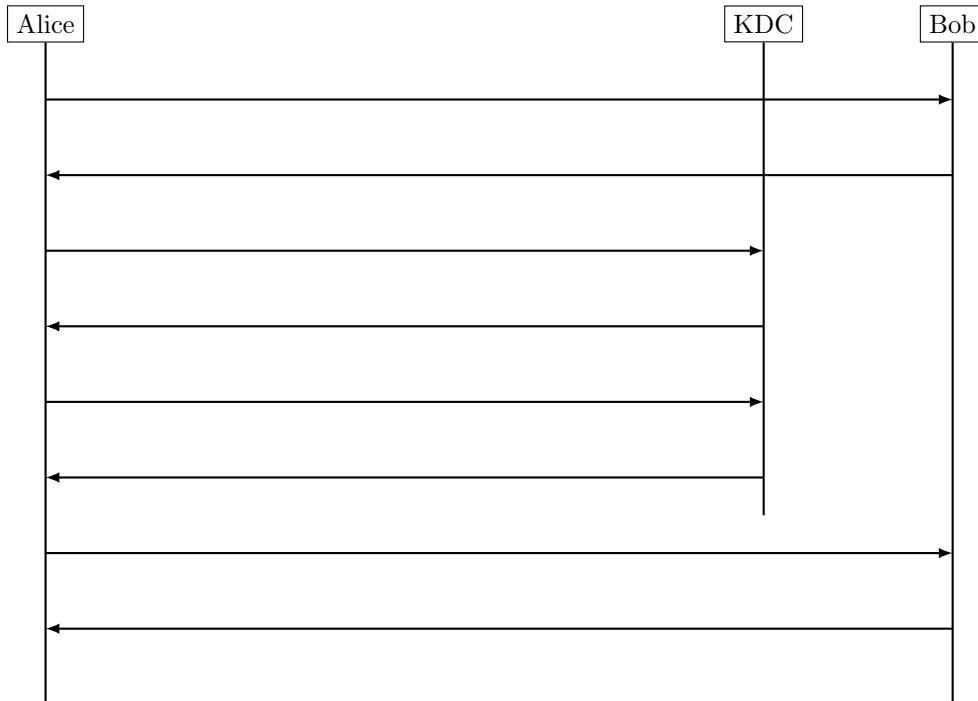


Figure 5.9: Key Distribution Center Protocol, without use of synchronized clocks (exercise 5.9).

to the base *per-session credentials*, allowing the base to authenticate the client and to setup a shared-key with it, to be used to encrypt the communication for confidentiality. We now provide some more details on this process, and later discuss some significant vulnerabilities.

Fig. 5.10 shows a simplification of the flows of the GSM handshake protocol. The handshake begins with the mobile sending its identifier *IMSI* (*International Mobile Subscriber Identity*). The base forwards the IMSI to the home, which retrieves the key of the user owning this IMSI, which we denote  $k_i$ . The home then selects a random value  $r$ , and computes the pair of values  $(K, s)$  as  $(K, s) \leftarrow A38(k_i, r)$ . Here,  $K$  is a symmetric *session key*, and  $s$  stands for *secret result*, a value used to authenticate the mobile. The function<sup>2</sup>  $A38$  should be a Pseudo Random Function (PRF).

The home sends the resulting *GSM authentication triplet*  $(r, K, s)$  to the

<sup>2</sup>The spec actually computes  $K, s$  using two separate functions:  $s \leftarrow A3(k_i, r)$  and  $K \leftarrow A8(k_i, r)$ , and does not specify the required properties from  $A3, A8$ , without specifying their properties. However, for security, the functions should be ‘independently pseudorandom’ - for example, using  $A3 = A8$  is clearly insecure. Indeed, in practice, both are computed by a joint function - usually a specific function called  $COMP128$ . We use the abstract notation  $A38$  for this function;  $COMP128$  is a possible instantiation.

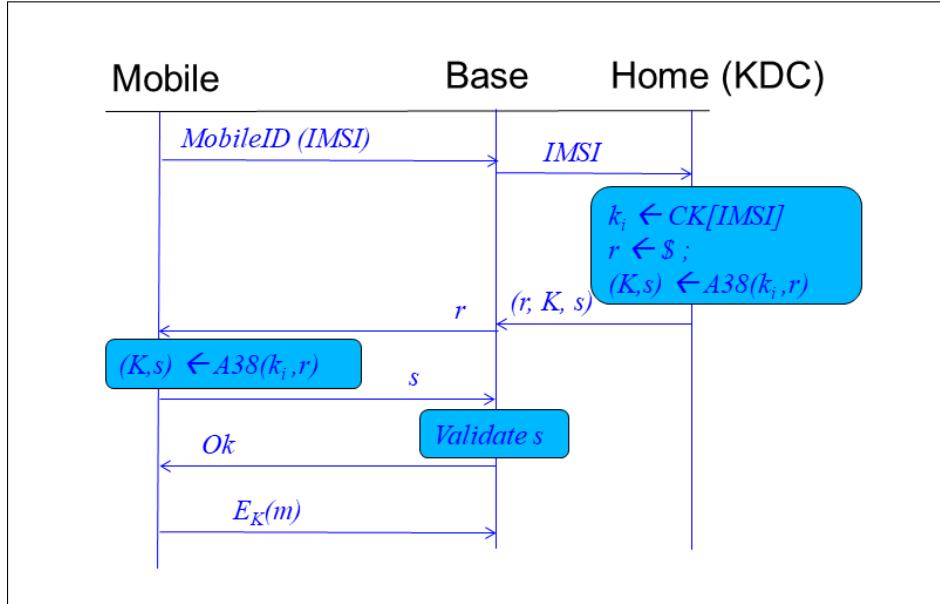


Figure 5.10: The GSM key distribution protocol (simplified). Time proceeds from top to bottom.

base. For efficiency, the home usually sends multiple triplets to the base (not shown). Another way for the base to avoid requesting again an authentication triplet from the home, is for the base to reuse a triplet in multiple connections with the same mobile. Note that the GSM spec uses the terms  $K_C$ ,  $rand$  and  $sres$  for what we denote by  $K$ ,  $r$  and  $s$ , respectively.

Fig. 5.10 also shows an example of a message  $m$  sent from mobile to base, encrypted using the connection's key  $k$ . Of course, in typical real use, the mobile and the base exchange a session consisting of many messages encrypted with  $E_k$ .

In contrary to Kerckhoffs principle (principle Principle 3), all of GSM's cryptographic algorithms, including the stream ciphers as well as the A3 and A8 algorithms, were kept secret, apparently in the (false) hope that this will improve security. We believe that the choice of keeping GSM algorithms secret, contributed to the fact that these algorithms proved to be vulnerable - and furthermore, that serious, and pretty obvious, vulnerabilities exist also in the GSM security protocols. We discuss two of these in the next two subsections.

### 5.5.2 Replay attacks on GSM

We now describe a simple attack against the GSM handshake, as in Figure 5.10. The attack involves a *false base provider*, i.e., an attacker impersonating as a legitimate base. Note that GSM designers seem to have assumed that such

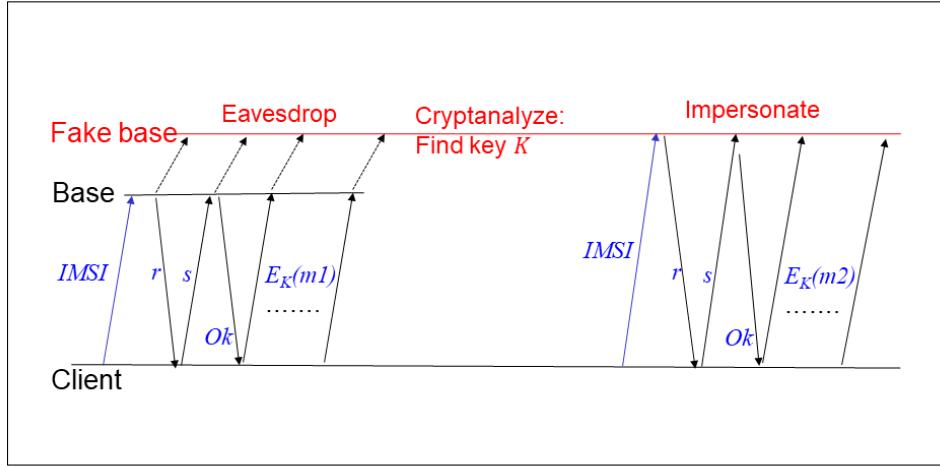


Figure 5.11: Fake-Base Replay attack on GSM. Time proceeds from left to right.

attack is infeasible as building such device is ‘too complex’; this proved short-sighted, and fake base attacks have been, and still are, common. This assumption, and, in particular, the implication that GSM is insecure against a MitM adversary, has not been done clearly, and certainly was not really necessary – security against a MitM attacker requires only minor changes and no significant overhead; i.e., GSM design violated both principle 1, *clear attack model*, and principle 6, *conservative design*.

One typical, simple variant of the attack is the *fake-base replay attack*, shown in Figure 5.11. Note that both Figure 5.10 and Figure 5.11 are schedule diagrams. They are shown differently: in Figure 5.11, time proceeds horizontally from left to right, while in Figure 5.11 time proceeds vertically, from top to bottom.

The attack has three phases:

**Eavesdrop:** in the first phase, the attacker eavesdrops on a legitimate connection between the mobile client and a legitimate base. The handshake between the client and the base is exactly like in Figure 5.11, except that Figure 5.11 does not show the home (and the messages sent to it).

**Cryptanalyze:** in the second phase, the attacker cryptanalyzes the ciphertexts collected during the eavesdrop phase. Assume that the attacker is successful in finding the session key  $K$  shared between client and base; this is reasonable, since multiple effective attacks are known on the GSM ciphers A5-1 and A5-2.

**Impersonate:** finally, once cryptanalysis exposed the session key  $K$ , the attacker sets up a fake base station, and uses that key to communicate

correctly with the client. This allows the attacker to eavesdrop and modify the communication. The attacker may also relay the messages from the client to the ‘real’ base; one reason to do this is so that the client is unlikely to detect that she is actually connected via a rogue base.

### 5.5.3 Cipher-agility and Downgrade Attacks

Practical cryptographic protocols should be designed in a modular manner, and in particular, allow the use of different cryptographic systems, as long as they fulfill the requirements, e.g., encryption, PRF or MAC; this property is usually referred to as *cipher-agility*. The set of cryptographic schemes used in a particular execution of the protocol is referred to as *ciphersuite*, and the process of negotiating the ciphersuite is called *ciphersuite negotiation*. Although ciphersuite negotiation is not complex, it is all too often done insecurely, allowing different  *downgrade attacks*, which allow an attacker to cause the trick the parties into using a particular ciphersuite chosen by the attacker, typically a vulnerable to weaker one. These attacks usually involve a Monster-in-the-Middle (MitM) attacker.

GSM supports cipher-agility, in the sense that the base and the client (mobile) negotiate the stream-cipher to use; GSM defines three stream-ciphers, denoted  $A5/1$ ,  $A5/2$  and  $A5/3$  (also called Kasumi), and also the ‘null’  $A5/0$  which simply means that no encryption is applied. However, the GSM ciphersuite negotiation is not protected at all, allowing trivial downgrade attacks. Worse, GSM has a very unusual property, making downgrade attacks much worse than with most systems/protocols: the same key is used by all encryption schemes, allowing an attacker to find a key used with one (weak) scheme, and use it to decipher communication protected with a different (stronger) cipher. In this subsection, we describe the GSM ciphersuite negotiation process and the related vulnerabilities and attacks.

**GSM ciphersuite negotiation.** The GSM ciphersuite negotiation process is shown in Figure 5.12. In the first message of the handshake, containing the mobile client’s identity (IMSI), the client also sends the list *ciphers* of the stream-ciphers it supports. The base selects, among the stream ciphers, the scheme  $A5 - i$  that it prefers. Usually, the base would select the stream cipher considered most secure among these it supports. In the case of GSM, the  $A5/2$  is well known to be relatively weak; in fact, some of its weakness is by design - the  $A5/2$  cipher was designed to be weak, to allow export of GSM devices, to countries to which it was not allowed, at the time, to export encryption devices. Indeed, very effective attacks were found against  $A5/2$ , and quite effective attacks were also found against  $A5/1$ , making  $A5/3$  the preferred algorithm (with some attacks against it too). We denote encryption with stream-cipher  $A5/i$  and key  $K$  as  $E_K^{A5/i}$ .

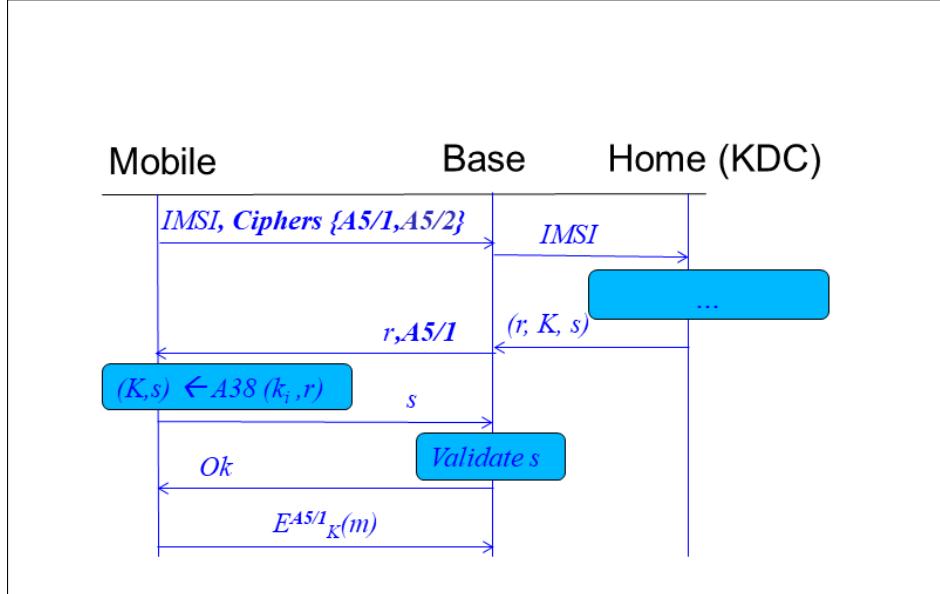


Figure 5.12: Cipher-suite negotiation in GSM.

**A simplified, unrealistic downgrade attack.** We first present a simplified, unrealistic downgrade attack against GSM in Figure 5.13. In this attack, the clients supports A5/1 and A5/2, but the MitM attacker ‘removes’ A5/1 and only offers A5/2 to the base. As a result, the entire session between base and client is only protected using the (extremely vulnerable) A5/2.

**A ‘real’ downgrade attack.** The simplified attack in Figure 5.13 usually fails, i.e., is mostly impractical. The reason is that the GSM specifies that *all* clients should support the A5/1 cipher, and, furthermore, that a base supporting A5/1 should refuse to use A5/2 (or A5/0, which means no encryption at all). However, a minor variant of the attack circumvents this problem, by modifying the message from the base to the client, rather than the message from the client to the base, as shown in Figure 5.14.

In the ‘real’ downgrade attack on GSM, as in Figure 5.13, the base still uses the ‘stronger’ cipher, e.g., A5/1. The attacker modifies the ciphersuite *selection*, sent from the base to the client, causing the *client* to use the vulnerable cipher, e.g., A5/2. Since A5/2 is very weak, it may be broken in few seconds, or even in under a second.

However, note that the GSM base waits significantly less than a second for the first encrypted message, from the time it sends the *Start* message. If the MitM would only begin the cryptanalysis of A5/2 after receiving the *Start* message, the base would time-out and ‘break’ the connection.

However, this issue is easily circumvented, by exploiting the fact that GSM

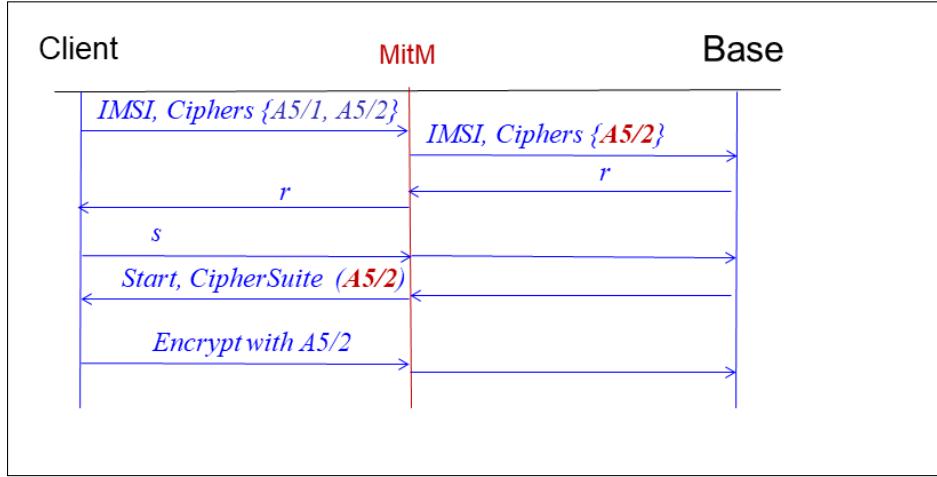


Figure 5.13: A simplified, unrealistic Downgrade Attack on GSM. This version usually fails, since GSM specifies that *all* clients should support the A5-1 cipher, so Base should refuse to use A5-2.

bases allow the client much more time, to compute the authenticator  $s$  - more than 10 seconds typically. The MitM attacker exploits this fact, by delaying the transmission of the authenticator  $s$ , until it finishes cryptanalysis of the message encrypted with A5/2, and found the key  $K$ . Only at that point, the MitM forwards  $s$  to the base; when receiving the *Start* command, the MITM attacker can easily forward the messages from the client after re-encrypting them with A5/1. See Figure 5.14.

Several additional variants of this attack are possible; see, for example, the following exercise.

**Exercise 5.10** (GSM combined replay and downgrade attack). *Present a sequence diagram, like Figure 5.11, showing a ‘combined replay and downgrade attack’, allowing an attacker which eavesdrop on the entire communication between mobile and base on day D, encrypted using a ‘strong’ cipher, say A5/3, to decrypt all of that ciphertext communication, by later impersonating as a base and performing a downgrade attack.*

*Hint:* the attacker will resend the value of  $r$  from the eavesdropped-upon communication (encrypted using ‘strong’ cipher), to cause the mobile to re-use the same key - but with a weak cipher, allowing attacker to expose the key.  $\square$

**Protecting against downgrade attacks.** Downgrade attacks involve modification of information sent by the parties - specifically, the possible and/or chosen ciphers. Hence, the standard method to defend against downgrade attacks is to *authenticate* the exchange, or at least, the ciphersuite-related indicators.

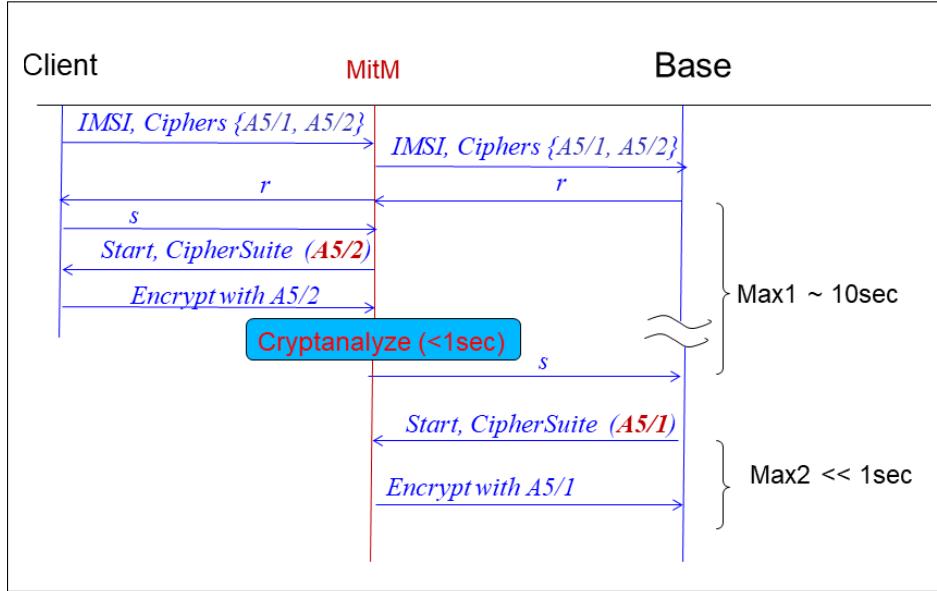


Figure 5.14: GSM Downgrade Attack - the ‘real’ version. This version works, since GSM clients are willing to use the weak A5/2 cipher.

Note that this requires the parties to agree on the authentication mechanism, typically, MAC scheme. It may be desirable to also negotiate the authentication mechanism. In such case, the negotiation should be bounded to reasonable time, and the use of the authentication scheme and key limited to few messages, to foil downgrade attacks on the authentication mechanism. Every authentication mechanism supported, should be secure against this (weak) attack.

It is also necessary to avoid the use of the same key for different encryption schemes, as done in GSM, and exploited, e.g. by the attacks of Figure 5.14 and Exercise 5.10. This is actually easy, and does not require any significant resources - it seems that there was no real justification for this design choice in GSM, except for the fact that this allows the home to send just one key  $K$  before knowing which cipher would be selected by the mobile and base. Indeed, the next exercise shows that a significant improvement in security may be possible even using the existing protocol, by an appropriately-designed new cipher option.

**Exercise 5.11** (Fix GSM key reuse.). *Show a fix to the vulnerability caused by reuse of same key for different ciphers in GSM, as exploited in Exercise 5.10. The fix should not require any change in the messages exchanged by the protocol, and simply appear as a new cipher, say A5/9, using only information available to the client and base. You may assume that the client has an additional secret key,  $k_{9,i}$  and that the home sends to the base an additional key  $K_9$ .*

*Hint:* Note that your fix is not required to prevent the attack of Figure 5.14, although this is also easy - if the Base may assume that *all* clients support A5/9.  $\square$

Note that the GSM response to these vulnerabilities was simply to abolish the use of the insecure A5/2 in mobiles. This prevents downgrade attacks to A5/2, but still allows degrading from A5/3 to A5/1.

## 5.6 Resiliency to key exposure: forward secrecy, recover secrecy and beyond

One of the goals of deriving pseudorandom keys for each session, was to reduce the damage due to exposure of one or some of the session keys. A natural question is, *can we reduce risk from exposure of the entire state of the parties, including, in particular, exposure of the ‘initialization/master’ key  $\kappa$ ?*

One approach to this problem was already mentioned: place the master key  $\kappa$  within a *Hardware Security Module (HSM)*, so that it is assumed *not* to be part of the state exposed to the attacker. However, often, the use of an HSM is not a realistic, viable option. Furthermore, cryptographic keys may be exposed even when using an HSM - by some weakness of the HSM, such as side-channel allowing (immediate or gradual/partial) exposure of keys, or the keys may be exposed via cryptanalysis.

In this section, we discuss a different approach to handle key exposures: design the handshake protocol to ensure security, even in scenarios where the attacker may obtain some of the secret information, e.g., private (master and session) keys. We mostly focus on two notions of resiliency to key exposure: *forward secrecy* and *recover secrecy*. We explain these two notions, and present handshake protocols satisfying them. We also briefly discuss additional, even stronger notions of resiliency to key exposures, mainly, an extension for each of the two notions: *perfect forward secrecy (PFS)* and *perfect recover secrecy (PRS)*.

### 5.6.1 Forward secrecy handshake

We use the term *forward secrecy* to refer to key setup protocols where exposure of *all* keys in some future time (and session), including the master and session keys which are kept at that future time, does not expose the keys or contents of already-completed sessions, even if the ciphertext exchanges were recorded by the attacker. Note that this implies that each period  $i$  must use a separate master key  $MK_i$ , and at the beginning of session  $i$ , we must erase any previous master key (e.g.,  $MK_{i-1}$ ). Definition follows. Note that some authors refer to this notion as *weak forward secrecy*, and we often adopt this, to emphasize the distinction from the stronger notion of *perfect forward secrecy* (which we present later).

**Definition 5.3** (Handshake with (Weak) Forward Secrecy). *A handshake protocol  $\pi$  ensures forward secrecy if exposure of the entire state of an entity at*

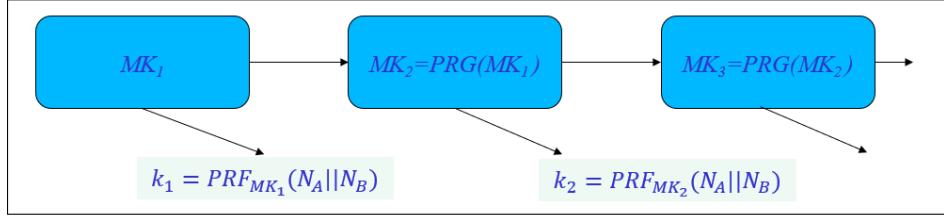


Figure 5.15: (Weak-)Forward-secrecy Handshake, implemented as in Eq. 5.5, i.e.,  $MK_i = PRG(MK_{i-1})$ . Exposure of all the information of an entity at time  $t$ , does not compromise the confidentiality of past sessions. Information sent by the entity or sent to the entity, in any session which *ended before* time  $t$ , remains secure even after the keys kept at time  $t$  are exposed. Session keys may be derived from the current master key  $MK_i$  as in Eq. 5.6, similarly to the derivation of session keys from the (fixed) master key in Eq. 5.2.

*time  $t$ , including the current master (and session) keys, does not compromise the confidentiality of information sent by the entity or sent to the entity, in any session which ended before time  $t$ .*

We next present (*weak*) *forward secrecy key-setup handshake*, a forward-secrecy variant of the key-setup 2PP extension, which we discussed and presented earlier, in subsection 5.4.2. The difference is that instead of using a single master key  $\kappa$ , received during initialization, the forward-secrecy handshake uses a *sequence of master keys*  $MK_0, MK_1, \dots$ ; for simplicity, assume that each master key  $MK_i$  is used only for the  $i^{th}$  handshake, with  $MK_0$  received during initialization.

The key to achieving the (*weak*) forward secrecy property, is to allow easy derivation of the future master keys  $MK_{i+1}, \dots$  from the current master key  $MK_i$ , but prevent the reverse, i.e., maintain the previous master keys  $MK_{i-1}, MK_{i-2}, \dots, MK_0$  pseudorandom, even for an adversary who knows  $MK_i, MK_{i+1}, \dots$ . A simple way to achieve this is by using PRG, namely:

$$MK_i = PRG(MK_{i-1}) \quad (5.5)$$

The session key  $k_i$  for the  $i^{th}$  session can be derived using the corresponding minor change to Eq. 5.2, namely:

$$k_i = PRF_{MK_i}(N_A || N_B) \quad (5.6)$$

The resulting protocol is illustrated in Fig. 5.16. The use of  $N_A$  and  $N_B$  in Eq. (5.6) is not really necessary, since each master key is used only for a single handshake.

Note that it is easy to implement a PRG using a PRF (Ex. 2.48). Hence, we can use a PRF to derive both  $k_i$  and  $\kappa_i$ .

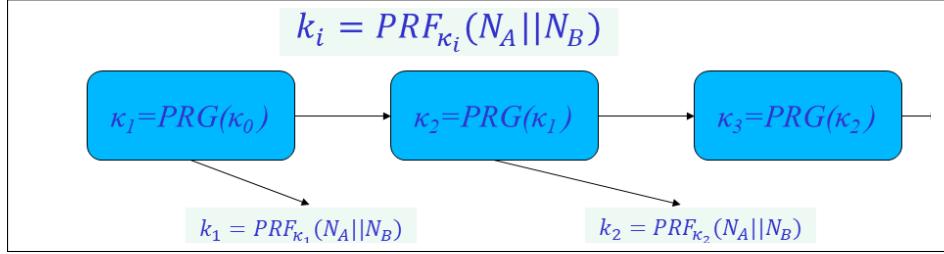
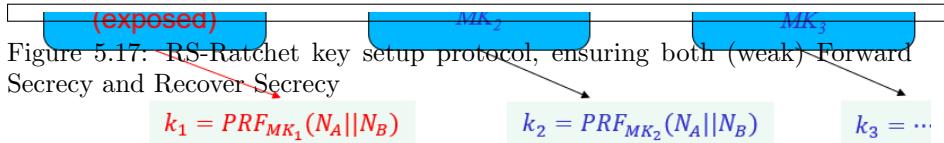


Figure 5.16: Forward-secrecy key setup: derivation of per-session master keys and session keys



### 5.6.2 Recover-Security Handshake Protocol

We use the term *recover security*, or weak recover security, to refer to key setup protocols where a single session without eavesdropping or other attacks, suffices to recover security from previous key exposures. Definition follows.

**Definition 5.4** ((Weak) Recover security handshake). *A handshake protocol  $\pi$  ensures (weak) recover secrecy if secrecy is ensured for messages exchanged during session  $i$ , provided that there exists some previous session  $i' < i$  such that:*

1. *There is no exposure of keys from session  $i'$  to session  $i$ ; all keys and other state prior to session  $i'$  is exposed (known to attacker).*
2. *During session  $i'$  itself, all messages are delivered correctly, without eavesdropping, injection or modification. We allow MitM attacks on communication in other sessions (including session  $i$ ).*

The protocols of Figure 5.16 and Figure 5.18 ensure *forward secrecy* - but not *recover secrecy*. This is since the attacker can use one exposed master key, say  $MK_{i'-1}$ , to derive all the following master keys, including  $MK_i$ , using Eq. 5.5; in particular,  $MK_i = PRG(MK_{i'-1})$ .

However, a simple extension suffices to ensure recover secrecy, as well as forward secrecy. The extension is simply to use the random values exchanged in each session, i.e.,  $N_A, N_B$ , in the derivation of the next master key, i.e.:

$$MK_i = PRG(MK_{i-1}) \oplus N_A \oplus N_B \quad (5.7)$$

Figure 5.17 illustrates the resulting key-setup handshake protocol, the *RS-Ratchet* protocol.

By computing the new master key as exclusive-OR of these three values, it is secret as long as at least *one* of these three values is secret - this could be viewed as one-time-pad encryption (OTP, see § 2.4). Since the recover secrecy requirement assumes at least one session where the attacker does not eavesdrop or otherwise interfere with the communication, then both  $N_A$  and  $N_B$  are secret, hence the new master key  $\kappa_i$  is secret. Indeed, XOR with one of the two keys is sufficient; by XOR-ing with both of them, we ensure secrecy of the master key even if the attacker is able to capture one of the two flows, i.e., even stronger security.

Note that the RS-Ratchet protocol requires the parties to have a source of randomness, which is secure - even after the party was broken-into (and keys exposed). In reality, many systems only rely on pseudo-random generators (PRGs), whose future values are computable using a past value. In such case, it becomes critical to use also the input from the peer ( $N_A$  or  $N_B$ ), and these values should be also used to re-initialize the PRG, so that new nonces ( $N_A, N_B$ ) are pseudorandom and not predictable.

### 5.6.3 Stronger notions of resiliency to key exposure

Forward secrecy and recover secrecy significantly improve the resiliency against key exposure. There are additional and even stronger notions of resiliency to key exposure, which are provided by more advanced handshake protocols; we only cover few of these in this course- specifically, the ones in Table 5.2.

Protocols for many of the more advanced notions of resiliency, use public key cryptology, and in particular, key-exchange protocols such as the Diffie-Hellmen (DH) protocol. Indeed, it seems plausible that public-key cryptography is necessary for many of these notions. This includes the important notions of *Perfect Forward Secrecy (PFS)* and *Perfect Recover Secrecy (PRS)*, which, as the names imply, are stronger variants of forward and recover secrecy, respectively. We now briefly discuss PFS and PRS, to understand their advantages and why they require more than the protocols we have seen in this chapter; we discuss these notions further, with implementations, in § 6.4.

**Perfect Forward Secrecy (PFS).** Like ‘regular’ forward-secrecy, PFS also requires resiliency to exposures of state, including keys, occurring in the future. However, unlike ‘regular’ forward-secrecy, PFS also requires resilience to exposure of the *previous* state, again including keys, as long as the attacker can only eavesdrop on the session.

**Definition 5.5** (Perfect Forward Secrecy (PFS)). *A handshake protocol  $\pi$  ensures perfect forward secrecy (PFS) if exposure of all of the state of all sessions except session  $i$ , does not expose the keys of session  $i$ , provided that during session  $i$ , attacker can only eavesdrop on messages (and the keys of session  $i$  are not otherwise disclosed).*

We discuss some PFS handshake protocols in the next chapter, which deals with asymmetric cryptography (also called public-key cryptography, PKC). Indeed, all known PFS protocols are based on PKC.

**Exercise 5.12** (Forward Secrecy vs. Perfect Forward Secrecy (PFS)). *Present a sequence diagram, showing that the forward-secrecy key-setup handshake protocol presented in subsection 5.6.1, does not ensure Perfect Forward Secrecy (PFS).*

**Perfect Recover Secrecy (PRS).** We use the term *perfect recover secrecy* to refer to key setup protocols where a single session without MitM attacks, suffices to recover secrecy from previous key exposures. Definition follows.

**Definition 5.6** (Perfect Recover Secrecy (PRS) handshake). *A handshake protocol  $\pi$  ensures perfect recover secrecy (PRS), if secrecy is ensured for messages exchanged during session  $i$ , provided that there exists some previous session  $i' < i$  such that:*

1. *There is no exposure of keys from session  $i'$  to session  $i$ ; all keys and other state prior to session  $i'$  is exposed (known to attacker).*
2. *During session  $i'$  itself, all messages are delivered correctly, without MitM attacks (injection or modification); however, eavesdropping is allowed. Allow MitM attacks on communication in other session (including session  $i$ ).*

Note the similarity to PFS, in allowing only eavesdropping during the ‘target’ session ( $i$  for PFS,  $i'$  for PRS). Also similarly to PFS, we discuss some PRS handshake protocols in the next chapter, which deals with asymmetric cryptography. Indeed, all known PRS protocols are based on asymmetric cryptography.

**Additional notions of resiliency.** The research in cryptographic protocols includes additional notions of resiliency to key and state exposures, which we do not cover in this course. These includes *threshold security* [37], which ensures that the entire system remains secure even if (up to some threshold) of its modules are exposed or corrupted, *proactive security* [30], which deals with recovery of security of some modules after exposures, and *leakage-resiliency* [48], which ensures resiliency to gradual leakage of parts of the storage. These - and other - notions, are beyond our scope.

#### 5.6.4 Per-goal Keys Separation.

Our discussion focused on deriving a single session key. However, for security, we should use a separate key for encryption, from the key we use for authentication. Furthermore, the attacker may be more likely to attack traffic in one direction, say Alice to Bob, than traffic in the other direction, e.g., since the

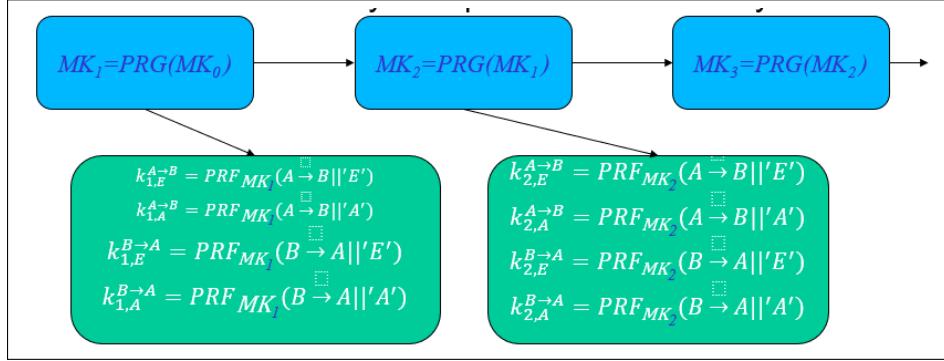


Figure 5.18: Forward-secrecy key setup: derivation of per-session master keys and per-goal session keys

attacker can control the traffic in one direction (chosen plaintext attack) or to predict it (known plaintext attack). See the *key-separation principle* (principle 5) and the discussion in subsection 5.4.3.

We can implement such separation using a Pseudo-Random Function (PRF). Specifically, derive  $(k_{i,E}^{A \rightarrow B}, k_{i,A}^{A \rightarrow B}, k_{i,E}^{B \rightarrow A}, k_{i,A}^{B \rightarrow A})$ , the four session keys for session  $i$ , as follows (see also Figure 5.18):

$$\begin{aligned} k_{i,E}^{A \rightarrow B} &= PRF_{\kappa_i}(A \rightarrow B || 'E') \\ k_{i,A}^{A \rightarrow B} &= PRF_{\kappa_i}(A \rightarrow B || 'A') \\ k_{i,E}^{B \rightarrow A} &= PRF_{\kappa_i}(B \rightarrow A || 'E') \\ k_{i,A}^{B \rightarrow A} &= PRF_{\kappa_i}(B \rightarrow A || 'A') \end{aligned}$$

Since  $\kappa_i$  is secret, and  $PRF$  is a pseudo-random function, then  $PRF_{\kappa_i}$  is indistinguishable from a random function (over same domain and range). Since the inputs for deriving each of the four keys are different, the outputs of a random function would be all uniformly random (and independent of each other). Since  $PRF_{\kappa_i}$  is indistinguishable from a random function, it follows that the four keys are independently pseudo-random.

### 5.6.5 Resiliency to exposures: summary

In this section, we discussed several notions of resiliency of handshake protocols to exposure of secret information, including cryptographic keys. We mostly focused on two notions of resiliency to key exposure: *forward secrecy* and *recover secrecy*. We explained these two notions, and presented handshake protocols satisfying them. We also briefly discussed additional, even stronger notions of resiliency to key exposures, mainly, an extension for each of the two notions: *perfect forward secrecy* (*PFS*) and *perfect recover secrecy* (*PRS*). Designs for PFS and PRS involve public key cryptology; we cover these designs in chap-

Notion	Session $i$ is secure, even when attacker...	Crypto
Secure key-setup	... is given <u>session key</u> $k_j$ ( $j \neq i$ ). [ Not resilient to exposure of master key! ]	Shared key
(Weak) Forward secrecy	... is given <u>all keys used after</u> session $i$	Shared key
Perfect Forward Secrecy (PFS)	... is given, <u>after</u> session $i$ , <u>all keys of all sessions except</u> session $i$	Public key
(Weak) Recover Secrecy	... is given <u>all keys of all sessions, except</u> keys of sessions $i_R$ to $i > i_R$ , and <u>assuming no eavesdropping during session</u> $i_R$	Shared key
Perfect Recover Secrecy (PRS)	... is given <u>all keys of all sessions, except</u> keys of sessions $i_R$ to $i > i_R$ , and <u>assuming no MitM during session</u> $i_R$	Public key

Table 5.2: Notions of resiliency to key exposures of key-setup handshake protocols. See implementations of forward and recover secrecy in subsection 5.6.1, subsection 5.6.2 respectively, and for the corresponding ‘perfect’ notions (PFS and PRS) in subsection 6.4.1 and subsection 6.4.2, respectively.

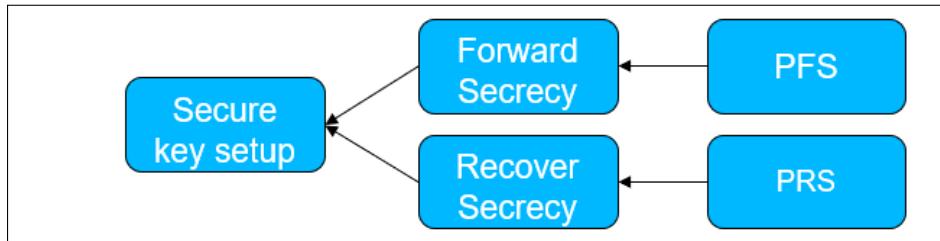


Figure 5.19: Relations between notions of resiliency to key exposures. An arrow from notion A to notion B implies that notion A implies notion B. For example, a protocol that ensures Perfect Forward Secrecy (PFS), also ensures Forward Secrecy.

ter 6. *Forward secrecy* and *recover secrecy*, the two notions on which we focused in this section, are achievable by shared-key key-setup protocols.

We compare the four notions, along with the ‘regular’ secure key-setup handshake, in Table 5.2, and present the relationships between them in Fig. 5.19.

**Terminology for resiliency properties.** Many different notions of resiliency are considered in the literature; we focused only on few of them, which we consider most basic and important. Even for these notions, there are some different interpretations and definitions, and sometimes inconsistent usage of terms. In

particular, some experts use the term *forward secrecy* as synonym to *perfect forward secrecy (PFS)*. Furthermore, while *break-in recovery* is often mentioned as a goal for the ratchet protocols, we introduced the terms *recover secrecy* in subsection 5.6.2 and *perfect recover secrecy (PRS)* in subsection 6.4.2.

## 5.7 Shared-Key Session Protocols: Additional Exercises

**Exercise 5.13** (Vulnerability of Compress-then-Encrypt). *One of the important goals for encryption of web communication, is to hide the value of ‘cookies’, which are (often short) strings sent automatically by the browser to a website, and often used to authenticate the user. Attackers often cause the browser to send requests to a specific website, to try to find out the value of the cookie attached by the browser. Assume, for simplicity, that the attacker controls the entire contents of the request, before and after the cookie, except for the cookie itself (which the attacker tries to find out). Furthermore, assume that the length of the cookie is known. Further assume that the browser uses compression before encrypting requests sent to the server.*

*Finally, assume a very simple compression scheme, that only replaces: (1) sequences of three or more identical characters with two characters, and (2) repeating strings (of two or more characters, e.g.,  $xyxy$ ), with a single string plus one character (e.g.,  $\perp xy$ ; assume that the  $\perp$  does not appear in the request except to signal compression).*

1. Present an efficient attack which exposes the first character of the cookie.
2. Extend your attack to expose the entire cookie.

**Exercise 5.14.** *Some applications require only one party (e.g., a door) to authenticate the other party (e.g., Alice); this allows a somewhat simpler protocol. We describe in the two items below two proposed protocols for this task (one in each item), both using a key  $k$  shared between the door and Alice, and a secure symmetric-key encryption scheme  $(E, D)$ . Analyze the security of the two protocols.*

1. *The door selects a random string (nonce)  $n$  and sends  $E_k(n)$  to Alice; Alice decrypts it and sends back  $n$ .*
2. *The door selects and sends  $n$ ; Alice computes and sends back  $E_k(n)$ .*

*Repeat the question, when  $E$  is a block cipher rather than an encryption scheme.*

**Exercise 5.15.** *Consider the following mutual-authentication protocol, using shared key  $k$  and a (secure) block cipher  $(E, D)$ :*

1. *Alice sends  $N_A$  to Bob.*
2. *Bob replies with  $N_B, E_k(N_A)$ .*
3. *Alice completes the handshake by sending  $E_k(N_B \oplus E_k(N_A))$ .*

*Show an attack against this protocol, and identify design principles violated by the protocol (allowing such attacks).*

**Exercise 5.16** (GSM). In this exercise we study some of the weaknesses of the GSM handshake protocol, as described in subsection 5.5.1. In this exercise we ignore the existence of multiple types of encryption and their choice ('ciphersuite').

1. In this exercise, and in usual, we ignore the fact that the functions  $A_8$ ,  $A_3$  and the ciphers  $E_i$  were kept secret; explain why.
2. Present functions  $A_3, A_8$  such that the protocol is insecure when using them, against an eavesdropping-only adversary.
3. Present functions  $A_3, A_8$  that ensure security against MitM adversary, assuming  $E$  is a secure encryption. Prove (or at least argue) for security. (Here and later, you may assume a given secure PRF function,  $f$ .)
4. To refer to the triplet of a specific connection, say the  $j^{\text{th}}$  connection, we use the notation:  $(r(j), sres(j), k(j))$ . Assume that during connection  $j'$  attacker received key  $k(\hat{j})$  of previous connection  $\hat{j} < j'$ . Show how a MitM attacker can use this to expose, not only messages sent during connection  $\hat{j}$ , but also messages sent in future connections (after  $j'$ ) of this mobile.
5. Present a possible fix to the protocol, as simple and efficient as possible, to prevent exposure of messages sent in future connections (after  $j'$ ). The fix should only involve changes to the mobile and the base, not to the home.

**Exercise 5.17.** Fig. 5.20 illustrates a simplification of the SSL/TLS session-security protocol; this simplification uses a fixed master key  $k$  which is shared in advance between the two participants, Client and Server. This simplified version supports transmission of only two messages, a 'request'  $M_C$  sent by the client to the server, and a 'response'  $M_S$  sent from the server. The two messages are protected using a session key  $k'$ , which the server selects randomly at the beginning of each session, and sends to the client, protected using the fixed shared master key  $k$ .

The protocol should protect the confidentiality and integrity (authenticity) of the messages ( $M_C, M_S$ ), as well as 'replay' of messages, e.g., client sends  $M_C$  in one session and server receives  $M_C$  on two sessions.

1. The field `cipher_suite` contains a list of encryption schemes ('ciphers') supported by the client, and the field `chosen_cipher` contains the cipher in this list chosen by the server; this cipher is used in the two subsequent messages (a fixed cipher is used for the first two messages). For simplicity consider only two ciphers, say  $E_1$  and  $E_2$ , and suppose that both client and server support both, but that they prefer  $E_2$  since  $E_1$  is known to be vulnerable. Show how a MitM attacker can cause the parties to use  $E_1$  anyway, allowing it to decipher the messages  $M_C, M_S$ .

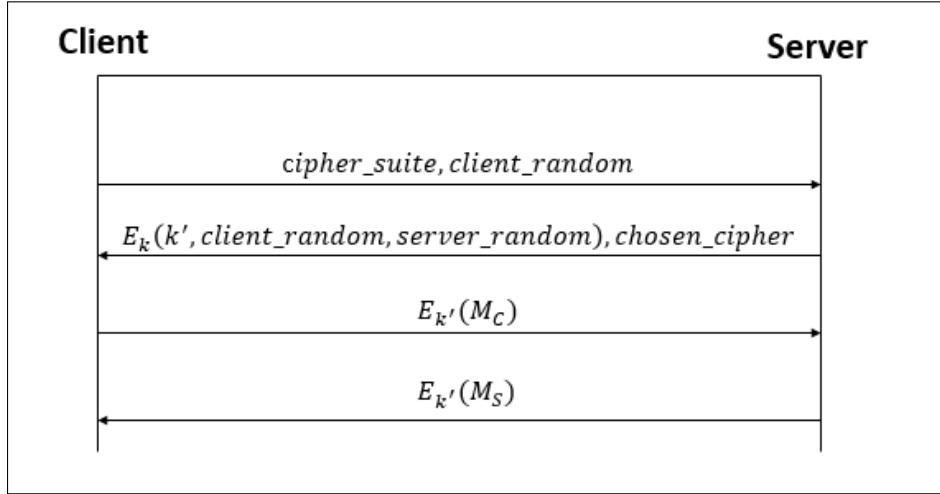


Figure 5.20: Simplified SSL

2. Suggest a minor modification to the protocol to prevent such ‘downgrade attacks’.
3. Ignore now the risk of downgrade attacks, e.g., assume all ciphers supported are secure. Assume that  $M_C$  is a request to transfer funds from the clients’ account to a target account, in the following format:

Date (3 bytes)	Operation type (1 byte)	Comment (20 bytes)	Amount (8 bytes)	Target account (8 bytes)
-------------------	----------------------------	-----------------------	---------------------	-----------------------------

Assume that  $E$  is CBC mode encryption using an 8-bytes block cipher. The solution should not rely on replay of the messages (which will not work since only one message is sent in each direction on each usage).

Mal is a (malicious) client of the bank, and eavesdrop on a session where Alice is sending a request to transfer 10\$ to him (Mal). Show how Mal can abuse his Man-in-the-Middle abilities to cause transfer of larger amount. Explain a simple fix to the protocol to prevent this attack.

**Exercise 5.18.** Consider the following protocol for server-assisted group-shared-key setup. Assume that every client, say  $i$ , shares a key  $k_i$  with the server, and let  $G$  be a set of users s.t.  $i \in G$ . Client  $i$  may ask the server to provide it with a key  $k_G$ , shared with all the users in  $G$ , by sending the server a request specifying the set  $G$ , authenticated using  $k_i$ . The server replies by sending  $x_G = k_G + \prod_{j \in G} \text{PRF}_{k_j}(t)$  to the clients in  $G$ , where  $t$  is the time<sup>3</sup>. User  $i$  then computes  $\text{PRF}_{k_i}(t)$  and then finds  $k_G = x_G \bmod \text{PRF}_{k_i}(t)$ . Every other user in the set  $G$  can similarly receive  $x_G$  and compute from it  $k_G$  similarly.

<sup>3</sup>The  $\Pi$  notation denotes multiplication of the elements much like  $\Sigma$  denotes addition, e.g.,  $\prod_{j \in \{1,2,3\}} a_j = a_1 \cdot a_2 \cdot a_3$ .

*Present an attack allowing a malicious client, e.g., Mal, to learn the key  $k_G$  for a group it does not belong to, i.e.,  $Mal \notin G$ ; e.g., assume  $G = \{a, b, c\}$ . Mal may eavesdrop to all messages, and request and receive  $k_{G'}$  for any group  $G'$  s.t.  $Mal \in G'$ .*

**Exercise 5.19.** *In the GSM protocol, the home sends to the base one or more authentication triplets  $(r, K, s)$ . The base and the mobile are to use each triplet only for a single handshake; this is somewhat wasteful, as often the mobile has multiple connections (and handshakes) while visiting the same base.*

1. *Suppose a base decides to re-use the same triplet  $(r, K, s)$  in multiple handshakes, for efficiency (less requests to home). Present message sequence diagram showing that this may allow an attacker to impersonate as a client. Namely, that client authentication fails.*
2. *Suggest an improvement to the messages sent between mobile and base, that will allow the base to reuse the  $(r, K, s)$  triplet received from base, for multiple secure handshakes with the mobile. Your improvement should consist of a single additional challenge  $r_B$  which the base selects randomly and sends to the mobile, together with the challenge  $r$  received in the triplet from the home; and a single response  $s_B$  which the mobile returns to the server, instead of sending the response  $s$  as in the original protocol. Show the computation of  $s_B$  by mobile and base:  $s_B = \dots$ . Your solution may use an arbitrary pseudo-random function PRF.*
3. *GSM sends frames (messages) of 114 bits each, by bit-wise XORing the  $n^{th}$  plaintext frame with 114 bits output from  $A5/i_K(n)$ . Here,  $A5/i$ , for  $i = 1, 2, \dots$ , is a cryptographic function,  $n$  is the frame number, and  $K$  was a key received from the home.  $A5/1$  and  $A5/2$  are described in the specifications - and both are known to be vulnerable; other functions can be agreed between mobile and base. Both  $A5/1$  and  $A5/2$  are insecure; for this question, assume the use of a secure cipher, say  $A5/5$ . Suppose, again, that a base decides to re-use the same triplet  $(r, K, s)$  in multiple handshakes. Present a message sequence diagram in which an A mobile has two connections to the base, sending message  $m_1$  in the first connection and message  $m_2$  in the second connection. Assume that the base re-use the same triplet  $(r, K, s)$  in both connections, and that the attacker knows the contents of  $m_1$ . Show how the attacker can find  $m_2$ . Note: the improvement suggested in the previous item  $(r_B, s_B)$  does not have significant impact on this item - you can solve with it or without it.*
4. *To prevent the threat presented in the previous item, the mobile and base can use a different key  $K' = \dots$  (instead of using  $K$ ).*
5. *Design a base-only forward secrecy improvement to  $A5/5$ . Namely, even if attacker is given access to all of the base memory after the  $j^{th}$  handshake using the same  $r$ , the attacker would still not be able to decipher information exchanged in past connections. Your design may send the*

value of  $j$  together with  $r$  from base to mobile, and may change the stored value of  $s$  at the end of every handshake; let  $s_j$  denote the value of  $s$  at the  $j^{\text{th}}$  handshake, where the initial value is  $s$  received from the home (i.e.,  $s_1 = s$ ). Your solution consists of defining the value of  $s_j$  given  $s_{j-1}$ , namely:  $s_j = \underline{\hspace{2cm}}$ .

**Exercise 5.20.** The GSM protocol is very vulnerable to downgrade attacks; let's design a method to prevent downgrade attacks on new ciphers, when both client (mobile) and base support (the same) new cipher.

1. An attacker impersonating as a base, may try to behave as if it is a base supporting only an older cipher (without protection against downgrade attack).

**Exercise 5.21.** Consider the following key establishment protocol between any two users with an assistance of a server  $S$ , where each user  $U$  shares a secret key  $K_{US}$  with a central server  $S$ .

$A \rightarrow B : (A, N_A)$

$B \rightarrow S : (A, N_A, B, N_B, \mathcal{E}_{K_{BS}}(A||N_A))$

$S \rightarrow A : (A, N_A, B, N_B, \mathcal{E}_{K_{AS}}(N_A||sk), \mathcal{E}_{K_{BS}}(A||sk), N_B)$

$A \rightarrow B : (A, N_A, B, N_B, \mathcal{E}_{K_{BS}}(A||sk))$

Assume that  $\mathcal{E}$  is an authenticated encryption. Show an attack which allows an attacker to impersonate one of the parties to the other, while exposing the secret key  $sk$ .

**Exercise 5.22** (Hashing vs. Forward Secrecy). We discussed in §5.6.1 the use of PRG or PRF to derive future keys, ensuring Forward Secrecy. Could a cryptographic hash function be securely used for the same purpose, as in  $\kappa_i = h(\kappa_{i-1})$ ? Evaluate if such design is guaranteed to be secure, when  $h$  is a (1) CRHF, (2) OWF, (3) bitwise-randomness extracting.

## Chapter 6

# Public Key Cryptology

In chapters 2 and 3, we studied *symmetric (shared key)* cryptographic schemes - for encryption and for message-authentication, respectively. Indeed, until the 1970s, *all* of cryptology was based on the use of symmetric keys - and almost all of it was done only within defense and intelligence organization, with very few publications, academic research and commercial products.

This changed quite dramatically in the 1970s, with the beginning of what we now call *modern cryptology*, as we described in subsection 1.2.1. In particular, in their seminal paper [40] from 1976, Diffie and Hellman observed that there could be significant advantages in cryptographic schemes that used different keys - a public one (e.g. for encryption) and a private one (for decryption). This revolutionary idea has developed into the huge area of *public key cryptology*, also referred to as *asymmetric cryptology*. Diffie and Hellman also identified three types of public-key schemes: *public-key encryption*, *digital signatures* and *key exchange*, and presented the first public-key construction: the *DH key exchange* protocol.

In this chapter, we introduce public key cryptology, beginning with key-exchange protocols. We discuss public-key encryption and signature schemes mostly in the later sections, except for brief introduction in the next section.

### 6.1 Introduction to PKC

The basic observation underlying asymmetric cryptology is quite simple, at least in hindsight: *security requirements are asymmetric*. For example, an encryption scheme should prevent a MitM attacker from *decrypting ciphertext*, when it does not have the intended-receiver's key; but we may not care if the attacker may *encrypt* messages. Notice, indeed, that the security definition we presented for encryption, Def. 2.8, does not require preventing the attacker from *encrypting* plaintexts, only from *decrypting* ciphertexts.

Specifically, Diffie and Hellman defined three basic goals for public key cryptosystems; and these still remain the three most important types of public

key cryptology: *public key cryptosystem (PKC)*, *digital signature schemes*, and *key exchange* protocols.

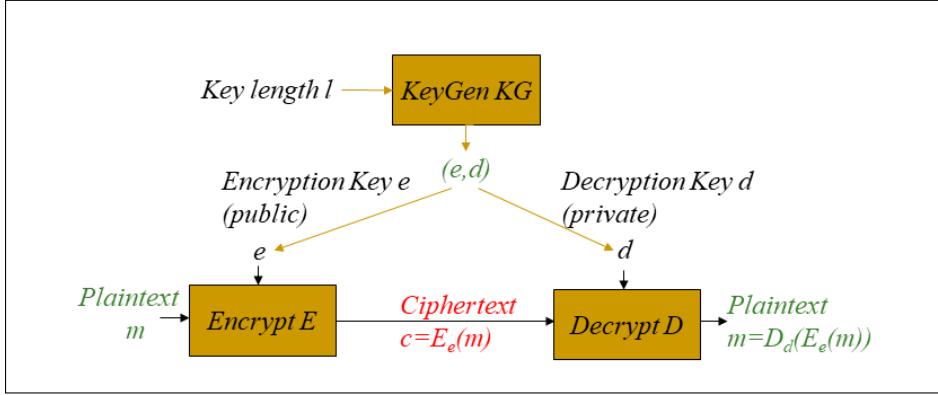


Figure 6.1: Public-Key Cryptosystem ( $KG, E, D$ ). The Key-Generation algorithm  $KG$  outputs a (public,private) key-pair  $(e, d)$  of given length  $l$ . The encryption algorithm  $E$  uses the public encryption key  $e$  to compute the ciphertext  $c = E_e(m)$ , given plaintext message  $m$ . The decryption algorithm  $D$  uses the private decryption key  $d$  to compute the plaintext; correctness holds if for every pair of matching keys  $(e, d) = KG(l)$  and every message  $m$ , holds  $m = D_d(E_e(m))$ .

### 6.1.1 Public key cryptosystems

Public key cryptosystems (PKC) are encryption schemes consisting of three algorithms,  $(KG, E, D)$ , and using a pair of keys: a *public key*  $e$  for encryption, and a *private key*  $d$  for decryption. Both keys are generated by the *key generation* algorithm  $KG$ . The encryption key is not secret; namely, we assume that it is known to the attacker. The correctness requirement of PKCs is similar to the one for shared-key cryptosystems, namely:

$$(\forall m \in M, (e, d) \leftarrow KG(l)) \quad D_d(E_e(m)) = m \quad (6.1)$$

We illustrate public key cryptosystems in Figure 6.1.

Note that we only define a stateless version of PKC. The reason for that is that typically, the public key  $e$  may be shared and used by multiple parties; we obviously cannot assume that these parties share a state variable among them, and therefore, they also cannot be synchronized with the ‘recipient’ decrypting the ciphertexts.

We further discuss PKC in sections 6.5 to 6.7.

### 6.1.2 Digital signature schemes

Digital *signature schemes* consist of three algorithms,  $(KG, S, V)$ , for Key Generation, Signing and Verifying, respectively. Key Generation ( $KG$ ) is a randomized algorithm, and it outputs a pair of correlated keys: a *private signing key*  $s$  for ‘signing’ a message, and a *public validation key*  $v$  for validating a given signature for a given message. The validation key  $v$  is not secret: it should only allow validation of authenticity, and should not facilitate signing.

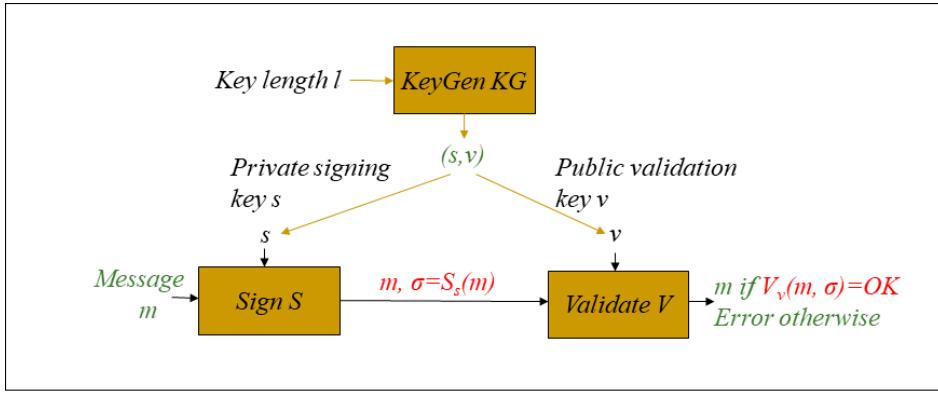


Figure 6.2: Digital signature scheme  $(KG, S, V)$ . The Key-Generation algorithm  $KG$  outputs a (private,public) keypair  $(s, v)$  of given length  $l$ . The signing algorithm  $S$  uses the private signature key  $s$  to compute the signature  $\sigma = S_s(m)$ , given plaintext message  $m$ . The validation algorithm  $V$  uses the public validation key  $v$  to validate the plaintext; if validation returns FALSE, then  $\sigma$  is not a valid signature of  $m$ .

Digital signatures serve a similar function to Message Authentication Code (MAC) schemes, except that rather than using a single function  $MAC$  and a single key  $k$  for authenticating and for validating, they use a distinct key  $s$  and function  $S$  for signing (authenticating), and a distinct key  $v$  and function  $V$  for validating authenticity. The correctness requirement of signature schemes is similar to the one for MAC schemes, namely:

$$(\forall m \in M, (s, v) \leftarrow KG(l)) \quad V_v(m, S_s(m)) = \text{TRUE} \quad (6.2)$$

We illustrate digital signature schemes in Figure 6.2, and discuss them in § 6.8.

### 6.1.3 Key exchange protocols

Key exchange protocols are defined by a pair of functions,  $(G, KD)$ . A run of the protocol involves two parties, typically referred to as Alice (or A) and Bob (or B). We call  $G$  the *generating function*; it receives as input a length  $l$ , and output a pair of values, one public and one private. We call  $KD$  the the *key derivation function*; it receives as input a public value (from one party) and a

private value (from the other party), and produces a (shared) key. Each party, e.g., Alice, uses  $G$  to generate a pair  $(a, P_A)$  of private value  $a$ , kept by Alice, and a public value  $P_A$ , which Alice sends to its peer, Bob. Each party also receives public information from the peer: Alice receives  $p_B$  from Bob, and Bob receives  $P_A$  from Alice. Each party now used  $KD$  to derive a key; the protocol should ensure *correctness*, i.e., that both parties will derive the same key, as formalized as follows.

$$(\forall (a, P_A) \leftarrow G(l), (b, P_B) \leftarrow G(l)) \quad KD(a, P_B) = KD(b, P_A) \quad (6.3)$$

The security requirement is that an eavesdropper cannot learn anything about the shared key, i.e., cannot distinguish between it and a random string.

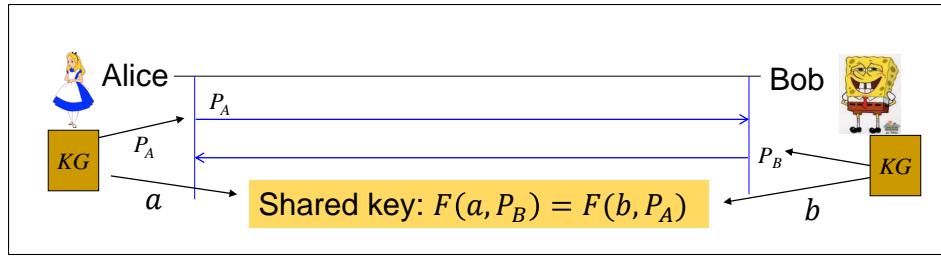


Figure 6.3: Key Exchange protocol. Each party, Alice and Bob, runs the Key-Generation algorithm  $KG$ , which outputs a (private,public) keypair:  $a, P_A$  for Alice, and  $b, P_B$  for Bob. The parties exchange their public keys ( $P_A$  and  $P_B$ ). Then, each party applies a function  $F$  to its own private key ( $a$  for Alice and  $b$  for Bob), and to the public key received from the peer ( $P_A$  from Alice and  $P_B$  from Bob). A key-exchange protocol should ensure that  $F(a, P_B) = F(b, P_A)$ , and then the parties can use this value as a secret, and derive from it a shared secret key  $k$ .

#### 6.1.4 Advantages of Public Key Cryptology (PKC)

Public key cryptology is not just a cool concept; it is very useful, allowing solutions to problems which symmetric cryptology fails to solve, and making it easier to solve other problems.

We first identify three important challenges which require the use of asymmetric cryptology:

**Signatures provide evidences.** Only the owner of the private key can digitally sign a message, but everyone can validate this signature. This allows a recipient of a signed message to know that once he validated the signature, he has the ability to convince other parties that the message was signed by the sender. This is impossible using (shared-key) MAC schemes, and allows many applications, such as signing an agreement, payment order or recommendation/review. An important special case is

signing a *public key certificate*, linking between an entity and its public key.

**Establishing security.** Using public key cryptology, we can establish secure communication between parties, without requiring them to previously exchange a secret key or to communicate with an additional party (such as a *KDC*, see § 5.5). We can send the public key signed by a trusted party (in a certificate); or, if the attacker is only eavesdropper, use a key-exchange protocol (this is not secure against a MitM attacker). Finally, in the common case where one party (the client) knows the public key of the other party (the server), the client can encrypt a shared key and send to the server.

**Stronger resiliency to exposure.** In § 5.6 we discussed the goal of resiliency to exposure of secret information, in particular, of the ‘master key’ of shared-key key-setup protocols, and presented the *forward secrecy key-setup handshake*. In subsection 5.6.3, we also briefly discussed some stronger resiliency properties, including *Perfect Forward Secrecy (PFS)*, *Threshold security* and *Proactive security*. Designs for achieving such stronger resiliency notions are all based on public key cryptology; see later in this chapter.

Public key cryptology (PKC) also makes it *easier* to design and deploy secure systems. Specifically, public keys are easier *to distribute*, since they can be given in a public forum (such as directory) or in an incoming message; note that the public keys still need to be authenticated, to be sure we are receiving the correct public keys, but there is no need to protect their secrecy. Distribution is also easier since each party only needs to distribute one (public) key to all its peers, rather than setting up different secret keys, one per each peer.

Furthermore, public keys are easier to maintain and use, since they may be kept in non-secure storage, as long as they are validated before use - e.g., using MAC with a special secret key. Finally, only one public key is required for each party, compared to  $O(n^2)$  shared keys required for  $n$  peers; namely, we need to maintain - and refresh - less keys.

### 6.1.5 The price of PKC: assumptions, computation costs and key-length

With all the advantages listed above, it may seem that we should *always* use public key cryptology. However, PKC has three significant drawbacks: computation time, key-length and potential vulnerability. We discuss these in this subsection.

All of these drawbacks are due to the fact that when attacking a PKC scheme, the attacker has the *public key* which corresponds to the private key. The private key is closely related to the public key - for example, the private decryption key ‘reverses’ encryption using the public key; yet, the public key

should not expose (information about) the private key. It is challenging to come up with a scheme that allows this relationship between the encryption and decryption keys, and yet where the public key does not expose the private key. In fact, as discussed in subsection 1.2.1, the *concept* of PKC was ‘discovered’ twice - and in both times, it took *years* until the first realization of PKC was found; see [81].

Considering the challenge of designing asymmetric cryptosystems, it should not be surprising that all known public-key schemes have considerable drawbacks compared to the corresponding shared-key (symmetric) schemes. There are two types of drawbacks: *overhead* and *required assumptions*.

**PKC assumptions and quantum cryptanalysis** Applied PKC algorithms, such as RSA, DH, El-Gamal and elliptic-curve PKCs, all rely on specific computational assumptions, mostly on the hardness of specific number-theoretic problems such as factoring; there are other proposed algorithms, but their overheads are (even) much higher.

There is reason to hope that these specific hardness assumptions are well-founded. The basic reason is that many mathematicians have tried to find efficient algorithms for these problems for many years, long before their use for PKC was proposed; and efforts increased by far as PKC became known and important.

However, it is certainly conceivable that an efficient algorithm exists - and would someday be found. In fact, such discovery may even occur suddenly and soon - such unpredictability is the nature of algorithmic and mathematical breakthroughs. Furthermore, since all of the widely-used PKC algorithms are so closely related, it is even possible that such cryptanalysis would apply to all of them - leaving us without any practical PKC algorithm. PKC algorithms are the basis for the security of many systems and protocols; if suddenly there will not be a viable, practical and ‘unbroken’ PKC, that would be a major problem.

Furthermore, efficient algorithms to solve these problems *are known* - if an appropriate *quantum computer* can be realized. There has been many efforts to develop quantum computers, with significant progress - but results are still very far from the ability to cryptanalyze these PKC schemes. But here, too, future developments are hard to predict.

All this motivates extensive work by cryptographers, to identify additional candidate PKC systems, which will rely on other, ‘independent’ or - ideally - more general assumptions, as well as schemes which are secure even if large-scale quantum computing becomes feasible (*post-quantum cryptology*). In particular, this includes important results of PKC schemes based on lattice problems. Lattice problems are very different from number-theoretic problems, and seem resilient to quantum-computing; furthermore, some of the results in this area have proofs of security based on the general and well-founded complexity assumption of NP-completeness. Details are beyond our scope; see, e.g., [2].

**PKC overhead: key-length and computation.** One drawback of asymmetric cryptology, is that all proposed schemes - definitely, all proposed scheme which were not broken - have much higher overhead, compared to the corresponding shared-key schemes. There are two main types of overheads: *computation time* and *key-length*.

The system designers choose the key-length of the cryptosystems they use, based on the *sufficient effective key length* principle (principle 4). These decisions are based on the perceived resources and motivation of the attackers, on their estimation or bounds of the expected damages due to exposure, and on the constraints and overheads of the relevant system resources. Finally, a critical consideration is the estimates of the required key length for the cryptosystems in use, based on known and estimated future attacks. Such estimates and recommendations are usually provided by experts proposing new cryptosystems, and then revised and improved by experts and different standardization and security organizations, publishing key-length recommendations.

We present three well-known recommendations in Table 6.1. These recommendations are marked in the table as *LV'01*, *NIST2014* and *BSI'17*, and were published, respectively, in a seminal 2001 paper by Lenstra and Verheul [80], recommendations by NIST from 2014 [7] and recommendations by the German BSI from 2017 [29]. See these and much more online at [56].

Recommendations are usually presented with respect to a particular *year* in which the ciphertexts are to remain confidential. Experts estimate the expected improvements in the cryptanalysis capabilities of attackers over years, due to improved hardware speeds, reduced hardware costs, reduced energy costs (due to improved hardware), and, often more significantly by hardest to estimate, improvements in methods of cryptanalysis. Such predictions cannot be done precisely, and hence, recommendations differ, sometimes considerably. Accordingly, Table 6.1 compares the recommended key lengths for three years: 2020, 2030 and 2040.

Table 6.1 presents the recommendations for four typical, important cryptosystems, in three columns: two to four. Column two presents the recommendations for *symmetric cryptosystems* such as AES. The same recommendations hold for any other symmetric (shared-key) cryptosystem, for which there is not ‘shortcut’ cryptanalytical attack (which provides significantly better efficiency compared to exhaustive search).

Column three presents the recommendations for RSA and El-Gamal, the two oldest and most well-known public-key cryptosystems; we discuss both cryptosystems, in sections 6.7 and 6.6.2. This column also applies to the Diffie-Hellman (DH) key-exchange protocol; in fact, the El-Gamal cryptosystem is essentially a variant of the DH protocol, as we explain in subsection 6.6.2. RSA and El-Gamal/DH are based on two different number-theoretic problems: the factoring problem (for RSA) and the discrete-logarithm problem (for DH/El-Gamal); but the best-known attacks against both are related, with running time which is exponential in *half* the key-length. We briefly discuss these problems in subsection 6.1.7.

The fourth column of table 6.1 presents the recommendations for elliptic-

Year	Symmetric (e.g., AES)			Factoring (RSA), DiscLog (DH)			Elliptic-Curve Encryption (ECIES)		
	LV '01	NIST 2016	BSI '17	LV 2002	NIST 2016	BSI '17	LV '02	NIST 2016	BSI '17
2020	86	112	128	1881	2048	2000	161	224	250
2030	93	112	128	2493	2048	3000	176	224	250
2040	101	128	128	3214	3072	3000	191	256	250
Cr++	<b>4525 MiB/s</b> <b>AES/CTR 128b</b>			0.01ms(1024b), 0.03ms(2048b)			<b>1ms (256b ECIES)</b>		

Table 6.1: Comparison of key-length and computational overheads: symmetric cryptosystems, vs. main types of asymmetric cryptosystems

curve based public-key cryptosystems such as ECIES. As the table shows, the recommended key-lengths for elliptic-curve based public-key cryptosystems are, quite consistently, much lower than the recommendations for the ‘older’ RSA and El-Gamal/DH systems; this makes them attractive in applications where longer keys are problematic, due to storage and/or communication overhead. We do not cover elliptic-curve cryptosystems in this course; these are covered in other courses and books, e.g., [2, 63, 106].

Table 6.1 shows that the required key-length is considerably higher for public-key schemes - about twice than symmetric schemes for Elliptic-curve cryptosystem, and over twenty times (!) for RSA and DH schemes. The lower key-length recommendations for Elliptic-curve cryptography, makes these schemes attractive in the (many) applications where key-length is critical, such as when communication bandwidth and/or storage are limited.

The bottom row of Table 6.1 compares the *running time* of implementations of AES with 128 bit key in counter (CTR) mode, RSA with 1024 and 2048 bit key, and 256 bit ECIES elliptic curve cryptosystem. We see that the symmetric cryptosystem (AES) is many orders of magnitude faster. It supports more  $4 \cdot 10^9$  bytes/second, compared with less than  $10^6$  bytes/second for the comparably-secure 2048-bit RSA, and less than  $32 \cdot 10^3$  bytes/second for ECIES. We used the values reported for one of the popular Cryptographic libraries, Crypto++ [36]. Energy costs would usually be even worse than this factor of 4000 to 4,000,000 times the costs of comparable-security shared key algorithms.

From bottom row, we derive the bottom line:

**Principle 10** (Minimize use of PKC). *Designers should avoid, or, where absolutely necessary, minimize the use of public-key cryptography.*

In particular, consider that typical messages are much longer than the size of inputs to the public-key algorithms. In theory, we could use designs as presented for encryption and MAC, for applying the public-key algorithms to multiple blocks. However, the resulting computation costs would have been absurd. Even more absurd would be an attempt to modify the public-key operation to support longer inputs. Luckily, there are simple and efficient solutions, to both encryption and signatures, which are used essentially universally, to apply these schemes to long (or VIL) messages:

**Signatures:** use the *hash-then-sign (HtS)* paradigm, see subsection 4.2.4.

**Encryption:** use the *hybrid encryption* paradigm, see the following subsection (subsection 6.1.6).

### 6.1.6 Hybrid Encryption

The huge performance overhead of asymmetric cryptosystems implies that they can rarely be used directly to encrypt messages, or: are usually used only for very short messages. Typical, longer messages, are usually encrypted with a symmetric key. If the parties do not have a shared symmetric key, then they setup such shared key - using either key-exchange protocol between them, or by having one party use the public key of its peer to encrypt the shared key and send it, encrypted, to the peer. The later, widely-used, technique is called *hybrid encryption*; details follow.

Hybrid encryption combines a public key cryptosystem ( $PKG, PKE, PKD$ ) with a shared key cryptosystem ( $SKG, SKE, SKD$ ), to allow efficient public key encryption of long messages. Our description assumes that the recipient knows the sender's public key  $e$ .

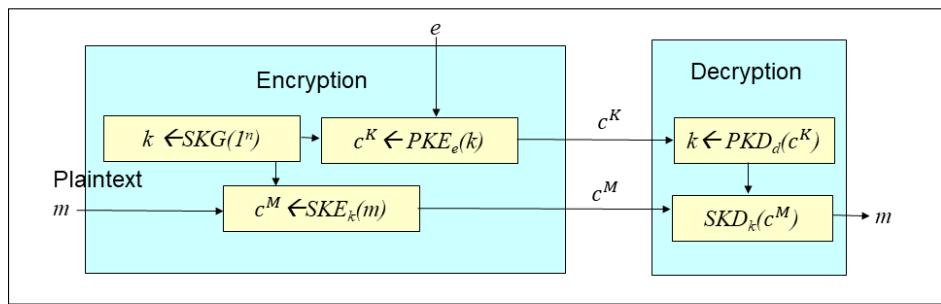


Figure 6.4: Hybrid encryption: combining public key cryptosystem ( $PKG, PKE, PKD$ ) with shared key cryptosystem ( $SKG, SKE, SKD$ ), to allow efficient public key encryption of long message  $m$  using public key  $e$ .

To perform hybrid encryption, the sender selects a random key  $k \leftarrow SKG(1^n)$ ; the sender will share  $k$  with the recipient. The sender encrypts  $k$  using the public key  $e$ , resulting in *cipher-key*  $c^K$ , namely:  $c^K = PKE_e(k)$ . The sender then encrypts the plaintext message  $m$ , using this shared key  $k$ , resulting in the *cipher-message*  $c^M = SKE_k(m)$ . Finally, we send the ciphertext  $(c^K, c^M)$  to the recipient. Decryption is left as an exercise. See Figure 6.4.

### 6.1.7 The Factoring and Discrete Logarithm Hard Problems

Public-key cryptology is based on the theory of complexity, and specifically on (computationally) *hard problem*. Intuitively, a hard problem is a family of computational problems, with two properties:

**Easy to verify:** there is an efficient algorithm to verify solutions. The efficient is the formal definitions usually refers to the standard complexity-theory notion of Probabilistic Polynomial Time (PPT) algorithms.

**Hard to solve:** there is no known efficient algorithm that solves the problem (with significant probability). We refer here to *known* algorithms; it is unreasonable to expect a *proof* that there is no efficient algorithm to a problem for which there is an efficient (PPT) verification algorithm. The reason for that is that such a proof would also solve the most important, fundamental open problem in the theory of complexity, i.e., it would show the  $NP \neq P$ ; see [57].

Intuitively, public key schemes use hard problems, by having the secret key provide the solution to the problem, and the public key provide the parameters to verify the solution. To make this more concrete, we briefly discuss *factoring* and *discrete logarithm*, the two hard problems which are the basis for many public key schemes, including the oldest and most well known: *RSA*, *DH*, *El-Gamal*. For more in-depth discussion of these and other schemes, see courses and books on cryptography, e.g., [106]. Note that while so far, known attacks are equally effective against both systems (see Table 6.1), there is not yet a proof that an efficient algorithm for one problem implies an efficient algorithm against the second.

#### Factoring

The factoring problem is one of the oldest problems in algorithmic number theory, and is the basis for RSA and other cryptographic schemes. Basically, the factoring problem involves finding the prime divisors (factors) of a large integer. However, most numbers have small divisors - half of the numbers divide by two, third divide by four and so on... This allows efficient *number sieve* algorithms to factor most numbers. Therefore, the factoring hard problem refers specifically to factoring of numbers which have only *large* prime factors.

For the RSA cryptosystem, in particular, we consider factoring of a number  $n$  computed as the product of two large random primes:  $n = pq$ . The factoring

hard problem assumption is that given such  $n$ , there is no efficient algorithm to factor it back into  $p$  and  $q$ .

Verification consists simply of multiplying  $p$  and  $q$ , or, if given only one of the two, say  $p$ , of dividing  $n$  by  $p$  and confirming that the result is an integer  $q$  with no residue.

### Discrete logarithm

The *discrete logarithm* problem is another important, well-known problem from algorithmic number theory - and the basis for the DH (Diffie-Hellman) key-exchange protocol, the El-Gamal cryptosystem, elliptic-curve cryptology, and additional cryptographic schemes.

**Note:** readers are expected to be familiar with the basic notions of number theory. Our discussion of the public key cryptosystems (RSA, DH, El-Gamal) and the relevant mathematical background is terse; if desired/necessary, please consult textbooks on cryptology and/or number theory, e.g., [106].

Discrete logarithms are defined for a given *cyclic group*  $G$  and a generator  $g$  of  $G$ . A cyclic group  $G$  is a group whose elements can be generated by a single element  $g \in G$ , called a *generator* of the group, by repeated application of the *group operation*. The usual notation for the group operation is the same as the notation of multiplication in standard calculus, i.e., the operation applied to elements  $x, y \in G$  is denoted  $x \cdot y$  or simply  $xy$ , and the group operation applied  $a$  times to element  $x \in G$ , where  $a$  is an integer (i.e.,  $a \in \mathbb{N}$ ), is denoted  $x^a$ ; in particular,  $g^a = \prod_{i=1}^a g$ . Applying these notations, a cyclic group is a group where  $(\forall x \in G)(\exists a \in \mathbb{N})(x = g^a)$ .

Given generator  $g$  of cyclic group  $G$  and an element  $x \in G$ , the *discrete logarithm* of  $x$  over  $G$ , w.r.t.  $g$ , is the integer  $a \in \mathbb{N}$  s.t.  $x = g^a$ . This is similar to the ‘regular’ logarithm function  $\log_b(a)$  over the real numbers  $\mathbb{R}$ , which returns the number  $\log_b(a) \equiv x \in \mathbb{R}$  s.t.  $b^x = a$ , except, of course, that discrete logarithms are restricted to integers, and defined for given cyclic group  $G$ .

There are efficient algorithms to compute logarithms over the reals; these algorithms work, of course, also when the result is an integer. However, for *some* finite cyclic groups, computing discrete logarithms is considered hard. Note that the *verification* of discrete logarithms only requires exponentiation, which can be computed efficiently.

**Definition 6.1** (Discrete logarithm problem). *Let  $G$  be a cyclic group,  $g$  be a generator for  $G$  and  $x \in G$  be an element of  $G$ . A discrete logarithm of  $x$ , for group  $G$  and generator  $g$ , is an integer  $a \in \mathbb{N}$  s.t.  $x = g^a$ . The discrete logarithm problem for  $G$  is to compute integer  $a \in \mathbb{N}$  s.t.  $x = g^a$ , given  $x \in G$  and generator  $g$  for  $G$ .*

In practical cryptography, the discrete logarithm problem is used mostly for groups defined by multiplications ‘mod  $p$ ’ over the group  $\mathbb{Z}_p^* \equiv \{1, 2, \dots, p-1\}$  for a prime  $p$ , or for elliptic curve groups. We focus on the ‘mod  $p$ ’ groups. For

such groups, the discrete logarithm problem is to compute integer  $a \in \mathbb{N}$  s.t.  $x = g^a \pmod{p}$ , given prime integer  $p \in \mathbb{N}$  and  $x, g \in \mathbb{Z}_p^* \equiv \{1, 2, \dots, p-1\}$ .

When  $p-1$  has only ‘small’ prime factors, then there are known algorithms, such as the *Pohlig-Hellman algorithm*, that efficiently compute discrete logarithms. To avoid this and other known efficient discrete-logarithm algorithms, one common solution is to use a modulus  $p$  which is not just a prime, but a *safe prime*. A prime  $p$  is *safe* if  $p = 2q + 1$ , where  $q$  is also a prime. Calculation of discrete logarithm is considered computationally-hard for  $\mathbb{Z}_p^* \equiv \{1, 2, \dots, p-1\}$ , when using a *safe prime*  $p = 2q + 1$  (for prime  $q$ ); note that it is easy to find a generator for safe prime groups. In other words, the probability of finding the discrete logarithm is *negligible*. We next define safe-primes and the DLA assumption.

**Definition 6.2** (Discrete logarithm assumption and safe primes). *A prime number  $p \in \mathbb{N}$  is called a safe prime, if  $p = 2q + 1$  for some prime  $q \in \mathbb{N}$ . The discrete logarithm assumption (DLA) for safe prime groups holds, if for every PPT algorithm  $\mathcal{A}$  holds:*

$$\Pr_{\substack{a \leftarrow \mathbb{Z}_p^*}} [a = \mathcal{A}(g^a \pmod{p})] \in \text{NEGL}(n) \quad (6.4)$$

for every safe prime  $p$  of at least  $n$  bits, and every generator  $g$  of  $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$ .

**Least-significant bits of the discrete logarithm are computable.** It is possible to efficiently compute some *partial* information about the exponent  $a$ . In particular, the exponent  $a$  is even, i.e., its least-significant bit is zero, if and only if  $g^a \pmod{p}$  is a *quadratic residue*, i.e.:

$$g^a \pmod{p} \in QR(p) \equiv \{x^2 \pmod{p} \mid x \in \mathbb{Z}_p^*\} \quad (6.5)$$

*Euler’s criterion* allows to efficiently test if  $g^a \pmod{p}$  is a *quadratic residue*, hence, to find the least-significant bit of  $a$ . These well-known facts are summarized in the following lemma.

**Lemma 6.1** (Euler’s criterion). *Given  $y = g^a \pmod{p}$ , for  $a \in \mathbb{Z}_p^*$ , the least-significant bit of  $a$  is zero if and only if  $y \in QR(p)$ , which holds if and only if Euler’s criterion holds for  $y = g^a \pmod{p}$ , namely if:*

$$y^{(p-1)/2} = 1 \pmod{p} \quad (6.6)$$

*Proof:* See in textbooks on number theory and/or cryptography, e.g., [106].  $\square$

**Discrete logarithms over other groups.** Using  $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$  with a *safe prime*  $p$  is not the only option. Sometimes there are computational and even security advantages in using other finite cyclic groups for which the

discrete logarithm problem is considered hard, rather than using safe prime groups. However, the use of such non-safe groups should be done carefully; multiple ‘attacks’, i.e., efficient discrete-logarithm algorithms, are known for some groups. In particular, discrete logarithm can be computed efficiently when  $p - 1$  is *smooth*, which means that  $p - 1$  has only small prime factors, e.g., when  $p$  is of the form  $p = 2^x + 1$ .

This problem is not just a theoretical concern; [108] shows weaknesses in the groups used by multiple implementations, including in popular products such as OpenSSL, and discusses the reasons that may have led to the insecure choices. Therefore, we mostly focus on the use of safe-prime groups.

## 6.2 The DH Key Exchange Protocol

A major motivation for public key cryptology, is to secure communication between parties, without requiring the parties to previously agree on a shared secret key. In their seminal paper [40], Diffie and Hellman introduced the *concept* of public key cryptology, including public-key cryptosystem (PKC), which indeed allows secure communication without pre-shared secret key. However, this paper did not contain a proposal for *implementing* a PKC.

Instead, [40] introduced the *key exchange* problem, and present the *Diffie-Hellman (DH) key exchange protocol*, often referred to simply as the DH protocol. Although a key-exchange protocol is not a public key cryptosystem, yet it also allows secure communication - without requiring a previously shared secret key. In fact, the *goal* of a key exchange protocol, is to *establish* a shared secret key.

In this section, we explain the DH protocol, by developing it in three steps - each in a subsection. In subsection 6.2.1 we discuss a ‘physical’ variant of the DH protocol, which involves physical padlocks and exchanging a box (locked by one or two locks).

### 6.2.1 Physical key exchange

To help understand the Diffie-Hellman key exchange protocol, we first describe a *physical* key exchange protocol, illustrated by the sequence diagram in Fig. 6.5. In this protocol, Alice and Bob exchange a secret key, by using a *box*, and two padlocks - one of Alice and one of Bob. Note that initially, Alice and Bob do not have a shared key - and, in particular, Bob cannot open Alice’s padlock and vice versa; the protocol nevertheless, allows them to securely share a key.

Alice initiates the protocol by placing the key to be shared in the box, and locking the box with her padlock. When Bob receives the locked box, he cannot remove Alice’s padlock and open the box. Instead, Bob *locks* the box with his own padlock, *in addition to Alice’s padlock*. Bob now sends the box, locked by *both* padlocks, to Alice.

Upon receiving the box, locked by both padlocks, Alice removes her own padlock and sends back the box, now locked only by Bob’s padlock, back to

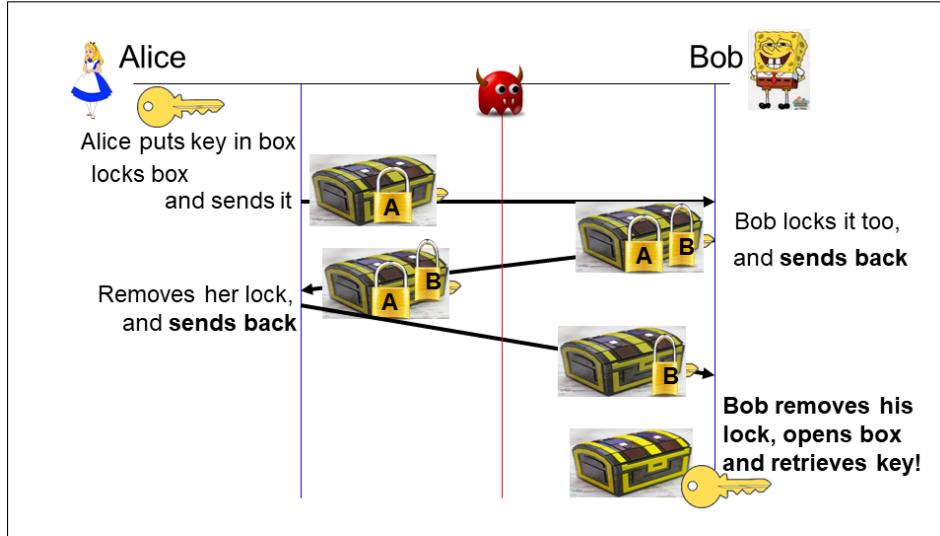


Figure 6.5: Physical Key Exchange Protocol

Bob. Finally, Bob removes his own padlock, and is now able to open the box and find the key send by Alice. We assume that the adversary - Monster in the Middle - cannot remove Alice's or Bob's padlocks, and hence, cannot learn the secret in this way. The Diffie-Hellman protocol replaces this physical assumption, by appropriate cryptographic assumptions.

However, notice that there is a further limitation on the adversary, which is crucial for the security of this physical key exchange protocol: the adversary should be unable to send a *fake padlock*. Note that in Figure 6.5, both padlocks are stamped by the initial of their owner - Alice or Bob. The protocol is not secure, if the adversary is able to put her own padlock on the box, but stamping it with A or B, and thereby making it appear as Alice's or Bob's padlock, respectively. This corresponds to the fact that the Diffie-Hellman protocol is only secure against an *eavesdropping adversary*, but insecure against a *MitM* adversary.

The critical property that facilitated the physical key exchange protocol, is that Alice can remove her padlock, even after Bob has added his own padlock. Namely, the 'padlock' operation is 'commutative' - it does not matter if Alice placed her padlock first and Bob second, she can still remove her padlock as is it was applied last. In a sense, the key to cryptographic key exchange protocols such as Diffie Hellman, is to perform a mathematical operation which is also commutative. s

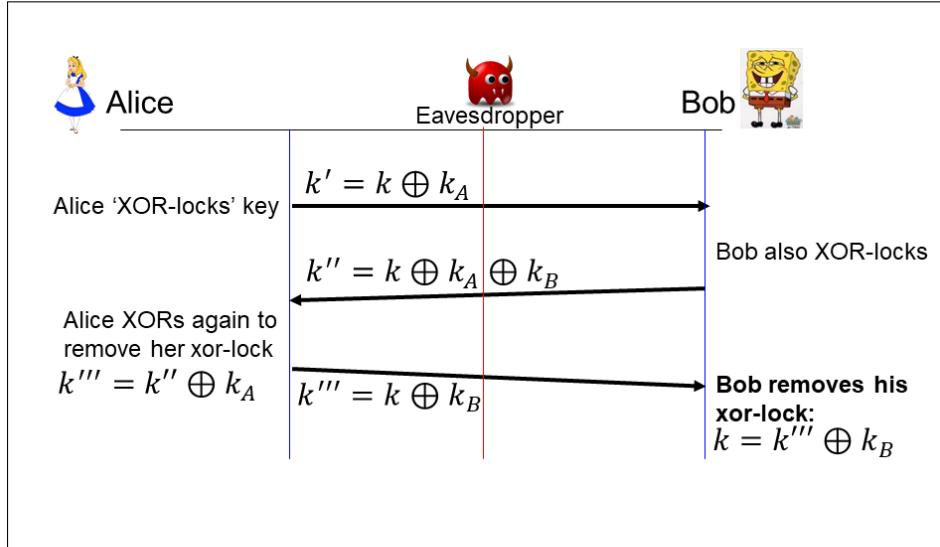


Figure 6.6: XOR Key Exchange Protocol

### 6.2.2 Key exchange protocol: prototypes

In this subsection, we present few ‘prototype’ key-exchange protocols, which help us to properly explain the Diffie-Hellman protocol. Unlike the physical key exchange protocol of subsection 6.2.1, these are ‘real protocols’, i.e., involve only exchange of messages - no physical objects or assumptions.

**XOR, Addition and Multiplication key exchange protocols.** The sequence diagram in Figure 6.6 presents the first prototype: the *XOR key exchange protocol*. This prototype tries to use the XOR operator, to ‘implement the padlocks’ of Figure 6.5. XOR is a natural candidate, as it is commutative - and known to provide confidentiality, when used ‘correctly’ (as in one-time pad).

However, as the next exercise shows, the XOR key exchange protocol allows an attacker to find out the exchanged key.

**Exercise 6.1** (XOR key exchange protocol is insecure). *Show how an eavesdropping adversary may find the secret key exchanged by the XOR Key Exchange protocol, by (only) using the values sent between the two parties.*

*Solution (sketch):* attacker XORs all three messages, to obtain:  $k = (k \oplus k_A) \oplus (k \oplus k_A \oplus k_B) \oplus (k \oplus k_B)$ .  $\square$

The following exercise shows that a similar vulnerability occurs if we use multiplication or addition instead of XOR.

**Exercise 6.2** (Addition/multiplication key exchange is insecure). *Present two sequence diagrams similar to Figure 6.6, but using addition, and then using*

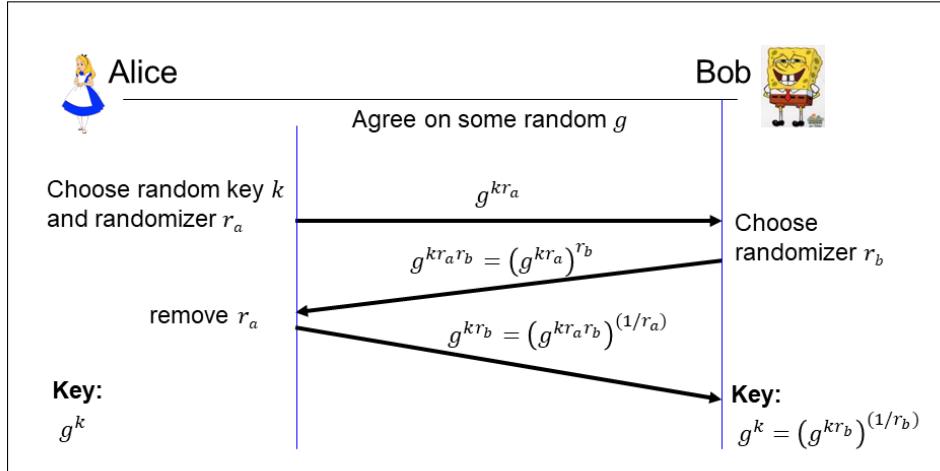


Figure 6.7: Exponentiation Key Exchange Protocol

*multiplication, instead of XOR. Shows that both versions are vulnerable to MitM attackers, similar to Exercise 6.1.*

**Goal is limited to security against eavesdropper.** Notice that the attacker against the XOR key exchange protocol, only needs to *eavesdrop* on the communication, as marked in Figure 6.6. In fact, all the key-exchange protocols in this section, including the Diffie-Hellman protocol, are evaluated only against an eavesdropping adversary. To ensure security against Monster-in-the-Middle (MitM) attacker, see the *authenticated key-exchange* protocols of subsection 6.4.1.

**Exponentiation key exchange.** So far, we tried to ‘implement’ the commutative property of physical keys, using three commutative mathematical operators: XOR, addition and multiplication, and all turned out insecure. Essentially, the vulnerabilities stem from the fact that the attacker was able to ‘remove’ elements, by performing the *dual* operation: subtraction for addition and division for multiplication. We remove XOR using XOR - but that is still consistent, since XOR is the ‘dual operation’ to itself.

In Fig. 6.7, we try to use another basic commutative mathematical operator: *exponentiation*.

Unfortunately, the exponentiation operation may also be removed to find the exponent - by computing the logarithm. The logarithm function is not as efficient as addition, multiplication or even exponentiation, but, over the integers or real numbers, it is still a rather efficient operation.

If the base  $g$  is known, then an eavesdropper can simply remove all the exponentiations, which reduces the protocol to the (insecure) multiplication key exchange (Ex. 6.2). Namely, the attacker applies the logarithm operator,

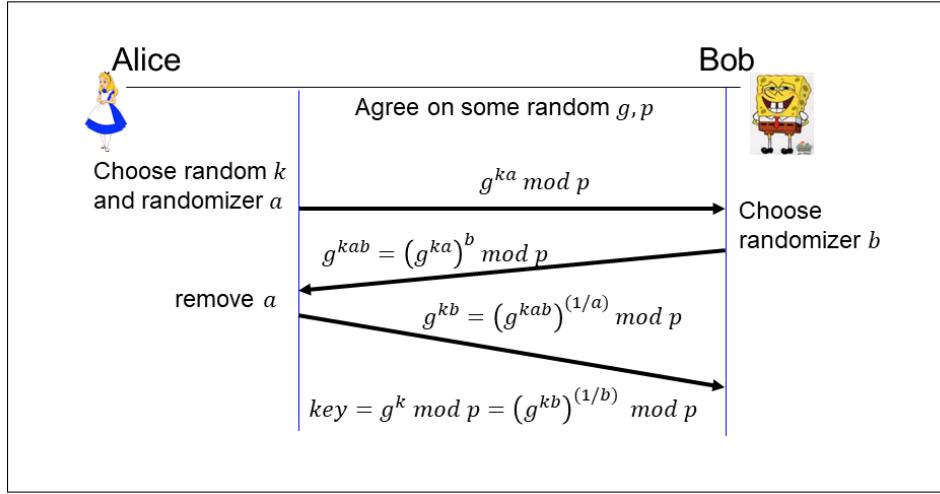


Figure 6.8: Modular-Exponentiation Key Exchange Protocol, for safe prime  $p$  groups. The values  $k, a$  and  $b$  are integers smaller than  $p - 1$ .

with basis  $g$ , to the three messages of Fig. 6.7 - resulting in the values  $k \cdot r_a$ ,  $k \cdot r_a \cdot r_b$  and  $k \cdot r_b$ . The attacker can now combine these three values to find  $k$ , as in Exercise 6.2.

But what if  $g$  is randomly chosen by Alice and not made public? Then, the attacks of Exercises 6.1 and 6.2 above, against the XOR, addition and multiplication key exchange protocols, do *not* apply against the *Exponentiation Key Exchange Protocol*, as there is an obvious way to use  $g^x$  in order to compute  $g^y$  out of  $(g^x)^y$ .

Unfortunately,  $g$  is not really needed - we can ‘fix’ the attack to work without it. The attacker can compute the logarithm of the second flow  $(g^{kr_a})^{rb}$  for the base of the first flow  $g^{kr_a}$ , which gives the exponent  $r_b$  - and hence allows the attacker to find the key  $g^k$  just like Bob finds it. Hence, the Exponentiation Key Exchange Protocol (Fig. 6.7) is, indeed, insecure. However, we will not give up - and we next show how we ‘fix’ this protocol, and finally present a possibly-secure key exchange protocol!

**Modular-Exponentiation Key Exchange.** We now try to ‘fix’ the Exponentiation Key Exchange Protocol (Fig. 6.7). The attack against it used the fact that the computations in Fig. 6.7 are done over the field of the real (or natural) numbers -  $\mathbb{R}$  (or  $\mathbb{N}$ ), where there are efficient algorithms to compute logarithms. This motivates changing this protocol, to use, instead, operations over a group in which the (discrete) logarithm problem is considered hard. Such groups exist, e.g., the ‘mod  $p$ ’ group, for safe prime  $p$ . We present this variant in Fig. 6.8.

One way to try to break the Modular-Exponentiation Key Exchange Pro-

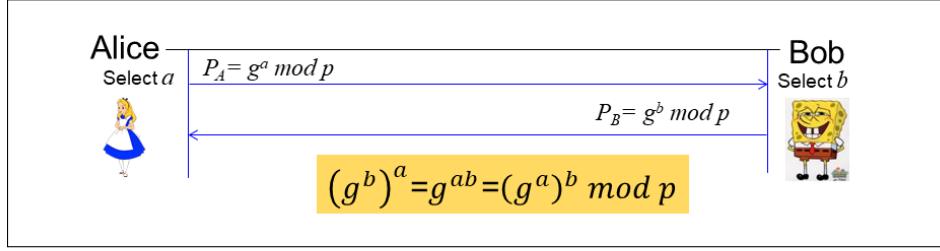


Figure 6.9: Diffie-Hellman Key Exchange Protocol, for safe prime  $p$  groups;  $a, b$  are integers smaller than  $p - 1$ .

tocol of Fig. 6.8, is to compute the (discrete) logarithm of three the values exchanged by the protocol - like the attack above against the ‘regular’ Exponentiation Key Exchange Protocol. This works when discrete logarithm can be computed efficiently, e.g., when  $p - 1$  is *smooth*, i.e., has only small prime factors, e.g.,  $p = 2^x + 1$ .

However, this attack requires computing discrete logarithms - and this is believed to be computationally-hard for certain modulus. In particular, discrete logarithm is assumed to be computationally hard when the modulus  $p$  is a large *safe prime*, i.e.,  $p = 2q + 1$  for some prime  $q$ , see Def. 6.2.

In the following subsection, we present the Diffie-Hellman protocol - which is essentially an improved variant of the Discrete-Exponentiation Key Exchange Protocol of Fig. 6.8.

### 6.2.3 The Diffie-Hellman Key Exchange Protocol and Hardness Assumptions

In Fig. 6.9, we - finally - present the Diffie-Hellman (DH) key exchange protocol, for safe prime groups. The protocol assumes that the parties agree on a safe prime  $p$  and on a generator  $g$  for the multiplicative group  $Z_p^*$ . The protocol consists of only two flows: in the first flow, Alice sends  $g^a \bmod p$ , where  $a$  is a *private key* chosen by Alice; and in the second flow, Bob responds with  $g^b \bmod p$ , where  $b$  is a private key chosen by Bob. The result of the protocol is a shared secret value  $g^{ab} \bmod p$ , computed by Alice as  $g^{ab} \bmod p = (g^b \bmod p)^a \bmod p$ , and by Bob as  $g^{ab} \bmod p = (g^a \bmod p)^b \bmod p$ .

The Diffie-Hellman protocol is, essentially, a simplified, and slightly optimized, variant of the Discrete-Exponentiation Key Exchange Protocol of Fig. 6.8. The basic difference is that instead of exchanging  $g^k \bmod p$ , for some random  $k$ , we now exchange  $g^{ab}$ ; this is a bit simpler, and more efficient: only two flows instead of three, no need to compute inverses  $(a^{-1}, b^{-1})$ , and one less exponentiation.

Notice that both Diffie-Hellman and the Modular-Exponentiation key exchange protocol, are designed only against an *eavesdropping adversary*, and are

definitely insecure against a MitM attacker. Show this by solving the following exercise.

**Exercise 6.3.** *Present a schedule diagram showing that the DH protocol is insecure against a MitM attacker. Specifically, show that a MitM attacker can cause Alice and Bob to believe they have a secure channel between them, using shared key exchanged by DH, while in fact the attacker has the keys they use to communicate.*

*Hint:* the attacker impersonates as Alice to Bob and as Bob to Alice, thereby establishing a shared key with both of them, who now believe they use a secret shared key.  $\square$

In fact, all a MitM attacker needs to do is to fake the message from a party, allowing it to impersonate as that party (with a shared key with the other party). Indeed, in practice, we always use *authenticated* variants of the DH protocol, as we discuss in subsection 6.4.1.

Also, note that both Diffie-Hellman and the Modular-Exponentiation key exchange protocols, generalize to any multiplicative group. However, their security requires the discrete-logarithm to be a hard problem over that group, which does not hold for all groups - e.g., it does not hold over the real numbers, and also not for many modular groups, even with prime modulo  $p$  (e.g., if  $p - 1$  is ‘smooth’, i.e., a multiplication of small primes).

So, can we assume that the DH protocol is secure against an *eavesdropping* adversary, when computed over a group in which discrete logarithm is assumed ‘hard’, e.g., the ‘mod  $p$ ’ group where  $p$  is a safe-prime? The answer is not trivial: can an adversary, knowing  $g^b \pmod{p}$  and  $g^a \pmod{p}$ , somehow compute  $g^{ab} \pmod{p}$ , without requiring knowledge of  $a, b$  or  $ab$ ? This is one of the important open questions in applied cryptography; there has been several (positive) results, but none of them yet applies to practical groups such as the ‘mod  $p$ ’ group.

The standard approach is to assume a *stronger* assumption, such as the *Computational DH (CDH)* assumption. The CDH assumption essentially means that it is infeasible to compute the DH shared secret, when using the ‘mod  $p$ ’ group, for safe-prime  $p$ . The assumption generalizes to some other groups - but not to others, e.g., it does not hold over the real numbers.

**Definition 6.3 (Computational DH (CDH) for safe prime groups).** *The Computational DH (CDH) assumption for safe prime groups holds, if for every PPT algorithm  $\mathcal{A}$ , every constant  $c \in \mathbb{R}$ , and every sufficiently-large integer  $n \in \mathbb{N}$ , holds:*

$$\Pr_{\substack{a,b \xleftarrow{\$} Z_p^*}} [\mathcal{A}(g^a \pmod{p}, g^b \pmod{p}) = g^{ab} \pmod{p}] < n^c \quad (6.7)$$

*for every safe prime  $p$  of at least  $n$  bits and every generator  $g$  of  $Z_p^* = \{1, 2, \dots, p-1\}$ .*

However, note that even if the CDH assumption for safe prime groups holds, an eavesdropper may still be able to learn some *partial* information about  $g^{ab} \bmod p$ . In particular, from Lemma 6.1, an eavesdropper can use Euler's criterion (Eq. 6.6) to find out if  $g^{ab} \bmod p \in QR(p)$ , i.e., is a quadratic residue.

Therefore, using  $g^{ab} \bmod p$  directly as a key, may allow an attacker to learn some partial information about the key - even assuming that the CDH assumption is true. So, how can we use the DH protocol to securely exchange a key ? There are two main options; we present them below.

**First option: use DDH groups.** The first option is not to use a safe prime group; instead, use a group which is believed to satisfy a stronger assumption - the *Decisional DH (DDH) Assumption*. Some of the groups where DDH is assumed to hold, and are used in cryptographic schemes, include *Schnorr's group* and some elliptic-curve groups; we will not get into such details (see, e.g., in [106]).

**Definition 6.4** (The Decisional DH (DDH) Assumption). *The Decisional DH (DDH) Assumption is that there is no PPT algorithm  $\mathcal{A}$  that can distinguish, with significant advantage compared to guessing, between  $[g^a, g^b, g^{ab}]$  and  $[g^a, g^b, g^c]$ , for random  $a, b$  and  $c$ . A group for which the DDH assumption is believed to hold, is called a DDH group.*

Note, however, that this technique requires use of a group where DDH is assumed to hold, and in particular, a safe-prime group cannot be used; this is a significant hurdle.

**Exercise 6.4** (Safe prime groups do not satisfy DDH). *Show that safe prime groups are not DDH groups, i.e., they do not satisfy the DDH assumption.*

Another challenge with the use of this method is that the resulting shared-value  $g^{ab}$  is 'only' indistinguishable from a random group element, but not from a random string. This implies that we cannot use  $g^{ab}$  directly as a key. However, this challenge was essentially resolved by Fouque et al. in 2006 [51], who showed that the least-significant bits of  $g^{ab}$  are indistinguishable from random - enough bits to use as a shared key, for typical key-length and prime size.

**Second option: use CDH group, with key derivation.** The second option is to use a safe prime group - or another group where 'only' CDH is assumed to hold - but not use the bits of  $g^{ab} \bmod p$  directly as a key. Instead, we *derive* a shared key  $k$  from  $g^{ab} \bmod p$ , using a *Key Derivation Function (KDF)*. We discuss key derivation in the following subsection.

### 6.3 Key Derivation Functions (KDF) and the Extract-then-Expand paradigm

[Section not yet completed]

Key derivation is useful whenever we want to generate a key  $k$  from some ‘imperfect secret’  $x$ ; such imperfect secret is a string which is longer than the derived key, but is not fully random; e.g.,  $x = g^{ab} \bmod p$  is a typical example.

*Key derivation* is done by applying a function to the shared secret  $g^{ab}$ ; this function may be randomized or deterministic.

**Key derivation function (KDF).** A (randomized) *Key Derivation Function (KDF)* has two inputs,  $k = KDF_s(x)$ . The first input,  $x$ , is the imperfect secret; the second input,  $s$  (*salt*), is a public random value, used to ensure that the output key is uniformly random; details follow.

### Deterministic key derivation using a randomness-extracting hash function.

An alternative is to derive the key as  $k = h(g^{ab} \bmod p)$ , where  $h$  is a (deterministic) *randomness extracting hash function*. A randomness extracting hash function does not require the parties to share a uniformly-random string  $s$  (*salt*), in contrast to a KDF. In typical applications of key-exchange, such shared randomness is not available, so that is an important advantage. We discuss randomness-extraction and other properties of cryptographic hash function in chapter 4.

We focus in this section on (randomized) Key Derivation Functions. Intuitively, a KDF ensures that its output  $k = KDF_s(x)$  is pseudorandom, provided that the salt  $s$  is uniformly-random, and that the imperfect secret  $x$  is ‘sufficiently unpredictable’.

To define this ‘sufficiently unpredictable’ requirement, we assume that the imperfect secret  $x$  is generated by some probabilistic algorithm  $G$ , i.e.,  $x = G(r)$  where  $r$  is a uniformly random string. We refer to  $G$  as the *secret sampler*; it models the process using which we generate the shared secret  $x$ . Note that often, imperfect secrets are not really the product of an algorithm, but are due to some physical process; the algorithm  $G$  is just a model of the process generating the randomness. Of course, the output of the Diffie-Hellman key exchange can be precisely modeled as a sampler  $G$ .

**Definition 6.5** (Key Derivation Function (KDF) wrt  $G$ ). *A key derivation function (KDF)  $KDF_s(x)$  is a PPT algorithm with two inputs, a salt  $s$  and an imperfect secret  $x$ . We say that  $KDF_s(x)$  is secure with respect to PPT algorithm  $G$  (secret sampler), if for every PPT algorithm  $D$  (distinguisher) holds:*

$$\Pr_{s,x}[D(s, KDF_s(G(x))) = \text{‘Rand’}] - \Pr_{s,r}[D(s, r) = \text{‘Rand’}] \in \text{NEGL} \quad (6.8)$$

Where the probabilities are taken over random coin tosses of  $ADV$ , and over  $s, x, r \xleftarrow{\$} \{0, 1\}^n$ .

For more details on key derivation, including provably-secure constructions, see [77].

	PRF $f_k(x)$	KDF $f_k(x)$	Rand.-extracting hash $h(x)$
Key $k$	Secret, random	Public, random	No key
Data $x$	Arbitrary	‘Sufficiently’ secret and random	‘Sufficiently’ secret and random
Output	Pseudorandom	Pseudorandom	Pseudorandom

Table 6.2: Comparison: PRF, KDF and Randomness-extracting hash function

Note that a KDF seems quite similar, in its ‘signature’ (inputs and outputs), to a Pseudo-Random Function (PRF). Let’s compare PRF to KDF, adding to the comparison another related primitive - *randomness-extracting hash function*. See also Table 6.2.

We first note that the goal of all three mechanisms is to output uniformly pseudorandom strings (unpredictable by attacker). Furthermore:

- Both KDF and PRF use, for this purpose, a uniformly-random key; however, PRF require this key to be secret, while KDF allows a public key. Of course, randomness-extracting hash does not require any key (so, in a sense, it’s more ‘extreme’ than KDF).
- A PRF ensures that the output is pseudorandom, even for arbitrary input, possibly known or even determined by the attacker. In contrast, both KDF and randomness-extracting hash assume that the input is ‘sufficiently’ secret and random. For one definition of ‘sufficiently random input’ , see the hashing lecture (where I presented such definition for randomness-extracting hash). But for this course purposes, an intuitive understanding should suffice.

The following exercise shows that the KDF and PRF definitions are ‘incomparable’ : a function can be a PRF but not a KDF and vice versa. An interesting example is the popular HMAC construction, which is used in many applications. Some of these applications rely on HMAC to satisfy the PRF properties, e.g., for message authentication; other applications rely on HMAC to satisfy the KDF properties, e.g., to derive a key from  $g^{ab} \bmod p$  as in the DH key exchange; and yet other applications rely on HMAC to satisfy *both* the KDF properties *and* the PRF properties, e.g., strong resiliency to key exposure of the *Double-ratchet key-exchange* protocols, see subsection 6.4.4.

**Exercise 6.5** (PRF vs. KDF). *Assume that  $f$  is a (secure) PRF and  $g$  is a (secure) KDF. Derive from these a (secure) PRF  $f'$  and a (secure) KDF  $g'$ , such that  $f'$  is not a secure KDF and  $g'$  is not a secure PRF.*

## 6.4 Using DH for Resiliency to Exposures: PFS and PRS

As discussed above, and demonstrated in Ex. 6.3, the DH protocol is vulnerable to a MitM attacker; its security is only against a passive, eavesdropping-only

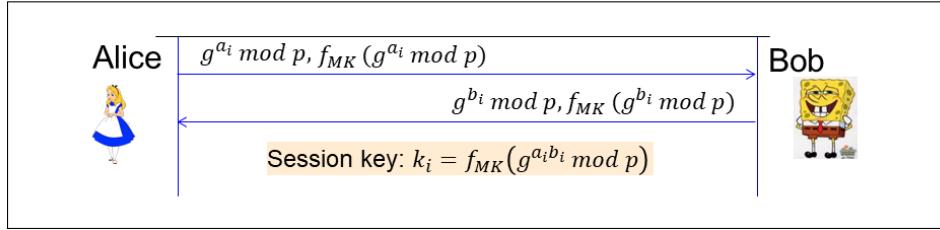


Figure 6.10: The Authenticated DH Protocol, ensuring PFS (but not recover security)

attacker. In most practical scenarios, attackers who are able to eavesdrop, have some or complete ability to also perform active attacks such as message injection; it may seem that DH is only applicable to the relatively few scenarios of eavesdropping-only attackers.

In this section, we discuss extensions of the DH protocol, extensively in practice to improve resiliency to adversaries which have MitM abilities, combined with key-exposure abilities. Specifically, these extensions allow us to ensure the *Perfect Forward Secrecy (PFS)* and *Perfect Recover Secrecy (PRS)*, the two strongest notions of resiliency to key exposures of secure key setup protocols as presented in Table 5.2 (§ 5.6).

#### 6.4.1 Authenticated DH: Perfect Forward Secrecy (PFS)

Assuming that the parties share a secret master key  $MK$ , it is quite easy to extend the DH protocol in order to protect against MitM attackers. All that is required is to use a *Message Authentication Code (MAC)* scheme, and authenticate the DH flows. In particular, we can use a pseudo-random function (PRF), say  $f$ , for both a MAC and to derive the key; see Fig. 6.10, showing an *authenticated* variant of the DH protocol.

**Lemma 6.2** (Informal: authenticated DH ensures PFS). *The Authenticated DH protocol (Fig. 6.10) ensures secure key-setup with perfect forward security (PFS).*

*Sketch of proof:* The PFS property follows immediately from the fact that  $k_i$ , the session key exchanged during session  $i$ , depends only on the result of the DH protocol, i.e., is secure against an eavesdropping-only adversary. The protocol also ensures secure key setup, since a MitM adversary cannot learn  $MK$  and hence cannot forge the DH messages.  $\square$

**Exercise 6.6.** *Alice and Bob share master key  $MK$  and perform the authenticated DH protocol daily, at the beginning of every day  $i$ , to set up a ‘daily key’  $k_i$  for day  $i$ . Assume that Mal can eavesdrop on communication between Alice and Bob every day, but perform MitM attacks only every even day ( $i$  s.t.  $i \equiv 0 \pmod{2}$ ). Assume further that Mal is given the master key  $MK$ , on the*

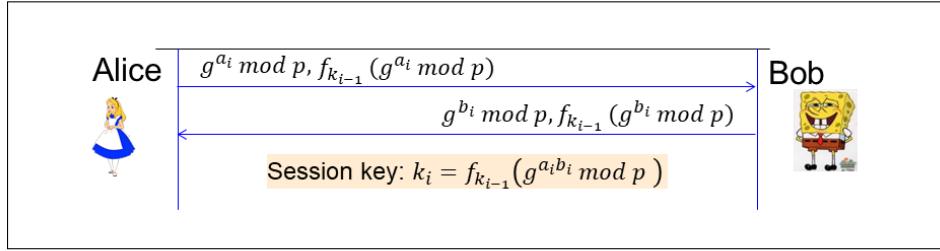


Figure 6.11: The Synchronous-DH-Ratchet protocol: PFS and PRS

*fifth day. Could Mal decipher messages sent during day  $i$ , for  $i = 1, \dots, 10$ ? Write your responses in a table.*

Note that the results of Ex. 6.6 imply that the authenticated DH protocol does *not* ensure recover security. We next show extensions that improve resiliency to key exposures, and specifically recover security after exposure, provided that the attacker does not deploy MitM ability for one handshake.

#### 6.4.2 The Synchronous-DH-Ratchet protocol: Perfect Forward Secrecy (PFS) and Perfect Recover Security (PRS)

The authenticated DH protocol ensures perfect forward secrecy (PFS), but does *not* recover secrecy; namely, a single key exposure in some time  $t$ , suffices to make all future handshakes vulnerable to MitM attacker - even if there has been some intermediate handshakes without (any) attacks, i.e., where the attacker has neither MitM nor eavesdropper capabilities. To see that the authenticated DH protocols does not ensure recovery, see the results of Ex. 6.6.

Note that the (shared key) RS-Ratchet protocol presented in subsection 5.6.2 (Fig. 5.17), achieved recovery of secrecy - albeit, not Perfect Recover Secrecy (PRS). Namely, the authenticated DH protocol does not even strictly improve resiliency compared to RS-Ratchet protocol: authenticated DH ensures PFS (which RS-Ratchet does not ensure), but RS-Ratchet ensures recovery of secrecy (which authenticated DH does not ensure).

In the remainder of this section, we present three protocols which ensure *both* PFS and PRS, by combining Diffie-Hellman (DH) with a ‘ratchet’ mechanism; we refer to these as *Diffie-Hellman Ratchet* protocols. We begin, in this subsection with the *Synchronous DH Ratchet* protocol, as illustrated in Fig. 6.11.

Like the Authenticated DH protocol presented above, the Synchronous DH Ratchet protocol also authenticated the DH exchange; hence, as long as the authentication key is unknown to the attacker *at the time when the protocol is run*, then the key exchanged by the protocol is secret. The improvement cf. to the authenticated DH protocol is in the key used to authenticate the DH exchange; instead of using a fixed master key ( $MK$ ) as done by the authenticated

DH protocol (Fig. 6.10), the Synchronous DH Ratchet protocol authenticate the  $i^{\text{th}}$  DH exchange, using the *session key* exchanged in the *previous* exchange, i.e.,  $k_{i-1}$ . An initial shared secret key  $k_0$  is used to authenticate the very first DH exchange,  $i = 1$ .

**Lemma 6.3** (Informal: Synchronous-DH-Ratchet ensures PFS and PRS). *The Synchronous-DH-Ratchet protocol (Fig. 6.11) ensures secure key-setup with perfect forward secrecy (PFS) and perfect recovery secrecy (PRS).*

*Sketch of proof:* The PFS property follows, like in Lemma 6.2, from the fact that  $k_i$ , the session key exchanged during session  $i$ , depends only on the result of the DH protocol, i.e., is secure against an eavesdropping-only adversary. The protocol also ensures secure key setup, since a MitM adversary cannot learn  $MK$  and hence cannot forge the DH messages.

The PRS property follows from the fact that if at some session  $i'$  there is only eavesdropping adversary, then the resulting key  $k_{i'}$  is secure, i.e., unknown to the attacker, since this is assured when running DH against eavesdropping-only adversary. It follows that in the following session ( $i'+1$ ), the key used for authentication is unknown to the attacker, hence the execution is again secure - and result in a new key  $k_{i'+1}$  which is again secure (unknown to attacker). This continues, by induction, as long as the attacker is not (somehow) given the key  $k_i$  to some session  $i$ , before the parties receive the messages of the following session  $i+1$ .  $\square$

#### 6.4.3 The Asynchronous-DH-Ratchet protocol

The synchronous DH-ratchet protocol ensures both PFS and PRS; we next [tbc]

In Fig. 6.12 we present the *Asynchronous-DH-Ratchet* protocol. The Asynchronous-DH-Ratchet protocol ensures both perfect forward secrecy (PFS) and perfect recover secrecy (PRS), but these properties were already ensured by the Synchronous-DH-Ratchet protocol of Fig. 6.11. However, the asynchronous protocol has a different security advantage: a party can move to a new key even when the peer is non-responsive. This can be a significant advantage, esp. in scenarios where one party may be disconnected for long periods, e.g., in securing communication between mobile users.

In the Asynchronous-DH-Ratchet protocol, the keys used by the two parties are not always synchronized; namely, at a given time, each party may use a different session key for messages it sends, and for the messages it receives. We identify each key by two indexes,  $k_{i_A, i_B}$ , where  $i_A$  counts DH messages sent by Alice, and  $i_B$  counts DH messages sent by Bob. The counter are not always the same ('in sync'), but never differ by more than one, i.e.,  $|i_A - i_B| \leq 1$ .

**Authenticating messages with the asynchronous ratchet protocol**  
(Figure 6.13)... [tbc]

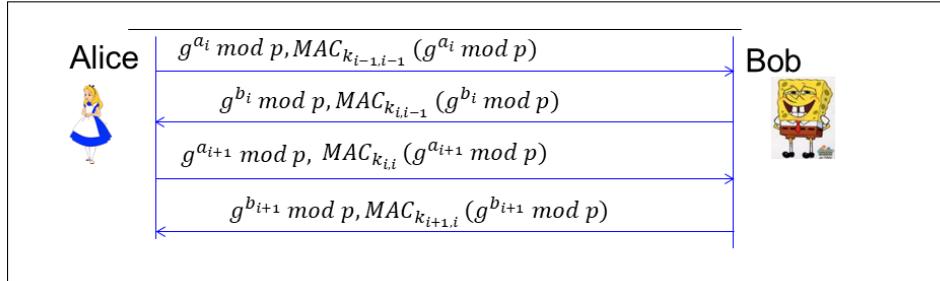


Figure 6.12: The Asynchronous-DH-Ratchet protocol

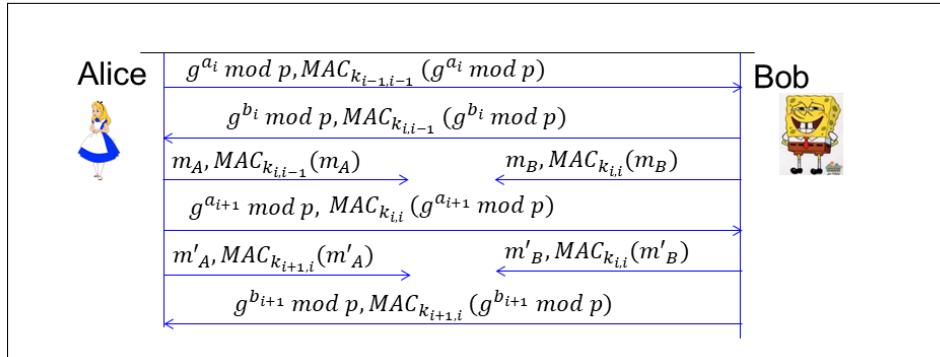


Figure 6.13: Authenticating messages with the Asynchronous-DH-Ratchet protocol

#### 6.4.4 The Double Ratchet Key-Exchange protocol

. The ratchet key-exchange protocols, presented above, provided very strong resiliency to key exposure: *perfect-forward secrecy ((PFS))* and *perfect-recover security ((PRS))*. However, to ensure such resiliency with respect to periods of length  $T$  seconds, requires performing the ratchet exchange - and the two modular exponentiation operations it involves (per party) - once every  $T$  seconds. Modular exponentiation is ‘efficient’ in the sense of requiring only polynomial running time, but this time is still quite high for frequent application.

As a result, practical deployments of ratchet key-exchange protocols would usually use rather ‘long’ periods, i.e., use large values of  $T$ . To provide additional security against key-exposure, even between runs of the ratchet exchange (involving modular exponentiations), we combine the strong PFS+PRS security of the ratchet key-exchange protocols, with the weaker security guaranteed of the *Recover-Security Handshake* protocol of subsection 5.6.2. The resulting *Double-Ratchet Key-Exchange* protocol is illustrated in Figure 6.14; this is the essence of the TextSecure protocol, deployed in popular instant messaging applications, e.g., WhatsApp.

[TBC]

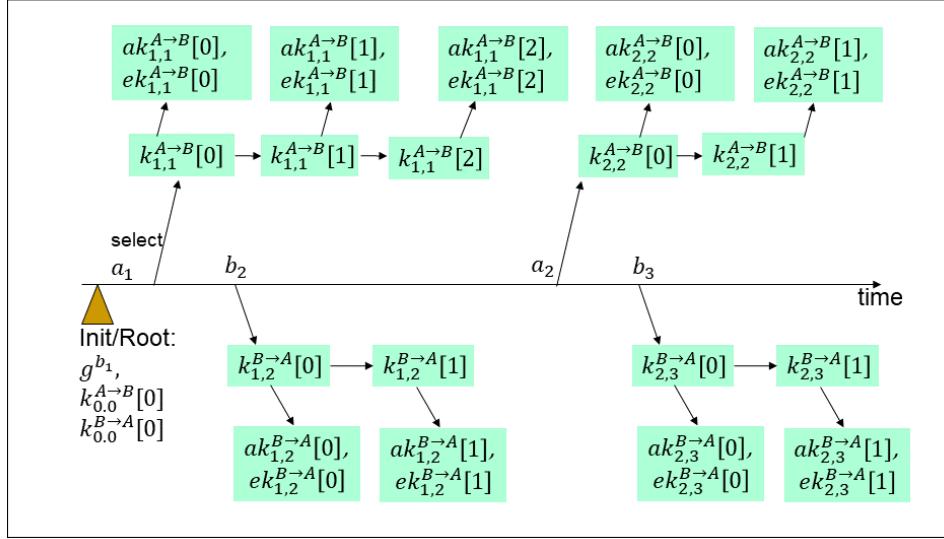


Figure 6.14: The Double-Ratchet Key-Exchange protocol

## 6.5 Security for Public Key Cryptosystems

We now return to public key cryptosystems (PKC), as illustrated in Figure 6.1. We defined PKC and their correctness properties in § 6.1. In this section we present an CPA-indistinguishability definition of security for PKCs. The definition is, essentially, an extension of the definition of CPA-IND security for stateless<sup>1</sup> shared-key cryptosystems (Definition 2.7).

**Definition 6.6** (PKC IND-CPA). *A public-key cryptosystem  $\langle KG, E, D \rangle$  is CPA-indistinguishable (IND-CPA), if for every PPT adversary  $ADV$  holds:*

$$\Pr [b = IndCPA_{ADV, \langle KG, E, D \rangle}(b, n)] \leq \frac{1}{2} + NEGL(n)$$

Where the probability is over coin tosses of  $ADV$ ,  $KG$  and  $E$ , the random choice  $b \xleftarrow{\$} \{0, 1\}$ , and where  $IndCPA_{ADV, \langle KG, E, D \rangle}(b, n)$  is:

```
{
  (e, d) ← KG(1n)
  (m0, m1, sADV) ← ADV( $e$ , ‘Choose’, 1n) s.t. |m0| = |m1|
  c* ← Ee(mb)
  b* = ADV( $e$ , ‘Guess’, sADV, c*)
  Return b*
}
```

<sup>1</sup>Recall from § 6.1, that we only consider stateless PKC.

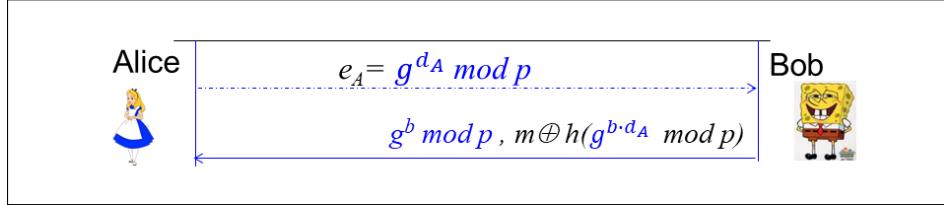


Figure 6.15: The DH Public Key Cryptosystem. The private decryption key  $d_A$  and random  $b$  are random integers less than  $p - 1$ . The text in blue is the same as the two flows of the DH protocol.

The main change between the CPA-IND definition for PKC (Definition 6.6, above) and the CPA-IND definition for shared-key cryptosystems (Definition 2.7), is that the adversary  $ADV$  is given the public key. Hence,  $ADV$  can encrypt at will, without the need to make encryption queries, as enabled by the oracle calls in Definition 2.7.

In the following two sections, we discuss three specific public key cryptosystems: DH and El-Gamal in § 6.6, and RSA in § 6.7.

## 6.6 Discrete-Log based public key cryptosystems: DH and El-Gamal

### 6.6.1 The DH PKC

In their seminal paper [40], Diffie and Hellman presented the concept of public-key cryptosystems - but did not present an implementation. On the other hand, they did present the DH key exchange protocol (Figure 6.9). We next show that a minor tweak allows us to turn the DH key exchange protocol into a PKC; we accordingly refer to this PKC as the DH PKC.

Fig. 6.15 presents the DH public key cryptosystem, by slightly changing the presentation of the DH key exchange protocol (Figure 6.9). Essentially, instead of Alice selecting random secret  $a$  and sending  $g^a \text{ mod } p$  to Bob in the first flow of the DH protocol, we now view  $g^{d_A} \text{ mod } p$  as Alice's public key  $e_A$ , and  $d_A$  is Alice's private key. To encrypt a message  $m$  for Alice, using her public key  $e_A = g^{d_A} \text{ mod } p$ , Bob essentially performs his role in the DH protocol, i.e., selects random random value  $b \in [2, p - 2]$ , and computes the ciphertext, which is a pair  $(g^b \text{ mod } p, m \oplus g^{d_A \cdot b} \text{ mod } p)$ . It should not be too difficult to write the formulas for the DH PKC.

**Exercise 6.7.** Write formulas for the three functions comprising the DH ENC:  $(KG^{DH}(n), E^{DH}, D^{DH})$ .

Note that in Figure 6.15 we use a keyless key-derivation function  $h$ ; the DH PKC can be implemented also using a keyed key-derivation function, with a uniformly-random key/salt  $s$ ; see § 6.3. We can completely avoid the use of a

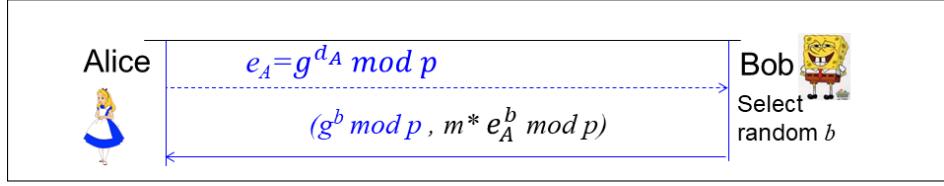


Figure 6.16: The El-Gamal Public-Key Encryption

key-derivation function, if we use a DDH group; that's essentially the El-Gamal PKC.

### 6.6.2 The El-Gamal PKC

The El-Gamal PKC is a minor variant of the DH PKC, which assumes that that the group used satisfy the DDH assumption (Definition 6.4).

As shown in Fig. 6.16, the El-Gamal encryption of plaintext  $m$ , denoted  $E_{e_A}^{EG}(m)$ , is computed as follows, using Alice's public key  $e_A = g^{d_A} \text{ mod } p$ :

$$b \xleftarrow{\$} [2, p - 1] ; x = g^b \text{ mod } p ; y = m \cdot e_A^b \text{ mod } p \quad (6.9)$$

El-Gamal decryption is therefore:

$$D_{d_A}(x, y) = \frac{y}{x^{d_A}} \quad (6.10)$$

The correctness property holds since:

$$\begin{aligned} D_{d_A}(x, y) &= \frac{y}{x^{d_A}} \\ &= \frac{m \cdot (g^{d_A} \text{ mod } p)^b \text{ mod } p}{(g^b \text{ mod } p)^{d_A} \text{ mod } p} \\ &= \frac{m \cdot g^{b \cdot d_A} \text{ mod } p}{g^{b \cdot d_A} \text{ mod } p} \\ &= m \end{aligned}$$

□

### 6.6.3 Homomorphic encryption and re-encryption.

The El-Gamal PKC is an *homomorphic encryption* scheme. Namely, given two messages  $m_1, m_2$ , let  $(x_1, y_1) = E_{e_A}^{EG}(m_1)$ ,  $(x_2, y_2) = E_{e_A}^{EG}(m_2)$  be their El-Gamal encryptions. Then we can compute the El-Gamal encryption of the multiplication of the two plaintexts,  $m_1 \cdot m_2 \text{ mod } p$ , as follows:

$$E_{e_A}^{EG}(m_1 \cdot m_2 \text{ mod } p) = (x_1 \cdot x_2, y_1 \cdot y_2) \text{ mod } p \quad (6.11)$$

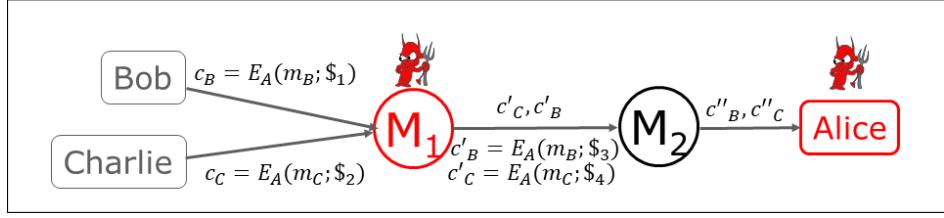


Figure 6.17: Re-encryption based mixnets for sender anonymity. Use universal reencryption for receiver anonymity.

**Exercise 6.8.** Show that Eq. 6.11 is correct encryption of  $m_1 \cdot m_2 \pmod{p}$ .

*Hint:* compute the decryption and show it returns  $m_1 \cdot m_2 \pmod{p}$ .  $\square$

Homomorphic encryption has multiple applications. One example is *re-encryption*: given one ciphertext  $c = E_{e_A}(m)$ , compute another ciphertext for the same message  $m$  - without knowing the message or the decryption key  $d_A$ . For example, many senders may send encrypted messages to a first *mix server*  $M_1$ , which shuffles and re-encrypt them, then forwards to a second mix  $M_2$ , which delivers them to the destination, who has the decryption key. This hides the identity of the sender from the destination, even if one of the two mixes collaborates with the destination. See Figure 6.17.

With homomorphic encryption, e.g., El-Gamal, we can compute the re-encryption of a message by encrypting the message 1, i.e., obtaining  $(x_1, y_1) = E_{e_A}(1)$ . We then use the homomorphic property to combine  $(x_1, y_1) = E_{e_A}(1)$  with  $(x, y) = E(m)$ , thereby computing new ciphertext  $(x', y')$  as  $x' = x \cdot x_1$ ,  $y' = y \cdot y_1$ . From the homomorphic property,  $m = D_{d_A}(x', y')$ , since  $m = m \cdot 1$ .

Notice that the above method of re-encryption of El-Gamal encryption, requires use of the public key with which the message was encrypted. However, by appending the encryption  $E_{e_A}(1)$  to each ciphertext, we can re-encrypt messages *without knowing the public key*. This is called *universal re-encryption* [61].

Note also that we only discussed re-encryptions which preserved the same decryption key. El-Gamal also allows *proxy re-encryption* [26], where a proxy is given a special key, denoted  $e_{A \leftarrow B}$  and computed by Alice, that allows the proxy to transform ciphertext encrypted to Alice,  $c = E_{e_A}(m)$ , into encryption of the same message with Bob's key,  $c' = E_{e_B}(m)$ .

We discussed homomorphism for multiplication of plaintexts; it is possible to also consider homomorphism for addition of plaintexts. An encryption scheme which is homomorphic for *both* addition and multiplication, is called *fully homomorphic*. Such schemes are subject for extensive research, and facilitate exciting applications, but are beyond our scope [109].

## 6.7 The RSA Public-Key Cryptosystem

In 1978, Rivest, Shamir and Adelman presented the first proposal for a public-key cryptosystem - as well as a digital signature scheme [96]. This beautiful scheme is usually referred to by the initials of the inventors, i.e., RSA; it was awarded the Turing award in 2002, and is still widely used. We would only cover here some basics details of RSA; a more in-depth study is recommended, by taking a course in cryptography and/or reading one of the many books on cryptography covering RSA in depth.

### 6.7.1 RSA key generation.

Key generation in RSA is more complex than for DH and El-Gamal; we first list the steps, and then explain them:

- Select a pair of large prime numbers  $p, q$ ; let  $n = p \cdot q$  and let  $\phi_n = (p - 1) \cdot (q - 1)$ .
- Select a value  $e$  which is co-prime to  $\phi_n$ , i.e.,  $\gcd(e, \phi_n) = 1$ .
- Compute  $d$  s.t.  $ed \equiv 1 \pmod{\phi_n}$ .
- The public key is  $(e, n)$  and the private key is  $(d, n)$ .

Selecting  $e$  to be co-prime to  $\phi_n$  is necessary - and sufficient - to ensure that  $e$  has a multiplicative inverse  $d$  in the group  $\pmod{\phi_n}$ . To find the inverse  $d$ , we can use the extended Euclidean algorithm. This algorithm efficiently finds numbers  $d, x$  s.t.  $e \cdot d + \phi_n \cdot x = \gcd(e, \phi_n) = 1$ ; namely,  $ed \equiv 1 \pmod{\phi_n}$ .

The public key of RSA is  $\langle e, n \rangle$  and the private key is  $\langle d, n \rangle$ , since the modulus  $n$  is required for both encryption and decryption. However, we - and others - often abuse notation and refer to the keys simply as  $e$  and  $d$ .

Notice that exactly the same process is used for key generation for RSA signature schemes (discussed below), except that we denote the public verification key by  $v$  (instead of public encryption key  $e$ ) and the private signing key by  $s$  (instead of private decryption key  $d$ ).

### 6.7.2 Textbook RSA: encryption, decryption and correctness.

We first present *textbook RSA*, which is a bit simpler than ‘real’ RSA encryption, presented later. The textbook RSA encryption  $c = E_e(m)$  and decryption  $m = D_d(c)$  processes are simple and similar:

$$E_e^{RSA}(m) = m^e \pmod{n} \quad (6.12)$$

$$D_d^{RSA}(c) = c^d \pmod{n} \quad (6.13)$$

Where the message  $m$  is encoded as a positive integer, and limited to  $m < n$ , ensuring that  $m = m \pmod{n}$ . To handle the common case where the plaintext

is longer, we use *hybrid encryption*, i.e., use the public key encryption to encrypt a shared key and then use the shared key to efficiently encrypt the long message; see §6.1.6.

The reason this method is referred to as *textbook RSA*, is that, to ensure security, we usually pre-process messages before applying encryption, in a (key-less) process usually referred to as *padding*. We explain the need for padding in §6.7.3, and OAEP, the commonly-used padding scheme, in §6.7.4.

### Correctness of textbook RSA

Let us explain why these processes ensure correctness of textbook RSA, i.e.,  $m = D_d^{RSA}(E_e^{RSA}(m)) = (m^e \bmod n)^d \bmod n$ . This is based on a fundamental result from number theory: *Euler's Theorem*.

Before we present Euler's theorem, we introduce the *Euler's function*  $\phi(n)$ . The value of  $\phi(n)$  is defined as the number of positive integers which are less than  $n$  and co-prime to  $n$ , i.e.,

$$\phi(n) \equiv |\{i \in \mathbb{N} | i < n \wedge \gcd(i, n) = 1\}| \quad (6.14)$$

Note that for any primes  $p, q$  holds  $\phi(p) = p - 1$ ,  $\phi(q) = q - 1$ , and  $\phi(p \cdot q) = (p - 1)(q - 1)$ . This is the reason for us using  $\phi_n = \phi(n) = \phi(p \cdot q) = (p - 1)(q - 1)$  in the RSA key generation process.

We now present Euler's theorem:

**Theorem 6.1** (Euler's theorem). *For any co-prime integers  $m, n$  hold  $m^{\phi(n)} = 1 \bmod n$ .*

We will not prove the theorem; interested readers can find proofs in many texts. However, let us use the theorem, to explain RSA's correctness, i.e., why  $D_d(E_e(m)) = m$ .

Our explanation is a bit simplified, since the theorem holds only provided that  $m$  and  $n$  are co-primes, and this may not hold; the message  $m$  may be any positive integer smaller than  $n$ . Most of these - specifically,  $\phi(n)$ , which we know to be  $(p - 1) \cdot (q - 1) = n - q - p + 1$  - are co-prime to  $n$ . However, there are  $p + q - 2$  values which are *not* co-prime to  $n$ . Correctness holds also for these values, but a slightly more elaborate argument is required, based on the Chinese Remainder Theorem; we omit this here, and provide only the argument for the case that  $m$  and  $n$  are co-prime ( $\gcd(m, n) = 1$ ).

Recall that  $ed = 1 \bmod \phi(n)$ , i.e., for some integer  $i$  holds  $ed = 1 + i \cdot \phi(n)$ . Hence:

$$\begin{aligned} m^{ed} \bmod n &= m^{1+i \cdot \phi(n)} \bmod n \\ &= m \cdot (m^{\phi(n)})^i \bmod n \end{aligned}$$

Recall Eq. (2.7) :  $a^b \bmod n = (a \bmod n)^b \bmod n$ . Hence, by substituting  $m^{\phi(n)} = 1 \bmod n$  (Euler's theorem), we have

$$m^{ed} \bmod n = m \cdot (m^{\phi(n)} \bmod n)^i \bmod n = m \cdot 1^i = m$$

Showing the correctness of textbook RSA (eqs. (6.12), (6.12)).

### Choosing $e$ to improve efficiency

The public exponent  $e$  is not secret, and, so far, it was only required to be co-prime to  $\phi(n)$ . This motivates choosing  $e$  that will improve efficiency - usually, to make encryption faster.

In particular, choosing  $e = 3$  implies that encryption - i.e., computing  $m^e \bmod n$  - requires only two multiplications, i.e., is very efficient (compared to exponentiation by larger number). Note, however, that there are several concerns with such extremely-small  $e$ ; in particular, if  $m$  is also small, in particular, if  $c = m^e < n$  (without reducing  $\bmod n$ ), then we can efficiently decrypt by taking the  $e$ -th root:  $c^{1/e} = (m^e)^{1/e} = m$ . This particular concern is relatively easily addressed by *padding*, as discussed below; but there are several additional attacks on RSA with very-low exponent  $e$ , in particular in the case where sending the same message, or ‘highly related’ messages, to multiple recipients. These attacks motivate the use of padding to break any possible relationships between messages (§6.7.4, as well as the choice of slightly larger  $e$ , such as 17 or even  $2^{16} + 1 = 65537$ , primes that require only 5 or 17 multiplications, respectively; see next exercise.

**Exercise 6.9.** Given integer  $m$ , show how to compute  $m^{17}$  in only five multiplications, and  $m^{2^{16}+1}$  in only 17 multiplications.

*Hint:* use the following idea: compute  $m^8$  with three multiplications by  $m^8 = ((m^2)^2)^2$ . □

### 6.7.3 The RSA assumption and security

Now that we have seen that RSA is a PKC and ensures correctness, it is time to discuss the security of RSA.

Intuitively, the security of RSA is based on the following assumption, usually referred to as the RSA assumption.

**Definition 6.7** (RSA assumption). Choose  $n, e$  as explained above, i.e.,  $n = pq$  for  $p, q$  chosen as random  $l$ -bit primes, and  $e$  is co-prime to  $\phi_n$ . The RSA assumption is that for any efficient (PPT) algorithm  $A$  and constant  $c$ , and for sufficiently large  $l$  (primes), holds:

$$\Pr [A((e, n), m^e \bmod n) = m] < l^c \quad (6.15)$$

Where  $m$  is chosen randomly  $m \xleftarrow{\$} [1, n - 1]$ .

The RSA assumption is also referred to sometimes as the *RSA trapdoor one-way permutation assumption*. The ‘trapdoor’ refers to the fact that  $d$  is a ‘trapdoor’ that allows inversion of RSA; the ‘one-way’ refers to the fact that computing RSA (given public key  $(e, n)$ ) is easy, but inverting is ‘hard’; and the

‘permutation’ is due to RSA being a permutation (and in particular, invertible). See also the related concept of *one-way functions* in §4.4.

One obvious question is the relation between the RSA assumption and the assumption that factoring is hard. Assume that some adversary  $A_F$  can efficiently factor large numbers, specifically, the modulus  $n$  (which is part of the RSA public key). Hence,  $A_F$  can factor  $n$ , find  $q$  and  $p$ , compute  $\phi_n$  and proceed to compute the decryption key  $d$ , given the public key  $\langle e, n \rangle$ , just like done in RSA key generation. We therefore conclude that if factoring is easy, i.e., exists such adversary  $A$ , then the RSA assumption cannot hold (and RSA is insecure).

### RSA Security.

Another question is the relationship between the RSA assumption and the security of RSA as used as a public key cryptosystem (PKC). Trivially, if the RSA assumption is false, then RSA is not a secure cryptosystem - definitely not the ‘textbook RSA’ defined above.

However, what we really care about is the reverse direction, i.e., having a cryptosystem which is secure assuming that the RSA assumption holds. This does *not* hold for textbook RSA. There are few problems we should address, to build a secure PKC from RSA:

1. If we apply textbook RSA, i.e., as in Eq. (6.12), then it cannot be IND-CPA secure, since it is completely deterministic. Some randomness must be added to the process. Namely, the IND-CPA experiment allows the attacker to know (even to choose) two specific messages  $m_0, m_1$ . The attacker is then given encryption  $c^* = e^{m_b} \bmod n$  for random bit  $b$ , and ‘wins’ if it finds  $b$ . However, the attacker can compute  $c_0 = e^{m_0} \bmod n$  and  $c_1 = e^{m_1} \bmod n$ , and compare to  $c^*$  to immediately find  $b$  and ‘win’. Obviously, randomization is necessary.
2. The RSA assumption, as stated, does not rule out a potential exposure of partial information about the plaintext, e.g., a particular bit. Note that the  $\log n$  least-significant bits were shown to be secure [1].
3. RSA is vulnerable to *chosen ciphertext attack (CCA)*. Specifically, the attacker wants to decrypt ciphertext  $c = m^e \bmod n$  - without making a decryption query with  $c$  (as that would not be considered ‘winning’). Instead, attacker asks for decryption of a different ciphertext:  $\hat{c} = c \cdot \hat{m}^e \bmod n$ . This returns  $m \cdot \hat{m}$ , from which the attacker obtains the original plaintext  $m$ .
4. Furthermore, textbook RSA - as well as version 1.5 and earlier of the PKCS 1 standard - were shown vulnerable to a practical *feedback-only CCA attack* by Bleichenbacher [27].

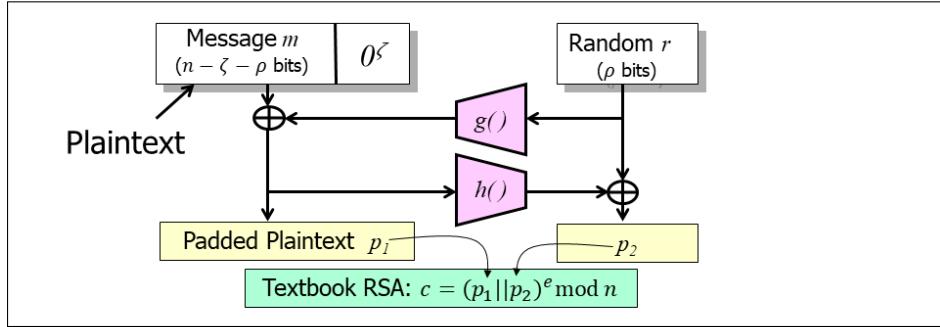


Figure 6.18: RSA with OAEP (Optimal Asymmetric Encryption Padding).

5. As additional motivation, recall the attacks mentioned above against RSA when using a small exponent (e.g.,  $e = 3$ ) and sending identical or related messages to multiple recipients.

The standard way to deal with these concerns is to apply a randomized ‘padding’ function to the plaintext, and only then apply the ‘textbook’ RSA function (eq. 6.12), as we discuss next.

#### 6.7.4 RSA with OAEP (Optimal Asymmetric Encryption Padding)

So far, we discussed textbook RSA (§6.7.2); however, this version is rarely used in practice, due to known attacks and vulnerabilities, listed above. Bellare and Rogaway presented in [17] the OAEP (Optimal Asymmetric Encryption Padding) scheme, which is widely used for padding of messages before applying the ‘textbook RSA’ exponentiation (Eq. (6.12)).

The goal of OAEP is to prevent different attack, including CCA attacks; to prevent CCA, OAEP includes a verification mechanism, which allows the recipient to confirm that the ciphertext is result of legitimate encryption rather than manipulation of intercepted (different) ciphertext.

[TBC]

The security of OAEP is analyzed using the Random Oracle Methodology (ROM), see §4.6.

### 6.8 Public key signature schemes

We now discuss the third type of public-key cryptographic schemes: *signature schemes*, introduced in subsection 6.1.2. Signature schemes consist of three efficient algorithms ( $KG, S, V$ ), illustrated in Figure 6.2:

**Key-generation  $KG$** , a randomized algorithm, whose input is the key length  $l$ , and which outputs the private signing key  $s$  and the public validation key  $v$ , each of length  $l$  bits.

**Signing**  $S$  , a (deterministic or randomized) algorithm, whose inputs are a message  $m$  and the signing key  $v$ , and whose output is a signature  $\sigma$ .

**Validation**  $V$  , a deterministic algorithm, whose inputs are a message  $m$ , signature  $\sigma$  and validation key  $v$ , and which outputs an indication whether this is a valid signature for this message or not.

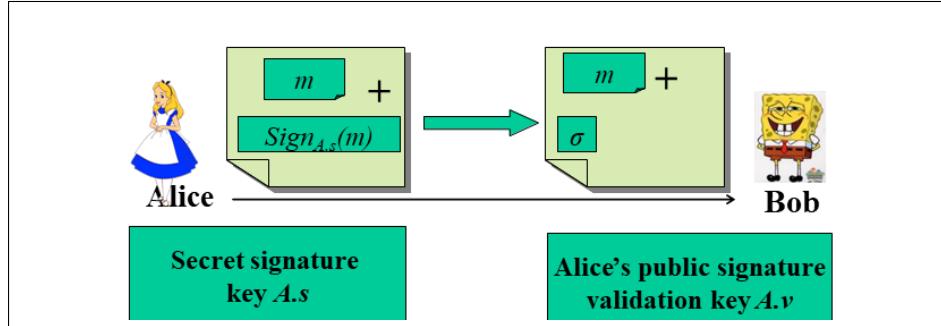


Figure 6.19: Public key signature and validation processes.

Figure 6.19 illustrates the process of signing a message (by Alice) and validation of the signature (by Bob). We denote Alice's keys by  $A.s$  (for the private signing key) and  $A.v$  (for the public validation key); note that this figure assumes that Bob knows  $A.v$  - we later explain how signatures also facilitate distribution of public keys such as  $A.v$ .

Signature schemes have two critical properties, which make them a critical enabler to modern cryptographic systems. First, they *facilitate secure remote exchange in the MitM adversary model*; second, they facilitate *non-repudiation*. We begin by briefly discussing these two properties.

**Signatures facilitate secure remote exchange of information in the MitM adversary model.** Public key cryptosystems and key-exchange protocols, facilitate establishing of private communication and shared key between two remote parties, using only public information (keys). However, this still leaves the question of authenticity of the public information (keys).

If the adversary is limited in its abilities to interfere with the communication between the parties, then it may be trivial to ensure the authenticity of the information received from the peer. In particular, many works assume that the adversary is passive, i.e., can only eavesdrop to messages; this is also the basic model for the DH key exchange protocol. In this case, it suffices to simply send the public key (or other public value).

Some designs assume that the adversary is inactive or passive during the *initial* exchange, and use this exchange information such as keys between the two parties. This is called the *trust on first use (TOFU)* adversary model.

In other cases, the attacker may inject fake messages, but cannot eavesdrop on messages sent between the parties; in this case, parties may easily authenticate a message from a peer, by previously sending a challenge to the peer, which the peer includes in the message.

However, all these methods fail against the stronger *monster-in-middle (MitM)* adversary, who can modify and inject messages as well as eavesdrop on messages. To ensure security against such attacker, we must use strong, cryptographic authentication mechanisms. One option is to use message authentication codes, however, this requires the parties to share a secret key in advance ; if that's the case, the parties could use this shared key to establish secure communication directly.

Signature schemes provide a solution to this dilemma. Namely, a party receiving signed information from a remote peer, can validate that information, using only the public signature-validation key of the signer. Furthermore, signatures also allow the party performing the signature-validation, to first validate the public signature-validation key, even when it is delivered by an insecure channel which is subject to a MitM attack, such as email. This solution is called *public key certificates*.

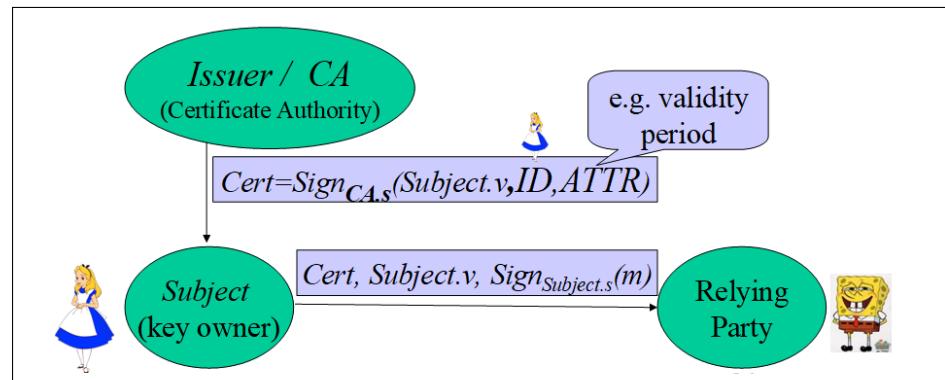


Figure 6.20: Public key certificate issuing and usage processes.

As illustrated in Fig. 6.20, a public key certificate is a signature by an entity called the *issuer* or *certificate authority (CA)*, over the public key of the *subject*, e.g., Alice. In addition to the public key of the subject, *subject.v*, the signed information in the certificate contains attributes such as the validity period, and, usually, an identifier and/or name for the subject (Alice).

Once Alice receives her signed certificate *Cert*, she can deliver it to the *relying party* (e.g., Bob), possibly via insecure channels such as email. This allows the relying party (Bob) to use Alice's public key, i.e., rely on it, e.g., to validate Alice's signature over a message *m*, as shown in Fig. 6.20. Note that this requires Bob to trust this CA and to have its validation key, *CA.v*.

This discussion of certificates is very basic; more details will be provided in chapter 8, which discusses public-key infrastructure and the TLS/SSL protocol.

**Signatures facilitate non-repudiation.** The other unique property of digital signature schemes is that they facilitate non-repudiation. Namely, upon receiving a properly signed document, together with a signature by some well-known authority establishing the public signature-validation key, the recipient is assured that she can convince other parties that she received the document signed properly. This is very useful property. This property does not hold for message-authentication codes (MAC schemes), where a recipient can validate an incoming message has the correct MAC code, but cannot prove this to another party - in particular, since the recipient is able to compute herself the MAC code for arbitrary messages.

**Designs of signature schemes.** We next briefly present two designs of signature schemes: RSA-based signature, and Discrete-Log based signatures (focusing on a variant known as El-Gamal signatures). Both designs also make use of cryptogrpthic hash functions  $h(\cdot)$ , which we discuss in the next chapter.

### 6.8.1 RSA-based signatures

RSA signatures were proposed in the seminar RSA paper [96], and are based on the RSA assumption, with exactly the same key-generation process as for the RSA PKC. The only difference in key generation, is that for signature schemes, the public key is denoted  $v$  (as it is used for *validation*), and the private key is denoted  $s$  (as it is used for signing).

There are two main variants of RSA signatures: signature with message recovery, and signature with appendix. We begin with signatures with appendix, as in practice, almost all applications of RSA signatures are with appendix; in fact, we present (later) signatures with message recovery mainly since they are often mentioned, and almost as often, a cause for confusion.

#### RSA signature with appendix.

In the (theoretically-possible) case that input messages are very short, and can be encoded as a positive integer which is less than  $n$ , we can sign using RSA by applying the RSA exponentiation directly to the message, resulting in the signature  $\sigma$ . In this case, the signature and validation operations are defined as:

$$\begin{aligned} S_s^{RSA}(m) &= (m^s \mod n, m) \\ V_v^{RSA}(\sigma, m) &= \{m \text{ if } m = \sigma^v \mod n, \text{'error' otherwise}\} \end{aligned}$$

Above,  $s$  is the private signature key, and  $v$  is the public validation key. The keys are generated using the RSA key generation process; see subsection 6.7.1

In practice, as discussed in §4.2.4, input messages are of variable length - and rarely shorter than modulus. Hence, real signatures apply the *Hash-then-Sign (HtS)* paradigm (Definition 4.3), using some cryptographic hash function

$h$ , whose range is an integer  $\mod n$ . Applied to the RSA FIL signature as defined above, we have the signature scheme  $(S_s^{RSA,h}, V_v^{RSA,h})$ , defined as follows:

$$\begin{aligned} S_s^{RSA,h}(m) &= ([h(m)]^s \mod n, m) \\ V_v^{RSA,h}(\sigma, m) &= \{m \text{ if } h(m) = \sigma^v \mod n, \text{ error otherwise}\} \end{aligned}$$

The resulting signature scheme is secure, if  $h$  is a CRHF; see §4.2.

This signature scheme is called *signature with appendix* since it requires transmission of both original message and its signature. This is in contrast to a rarely used variant of RSA signatures which is called *signature with message recovery*, which we explain next. ‘Signature with recovery’ is rarely, if ever, applied in practice; we describe it since there is a lot of reference to it in literature, and in fact, this method causes quite a lot of confusion among practitioners. Hopefully the text below will help to avoid such confusion.

### RSA signature with message recovery.

RSA signatures with message recovery have the cute property, that they only require transmission of one  $\mod n$  integer - the signature; the message itself does not need to be sent, as it is *recovered* from the signature. This cute property would result in a small savings of bandwidth, compared to signature with appendix, when both methods are applicable. However, as we explain below, this method is rarely applicable; furthermore, it is cause for frequent confusion.

RSA signatures with message recovery require the use of an invertible *padding* function  $R(\cdot)$  which is applied to the messages to be signed. The main goal of  $R$  is to ensure sufficient, known *redundancy* (in  $R(m)$ ; this is why we denote it by  $R$ ). This redundancy, applied to the message before the public key signature operation, should make it unlikely that a random value would appear as a valid signature.

The output of  $R(m)$  is used as input to the RSA exponentiation; to ensure recovery of  $m$ , the value of  $R(m)$  must be less than the RSA modulu  $n$ . Note that this implies that  $m$  is even more restricted - since  $R(m)$  must contain all of  $m$ , plus the redundancy.

Once  $R$  is defined, the signature and validation operations for RSA with Message Recovery (RSAwMR) would be:

$$S_s^{RSAwMR}(m) = [R(m)]^s \mod n \quad (6.16)$$

$$V_v^{RSAwMR}(x) = \{R^{-1}(x^v \mod n) \text{ if defined, else error}\} \quad (6.17)$$

For validation to be meaningful, there should be only a tiny subset of the integers  $x$  s.t.  $x^v \mod n$  would be in the range of  $R$ , i.e., the result of the mapping of some message  $m$ . Since there are at most  $n$  values of  $x^v \mod n$  to begin with, this means that the range of  $R$ , i.e., the set of legitimate messages,

must be tiny in comparison with  $n$  - which means that the message space should be *really* tiny.

In reality, messages being signed are almost always much longer than the tiny message space available for signatures with message recovery. Hence, the use of this method is almost non-existent. In fact, our description of signature schemes (Figure 6.2) assumed that the message is sent along with its signature, i.e., our definition did not even take into consideration schemes like this, that avoid sending the original message entirely.

Note that RSA signatures with message recovery are often a cause of *confusion*, due to their syntactic similarity to RSA encryption. Namely, you may come across people referring to the use of ‘RSA encryption with the private key’ as a method to authenticate or sign messages. What these people really mean is to the use of RSA signatures with message recovery. We caution to avoid such confusing use of terminology; RSA signatures are usually used with appendix, but even in the rare cases of using RSA signatures with message recovery, *RSA signing is not the same as encryption with the private key!*

### 6.8.2 Discrete-Log based signatures

[TBD]

## 6.9 Additional Exercises

**Exercise 6.10.** It is proposed that to protect the DH protocol against an imposter, we add an additional ‘confirmation’ exchange after the protocol terminated with a shared key  $k = h(g^{ab} \bmod p)$ . In this confirmation, Alice will send to Bob  $MAC_k(g^b)$  and Bob will respond with  $MAC_k(g^a)$ . Show the message-flow of an attack, showing how an attacker (Monster) can impersonate as Alice (or Bob). The attacker has ‘MitM capabilities’, i.e., it can intercept messages (sent by either Alice or Bob) and inject fake messages (incorrectly identifying itself as Alice or Bob).

**Exercise 6.11.** Suppose that an efficient algorithm to find discrete log is found, so that the DH protocol becomes insecure; however, some public-key cryptosystem  $(\mathcal{G}, \mathcal{E}, \mathcal{D})$  is still considered secure, consisting of algorithms for, respectively, key-generation, encryption and decryption.

1. Design a key-agreement protocol which is secure against an eavesdropping adversary, assuming that  $(\mathcal{G}, \mathcal{E}, \mathcal{D})$  is secure (as a replacement to DH).
2. Explain which benefits the use of your protocol may provide, compared with simple use of the cryptosystem  $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ , to protect the confidentiality of messages sent between Alice and Bob against a powerful MitM adversary. Assume Alice and Bob do have known public keys.

**Exercise 6.12.** Assume that there is an efficient (PPT) attacker  $\mathcal{A}$  that can find a specific bit in  $g^{ab} \bmod p$ , given only  $g^a \bmod p$  and  $g^b \bmod p$ . Show that the DDH assumption does not hold for this group, i.e., that there is an efficient (PPT) attacker  $\mathcal{A}$  that can distinguish, with significant advantage over random guess, between  $g^{ab} \bmod p$  and between  $g^x$  for  $x$  taken randomly from  $[1, \dots, p-1]$ .

**Exercise 6.13.** It is frequently proposed to use a PRF as a Key Derivation Function (KDF), e.g., to extract a pseudo-random key  $k' = \text{PRF}_k(g^{ab} \bmod p)$  from the DH exchanged value  $g^{ab} \bmod p$ , where  $k$  is a uniform random key (known to attacker). Show a counterexample: a function  $f$  which is a secure PRF, yet insecure if used as a KDF. For your construction, you may use a secure PRF  $f'$ .

**Exercise 6.14** (How not to ensure resilient key exchange). Fig. 6.21 illustrates a slightly different protocol for using authenticated DH to ensure resilient key exchange. Present a sequence diagram showing that this protocol is not secure.

*Hint:* show how an attacker is able to impersonate as Alice, without knowing any of Alice’s previous keys; at the end of the handshake, Bob will believe it has exchanged key  $k$  with Alice, but the key was actually exchanged with the attacker.  $\square$

**Exercise 6.15** (Insecure variant of authenticated DH). The protocol in Fig. 6.22 is a simplified version of the authenticated DH protocol of Fig. 6.10.

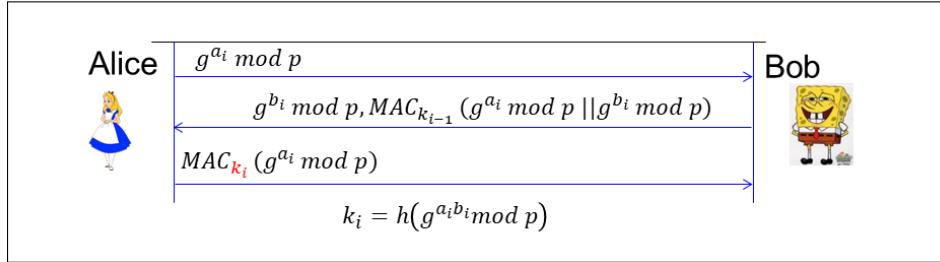


Figure 6.21: How *not* to ensure resiliency - illustration for Ex. 6.14

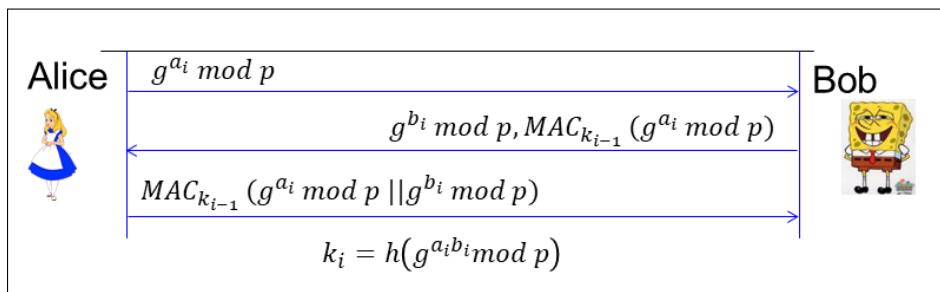


Figure 6.22: Insecure variant of the authenticated DH protocol, studied in Exercise 6.15

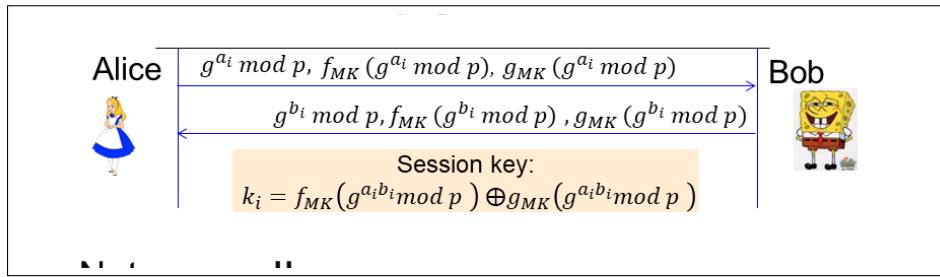


Figure 6.23: Insecure ‘robust-combiner’ authenticated DH protocol, studied in Exercise 6.16

1. Show a sequence diagram for an attack showing that this variant is insecure. Hint: your attack may take advantage of the fact that there is no validation of the incoming flows, except for validation of the MAC values.
2. Explain why this attack does not apply to the ratchet protocol (Fig. 6.12).

**Exercise 6.16.** The protocol in Fig. 6.23 is an (incorrect) attempt at a robust-combiner authenticated DH protocol.

1. Show a sequence diagram for an attack showing that this variant is insecure.
2. Show a simple fix that achieves the goal (robust combiner authenticated DH protocol).

**Exercise 6.17.** Alice and Bob use low-energy devices to communicate. To ensure secrecy, they run, daily, the Sync-DH-Ratchet protocol (Fig. 6.11), but want to further improve security, by changing keys every hour, but avoiding additional exponentiations. Let  $k_i^j$  denote the key they share after the  $j^{\text{th}}$  hour of the  $i^{\text{th}}$  day, where  $k_i^0 = k_i$  (the key exchanged in the ‘daily exchange’ of Fig. 6.11).

1. Show how Alice and Bob should set their hourly shared secret key  $k_i^j$ .
2. Define the exact security benefit achieved by your solution.

**Exercise 6.18.** Assume it takes 10 seconds for any message to pass between Alice and Bob.

1. Assume that both Alice and Bob initiate the ratchet protocol (Fig. 6.12) every 30 seconds. Draw a sequence diagram showing the exchange of messages between time 0 and time 60seconds; mark the keys used by each of the two parties to authenticate messages sent and to verify messages received.
2. Repeat, if Bob’s clock is 5 seconds late.
3. Repeat, when using the ‘double-ratchet’ variant, where both parties perform the PRF derivation whenever 10 seconds passed since last changing the key.

**Exercise 6.19.** In the ratchet protocol, as described (Fig. 6.12), the parties derive symmetric keys  $k_{i,j}$  and use them to authenticate data (application) messages they exchange between them, as well as the first message of the next handshake.

1. Assume a chosen-message attacker model, i.e., the attacker may define arbitrary data (application) messages to be sent from Alice to Bob and vice versa at any given time, and ‘wins’ if a party accepts a message never sent by its peer (i.e., that message passes validation successfully). Show that, as described, the protocol is insecure in this model.
2. Propose a simple, efficient and secure way to avoid this vulnerability, by only changing how the protocol is used - without changing the protocol itself.

**Exercise 6.20.** The DH protocol, as well as the ratchet protocol (as described in Fig. 6.12), are designed for communication between only two parties.

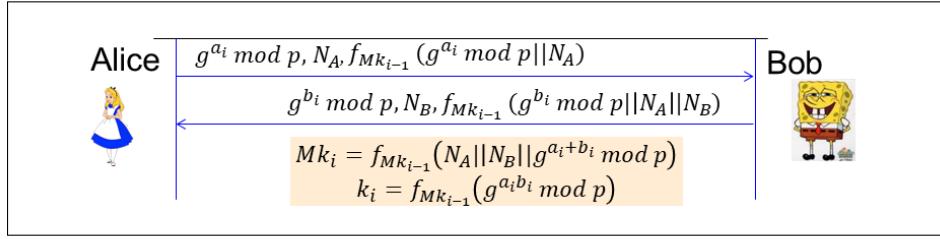


Figure 6.24: Authenticated DH with changing Master Keys (Ex. 6.22)

1. Extend DH to support key agreement among three parties.
2. Similarly extend the ratchet protocol.

**Exercise 6.21.** In the double-ratchet protocol, as described in class, at the beginning of ‘turn’  $t$  in a party, the party uses the ‘current key’  $k_{i,j}^t$  to derive two keys:  $k_{i,j}^{t+1}$ , to be used at the next ‘turn’, and  $\widehat{k_{i,j}^{t+1}}$ , used to authenticate and encrypt messages sent and received between the peers.

1. Explain why  $\widehat{k_{i,j}^{t+1}}$  is used to authenticate and encrypt, rather than using  $k_{i,j}^{t+1}$ .
2. Explain how to use  $\widehat{k_{i,j}^{t+1}}$  (to authenticate and encrypt messages sent between the peers).

**Exercise 6.22** (Authenticated DH with changing Master Keys). Figure 6.24 shows a variant of the Authenticated DH protocol, where the master key is changing (as indicated). Assume that this protocol is run daily, from day  $i = 1$ , and where  $Mk_0$  is a randomly-chosen secret initial master key, shared between Alice and Bob; messages on day  $i$  are encrypted using session key  $k_i$ . An attacker can eavesdrops on the communication between the parties on all days, and on days 3, 6, 9, . . . it can also spoof messages (send messages impersonating as either Alice or Bob), and act as Monster-in-the-Middle (MitM). On the fifth day ( $i = 5$ ), the attacker is also given the initial master key  $Mk_0$ .

- What are the days that the attacker can decrypt (know) their messages, upon day ten?
- Show a sequence diagram of the attack, and list every calculation done by the attacker. For every value used by the attacker, explain why/how that value known to the attacker.

**Exercise 6.23** (Insecure variant of Ratchet DH). Figure 6.25 shows a vulnerable variant of the Ratchet DH protocol, using a (secure) pseudorandom function  $f$  to derive the session key. Assume that this protocol is run daily, from day  $i = 1$ , and where  $k_0$  is a randomly-chosen secret initial key, shared between

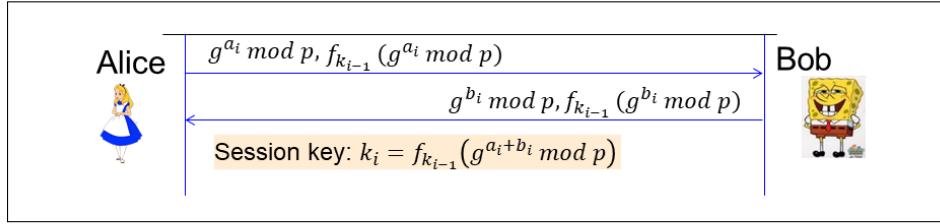


Figure 6.25: Insecure variant of Ratchet DH Protocol (Ex. 6.23)

*Alice and Bob; messages on day  $i$  are encrypted using key  $k_i$ . An attacker can eavesdrops on the communication between the parties on all days, and on days 3, 6, 9, ... it can also spoof messages (send messages impersonating as either Alice or Bob), and act as Monster-in-the-Middle (MitM). On the fifth day ( $i = 5$ ), the attacker is also given the initial key  $k_0$ .*

- *On which day can attacker first decrypt messages?* Answer:
- *On the day you specified, what are the days that the attacker can decrypt messages of?*
- *Explain the attack , including a sequence diagram if relevant. Include every calculation done by the attacker.*

**Exercise 6.24** (Improving security of Sync-Ratchet). *Alice and Bob use low-energy devices to communicate. To ensure secrecy, they run, daily, the Sync-Ratchet protocol (Fig. 5.12), but want to further improve security, by changing keys every hour, but avoiding additional exponentiations. Let  $k_i^j$  denote the key they share after the  $j^{\text{th}}$  hour of the  $i^{\text{th}}$  day, where  $k_i^0 = k_i$  (the key exchanged in the ‘daily exchange’ of Fig. 5.12).*

1. *Show how Alice and Bob should set their hourly shared secret key  $k_i^j = \underline{\hspace{1cm}}$ .*
2. *Define the exact security benefit achieved by your solution.*

*Hint:* Compute  $k_i^j = \text{PRF}_{k_i^{j-1}}(\text{'next'})$ . □

**Exercise 6.25.** *We saw that El-Gamal encryption (Eq. (6.9) may be re-randomized, using the recipient’s public key, and mentioned that this may be extended into an encryption scheme which is univerally re-randomizable, i.e. where re-randomization does not require the recipient’s public key. Design such encryption scheme. Hint: begin with El-Gamal encryption, and use as part of the ciphertext, the result of encrypting the number 1.*

**Exercise 6.26.** *Design a simple and efficient protocol allowing a set of users  $\{1, \dots, n\}$  to exchange messages securely among them, i.e., ensuring confidentiality, authenticity and integrity of the messages. Your design may assume that users know the public keys of each other; furthermore, you may assume*

that each user  $i$  has two pairs of public-private keys: an encryption-decryption key-pair  $(e(i), d(i))$  and a signing-verifying key-pair  $(s(i), v(i))$ . Assume that the public keys  $e(i), v(i)$  of every user  $i$ , are known to all users. Your description should be very clear, well-defined and concise, but can be high-level, e.g., use  $(a, b, c) \leftarrow \alpha$  to denote ‘splitting’ a tuple  $\alpha = (a, b, c)$  into its components. Please read all three below parts before answering; the third part is an additional requirement, that you may meet already in answer of parts 1 and 2, this may save time for answering part 3.

1. A send process that receives message  $m$ , receiver identifier  $i$ , a sender identifier  $j$  and the sender’s private keys  $d(j), s(j)$ , and produces a ‘ciphertext’  $c$  to be sent to  $i$ , i.e.,  $c = \text{send}_{j,d(j),s(j)}(m, i)$ .
2. A receive process that receives ‘ciphertext’  $c$ , receiver identifier  $i$ , and the receiver’s private keys  $d(i), s(i)$ , and produces message  $m$  and sender identifier  $j$ , if  $c$  was output of send executed by  $j$  on message  $m$  with receiver  $i$ , and an error indicator  $\perp$  otherwise.
3. Another goal is to prevent recipients from proving to somebody else that the sender sent the message (as when showing signed message). Is this provided by your solutions? If not, present processes for sending and receiving to support this goal.

**Exercise 6.27.** The RSA algorithm calls for selecting  $e$  and then computing  $d$  to be its inverse ( $\mod \phi(n)$ ). Explain how the key owner can efficiently compute  $d$ , and why an attacker cannot do the same.

**Exercise 6.28** (Tiny-message attack on textbook RSA). We discussed that RSA should always be used with appropriate padding, and that ‘textbook RSA’ (no padding) is insecure, in particular, is not probabilistic so definitely does not ensure indistinguishability.

1. Show that textbook RSA may be completely decipherable, if the message length is less than  $|n|/e$ . (This is mostly relevant for  $e = 3$ .)
2. Show that textbook RSA may be completely decipherable, if there is only a limited set of possible messages.
3. Show that textbook RSA may be completely decipherable, if the message length is less than  $|n|/e$ , except for a limited set of additional (longer) possible messages.

**Exercise 6.29.** Consider the use of textbook RSA for encryption (no padding). Show that it is insecure against a chosen-ciphertext attack.

**Exercise 6.30.** Consider a variation of RSA which uses the same modulus  $N$  for multiple users, where each user, say Alice, is given its key-pair  $(A.e, A.d)$  by a trusted authority (which knows the factoring of  $N$  and hence  $\phi(N)$ ). Show that one user, say Mal, given his keys  $(M.e, M.d)$  and the public key of other

users say  $A.e$ , can compute  $A.d$ . Note: recall that each user's private key is the inverse of the public key ( $\pmod{\phi(n)}$ , e.g.,  $M.e = M.d^{-1} \pmod{\phi(n)}$ ).

**Exercise 6.31.** Public-key algorithms often use term ‘public key’ to refer to only one component of the public key. For example, with RSA, people often refer to  $e$  as the public key, although the actual RSA public key consists of the pair  $(e, n)$ , i.e., also includes the modulus  $n$ .

Consider an application which receives an RSA signature  $(e_A, n)$ , where  $e_A$  is the same as in the public key  $(e_A, n_A)$  of user Alice, but  $n \neq n_A$ ; however, the application still concludes that this is a valid signature by Alice. Show how this allows an attacker to trick the recipient into believing - incorrectly - that an incoming message (sent by the attacker) was signed by Alice.

Note :similar situation exists with other public key algorithms, e.g., elliptic curves, where the public key consists of a specification of a curve and of a particular ‘public point’ on the curve, but often people refer only to the point as if it is the (entire) public key. In particular, this led to the ‘Curveball’ vulnerability in the Windows certificate-validation mechanism [?], which was due to validation of only the ‘public point’ and use of the curve selected by the attacker.

**Exercise 6.32.** You are given textbook-RSA ciphertext  $c = 281$ , with public key  $e = 3$  and modulus  $n = 3111$ . Compute the private key  $d$  and the message  $m = c^d \pmod{n}$ .

*Hint:* it is probably best to begin by computing the factorization of  $n$ .  $\square$

**Exercise 6.33.** Consider the use of textbook RSA for encryption as well as for signing (using hash-then-sign), with the same public key  $e$  used for encryption and for signature-verification, and the same private key  $d$  used for decryption and for signing. Show this is insecure against chosen-ciphertext attacks, i.e., allows either forged signatures or decryption.

**Exercise 6.34.** The following design is proposed to send email while preserving sender-authentication and confidentiality, using known public encryption and verification keys for all users. Namely, assume all users know the public encryption and verification keys of all other users. Assume also that all users agree on public key encryption and signature algorithms, denoted  $E$  and  $S$  respectively.

When one user, say Alice, wants to send message  $m$  to another user, say Bob, it computes and sends:  $c = E_{B.e}(m || 'A' || S_{A.s}(m))$ , where  $B.e$  is Bob's public encryption key,  $A.s$  is Alice's private signature key, and ‘ $A$ ’ is Alice’s (unique, well-known) nickname, allowing Bob to identify her public keys. When Bob receives this ciphertext  $c$ , it first decrypts it, which implies it was sent to him. He then identifies by the ‘ $A$ ’ that it was purported to be sent by Alice. To validate this, he looks up Alice's public verification key  $A.v$ , and verifies the signature.

1. Explain how a malicious user *Mal* can cause *Bob* to display a message  $m$  as received from *Alice*, although *Alice* never sent to *Bob* that message. (*Alice* may have sent a different message, or sent that message to somebody else.)
2. Propose a simple, efficient and secure fix.

**Exercise 6.35** (Signcryption). Many applications require both confidentiality, using recipient's public encryption key, say  $B.e$ , and non-repudiation (signature), using sender's verification key, say  $A.v$ . Namely, to send a message to *Bob*, *Alice* uses both her private signature key  $A.s$  and *Bob*'s public encryption key  $B.e$ ; and to receive a message from *Alice*, *Bob* uses his private decryption key  $B.d$  and *Alice*'s public verification key  $A.v$ .

1. It is proposed that *Alice* will select a random key  $k$  and send to *Bob* the triplet:  $(c^K, c^M, \sigma) = (E_{B.e}(k), k \oplus m, \text{Sign}_{A.s}(\text{'Bob'} || k \oplus m))$ . Show this design is insecure, i.e., a MitM attacker may either learn the message  $m$  or cause *Bob* to receive a message 'from *Alice*' - that *Alice* never sent.
2. Propose a simple, efficient and secure fix. Define the sending and receiving process precisely.
3. Extend your solution to allow prevention of replay (receiving multiple times a message sent only once).

## Chapter 7

# The TLS/SSL protocols for web-security and beyond

In this chapter, we discuss the *Transport-Layer Security (TLS)* protocol, which is the main protocol used to secure connections over the Internet - and, in particular, web-communication. TLS is currently at version 1.3, and we will also discuss earlier versions: TLS 1.2, 1.1 and 1.0, and versions 2.0 and 3.0 of SSL - the predecessor of TLS.

The SSL/TLS protocol is arguably the most ‘successful’ security protocol - it is definitely very widely used; but even more significantly, it is used in many diverse scenarios and environments than any other security protocol.

From the security point of view, the wide popularity of SSL/TLS is a double-edged sword. On the one hand, this wide popularity implies that ‘black-hat crackers’ have strong motivation to find vulnerabilities in the SSL/TLS protocols and in the popular implementations. In fact, the desire to be able to ‘break’ secure connections, even motivates powerful organizations such as the NSA to invest extensive efforts in ‘injecting’ intentional, hidden vulnerabilities (*backdoors*) into the protocols and implementations. One important example are the vulnerabilities built-into the Dual-EC Deterministic Random Bit Generator (DRBG) [31], which was later ‘pushed’ into widely-used libraries used by SSL/TLS implementations - all as part of the efforts of the NSA to improve its abilities to eavesdrop on communications.

On the other hand, this wide popularity also motivates extensive efforts by the ‘white-hat’ security community, including researchers from academia and industry, to identify vulnerabilities and improve the security of the protocols and their implementations. As a result, many extensions and changes have been proposed over the years to improve security, as well as to adapt the protocol to new requirements or scenarios. Some of these extensions and changes were adopted as an inherent part of a new revision of the protocol, and many others can be deployed using the built-in extensions mechanism, which itself was added as an extension (mostly from TLS version 1.1).

This wide use of SSL/TLS also has dual implication for learning and teach-

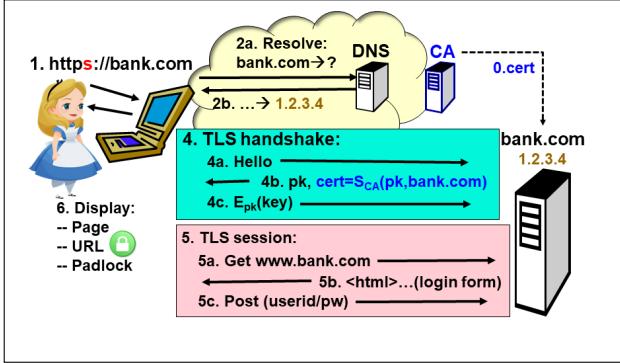


Figure 7.1: Overview of the use of TLS/SSL, with server public key certificate, to secure the connection between browser and website.

ing SSL/TLS. On the one hand, being such an important, widely-used and widely-studies protocol, implies that this is probably the most important and most interesting protocol to study. On the other hand, this also means that there is an excessive wealth of important and interesting information - indeed, entire books were dedicated to cover SSL/TLS, e.g., [90, 95], and even they do not cover all aspects. We have tried to maintain a reasonable balance, however, there were many hard choices and surely there is much to improve; as in other aspects, your feedback would be appreciated.

## 7.1 Introducing TLS/SSL

### 7.1.1 TLS/SSL: High-level Overview

The TLS and SSL protocols were originally designed to secure the communication between a web-browser and a web-server, and, while they are now widely deployed for additional applications, web-security remains their main application. We present a highly-simplified overview of this typical use-case in Figure 7.1.

In Figure 7.1, we show Alice, a typical web user, browsing to the protected website *https://bank.com*. Notice that the URL begins with the protocol name *https*, rather than the protocol name *http*, used for unprotected web sites. The process consists of the following steps:

1. The user (Alice) enters the desired *Universal Resource Locator (URL)*; in the given example, the URL is *https://bank.com*. The URL consists of the protocol (*https*) and the *domain name* of the desired web-server (*bank.com*); in addition, the path may contain identification of a specific *object* in the server. In this example, Alice does not specify any specific object; and the browser considers this a request for the default object *index.html*. The choice of the *https* protocol, instructs the browser to

open a *secure connection*, i.e., send the HTTP requests over an SSL/TLS session, rather than directly over an unprotected TCP connection. Note that the request may be specified in one of three ways: (1) by the user ‘typing’ the URL into the address-bar of the browser, i.e., ‘manually’, (2) ‘semi-automatically’, by the user clicking on an *hyperlink* which specified this URL, or (3) ‘fully automatically’, by a script running in the browser; scripts can instruct the browser to request different objects.

2. This step is used, if necessary, to resolve the *Internet-Protocol (IP) address* of the domain name of the server (*bank.com*). The step is not necessary, and hence skipped, if this IP address is already known (e.g., cached from previous connection). To find the IP address, the browser sends a *resolve* request to a *Domain Name System (DNS) server*, specifying the domain (*bank.com*). The server responds with the IP address of the *bank.com* domain, e.g., *1.2.3.4*.
3. The browser sends the TCP SYN message to the web-server’s IP address (*1.2.3.4*), and the server responds with the TCP SYN+ACK message, thereby opening the TCP connection between the browser and the web-server.
4. The browser sends the TLS Hello message, and the *bank.com* responds with its public key (*pk*) and its *certificate*, linking the public key *pk* to the domain name *bank.com*, and signed, much in advance, by a trusted *certificate authority (CA)*. The client validates PK based on certificate, and then uses the public key *pk* to encrypt a random, secret shared *key*. These steps are referred to as the *TLS handshake*.
5. Next, the client and server communicate in a *TLS session*, sending message back and forth. This includes sending GET, POST and other *requests*, from the browser to the web-server, and receiving the responses. The figure shows some typical flows.
6. Finally, the browser displays the page to the user (Alice), together with few *security indicators* such as the URL and the padlock.

### 7.1.2 TLS/SSL: security goals

TLS and SSL are designed to ensure security between two computers, usually referred to as a *client* and a *server*, in spite of attacks by a MitM (Monster-in-the-Middle) attacker. Table 7.1 lists these security goals, and indicates whether they are achieved by different versions of the TLS/SSL handshake protocol. Most goals are met by all versions - except for vulnerabilities, which often allowed attackers to circumvent some of the goals. Few goals, mainly Perfect Forward Secrecy (PFS), were not achieved by SSLv2 - but all were achieved by later versions.

The goals of the different versions of TLS/SSL, as summarized in Table 7.1, include:

Goal	SSLv2	SSLv3	TLS1.0-1.2	TLS1.3
Key exchange	●	●	●	●
Server authentication	●	●	●	●
Client authentication	●	●	●	●
Connection integrity	●	●	●	●
Confidentiality	●	●	●	●
Cipher agility	○	●	●	●
PFS	○	●	●	●

Table 7.1: TLS/SSL: security goals vs. versions: ●: achieved , ○: achieved partially, ○: not achieved. Note that known attacks and vulnerabilities, may cause failure to achieve a goal (even where marked here as ‘achieved’) - we present several of these.

**Key exchange:** securely setup a secret shared key, preventing exposure of this key to a MitM attacker.

**Server authentication:** authenticate the identity of the server, i.e., assure the client that it is communicating with the right server.

**Client authentication:** authenticate the identity of the client. Client authentication is optional; in fact, TLS/SSL is usually used without client authentication, allowing an anonymous, unidentified client to connect to the server. When client authentication is desired, it is usually performed by sending a *secret credential* within the TLS/SSL secure connection, such as a password or cookie.

**Connection Integrity:** Ensure that the communication received by one party, is exactly identical to the communication sent by the peer (in spite of a MitM attacker). This includes preventing message re-ordering and truncating attacks. Of course, an attack - or even benign failure - could disrupt communication, leaving some information sent but never received by the peer, but TLS/SSL would detect such events.

**Connection confidentiality:** Ensure that a MitM attacker cannot learn anything about the information sent between the two parties, except for the ‘traffic pattern’ - amount of information sent/received information.

**Cipher agility:** Cipher agility allows us to *change* the cryptographic algorithms used (i.e., *cipher-suite negotiation*). Cipher agility is an important property of cryptographic protocols; in particular, it is essential, when a vulnerability is found or suspected in a particular algorithm (mainly, cryptosystem, MAC or hash).

**Perfect forward secrecy (PFS):** From version 3 of SSL, the SSL/TLS handshakes support the (optional) use of authenticated DH key agreement,

which ensures perfect forward secrecy (PFS), as discussed in subsection 6.4.1.

**Robust cryptography.** Cipher agility helps us to recover *after* a possible cryptographic vulnerability is *identified*; until then, the system could be vulnerable. In contrast, *cryptographic robustness* requires the protocol to ensure its security properties, *even while* some of its cryptographic modules or assumptions are weak. In particular, the protocol should maintain security even if its sources of randomness are imperfect, and when one of its cryptographic algorithms ('building blocks') is vulnerable, i.e., deploy *robust combiner* design. Cryptographic robustness was not defined as an explicit goal in SSL/TLS, however, there have been several elements of robust cryptographic designs since SSLv3.

### 7.1.3 SSL/TLS: Engineering goals

In addition to the security goals, the success of TLS/SSL is largely due to its focus - from the very first versions - on the generic 'engineering goals', applicable to any system, of *efficiency*, *ease of deployment and use*, and *flexibility*. By addressing these goals, TLS/SSL is widely used and applicable in a very wide range of applications and scenario. Let us briefly discuss these three *engineering goals*.

**Efficiency - and session resumption.** Efficiency is always a desirable goal. In the case of TLS/SSL, there are two main efficiency considerations: *computational overhead* and *latency*. In terms of *computational overhead*, the main consideration is minimizing the computationally-intensive public-key operations. To minimize public-key operation, once the handshake establishes a shared key (using public key operations), the parties may reuse this key to establish future connections without requiring additional public-key operations; we refer to the set of connections based on the same public-key exchange as a *session*, and an handshake that reuses the pre-exchanged shared key as a *session-resumption* handshake.

In terms of minimizing *latency*, the main consideration is to minimize the number of round trip exchanges; end-to-end delays are typically in order of tens to hundreds milliseconds, which is usually much higher than the transmission delays, esp. for the limited amount of information sent in TLS/SSL exchange. Reducing the number of round trips became even more important as transmission speeds increased; this is reflected by the fact that until TLS 1.3, all designs had the fixed number of two round-trips to complete the handshake, only then allowing the client to send a protected message (already in the third exchange). In contrast, TLS 1.3 handshake requires only a single round-trip (before sending a protected message), and even allows the clients to send a request already in the first exchange (with some limitations and somewhat reduced security properties, see later).

A more minor efficiency consideration is minimization of bandwidth; this is mainly significant in scenarios where bandwidth is limited, such as very noisy wireless connections.

**Extensibility and versatility.** The SSL/TLS protocol is arguably the most ‘successful’ security protocol - it is definitely very widely used; but even more significantly, it is used in many diverse scenarios and environments than any other security protocol. The use of the protocol in such diverse scenarios is based on its *extensibility and versatility*. The protocol supports many optional mechanisms, e.g., client authentication, and flexibility such as cipher-agility; furthermore, from TLS version 1.1 and even earlier, the TLS protocol supports a built-in extensions mechanisms, providing even greater flexibility.

**Ease of deployment and use.** Finally, the success and wide-use of the SSL/TLS protocols is largely due to their ease of deployment and usage. As shown in Figure 7.2, the TLS/SSL protocol is typically implemented ‘on top’ of the popular *TCP sockets API*, and then used by applications, directly or via the *HTTSPS* or other protocols. This architecture makes it easy to install and use SSL/TLS, without requiring changes to the operating-system and kernel. This is in contrast to some of the other communication-security mechanisms, in particular the IPsec protocol [44,52], which, like TLS, is also an IETF standard.

#### 7.1.4 TLS/SSL and the TCP/IP Protocol Stack

See Figure 7.2 for the placement of the TLS protocols with respect to the TCP/IP protocol stack, and Figure 7.3 for a typical connection.

[This subsection is yet to be written; this material is well covered in many textbooks on networking, e.g., [78].]

#### 7.1.5 The SSL/TLS record protocol

We now discuss the SSL/TLS record protocol; our discussion is very brief, since our focus is on the handshake protocol.

The SSL/TLS record protocol assumes that it is run ‘on-top’ of an underlying reliable communication protocol - typically, TCP. Hence, it assumes that, if there is no attack, messages sent are received reliably, without losses, duplications or re-ordering; any deviation must indicate an attack and justifies closing the connection.

The record protocol provides four services, in the following order:

**Fragment:** break the TCP stream into *fragments*. Each fragment consists of up to 16KB. The motivation for fragmenting is to allow *pipeline* operation, reducing the latency. For example, we may already be sending the first fragment, while encrypting and computing MAC for the second fragment and receiving the third fragment.

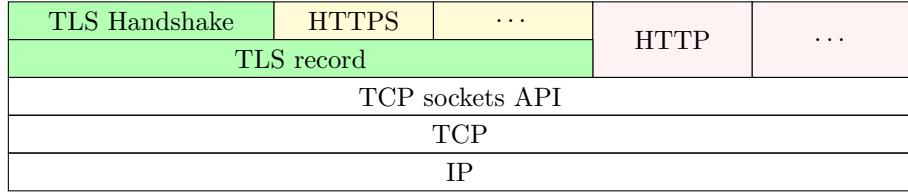


Figure 7.2: Placement of TLS/SSL in the TCP/IP protocol stack. The TLS/SSL record protocol (in green) The TLS handshake protocol is marked in green; it establishes keys for, and also uses, the TLS record layer protocol (also in green). We mark in yellow, the HTTPS protocol, and other application protocols that use the TLS/SSL record protocol. Application protocols that do *not* use TLS/SSL for security, including the HyperText Transfer Protocol (HTTP), are marked in pink. These protocols, as well as TLS/SSL itself, all use the TCP protocol, via the sockets library layer. TCP ensures reliable communication, on top of the (unreliable) Internet Protocol (IP).

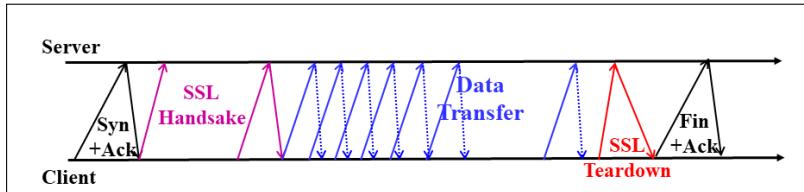


Figure 7.3: Phases of TLS/SSL connection. The black flows (Syn+Ack and later Fin+Ack) are the TCP connection setup and tear-down exchanges, required to ensure reliability. The fuchsia flows represent the TLS/SSL handshake; the blue flows represent the data transfer, protected using TLS/SSL record layer; and the red flows represent the TLS/SSL connection tear-down exchange.

**Compress:** apply lossless compression to each fragment. Compression may reduce the processing overhead and the communication. Note that ciphertext cannot be compressed – therefore, if we want to compress, this must be done before encryption. Note, however, that the length of compressed data depends on the amount of redundancy, and that encryption may not necessarily hide the *length* of the (compressed) plaintext; hence, there is a potential risk of exposure of some measure of the redundancy of data when applying compress-then-encrypt. Indeed, the fact that SSL/TLS applies compression before encryption was exploited in several *compression attacks*, including TIME and CRIME [9, 86, 97, 103]. Note that even if TLS compression is disabled, compression attacks may still be possible by exploiting application-level compression, as done in the BREACH attack.

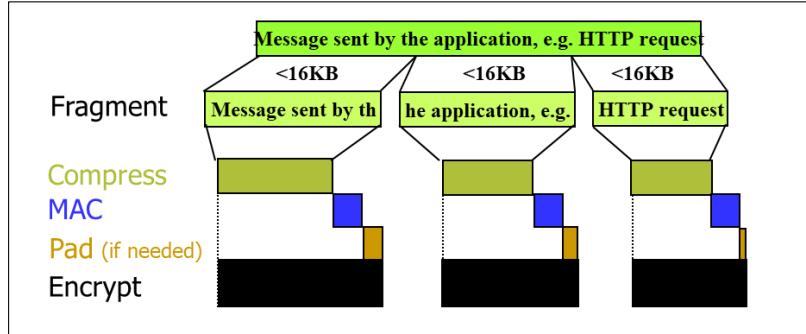


Figure 7.4: The SSL/TLS record layer protocol.

**Authenticate:** SSL/TLS authenticates the plaintext by applying a MAC function to the concatenation of the *sequence number*, *type*, *version*, *length* and the *compressed plaintext fragment*. The outcome is concatenated to the plaintext before applying encryption. The *sequence number* is the number of bytes in the stream of plaintext prior to this fragment; the *type* identifies the application protocol (e.g., *http*); the *version* identifies the TLS/SSL version, and the *length* is the number of bytes in the fragment.

**Encrypt:** SSL/TLS encrypts the concatenation of the *compressed plaintext fragment*, *MAC* and, if necessary, *padding*. Padding is required when using mode-of-operation of block cipher; it is not required for stream ciphers.

## 7.2 The beginning: the handshake protocol of SSLv2

We begin our discussion of the SSL/TLS handshake protocols by presenting, in this section, the *SSLv2 (SSL version 2.0)* handshake protocol, and discussing its features - and some of its main vulnerabilities and deficiencies.

SSL version 2 is the earliest published version of the SSL protocol. The SSLv2 handshake protocol is interesting - and not just for the historical importance of being the first published version of TLS/SSL. One motivation to study it, is that SSLv2 already introduces much of the basic concepts and designs used in later versions - and, since it is a bit simpler, it is a good way for us to introduce these basic TLS/SSL concepts and designs. Second, SSLv2 has some serious vulnerabilities, which are instructive and teaching. Third, amazingly enough, it was recently shown that a significant fraction of web servers still support SSLv2, although they also support (and prefer) later versions - furthermore, this allows attack on clients, even when the client supports *only* newer versions of TLS/SSL.

The SSLv2 handshake is already a non-trivial cryptographic protocol, with support for multiple options and mechanisms - all supported also by later ver-

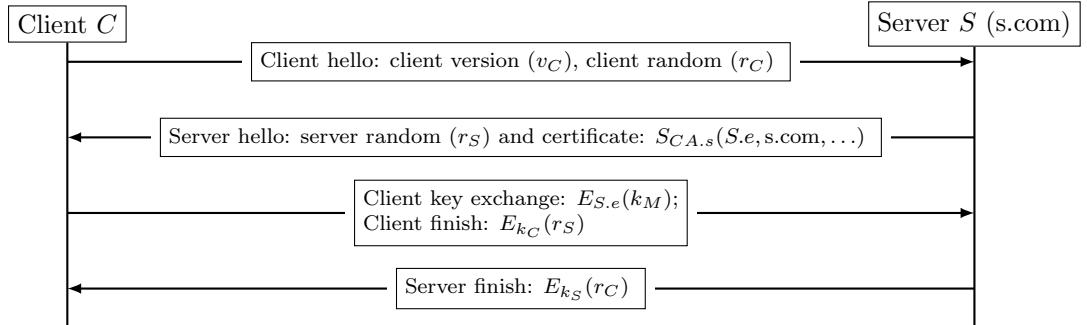


Figure 7.5: ‘Basic’ SSLv2 handshake. The client selects randomly a shared *master key*  $k_M$ , encrypts it using RSA encryption with the server’s public key  $S.e$ , and sends to the server. The client key  $k_C$  and server key  $k_S$  are derived from the master key  $k_M$  and the randoms  $r_C, r_S$  using the MD5 crypto-hash, see Eqs. (7.1,7.2). See more details in Figure 7.5. This ‘basic’ handshake does not include ciphersuite negotiation, session resumption and client authentication, which we illustrate in the following figures.

sions (of SSL and TLS), often with extensions and improvements. We describe the protocol in the following three subsections. In §7.2.1 we present the ‘basic’ handshake, namely, the handshake when there is no existing session (already established shared key), and the protocol uses public-key operations to share a key. In contrast, in §7.2.2 we present the *session resumption* handshake, allowing to re-use the public key exchanged in a previous handshake between the same client and server, to open a new connection without additional public key operations. In §7.2.3 we discuss how SSLv2 handles ciphersuite negotiation, and explain how an attacker may exploit the (insecure) SSLv2 ciphersuite negotiation mechanism, to launch a *simple downgrade attack*. Finally, in §7.2.4 we discuss how SSLv2 supports the (optional) client-authentication feature.

**Terms of SSLv2.** Note that SSLv2, as described in the original publications, e.g., in [66], uses several terms which were modified in later versions; for simplicity and consistency, we use the later terms also when describing SSLv2.

### 7.2.1 SSLv2: the ‘basic’ handshake

In this subsection we discuss the ‘basic’ SSLv2 handshake, illustrated in Fig. 7.5, which is a somewhat simplified version of the SSLv2 handshake protocol. In particular, this simplified version does not include ciphersuite negotiation, session resumption and client authentication. We discuss these additional aspects of SSLv2 in the following subsections.

**Key-derivation and randomization in SSLv2.** The SSLv2 handshake protocol establishes a shared *master key*, which we denote  $k_M$ . The master key is selected by the client, and simply sent encrypted to the server in the Client key exchange message, as  $E_{S,e}(k_M)$ .

The public key encryption of  $k_M$  is the most computationally-intensive operation by the client; therefore, it is desirable for the protocol to be secure even if the client reuses the same master key  $k_M$  and its encryption  $E_{S,e}(k_M)$  in multiple connections, assuming that the master key was not exposed. To ensure this, the client and server each selects and exchange a random ‘nonce’,  $r_C$  and  $r_S$ , respectively, and derive the cryptographic keys to protect communication in the connection from the master key together with both these random numbers.

The use of both random numbers  $r_C$  and  $r_S$  is required, to ensure that a different key is used in different connections. This is essential, to prevent replay of messages - from either client or server; see exercise 7.1.

**Exercise 7.1.** *IoT devices sometimes use very slow, computationally-limited processors. For costs and energy savings, IoT devices often have limited computation power, and often do not have a source of random bits, and no non-volatile memory. Consider such an IoT-lock, used to lock/unlock a car or a door. The IoT-lock is implemented as a simple web server, and uses SSLv2 to authenticate the client requests. Specifically, to unlock (or lock), the client sends to the IoT-lock the corresponding command (‘unlock’ or ‘lock’), together with a ‘secret password/code’, say consisting of 20 alphanumeric characters. Assume that the IoT-lock uses the same server-random string  $r_S$  in all connections, selected randomly upon initialization of the IoT-lock. Show how a message sequence diagram demonstrating how a MitM may trick the IoT-lock into unlocking, by replaying messages from a legitimate connection between the client and the IoT-lock. Note: this attack works for any version of SSL/TLS, for implementations which reuse  $r_S$ .*

In fact, in SSLv2, the parties derive and use *two* keys from  $k_M$  and the random nonces  $r_C, r_S$ : the *client key*  $k_C$ , used by the client to protect messages it sends, and the *server key*  $k_S$ , used by the server to protect messages it sends. These are derived as follows:

$$k_C = MD5(k_M || "0" || r_C || r_S) \quad (7.1)$$

$$k_S = MD5(k_M || "1" || r_C || r_S) \quad (7.2)$$

The SSLv2 designers chose, wisely, to separate between  $k_C$ , used to protect messages from client to server, vs.  $k_M$ , used to protect messages from server to client ( $k_S$ ). In particular, many websites are public, and send exactly the same information to all users; however, we may want to protect the confidentiality of the contents, e.g., queries, sent by the users. By separating between  $k_C$  and  $k_S$ , the attacker cannot use the large amount of known plaintext sent from server to client, to cryptanalyze the ciphertext sent from client to server. This separation follows the principle of key separation (§2.5.6).

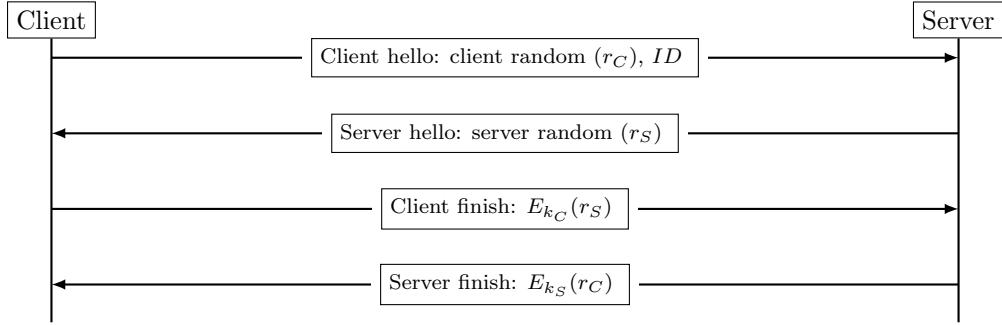


Figure 7.6: SSLv2 handshake, with  $ID$ -based session resumption. The client-hello includes session- $ID$ , received from server in a previous connection. If the server does not have the  $(ID, k_M)$  pair, then it simply ignores the  $ID$  and sends server-hello, as in the ‘basic’ handshake (Fig.7.5). However, if the server still has  $(ID, k_M)$ , from a previous connection, then it reuses  $k_M$ , i.e., ‘resumes the session’, and derives new shared keys from it (using Eqs. (7.1,7.2)). This avoids the public key operations, encryption by client and decryption by server of master key  $k_M$ , as well as the overhead of transmitting the certificate  $S_{CA.s}(S.e, s.com)$ .

However, the SSLv2 design does not fully follow the principle of key-separation. In particular, it uses the same key for confidentiality (encryption) and message-authenticity (MAC). This is improved in later versions of SSL/TLS, which we present in the following sections.

### 7.2.2 SSLv2: $ID$ -based Session Resumption

The main overhead of the SSL/TLS protocol is due to the computationally-intensive public key operations. Often, there are multiple connections between the same (client, server) pair, over short period of time; in such cases, the server and client may re-use the master key exchanged previously, thereby avoiding additional public key operations. To identify the re-use of the same master key, the server includes an identifier  $ID$  at the end of a handshake where the key is exchanged, and the clients sends this  $ID$  with its client-hello, to re-use the same master key in another connection (without additional public key operations). This process is called *session resumption*, and we illustrate it in Figure 7.6.

The impact of session-resumption can be quite dramatic. The savings are mostly on the computation (CPU) time; instead of computing public-key encryption (for client) and decryption (for server) for every TCP connection, we now need only require these operations for the first TCP connection in a session. The ratio of the computation time with and without session resumption is typically on the orders of 100, for typical usage such as for protecting web communication, using the *https* protocol, i.e., running *http* over *SSL/TLS*.

However, the *ID*-based session resumption mechanism, which is the only one supported in SSLv2, has a significant drawback: it requires the server to be *stateful*, specifically, to maintain state for each session (for the session-*ID* and the master key). In the typical case where the same web-server is running over multiple machines, this requires that this storage be shared between all of these servers, or to ensure that a client will contact the same machine each time - a difficult requirement that sometimes is infeasible. These drawbacks motivate the adoption of alternative methods for session resumption, most notably, the session-token mechanism that we discuss later (and which requires the use of TLS).

Note that the session resumption protocol is one reason for requiring the use of client and server random numbers; see the following exercise.

**Exercise 7.2.** Consider implementations of the SSLv2 protocol, where the (1) client random or (2) server random fields are omitted (or always sent as a fixed string). Show a message sequence diagram for corresponding attacks, allowing replay of messages to the client or to the server.

*Hint:* perform replay of messages from one connection to a different connection (both using the same master key, i.e., same session).  $\square$

### 7.2.3 SSLv2: ciphersuite negotiation and downgrade attack

In §2.7 we presented the *cryptographic building blocks principle* (Principle 7), stating that systems should base security on few, well-defined building blocks. The goals of this principle include *cipher-agility*, i.e., allowing flexibility, replacement and upgrade of the cryptographic mechanisms. For example, in §5.5.3 we discuss how GSM supports cipher-agility, to allow use of the weak-security, exportable A5/2 encryption scheme, as well as of the stronger, non-exportable encryption algorithms A5/1 and A5/3; we also discuss how the GSM support for cipher-agility is vulnerable to *downgrade attack*.

SSLv2 also supports cipher-agility. Figure 7.7 illustrates the SSLv2 ciphersuite negotiation mechanism, and Fig 7.8 is an example of the negotiation process, when the client supports three ciphersuites, all using the MD5 hashing algorithm, but with three different ciphers (128-bit RC4, 40-bit RC4 and 64-bit DES), and the server supports two ciphers (128-bit RC4 and 40-bit RC4).

In SSLv2, the finish messages only confirm that the parties share the same server and client keys ( $K_S$  and  $K_C$ , respectively), but not the integrity of the rest of the hello messages - in particular, there is no authentication of the ciphersuites sent by server and client. This allows simple downgrade attacks, removing ‘strong’ ciphers from the list of ciphers supported by client and/or server. Figure 7.9 illustrates how a Monster-in-the-Middle (MitM) attacker may perform this downgrade attack on SSLv2; in the example illustrated, the attacker removes the ‘regular-version’ 128-bit RC4 encryption from the list of ciphers supported by the client, leaving only the weaker ‘export-version’ 40-bit RC4 encryption. Note that the SSLv2 downgrade attack is even simpler than the downgrade attack on GSM (§5.5.3).

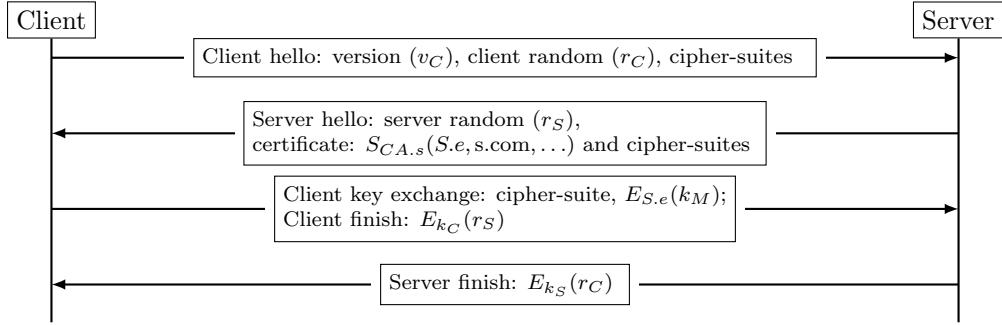


Figure 7.7: SSLv2 handshake, with ciphersuite negotiation. Both client and server indicate the cipher-suites they support in their respective hello messages; then, the client key specifies its preferred cipher-suite in the exchange message (this aspect was modified in later versions).

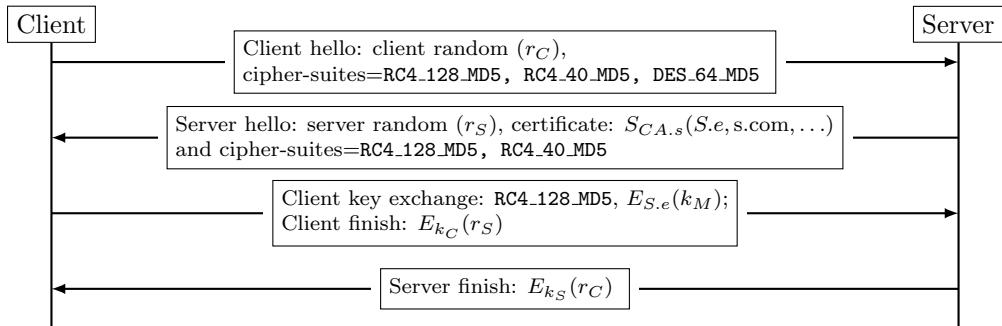


Figure 7.8: SSLv2: example of cipher-suite negotiation.

#### 7.2.4 Client authentication in SSLv2

All versions of SSL and TLS, including SSLv2, support an (optional) *client authentication* mechanism, where the client proves its identity by sending a certificate for a public signature-validation key, and then signs content sent by the server. We do not present details of the client authentication mechanism in SSLv2, since it differs considerably from the design in later versions - but not in a very interesting way, so it seems unnecessary to describe it in details. Later versions simply implement client authentication in slightly simpler and more efficient manner. We discuss client authentication in the following sections, presenting the later versions of SSL and TLS.

### 7.3 The Handshake Protocol: from SSLv3 to TLSv1.2

We now discuss the evolution of the SSL/TLS handshake protocol after version 2, from version 3 of SSL [53], to versions 1.0, 1.1 and 1.2 of TLS [38, 39, 110].

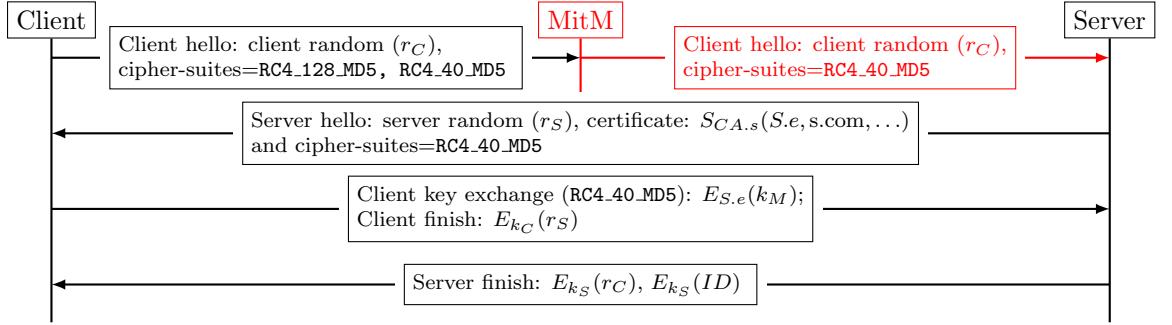


Figure 7.9: The *SSLv2 downgrade attack*. Server and client end up using master key  $k_M$  with only 40 secret bits, which the attacker can find by exhaustive search. Attacker does not need to find key during handshake; parties use the 40-bit key for entire connection, attacker may even just record ciphertexts and decrypt later.

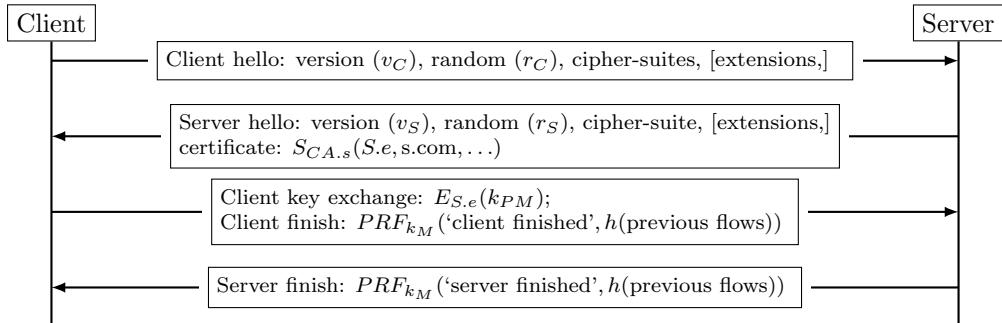


Figure 7.10: The ‘basic’ RSA-based handshake, for SSLv3 and TLS 1.0, 1.1 and 1.2. The master key  $k_M$  is computed, as in Eq. (7.3), from the pre-master key  $k_{PM}$ , which is sent in the *client key exchange* message.

These four handshake protocols are quite similar - we will mention the few major differences. Later, in §7.4, we present version 1.3 of TLS, which is the latest - and involves more significant differences, compared to the more incremental changes of these earlier versions.

Figure 7.10 illustrates the ‘basic’ variant of the handshake protocol, of the SSLv3 protocol and the TLS protocol (versions 1.0 to 1.2). Like SSLv2, this ‘basic’ variant uses RSA encryption to send encrypted key from client to server.

In the following subsections, we discuss the main improvements introduced in these later versions of SSL/TLS, including:

**Improved key derivation and  $k_{PM}$  (§7.3.1):** the key derivation process was significantly overhauled between SSLv2 and the later versions, beginning with SSLv3. In particular, the client-key-exchange message of the basic

exchange includes the premaster key  $k_{PM}$ , from which the protocol derives the master key  $k_M$ . As before, the master key  $k_M$  is used to derive the keys for the record-protocol, used to encrypt and authenticate data on the connection.

**Improved negotiation and handshake integrity (§7.3.2):** from SSLv3, the finish message authenticates all the data of all previous flows of the handshake; this prevents the SSLv2 downgrade attack (Figure 7.9). TLS, and to lesser degree SSLv3 too, also improve other aspects of the negotiation, in particular, support for extensions, negotiation of the protocol version, and negotiation of additional mechanisms, including key-distribution and compression.

**DH key exchange and PFS (§7.3.4):** From SSLv3, the SSL/TLS protocols supports DH key exchange, as an alternative or complementary mechanism to the use of RSA-based key exchange (the only method in SSLv2). The main advantage is support for *Perfect forward secrecy (PFS)*.

**Session-Ticket Resumption (§7.3.5):** an important TLS extension allows *Session-Ticket Resumption*, a new mechanism for session resumption. Session-ticket resumption allows the server to avoid keeping state for each session, which is often an important improvement over the *ID-based session resumption mechanism* supported already in SSLv2 (but which requires servers to maintain state for each session).

Two of these changes - improved key derivation and improved handshake integrity - have impact already on the ‘basic’ handshake. To see this impact, compare Figure 7.10 (for SSLv3 to TLS 1.2) to Figure 7.5 (the corresponding ‘basic’ handshake of SSLv2). We therefore begin our discussion with these two.

### 7.3.1 SSLv3 to TLSv1.2: improved derivation of keys

#### Deriving master key from premaster key

From SSLv3, the handshake protocol exchanges a *pre-master key*  $k_{PM}$ , instead of the master key  $k_M$  exchanged in SSLv2. The parties derive the master key  $k_M$  from the pre-master key  $k_{PM}$ , using a PRF, as in Eq. (7.3):

$$k_M = \text{PRF}_{k_{PM}}(\text{"master secret"} || r_C || r_S) \quad (7.3)$$

The main motivation for this additional step is that the value exchanged between the parties, may not be a perfectly-uniform, secret binary string, as required for a cryptographic key. When exchanging the shared key using the ‘basic’, RSA-based handshake, this may happen when the client does not have a sufficiently good source of randomization, or if the client simply resends the same encrypted premaster key as computed and used in a previous connection to the same server - not a recommended way to use the protocol, of course, but possibly attractive for some very weak clients.

Table 7.2: Derivation of connection keys and IVs, in SSLv3 to TLS1.2

$key-block = PRF_{k_M}(\text{'key expansion'}    r_C    r_S)$					
$k_C^A$	$k_S^A$	$k_C^E$	$k_S^E$	$IV_C$	$IV_S$

When exchanging the shared key using the DH protocol, there is a different motivation for using this additional derivation step, from premaster key to master key. Namely, the standard DH groups are all based on the use of a safe prime; as we explain in §6.2.3, this implies that we rely on the Computational DH assumption (CDH), and that the attacker may be able to learn at least one bit of information about the exchanged key. By deriving the master key from the premaster key, we hope to ensure that the entire master key would be pseudorandom.

### Deriving connection keys

Another important improvement of the handshake protocols of SSLv3 to TLS1.2, compared to the SSLv2 handshake, is in the derivation of the connection keys, used for encryption and authentication by the record protocol. This aspect is not apparent from looking at the flows (Fig. 7.10).

Specifically, recall that in SSLv2, we derived from the master-key  $k_M$  two keys,  $k_S$  for protecting traffic sent by the server  $S$ , and  $k_C$  for protecting traffic sent by the client  $C$ , as in Eqs. (7.1, 7.2). In SSLv3 and TLS, we use  $k_M$  to derive, for traffic sent by the client  $C$  and server  $S$ , *three* keys/values each, for a total of *six* keys/values: two authentication (MAC) keys,  $(k_C^A, k_S^A)$ , two encryption keys,  $(k_C^E, k_S^E)$ , and two initialization vectors,  $(IV_C, IV_S)$ . In each pair we have one key/value for traffic sent by client  $C$ , and the other for traffic sent by server  $S$ .

To derive these six keys/values, we generate from  $k_M$  a long string which is referred to as *key block*, which we then partition into the six keys/values. The exact details of the derivation differ between these different versions of the handshake protocol, and arguably, *none* of the derivations is fully justified by standard cryptographic definitions and reductions. We present the following simplification, leaving the exact details for exercises; the interested reader can find the full details in the corresponding RFC specifications.

Our simplification is defined using a generic Pseudo-Random Function  $PRF$ , whose input is an arbitrary-length string, and whose output is a ‘sufficiently long’ pseudo-random binary string called *key-block*, as follows:

$$key-block = PRF_{k_M}(\text{'key expansion'} || r_C || r_S) \quad (7.4)$$

The key-block is then partitioned into the *six* keys/values, as illustrated in Table 7.2.

### 7.3.2 Crypto-agility, backwards compatibility and downgrade attacks

SSLv2 already supports cipher-agility, with the negotiation of ciphersuites. However, the negotiation mechanism in SSLv2 is both very limited and insecure, resulting in significant overhaul in later versions of the SSL/TLS handshake. Many of these changes were required to deal with vulnerabilities and attacks exploiting weaknesses of the negotiation mechanisms, beginning with the simple downgrade attack on SSLv2 (Figure 7.9); others were required to add additional flexibility and options, often required for critical security or functionality features; some examples follow.

We begin by discussing a change which is rather insignificant, yet, it is visible already by comparing the ‘basic’ handshake flows of the two versions: Figure 7.10 (for SSLv3 to TLS 1.2) vs. Figure 7.5 (the corresponding ‘basic’ handshake of SSLv2). Namely, in both versions, the client-hello message includes the list of cipher-suites supported by the client. However, in SSLv2, the server responds with the subset of this list which the server also supports (server’s cipher-suites), and the client chooses its preferred cipher-suite from this list. In contrast, from SSLv3, the server responds, in server-hello, with its choice of the cipher-suite, rather than sending its own cipher-suite list to let the client decide, as in SSLv2. The goal of this change appears merely to simplify the handshake a bit, since, in practice, SSLv2 servers almost always sent only one cipher-suite back, the one they most preferred among those offered by the client, making it redundant for the client to send back the same value.

With this out of the way, let us consider the other changes, which, in contrast, have significant security impact.

#### Handshake integrity - preventing SSLv2 downgrade attack

Beginning with SSLv3, the handshake protocol includes a simple mechanism for validating the integrity of all handshake messages. Namely, each side, client and server, authenticates the entire handshake, using the master key derived for that connection. This improvement prevents the SSLv2 downgrade attack.

Specifically, from SSLv3, the client and server send, in their respective *finish* message, a *validation value* denoted  $v_C, v_S$  respectively, whose goal is to validate the integrity of all previously exchanged messages in that handshake; upon receiving the finish-message from the peer (server or client, respectively), the value is checked and if incorrect, the handshake is aborted.

Similarly to the keys derivation process (§7.3.1), the details slightly differ among the different versions, and we present a slight simplification, consistent with the one we used in §7.3.1. Namely, similarly to the derivation of the key-block (Eq. (7.4)), the client and server compute the validation values  $v_C, v_S$  as follows, using a pseudorandom function  $PRF$  with arbitrary-length inputs:

$$v_C = PRF_{k_M} (\text{‘client finished’} || h(\text{handshake-messages})) \quad (7.5)$$

$$v_S = PRF_{k_M} (\text{‘server finished’} || h(\text{handshake-messages})) \quad (7.6)$$

The equations use a cryptographic hash function  $h$ , whose definition differs between the different versions. Specifically, in TLS 1.2, the hash function is implemented simply as SHA-256, i.e.,  $h(m) = \text{SHA\_256}(m)$ . The TLS 1.0 and 1.1 design is more elaborate, and follows the ‘robust combiner for MAC’ design of §3.4.2; specifically, the hash is computed by concatenating the results of two cryptographic hash functions,  $MD5$  and  $SHA1$ , as:  $h(m) = MD5(m)||SHA1(m)$ . SSLv3 also similarly combines MD5 and SHA1, however, there the combination is in the computation of  $PRF$  itself; details omitted.

### **Backwards compatibility and protocol-version negotiation**

SSL was an immediate success; it was widely deployed soon after it was released (as SSLv2). Hence, when introducing SSLv3, designers had to seriously consider *backward compatibility*, namely, allowing a client/server running a new version of SSL/TLS, to interact with a server/client, respectively, running an older version. Note that already SSLv2 includes a *version number* in the client hello and server hello messages.

The idea is that new servers can use an older version of the protocol, when the client-hello message indicates that the client only supports this old version. Similarly, to allow new clients to interact with servers running older versions, all versions use client-hello messages which are compatible with SSLv3. This allows the server to process the hello message, responding with its (older, lower) version; the handshake then proceeds using the older, lower version.

Note that this backwards-compatibility mechanism requires support by the server. In reality, some servers simply fail to respond to new protocol versions. Some clients try to work with such servers anyway, by a *downgrade dance*: try first to connect using the latest version, but if receiving no response (or error), try with older versions. However, this is vulnerable; an attacker can simple block connection attempts (or send back a fake error message), causing the client to use an older, vulnerable version of the protocol; see exercise below.

**Exercise 7.3.** Consider client that supports ‘downgrade dance’ as described above, trying first to connect using TLS 1.2, if fails - using TLS 1.1, and if fails- using TLS 1.0. Present message sequence diagram for a MitM attack, tricking this client into using TLS 1.0.

### **Securing the downgrade dance: the SCSV cipher-suite and beyond**

Exercise 7.3 shows a potential vulnerability for the common case, where clients use ‘downgrade dance’ to ensure backwards compatibility with servers supporting older (lower) versions of the TLS/SSL protocol. How can we mitigate this risk, while still allowing clients running new versions of TLS to interact with servers running older versions?

The standard solution is the *Signaling Cipher Suite Value (SCSV)* cipher-suite, specified in RFC 7507 [87]. Clients that support SCSV, first try to connect to the server using their current TLS version - no change from clients not supporting SCSV. The difference is only when this initial connection fails,

and the client decides to try the ‘downgrade dance’, to support connections with servers supporting (only) older versions of TLS/SSL.

In these ‘downgrade dance’ handshakes, the client adds a special ‘cipher suite’ to its list of supported *cipher suites*, sent as part of the *ClientHello* message. The special ‘cipher suite’ is called *TLS\_FALLBACK\_SCSV*, and is encoded by a specific string. Unlike the original (and main) goal of the *cipher suites* field, the SCSV is not an indication of cryptographic algorithms supported by the client. Instead, the existence of SCSV indicates to the server, that this handshake message is sent as part of a downgrade dance by the client, i.e., that the client supports a *higher* version than the one specified in the current handshake. If the server receives such handshake, and supports a higher version of the protocol itself, this would indicate an error or attack, as this client and server should use the higher version. Therefore, in this case, the server responds with an appropriate indication to the client.

This use of the cipher-suites fields for signaling the downgrade dance is a ‘hack’ - it is not the intended, typical use of this field. A ‘cleaner’ alternative would be to achieve similar signaling using a dedicated extension mechanism; later in this section, we describe the TLS extension mechanism, which could have been used for this purpose. We believe that the reason that SCSV was defined using this ‘hack’ (encoding of a non-existent cipher suite) rather than using an appropriate TLS extension, was the desire to support downgrade dance to older versions of the TLS/SSL protocol, that do not support TLS extensions.

### **SSL-Stripping and HSTS: Downgrade to Insecure Connection and Defenses**

An even more extreme downgrade attack, is to trick the client into using an insecure connection, even when the server supports secure (TLS/SSL) connections. In particular, this attack may be attempted against web connections, which are done over either the (unprotected) HTTP protocol, or over the (protected) HTTPS protocol, which is essentially HTTP running over TLS/SSL.

Browsers connect to websites using the protocol - HTTP or HTTPS - specified in the URL, which is often received as a hyperlink in the previously-received webpage. If that previously-received webpage is unprotected, then the hyperlink may be modified by a MitM attacker - specifically, changing from the protected HTTPS to the unprotected HTTP.

Browsers provide an indication to the user of the protocol used (HTTP or HTTPS), but many or most users are unlikely to notice that this downgrade (from HTTPS to HTTP). This attack is referred to as *SSL-Stripping*, and was first presented by Marlinspike [?].

The best defense against this kind of attack, if for the browsers to detect or prevent HTTP hyperlinks to a website which always uses (or offers) HTTPS connections. The standard mechanism to ensure that is the *HSTS (HTTP Strict Transport Security)* policy, defined in RFC 6797 [67]. The HSTS policy indicates that a particular domain name (of a web server), should be used only

via HTTPS (secure) connections, and not via unprotected (HTTP) connections. HSTS is sent as an HTTP header field (Strict-Transport-Security), in an HTTP response sent by the web server to the client.

The HSTS policy specifies that the specific domain name, and optionally also subdomains, should always be connected using HTTPS, i.e., a secure connection. Specifically,

1. The browser should only use secure connections to the server; in particular, if the browser receives a request to connect to the server using the (unprotected) HTTP protocol, the browser would, instead, connect to the server using the HTTPS (protected) protocol, i.e., using HTTP over SSL/TLS.
2. The browser should terminate any secure transport connection attempts upon any secure transport errors or warnings, e.g., for use of invalid certificate.

The HSTS policy is designed to prevent attacks by a MitM attacker, hence, the HSTS policy itself must be protected - and, in particular, the attacker should not be able to ‘drop’ it. For this reason, HSTS policy must be known to the browser *before* it connects to the server. This may be achieved in two ways:

**Caching - *max-age*:** The HSTS header field has a parameter called *max-age*, which defines a period of time during which the browser should ‘remember’ the HSTS directive, i.e., keep it in cache. Any connection within this time, would be protected by HSTS. This works if the browser had a secure connection with the site, and received the HSTS header, *before* the amount of time specified in *max-age*. This motivates the use of large *max-age*; however, notice that if a domain must move back to HTTP for some reason, or there are failures in the secure connection attempts for some reason, e.g., expired certificate, then the site may be unreachable for *max-age* time units.

**Pre-loaded HSTS policy:** The browser maintains a list of HSTS domains which are *preloaded*, i.e., does not require a previous visit to the site by this browser. This avoids the risk of a browser accessing an HSTS-using website but without a cached HSTS policy. However, this requires the browser to be preloaded with the HSTS policy - a burden on the site and on the browser, and some overhead for this communication. An optional parameter of the HSTS header, instructs search engines to add the site to the HSTS preload list of related browsers. This is used by Google to maintain the pre-loaded HSTS list of the Chrome browser.

### Backward compatibility with SSLv2

The SSLv3/TLS backward-compatibility mechanism does not extend to SSLv2, since SSLv2 uses a different client-hello format. To allow interoperability with

SSLv2 servers, the SSLv3 specifications [53] allows clients to interact with SSLv2 servers, by sending the client-hello message using the SSLv2 format, specifying the use of version 3. However, the standard warns that this method for backwards compatibility will be ‘phased out with all due haste’. The reason is that a client that uses the SSLv2 hello message, may be subject to a simple MitM downgrade attack causing them to use the SSLv2 handshake, even when interacting with an SSLv3 server; see following exercise.

**Exercise 7.4** (Protocol downgrade attack on SSLv3). *Show message sequence diagram for a MitM version downgrade attack, tricking an SSLv3 server and an SSLv3 client who sends SSLv2-format client-hello (for backward compatibility), into completing the handshake using SSLv2 and using a weak (40-bit) cipher.*

*Hint:* see this attack (referred to as ‘version rollback attack’) in [113].  $\square$

Note that interactions between many SSLv3 servers and clients are actually protected from the protocol downgrade attack of Exercise 7.4, by an ingenious ‘trick’, designed by Paul Kocher. These clients signal their support of SSLv3, by encoding a ‘signal’ of that in the padding used in the RSA encryption. For details on this and other issues related to MitM version rollback, see [113] and appendix E.2 of RFC2246 (TLS1.0).

### Extended cipher-agility (ciphersuites)

To conclude this subsection, we note that beginning with SSLv3, the ciphersuites supported define additional aspects, mainly, the key-exchange mechanism. Specifically, in addition to the ‘basic’ key-exchange mechanism based on RSA encryption, as in SSLv2, the later versions support also Diffie-Hellman (DH) key-exchange, using either ‘static’ or ‘ephemeral’ DH public keys (§7.3.4), and other key-exchange methods. In the next subsection, we discuss the support in SSLv3 and TLS for DH-based key exchange.

#### 7.3.3 Secure extensibility principle and TLS extensions

We see that the lack of secure version and ciphersuite negotiation in SSLv2, resulted in significant challenge in providing a secure yet backwards-compatible new versions for SSL/TLS. In general, when a security protocol is upgraded, there is a potential conflict between the strong desire to preserve backward compatibility, and the desire to protect against version downgrade attacks. This is one of the many motivations for the important principle of *extensibility by design*, which requires pre-designed, built-in secure mechanisms for extensions and backward compatibility. This is an important design principle for cryptographic systems and protocols.

**Principle 11** (Secure extensibility by design). *When designing a security systems or protocols, one goal must be to build-in secure mechanisms for extensions, downward compatible versions, and negotiation of options and cryptographic algorithms (cipher-agility).*

Note that the extensibility-by-design principle implies that extension mechanisms must be *designed*, not only that they be secure. By designing protocols to be extensible and flexible, we allow the use of the same protocol in many diverse scenarios, environments and applications. This allows more efforts to be dedicated to analyzing the security of the protocol and its implementations. We next discuss the TLS extensions mechanism, which is a great example - in fact, some of the supported extensions have become very popular and had great impact on the adoption of TLS (and migration of web servers from SSL).

### Secure extensibility: version-dependent key separation

While modern protocols like TLS are adopting the extensibility-by-design principle and support secure extensions and downward-compatible versions, there is yet an element of extensibility that is often neglected: *key separation*. Namely, suppose the same key - in particular, public-private key pair - is used by both a vulnerable protocol and a secure protocol. Then, it may be possible to expose the key by running the vulnerable protocol, and exploit this to attack the system also when using the secure protocol. In particular, a significant number of web-servers were found to support SSLv2, with the same key-pair they use for improved-security TLS handshake, opening them to the DROWN vulnerability [4]. We conclude that we need to extend the key-separation principle (principle 5) to also require separate keys for different versions of a protocol.

**Principle 12** (Key-separation (improved)). *Use separate, independent pseudorandom keys for: (1) each different cryptographic scheme, (2) different types and sources of plaintext, (4) different periods, and (5) different versions of the protocol or scheme.*

### The TLS Extensions mechanism

One of the most improvements of TLS over SSL, is that TLS support a flexible and secure *extensions mechanism*. This mechanism allows clients to specify additional fields, not defined in the protocol, but supported (and ‘understood’) by some of the servers. Once a server receives an extension that it supports, its behavior may change from the ‘standard protocol’ in arbitrary way (as defined by the extension); however, servers should ignore any unknown extension.

Extensions were supported as early as TLS1.0, where servers are required to ignore any unknown fields appended beyond the known fields, as defined in [24, 25]. Support for extensions became mandatory from TLS1.1. Some standard extensions facilitate important functionality, and some are needed for security; and users may define additional extensions.

*Server Name Indication (SNI)* is an example of an important , popular extensions; in fact, SNI became mandatory from TLS 1.1, and was of the main factors motivating websites and clients to adopt TLS. SNI is designed to support the common scenario, where the same web server is used to provide web-pages belonging to multiple different domain names, e.g., *a.com* and *b.org*. Each domain name may require a different certificate; the SNI extension allows

the client to indicate the desired server domain name early on in the protocol, before the server has to send a certificate to the client - allowing the server to send the desired certificate based on the web-page that is being requested. Before TLS, using SSL, the common way to a web-server to support multiple web-sites, with different domain names, was by having each site use a dedicated port - an inconvenient and inefficient solution.

#### 7.3.4 SSLv3 to TLSv1.2: DH-based key exchange

From SSLv3, the TLS/SSL handshake supports two methods of DH key exchange: *ephemeral* and *static (certified)*. There is a huge difference in the popularity of the two methods: the ephemeral method is very widely used, while the static (certified) method is rarely used. However, the two methods are actually similar.

In both methods, the parties derive a shared key  $k_{PM}$ , referred to as the *pre-master key*, following the DH protocol. Specifically, TLS/SSL use a modular group, with an agreed upon safe prime  $p$  and generator  $g$ . The parties exchange their ‘public keys’,  $g^{S.x}$  (for the server) and  $g^{C.y}$  (for the client), where each party uses a randomly-generated private key:  $S.x$  for the server and  $C.y$  for the client. The parties then derive the pre-master key  $k_{PM}$ , again as in ‘plain’ DH key exchange, namely:

$$k_{PM} = g^{C.y \cdot S.x} \pmod{p} \quad (7.7)$$

Recall that when using a modular group, the value exchanged by the DH protocol is not pseudorandom; namely, security may rely only on the computational DH assumption (CDH), as we know that the stronger DDH (decisional DH) assumption does not hold for such groups. This is one motivation for not using  $k_{PM}$  directly as a key to cryptographic functions. Instead, we derive from the pre-master key  $k_{PM}$  another key, the *master key*  $k_M$ , which should be pseudorandom. See §7.3.1, where we discuss the derivation of the master key and of the keys for specific cryptographic functions, such as PRF, MAC or shared-key encryption.

**Static (certified) DH handshake.** In static (certified) DH key exchange, the server’s DH public key is signed as part of the signing process of a *public key certificate*. Namely, the signing entity is a *certificate authority* which is trusted by the browser, and the certificates contains the domain name (e.g., s.com) and other parameters such as expiration date:  $S_{CA.s}((g, p, g^{S.x} \pmod{p}), s.com, \dots)$ . See Figure 7.11.

In practice, the use of a certificate implies that the server’s DH public key,  $g^{S.x}$ , is fixed for long periods, similarly to the typical use of RSA or other public key methods. Hence, the static (certified) DH key exchange is similar in its properties to the RSA key exchange; the difference is simply that instead of using RSA encryption to exchange the key, and relying on the RSA (and

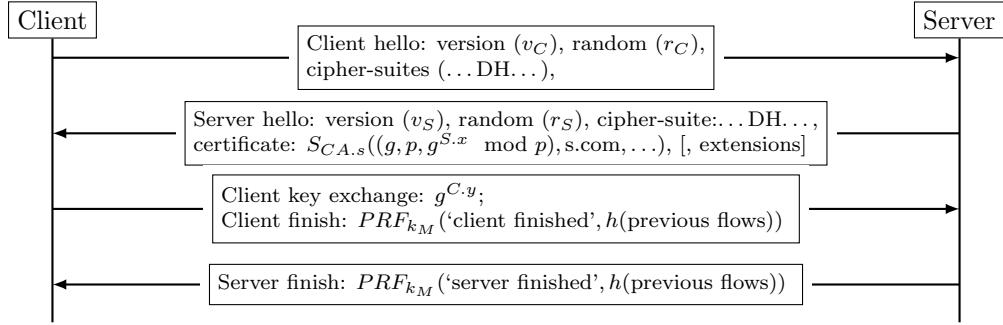


Figure 7.11: SSLv3 to TLSv1.2: handshake with static (certified) DH public key for the server,  $g^{S.x} \bmod p$ . Pre-master key  $k_{PM}$  is computed as in Eq. (7.3), and master key  $k_M$  is computed - from  $k_{PM}$  - as in Eq. (7.3).

factoring) assumptions, the static (certified) DH key exchange relies on the DH (and discrete-logarithm) assumptions.

#### Ephemeral DH handshake: ensuring Perfect Forward Secrecy (PFS).

The *ephemeral* DH key exchange uses a different, randomly-chosen private key for each exchange; for DH, this means that each party selects a new private exponent ( $S.x$  for the server,  $C.y$  for the client) in each handshake. This is illustrated in Figure 7.12.

Once the TLS session terminates, the private exponents are erased - as well as any keys derived from them, including the pre-master key  $k_{PM}$ , the master key  $k_M$ , the derived key block (Eq. (7.4)) and the keys derived from it ( $k_S^A$ ,  $k_S^E$ ,  $k_C^A$ ,  $k_C^E$ ). This ensures *perfect forward secrecy (PFS)*, i.e., the  $i^{th}$  session between client and server is secure against a powerful MitM attacker, even if the attacker is given, all the keys and other contents of the memory of both client and server before and after the  $i^{th}$  session, as long as the keys are given only *after* the  $i^{th}$  handshake is completed.

**Security assumptions of DH key exchange.** An obvious, important difference between the RSA key exchange and the DH key exchange methods, is that instead of using RSA encryption to exchange the key, and relying on the RSA (and factoring) assumptions, the static (certified) DH key exchange relies on the computational-DH (and discrete-logarithm) assumptions. Notice, however, that in the typical case where the certificate uses RSA signatures, the security of the handshake still relies also on the RSA (and factoring) assumptions. Namely, the DH key exchanges require *both* the computational-DH (and discrete logarithm) assumption, *and* the RSA (and factoring) assumption. In this sense, DH key exchanges are ‘less secure’ (or, rely on more assumptions) compared to the RSA key exchange; in the case of ephemeral DH, this is off-

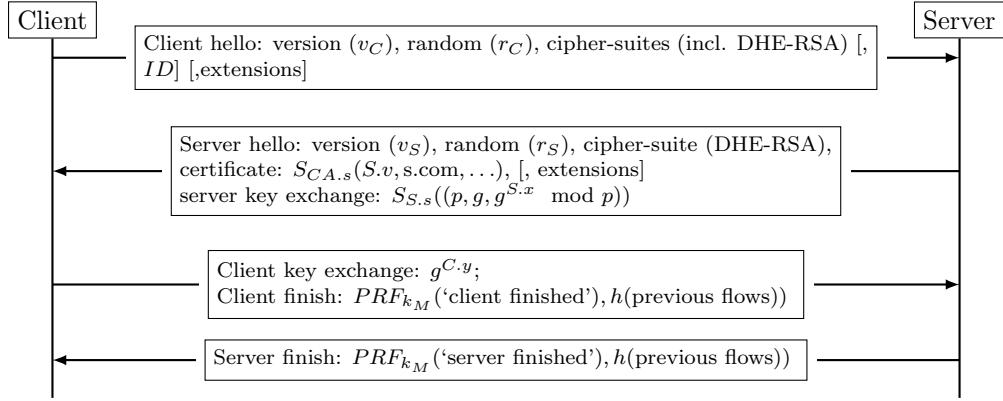


Figure 7.12: SSLv3 to TLSv1.2: handshake with ephemeral DH public keys  $g^{S.x} \bmod p$ ,  $g^{C.y} \bmod p$ , signed using RSA (a DHE-RSA cipher-suite). Pre-master key  $k_{PM}$  is computed as in Eq. (7.3), and master key  $k_M$  is computed as in Eq. (7.3).

set by the security advantage of ephemeral DH handshake, namely, ensuring perfect forward secrecy.

### 7.3.5 SSLv3 to TLSv1.2: session resumption

Both SSLv3 and TLS, like SSLv2, support the (stateful) *ID*-based session resumption mechanism; however, many TLS servers also support extensions, including the *session-ticket* extension, which is an alternative, ‘stateless’ method for session resumption. In this subsection we discuss these two methods.

#### *ID*-based session resumption in SSLv3 and TLS 1.0-1.2

We begin with the (stateful) *ID*-based session resumption mechanism, which did not change much from its implementation in SSLv2.

Figure 7.13 illustrates the handling of *ID*-based session resumption, in the SSLv3 handshake protocol, and in versions 1.0-1.2 of the TLS protocol. In the figure, the client-hello message contains the session-ID, denoted simply *ID*, which was received from server in a previous connection.

Session resumption is possible, when the server still has the corresponding entry (*ID*,  $k_M$ ,  $\gamma$ ) saved from a previous connection; *ID* is the *session identifier*,  $k_M$  is the session’s master key, and  $\gamma$  contains ‘related information’ such as the ciphersuite used in the session.

When the server has the (*ID*,  $k_M$ ,  $\gamma$ ) entry, it reuses  $k_M$  and  $\gamma$ , i.e., ‘resumes the session’, and derives new shared keys from it (using Eqs. (7.1,7.2)). This avoids the public key encryption (by client) and decryption (by server) of master key  $k_M$ , as well as the transmission of the relevant information, most significantly, the public key certificate.

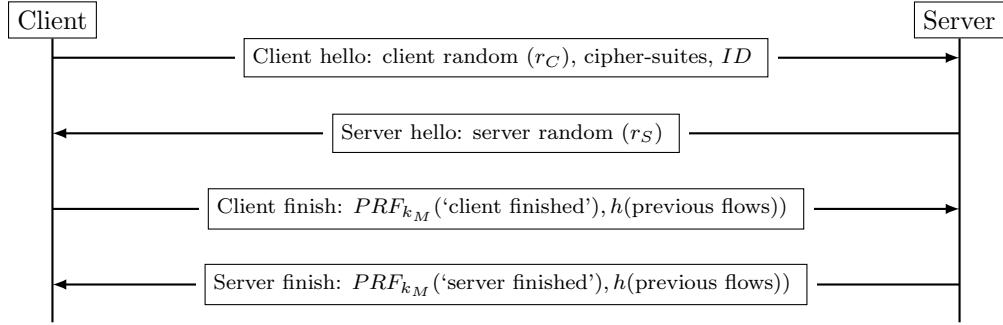


Figure 7.13: SSLv3 to TLS1.2 handshake, with  $ID$ -based session resumption.

Note that when either the client or the server, or both, do not have a valid  $(ID, k_M)$  pair, then the handshake is essentially the same as for a ‘basic’ handshake (without resumption), as in Fig. 7.5. The only changes are the inclusion of the  $ID$  from client (if it has it), and the inclusion of an  $ID$  in the ‘server-finish’ message, to be (optionally) used for future resumption of additional connections (in the same session).

The session resumption mechanism can have a significant impact on performance; in particular, websites often involve opening of a very large number of TCP connections to the same server, to download different objects. The reduction in CPU time can easily be a ratio of dozens or even hundreds. Therefore, this is a very important mechanism; however, it also has some significant challenges and concerns, as we next discuss.

#### Session-ID resumption: challenges and concerns

The basic challenge of ID-based session resumption is the need to maintain state, and lookup the state - and key - using the  $ID$ . To minimize the storage and lookup time overhead, the cache of saved  $(ID, k_M)$  pairs cannot be too large; on the other hand, if the cached is too small, then the resumption mechanism is less effective.

This challenge is made much harder, since web servers are usually replicated - to handle high load and to reduce latency by placing the server closer to the clients, e.g., in a Content Distribution Network (CDN).

**Ensuring PFS with  $ID$ -based session resumption** Another challenge is that the exposure of the master key  $k_M$ , exposes the entire communication of every connection to an eavesdropper; namely, the storage of the key may foil the perfect-forward secrecy (PFS) mechanism. To ensure PFS, we must ensure that all copies of the key  $k_M$  are discarded, without any copies remaining - a non-trivial challenge.

This challenge is often made even harder due to the way that web-server implement the  $(ID, k_M)$  cache. Specifically, in some popular servers, e.g. Apache,

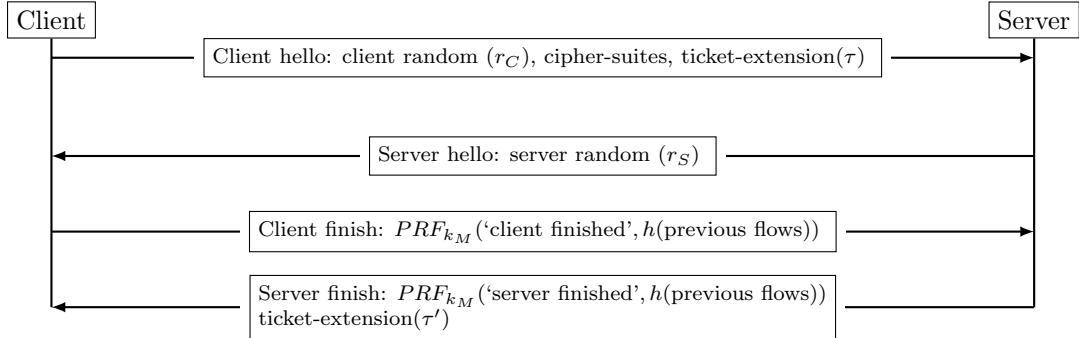


Figure 7.14: Ticket-based session resumption.

the operator can only define the *size* of the  $(ID, k_M)$  cache. Suppose the goal is to ensure PFS on daily basis, i.e., to change keys daily. Then the cache size must be small enough to ensure that entries will be thrown out after at most a day, yet, if it is too small, there will be many cache misses, i.e., the efficiency-gain of the resumption mechanism will be reduced. Furthermore, even if we use a small cache, a client which continues a session for very long time may never get evicted from the cache, and hence we may not achieve the goal of ensuring PFS on daily basis, if the cache uses the (usual) paradigm of throwing out the least-recently-used element; to ensure entries are thrown after one day at most, it should operate as a queue (last-in-first-out).

**Exercise 7.5.** Consider a web server which has, on average, one million daily visitors, but the number in some days may be as low as one thousand. What is the required size of the  $ID$ -session cache, in terms of number of  $(ID, k_M)$  entries, to ensure PFS on daily basis, when entries are removed from the cache only when necessary to make room for new entries? Can you estimate or bound, how many of the connections will be served from cache on a typical day? Assume the  $ID$ -session cache operates using LIFO eviction paradigm.

### The Session-Ticket extension and its use for session resumption

The TLS extensions mechanism provides an alternative, *stateless* session-resumption mechanism. The idea is simple: together with the *finish* message of a successful handshake, the server attaches a *session-ticket extension*. Later, when the client re-connects to the same server, it attaches the previously-received session-ticket extension. See Figure 7.14.

The ticket should allow any of the ‘authorized servers’ (e.g., running the website), to recover the value of the master key  $k_M$  of the session with the client - but prevent attackers, eavesdropping on the ticket as sent by the client, from finding  $k_M$ . This is achieved by having  $k_M$ , and other values sent in the ticket, encrypted using a secret, symmetric *Session Ticket Encryption Key (STEK)*, which we denote  $k_{STEK}$ , known (only) to all authorized servers. Clients cannot

encrypt the tickets; hence, they must store both ticket and (unencrypted) copy of the session’s master key  $k_M$ , to allow the client to perform its part of the handshake.

The contents of the session ticket is only used by the servers, and not opaque to the clients, i.e., not ‘understood’ or used by the clients; hence, different implementations may use different tickets. RFC5077 [100] recommends a structure which uses Encrypt-then-Authenticate, where the encrypted contents include the protocol version, ciphersuite, compression method, master secret, client identity and a timestamp.

**Session tickets and PFS.** To preserve PFS, e.g., on daily basis, we need to make sure that each ticket key  $k_{STEK}$  is kept for only the allowed duration - e.g., up to 24 hours (‘daily’). In principle, this is easy; we can maintain this key only in memory, and never write it to disk or other non-volatile storage, making it easier to ensure it is not kept beyond the desired period (e.g., daily). This rule may require us to maintain several ticket-keys concurrently, e.g., generate a new key once an hour, allowing it to ‘live’ for up to 24 hours.

In the typical case of replicated servers, the ticket keys  $k_{STEK}$  should be distributed securely to all replicates. Changing the key becomes even more important, with it being used in so many machines.

Unfortunately, like for *ID*-based resumption, many popular web-servers implement ticket-based resumption in ways which are problematic for perfect forward secrecy (PFS). These web-server implementations do not provide a mechanism to limit the lifetime of the ticket key, except by restarting the server (to force the server to choose a new ticket key). For some administrators and scenarios, this lack of support for PFS may be a consideration for choosing a server, or for using session-IDs and disabling session-tickets.

### 7.3.6 Client authentication

The SSL and TLS protocols support, already from SSLv2, a mechanism for authenticating client, as an optional service of the handshake. In this subsection we describe how this optional *client authentication* mechanism works, in SSLv3 and in TLS 1.0 to 1.2.

The SSL/TLS client authentication mechanism is illustrated in Figure 7.15. The mechanism consists of tree additions to the ‘basic’ handshake. First, the server signals the need for client authentication, by including the *certificate request* field together with the server-hello message. The certificate-request field identifies the certificate-authorities (issuers) which are accepted by this server; namely, client authentication is possible only if the client has a certificate from one of these entities.

Next, the client attaches, to its *client key exchange* message, two fields. The first is the certificate itself; the second, called *certificate verify*, is a *digital signature* over the handshake messages. The ability to produce this signature, serves as proof of the identity of the client.

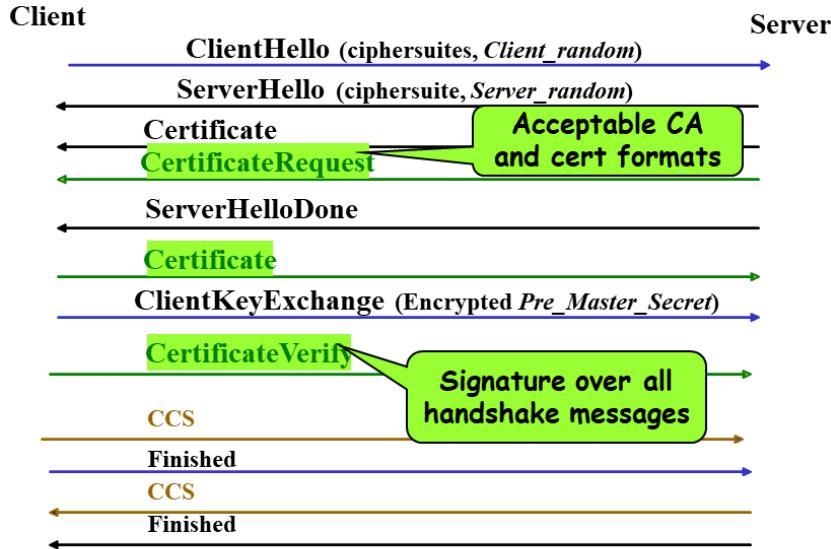


Figure 7.15: Client authentication in SSLv3 to TLS 1.2.

This client authentication mechanism is quite simple and efficient; however, it is not widely deployed. In reality, TLS/SSL is typically deployed using only the public key (and certificate) of the server, i.e., only allowing the client to authenticate the server, but without client authentication. The reason for that is that SSL/TLS client authentication requires clients to use a private key, and to obtain a certificate on the corresponding public key; furthermore, that certificate must be signed by an authority trusted by the server.

This raises two serious challenges. First, clients often use multiple devices, and this requires them to have access to their private keys on these multiple devices, which raises both usability and security concerns. Second, clients must obtain a certificate - and from an authority trusted by the server. As a result, most websites prefer to avoid the use of SSL/TLS client authentication; when user authentication is required, they rely on sending secret credentials such as passwords or cookies, over the SSL/TLS secure connection.

Note also that the client authentication mechanism requires the client to send her certificate ‘in the clear’. This may be a privacy concern, since the certificate may allow identification of the client.

## 7.4 State-of-Art: TLS 1.3

Finally, we briefly discuss TLS 1.3 - the latest version. This version is the result of a major re-design; we only discuss the main aspects related to the

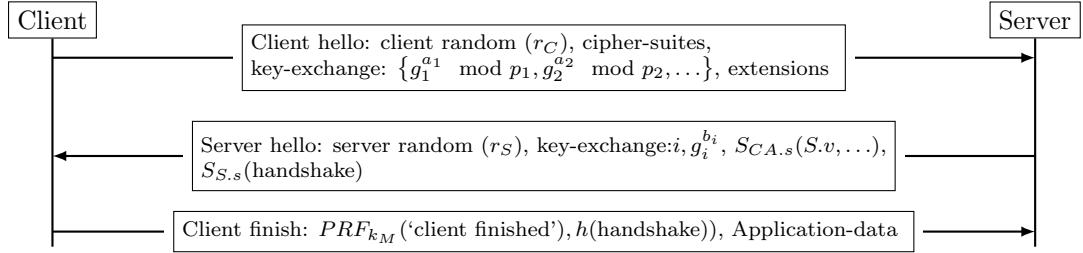


Figure 7.16: TLS 1.3 1-RTT ('full') handshake.

handshake protocol.

One of the main goals of TLS 1.3 has been to *improve (reduce) latency*. This is due to the fact that modern network connections often have very high latency, however, the propagation delays are due to physical limitations, such as the speed of light, and cannot be much reduced; queuing delays are also often significant. Hence, to minimize delay, it is desirable to minimize the number of 'round trips', where a party has to wait for a response before continuing with the protocol.

In Figure 7.16, we present the TLS 1.3 'full, 1-RTT handshake'. In contrast to the 'basic handshake' of the previous sections (and versions), this handshake uses the DH protocol for key exchange. Besides the key exchange, the handshake includes a signature by the server (over its DH exponent).

The TLS 1.3 full (1-RTT) handshake allows the client to send the request after a *single* round-trip (hence, it is called *1-RTT handshake*). Namely, when the server sends its (single) flow, this contains both the server's *hello* message (with the server's DH exponent, extensions, certificate and signature), *and* the server's *finished* message, proving the integrity of the exchange.

Notice that in order to allow the server to send the *finished* message in its (single) flow, the client has to send its key-exchange message, containing its exponentiation (contribution to the DH key exchange), before yet agreeing on the specific parameters (a pair  $(p_i, g_i)$  of prime and generator). The client can do this by sending the exponentiations using multiple  $(p_i, g_i)$  pairs,  $\{g_1^{a_1} \bmod p_1, g_2^{a_2} \bmod p_2, \dots\}$ . There is a cost here in overhead of computing and sending these values - but the savings in RTT are usually much more important.

In Figure 7.18 we present the *Zero Round-Trip-Time (Zero-RTT)* variant of the TLS 1.3 handshake.

*To be completed... session resumption... maybe an exercise on delta from 1.2?*

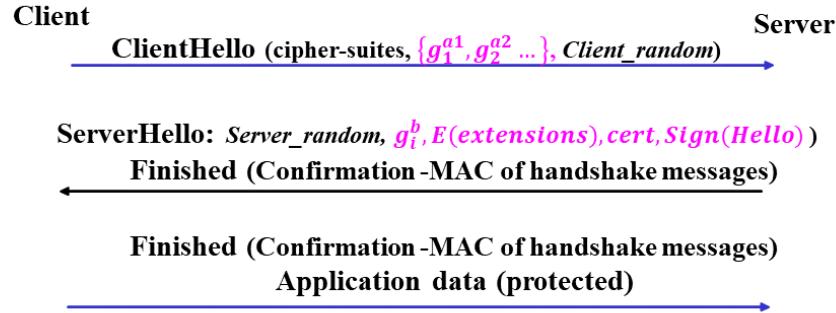


Figure 7.17: TLS 1.3: ‘full handshake’.

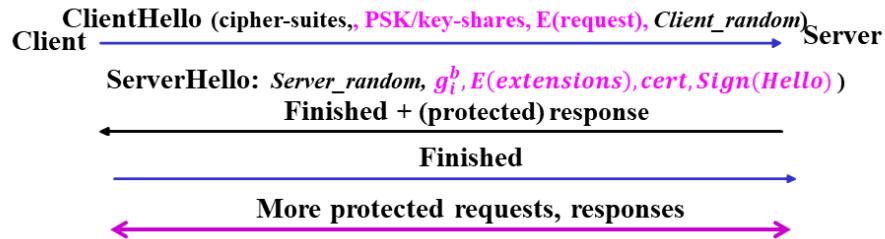


Figure 7.18: TLS 1.3: Zero-RTT handshake.

## 7.5 TLS/SSL: Additional Exercises

- Exercise 7.6** (Record protocol).
1. *SSL and TLS record protocol uses fragments of size up to 16KB. Explain potential disadvantage of using much longer fragments (or no fragments).*
  2. *Explain potential disadvantage of using much shorter fragments.*
  3. *Explain why fragmentation is applied before compression, authentication and encryption.*
  4. *The SSL/TLS record protocols apply authentication and then encryption (AtE). Is it possible to reverse the order, i.e., apply encryption and then authentication (EtA)? Can you identify advantages to AtE and/or to EtA?*

5. The SSL/TLS record protocols apply compression, then authentication. Is it possible to reverse the order, i.e., apply authentication and then compression? Can you identify advantages to either order?

**Exercise 7.7** (SSL 2 key derivation). *SSL uses MD5 for key derivation. In this question, we explore the required properties from MD5 for the key derivation to be secure.*

1. Show that it is not sufficient to assume that MD5 is collision-resistant, for the key derivation to be secure.
2. Repeat, for the one-way function property.
3. Repeat, for the randomness-extraction property.
4. Define a simple assumption regarding MD5, which ensures that key derivation is secure. The definition should be related to cryptographic functions and properties we defined and discussed.

**Exercise 7.8** (TLS handshake: resiliency to key exposure). *Fig. 7.10 presents the RSA-based SSL/TLS Handshake. This variant of the handshake protocol was popular in early versions, but later ‘phased out’ and completely removed in TLS 1.3. The main reason was the fact that this variant allows an attacker that obtains the server’s public key, to decrypt all communication with the server using this key - before and after the exposure.*

1. Show, in a sequence diagram, how a MitM attacker who is given the private key of the server at time  $T_1$ , can decrypt communication of the server at past time  $T_0 < T_1$ .
2. Show, in a sequence diagram, how TLS 1.3 avoids this impact of exposure of the private key.
3. Show, in a sequence diagram, how a MitM attacker who is given the private key of the server at time  $T_1$ , can decrypt communication of the server at future time  $T_2 > T_1$ .
4. Explain which feature of TLS 1.3 can reduce the exposure of future communication, and how.

**Exercise 7.9** (Protocol version downgrade attack). *Implementations of SSL specify the version of the protocol in the ClientHello and ServerHello messages. If the server does not support the client’s version, then it replies with an error message. When the client receives this error message (‘version not supported’), it re-tries the handshake using the best-next version of TLS/SSL supported by the client.*

*Present a sequence showing how a MitM attacker can exploit this mechanism to cause the server and client to use an outdated version of the protocol, allowing the attacker to exploit vulnerabilities of that version.*

**Exercise 7.10** (Client-chosen cipher-suite downgrade attack). *In many variants of the SSL/TLS handshake, e.g., the RSA-based handshake in Fig. 7.10, the authentication of the (previous) handshake messages in the Finish flows, is relied upon to prevent a MitM attacker from performing a downgrade attack and causing the client and server to use a less-preferred (and possibly less secure) cipher-suite. However, in this process, the server can choose which of the client's cipher-suites would be used. To ensure the use of the cipher-suite most preferred by the client, even if less preferred by the server, some client implementations send only the most-preferred cipher-suites. If none of these is acceptable to the server, then the server responds with an error message. In this case, the client will try to perform the handshake again, specifying now only the next-preferred cipher-suite(s), and so on - referred to as 'downgrade dance'.*

1. Show how a MitM attacker can exploit this mechanism to cause the server and client to use a cipher-suite that both consider inferior.
2. Suggest a fix to the implementation of the client which achieves the same goal, yet is not vulnerable to this attack. Your fix should not require changes in the handshake protocol itself, or in the server.
3. Suggest an alternative fix, which only involves change in the handshake, and does not require change in the way it is used by the implementation.

**Exercise 7.11** (TLS server without randomness). *An IoT device provides http interface to clients, i.e., acts as a tiny web server. For authentication, clients send their commands together with a secret password, e.g., on, `[password]`, and off, `[password]`. Communication is over TLS for security, with the RSA-based SSL/TLS handshake, as in Figure 7.10.*

*The IoT device does not have a source of randomness, hence, it computes the server-random  $r_S$  from the client-random, using a fixed symmetric key  $k_S$  (kept only by the device), as:  $r_S = AES_{k_S}(r_C)$ .*

1. Present a message scheduling diagram showing how an attacker, which can eavesdrop on a connection in which the client turned the device 'on', can later turn the device 'on' again, without the client being involved.
2. Would your answer change (and how), if the device supports ID-based session resumption? Ticket-based session resumption?
3. Show a secure method for the server to compute the server-random method, which will not require a source of randomness. The IoT device may use and update a state variable  $s$ ; your solution consists of the computation of the server-random:  $r_S = \underline{\hspace{1cm}}$  and of the update to the state variable performed at the end of every handshake:  $s = \underline{\hspace{1cm}}$ .

**Exercise 7.12.** Consider a client and server that use TLSv1.2 with ephemeral DH public keys, as in Fig. 7.12. Assume that the client and server run this

protocol daily, at the beginning of every day  $i$ . (Within each day, they may use session resumption to avoid additional public key operations; but this is not relevant to the question). Assume that Mal can (1) eavesdrop on communication every day, (2) perform MitM attacks (only) every even day ( $i$  s.t.  $i \equiv 0 \pmod{2}$ ), (3) is given all the keys known to the server on the fourth day. Note: the server erases any key once it is not longer in use (i.e., on fourth day, attacker is not given the ‘session keys’ established  $n$  previous days).

Fill the ‘Exposed on’ column of day  $i$  in Table 7.12, indicating the first day  $j \geq i$  in which the adversary should be able to decrypt (expose) the traffic sent on day  $i$  between client and server. Write ‘never’ if the adversary should never be able to decrypt the traffic of day  $i$ . Briefly justify.

Day	Eavesdrop?	MitM?	Given keys?	Exposed on...	Justify
1	Yes	No	No		
2	Yes	Yes	No		
3	Yes	No	No		
4	Yes	Yes	Yes		
5	Yes	No	No		
6	Yes	Yes	No		
7	Yes	No	No		
8	Yes	Yes	No		

**Exercise 7.13** (TLS with PRS). Consider a client that has three consecutive TLS connections to a server, using TLS 1.3. An attacker has different capabilities in each of these connections, as follows:

- In the first connection, attacker obtains all the information kept by the server (including all keys).
- In the second connection, attacker is disabled.
- In the third connection, attacker has MitM capabilities.

Is the communication between client and server exposed, during the third connection?

1. Show a sequence diagram showing that with TLS 1.3, communication during third connection is exposed to attacker.
2. Present an improvement to TLS 1.3 that will protect communication during third connection.
3. Further to provide same protection, even if attacker can eavesdrop to communication during the second connection.
4. How can your improvement be implemented using TLS 1.3, allowing ‘normal’ TLS 1.3 interaction if client/server do not support your improvement?

**Exercise 7.14.** A Pierpont prime is a prime number of the form  $2^u \cdot 3^v + 1$ , where  $u, v$  are non-negative integers; Pierpont primes are a generalization of Fermat primes. Assume that the client's key exchange sent in the Client Hello message of TLS 1.3 includes  $\{g_i^{a_i} \bmod p_i\}$ , where for some  $i$ , say  $i = 3$ , the prime  $p_3$  is a Pierpont prime.

1. Assume that the server selects to use this  $p_3$ , i.e., sends back  $g_3^{b_3} \bmod p_3$  (as part of the server-hello message, and signed). Explain how a MitM attacker would be able to eavesdrop and modify messages sent between Alice and Bob.
2. Assume that the server prefers  $p_2$ , which is a safe prime. Show that the attacker is able to (adapt the attack and still) eavesdrop/modify messages between Alice and Bob.
3. Would this attack be possible, if Alice authenticates to the website (1) using userid-pw, or (2) using TLS client authentication ? Explain.

## Chapter 8

# Public Key Infrastructure (PKI)

### 8.1 Public Key Certificates and Infrastructure

As we have seen in the two preceding chapters, public keys have many important applications. In particular, some of the applications where public key is widely deployed and/or standardized include: (a) securing web connections, (b) signing software (to detect fake, modified versions), (c) secure messaging, including email, and (d) crypto-currencies, block-chains and other financial-cryptography applications.

However, to safely use public key cryptography, in realistic scenarios where attackers may have Man-in-the-Middle (MitM) capabilities, we must properly authenticate public keys before using them. Often, the most convenient way to authenticate the public key of a ‘subject’ entity is to send a *public key certificate*, signed by a *certificate authority (CA)*.

Figure 8.1 illustrates the basic PKI entities and interactions. The simple scenario in this figure includes three parties: the *Certificate Authority (CA)*, also referred to as the *issuer*, the subject of the certificate, i.e., the entity whose public key is certified, and the relying party, i.e., the party who receives the subject’s certificate in order to validate the subject’s public key. In this example, the subject is the website *bob.com*, and the relying party is Alice’s browser. The certificate is a signature by the private signing key of the CA, *CA.s*, over the subject’s identity, in this case the domain name *bob.com*, and the subject’s public key, in this case the public encryption key *bob.e*. The certificate is computed by the CA, using its private signing key *CA.s*, and provided to the relying party (*bob.com*), allowing the subject to forward it to the relying party.

Indeed, in the SSL/TLS protocol as implemented by web browsers, the clients (e.g., browsers) receives a certificate authenticating the server’s public key, and binding it to the server’s domain; this allows SSL/TLS clients to authenticate the server and establish shared key. The certificate must be signed by a trusted certificate authority (CA); each browser maintain a list of trusted (‘root’) certificate authorities. Many Internet domains obtained certificates from one of these CAs, mainly for use in SSL/TLS.



Figure 8.1: PKI Entities, for the simple case of certificate directly issued by a trusted CA ('trust anchor'). Indirectly-issued certificate use an intermediate-CA (Figure 8.6), or a path of multiple intermediate-CAs (Figure 8.7).

*Note that SSL/TLS also supports (optional) client authentication; however, SSL/TLS client authentication requires a client certificate - and only few clients have these client certificates. Similarly, client certificates are required for end-to-end secure email services, e.g., using S/MIME [94]; but, again, only a tiny fraction of the users went through the process of obtaining a client certificate, and as a result, these secure email services are not widely used. The difficulties of obtaining client certificates are probably one reason for the fact that secure messaging applications rely on authentication by the provider, and sometimes also by the peer user, but not on client certificates.*

We next define public-key certificates and related notions, to further clarify these critical concepts.

**Definition 8.1** (Public key certificate). *Let  $(S, V)$  be a signature scheme, and  $(CA.s, CA.v)$  be the (private signing, public verification) keys of an entity we call certificate authority (CA). We sometimes use the term issuer as a synonym for CA.*

A public key certificate is a triplet  $(pk, attr, \sigma)$  where  $pk$  is a public key,  $attr$  is a set of attributes and  $\sigma$  is the CA's signature (i.e., using  $CA.s$ ) over  $pk$  and  $attr$ , i.e.:

$$\sigma = S_{CA.s}(pk || attr), V_{CA.v}(pk || attr, \sigma) = \text{TRUE}$$

*Terminology.* Certificate is a shorter version of public key certificate. We use the terms subject to refer to the entity who knows ('owns') the private key corresponding to the public key in the cert, and relying party to refer to a party receiving the certificate (and deciding whether to rely on it or not). We also use the term Certificate as abbreviation of public key certificate.

2001	VeriSign: attacker gets code-signing certs
2008	Thawte: email-validation (attackers' mailbox)
2008,11	Comodo not performing domain validation
2011	DigiNotar compromised, over 500 rogue certs discovered, incl. wildcard cert for Google (used for mass-interception in Iran)
2011	TurkTrust issued intermediate-CA certs to users
2014	India CCA / NIC compromised (and issued rogue certs)
2015	Root CA CNNIC (China) issued CA-cert to MCS (Egypt), who issued rogue certs
2015,17	Symantec issued unauthorized certs for over 176 domains

Table 8.1: Some PKI failures.

In this chapter, we discuss the *Public Key Infrastructure (PKI)*. The PKI is a set of conventions and protocols for public key certificates, and its main function is to determine *which CA is trustworthy - and which certificate is valid*. PKI is, therefore, an essential component for practical deployments of PKC, and its most important application is, probably, the TLS/SSL handshake protocols, presented in chapter 7.

The main goal of PKI schemes is to determine if to trust CAs, certificates and specific attributes specified in a public key certificate. In the first years of deploying PKI, many practitioners expected that CAs would all be highly trustworthy and secure organizations, motivating the use of ‘static PKI’, with a fixed list of trusted *root CAs*. However, as the use of SSL/TLS to secure web traffic became popular, there was growing reliance on CAs and certificates, and, even more significantly, there was an alarmingly, growing list of *PKI failures*, allowing attackers to obtain many fake certificates. See Table 8.1.

The security of the PKI is critical; if an attacker can get fake keys certified, then she could impersonate as a legitimate entity, even if the protocols use cryptography correctly. The multiple failures of the current PKI system, e.g., in Table 8.1, caused renewed interest in PKI schemes, with new proposals, designs and deployments with additional security goals and mechanisms.

In this chapter, we first introduce the ‘traditional’ PKI approach of X.509 and PKIX. Then, we focus on the challenge of *certificate revocation*, and the solutions in X.509/PKIX as well as complementing solutions. We then discuss different approaches for dealing with corrupt certificate authorities (CAs), including the *Certificate Transparency (CT)* standard-track proposal. Finally, we discuss additional challenges and conclusions.

## 8.2 Basic PKI Concepts from the X.509 Standard

In this section, we discuss the basic notions of the X.509 PKI standard, which was developed as part of the X.500 global directory standard. X.509 is the most widely deployed PKI specifications, and also includes some of the more advanced PKI concepts which we cover in the following sections.

### 8.2.1 The X.500 Global Directory Standard

X.500 [?] is an ambitious, extensive set of standards of the *International Telecommunication Union (ITU)*, a United Nations agency whose role is to facilitate international connectivity in communications networks. The goal of X.500 is to facilitate the interconnection of *directory services* provided by different organizations and systems. The first version of X.500 was published as early as 1988, and numerous extensions and updates were published over the years.

The basic idea of X.500 is to provide a *trusted, unified* and ideally *global* directory, by combining the data and services of its multiple component directories. Such a unified directory would be operated by interoperability of trustworthy providers, such as telcos. While X.500 was deployed by some systems, its deployment is quite limited, and definitely far from the vision of a global directory. Among the possible reasons for the limited deployment are the high complexity of the X.500 design, concerns that X.500 interoperability may cause exposure of sensitive information, and lack of sufficient trust among different directory providers.

However, some concepts from X.500 live on; in particular, X.500 contributed extensively to the development of PKI schemes. The X.500 designers observed that an interoperable directory should bind *standard identifiers* to *standard attributes*.

One important set of attributes define the *public key(s)* of each entities. The entity's *public encryption key* allows relying parties to *encrypt* messages so that only the intended recipient may decrypt them. Similarly, the entity's *public validation key* allows relying parties to *validate* statements *signed* by the entity.

We next discuss the main form of *standard identifier* defined in X.500: the *distinguished name*.

### 8.2.2 The X.500 Distinguished Name

The X.500 directory standard, has built extensively on the experience of telecoms with provision of directory services to phone users. Phone directory services are mostly based on looking up the person's *common name*; the common name has the obvious advantage of being a *meaningful identifier* - we usually know the common name of a person when we ask the directory for that person's information. Phone directories would normally also allow specification of the relevant *area*, e.g., in form of *locality*; by limiting search to specific areas or localities, the directory services can be *decentralized*.

However, obviously, a common name is not a unique identifier - in fact, some common names are quite common, if you excuse the pun. In classical phone directories, this is addressed by returning a set of results containing all relevant entries, along with the relevant common-name and other attributes (e.g., location).

The X.500 designers decided that in order to allow efficient use of large, global directories, returning multiple results is not a viable option. Instead,

C	Country
L	Locality
O	Organization name
OU	Organization unit
CN	Common name

Table 8.2: Standard keywords/attributes in X.500 Distinguished Names

they decided to use a more refined identifier, with multiple keywords - where the common name will simply be one of these keywords. This identifier is the X.500 *Distinguished Name (DN)*. The distinguished name was designed to satisfy the following three main *goals for identifiers*:

**Meaningful:** identifiers should be meaningful and recognizable by humans.

This make it easier to memorize the identifier, as well as to link it with off-net identifier, with potential legal and reputation implications.

**Unique:** identifiers should be unique, i.e., different subjects should have different identifiers, allowing identiers to be mapped to a specific subject.

**Decentralized management:** identifiers can be assigned by multiple, different issuers.

The uniqueness requirement is an obvious challenge, as common names are obviously not unique. To facilitate unique DNs for people sharing the same common name, X.500 distinguished names consist of a *sequence of several keyword-value pairs*. The inclusion of multiple keywords - also referred to as attributes - helps to ensure unique identification, when combined with the common name. Typical, standard keywords are shown in Table 8.2; however, a directory is free to use any keyword it desires.

To satisfy the ‘meaningful’ goal, identifiers should have readable representations. RFC 1779 [72] specifies a popular string representation for distinguished names, where keyword-value pairs are separated by the equal sign, and different pairs are separated by comma or semicolon. Other representations are possible too, e.g., Figure 8.2 includes encoding of a DN using slash for separation.

Using the RFC 1779 representation, the distinguished name for an IBM UK employee with the name Julian Jones may be written as:

$$CN=Julian\ Jones,\ O=IBM,\ C=GB$$

Note that keyword-value pairs comprising an X.500 distinguished name are specified in a sequence, i.e., as an *ordered* list. This allows the distinguished names to be organized as a hierarchy, using the sequence of keywords as the nodes, as illustrated in Figure 8.2. By assigning a specific, single entity to assign identifiers in a subtree of the X.500 DN hierarchy, this entity can ensure

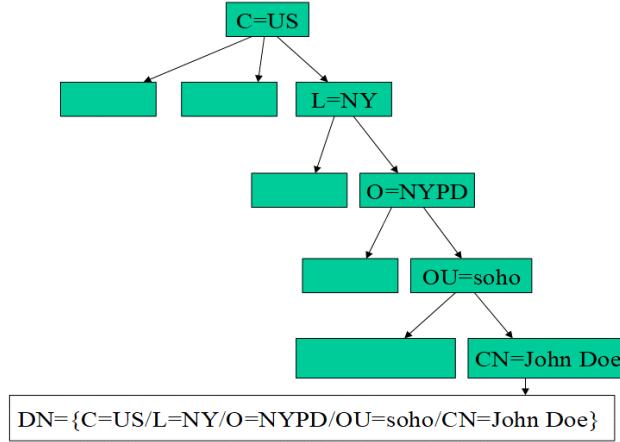


Figure 8.2: Example of the Distinguished Name hierarchy.

*uniqueness* by never allocating the same identifier (DN) to two different subjects, e.g., the Soho precinct of the NYPD may maintain its own sub-directory. This also allows queries over the entire set of distinguished names which begin with a particular prefix of keyword-value pairs.

However, this implies that X.500 distinguished names cannot be issued in an entirely decentralized manner - some control and coordination is required. Furthermore, there are also some caveats with respect to the other goals - unique and meaningful identifiers.

Let us first consider the goal of meaningful identifiers. The use of subdivisions such as ‘organization unit (OU)’ may help to reduce the likelihood of two persons with the same common name in the same ‘bin’, this possibility still exists. As a result, administrators may have to enforce uniqueness by ‘modifying’ the common name. For example, if there are multiple IBM UK employees with the name Julian Jones, one of them may be assigned the DN:

*CN=Julian Jones2, O=IBM, C=GB*

This results in *less meaningful distinguished names*; e.g., it is easy to confuse between the DNs of the two employees. For example, the author has sent to *CN=Julian Jones, O=IBM, C=GB* messages intended for *CN=Julian Jones2, O=IBM, C=GB*. Luckily, they were both understanding of the mistake.

Another cause of mistakes and ambiguity is there are no rules governing the *order* of the keywords, i.e., the structure of the hierarchy. For example, some multinational organizations may use the country as the top level category, as in *CN=Julian Jones, O=IBM, C=GB*, while others may view the organization itself as the top-level category, as in *CN=Julian Jones, C=GB, O=IBM*. These two distinguished names are *different*; this distinction may not be obvious to a non-expert, further reducing from the goal of ‘meaningful’ names.

There are also cases where uniqueness is not ensured. Some namespaces are shared by design, and cannot be segregated with a single authority assigning identifiers in each segment. For example, consider Internet domain names; multiple registrars are authorized to assign names in several top-level domains such as `com` and `org`. There is a coordination process between registrars, but if not followed correctly, conflicts may occur.

This problem is more severe with respect to public key certificates for Internet domain names, which can be issued by multiple Certificate Authorities; any faulty authority may issue a certificate to an entity who does not rightfully own the certified domain name. Such incidents occurred - often due to intentional attack; e.g., see Table 8.1. This is a major concern for PKI, and we discuss it further later in this chapter.

We conclude that X.500 distinguished names are not perfectly *meaningful* and *decentralized*, and, sometimes, may even not perfectly ensure *uniqueness*. Indeed, there seem to be an inherent challenge in satisfying all three goals, although achieving any two of these three properties is definitely feasible - a classical trilemma scenario.

### The identifiers trilemma

We argued that X.500 distinguished names may fail to ensure each of the three goals - *uniqueness*, *meaningfulness* and *decentralized management*. In contrast, several other identifiers ensure pairs of these three properties:

**Common names** are meaningful - and decentralized, as any person can decide on the name. However, they are definitely not unique.

**Public keys and random identifiers** are unique and decentralized, but obviously not meaningful to humans.

**Email address** are unique and meaningful. They may also appear to be decentralized, as there are multiple entities issuing them; however, each entity can only issue in a separate part of the name space, i.e., not all identifiers

This begs the question: is there a scheme which will ensure identifiers which fully satisfy all three properties, i.e., would be unique, meaningful and decentrally managed and issued? It seems that this may be hard or impossible, i.e., it may be possible to only fully ensure two of these three goals, but not all three. We refer to this challenge as the *Identifiers Trilemma*<sup>1</sup>, and illustrate it in Figure 8.3.

---

<sup>1</sup>This challenge is also referred to as *Zooko's triangle*; however, Zooko has apparently referred to a different trilemma, albeit also related to identifiers. Specifically, Zooko considered the challenge of identifiers which will be distributed, meaningful for humans, and also *self-certifying*, allowing recipients to locally confirm the mapping from name to value.

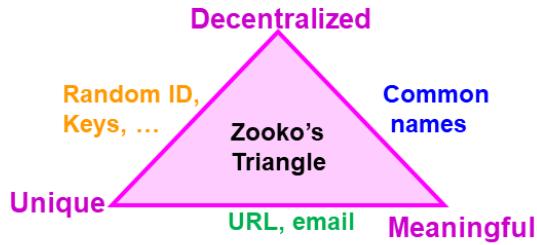


Figure 8.3: The Identifiers Trilemma: the challenge of co-ensuring unique, decentralized and meaningful identifiers.

#### Additional concerns re X.500 Distinguished Names

We conclude our discussion of X.500 distinguished names, by discussing few additional concerns.

*Privacy.* The inclusion of multiple categorizing fields in X.500 DNs, may expose information in an unnecessary, and sometimes undesired, manner. For example, employees may not always want to expose their location or organizational unit.

*Flexibility.* People may change locations, organization units and more; with X.500 DNs, this may result in ‘incorrect’ DN, or require change of the DN - both undesirable.

*Usability.* X.500 DNs are designed to be *meaningful*, i.e., users can easily *understand* the different keywords and values. However, sometimes this may not suffice to ensure *usability*. In particular, consider two of the most important applications for public key cryptography and certificates: secure web-browsing and secure email/messaging.

**Secure web-browsing:** users, as well as hyperlinks, specify the desired website using an Internet domain name, and not a distinguished name. Hence, the relevant identifier for the web site is that domain name - provided by the user or in the hyperlink. This requires mapping from the domain name to the distinguished name. A better solution is for the certificate to directly include the domain name; this is supported by the PKIX standard, explained below.

**Secure email/messaging:** users also do not use distinguished names to identify peers with whom they communicate using email and instant messaging applications. Instead, they use email addresses - or application-specific identification. This problem may not be as meaningful, since most end users do not have a public key certificate at all; and, again, PKIX allows certificates to directly specify email address.

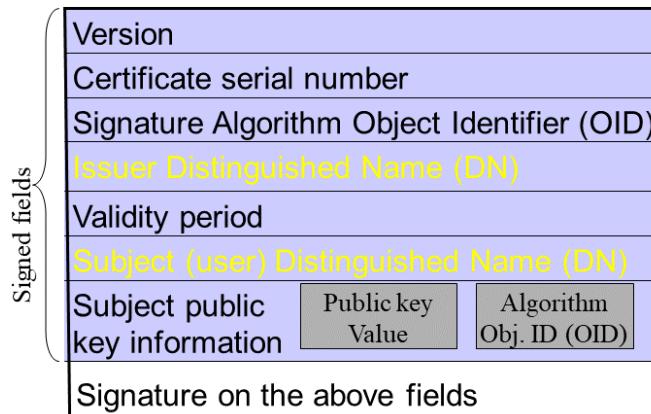


Figure 8.4: X.509 version 1 certificate.

### 8.2.3 X.509 Public Key Certificates

The X.500 standard included a dedicated sub-standard, X.509, which defined *authentication mechanisms*, allowing entities to authenticate themselves to the directory. X.509 defined multiple authentication mechanisms, e.g., the use of password based authentication. However, one of these authentication methods became a very important, widely used standard: the *X.509 public key certificate*.

Originally, the main goal of the X.509 authentication was to allow each entity to maintain its record with the directory, e.g., to change address. However, it was soon realized that public key certificates allow many more applications, since they allow recipients to authenticate the public key of a party without requiring any prior communication. As a result, X.509 certificates became a widely deployed standard, which is used for SSL/TLS, code-signing, secure messaging (PGP, S/MIME), IPsec and more - and all that, in spite of complaints about the complexity of the X.509 specifications and encoding formats.

The definition of the X.509 certificates did not change too much from the first version of X.509; the contents (fields) of that first version of X.509 are shown in Figure 8.4. These fields are, by their order in the certificate:

**Version:** the version of the X.509 certificate and protocol.

**Certificate serial number:** a serial number of the certificate, incremented by the issuing CA whenever it signs an certificate.

**Signature-process Object Identifier (OID):** this is an identifier of the process used for signing the certificate, typically using the ‘hash then sign’ design. This identifier specifies both the underling public key signature algorithm, e.g., RSA, as well as the hash algorithm, e.g. SHA-256. The algo-

rithm may be written as a string for readability, and standard string terms are used for widely used methods, e.g., `sha256WithRSAEncryption`; notice the use of the term ‘RSA Encryption’ when referring to RSA signatures - a common misnomer. In the certificate itself, the algorithm is typically specified using the *Object-Identifier (OID)* standard; see note 8.1.

**Issuer Distinguished Name:** the distinguished name of the certificate authority which issued, and signed, the certificate.

**Validity period:** the period of time during which the certificate is to be considered valid.

**Subject Distinguished Name:** the distinguished name of the *subject* of the certificate, i.e., the entity to whom the certificate was issued. This entity is expected to know the private key corresponding to the certified public key.

**Subject public key information:** this field contains two sub-fields. The first simply specifies the public key of the subject. The second field identifies the allowed *usage* of the certified public key - e.g., to encrypt messages sent to the subject, or to validate signatures by the subject.

**Signature:** finally, this field contains the result of the application of the signature algorithm (identified by the OID field above), to all of the other fields in the certificate, using the private signing key of the issuer (certificate authority), and applying the signature process identified in the certificate. The sequence of all these fields in the certificate, excluding the signature field itself, is referred to as the *to-be-signed* fields; see Figures 8.4,8.5. This allows the *relying party* to validate the authenticity of the fields in the certificate, e.g., the validity period, the subject distinguished name and the subject public key.

**Exercise 8.1.** Explain why the signature process is specified as one of the (signed) fields within the certificate. Do this by constructing two ‘artificial’ CRHFs,  $h^A$  and  $h^B$ ; to construct  $h^A$  and  $h^B$ , you may use a given CRHF  $h$ . Your constructions should allow you to show that it could be insecure to use certificates where the signature process (incl. hashing) is not clearly identified as part of the signed fields. Specifically, design  $h^A$ ,  $h^B$  to show how an attacker may ask a CA to sign a certificate for one name, say Attacker, and then use the resulting signature over the certificate to forge a certificate for a different name, say Victim.

### X.509 Certificates after version 1

Following X.509 version 1, the X.509 certificates were extended by few additional fields; see Figure 8.5.

Note 8.1: Object identifiers (OIDs)

An *object identifier (OID)* is a standard way for uniquely referring to arbitrary objects, standardized by the ITU and ISO/IEC. Object identifiers are specified as a sequence of numbers, e.g., 1.16.180.1.45.34. OID numbers are assigned hierarchically to organizations and to ‘individual objects’; when an organization is assigned a number, e.g., 1.16, it may assign OIDs whose prefix is 1.16 to other organizations or directly to objects, e.g. 1.16.180.1.45.34. The top level numbers are either zero (0), allocated to ITU, 1, allocated to ISO, or 2, allocated jointly to ISO and ITU. RFC 3279 [8] defines OIDs for many cryptographic algorithms and processes used in Internet protocols, e.g., RSA, DSA and elliptic-curve signature algorithms; when specifying a signature process, the OID normally also specifies both the underlying public key signature algorithm and key length, e.g., RSA-2048, and the hashing function, e.g. SHA-256, used to apply the ‘hash-then-sign’ process. X.509 uses OIDs to identify signature algorithms and other types of objects, e.g., extensions and issuer-policies. The use of OIDs allows identification of the specific type of each object, which helps interoperability between different implementations.

Signed fields	<b>Version</b>
	<b>Certificate serial number</b>
	<b>Signature Algorithm Object Identifier (OID)</b>
	<b>Issuer Distinguished Name (DN)</b>
	<b>Validity period</b>
	<b>Subject (user) Distinguished Name (DN)</b>
	<b>Subject public key information</b>
	Public key Value
	Algorithm Obj. ID (OID)
	<b>Issuer unique identifier (from version 2)</b>
	<b>Subject unique identifier (from version 2)</b>
	<b>Extensions (from version 3)</b>
<b>Signature on the above fields</b>	

Figure 8.5: X.509 version 3 certificate.

Version 2 of X.509 added two fields, both of them for unique identifiers - one for the subject and one for the issuer (CA). These fields were defined to ensure uniqueness, in situations where the distinguished name may fail to ensure uniqueness, as discussed in subsection 8.2.2. However, these unique identifier fields are not in wide use, as they are entirely unrelated to the meaningful identifiers used in typical applications.

Version 3 of X.509 added ‘just’ one field for extensions. However, in reality, the extensions are a very important part of the X.509 certificates and are used

for many application. Note also that the extensions are usually much longer than all the rest of the certificate. We discuss the extensions next.

#### 8.2.4 The X.509 Extensions Mechanism

As shown in Figure 8.5, X.509 certificates, from version 3, include a field that can contain one, or more, extensions. Extensions are specified by three simple fields, which we describe in this subsection; in the following subsections, we discuss some specific, important extensions.

Each extension is specified, using the following three fields:

**Extension identifier:** specifies the type of the extension. The extension identifier is specified using an object identifier (OID), to facilitate interoperability. The following subsections discuss some important extensions, e.g., *key usage* and *naming constraints*.

**Extension value:** this is an arbitrary string which provides the value of the extension. For example, a possible value for the key-usage extension would indicate that the certified key is to be used as a public encryption key, which a possible value for the naming constraints extension may be *Permit C=GB*, allowing the subject of the certificate to issue its own certificates, but only with the value ‘GB’ (Great Britain) to their ‘C’ (country) keyword.

**Criticality indicator:** this is a binary flag, i.e., an extension can be marked as ‘critical’ or as ‘non-critical’. The value of the criticality indicator flag in an extension, instructs relying parties, how to handle the certificate, if the relying party is not familiar with this type of extension, as indicated by the extension identifier. A relying party must never use a certificate, which include an extension marked as critical - and unknown to this relying party. Relying parties should use certificates with unknown extension types, if they are marked as ‘non-critical’. When the relying party is familiar with the type of an extension, the value of the criticality indicator is not applicable.

The *criticality indicator* flag is a simple mechanism - but very valuable, by allowing both *critical extensions* and non-critical extensions. Both type of extensions may be required, when extending a standard protocol - in this case, a certificate. Indeed, some extensions are always marked ‘critical’, others are always marked ‘non-critical’, and others are marked differently depending on needs. Note that X.509 often allows an extensions to be marked as either ‘critical’ or ‘non-critical’, even when PKIX specifies that the extension should be marked as ‘critical’ (or as ‘non-critical’). Few examples follow:

**Always critical example:** PKIX specifies that the ‘key usage’ extensions must be marked ‘critical’, since a relying party which is not familiar with its meanings may do critical security mistakes. For instance, such relying party may use a signature-validation key to encrypt messages.

However, X.509 allows this extension to be marked as either ‘critical’ or ‘not-critical’.

**Always non-critical example:** an X.509 extension called *TLS feature* is defined in RFC 7633 [62]. This extension, in the server certificate, indicates that the server supports specific TLS ‘feature’. The term ‘feature’ here is used to refer to a specific TLS extension (see subsection 7.3.3); we use the term ‘feature’ instead of ‘extension’, to avoid confusion with X.509 extensions. The ‘TLS feature’ X.509 extension allows the server to indicate to the client that the server supports certain important TLS features; however, some clients may not support this extensions, so if it would be marked ‘critical’, they would have rejected the certificate and the connection would fail. Hence, this X.509 extension should be marked ‘non-critical’. Note that this extension isn’t one of the standard extensions defined in either X.509 or PKIX.

**Per-application critical/non-critical example:** the ‘extended key usage’ extension allows the issuer to define allowed usage for the public key certified, which is in addition or in place of the usage specified in the ‘key usage’ extension. In some scenarios, the ‘extended key usage’ should be ‘critical’, e.g., to prevent incorrect usage based on the ‘key usage’ extension, by clients not supporting ‘extended key usage’. In other scenarios, the ‘extended key usage’ should be ‘non-critical’, e.g., when allowing some additional usage over that specified already in the ‘key usage’ extension.

The flexibility offered by the ‘criticality indicator’ makes the X.509 extensions mechanism much more versatile and useful; it is a pity that this idea has not been adopted by many other extension mechanisms, e.g., for TLS extensions.

**Exercise 8.2.** *When a relying party receives a known extension, it may find that the extension contains some unrecognized elements. Such unrecognized elements may be due to a mistake, or to a new element recently defined for the extension, and not known to the relying party. X.509 default behavior is to consider the entire extensions as unrecognized; if the extension is marked ‘critical’, this results in rejecting the entire certificate, i.e., considering it invalid. However, X.509 allows a specific extension to define other behavior for handling unrecognized elements.*

*Specify a format for an extension, which will allow defining additional new elements of two types: mandatory elements, that must be recognized by the relying party or the entire extension should be deemed unrecognized, and optional elements, which may be safely ignored by the relying party if not recognized, while still handling the other elements in the extensions.*

## 8.3 Certificate Validation and Standard Extensions

Upon receiving a certificate, the relying party must decide whether it can rely on and used the certified public key, for a particular application. If the certificate is signed by a CA trusted by the relying party, then the relying party would immediately apply the *certificate validation process*, which uses the public signature-validation key of the CA,  $CA.v$ , to determine if the given certificate is valid. We refer to such a directly-trusted CA as a *trust anchor*. If the certificate is not signed by a trust anchor, then the relying party should perform a more complex *certification path* validation process, which would determine if the relying party may trust this certificate, based on an additional set of certificates.

Most of our discussion applies to both the *X.509 specifications* as defined by the ITU, and to their adaptation for use in Internet protocols, as defined by the IETF in RFC 5280 [34], *Public Key Infrastructure (PKIX) certificate and CRL profile*. We point out few points where PKIX and X.509 differ.

In the following subsections, we discuss the certificate validation processes and the most important X.509 and PKIX standard extensions. In subsection 8.3.1, we discuss the validation process for certificates signed by a trust-anchor. In subsection 8.3.2, we discuss the standard *alternative name* extensions, providing alternative identification for both issuer and subject. In subsection 8.3.3 we discuss standard extensions dealing with the usage of the certified key, and the *certificate policies* related to the issuing and usage of the certificate. In subsection 8.3.5 we discuss standard extensions defining constraints on certificates issued by the subject, for the special case where the subject of a certificate is also a CA. In subsection 8.3.4 we discuss the process of *certificate path validation*, allowing validation of certificates which are not trusted directly by a trust-anchor, but, instead, by *establishing trust* in *intermediate CAs*, and the related standard extensions.

### 8.3.1 Trust-Anchor-signed Certificate Validation

We begin our discussion of the X.509 certificate validation process, by considering the (simpler) case, where the issuer is directly trusted by the relying party, i.e., a *trust anchor*. We later discuss the case where the issuer is not trusted directly, which requires further validation of the certificate path defined by a given sent of additional certificates.

Assume, therefore, that a relying party receives a certificate signed (issued) by a *trust anchor*, i.e., the relying party trusts  $I$ , the issuer CA, and knows its public validation key  $I.v$ . To validate the certificate, the relying party uses  $I.v$  and the contents of the certificate, as follows:

**Issuer.** The relying party verifies that the issuer  $I$  of the certificate, as identified by the *issuer distinguished name* field, is a trusted CA, i.e., a trust anchor.

**Validity period.** The relying party checks the validity period specified in the certificate. If the public key is used for encryption or to validate signatures on responses to challenges sent by the relying party, then the certificate should be valid at the relevant times, including at the current time. If the public key is used to validate signature generated at the past, then it should be valid at a time when these signatures already existed, possibly attested by supporting validation by trusted time-stamping services.

**Subject.** The relying party verifies that the subject, as identified, using distinguished name, in the ‘subject’ field, is an entity that the relying party expected. For example, when the relying party is a browser and it receives a website certificate, then the relying party should confirm that the website identity (e.g., domain name) is the same as indicated in the ‘subject distinguished name’ field of the certificate.

**Acceptable signature algorithm?** The relying party confirms that it can apply and trust the validation algorithm of the signature scheme identified in the *signature algorithm OID* field of the certificate. If the certificate is signed using an unsupported algorithm, or an algorithm known or suspected to be insecure, validation fails.

**Issuer and subject unique identifiers.** From version 2, X.509 certificates also include fields for unique identifiers for the issuer and the subject, which the relying party should use to further confirm their identities. In PKIX, these identifiers are usually not used, and PKIX does not require their validation. This is probably since in PKIX, the issuer and subject identifiers are typically in corresponding extensions.

**Extensions.** First, the relying party should validate that it is familiar with any extension marked as ‘critical’; the existence of any unrecognized extension, marked as critical, would invalidate the entire certificate. Then, the relying party should validate the provided set of *key and policy extensions*, as discussed in the next subsection. Finally, the relying party should validate the existence and validity of any non-standard extensions, which is required or supported by the relying party.

**Validate signature.** The relying party next uses the trusted public validation key of the CA,  $CA.v$ , and the signature-validation process as specified in the certificate, to validate the signature over all the ‘to be signed’ fields in the certificate, i.e., all fields except the signature itself.

In the following subsections, we discuss the main standard extensions defined in X.509 and PKIX, and then, in subsection 8.3.4, the validation of certificates which are *not* signed by a trust anchor, but, instead, support by a set of certificates.

### 8.3.2 Standard Alternative-Naming Extensions

Both X.509 and PKIX define the standard *SubjectAltName (SAN)* and *IssuerAltName* extensions, provide alternative identification mechanism to complement or replace the *Distinguished Name* mechanism, providing identification for identifying, respectively, the subject and the issuer. These alternative fields allow the use of other forms of names, identifiers and addresses for the subject and/or issuer. Note that a certificate may contain multiple SANs.

The most important form of an alternative name is a *Domain Name System (DNS)* name, referred to as *dNSName*, e.g., *example.com*. These dNSNames are used by most Internet protocols, and are familiar to most users. Also allowed but rarely used alternative names, include email addresses, IP addresses, and URIs.

In fact, the use of alternate names is so common, that in many PKIX certificates, the subject and issuer distinguished-name fields are left empty. Indeed, PKIX (RFC 5280) specifies that this must be done, when the Certificate Authority can only validate one (or more) of the alternative name forms, which is often the case in practice. PKIX specifies that in such cases, where the SubjectAltName extensions is the only identification and the subject distinguished name is empty, then the extension should be marked as *critical*, and otherwise, when there is a subject distinguished name, it should be marked as non-critical.

Note that PKIX (RFC 5280) specifies that the *Issuer Alternative Name* extension should always be marked as non-critical. In contrast, the X.509 standard specifies that both alternative-naming extensions, may be flagged as either *critical* or *non-critical*.

Also, note that implementations of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols, often allow certificates to include *wildcard certificates*, which, instead of specifying a specific domain name, use the *wildcard notation* to specify a set of domain name. For TLS, this support is clearly defined in RFC 6125 [99]. Wildcard domain names are domain names where some of the alphanumeric strings are replaced with the *wildcard character* ‘\*’; there are often restrictions on the location of the wildcard character, e.g., it may be allowed only in the complete left-most label of a DNS domain name, as in *\*.example.com*. Wildcard domain names are not addressed in PKIX (RFC5280) or X.509, and RFC 6125 mentions several security concerns regarding their use.

### 8.3.3 Standard key-usage and policy extensions

We next discuss another set of standard extensions, defined in both X.509 and PKIX, which deal with the usage of the certified key and with the *certificate policies* related to the issuing and usage of the certificate. This includes the following extensions:

**The authority key identifier extension.** Provides an identifier for the issuer’s public key, allowing the relying party to identify which public val-

idation key to use to validate the certificate, if the issuer has multiple public keys. It is always non-critical.

**The subject key identifier extension.** Provides an identifier for the certified subject's public key, allowing the relying party to identify that key when necessary, e.g., when validating a signature signed by one of few signature keys of the subject - including signatures on (other) certificates. It is always non-critical.

**The key usage extension.** Defines the allowed usages of the certified public key of the subject, including for signing, encryption and key exchange. The specification allows the use of same key for multiple purposes, e.g., encryption and validating signatures, however, this should not be used, as the use of the same key for such different purposes may be vulnerable - security would not follow from the pure security definitions for encryption and for signatures. An exception, of course, is when using schemes designed specifically to allow both application, such as signcryption schemes. This extension may be marked as critical or not.

**The extended key usage extension.** Allows definition of specific key-usage purposes as supported by relying parties. The specification also allows the CA to indicate that other uses, as defined by the key-usage extension, are also allowed; otherwise, only the specified purposes are allowed. This extension may be marked as critical or not.

**The private key usage period extension.** This extension is relevant only for certification of signature-validation public keys; it indicates the allowed period of use of the private key (to generate signatures). Always marked non-critical.

**The certificate policies extension.** This extension identifies one or more *certificate policies* which apply to the certificate; for brief discussion of certificate policies, see Note 8.2. The extensions identifies certificate policies using object identifiers (OID). This extension may be marked as critical or as non-critical.

**The policy mappings extension.** This extension is used only in certificates issued to another CA, called *CA certificates*. It specifies that one of the issuer's certificate policies, can be considered equivalent to a given (different) certificate policy used by the subject (certified) CA. This extension may be marked as critical or as non-critical.

**Exercise 8.3.** *Some of the extensions presented in this subsection should always be non-critical, while others may be marked either critical or non-critical. Justify each of these designations, e.g., for each of these extensions, give an example of a case where it should be non-critical.*

#### Note 8.2: Certificate policy (CP)

A *certificate policy (CP)* is a set of rules that indicate the applicability of the certificate to a particular use, such indicating a particular community of relying parties that may rely on the certificate, and/or a class of relying party applications or security requirements, which may rely on the certificate. Certificate policies inform relying parties of the level of confidence they may have in the correctness of the bindings between the certified public key and the information in the certificates regarding the subject, including the subject identifiers. Namely, the Certificate Policy provides information which may assist the relying party to decide whether or not to trust a certificate. The certificate policy may also be viewed as a legally-meaningful document, which may define, and often limit, the liability and obligations of the issuer (CA) for potential inaccuracies in the certificate, and define statutes to which the CA, subject and relying parties must conform.

#### 8.3.4 Certificate path validation

Certificate path validation allows validation of certificates which are not trusted directly by a trust-anchor, but, instead, by *establishing trust* in *intermediate CAs*, and the related standard extensions. PKI schemes require the relying parties to *trust* the contents of the certificate, mainly, the binding between the public key and the identifier. In the simple case, the certificate is signed by a CA trusted directly by the relying parties, as in Figure 8.1. Such a CA, which is directly trusted by a relying party, is called a *trust anchor* of that relying party.

Direct trust in one or more trust-anchor (directly trusted) CAs, might suffice for small, simple PKI systems. However, many PKI systems are more complex. For example, browsers deploy PKI to validate certificates provided by a website, during the SSL/TLS handshake. Browsers typically directly trust a large number - about 100 - of trust anchor CAs, referred to in browsers as *root CAs*. Furthermore, in addition to the root CAs, browsers also *indirectly trust* certificates signed by other CAs, referred to as *intermediate CA*; an intermediate CA must be certified by root CA, or by a properly-certified, indirectly-trusted intermediate CA.

Different relying parties may have different trust anchors, and different requirements for trusting intermediate CAs. The same CA, say  $CA_A$ , may be a trust anchor for Alice, and an intermediate CA for Bob, who has a different trust anchor, say  $CA_B$ .

Relying parties and PKIs may apply different conditions for determining which certificates (and CAs) to trust. For example, in the PGP *web-of-trust* PKI, every party can act certify other parties. One party, say Bob, may decide to indirectly trust another party, say Alice, if Alice is properly certified by a ‘sufficient’ number of Bob’s trust anchors, or by a ‘sufficient’ number of parties which Bob trusts indirectly. The trust decision may also be based on *ratings* specified in certificates, indicating the amount of trust in a peer. Some designs may also allow ‘negative ratings’, i.e., one party recommending *not to*

trust another party. The determination of whether to trust an entity based on a set of certificates - and/or other credentials and inputs - is referred to as the *trust establishment* or *trust management* problem, and studied extensively; see [?, ?, ?, ?, ?] and citations of and within these publications.

We focus on the simpler case, where *a single valid certification path* suffices to establish trust in a certificate. This is the mechanism deployed in most PKI systems and by most relying parties, and specified in X.509 and PKIX. In both of these, the validation of the certificate path is based on several *certificate path constraints extensions*, which we discuss in the following subsections.

### 8.3.5 The certificate path constraints extensions

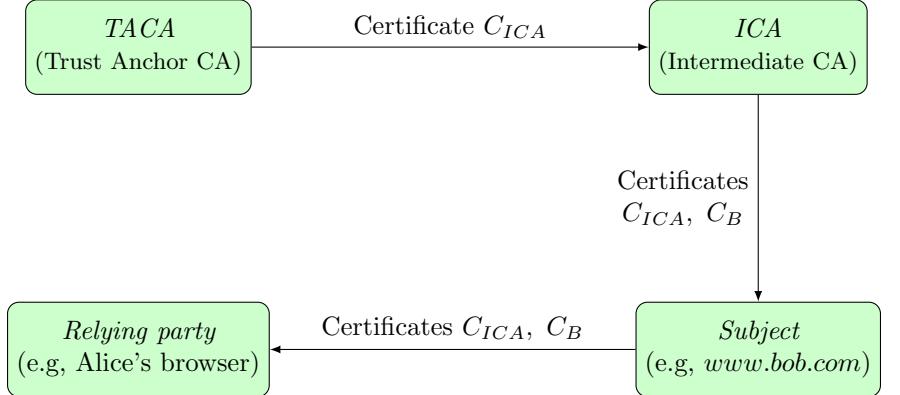
In this subsection, we present the three certificate path constraints extensions: *basic constraints*, *naming constraints* and *policy constraints*. These constraints are relevant only for certificates issued to a subject, e.g. *www.bob.com*, by some *intermediate CA (ICA)*, i.e., *ICA* is *not* directly trusted by the relying party (say Alice), i.e., is not one of Alice's *trust anchors*.

Since an *intermediate CA (ICA)* is not a trust anchor for Alice (the relying party), then Alice would only trust certificates issued by the ICA is 'properly certified' by some trust anchor CA; we use *TACA* to refer to a specific *Trust Anchor CA* which Alice trusts, and based on this trust, may or may not trust a given ICA.

In the simple case, illustrated in Figure 8.6, the relying party (Alice) receives two certificates: a certificate for the subject, e.g., the website *www.bob.com*, signed by some Intermediate CA, which we denote ICA; and a certificate for ICA, signed by the trust anchor CA, TACA. In this case, we will say that the subject, *www.bob.com*, has a *single-hop certification path* from TACA, since ICA is certified by the trust anchor TACA. In this case, therefore, the certification path consists of two certificates:  $C_{ICA}$ , the certificate issued by the trust anchor TACA to the intermediate CA ICA, and  $C_B$ , the certificate issued by the intermediate CA ICA to the subject (*www.bob.com*).

In more complex scenarios there are additional Intermediate CAs in the *certification path* from the trust anchor to the subject, i.e., the certification path is indirect, or in other words, contains multiple hops. For example, Figure 8.7 illustrates a scenario where the subject, *www.bob.com*, is certified via an indirect certification path with three hops, i.e., including three intermediate CAs: ICA1, ICA2 and ICA3. The subject *www.bob.com* is certified by ICA3, which is certified by ICA2, which is certified by ICA1, and only ICA1 is certified by a trust anchor CA, TACA. Hence, in this example, the certification path consists of *four* certificates: (1)  $C_{ICA1}$ , the certificate issued by the trust anchor TACA to the intermediate CA ICA1, (2) and (3), the two certificates  $C_{ICA2}$  and  $C_{ICA3}$ , issued by the intermediate CAs ICA1 and ICA2, respectively, to the intermediate CAs ICA2 and ICA3, respectively, and finally (4)  $C_B$ , the certificate issued by the intermediate CA ICA3 to the subject (*www.bob.com*).

We use the terms *subsequent certificates* to refer to the certificates in a certification path which were issued by intermediate CAs, and the terms *root*

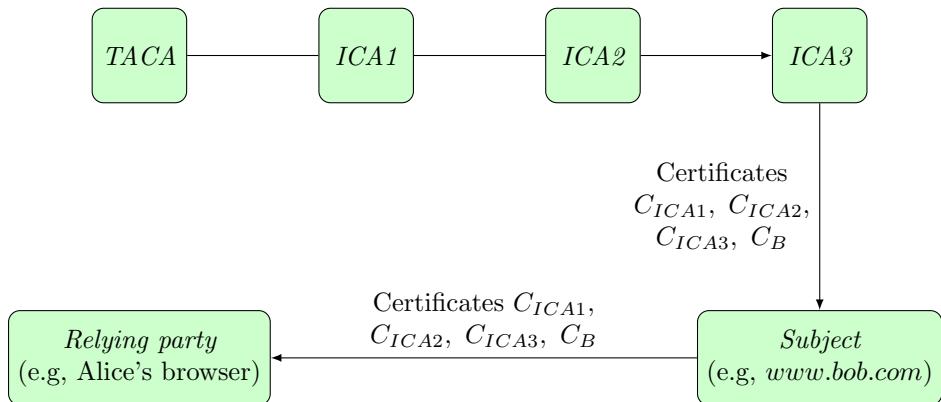


CICA constraints extensions						CB valid?
Basic		Naming		Policy		
cA	pathLen	Permit	Exclude	Req. Policy		
1	No	(any)	(any)	(any)	(any)	No
2	Yes	(any)	bob.com	none or x.bob.com	none or > 1	Yes
3	Yes	(any)	cat.com	(any)	(any)	No
4	Yes	(any)	bob.com	www.bob.com	(any)	No
5	Yes	(any)	(any)	(any)	0	No
6	Yes	(any)	cat.com	(any)	(any)	No
7	Yes	(any)	(none)	bob.com	(any)	No

Figure 8.6: A *single-hop* (length one) certificate-path, consisting of trust-anchor CA *TACA*, an intermediate CA *ICA*, and a subject (e.g., website *www.bob.com*). The table shows the impact of different certificate path constraints extensions (see subsection 8.3.5). For the examples in the table, assume that the certificate is for domain name *www.bob.com* and has no certificate policies extension.

*certificate* or *trust-anchor certificate* to refer to the ‘first’ certificate on the path, i.e., the one issued by the trust-anchor CA. The second certificate along the path is certified by the intermediate CA certified by the trust anchor (in the trust-anchor certificate); and any following certificate along the path, say the *i*th certificate along the path (for *i* > 1), is certified by the intermediate CA which was certified in the (*i* – 1)th certificate in the path. The *length* of a certificate path is the number of intermediate CAs along it, which is one less than the number of certificates along the path.

Note that, somewhat contrary to their name, the certification path constraints cannot prevent or prohibit Intermediate CAs from signing certificates which do not comply with these constraints; the constraints only provide information for the *relying party*, say Alice, instructing Alice to trust a certificate



	$C_{ICA1}$ constraints extensions					$C_B$ valid?
	Basic		Naming		Policy	
	cA	pathLen	Permit	Exclude	Req. Policy	
1	Yes	$< 2$	(any)	(any)	(any)	No
2	Yes	none or $\geq 2$	<i>bob.com</i>	none or <i>x.bob.com</i>	none or $> 3$	Yes
3	Yes	(any)	(any)	(any)	$\leq 3$	No
4	Yes	(any)	<i>cat.com</i>	(any)	(any)	No
5	Yes	(any)	(none)	<i>bob.com</i>	(any)	No

Figure 8.7: A *length 3* certificate-path, consisting of trust-anchor CA *TACA*, three intermediate CAs (*ICA1*, *ICA2*, *ICA3*), and a subject (e.g., website *www.bob.com*). The table shows the impact of the different certificate path constraints extensions (see subsection 8.3.5), in particular, of the *pathLen* (path length) parameter of the basic constraints extension. For the examples in the table, assume that the certificate is for domain name *www.bob.com* and has no certificate policies extension; and assume that the intermediate certificates  $C_{ICA2}$ ,  $C_{ICA3}$  do not indicate the *cA* flag in the basic-constraint extensions, and do not include any constraints, in their certificates, and that one (or more) of the certificates sent from *www.bob.com* to Alice, have no certificate policies extension.

signed by  $ICA$ , only if it conforms with the constraints specified in the certificates issued to the intermediate CAs.

### 8.3.6 The basic constraints extension

The *basic constraints* extension defines whether the subject of the certificate, say  $example.com$ , is allowed to be a CA itself, i.e., if  $example.com$  may also sign certificates (e.g., for other domains or for employees). More specifically, the extension defines two values: a Boolean flag denoted simply  $cA$  (with this non-standard capitalization), and an integer called  $pathLenConstraint$  (again, with this capitalization).

The  $cA$  flag indicates if the subject ( $example.com$ ) is ‘allowed’ to issue certificates, i.e., act as a CA; if  $cA = TRUE$ , then  $example.com$  may issue certificates, and if  $cA = FALSE$ , then it is not ‘allowed’ to issue certificates. Recall that this is really just a signal to the relying parties receiving certificates signed by  $example.com$ ; also, this only restricts the use of the certificate that  $I$  issued to  $example.com$  for validation of certificates issued by  $example.com$ , it does not prevent or prohibit  $example.com$  from issuing certificates, which a relying party may still trust, either since it directly trusts  $example.com$  (i.e., it is a *trust anchor*), or since it receives also an additional certificate for  $example.com$  signed by a different trusted CA, and that certificate allows  $example.com$  to be a CA, e.g., by having the value  $TRUE$  to the  $cA$  flag in the basic constraints extension.

The value of the  $pathLenConstraint$  is relevant only when there is a ‘path’ of more than one intermediate CA, between the Trust Anchor CA and the subject; i.e., it is relevant only in Figure 8.7, and not in Figure 8.6.

For example, in both Figure 8.6 and Figure 8.7, the Trust Anchor CA (TACA) signs certificate  $C_{ICA1}$ , where it should specify the  $ICA1$  is a trusted (intermediate) CA. Namely, it must set the  $cA$  flag in the basic-constraints extension of  $C_{ICA1}$  to  $TRUE$ . However, in Figure 8.7, ICA1 further certifies ICA2 which certifies ICA3 - and only ICA3 certifies the subject ( $www.bob.com$ ). Therefore, for the relying party to ‘trust’ certificate  $C_B$  for the subject, signed by ICA3, it is required that  $C_{ICA1}$  will also contain the path-length ( $pathLen$ ) parameter in the basic constraint extension, and this parameter must be at least 2 - allowing two more CAs till certification of the subject. Similarly, the certificate issued by ICA1 to ICA2 must contain the basic constraints extension, indicating  $cA$  as  $TRUE$ , as well as value of 1 at least for the  $pathLen$  parameter.

**Exercise 8.4** (IE failure to validate basic constraint). *Old versions of the IE browser failed to validate the basic constraint field. Show a sequence diagram for an attack exploiting this vulnerability, allowing a MitM attacker to collect the user’s password to trusted sites which authenticate the user using user-id and password, protected using SSL/TLS.*

### 8.3.7 The naming constraints extension

The naming constraints extension is used in certificates issued to a subject CA, such as the intermediate CAs in Figure 8.7 and Figure 8.6. As can be inferred from the tables in these figures, the naming constraints extension restricts the set of subject-names to be certified by the intermediate CAs. It has two possible parameters, *permit* (to define permitted name spaces) and *exclude* (to forbid name spaces, typically within the permitted name space).

Note that these examples focus on the typical case of DNS domain names, however, the restrictions may apply to other types of names, e.g., email names or X.509 domain names.

See Figure 8.8 for a typical application of the naming constraints extension, using X.509 domain names. In this example, the NTT Japan CA issues a certificate to IBM Japan, allowing the IBM Japan CA to certify any certificates with the value ‘IBM’ for the organization (O) keyword - implying that IBM Japan cannot certify other organizations. Also, see IBM Japan certifying the ‘main’, corporate IBM CA, but excluding sites where the value of the country (C) keyword is Japan, i.e., not allowing corporate IBM CA to certify sites in Japan, even IBM sites. Notice that such certificate issued by corporate IBM would also be trusted by relying parties using only NTT Japan as a trust anchor, provided that other relevant constraints such as certificate path length are satisfied (or not specified).

Figure 8.9 presents a similar example, but using DNS domain names instead of X.509 distinguished names.

### 8.3.8 The policy constraints extension

In addition to the *basic constraints* and *naming constraints* extensions, X.509 and PKIX also define a third standard extension that define additional constraints on subsequent certificates. This is the *policy constraints* extension, which is related to the certificate policies and certificate policy mappings extensions; see Note 8.2.

The policy constraints extension allows the CA to define two requirements which must hold, for subsequent certificates in a certificate path to be considered valid:

**requireExplicitPolicy:** if specified as a number  $n$ , and the path length is longer than  $n$ , then *all* certificates in the path must have a policy required by the user.

**inhibitPolicyMapping:** if specified as a number  $n$ , and the certificate path is longer than  $n$ , say  $C_1, \dots, C_n, C_{n+1}, \dots$ , then  $C_{n+1}$  and any subsequent certificate, should not have a *policy mapping* extension.

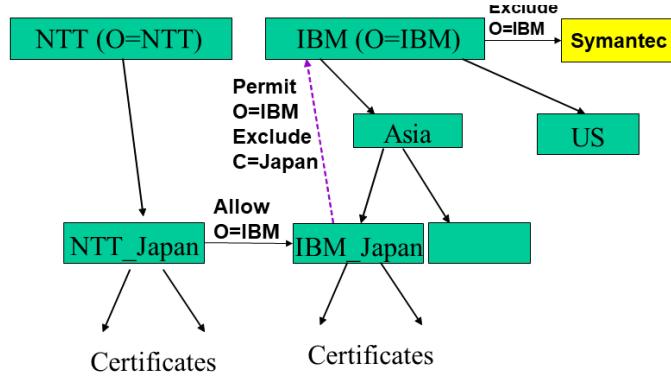


Figure 8.8: Example of the use of Naming Constraints, where constraints are over distinguished name keywords. NTT Japan issues a certificate to IBM Japan, with the naming constraint *Permit O=IBM*, i.e., allowing it to certify only distinguished names with the value ‘IBM’ to the ‘O’ (organization) keyword, since NTT does not trust IBM to certify other organizations. IBM Japan certifies the global IBM, only for names in the IBM organization (*Permit O=IBM*), and excluding names in Japan (*Exclude C=Japan*). Similarly, global IBM certifies Symantec for all names, except names in the IBM organization.

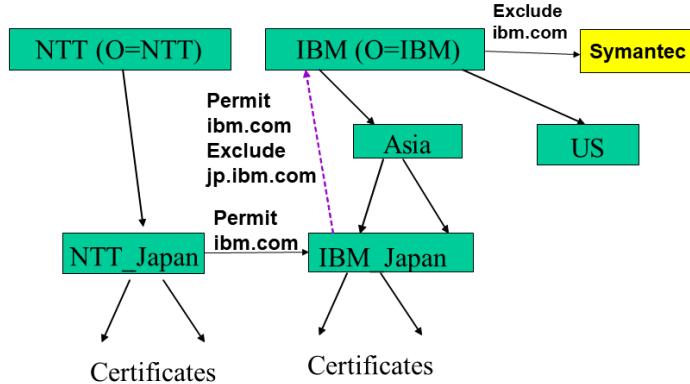


Figure 8.9: Example of the use of Naming Constraints, with similar constraints to the ones in Figure 8.8, but here using DNS names (dNSName).

## 8.4 Certificate Revocation

In several scenarios, it becomes necessary to revoke an issued certificate, prior to its planned expiration date. This may be due to security lapses or to administrative reasons:

**Revocation due to security lapses:** to mitigate *key compromise* and an *in-*

Signed fields	Version of CRL format
	Signature Algorithm Object Identifier (OID)
	CRL Issuer Distinguished Name (DN)
	This update (date/time)
	Next update (date/time) - optional
	Subject (user) Distinguished Name (DN)
	CRL Entry      Certificate      Revocation Date      CRL entry extensions Serial Number
	CRL Entry...      Serial...      Date...      extensions
	....
	CRL Extensions
Signature on the above fields	

Figure 8.10: X.509 Certificate Revocation List (CRL) format.

*correctly issued certificate*, e.g., CA compromise or failure of the CA to properly validate.

**Administrative revocation:** due to non-security end of usage, e.g. due to change required in the distinguished name other attribute, or if key is replaced.

However, this brings the question: how to revoke the certificates, i.e., how to inform the relying parties that a certificate was revoked? There are four main approaches:

**Certificate Revocation List (CRL):** in this approach, the CA periodically signs an object called a *certificate revocation list (CRL)*. A relying party can use the CRL to detect if a certificate issued by the CA was revoked. See subsection 8.4.1 and Figure 8.10.

**Online Certificate Status Protocol (OCSP):** OCSP replaces the CRLs, ‘pushed’ by the CA to all relying parties, with a request-response protocol between the relying party and the CA. Namely, the relying party sends a request to the CA for the status of a particular certificate (valid or revoked), and the CA sends the current status back. See subsection 8.4.2.

**OCSP stapling:** this protocol uses the same messages as OCSP, but avoids the challenge-response interaction between the relying party and the CA. Instead, the subject (e.g., website) provides a pre-retrieved OCSP response, together with the certificate, to the relying party. For details, see subsection 8.4.4.

**Short lived certificates:** in this approach, certificates are never revoked. Instead, certificates are always issued with a short validity period; instead of revoking a certificate, the CA simply does not extend the validity period. The impact is similar to OCSP stapling; however, with many CAs, the overhead of re-issuing a certificate may be higher; OCSP may also support more optimized certificates.

#### 8.4.1 Certificate Revocation List (CRL)

When using Certificate Revocation List (CRL), the CA periodically signs and distributes an object called a *certificate revocation list (CRL)*. A relying party can use the CRL to detect if a certificate issued by the CA was revoked. CRLs are defined as part of the X.509 and PKIX standards, with several variants and extensions, including an extension mechanism much like that of X.509 certificates; see Figure 8.10.

*Efficiency* is a major concern with CRLs, since the generation and distribution of CRLs can require excessive resources - think of a CA signing millions of certificates, and needing to revoke many of them over time. Three standard X.509 extensions are designed to improve performance:

**CRL distribution point extension** split certificates issued by a CA, to several sets ('distribution points'); to validate a specific certificate, you only need the CRL for that distribution list.

**Authorities Revocation List (ARL) extension** list only revoked CAs

**Delta CRL extension** – only new revocations since last ‘base CRL’

Even with such optimizations, CRLs may still introduce significant overhead. Furthermore, CRLs are (also) not immediate; who is responsible for the use of a revoked key, until the CRL is distributed?

As a result of these concerns, the use of CRLs is not very common. In particular, many browsers do not check CRLs; often, instead, they rely on OCSP. See subsection 8.4.2.

#### 8.4.2 Online Certificate Status Protocol (OCSP)

OCSP (Online Certificate Status Protocol) [101], shown in Figure 8.11 is a request-response protocol, providing a secure, *signed* indication to the relying party, showing the ‘current’ status of certificates - *revoked*, *good* or *unknown*.

The OCSP client, i.e., the entity sending the OCSP request, is either the relying party or another party, often the subject of the certificate (e.g., web server), who then provides the OCSP response to the relying party.

The OCSP responder, i.e., the entity that processes OCSP requests and sends responses, is an entity trusted by the relying party - typically, the CA itself or an entity delegated by the CA. Each OCSP response message is signed by the responder, allowing the relying party to validate it, even if received via an untrusted intermediary, e.g., the subject (website).

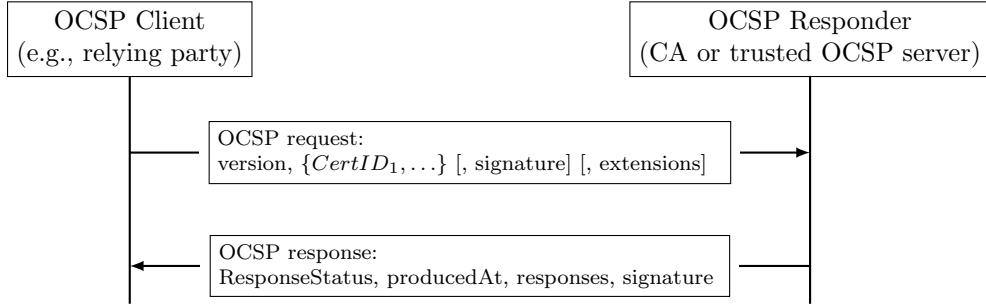


Figure 8.11: The Online Certificate Status Protocol (OCSP). The request includes one or more certificate identifiers  $\{CertID_1, \dots\}$ ; requests are optionally signed. The OCSP response is signed by the responder, and includes response for each  $CertID$  in the request. Each of these ‘individual responses’ includes the  $CertID$ , cert-status, time of this update, time of the next update, and optional extensions. Cert-status is either *revoked*, *good* or *unknown*.

To improve efficiency, a single OCSP request may specify (request status for) multiple certificates ( $CertIDs$ )<sup>2</sup>. Correspondingly, a single OCSP response, using a single signature, may include (signed) responses for multiple certificates. The support for OCSP request-response support for multiple certificates, is especially important to support certificates which are trusted by intermediate CAs, using a Certificate-Path; see Note 8.4.

OCSP status responses for each certificate may specify one of three values: *revoked*, *good* or *unknown*. The ‘unknown’ response is typically sent when the OCSP responder does not serve OCSP requests for the issuer of the certificate in question, or cannot resolve their status at the time (e.g., due to lack of response from the CA).

The length of an OCSP response is linear in the number of  $CertIDs$  in the corresponding OCSP request, rather than a function of the total number of revoked certificates of this CA, as is the case for CRLs. Furthermore, the computation required for sending an OCSP response is just one signature operation, plus some hash function applications, regardless of the number of revoked certificates or the number of certificates whose status is requested in this OCSP request. In the common case where the total number of revoked certificates may be large, this significantly reduces the overhead of generating and distributing often large CRL responses. Namely, OCSP provides an alternative which is often more efficient than CRLs; with CRLs, the CA must ‘push’ the list of all revocations to all relying parties, while with OCSP, a relying party receives information only about relevant certificates. As a result, OCSP is, in fact, widely deployed.

<sup>2</sup>Certificate identifiers ( $CertIDs$ ) may be specified using the hash of the issuer name and key, and a certificate serial number.

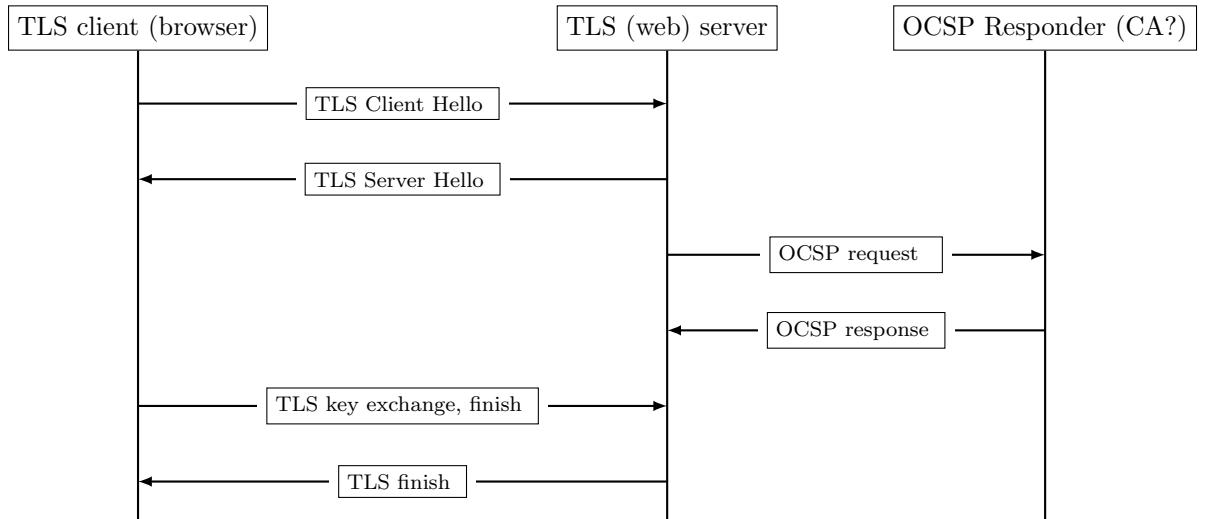


Figure 8.12: Use of the Online Certificate Status Protocol (OCSP), where the TLS client (e.g., browser) makes the OCSP request. There are several concerns with this form of using OCSP, including privacy exposure, overhead on CA, and handling of delayed/missing OCSP response by the client/browser. This last concern, illustrated in Figure 8.13, motivated updated browsers to support and prefer *OCSP-stapling* (see ??), where the TLS/web server makes the OCSP request, instead of the client/browser, and ‘staples’ the OCSP response to the TLS server hello message.

OCSP requests may fail due to several reasons, in which case the OCSP responder should send back an appropriate return code. Reasons for failure include:

- Lack of signature on OCSP request (when required by OCSP responder)
- Request not properly authorized/authenticated, e.g., not from known IP address, or missing/incorrect authentication information, when required by OCSP responder. Authentication information should be provided by the client in an appropriate OCSP extension.
- Technical reasons, such as overload or internal error.

### Deploying OCSP: Challenges and Solutions

The classical deployment of OCSP is where the OCSP client is the relying party; for example, in typical TLS (and web) connections, this would be the browser. This ‘classical’ OCSP deployment by browsers (or other TLS clients), is illustrated in Figure 8.12.

However, this OCSP deployment, where the relying party is the OCSP client, has serious problems. Let us first focus on the most critical problem: *unspecified timeout response*. Namely, the OCSP specification is not completely clear as to what should the OCSP client, typically a browser, do after it sends the OCSP request, if it does not receive a response within ‘reasonable time’. The following are the main options - and why each of them seems unsatisfactory:

**Wait:** the relying party may simply continue waiting for the OCSP response, possibly resending the request periodically, and never ‘giving up’ after some timeout value. However, OCSP servers could fail or become inaccessible forever, or for extremely long, leaving the relying party in this state. We do not believe any relying party has taken this approach.

**Hard-fail:** abort the connection (and inform the user). However, the OCSP interaction may often fail due to benign reasons, such as network connectivity issues or overload of the OCSP responder. In particular, usually, the OCSP responder is the CA, and CAs often do not have sufficient resources to handle high load of OCSP requests. Therefore, while this approach is rarely adopted.

**Ask user:** the relying party may, after some timeout, invoke a user-interface dialog and ask the user to make the determination. For example, a browser may invoke a dialog, informing the user that the certificate-validation process is taking longer than usually, and ask the user what action should it do. While this option seem to empower the user, in reality, users are rarely able to understand the situation and make an informed decision.

**Soft-fail:** finally, the relying party may simply continue as if it received a valid OCSP response. By far, this is the most widely-adopted option; in the typical case of a benign failure to receive the OCSP response, there is no harm in picking this option. However, this choice leaves the user vulnerable to an impersonation attack using a revoked certificate, when the attacker can block the OCSP response; see Figure 8.13.

As Figure 8.13 shows, the soft-fail approach essentially nullifies the value of OCSP validation - against an attacker that exposes the private key, or is able to obtain a fake (and later revoked) certificate, and is also able to block the OCSP response. Exposing the private key and obtaining a fake certificate are both challenging attacks; however, they do occur, otherwise, there was no need in revocations. The other condition, of being able to block the OCSP response, is often surprisingly easy for an attacker, e.g., by sending an excessive number of OCSP requests to the OCSP responder (e.g. the CA) at the same time as the OCSP request from the relying party. In particular, an attacker is likely to be able to launch such attack by intentionally invoking appropriate links from a web-site controlled by the attacker, in a so called *web-puppet* attack; see the web-security chapter of part II.

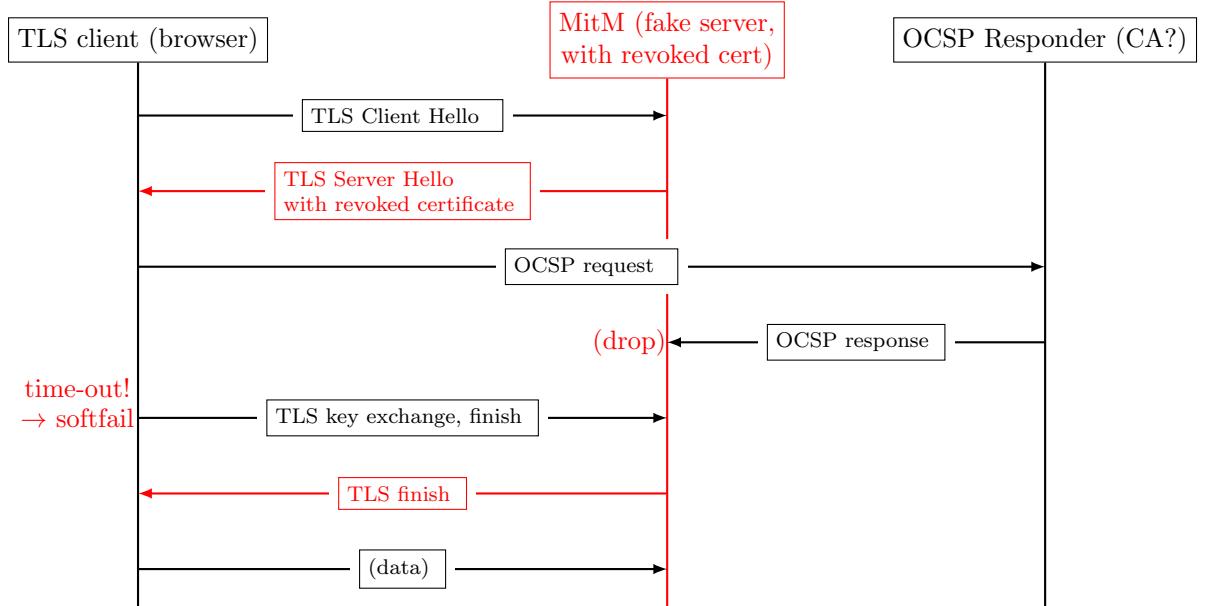


Figure 8.13: Attack on TLS connection, where the client (browser) uses OCSP (no stapling). The attacker is impersonating as a web site, to which the attacker has the private key; the corresponding certificate is already revoked, but the attack allows the attacker to trick the browser into accepting it anyway, allowing the impersonation attack to succeed. The browser queries the CA (or other OCSP server) to receive a fresh certificate-status. However, the attacker ‘kills’ the OCSP request, or the OCSP response (figure illustrates dropping of the response). After waiting for some time, the browser times-out, and accepts the revoked certificate sent by the impersonating web site, although no OCSP response was received. This ‘soft-fail’ behavior is used by most browsers, since the alternatives (very long timeout and/or ‘hard-fail’) are not accepted well by users.

The OCSP interaction may fail due to Internet connectivity issues; as a result, many OCSP implementations at relying parties, send the OCSP challenge - but proceed with the connection anyway, if response is not received in time. Unfortunately, this allows attackers to circumvent OCSP and use revoked certificates, by intentionally causing a failure to the OCSP challenge-response communication.

**Delay:** since OCSP is an online, request-response protocol, its deployment at the beginning of a connection often results in considerable delay.

**Privacy exposure:** the stream of OCSP requests (and responses) may expose the identities of web sites visited by the user to the OCSP responder, or to other agents able to inspect the network traffic. By default, OCSP

### Note 8.3: The UX beats Security Principle

In the *OCSP soft-fail vulnerability*, As described in § 8.4.2, most browsers support OCSP, but only using *soft-fail*, namely, if the OCSP-response is not received within some time, then the browser simply continues with the connection, i.e., ‘gives up’ on the OCSP validation and continues using the received certificate (assuming it was not revoked). It is well understood that this allows a MitM attacker to foil the OCSP validation, i.e., the use of the soft-fail approach results in a known vulnerability. Still, developers usually prefer to have this vulnerability, to the secure alternative of *hard-fail*, namely, aborting a connection after ‘giving up’ on the OCSP response. The reason is that the OCSP response may not arrive due to benign failures, such as network congestion or high load on the OCSP responder (typically, the CA), and aborting a connection in such cases would *harm the user experience (UX)*.

We see that here, developers preferred a vulnerable design with good UX, to a secure design with poor UX. Specifically, in this case, the user-experience was reduced due to abortion of a (benign) connection. There are many other scenarios, in which designers have to decide between one design with good security but poor UX, and another design with weaker security but better UX. The reduction in user-experience may be in terms of efficiency, usability, reliability, or features. Usually, designers prefer to retain good UX, even in the price of vulnerability; we refer to this as the *UX beats Security* principle.

**Principle 13** (UX beats Security). *Designers usually prefer a design with weak security but good UX, over a design with stronger security but with poor UX.*

The challenge for designers is to find solutions which will ensure sufficient security, without harming the user experience (UX).

requests and responses are not encrypted, exposing this information even to an eavesdropper; but even if encryption is used, privacy is at risk. First, the CA is still exposed to the identities of web-sites visited by a particular user. Second, even with encryption of OCSP requests and responses, the timing patterns may allow an eavesdropper to identify visited websites.

**Computational and communication overhead:** while OCSP often reduces overhead significantly cf. to CRLs, it still requires each response to be signed, which is computational burden on the OCSP responder. In addition to this computational overhead, there is the overhead due to the need of the OCSP responder to interact with any client; this overhead remains even if applying optimizations that reduces the OCSP computational overhead, e.g., as in Exercise 8.6.

Let us first focus on the last concern - the computational and communication overhead, mainly from the point of view of the OCSP responder. The signatures in OCSP responses imply significant processing overhead, as well as some additional bandwidth; these can be a significant concern to the OCSP responder, e.g., a CA which may need to support a ‘flood’ of requests from

#### Note 8.4: OCSP requests for Certificate-Path (CP)

An *indirectly trusted* certificate, certified via a certificate-path of (one or more) intermediate CAs, may be invalidated via revocation of any of these intermediate CAs. A relying party wishing to validate the status of the certificate, needs an updated status of the certificate of each intermediate CA, in addition to the status of the certificate of the subject. The fact that an OCSP request may include multiple certificates, may allow this process to be more efficient; a single OCSP request-response interaction may suffice to obtain updated status for all of these certificates, provided that the same OCSP responder is able to provide (signed) OCSP responses for all of these certificates (issued by different CAs). Note that the original OCSP stapling support in TLS, as defined in the certificate-status extension, does not support stapling of multiple certificates. To support this important case, browsers and servers should use the later-defined ‘multiple certificate status’ extension, RFC 6961 [92].

visitors of a (suddenly popular) website. This may motivate the delegation of the OCSP-responder function from the CA to a separate service; however the CA must still provide the relevant information to this service, and the relying parties must trust this service.

Due to the overhead concerns, OCSP responders may limit their services to authorized OCSP clients. To support this, OCSP requests may be signed; some servers may use other ways to authenticate their clients, e.g. using the optional extensions mechanism supported by OCSP requests. In subsection 8.4.3 we discuss additional methods to reduce the computational overhead of OCSP, mainly on the OCSP responders (often CA) but also on OCSP clients (relying parties, and possibly also subjects).

#### 8.4.3 Optimized variants of OCSP

##### The Certificate-Hash-Tree variant of OCSP

This OCSP variant uses the Merkle hash-tree technique, introduced in subsection 4.7.1, to allow the CA or OCSP responder to periodically perform a single signature operation, to provide OCSP responses indicating status for any OCSP requests. Assume that the CA issued a large set of certificates  $c_1, c_2, \dots, c_n$ , but each OCSP request will contain only one or few certificate-identifiers.

As shown in Figure 8.14, the signature is computed over the result of a hash-tree applied to the entire set of certificates issued by the CA (and their statuses), concatenated with the current time. The leaves of the hash-tree leaves are the pairs of individual certificates  $c_1, \dots, c_n$  and their corresponding statuses  $s_1, \dots, s_n$ . The construction uses a collision-resistant hash function (CRHF) denoted  $h$ .

The OCSP response for a query for status of certificate  $c_i$ , consists of this signature, plus the values of ‘few’ internal nodes, essentially, one node per layer of the hash tree. This allows the OCSP client to recompute the result of the hash tree, and then validate the signature. For example, to validate

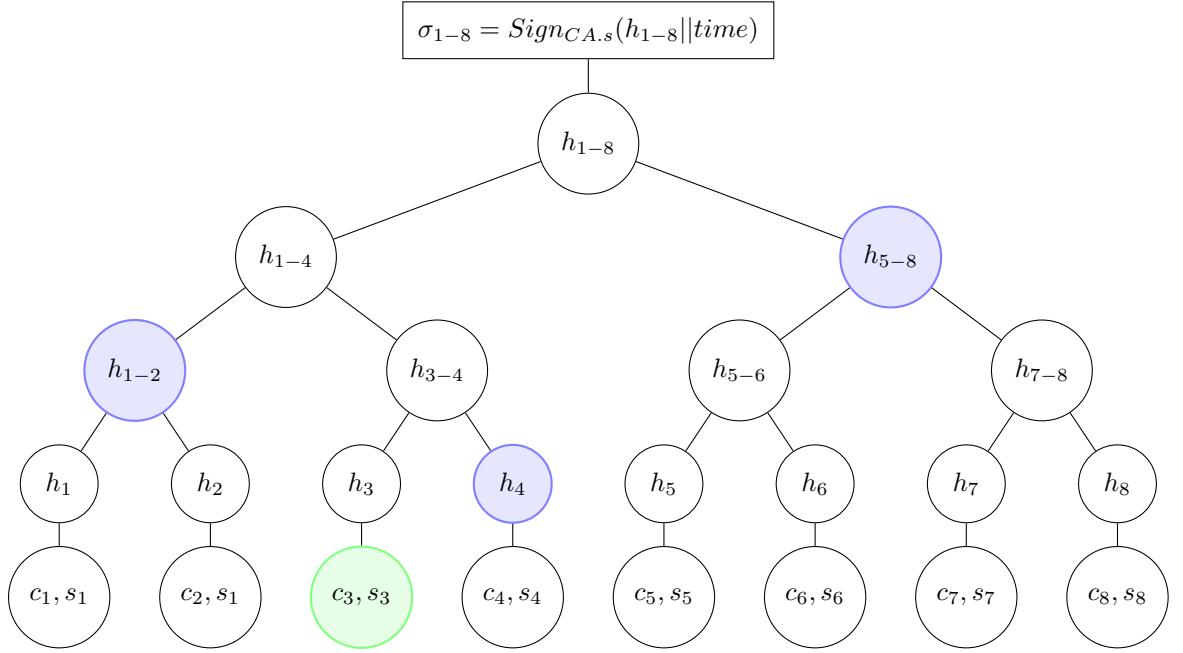


Figure 8.14: Signed certificates hash-tree. The leafs are the pair of certificate  $c_i$  and its status  $s_i \in \{good, revoked, unknown\}$ . The root is the signature over the hash-tree and the time. Every internal node is the hash of its children; in particular, for every  $i$  holds  $h_i = h(c_i)$ , and  $h_{i-(i+1)} = h(h_i || h_{i+1})$ . To validate any certificate, say  $c_3$ , provide the signature of the certificate hash-tree, i.e.,  $\sigma_{1-8}$ , the time-of-signing and the values of internal hash nodes required to validate the signed hash, namely  $h_4$ ,  $h_{1-2}$  and  $h_{5-8}$ .

the value of  $c_6$ , the response should include  $h_5$ ,  $h_{7-8}$  and  $h_{1-4}$ . To validate, compute  $h_6 = h(c_6)$ , then  $h_{5-6} = h(h_5 || h_6)$ , then  $h_{5-8} = h(h_{5-6} || h_{7-8})$ , then  $h_{1-8} = h(h_{1-4} || h_{5-8})$  and finally verify the signature over  $h_{1-8}$  and *time*, by validating that  $verify_{CA.v}(\sigma_{1-8}, h_{1-8} || time)$ . We refer to this set of values (e.g.,  $c_6, h_5, h_{7-8}, h_{1-4}$ ) as *proof of inclusion* of  $c_6$ .

**Exercise 8.5.** Consider certificate-hash-tree variant of OCSP, described above and illustrated in subsection 4.7.1.

1. Present pseudo-code for the OCSP client, including validation of the OCSP responses (including the proofs of inclusion).
2. Let  $n$  be the number of certificates issued by a CA,  $r < n$  be the number of revoked certificates, and  $i < r$  be the number of certificate-identifiers sent in a given OCSP request. What is the number of (1) signature operations, (2) signature-validation operations, (3) hash operations, required to (a) produce and send a CRL, (b) produce and send an OCSP response,

- (c) produce a certificate-hash-tree OCSP response, (d) validate a CRL,  
 (e) valid an OCSP response, (f) validate an certificate-hash-tree response.
3. This variant uses an (unkeyed) collision-resistant hash function (CRHF)
    - h. Explain why it may be desirable to avoid this assumption.
  4. Would it be Ok to use in the design a Second-Preimage Resistant (SPR) hash function, instead of the keyless CRHF? Present a convincing justification, preferably, with a reduction to prove security or with a counter-example showing insecurity (for the use of SPR hash function in this construction).
  5. Present an alternative way to replace the keyless CRHF with a different function which is about as efficient (as the original design using CRHF), yet is secure under a more acceptable assumption.

### **Exercises: other efficiency-improving variants of OCSP**

The OCSP certificate hash-tree variant allows the OCSP responder (e.g., CA) to use a single signature operation, to authenticate response to is an efficient way to validate many certificates using one signature operation. The following exercise discusses a different optimization, designed to provide OCSP response to a single certificate, optimized to avoid any signature in the typical case that the certificate is *not revoked*.

**Exercise 8.6** (OCSP Hash-chain Variant). *To avoid computational burden on the OCSP responder, it is proposed to use an alternative mechanism to OCSP, which will usually avoid the use of signatures as long as the certificate isn't , yet ensure secure revocation information. Specifically, add an extension, say called 'OCSP-chain', to the X.509 public-key certificate. The OCSP-chain extension will contain an n-bit binary string  $x_0 \in \{0, 1\}^n$ . A simple, efficient algorithm uses  $x_0$  to validate a 'daily validation token'  $(i, x_i)$  to be sent by the CA; the CA sends the token  $(i, x_i)$  only if the certificate was not revoked  $i$  days after it was issued. Your solution may use a cryptographic hash function  $h$ , and 'security under random oracle model' suffices, i.e., modeling as if  $h$  is a random function.*

1. Validation of token  $(i, x_i)$  is done by checking that  $x_0 = \underline{\hspace{10cm}}$ .
2. The CA may efficiently compute  $x_i$  from  $x_j$ , for  $j > i$ , by:  $x_i = \underline{\hspace{10cm}}$ .
3. Assume that the certificate's maximal validity period is 1000 days. The CA computes/selects  $x_0$  (in the certificate) and  $x_{1000}$  (for last day's token), by  $x_0 \leftarrow \underline{\hspace{10cm}}$  and  $x_{1000} \leftarrow \underline{\hspace{10cm}}$ .
4. Should the OCSP-chain extension be marked 'critical'?
5. Assume that the probability of a random certificate to be revoked on a particular day is less than  $10^{-5}$ . A further optimization of reduces the

number of hash-chain computations by the CA, by ‘grouping’ 100 certificates in a common ‘hash-chain group’. As long as none of these certificates is revoked, they all use the same ‘hash-chain group token’; only when one of them is revoked, will a per-certificate token be sent. This extension requires an extended version of the OCSP-chain extension, i.e., the extension should not simply contain an  $n$ -bit binary string  $x_0 \in \{0, 1\}^n$  as before. Instead, the extension should contain: \_\_\_\_\_; and the ‘daily validation token’ will change from  $(i, x_i)$  to \_\_\_\_\_ (before any of the certificates in the group is revoked) and to \_\_\_\_\_ (after one or more of the certificates in the group is revoked).

Further improvements may be possible. Next, in this paragraph and in Exercise 8.7, we consider the *Revoked-certificates hash-tree OSCP-variant*. This OSCP-variant uses a hash-tree of all *revoked certificate-identifiers*, sorted by certificate identifier. Since the tree is sorted, it can provide *efficient proof of non-revocation* of a certificate. For example, assume we use the certificate serial numbers to identify certificates, both in OCSP requests, and as the key for sorting the revoked identifiers hash-tree. Assume that an OSCP query contains a single certificate serial number, say  $i$ . The OCSP response will include the signed revoked-certificates hash tree, together with:

**If  $i$  was revoked:** proof of inclusion of  $i$  in the tree, similar to the one illustrated in Figure 8.14.

**If  $i$  was not revoked:** proof of inclusion of  $i'$  and  $i''$  in the tree, where  $i' = \max\{i' < i \wedge i' \text{ was revoked}\}$  and  $i'' = \min\{i'' > i \wedge i'' \text{ was revoked}\}$ .

**Exercise 8.7** (Revoked-certificates hash-tree OSCP-variant). *Following the brief outline in the previous paragraph:*

1. *Design and analyze an improved OCSP-variant using hash-tree of revoked certificates.*
2. *Extend your design to also incorporate the hash-chain technique (Exercise 8.6).*

#### 8.4.4 OCSP Stapling: query by subject (website)

In the previous subsection, we have seen several disadvantages of OCSP, as well as some possible efficiency improvements. However, even if these efficiency improvements are applied, some of the drawbacks remain, mainly the privacy exposure, the communication overhead on the CA, requiring the CA to send OCSP responses to every client, and, possibly most significantly, the Availability / Security dilemma.

A possible solution to these problems is to move the responsibility to obtain ‘fresh’ OCSP signed responses to the *subject* (e.g., web-server), rather than placing this responsibility (and burden) on every client. This immediately eliminates the privacy exposure as well as the availability/security dilemma,

and allows the OCSP responder to send a single signed OCSP response to each subject (website) - still a significant overhead, but way less than sending to every relying party (client).

Note that when a certificate is signed by an intermediate CA, it is necessary to validate the entire certification path; see Note 8.4.

### OCSP-stapling as defense against powerful attackers

Consider the following powerful attacker model: the attacker has Man-in-the-Middle capabilities, and *also* is able to obtain a fraudulent certificate to a website, e.g., *www.bob.com*. Suppose, further, that the fraudulent certificate is promptly discovered and revoked. Would the use of OCSP-stapling prevent the attacker from continuing to impersonate as *www.bob.com*?

Unfortunately, no; see Exercise 8.11. However, the *must-staple* certificate extension [62], as well as a possible extension to the OCSP-stapling mechanism, may help against this threat. These solutions are discussed in Exercise 8.11.

## 8.5 X.509/PKIX Web-PKI CA Failures and Defenses

We now focus on the application of PKI techniques to protect web communication, i.e., Web-PKI. As we discussed in § 8.1, there have been multiple, well-known incidents where certificate authorities issued invalid certificates, often as part of an attack on TLS/SSL applications for securing web connections; we presented some failures in Table 8.1. In this section, we discuss the weaknesses of the web-PKI usage of X.509/PKIX, which allowed these failures, and possible defenses. In the next section, we focus on the most ambitious defense - the *Certificate Transparency (CT)* PKI, which is a significant extension of X.509/PKIX.

### 8.5.1 Weaknesses of X.509/PKIX Web-PKI

In an utopia, each certificate is issued by one of a few, highly trusted CAs, with a clear focus and ability to provide secure certification service, controlling a specific name space, and preventing any misleading and unauthorized certificates. . However, the reality is very different. In particular, browsers trust too many CA (signing) certificates; e.g., a study from 2013 [46] found 1832 browser-trusted signing certificates, out of which 40% were academic institutions, and only 20% were commercial CA. Only 7 of these CAs were restricted to a particular name-space (name constraints), and most did not have any length constraints either.

The following seem to be the main weaknesses of the classical Web-PKI system:

**Insufficient CA assessment:** browsers, the most common type of relying party, are distributed with a list of many - typically, around hundred or

more - ‘root CAs’, i.e., trust-anchor CAs. The reason for this excessively-long list may be that browser manufacturers do not want to take the responsibility for judging the trustworthiness of different CAs, a role that may imply liability and potential anti-trust concerns. Indeed, the role of a browser is to provide functionality, not necessarily to represent the trust of individual users; ideally, each *user* should determine which CAs are ‘trust anchors’. Indeed, browsers typically provide user interface allowing users to edit the list of trusted CAs; however, a negligible fraction of the users actually modify the default list of trusted CAs. The bottom line is that some root-CAs may not be sufficiently careful, or, worse, may be subject to coercion or otherwise under control of attackers. Note that such attacks may be done intentionally by nation-states, which may mean that a CA may have to comply and cooperate.

**No restrictions on certified names/domains:** not only are there (too) many ‘root’ (trust-anchor) CAs, but, furthermore, any root-CA may certify any domain, without any restrictions. Furthermore, TLS/SSL certificates issued by root-CAs to intermediate CAs, rarely include ‘naming constraints’, and most browsers do not apply naming-constraints at all; hence, also any intermediate CA may publish certificate for any site and domain.

**‘Domain-Validated’ certificates:** to ensure security of certificates, it is crucial for the certificate authority to properly authenticate requests for certificates. However, proper authentication may require costly, time-consuming validation of the requesting entity and its right to assert the requested identifier (typically, domain name). To reduce costs and provide quicker, easier service to customers (requesting certificates), many certificate authorities use automated mechanisms for authentication, often based on *domain validation*. Domain validation refers to authenticating the ‘ownership’ for the requested domain name, by accessing Internet resources related to the given domain name. This relies on the security of the Internet’s infrastructure - the domain name system (DNS) and the routing mechanisms, both of which have significant vulnerabilities, as we discuss in the second part of this manuscript. Furthermore, some of the well-known PKI failures, involved insecure or missing domain validation, i.e., were circumvented even without compromise of either DNS or routing; e.g., see Table 8.1.

**Insufficient validation and coordination requirements:** certification authorities issuing SSL/TLS certificates for websites and domain owners, are only expected to validate the the requesting entity ‘owns’ the domain; as mentioned above, even that is often done using insufficiently-secure, automated ‘domain-validation’ mechanisms. However, even if the ‘ownership’ of a domain was validated, some risks would remain. In particular, currently, there may be multiple certificates for the same domain name - issued by the same CA or different CAs; i.e., *equivocation* is allowed,

which could be abused for phishing attacks. Furthermore, there is no requirement or standard mechanism for preventing the issuing of *misleadingly similar, deceptive* domain names, and these could also be used for phishing, often using *homograph attacks*; we discuss also these threats further in the second part of this manuscript.

### 8.5.2 Defenses against Corrupt/Negligent CAs

The naïve view of PKI security, is that certificate authorities are fully trusted, honest entities, that never fail to operate correctly, and in particular, to carefully vet any request for certificate. However, in view of the known failures (e.g., Table 8.1) and the weaknesses outlined above, a different approach seems advisable: PKI systems should ensure some security guarantees, even assuming that CAs may be corrupt or negligible.

For example, in the naïve view, the basic role of a public key certificate is to assure a mapping between the public key and the identity - *assuming* that the CA is trustworthy. However, an alternative view of this is certificates make a CA *accountable* for a fraudulent certificate. Namely, once we find a fraudulent certificate, properly signed by a CA, i.e., that validates as ‘correct’ using the public key of a CA, the CA becomes accountable for this fraudulent certificate. Such *accountability* can be viewed as a necessary requirement<sup>3</sup> for a secure PKI system.

Indeed, considering the reality of possibly Corrupt or Negligent CAs, there are additional proposed and deployed defenses against fraudulent certificates - beyond the basic CA accountability, directly assured by the signed certificate. The most significant in these proposals and efforts, is probably *certificate transparency*, which provides a public, audit-able log of certificates; we discuss it in the next section. Below, we discuss several other proposed defenses.

#### Use naming and other constraints.

One possible improvement to the security of the Web PKI system, is simply to *adopt and deploy* the certificate path constraints already defined in X.509 and PKIX, most notably, the *naming constraints*, discussed in subsection 8.3.7. This would allow restriction of the name space for certificates issued by a given CA, e.g., based on the DNS *top-level domain country codes (TLDcc)*, as defined in RFC 1591 [93]. For example, a Canadian CA may be restricted to the TLDcc for Canada (.CA). However, this idea is problematic; let us present two major concerns. First, the use of naming constraints requires a major change in the existing Web PKI system, which would be very hard to enforce. Second, there does not appear to be a natural way to place naming constraints on domain

---

<sup>3</sup>It is possible to define the accountability requirement, as well as other security requirements from PKI schemes, similarly to the definitions presented in earlier chapters for different cryptographic schemes such as encryption. Such definitions allow provably-secure PKI schemes; for more details on this approach, see [?].

names which are in one of the generic top level domains (gTLD) - and most domains belong to a gTLD, mainly *com*, *org*, *gov*, *edu* and *biz*.

### **Public key and/or certificate pinning.**

Fraudulent certificates can be foiled and/or detected by the relying party, typically the browser, when the browser is aware of the correct public key or certificate. For example, the Chrome browser had the public key of Google ‘burned-in’, allowing the browser to detect a fraudulent public key for Google (see the DigiNotar incident in Table 8.1). Such ‘burned-in’ public keys, are usually referred to as *static pinning* (or static key pinning). Both Google and FireFox support static key pinning for some domains. However, this is not a scalable solution.

*Dynamic pinning* of public keys or certificates, extend this mechanism, by allowing secure sites to direct the browser to ‘pin’ the given public key or certificate to the domain, for specified period; see specifications for HTTP public key pinning (*HPKP*) in RFC 7469 [49]. Similarly to static pinning, HPKP, and dynamic pinning in general, would foil and detect fraudulent keys/certificates, even if properly signed by a legitimate CA. With dynamic pinning, the server essentially declares ‘I always use this PK’ (or this certificate); and the client remembers this indication and refuses the use of other keys/certificates. In addition to or instead of refusing the ‘other’ key, the site may request clients to report cases where they receive a key or certificate, which differs from the pinned one. Dynamic pinning relies on the belief that attacks are the exception rather than the rule, and therefore, there is good likelihood that the first connection would be with the legitimate website, and then used to protect following connections; this approach is referred to as *Trust-On-First-Use (TOFU)*.

However, there are drawbacks to the use of key/certificate pinning, including:

1. If the private key is lost, e.g., due to failure, then there is a risk of losing the ability to serve the website, until the pinning times-out.
2. If an attacker has an fraudulent certificate and can provide it for the website *before* key pinning was done, then *the attacker* may perform key pinning, resulting in loss of control over the site for the specified pinning period. This may even be done intentionally to prevent access to the website - a type of Denial-of-Service (DoS) attack, or to blackmail the owner of the site. To address this concern, a backup copy of the secret key must be kept in secure, reliable storage.
3. Revoking a pinned key is difficult, since a replacement key is typically not pinned. One option is to wait for the pinning to time-out, but that implies period of unavailability of the site. An alternative is for the implementation to cancel key pinning once the certificate is revoked; however, the current specification does not require this behavior.

4. Key-pinning may be applied only for long-lived public keys, and may interfere with the use of rotating keys, and seems to completely prevent the use of ephemeral master keys, e.g., for perfect-forward recovery.

Due to these disadvantages, the use of key and certificate pinning may not be widely adopted, and support for it may be dropped.

Note, however, that there may still be value in *pinning of security policies*, e.g., pinning of the use of OCSP-stapling; see Exercise 8.11. Similarly, there may be a value in *CA pinning*, i.e., pinning of the identity of specific certificate authorities which the website would use (say, in the coming year), foiling attacks by certificates signed by rogue or negligent CAs.

#### **Pre-loading of ‘important’ revocations**

- . [TBD] Chrome ('CRLset'), FF ('OneCRL')

#### **Origin-bound certificates**

[TBD]

#### **Requiring certificate approval by multiple CAs**

[TBD] possible use of threshold signature scheme

## **8.6 Certificate Transparency (CT): Detecting CA failures**

**This section is incomplete. See lecture foils which are a bit more updated.**

*Certificate Transparency (CT)* [?, 79] is a proposal, originating in Google, for improving the security of the web PKI - and of PKI in general - when taking into account certification failures, due to corrupt or negligent certificate authorities. Intuitively, and as the name implies, the main goal of CT is to ensure *transparency* to the certification process. More specifically, as per RFC 6962, CT ‘aims to mitigate the problem of misissued certificates by providing publicly auditable, append-only, untrusted logs of all issued certificates’. Namely, ‘everyone’ should be able to know which certificates were issued, at what time, and by which CA, thereby avoiding the assumption that we can identify a set of certificate authorities which would be an infallible, always-trustworthy trust-anchor. Notice that this goal specifies that the CT logs would also be *untrusted*<sup>4</sup>, i.e., the idea is not just to move the trust requirements from one set of entities (CAs) to another (loggers), but to avoid entirely the need to assume fully-trusted entities.

---

<sup>4</sup>Note that the goal of allowing untrusted logs was removed in the current draft of CT 2.0 [?]. This may be only a temporary measure, and defenses against corrupt logs may be added in the final CT 2.0 RFC.

Transparency *improves* the *accountability* of certificate authorities, for the certificates they issue (sign). Even without CT, once a fraudulent certificate is discovered, the identity and accountability of the issuing CA is immediately established. With CT, accountability is established *proactively*, i.e., even without usage and discovery of the fraudulent certificate.

Transparency provides security benefits to both relying parties (users) and to certificate-subjects (websites/domains), as well as to trustworthy CAs:

**Benefits to websites/domains (subjects):** transparency allows owners of websites/domains, to detect issuing of fraudulent certificates for their domains. Furthermore, owners can also detect certificates issued for *misleading domain names*, which may trick users into believing they belong to the owner, e.g., abused in phishing attacks. Once detected, owners can take actions to mitigate the risk, e.g., by demanding revocation of these certificates. The owner may also take actions against the CA, e.g., ask for its removal from the list of root-CAs ‘trusted’ by browser, or sue the CA for damages. The increased risk to a CA issuing fraudulent certificates, would motivate CAs to be more secure.

**Benefit to relying parties (users):** users benefit from the reduced likelihood of falling victim to fraudulent websites, in particular, for phishing attacks.

**Benefit to trustworthy CAs:** trustworthy CAs benefit from reduced competition from shady CAs, whose operational costs may be lower due to insufficient security and/or insufficient scrutiny of submitted certificates.

To achieve these benefits, CT introduces three new entities to the PKI landscape:

**Loggers:** Certificate *loggers* are the main new entity in CT. These are entities which maintain logs of certificates issued by different ‘subscribing’ CAs. Namely, a CA that signs-up with a logger, is supposed to send to that logger every certificate it issues, allowing the logger to maintain a public log of all the certificates issued by that log, including time of issuance.

**Monitors:** CT *monitors* are services which perform *monitoring* of (typically multiple) loggers. Monitors observe either all certificates logged (which should be all certificates marked as conforming with CT), or certificates for or related

**Auditors:** *Auditors*, embedded in browsers or providing service to browsers, send to monitors queries containing CT certificates received (by the browser) from web-sites. The monitors respond with the status of the CT certificate; if the certificate is invalid, it becomes an evidence of misbehavior by the corresponding logger.

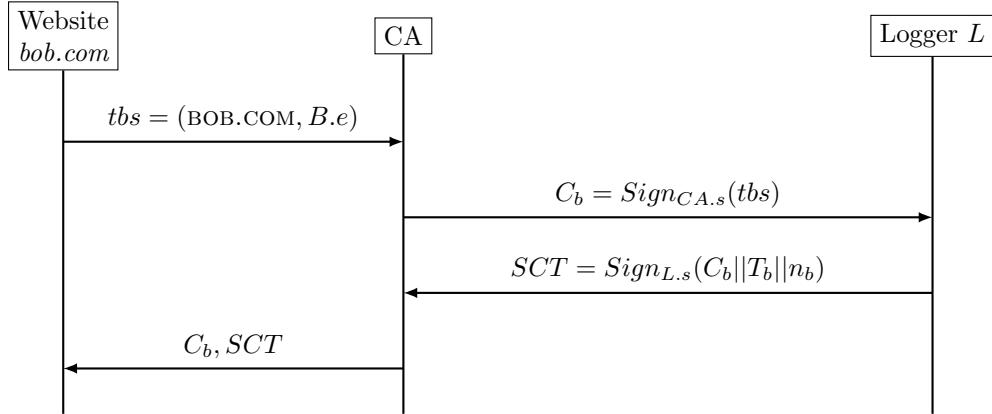


Figure 8.15: Issuing a CT certificate for *bob.com*, with public key  $B.e$ , using certificate authority  $CA$  and logger  $L$ .

### 8.6.1 Issuing CT certificates and the Signed Certificate Timestamp (SCT)

Issuing a CT certificate, begins like issuing of a ‘classical’ X.509/PKIX certificate. Namely, the subject, say website BOB.COM, sends its public key, say  $B.e$ , to the certificate authority ( $CA$ ). The certificate authority validates that the request - and in particular the public key  $B.e$  - come from the legitimate owner of BOB.COM; in particular, for *domain-validated* certificates, the  $CA$  merely validate ownership of the domain BOB.COM, e.g., by looking up a dedicated subdomain to find there encoding of  $B.e$ . Then, the  $CA$  signs the certificate, containing the subject identity, e.g., the domain name BOB.COM, and the public key  $B.e$ , and other fields, e.g., validity period and extensions.

To issue a CT certificate, the  $CA$  needs to then send the certificate to the *logger*, who adds the certificate to the log, and sends back a *Signed Certificate Timestamp (SCT)*. The SCT is a signature by the logger, on the certificate, current time, and the serial number of the SCT, i.e., the number of SCTs issued till now by this logger. This process is illustrated in Figure 8.15. The website receives the SCT from the  $CA$ , who receives it from the logger. The server sends the SCT to the client in a TLS extension, much like the use of the OCSP-stapling, see subsection 8.4.4.

[To be completed]

### 8.6.2 Monitoring: simplified by assuming Trustworthy Loggers

We begin with a simplified description of CT; this version does not include mechanisms to deal with corrupted loggers, i.e., it is only secure assuming all

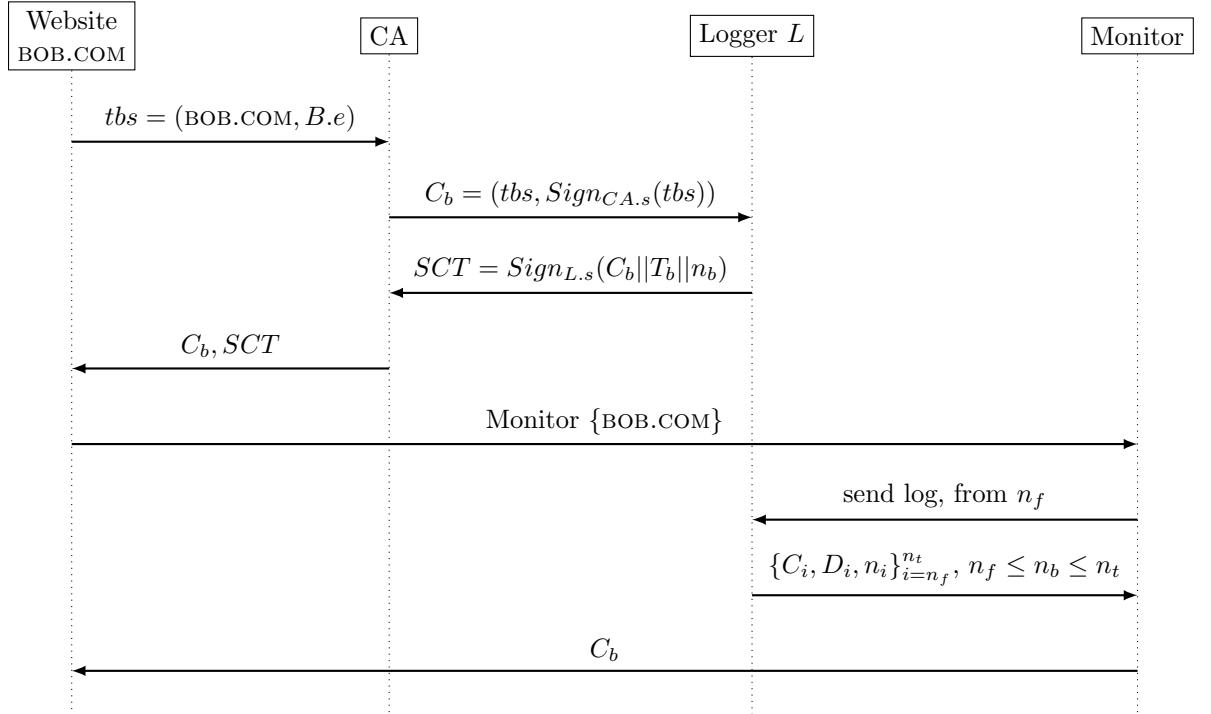


Figure 8.16: Monitoring domain name BOB.COM, using ‘simplified-CT’, which assumes that the logger  $L$  is honest. . Before this scenario is started, the monitor received up to certificate number  $f$  from this CA.

loggers are honest. In the following sections, we introduce additional defenses to avoid this assumption.

If the logger is trusted, then it suffices for the logger to sign a ‘timestamp’ whenever logging a certificate. This is shown in Figure 8.16 and Figure 8.17. Figure 8.16 shows the normal monitoring process, where a web-site, say BOB.COM asks the monitor to supervise the issuing of certs with this domain name, and the monitor only detects the legitimate issuing to BOB.COM itself. Figure 8.17 shows the attack-detection scenario, where the domain BOB.COM asks for monitoring of this domain and of the deceptively-similar domain B0B.COM, and the monitors report issuing of a suspect certificate for domain B0B.COM.

### 8.6.3 Auditing to detect rogue loggers

See Figure 8.18. [TBD]

See Figure 8.19

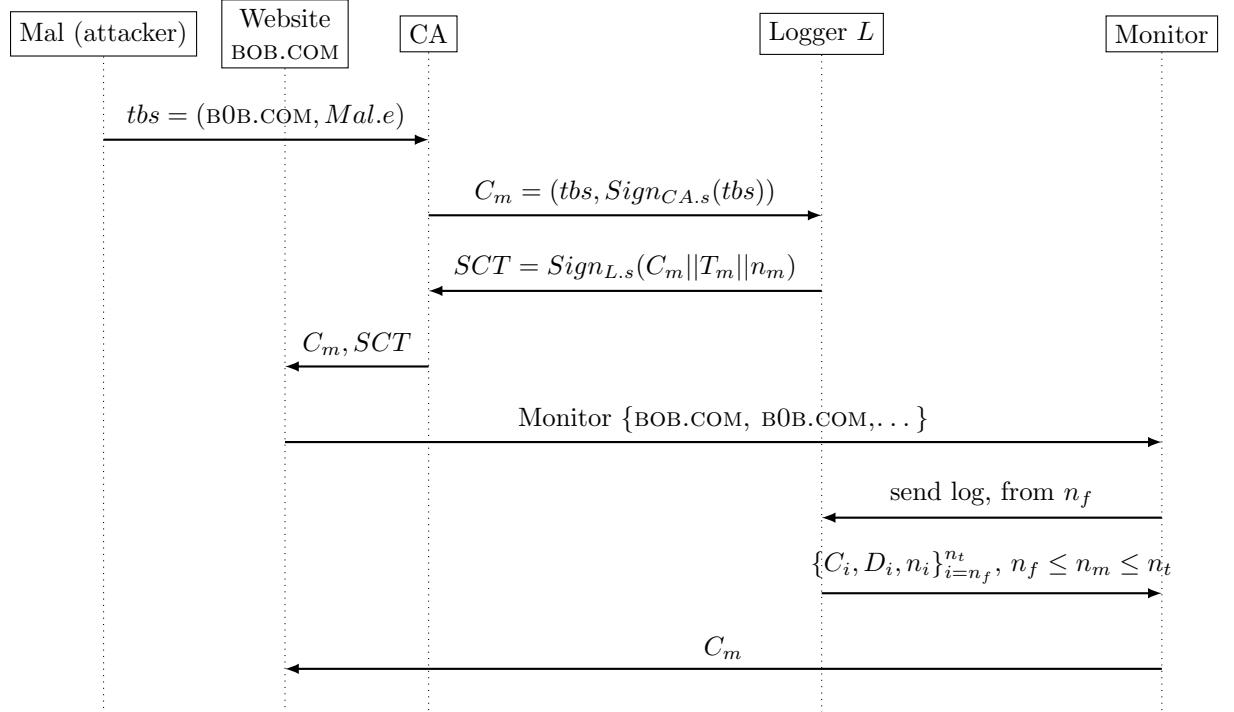


Figure 8.17: Detection, by a monitor, of a ‘misleading certificate’ issued for domain B0B.COM, using ‘simplified-CT’, which assumes that the logger  $L$  is honest. The monitor is asked to supervise BOB.COM, B0B.COM and other ‘potentially-misleading’ domains, by the owner of domain BOB.COM. Before this scenario is started, the monitor received up to certificate number  $f$  from this CA.

#### 8.6.4 Monitoring in CT, using Merkle tree of certificates

### 8.7 PKI: Additional Exercises

**Exercise 8.8.** *It is sometimes desirable for certificates to provide evidence of sensitive information, like gender, age or address. Such information may be crucial to some services, e.g., to limit access to a certain chat room only to individuals of specific gender. It is proposed to include such information in a special extension, and to protect the privacy of the information by hashing it; for example, the extension may contain  $h(DoB)$  where  $DoB$  stands for date of birth. When the subject wishes to prove her age, she provides the certificate together with the value  $DoB$ , allowing relying party to validate the age.*

1. *Should such extensions be marked ‘critical’? Why?*
2. *Show that the design above fails to preserve privacy, by describing the al-*

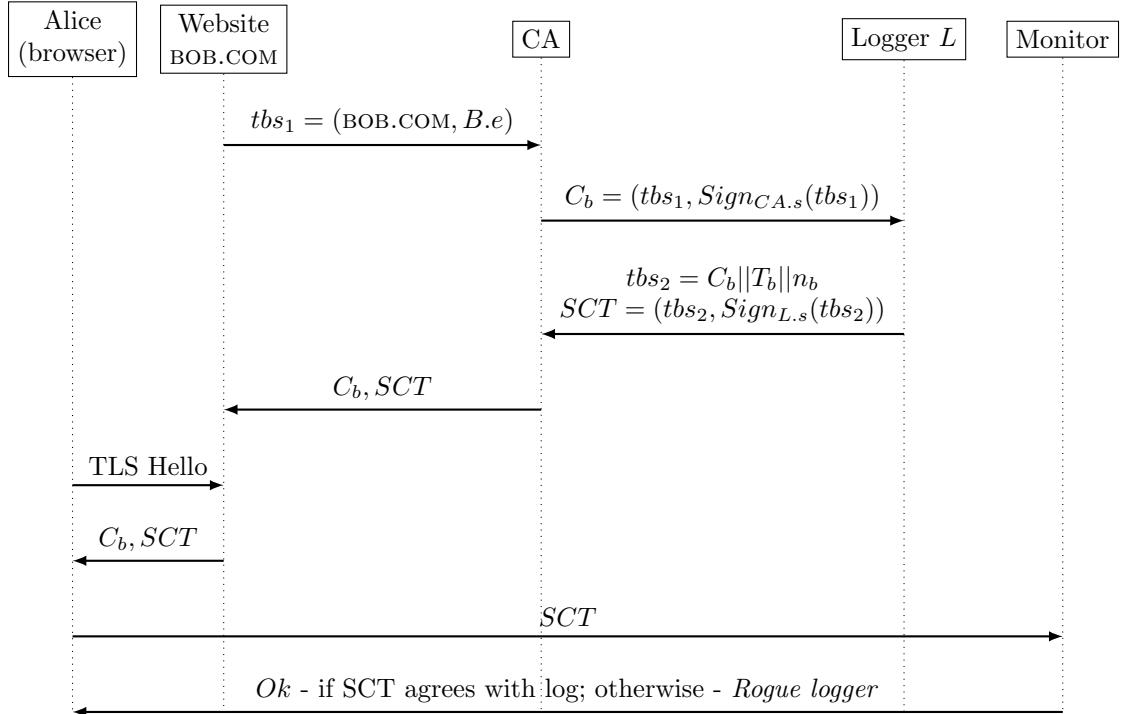


Figure 8.18: Auditing, simplified (assumes monitor keeps copy of entire log): TLS-client Alice (typically, browser) asks a monitor to audit an SCT that Alice received from website BOB.COM. If the SCT conforms with the copy of the log kept by the monitor, it returns Ok. Otherwise, the logger is exposed as corrupt.

gorithm an attacker would use to find out the sensitive information (e.g., DoB), without it being sent by the subject (e.g., Client). Use pseudo-code or flow-chart. Explain why your algorithm is reasonably efficient.

3. Present an improved design which would protect privacy properly. Your design should consist of the contents of the extension (instead of the insecure  $h(\text{DoB})$ ), and any additional information which should be sent inside or outside the certificate to provide evidence of age.
4. Prove, or present clear argument, for security of your construction, under the random oracle model.
5. Present a counterexample showing that it is not sufficient to assume that  $h$  is a collision-resistant hash function, for the construction to be secure.

**Exercise 8.9.** As shown in [46], a large number of browser-trusted CAs is operated by organizations such as universities, non-CA companies or other organizations, e.g., religious organizations. For example, assume that one of

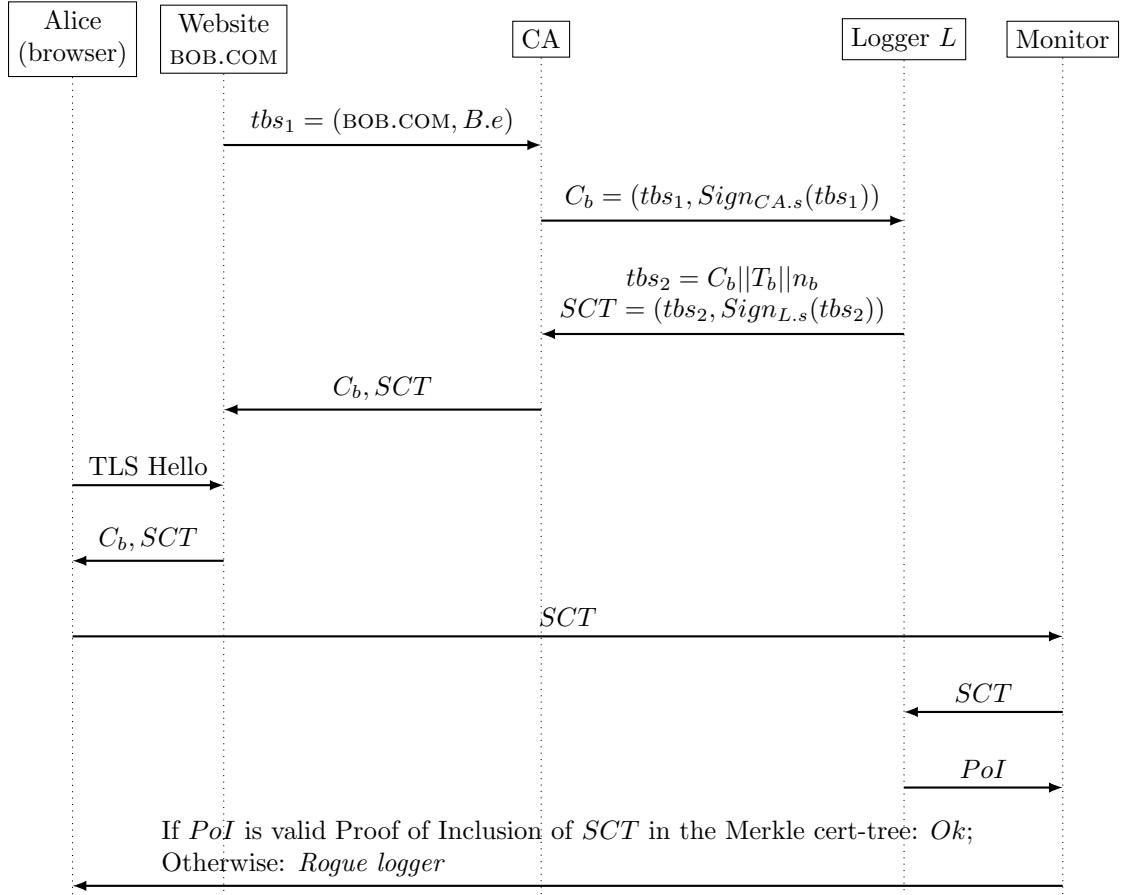


Figure 8.19: Auditing in CT, using Merkle tree. This scenario assumes that the Monitor has the Merkle cert-tree for this logger, computed all certificates issued till time  $T_b$  (or later).

these is nu.ne, a university in Niger, which obtained this signing certificate as an intermediate CA of a trusted root CA, say root.ca.

obtained this CA certificate so it may certify its different departments and project websites. Assume that the private key of nu.ne is compromised by a MitM attacker.

1. Would this allow the attacker to intercept the traffic between the user and (1) the university, (2) other Nigerian sites, (3) other universities, (4) additional sites (which)?
2. What would be a typical process for detection of the exposure of the private key of nu.ne? Can you bound the time or number of fake-certificates

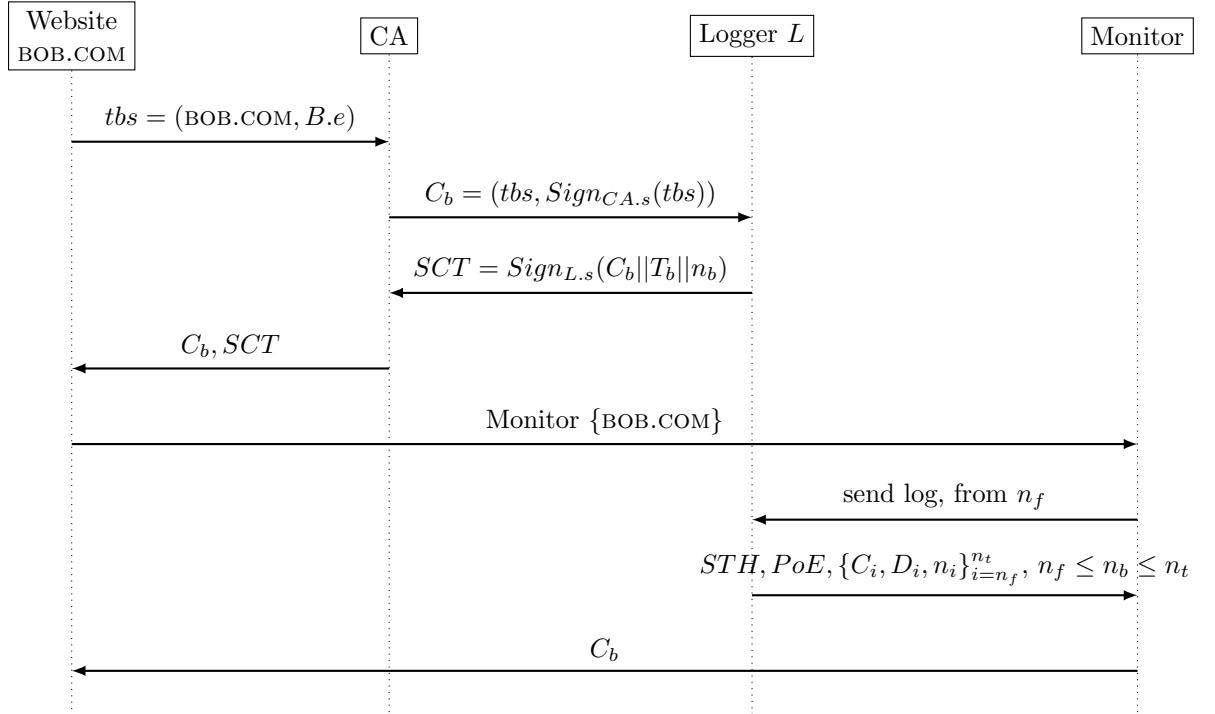


Figure 8.20: Monitoring domain name BOB.COM in CT, using Merkle tree of hashed-certificates.  $STH$  stands for Signed Tree Hash; it is a signature using the logger's key over the root of the certificates-hash tree. Before this scenario is started, the monitor received up to certificate number  $f$  from this CA, and corresponding  $STH$ .

*issued until exposure?*

3. *What should be the mitigation, once the exposure is detected?*

**Exercise 8.10** (Web-of-Trust). *In this exercise we consider Web-of-Trust PKI, first proposed and often associated with the PGP e-mail and file encryption/signing software. In a web of trust, each user can certify the (public key, name) mappings for people she knows, acting as a CA. To establish secure communication, two users exchange certificates they obtained for their keys, and possibly also certificates of the signers. Each user  $u$  maintains a directed graph  $(V_u, E_u)$  whose nodes  $V_u$  are (publickey, name) pairs from the set of all certificates known to  $u$ , and where there is an edge from one node, say  $(pk_A, Alice)$  to another, say  $(pk_B, Bob)$ , if  $u$  has a certificate over  $(pk_B, Bob)$  using key  $pk_A$ ; the graph has a special ‘trust anchor’ entry for  $u$  herself, denoted  $(pk_u, u)$ . Each user  $u$  decides on a maximal certificate path length  $L(u)$ ; we*

say that  $u$  trusts  $(pk_B, Bob)$  if there is a path of length at most  $L(u)$  from  $(pk_u, u)$  to  $(pk_B, Bob)$  in  $(V_u, E_u)$ .

Specify an efficient algorithm for determining for  $u$  to determine if she should trust  $(pk_B, Bob)$ . What is the complexity of this algorithm? Note: you may use well-known algorithm(s), in which case, just specify their names and reference, there is no need to copy them in your answer.

**Exercise 8.11** (OCSP-stapling against powerful attacker). Consider the following powerful attacker model: the attacker has Man-in-the-Middle (MitM) capabilities, and also is able to obtain a fraudulent certificate to a website, e.g., www.bob.com. Suppose, further, that the fraudulent certificate is promptly discovered and revoked.

1. Show a sequence diagram showing that this attacker can continue to impersonate as www.bob.com, in spite of the revocation of the certificate and of the use of OCSP-stapling by the website.
2. Explain how use of the must-staple certificate extension [62] would prevent this attack.
3. Propose a simple extension to OCSP-stapling, OCSP-stapling-pinning, that may also help against this threat, if the must-staple extension is not supported. Explain how your extension would foil the attack you presented. Hint: the required extension is related to the ‘key pinning’ mechanism discussed in subsection 8.5.2.
4. Discuss the applicability of the concerns of key pinning, discussed in subsection 8.5.2, to the extension.
5. Consider now an even more powerful attacker, who also has MitM capabilities, but also controls a root certificate authority (trusted by the browser). Show a sequence diagram showing that this attacker can continue to impersonate as www.bob.com, in spite of the use of the OCSP-stapling-pinning extension as you presented above.
6. Propose an improvement to the OCSP-stapling-pinning extension, that may help also against this threat.

**Exercise 8.12** (Keyed hash and certificates). Many cryptographic protocols and standard, e.g., X.509, PKIX and CT, rely on the use of the ‘hash-then-sign’ paradigm, based on the assumption that the hash function in use,  $h(\cdot)$ , a (keyless) collision-resistant hash function (CRHF).

1. Explain why this assumption is problematic, specifically, prove that there cannot exist any collision-resistant hash function. (If you fail to prove, at least present clear, convincing argument.)
2. One way to ‘fix’ such protocols, is by using, instead, a keyed CRHF. Explain why your argument for non-existence of a keyless CRHF, does not also imply non-existence of keyed-CRHF.

3. Present an extension to provide the keyed-hash key, for an X.509 certificate of a signature-verification public key. Explain, with example scenarios/applications, if the extension should be marked as ‘critical’, as non-critical, or if for some applications it should be marked ‘critical’ and for others marked ‘non-critical’.

**Exercise 8.13** (Malicious CA). Let  $MAC$  be a malicious CA, trusted (e.g., as a root CA) by browsers.

1. Assume bob.com generates a signing public-private key pair  $(B.s, B.v)$ , and has  $B.v$  certified by  $MAC$ . Present sequence diagram showing how  $MAC$  can eavesdrop on communication between the site https: bob.com and a client, Alice, in spite of their use of TLS 1.3. Your solution should be a very simple attack. Assume the use of only the defenses explicitly mentioned.
2. Assume that Alice uses CT and does not communicate with websites that do not provide a properly signed SCT. Assume a single, trustworthy logger,  $L$ , and a single, trustworthy monitor,  $M$ . Present a sequence diagram showing how this setup may prevent or deter  $MAC$  from performing the attack.
3. Assume that the private key of bob.com is exposed by  $MAC$ ; however, the exposure is detected, and Bob revokes its certificate. Furthermore, both Alice and bob.com deploy OCSP-stapling, and Bob’s public key certificate includes the must-staple extension. Show, with sequence diagram, how  $MAC$  may still eavesdrop on communication between Alice and Bob.

## Chapter 9

# Usable Security and User Authentication

### 9.1 Password-based Login

#### 9.1.1 Hashed password file

#### 9.1.2 One-time Passwords with Hash-Chain

### 9.2 Phishing

### 9.3 Usable Defenses against Phishing

### 9.4 Usable End-to-End Security

### 9.5 Usable Security and Authentication: Additional Exercises

**Exercise 9.1.** Many websites invoke third-party payment services, such as PayPal or ‘verified by Visa’. These services reduce the risk of exposure of client’s credentials such as credit-card number, by having the seller’s site open a new ‘pop-up’ window, at the payment provider’s site, say PayPal; and then having the users enter their credentials at PayPal’s site.

1. Assume that a user is purchasing at the attacker’s site. Explain how that site may be able to trick the user into providing their credentials to the attacker. Assume typical user, and exploit typical human vulnerabilities. Present the most effective attack you can.
2. Identify the human vulnerabilities exploited by your construction.
3. Propose up to three things that may help to reduce the chance of such attack.

**Exercise 9.2** (Homographic attacks). *To support non-Latin languages, domain names may include non-Latin characters, using unicode encoding. Some browsers display these non-Latin characters as part of the URL, while others display them only in the webpage itself, and, if they appear in the URL, display their punycode encoding (encoding of unicode as few ascii characters).*

1. *Discuss security and/or usability of these two approaches; where there is vulnerability, give an example scenario.*
2. *Some browsers display only using single font, i.e., never displaying domain names which mix Latin characters with unicode characters, or use punycode encoding in case of mixed fonts. What is the motivation? If this is open to abuse, give example.*
3. *Another proposal is that whenever displaying non-Latin characters, to add a special warning symbol at the end of the domain name, and if the user clicks on it, provide detailed warning. Identify any secure-usability principles and human vulnerabilities which are related to this proposal.*

*Solution to first part:* There is a usability advantage in allowing non-latin domain names to be displayed using the ‘correct’ font in the URL line: this allows such domain names to appear ‘correct’ to users familiar with the relevant (non-Latin) language. In fact, when such users use a browser that does NOT display non-Latin characters in the URL, then they may be practically unable to understand the URL; this may even open them to phishing attacks as they may get used to ignore the URL and not notice when the domain name is incorrect.

However, browsers displaying URLs with non-Latin characters may facilitate homographic attacks on websites in domain names consisting of Latin characters - the vast majority of websites, definitely popular ones. Specifically, an attacker may be able to buy a domain name which includes or consists of non-Latin characters and is available for sale, although it is visually very similar to a domain name used by some ‘victim’ website, due to using non-Latin characters which are visually similar (some are visually almost identical!) to some latin characters in the victim domain name. For example, the latin character P has a visually-similar Cyrillic character (also looking as P); hence the domain name APPLE.COM may be written using the Cyrillic P but appear visually identical to the ‘real’ APPLE.COM domain name.

**Exercise 9.3** (Anti-phishing browser). *A bank wants to design a special browser for its employees, which will reduce the risk of them falling to phishing attacks. It considers the following changes from regular browsers. For each, specify if it would have significant, small, negligible or no positive impact on security, and justify, based on secure usability principles.*

1. *Only allow surfing to SSL/TLS protected (https) sites.*
2. *Do not open websites as results of clicking on URLs received in emails.*

3. If user clicks on URL in email, browser displays warning and asks user to confirm that the URL is correct before requesting that page.
4. On the first time, per day, that the user surfs to a protected site, popup a window with the certificate details and ask the user to confirm the details, before requesting and displaying the site.

**Exercise 9.4.** In the Windows operating system, whenever the user installs new software, a pop-up screen displays details and asks the user to approve installation - or to abort. Many programs are signed by a vendor, with a certificate for that vendor from a trusted CA; in this case the pop-up screen displays the (certified) name of the vendor and the (signed) name of the program. Other programs are not signed, or the vendor is not certified; in these cases the pop-up screen displays the names given by the program for itself and for the vendor, but with clear statement that this was not validated.

1. Identify secure usability principles violated by this design, and explain how attacker can exploit human vulnerabilities to get malicious programs installed.
2. An organization wishes to prevent installation of malware, so it publishes to its employees a list of permitted software vendors, so that employees would verify their names before installing. Present criticism.
3. Propose an alternative method that an operating system could offer, that would allow organization a more secure way to ensure that only programs from permitted vendors would be run.

## Chapter 10

# Review exercises and solutions

### 10.1 Review exercises

In this section we present additional exercises, which are intended to be used to review the entire area; namely, solving the exercises requires identification of the relevant ‘tools’, e.g., should we use MAC, signatures, hashing, PRF, encryption - or another tool?

**Exercise 10.1** (Authorized-operations application). *The US Customs allows imported containers to be stored in special bonded warehouses, from which they are released to importers after payment of custom fees, and possibly, inspections. Specifically, containers are to be accepted and released, only upon receiving an appropriate approval slip from US Customs, containing approval code and details (container number, date, accept/release instruction, and other text - up to 1000 characters). Upon receiving an approval slip, the warehouse sends a receipt back to US customs.*

*Mistakes and fraud may happen, either at the warehouse or at the US customs, involving accepting and releasing containers without valid approval slip. The warehouse keeps a log of all approval slips received.*

1. *Design an efficient and secure processes for generating the approval codes (by the customs) and validating them, by the warehouse as well as by an arbitrary auditor. Your design should consist of two functions: generate code and validate approval slip.*
2. *Add two more functions: generate receipt (run by the warehouse) and validate receipt (run by US customs or arbitrary auditor).*
3. *Extend your design to allow the auditor to validate also the integrity of the log, i.e., to validate that the log contains all approval slips for which the customs received (valid) receipts. In this part, do not (yet) optimize your solution for efficiently auditing (validating) a very large number of approvals.*

4. Optimize your solution for efficiently auditing a very large number of approvals.

## 10.2 Solutions to selected exercises

In this section we present solutions to some exercises (beyond the solutions and hints presented earlier).

### Solution to Exercise 2.8

Define the (stateful) encryption and decryption functions  $\langle E, D \rangle$  for the OTP cipher.

*Solution:* We use the index  $i$  of the next bit to be encrypted as the state. Encryption  $E_s(m_i, i)$  returns  $(m_i \oplus PRG_i(s), i + 1)$ , and decryption  $D_s(c_i, i)$  returns  $(c_i \oplus PRG_i(s), i + 1)$ .  $\square$

### Solution to Exercise 2.31

1. The question was to prove that  $G$  is (or isn't) a secure PRG, provided that one or both of  $G_1, G_2$  is a secure PRG. And the answer to this part is *no, it isn't*. The proof is quite simple: we show a counterexample. To do this, we assume we are given a secure PRG, let's denote it  $G'$ . We now set both  $G_1$  and  $G_2$  to be equal to  $G'$ , i.e.,  $G_1(s) = G'(s)$  and  $G_2(s) = G'(s)$ . So trivially they are both secure PRGs... but surely for any seed/key  $s$  holds:

$$G(s) = G_1(s) \oplus G_2(s) = G'(s) \oplus G'(s) = 0^{2n}$$

Namely, the output is a string of zero bits, whose length is  $|G'(s)|$ , i.e.,  $2n$ .

2. This is again NOT a secure PRG. The argument is very similar to the previous one. The only difference is that we set  $G_2(s) = G'(s \oplus 1^{|s|})$ .
3. The output of  $G_2$  will be fixed since its input is fixed. But the question is if  $G$  is a secure PRG, so the fact that the output of  $G_2$  is fixed doesn't imply  $G$  isn't a vsecure PRG - on the contrary! The adversary can obviously compute the output of  $G_2(0^{|s|})$  by herself, without knowing  $s$  (just knowing its length); so if the adversary can distinguish btw  $G(s)$  and a random string, then the adversary can also distinguish between  $G_1(s)$  and a random string, i.e.,  $G_1$  isn't a PRG. So it follows that in this case,  $G$  is a PRG if and only if  $G_1$  is a PRG.

$\square$

### Solution to Exercise 2.51

Consider a set  $P$  of  $n$  sensitive (plaintext) records  $P = \{p_1, \dots, p_n\}$  belonging to Alice, where  $n < 10^6$ . Each record  $p_i$  is  $l > 64$  bits long ( $(\forall i)(p_i \in \{0, 1\}^l)$ ). Alice has very limited memory, therefore, she wants to store an encrypted version of her records in an insecure/untrusted cloud storage server  $S$ ; denote these ciphertext records by  $C = \{c_1, \dots, c_n\}$ . Alice can later retrieve the  $i^{\text{th}}$  record, by sending  $i$  to  $S$ , who sends back  $c_i$ , and then decrypting it back to  $p_i$ .

1. Alice uses some secure shared key encryption scheme  $(E, D)$ , with  $l$  bit keys, to encrypt the plaintext records into the ciphertext records. The goal of this part is to allow Alice to encrypt and decrypt each record  $i$  using a unique key  $k_i$ , but maintain only a single ‘master’ key  $k$ , from which it can easily compute  $k_i$  for any desired record  $i$ . One motivation for this is to allow Alice to give keys to specific record(s)  $k_i$  to some other users (Bob, Charlie,...), allowing decryption of only the corresponding ciphertext  $c_i$ , i.e.,  $p_i = D_{k_i}(c_i)$ . Design how Alice can compute the key  $k_i$  for each record  $(i)$ , using only the key  $k$  and a secure block cipher (PRP)  $(F, F^{-1})$ , with key and block sizes both  $l$  bits. Your design should be as efficient and simple as possible. Note: do not design how Alice gives  $k_i$  to relevant users - e.g., she may do this manually; and do not design  $(E, D)$ .

*Solution:*  $k_i = F_k(i)$

2. Design now the encryption scheme to be used by Alice (and possibly by other users to whom Alice gave keys  $k_i$ ). You may use the block cipher  $(F, F^{-1})$ , but not other cryptographic functions. You may use different encryption scheme  $(E^i, D^i)$  for each record  $i$ . Ensure confidentiality of the plaintext records from the cloud, from users (not given the key for that record), and from eavesdroppers on the communication. Your design should be as efficient as possible, in terms of the length of the ciphertext (in bits), and in terms of number of applications of the secure block cipher (PRP)  $(F, F^{-1})$  for each encryption and decryption operation. In this part, assume that Alice stores  $P$  only once, i.e., never modifies records  $p_i$ . Your solution may include a new choice of  $k_i$ , or simply use the same as in the previous part.

*Solution:*  $k_i = F_k(i)$ ,

$$E_{k_i}^i(p_i) = k_i \oplus p_i,$$

$$D_{k_i}^i(c_i) = k_i \oplus c_i.$$

3. Repeat, when Alice may modify each record  $p_i$  few times (say, up to 15 times); let  $n_i$  denote number of modifications of  $p_i$ . The solution should allow Alice to give (only) her key  $k$ , and then Bob can decrypt all records, using only the key  $k$  and the corresponding ciphertexts from the server. Note: if your solution is the same as before, this *may* imply that your solution to the previous part is not optimal.

*Solution:*  $k_i = F_k(i||n_i)$ ,  
 $E_{k_i}^i(p_i) = (n_i, k_i \oplus p_i)$ ,  
 $D_{k_i}^i(c_i) = D_{k_i}^i((n_i, c'_i)) = k_i \oplus c'_i$ . Note:  $n_i$  encoded as four bits (for efficiency).

4. Design an efficient way for Alice to validate the integrity of records retrieved from the cloud server  $S$ . This may include storing additional information  $A_i$  to help validate record  $i$ , and/or changes to the encryption scheme or keys as designed in previous parts. As in previous parts, your design should only use the block cipher  $(F, F^{-1})$ .

*Solution:* compute  $k_i$  as before and append to any ciphertext  $c_i$  a *MAC* value  $m_i = F_{k_i}(c_i)$ . Use the stored MAC value  $m_i$  to validate  $c_i$  upon retrieval.

5. Extend the keying scheme from the first part, to allow Alice to also compute keys  $k_{i,j}$ , for integers  $i, j \geq 0$  s.t.  $1 \leq i \cdot 2^j + 1, (i+1) \cdot 2^j \leq n$ , where  $k_{i,j}$  would allow (efficient) decryption of ciphertext records  $c_{i \cdot 2^j + 1}, \dots, c_{(i+1) \cdot 2^j}$ . For example,  $k_{0,3}$  allows decryption of records  $c_1, \dots, c_8$ , and  $k_{3,2}$  allows decryption of records  $c_{13}, \dots, c_{16}$ . If necessary, you may also change the encryption scheme  $(E^i, D^i)$  for each record  $i$ .

*Solution:* Assume for simplicity that  $n = 2^m$ , or, to avoid this assumption, let  $m = \lceil \log_2(n) \rceil$ , i.e.,  $n \leq 2^m$ . Let  $k_i = k_{i,m}$ , where we compute  $k_{i,m}$  be the following iterative process. First, let  $k_{1,0} = k$ . Then, for  $j = 1, \dots, m$ , we compute  $k_{i,j}$ , for  $i = \{1, 2, \dots, 2^j\}$ , as follows:

$$k_{i,j} = F_{k_{\lfloor \frac{i+1}{2} \rfloor, j-1}}(imod2)$$

### Solution to Exercise 5.19

1. The attack is simply a replay of an handshake - once the base reuses the same triplet (and specifically sends the same  $r$ ), the attacker simply replays the  $s$  value sent the user (and later messages).
2. The improvement to the protocol is simply to compute  $s_B = PRF_s(r_B)$  where  $s$  is derived as in the original protocol.
- 3.

### Solution to Exercise 6.23

Figure 6.25 shows a vulnerable variant of the Ratchet DH protocol, using a (secure) pseudorandom function  $f$  to derive the session key. Assume that this protocol is run daily, from day  $i = 1$ , and where  $k_0$  is a randomly-chosen secret initial key, shared between Alice and Bob; messages on day  $i$  are encrypted using key  $k_i$ . An attacker can eavesdrops on the communication between the parties on all days, and on days 3, 6, 9, ... it can also spoof messages (send messages impersonating as either Alice or Bob), and act as Monster-in-the-Middle (MitM). On the fifth day ( $i = 5$ ), the attacker is also given the initial key  $k_0$ .

- On which day can attacker first decrypt messages?  
*Answer:* fifth day.
- On the day you specified, what are the days that the attacker can decrypt messages of?  
*Answer:* all days till then (i.e., first to fifth) - would also be able to decrypt messages on following days, once sent.
- Explain the attack , including a sequence diagram if relevant. Include every calculation done by the attacker.  
*Answer:* The protocol (of Figure 6.25) contains a mistake: instead of the DH value  $g^{a_i \cdot b_i} \pmod{p}$ , it uses  $g^{a_i + b_i} \pmod{p}$ ; but this can be easily computed by an eavesdropper from the values sent by the parties, i.e., after each round  $i$  the attacker computes:

$$x_i \equiv g^{a_i + b_i} \pmod{p} = (g^{a_i} \pmod{p}) \cdot (g^{b_i} \pmod{p}) \pmod{p}$$

Hence, this attack requires only a passive, eavesdropping capabilities for the attacker, and there is no need in sequence diagram - the protocol is simply run as designed, i.e., on the  $i^{th}$  day, we have the exchange as in Figure 6.25.

On fifth day, once  $k_0$  is known, the attacker can also compute  $k_i = f_{k_{i-1}}(x_i)$  for all days till then, i.e.,  $i \in \{1, \dots, 5\}$ ; and on each following day  $j > 5$ , attacker can compute the day's key immediately after the parties perform the daily exchange.

### Solution to Exercise 6.35

Many applications require both confidentiality, using recipient's public encryption key, say  $B.e$ , and non-repudiation (signature), using sender's verification key, say  $A.v$ . Namely, to send a message to Bob, Alice uses both her private signature key  $A.s$  and Bob's public encryption key  $B.e$ ; and to receive a message from Alice, Bob uses his private decryption key  $B.d$  and Alice's public verification key  $A.v$ .

1. It is proposed that Alice will select a random key  $k$  and send to Bob the triplet:  $(c^K, c^M, \sigma) = (E_{B.e}(k), k \oplus m, \text{Sign}_{A.s}(\text{'Bob' || } k \oplus m))$ . Show this design is insecure, i.e., a MitM attacker may either learn the message  $m$  or cause Bob to receive a message 'from Alice' - that Alice never sent.

*Answer:* MitM attacker Monster captures the specified triplet  $(x, c, t)$  sent by Alice to Bob, where  $x = E_{B.e}(k)$ ,  $c = k \oplus m$  and  $t = \text{Sign}_{A.s}(\text{'Bob' || } k \oplus m)$ . Monster sends to Bob a modified triplet:  $(x', c, t)$ , i.e., it modifies just the first element in the triplet; this allows Monster to ensure that Bob will receive a message  $m'$  chosen by Monster, instead of  $m$  (and believe Bob will believe  $m'$  was sent by Alice). To do so, Monster sets  $x' = E_{B.e}(c \oplus m')$ .

2. Propose a simple, efficient and secure fix. Define the sending and receiving process precisely.

*Answer:* Sign also the ciphertext, i.e., change the last component in the triplet to  $t = \text{Sign}_{A.s}('Bob'||c||k \oplus m)$ .

3. Extend your solution to allow prevention of replay (receiving multiple times a message sent only once).

*Answer:* Sign also a counter (or time-stamp - any monotonously-increasing value), i.e., change the last component in the triplet to  $t = \text{Sign}_{A.s}('Bob'||i||c||k \oplus m)$  where  $i$  is the number of messages sent by Alice so far; each recipient has to keep the last message number received from any recipient.

# Index

- (PFS), 218
- 2PP, 159, 161
- AEAD, 154
- AES, 199
  - asymmetric cryptography, 184
  - asymmetric cryptology, 193
- Asynchronous-DH-Ratchet, 217
- attack model, 10
- attributes, 277
- Authenticated DH, 214, 215, 236
- authenticated key-exchange, 208
- authenticated-encryption with associated data, 154
- backward compatibility, 258
- basic constraints, 293, 296, 298
- birthday paradox, 117
- bitmask, 133
- bitwise-randomness extracting, 134
- block ciphers, 12, 39
- Blockchain, 113
- blockchain, 121
- BREACH, 155
- CA, 229, 276, 277
- CA certificates, 292
- CCA, 23, 24, 46
- CDH, 211, 212
- Certificate, 277
  - certificate authorities, 276, 278, 315
  - Certificate Authority, 276
  - certificate authority, 276, 277
  - Certificate policy, 292
  - certificate policy, 292
  - certificate revocation list, 299, 300
- Certificate Transparency, 278, 315
- certification path, 288, 294
- cetificate authority, 229
- Checksum, 153
- chosen-ciphertext, 24
- chosen-ciphertext attack, 23
- chosen-plaintext, 24
- chosen-plaintext attack, 23
- Cipher-agility, 176
- cipher-agility, 252
- ciphersuite, 176
  - ciphersuite negotiation, 156, 176
- ciphertext, 23
- Ciphertext-Only, 59
- ciphertext-only, 23, 24
- ciphertext-only attack, 20
- client authentication, 277
- client certificate, 277
- client-server, 171
- collision, 115
- collision-resistant, 115
- Compress-and-Encrypt Vulnerability, 155
- Computational DH, 211
- CP, 292, 307
- CPA, 23, 24
- CRIME, 155
- critical extensions, 287
- criticality indicator, 287
- CRL, 299, 300
- cryptographic building blocks principle, 252
- cryptographic hash function, 39
- CT, 278, 315
- CTO, 20, 21, 23, 24, 42, 46, 47, 59

cyclic group, 203  
 Data Encryption Standard, 7, 25  
 DDH, 212  
 Decisional DH, 212  
 Denial-of-Service, 314  
 denial-of-service, 158  
 DES, 7, 25  
 DH, 183, 199, 203  
 DH key exchange, 193  
 DH PKC, 220  
 DH protocol, 215  
 DH public key cryptosystem, 220  
 Diffie-Hellman, 199, 203, 210  
 Diffie-Hellman Key Exchange, 7  
 Diffie-Hellman Ratchet, 216  
 Diffie-Hellmen, 183  
 digital signature, 194, 195  
 digital signatures, 193  
 discrete logarithm, 202, 203  
 discrete logarithm assumption, 204  
 Distinguished Name, 280, 290  
 distinguished name, 280  
 distinguisher, 31  
 DLA, 204  
 DN, 280  
 DNS, 290  
 dNSName, 290  
 Domain Name System, 290  
 domain validation, 312  
 domain-validated, 316  
 DoS, 314  
 Double Ratchet Key-Exchange, 218  
 Double-Ratchet Key-Exchange, 218, 219  
 Double-ratchet key-exchange, 214  
 Downgrade Attack, 176  
 downgrade attack, 176, 252  
 eavesdropper, 196  
 eavesdropping adversary, 206  
 ECC, 47, 153  
 ECIES, 200  
 EDC, 153  
 effective key length, 117  
 El-Gamal, 199, 203

El-Gamal PKC, 221  
 Encrypt-then-Authenticate, 154  
 entity authentication, 155  
 Error Correction Code, 153  
 Error Detection Code, 153  
 Error-Correcting Code, 47  
 Euler's criterion, 204, 212  
 Euler's function, 224  
 Euler's Theorem, 224  
 evidence, 196  
 exhaustive search, 23, 199  
 Extract-then-Expand, 212  
 extract-then-expand, 135  
 factoring, 202  
 Feedback Shift Register, 34  
 FIL, 30, 32  
 fixed input length, 30, 32  
 Forward Secrecy, 192  
 forward secrecy, 180–182, 185, 197  
 freshness, 167  
 FSR, 34  
 generic attacks, 23  
 GSM, 21, 22, 34, 47, 59, 67, 170  
 GSM handshake protocol, 189  
 GSM security, 156, 172, 174  
 Hardware Security Module, 168, 180  
 hash-block, 113  
 hash-chain, 114, 130  
 Hash-then-Sign, 123  
 hash-then-sign, 113, 123, 201, 285  
 HMAC, 136, 214  
 homomorphic encryption, 221  
 HPKP, 313  
 HSM, 168, 180  
 HSTS, 259, 260  
 HtS, 123, 201  
 HTTP Strict Transport Security, 259  
 hybrid encryption, 201, 224  
 IETF, 288  
 IMSI, 173  
 IND-CCA, 46, 56, 57, 63  
 IND-CPA, 44, 46, 50, 56, 62, 219

independently pseudorandom, 41  
 intermediate CA, 293  
 International Mobile Subscriber Identity, 173  
 International Telecommunication Union, 279  
 IPsec, 152  
 issuer, 276, 277  
 Issuer Alternative Name, 290  
 IssuerAltName, 290  
 ITU, 279, 285, 288  
 KDC, 170, 197  
 KDF, 212–214  
 Kerberos, 172  
 Kerckhoffs' principle, 7  
 Kerckhoffs' Principle, 14  
 Key derivation, 213  
 key derivation, 213  
 Key Derivation Function, 212, 213  
 Key Distribution, 156  
 Key distribution Center, 170  
 Key Distribution Protocol, 170  
 Key exchange, 195  
 key exchange, 193, 194, 205  
 key generation, 194  
 key usage, 286  
 key-derivation function, 220  
 key-exchange, 183  
 key-length principle, 24  
 Key-Reinstallation Attack, 155  
 key-separation principle, 40, 169, 185  
 key-setup, 155  
 key-setup 2PP extension, 169, 181  
 known-plaintext, 23, 24  
 KPA, 23, 24, 46  
 KRACK, 155  
 letter-frequency attack, 20, 23, 24  
 MAC, 179, 195, 197, 215  
 master key, 215  
 Merkle-Damgard strengthening, 143  
 Merkle-tree, 122  
 Message Authentication Code, 195, 215  
 message recovery, 231  
 MitM, 151, 162, 167, 176, 177, 197, 206, 208, 211, 228, 229, 252, 276  
 Monster in the Middle, 151  
 monster-in-middle, 229  
 Monster-in-the-Middle, 162, 176, 208, 252  
 must-staple, 310, 322  
 naming constraints, 286, 293, 298  
 non-critical extensions, 287  
 non-repudiation, 228, 230  
 nonce, 159, 162  
 nonces, 167  
 Object Identifier, 284  
 object identifier, 285, 286  
 OCSP requests for Certificate-Path, 307  
 OID, 284–286, 292  
 one-time password, 130  
 one-time-pad, 183  
 One-Way Function, 129  
 one-way function, 129  
 one-way functions, 39  
 oracle access, 11  
 OTP, 130, 183  
 OTP-chain, 114, 130  
 OWF, 113, 129, 130  
 Per-goal keys, 41  
 per-goal keys, 169  
 Perfect Forward Secrecy, 183, 184, 186, 197, 215, 216  
 Perfect forward secrecy, 255  
 perfect forward secrecy, 180, 185  
 Perfect Recover Secrecy, 183, 184, 215, 216  
 perfect recover secrecy, 180, 184, 185, 187  
 Perfect Recover Security, 216  
 perfect-forward secrecy, 218  
 perfect-recover security, 218  
 PFS, 180, 183–186, 197, 214–216, 255

PGP, 320  
 PKC, 7, 184, 194, 197, 198, 205, 219  
 PKI, 278  
 PKIX, 288, 290  
 plaintext, 23  
 Pohlig-Hellman algorithm, 204  
 policy constraints, 293, 298  
 policy mapping, 299  
 PoW, 111, 114, 137  
 PPT, 202, 204, 211, 213  
 Preimage resistance, 129  
 preimage resistant, 129  
 preloaded, 260  
 PRF, 69, 111, 173, 185, 214  
 PRG, 29, 32, 69, 183  
 private key, 194  
 Proactive security, 197  
 proactive security, 184  
 Probabilistic Polynomial Time, 202  
 Proof of Work, 137  
 Proof-of-Work, 111, 113, 114  
 protocols, 149  
 proxy re-encryption, 222  
 PRS, 180, 183–185, 187, 214–216  
 Pseudo Random Function, 173  
 Pseudo-Random Function, 185, 214  
 pseudo-random function, 111, 185  
 Pseudo-Random Generator, 32  
 pseudo-random generator, 29, 183  
 public key, 194  
 public key certificate, 197, 229, 276,  
     277, 281  
 Public Key Cryptology, 7  
 Public key cryptology, 197  
 public key cryptology, 183, 185, 193,  
     205  
 Public key cryptosystem, 194  
 public key cryptosystem, 194  
 public key cryptosystems, 219  
 Public Key Infrastructure, 278, 288  
 public key(s), 279  
 public-key cryptography, 152, 184  
 public-key cryptosystem, 199, 205  
 public-key encryption, 193  
 quadratic residue, 204  
 Random Oracle Methodology, 135  
 Random oracle methodology, 112  
 Random Oracle Methodology (ROM),  
     135  
 random oracle model, 125  
 randomness extracting hash func-  
     tion, 213  
 randomness extraction, 132  
 RC4, 34  
 re-encryption, 222  
 record protocol, 152  
 recover secrecy, 180, 182, 183, 185,  
     187  
 Recover-Security Handshake, 182, 218  
 relying parties, 279  
 relying party, 276, 277, 287, 296  
 resiliency to exposure, 197  
 Resiliency to Exposures, 214  
 resiliency to exposures, 156  
 Resiliency to key exposure, 180  
 resiliency to key exposure, 183  
 robust combiner, 39, 245  
 ROM, 113, 125, 135  
 root CA, 293  
 root CAs, 278  
 RS-Ratchet, 182, 216  
 RSA, 26, 199, 202, 225  
 RSA assumption, 225  
 RSA signatures, 231  
 S/MIME, 277  
 safe prime, 204, 210, 212  
 salt, 213  
 SAN, 290  
 Schnorr’s group, 212  
 SCSV, 258  
 SCT, 316  
 second preimage resistance, 111  
 second-preimage resistance, 125  
 second-preimage resistant, 126  
 secure in the standard model, 136  
 Secure Session Protocols, 152  
 secure session transmission proto-  
     col, 152  
 Secure Socket Layer, 152  
 security parameter, 162

security requirements, 10  
session key, 155  
session protocol, 152  
Session Ticket Encryption Key (STEK), 267  
session-authentication, 155  
shared-key authentication-handshake protocols, 155  
Shared-key Entity-Authenticating Handshake Protocols, 156  
Signaling Cipher Suite Value, 258  
signature scheme, 195, 227  
signature schemes, 230  
signature with appendix, 230  
signature with message recovery, 230, 231  
Signed Certificate Timestamp, 316  
smooth, 205  
SNA, 159–161  
SNA handshake protocol, 159, 161  
SPR, 111, 125, 126  
SSL, 152, 153, 276  
SSL-Stripping, 259  
stream ciphers, 12  
subject, 277  
SubjectAltName, 290  
subsequent certificates, 296, 298  
sufficient effective key length, 199  
symmetric cryptography, 170  
symmetric cryptosystem, 199  
Synchronous DH Ratchet, 216  
Systems Network Architecture, 159

table look-up, 23  
table look-up attack, 25  
textbook RSA, 224  
TextSecure, 218  
Threshold security, 197  
threshold security, 184  
TIME, 155  
time-memory tradeoff, 23  
Timestamp, 166  
timestamp, 167  
Timestamps, 167  
TLDcc, 313  
TLS, 152, 153, 155, 241, 276

TLS feature, 287  
TLS\_FALLBACK\_SCSV, 259  
to-be-signed, 285  
TOFU, 228, 313  
top-level domain country codes, 313  
Transport-Layer Security, 152, 241  
truncation, 153  
trust anchor, 288–290, 292, 293, 296  
trust on first use, 228  
Trust-On-First-Use, 313  
trusted third party, 170  
TTP, 170

universal re-encryption, 222  
UX beats Security Principle, 305

variable input length, 30, 32  
VIL, 30, 32

weak collision resistance, 126  
Web-of-Trust, 320  
web-of-trust, 293  
WEP, 63  
Wi-Fi Protected Access, 63  
wildcard certificates, 291  
wildcard character, 291  
Wired Equivalency Privacy, 63  
WPA, 63  
WPA2, 155

X.500, 279, 280  
X.509, 288

# Bibliography

- [1] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr. Rsa and rabin functions: Certain parts are as hard as the whole. *SIAM Journal on Computing*, 17(2):194–209, 1988.
- [2] J.-P. Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Stratch Press, 2017.
- [3] J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein. Blake2: Simpler, smaller, fast as md5. In M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.
- [4] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Kasper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium*, Aug. 2016.
- [5] G. V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In M. Malek, E. Fernández-Medina, and J. Hernando, editors, *Proceedings of SECRYPT*, pages 99–109. INSTICC Press, 2006.
- [6] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *J. Cryptology*, 21(3):392–429, 2008.
- [7] E. Barker. Nist special publication 800-57 part 1 revision 4—recommendation for key management (part 1: General), 2016.
- [8] L. Bassham, W. Polk, and R. Housley. Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3279 (Proposed Standard), Apr. 2002. Updated by RFCs 4055, 4491, 5480, 5758.
- [9] T. Be’ery and A. Shulman. A Perfect CRIME? Only TIME Will Only Tell. In *Blackhat Europe*, March 2013.

- [10] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology—CRYPTO ’96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 18–22 Aug. 1996.
- [11] M. Bellare, R. Canetti, and H. Krawczyk. HMAC: Keyed-hashing for message authentication. Internet Request for Comment RFC 2104, Internet Engineering Task Force, Feb. 1997.
- [12] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 419–428. ACM, 1998.
- [13] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *J. Comput. Syst. Sci.*, 61(3):362–399, 2000.
- [14] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 21(4):469–491, Oct. 2008.
- [15] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Crypto*, volume 93, pages 232–249. Springer, 1993.
- [16] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73, 1993.
- [17] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 92–111. Springer, 1994.
- [18] M. Bellare and P. Rogaway. Provably secure session key distribution: the three party case. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 57–66. ACM, 1995.
- [19] S. M. Bellovin. Frank Miller: Inventor of the One-Time Pad. *Cryptologia*, 35(3):203–222, 2011.
- [20] S. M. Bellovin. Vernam, Mauborgne, and Friedman: The One-Time Pad and the index of coincidence. In P. Y. A. Ryan, D. Naccache, and J.-J. Quisquater, editors, *The New Codebreakers*, volume 9100 of *Lecture Notes in Computer Science*, pages 40–66. Springer, 2016.
- [21] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung. The kryptoknight family of light-weight protocols for authentication and key distribution. *IEEE/ACM Transactions on Networking (TON)*, 3(1):31–41, 1995.

- [22] R. Bird, I. Gopal, A. Herzberg, P. A. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of a family of attack-resistant authentication protocols. *IEEE Journal on Selected Areas in Communications*, 11(5):679–693, 1993.
- [23] J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In K. Nyberg and H. M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.
- [24] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003. Obsoleted by RFC 4366.
- [25] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), Apr. 2006. Obsoleted by RFCs 5246, 6066, updated by RFC 5746.
- [26] M. Blaze, G. Bleumer, and M. S. 0001. Divertible protocols and atomic proxy cryptography. In K. Nyberg, editor, *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 1998.
- [27] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.
- [28] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM-01)*, pages 180–188, New York, July 16–21 2001. ACM Press.
- [29] BSI. Kryptographische verfahren: Empfehlungen und schlussell, February 2017.
- [30] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 3(1):1–8, 1997.
- [31] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of dual ec in tls implementations. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 319–335. USENIX Association, 2014.

- [32] S. Cohney, M. D. Green, and N. Heninger. Practical state recovery attacks against legacy RNG implementations, October 2017. online at <https://duhkattack.com/paper.pdf>.
- [33] I. C. S. L. M. S. Committee. *IEEE 802.11: Wireless LAN Medium Access Control and Physical Layer Specifications*, Aug. 1999.
- [34] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFCs 6818, 8398, 8399.
- [35] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
- [36] W. Dai. Crypto++ 6.0.0 benchmarks, 2018. version of 01-23-2018.
- [37] Y. Desmedt. Threshold cryptosystems. In *International Workshop on the Theory and Application of Cryptographic Techniques*, pages 1–14. Springer, 1992.
- [38] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507, 7919.
- [39] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447.
- [40] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.
- [41] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, Mar. 1979.
- [42] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In M. Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.
- [43] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM review*, 45(4):727–784, 2003.
- [44] N. Doraswamy and D. Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

- [45] T. Duong and J. Rizzo. Here come the XOR ninjas. presented at Ecoparty and available at <http://www.hpcc.ecs.soton.ac.uk/~dan/talks/bullrun/Beast.pdf>, 2011.
- [46] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the https certificate ecosystem. In K. Papagiannaki, P. K. Gummadi, and C. Partridge, editors, *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*, pages 291–304. ACM, 2013.
- [47] M. J. Dworkin. Recommendation for block cipher modes of operation: The cmac mode for authentication. *Special Publication (NIST SP)-800-38B*, 2016.
- [48] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 293–302. IEEE, 2008.
- [49] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), Apr. 2015.
- [50] L. Ewing. Linux 2.0 penguins. Example of using The GIMP graphics software, online; accessed 1-Sept-2017.
- [51] P.-A. Fouque, D. Pointcheval, J. Stern, and S. Zimmer. Hardness of distinguishing the msb or lsb of secret keys in diffie-hellman schemes. In *International Colloquium on Automata, Languages, and Programming*, pages 240–251. Springer, 2006.
- [52] S. Frankel and S. Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071 (Informational), Feb. 2011.
- [53] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), Aug. 2011.
- [54] S. Garfinkel and N. Makarevitch. *PGP: Pretty Good Privacy*. O'Reilly International Thomson, Paris, France, 1995.
- [55] Y. Gilad, A. Herzberg, M. Sudkowitch, and M. Goberman. Cdn-on-demand: An affordable ddos defense via untrusted clouds. In *NDSS*. The Internet Society, 2016.
- [56] D. Giry. Cryptographic key length recommendation, 2018. version of 01-23-2018.
- [57] O. Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.
- [58] O. Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

- [59] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
- [60] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, Apr. 1984.
- [61] P. Golle, M. Jakobsson, A. Juels, and P. Syverson. Universal re-encryption for mixnets. In *Cryptographers' Track at the RSA Conference*, pages 163–178. Springer, 2004.
- [62] P. Hallam-Baker. X.509v3 Transport Layer Security (TLS) Feature Extension. RFC 7633 (Proposed Standard), Oct. 2015.
- [63] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.
- [64] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Information Theory*, 26(4):401–406, 1980.
- [65] A. Herzberg. Folklore, practice and theory of robust combiners. *Journal of Computer Security*, 17(2):159–189, 2009.
- [66] K. E. Hickman and T. Elgamal. The ssl protocol (version 2), 1995. Published as Internet Draft draft-hickman-netscape-ssl-01.txt.
- [67] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.
- [68] J. Jean. TikZ for Cryptographers. <https://www.iacr.org/authors/tikz/>, 2016. CBC figures by Diana Maimut.
- [69] D. Kahn. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet*. Simon and Schuster, 1996.
- [70] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In S. Vaudenay, editor, *Fast Software Encryption: 5th International Workshop*, volume 1372 of *Lecture Notes in Computer Science*, pages 168–188, Paris, France, 23–25 Mar. 1998. Springer-Verlag.
- [71] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX, 1883.
- [72] S. Kille. A String Representation of Distinguished Names. RFC 1779 (Historic), Mar. 1995. Obsoleted by RFCs 2253, 3494.
- [73] C. A. Kirtchev. A cyberpunk manifesto. *Cyberpunk Review*. Disponible en <http://www.cyberpunkreview.com/wiki/index.php>, 1997.
- [74] A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, 2008.

- [75] P. Koopman. 32-bit cyclic redundancy codes for internet applications. In *Proceedings 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 459–472, (Bethesda, MD) Washington, DC, USA, June 2002. IEEE Computer Society.
- [76] H. Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). In J. Kilian, editor, *Advances in Cryptology – CRYPTO ’2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2001.
- [77] H. Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In T. Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, 2010.
- [78] J. F. Kurose and K. W. Ross. *Computer networking: a top-down approach*, volume 4. Addison Wesley Boston, 2009.
- [79] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.
- [80] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
- [81] S. Levy. *Crypto: how the code rebels beat the government, saving privacy in the digital age*. Viking, 2001.
- [82] S. Ling, H. Wang, and C. Xing. *Algebraic Curves in Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 1 edition, 2013.
- [83] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, Apr. 1988.
- [84] I. Mantin and A. Shamir. A practical attack on broadcast RC4. In *International Workshop on Fast Software Encryption*, pages 152–164. Springer, 2001.
- [85] J. Mason, K. Watkins, J. Eisner, and A. Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 235–244. ACM, 2006.
- [86] C. Meyer and J. Schwenk. Lessons learned from previous SSL/TLS attacks - a brief chronology of attacks and weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.

- [87] B. Moeller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507 (Proposed Standard), Apr. 2015.
- [88] B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback, September 2014. Online, accessed 01-Sept-2017.
- [89] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications magazine*, 32(9):33–38, 1994.
- [90] R. Oppliger. *SSL and TLS: Theory and Practice*. Artech House, 2016.
- [91] R. Perlman, C. Kaufman, and M. Speciner. *Network security: private communication in a public world*. Pearson Education India, 2016.
- [92] Y. Pettersen. The Transport Layer Security (TLS) Multiple Certificate Status Request Extension. RFC 6961 (Proposed Standard), June 2013. Obsoleted by RFC 8446.
- [93] J. Postel. Domain Name System Structure and Delegation. RFC 1591 (Informational), Mar. 1994.
- [94] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard), Jan. 2010.
- [95] E. Rescorla. *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading, 2001.
- [96] 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, 1978.
- [97] J. Rizzo and T. Duong. Crime: Compression ratio info-leak made easy. In *ekoparty Security Conference*, 2012.
- [98] P. Rogaway. Authenticated-encryption with associated-data. In V. Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 98–107. ACM, November 2002.
- [99] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard), Mar. 2011.
- [100] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008. Obsoleted by RFC 8446, updated by RFC 8447.

- [101] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013.
- [102] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technicl Journal*, 28:656–715, Oct. 1949.
- [103] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457 (Informational), Feb. 2015.
- [104] S. Singh. *The Science of Secrecy: The Secret History of Codes and Code-breaking*. Fourth Estate, London, UK, 2001.
- [105] D. E. Standard et al. Federal information processing standards publication 46. *National Bureau of Standards, US Department of Commerce*, 1977.
- [106] D. R. Stinson. *Cryptography: theory and practice*. CRC press, 2005.
- [107] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.
- [108] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger. Measuring small subgroup attacks against diffie-hellman. In *NDSS’17*, 2017.
- [109] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.
- [110] S. Venkata, S. Harwani, C. Pignataro, and D. McPherson. Dynamic Hostname Exchange Mechanism for OSPF. RFC 5642 (Proposed Standard), Aug. 2009.
- [111] G. S. Vernam. Secret signaling system, July 22 1919. US Patent 1,310,719.
- [112] J. Von Neumann. Various techniques used in connection with random digits. *Applied Math Series*, 12(36-38):1, 1951.
- [113] D. Wagner, B. Schneier, et al. Analysis of the ssl 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, volume 1, pages 29–40, 1996.
- [114] N. Wiener. *Cybernetics, or control and communication in the animal and the machine*. John Wiley, New York, 1948.

- [115] Wikipedia. Block cipher mode of operation, 2017. [Online; accessed 1-Sept-2017].
- [116] K. Zuse. Method for automatic execution of calculations with the aid of computers (1936). In B. Randell, editor, *The Origins of Digital Computers: Selected Papers*, Texts and monographs in computer science, pages 163–170. Springer-Verlag, pub-SV:adr, third edition, 1982.