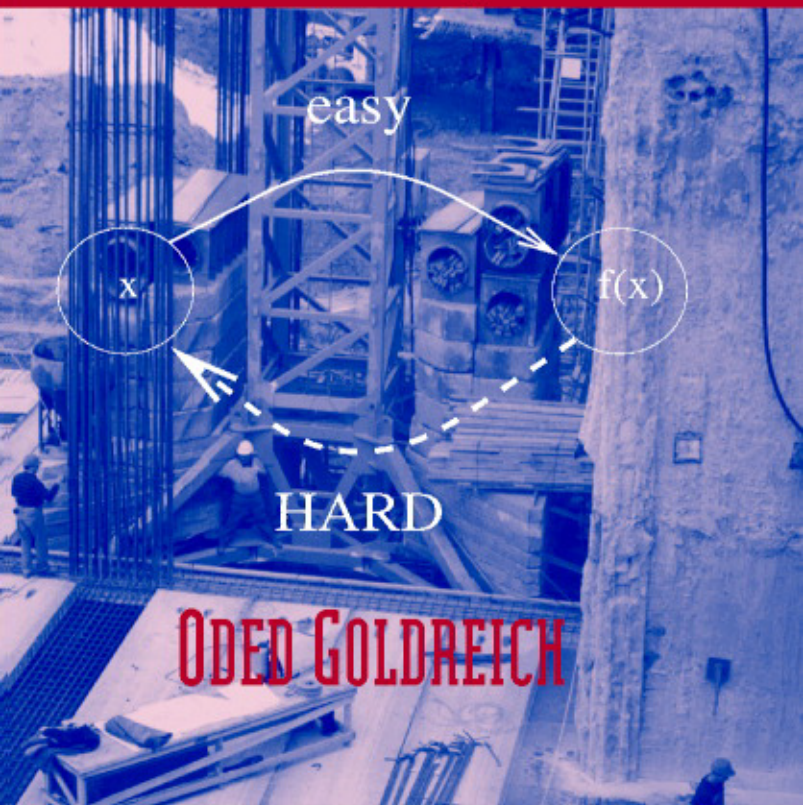


FOUNDATIONS OF CRYPTOGRAPHY

Basic Techniques



Foundations of Cryptography

Cryptography is concerned with the conceptualization, definition, and construction of computing systems that address security concerns. The design of cryptographic systems must be based on firm foundations. This book presents a rigorous and systematic treatment of the foundational issues: defining cryptographic tasks and solving new cryptographic problems using existing tools. It focuses on the basic mathematical tools: computational difficulty (one-way functions), pseudorandomness, and zero-knowledge proofs. The emphasis is on the clarification of fundamental concepts and on demonstrating the feasibility of solving cryptographic problems rather than on describing ad hoc approaches.

The book is suitable for use in a graduate course on cryptography and as a reference book for experts. The author assumes basic familiarity with the design and analysis of algorithms; some knowledge of complexity theory and probability is also useful.

Oded Goldreich is Professor of Computer Science at the Weizmann Institute of Science and incumbent of the Meyer W. Weisgal Professorial Chair. An active researcher, he has written numerous papers on cryptography and is widely considered to be one of the world experts in the area. He is an editor of *Journal of Cryptology* and *SIAM Journal on Computing* and the author of *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*, published in 1999 by Springer-Verlag.

Foundations of Cryptography

Basic Tools

Oded Goldreich

Weizmann Institute of Science



PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS

The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

<http://www.cambridge.org>

© Oded Goldreich 2004

First published in printed format 2001

ISBN 0-511-04120-9 eBook (netLibrary)

ISBN 0-521-79172-3 hardback

First published 2001

Reprinted with corrections 2003

To Dana

Contents

List of Figures	<i>page</i> xii
Preface	xiii
1 Introduction	1
1.1. Cryptography: Main Topics	1
1.1.1. Encryption Schemes	2
1.1.2. Pseudorandom Generators	3
1.1.3. Digital Signatures	4
1.1.4. Fault-Tolerant Protocols and Zero-Knowledge Proofs	6
1.2. Some Background from Probability Theory	8
1.2.1. Notational Conventions	8
1.2.2. Three Inequalities	9
1.3. The Computational Model	12
1.3.1. \mathcal{P} , \mathcal{NP} , and \mathcal{NP} -Completeness	12
1.3.2. Probabilistic Polynomial Time	13
1.3.3. Non-Uniform Polynomial Time	16
1.3.4. Intractability Assumptions	19
1.3.5. Oracle Machines	20
1.4. Motivation to the Rigorous Treatment	21
1.4.1. The Need for a Rigorous Treatment	21
1.4.2. Practical Consequences of the Rigorous Treatment	23
1.4.3. The Tendency to Be Conservative	24
1.5. Miscellaneous	25
1.5.1. Historical Notes	25
1.5.2. Suggestions for Further Reading	27
1.5.3. Open Problems	27
1.5.4. Exercises	28

2	Computational Difficulty	30
2.1.	One-Way Functions: Motivation	31
2.2.	One-Way Functions: Definitions	32
2.2.1.	Strong One-Way Functions	32
2.2.2.	Weak One-Way Functions	35
2.2.3.	Two Useful Length Conventions	35
2.2.4.	Candidates for One-Way Functions	40
2.2.5.	Non-Uniformly One-Way Functions	41
2.3	Weak One-Way Functions Imply Strong Ones	43
2.3.1.	The Construction and Its Analysis (Proof of Theorem 2.3.2)	44
2.3.2.	Illustration by a Toy Example	48
2.3.3.	Discussion	50
2.4.	One-Way Functions: Variations	51
2.4.1.*	Universal One-Way Function	52
2.4.2.	One-Way Functions as Collections	53
2.4.3.	Examples of One-Way Collections	55
2.4.4.	Trapdoor One-Way Permutations	58
2.4.5.*	Claw-Free Functions	60
2.4.6.*	On Proposing Candidates	63
2.5.	Hard-Core Predicates	64
2.5.1.	Definition	64
2.5.2.	Hard-Core Predicates for Any One-Way Function	65
2.5.3.*	Hard-Core Functions	74
2.6.*	Efficient Amplification of One-Way Functions	78
2.6.1.	The Construction	80
2.6.2.	Analysis	81
2.7.	Miscellaneous	88
2.7.1.	Historical Notes	89
2.7.2.	Suggestions for Further Reading	89
2.7.3.	Open Problems	91
2.7.4.	Exercises	92
3	Pseudorandom Generators	101
3.1.	Motivating Discussion	102
3.1.1.	Computational Approaches to Randomness	102
3.1.2.	A Rigorous Approach to Pseudorandom Generators	103
3.2.	Computational Indistinguishability	103
3.2.1.	Definition	104
3.2.2.	Relation to Statistical Closeness	106
3.2.3.	Indistinguishability by Repeated Experiments	107
3.2.4.*	Indistinguishability by Circuits	111
3.2.5.	Pseudorandom Ensembles	112
3.3.	Definitions of Pseudorandom Generators	112
3.3.1.	Standard Definition of Pseudorandom Generators	113

3.3.2.	Increasing the Expansion Factor	114
3.3.3.*	Variable-Output Pseudorandom Generators	118
3.3.4.	The Applicability of Pseudorandom Generators	119
3.3.5.	Pseudorandomness and Unpredictability	119
3.3.6.	Pseudorandom Generators Imply One-Way Functions	123
3.4.	Constructions Based on One-Way Permutations	124
3.4.1.	Construction Based on a Single Permutation	124
3.4.2.	Construction Based on Collections of Permutations	131
3.4.3.*	Using Hard-Core Functions Rather than Predicates	134
3.5.*	Constructions Based on One-Way Functions	135
3.5.1.	Using 1-1 One-Way Functions	135
3.5.2.	Using Regular One-Way Functions	141
3.5.3.	Going Beyond Regular One-Way Functions	147
3.6.	Pseudorandom Functions	148
3.6.1.	Definitions	148
3.6.2.	Construction	150
3.6.3.	Applications: A General Methodology	157
3.6.4.*	Generalizations	158
3.7.*	Pseudorandom Permutations	164
3.7.1.	Definitions	164
3.7.2.	Construction	166
3.8.	Miscellaneous	169
3.8.1.	Historical Notes	169
3.8.2.	Suggestions for Further Reading	170
3.8.3.	Open Problems	172
3.8.4.	Exercises	172
4	Zero-Knowledge Proof Systems	184
4.1.	Zero-Knowledge Proofs: Motivation	185
4.1.1.	The Notion of a Proof	187
4.1.2.	Gaining Knowledge	189
4.2.	Interactive Proof Systems	190
4.2.1.	Definition	190
4.2.2.	An Example (Graph Non-Isomorphism in \mathcal{IP})	195
4.2.3.*	The Structure of the Class \mathcal{IP}	198
4.2.4.	Augmentation of the Model	199
4.3.	Zero-Knowledge Proofs: Definitions	200
4.3.1.	Perfect and Computational Zero-Knowledge	200
4.3.2.	An Example (Graph Isomorphism in \mathcal{PZK})	207
4.3.3.	Zero-Knowledge with Respect to Auxiliary Inputs	213
4.3.4.	Sequential Composition of Zero-Knowledge Proofs	216
4.4.	Zero-Knowledge Proofs for \mathcal{NP}	223
4.4.1.	Commitment Schemes	223
4.4.2.	Zero-Knowledge Proof of Graph Coloring	228

4.4.3.	The General Result and Some Applications	240
4.4.4.	Second-Level Considerations	243
4.5.*	Negative Results	246
4.5.1.	On the Importance of Interaction and Randomness	247
4.5.2.	Limitations of Unconditional Results	248
4.5.3.	Limitations of Statistical ZK Proofs	250
4.5.4.	Zero-Knowledge and Parallel Composition	251
4.6.*	Witness Indistinguishability and Hiding	254
4.6.1.	Definitions	254
4.6.2.	Parallel Composition	258
4.6.3.	Constructions	259
4.6.4.	Applications	261
4.7.*	Proofs of Knowledge	262
4.7.1.	Definition	262
4.7.2.	Reducing the Knowledge Error	267
4.7.3.	Zero-Knowledge Proofs of Knowledge for \mathcal{NP}	268
4.7.4.	Applications	269
4.7.5.	Proofs of Identity (Identification Schemes)	270
4.7.6.	Strong Proofs of Knowledge	274
4.8.*	Computationally Sound Proofs (Arguments)	277
4.8.1.	Definition	277
4.8.2.	Perfectly Hiding Commitment Schemes	278
4.8.3.	Perfect Zero-Knowledge Arguments for \mathcal{NP}	284
4.8.4.	Arguments of Poly-Logarithmic Efficiency	286
4.9.*	Constant-Round Zero-Knowledge Proofs	288
4.9.1.	Using Commitment Schemes with Perfect Secrecy	289
4.9.2.	Bounding the Power of Cheating Provers	294
4.10.*	Non-Interactive Zero-Knowledge Proofs	298
4.10.1.	Basic Definitions	299
4.10.2.	Constructions	300
4.10.3.	Extensions	306
4.11.*	Multi-Prover Zero-Knowledge Proofs	311
4.11.1.	Definitions	311
4.11.2.	Two-Sender Commitment Schemes	313
4.11.3.	Perfect Zero-Knowledge for \mathcal{NP}	317
4.11.4.	Applications	319
4.12.	Miscellaneous	320
4.12.1.	Historical Notes	320
4.12.2.	Suggestions for Further Reading	322
4.12.3.	Open Problems	323
4.12.4.	Exercises	323
Appendix A: Background in Computational Number Theory		331
A.1.	Prime Numbers	331
A.1.1.	Quadratic Residues Modulo a Prime	331

CONTENTS

A.1.2.	Extracting Square Roots Modulo a Prime	332
A.1.3.	Primality Testers	332
A.1.4.	On Uniform Selection of Primes	333
A.2.	Composite Numbers	334
A.2.1.	Quadratic Residues Modulo a Composite	335
A.2.2.	Extracting Square Roots Modulo a Composite	335
A.2.3.	The Legendre and Jacobi Symbols	336
A.2.4.	Blum Integers and Their Quadratic-Residue Structure	337
Appendix B: Brief Outline of Volume 2		338
B.1.	Encryption: Brief Summary	338
B.1.1.	Definitions	338
B.1.2.	Constructions	340
B.1.3.	Beyond Eavesdropping Security	343
B.1.4.	Some Suggestions	345
B.2.	Signatures: Brief Summary	345
B.2.1.	Definitions	346
B.2.2.	Constructions	347
B.2.3.	Some Suggestions	349
B.3.	Cryptographic Protocols: Brief Summary	350
B.3.1.	Definitions	350
B.3.2.	Constructions	352
B.3.3.	Some Suggestions	353
Bibliography		355
Index		367

Note: Asterisks throughout Contents indicate advanced material.

List of Figures

0.1	Organization of the work	<i>page</i> xvi
0.2	Rough organization of this volume	xvii
0.3	Plan for one-semester course on the foundations of cryptography	xviii
1.1	Cryptography: two points of view	25
2.1	One-way functions: an illustration	31
2.2	The naive view versus the actual proof of Proposition 2.3.3	49
2.3	The essence of Construction 2.6.3	81
3.1	Pseudorandom generators: an illustration	102
3.2	Construction 3.3.2, as operating on seed $s_0 \in \{0, 1\}^n$	114
3.3	Hybrid H_n^k as a modification of Construction 3.3.2	115
3.4	Construction 3.4.2, as operating on seed $s_0 \in \{0, 1\}^n$	129
3.5	Construction 3.6.5, for $n = 3$	151
3.6	The high-level structure of the DES	166
4.1	Zero-knowledge proofs: an illustration	185
4.2	The advanced sections of this chapter	185
4.3	The dependence structure of this chapter	186
B.1	The Blum-Goldwasser public-key encryption scheme [35]	342

Preface

*It is possible to build a cabin with no foundations,
but not a lasting building.*

Eng. Isidor Goldreich (1906–1995)

Cryptography is concerned with the construction of schemes that should be able to withstand any abuse. Such schemes are constructed so as to maintain a desired functionality, even under malicious attempts aimed at making them deviate from their prescribed functionality.

The design of cryptographic schemes is a very difficult task. One cannot rely on intuitions regarding the typical state of the environment in which a system will operate. For sure, an *adversary* attacking the system will try to manipulate the environment into untypical states. Nor can one be content with countermeasures designed to withstand specific attacks, because the adversary (who will act after the design of the system has been completed) will try to attack the schemes in ways that typically will be different from the ones the designer envisioned. Although the validity of the foregoing assertions seems self-evident, still some people hope that, in practice, ignoring these tautologies will not result in actual damage. Experience shows that such hopes are rarely met; cryptographic schemes based on make-believe are broken, typically sooner rather than later.

In view of the foregoing, we believe that it makes little sense to make assumptions regarding the specific *strategy* that an adversary may use. The only assumptions that can be justified refer to the computational *abilities* of the adversary. Furthermore, it is our opinion that the design of cryptographic systems has to be based on *firm foundations*, whereas ad hoc approaches and heuristics are a very dangerous way to go. A heuristic may make sense when the designer has a very good idea about the environment in which a scheme is to operate, but a cryptographic scheme will have to operate in a maliciously selected environment that typically will transcend the designer's view.

This book is aimed at presenting firm foundations for cryptography. The foundations of cryptography are the paradigms, approaches, and techniques used to conceptualize, define, and provide solutions to natural “security concerns.” We shall present some of these paradigms, approaches, and techniques, as well as some of the fundamental results

obtained by using them. Our emphasis is on the clarification of fundamental concepts and on demonstrating the feasibility of solving several central cryptographic problems.

Solving a cryptographic problem (or addressing a security concern) is a two-stage process consisting of a *definitional stage* and a *constructive stage*. First, in the definitional stage, the functionality underlying the natural concern must be identified and an adequate cryptographic problem must be defined. Trying to list all undesired situations is infeasible and prone to error. Instead, one should define the functionality in terms of operation in an imaginary ideal model and then require a candidate solution to emulate this operation in the real, clearly defined model (which will specify the adversary's abilities). Once the definitional stage is completed, one proceeds to construct a system that will satisfy the definition. Such a construction may use some simpler tools, and its security is to be proved relying on the features of these tools. (In practice, of course, such a scheme also may need to satisfy some specific efficiency requirements.)

This book focuses on several archetypical cryptographic problems (e.g., encryption and signature schemes) and on several central tools (e.g., computational difficulty, pseudorandomness, and zero-knowledge proofs). For each of these problems (resp., tools), we start by presenting the natural concern underlying it (resp., its intuitive objective), then define the problem (resp., tool), and finally demonstrate that the problem can be solved (resp., the tool can be constructed). In the last step, our focus is on demonstrating the feasibility of solving the problem, not on providing a practical solution. As a secondary concern, we typically discuss the level of practicality (or impracticality) of the given (or known) solution.

Computational Difficulty

The specific constructs mentioned earlier (as well as most constructs in this area) can exist only if some sort of computational hardness (i.e., difficulty) exists. Specifically, all these problems and tools require (either explicitly or implicitly) the ability to generate instances of hard problems. Such ability is captured in the definition of one-way functions (see further discussion in Section 2.1). Thus, one-way functions are the very minimum needed for doing most sorts of cryptography. As we shall see, they actually suffice for doing much of cryptography (and the rest can be done by augmentations and extensions of the assumption that one-way functions exist).

Our current state of understanding of efficient computation does not allow us to prove that one-way functions exist. In particular, the existence of one-way functions implies that \mathcal{NP} is not contained in $\mathcal{BPP} \supseteq \mathcal{P}$ (not even “on the average”), which would resolve the most famous open problem of computer science. Thus, we have no choice (at this stage of history) but to assume that one-way functions exist. As justification for this assumption we can only offer the combined beliefs of hundreds (or thousands) of researchers. Furthermore, these beliefs concern a simply stated assumption, and their validity is supported by several widely believed conjectures that are central to some fields (e.g., the conjecture that factoring integers is difficult is central to computational number theory).

As we need assumptions anyhow, why not just assume what we want, that is, the existence of a solution to some natural cryptographic problem? Well, first we need

to know what we want: As stated earlier, we must first clarify what exactly we do want; that is, we must go through the typically complex definitional stage. But once this stage is completed, can we just assume that the definition derived can be met? Not really: The mere fact that a definition has been derived does *not* mean that it can be met, and one can easily define objects that cannot exist (without this fact being obvious in the definition). The way to demonstrate that a definition is viable (and so the intuitive security concern can be satisfied at all) is to construct a solution based on a *better-understood* assumption (i.e., one that is more common and widely believed). For example, looking at the definition of zero-knowledge proofs, it is not a priori clear that such proofs exist at all (in a non-trivial sense). The non-triviality of the notion was first demonstrated by presenting a zero-knowledge proof system for statements regarding Quadratic Residuosity that are believed to be difficult to verify (without extra information). Furthermore, contrary to prior belief, it was later shown that the existence of one-way functions implies that any \mathcal{NP} -statement can be proved in zero-knowledge. Thus, facts that were not at all known to hold (and were even believed to be false) were shown to hold by reduction to widely believed assumptions (without which most of modern cryptography would collapse anyhow). To summarize, not all assumptions are equal, and so reducing a complex, new, and doubtful assumption to a widely believed simple (or even merely simpler) assumption is of great value. Furthermore, reducing the solution of a new task to the assumed security of a well-known primitive task typically means providing a construction that, using the known primitive, will solve the new task. This means that we not only know (or assume) that the new task is solvable but also have a solution based on a primitive that, being well known, typically has several candidate implementations.

Structure and Prerequisites

Our aim is to present the basic concepts, techniques, and results in cryptography. As stated earlier, our emphasis is on the clarification of fundamental concepts and the relationships among them. This is done in a way independent of the particularities of some popular number-theoretic examples. These particular examples played a central role in the development of the field and still offer the most practical implementations of all cryptographic primitives, but this does not mean that the presentation has to be linked to them. On the contrary, we believe that concepts are best clarified when presented at an abstract level, decoupled from specific implementations. Thus, the most relevant background for this book is provided by basic knowledge of algorithms (including randomized ones), computability, and elementary probability theory. Background on (computational) number theory, which is required for specific implementations of certain constructs, is not really required here (yet a short appendix presenting the most relevant facts is included in this volume so as to support the few examples of implementations presented here).

Organization of the work. This work is organized into three parts (see Figure 0.1), to be presented in three volumes: *Basic Tools*, *Basic Applications*, and *Beyond the Basics*. This first volume contains an introductory chapter as well as the first part

Volume 1: Introduction and Basic Tools
Chapter 1: Introduction
Chapter 2: Computational Difficulty (One-Way Functions)
Chapter 3: Pseudorandom Generators
Chapter 4: Zero-Knowledge Proof Systems
Volume 2: Basic Applications
Chapter 5: Encryption Schemes
Chapter 6: Signature Schemes
Chapter 7: General Cryptographic Protocols
Volume 3: Beyond the Basics
...

Figure 0.1: Organization of the work.

(basic tools). It provides chapters on computational difficulty (one-way functions), pseudorandomness, and zero-knowledge proofs. These basic tools will be used for the basic applications in the second volume, which will consist of encryption, signatures, and general cryptographic protocols.

The partition of the work into three volumes is a logical one. Furthermore, it offers the advantage of publishing the first part without waiting for the completion of the other parts. Similarly, we hope to complete the second volume within a couple of years and publish it without waiting for the third volume.

Organization of this first volume. This first volume consists of an introductory chapter (Chapter 1), followed by chapters on computational difficulty (one-way functions), pseudorandomness, and zero-knowledge proofs (Chapters 2–4, respectively). Also included are two appendixes, one of them providing a brief summary of Volume 2. Figure 0.2 depicts the high-level structure of this first volume.

Historical notes, suggestions for further reading, some open problems, and some exercises are provided at the end of each chapter. The exercises are *mostly* designed to assist and test one’s basic understanding of the main text, not to test or inspire creativity. The open problems are fairly well known; still, we recommend that one check their current status (e.g., at our updated-notice web site).

Web site for notices regarding this book. We intend to maintain a web site listing corrections of various types. The location of the site is

<http://www.wisdom.weizmann.ac.il/~oded/foc-book.html>

Using This Book

The book is intended to serve as both a textbook and a reference text. That is, it is aimed at serving both the beginner and the expert. In order to achieve that goal, the presentation of the basic material is very detailed, so as to allow a typical undergraduate in computer science to follow it. An advanced student (and certainly an expert) will find the pace in these parts far too slow. However, an attempt has been made to allow the latter reader to easily skip details that are obvious to him or her. In particular, proofs typically are presented in a modular way. We start with a high-level sketch of the main ideas and

Chapter 1: <i>Introduction</i>
Main topics covered by the book (Sec. 1.1)
Background on probability and computation (Sec. 1.2 and 1.3)
Motivation to the rigorous treatment (Sec. 1.4)
Chapter 2: <i>Computational Difficulty (One-Way Functions)</i>
Motivation and definitions (Sec. 2.1 and 2.2)
One-way functions: weak implies strong (Sec. 2.3)
Variants (Sec. 2.4) and advanced material (Sec. 2.6)
Hard-core predicates (Sec. 2.5)
Chapter 3: <i>Pseudorandom Generators</i>
Motivation and definitions (Sec. 3.1–3.3)
Constructions based on one-way permutations (Sec. 3.4)
Pseudorandom functions (Sec. 3.6)
Advanced material (Sec. 3.5 and 3.7)
Chapter 4: <i>Zero-Knowledge Proofs</i>
Motivation and definitions (Sec. 4.1–4.3)
Zero-knowledge proofs for \mathcal{NP} (Sec. 4.4)
Advanced material (Sec. 4.5–4.11)
Appendix A: Background in Computational Number Theory
Appendix B: Brief Outline of Volume 2
Bibliography and Index

Figure 0.2: Rough organization of this volume.

only later pass to the technical details. The transition from high-level description to lower-level details is typically indicated by phrases such as “details follow.”

In a few places, we provide straightforward but tedious details in indented paragraphs such as this one. In some other (even fewer) places, such paragraphs provide technical proofs of claims that are of marginal relevance to the topic of the book.

More advanced material typically is presented at a faster pace and with fewer details. Thus, we hope that the attempt to satisfy a wide range of readers will not harm any of them.

Teaching. The material presented in this book is, on one hand, way beyond what one may want to cover in a semester course, and on the other hand it falls very short of what one may want to know about cryptography in general. To assist these conflicting needs, we make a distinction between *basic* and *advanced* material and provide suggestions for further reading (in the last section of each chapter). In particular, those sections marked by an asterisk are intended for advanced reading.

Volumes 1 and 2 of this work are intended to provide all the material needed for a course on the foundations of cryptography. For a one-semester course, the instructor definitely will need to skip all advanced material (marked by asterisks) and perhaps even some basic material; see the suggestions in Figure 0.3. This should allow, depending on the class, coverage of the basic material at a reasonable level (i.e., all material marked as “main” and some of the “optional”). Volumes 1 and 2 can also serve as a textbook for a two-semester course. Either way, this first volume covers only the first half of the material for such a course. The second half will be covered in Volume 2. Meanwhile,

Each lecture consists of one hour. Lectures 1–15 are covered by this first volume. Lectures 16–28 will be covered by the second volume.

Lecture 1: Introduction, background, etc.
(depending on class)

Lectures 2–5: *Computational Difficulty (One-Way Functions)*

Main: Definition (Sec. 2.2), Hard-core predicates (Sec. 2.5)

Optional: Weak implies strong (Sec. 2.3), and Sec. 2.4.2–2.4.4

Lectures 6–10: *Pseudorandom Generators*

Main: Definitional issues and a construction (Sec. 3.2–3.4)

Optional: Pseudorandom functions (Sec. 3.6)

Lectures 11–15: *Zero-Knowledge Proofs*

Main: Some definitions and a construction (Sec. 4.2.1, 4.3.1, 4.4.1–4.4.3)

Optional: Sec. 4.2.2, 4.3.2, 4.3.3, 4.3.4, 4.4.4

Lectures 16–20: *Encryption Schemes*

Definitions and a construction (consult Appendix B.1.1–B.1.2)

(See also fragments of a draft for the encryption chapter [99].)

Lectures 21–24: *Signature Schemes*

Definition and a construction (consult Appendix B.2)

(See also fragments of a draft for the signatures chapter [100].)

Lectures 25–28: *General Cryptographic Protocols*

The definitional approach and a general construction (sketches).

(Consult Appendix B.3; see also [98].)

Figure 0.3: Plan for one-semester course on the foundations of cryptography.

we suggest the use of other sources for the second half. A brief summary of Volume 2 and recommendations for alternative sources are given in Appendix B. (In addition, fragments and/or preliminary drafts for the three chapters of Volume 2 are available from earlier texts, [99], [100], and [98], respectively.)

A course based solely on the material in this first volume is indeed possible, but such a course cannot be considered a stand-alone course in cryptography because this volume does not consider at all the basic tasks of encryption and signatures.

Practice. The aim of this work is to provide sound theoretical foundations for cryptography. As argued earlier, such foundations are necessary for any *sound* practice of cryptography. Indeed, sound practice requires more than theoretical foundations, whereas this work makes no attempt to provide anything beyond the latter. However, given sound foundations, one can learn and evaluate various practical suggestions that appear elsewhere (e.g., [158]). On the other hand, the absence of sound foundations will result in inability to critically evaluate practical suggestions, which in turn will lead to unsound decisions. Nothing could be more harmful to the design of schemes that need to withstand adversarial attacks than misconceptions about such attacks.

Relationship to another book by the author. A frequently asked question concerns the relationship of this work to my text *Modern Cryptography, Probabilistic Proofs and Pseudorandomness* [97]. That text consists of three brief introductions to the related topics in the title. Specifically, in Chapter 1 it provides a brief (i.e., 30-page)

summary of this work. The other two chapters of *Modern Cryptography, Probabilistic Proofs and Pseudorandomness* [97] provide a wider perspective on two topics mentioned in this volume (i.e., probabilistic proofs and pseudorandomness). Further comments on the latter aspect are provided in the relevant chapters of this volume.

Acknowledgments

First of all, I would like to thank three remarkable people who had a tremendous influence on my professional development: Shimon Even introduced me to theoretical computer science and closely guided my first steps. Silvio Micali and Shafi Goldwasser led my way in the evolving foundations of cryptography and shared with me their ongoing efforts toward further development of those foundations.

I have collaborated with many researchers, but I feel that my work with Benny Chor and Avi Wigderson has had the most important impact on my professional development and career. I would like to thank them both for their indispensable contributions to our joint research and for the excitement and pleasure of working with them.

Leonid Levin deserves special thanks as well. I have had many interesting discussions with Leonid over the years, and sometimes it has taken me too long to realize how helpful those discussions have been.

Next, I would like to thank a few colleagues and friends with whom I have had significant interactions regarding cryptography and related topics. These include Noga Alon, Boaz Barak, Mihir Bellare, Ran Canetti, Ivan Damgard, Uri Feige, Shai Halevi, Johan Hastad, Amir Herzberg, Russell Impagliazzo, Joe Kilian, Hugo Krawczyk, Eyal Kushilevitz, Yehuda Lindell, Mike Luby, Daniele Micciancio, Moni Naor, Noam Nisan, Andrew Odlyzko, Yair Oren, Rafail Ostrovsky, Erez Petrank, Birgit Pfitzmann, Omer Reingold, Ron Rivest, Amit Sahai, Claus Schnorr, Adi Shamir, Victor Shoup, Madhu Sudan, Luca Trevisan, Salil Vadhan, Ronen Vainish, Yacob Yacobi, and David Zuckerman.

Even assuming I have not overlooked people with whom I have had significant interactions on topics related to this book, the complete list of people to whom I am indebted is far more extensive. It certainly includes the authors of many papers mentioned in the Bibliography. It also includes the authors of many cryptography-related papers that I have not cited and the authors of many papers regarding the theory of computation at large (a theory taken for granted in this book).

Finally, I would like to thank Alon Rosen for carefully reading this manuscript and suggesting numerous corrections.

Introduction

In this chapter we briefly discuss the goals of cryptography (Section 1.1). In particular, we discuss the basic problems of secure encryption, digital signatures, and fault-tolerant protocols. These problems lead to the notions of pseudorandom generators and zero-knowledge proofs, which are discussed as well.

Our approach to cryptography is based on computational complexity. Hence, this introductory chapter also contains a section presenting the computational models used throughout the book (Section 1.3). Likewise, this chapter contains a section presenting some elementary background from probability theory that is used extensively in the book (Section 1.2).

Finally, we motivate the rigorous approach employed throughout this book and discuss some of its aspects (Section 1.4).

Teaching Tip. Parts of Section 1.4 may be more suitable for the last lecture (i.e., as part of the concluding remarks) than for the first one (i.e., as part of the introductory remarks). This refers specifically to Sections 1.4.2 and 1.4.3.

1.1. Cryptography: Main Topics

Historically, the term “cryptography” has been associated with the problem of designing and analyzing *encryption schemes* (i.e., schemes that provide secret communication over insecure communication media). However, since the 1970s, problems such as constructing unforgeable *digital signatures* and designing *fault-tolerant protocols* have also been considered as falling within the domain of cryptography. In fact, cryptography can be viewed as concerned with the design of any system that needs to withstand malicious attempts to abuse it. Furthermore, cryptography as redefined here makes essential use of some tools that need to be treated in a book on the subject. Notable examples include one-way functions, pseudorandom generators, and zero-knowledge proofs. In this section we briefly discuss these terms.

We start by mentioning that much of the content of this book relies on the assumption that one-way functions exist. The definition of one-way functions captures the sort of computational difficulty that is inherent to our entire approach to cryptography, an approach that attempts to capitalize on the computational limitations of any real-life adversary. Thus, if nothing is difficult, then this approach fails. However, if, as is widely believed, not only do hard problems exist but also instances of them can be efficiently generated, then these hard problems can be “put to work.” Thus, “algorithmically bad news” (by which hard computational problems exist) implies good news for cryptography. Chapter 2 is devoted to the definition and manipulation of computational difficulty in the form of one-way functions.

1.1.1. Encryption Schemes

The problem of providing *secret communication over insecure media* is the most traditional and basic problem of cryptography. The setting consists of two parties communicating over a channel that possibly may be tapped by an adversary, called the *wire-tapper*. The parties wish to exchange information with each other, but keep the wire-tapper as ignorant as possible regarding the content of this information. Loosely speaking, an encryption scheme is a protocol allowing these parties to communicate *secretly* with each other. Typically, the encryption scheme consists of a pair of algorithms. One algorithm, called *encryption*, is applied by the sender (i.e., the party sending a message), while the other algorithm, called *decryption*, is applied by the receiver. Hence, in order to send a message, the sender first applies the encryption algorithm to the message and sends the result, called the *ciphertext*, over the channel. Upon receiving a ciphertext, the other party (i.e., the receiver) applies the decryption algorithm to it and retrieves the original message (called the *plaintext*).

In order for this scheme to provide secret communication, the communicating parties (at least the receiver) must know something that is not known to the wire-tapper. (Otherwise, the wire-tapper could decrypt the ciphertext exactly as done by the receiver.) This extra knowledge may take the form of the decryption algorithm itself or some parameters and/or auxiliary inputs used by the decryption algorithm. We call this extra knowledge the *decryption key*. Note that, without loss of generality, we can assume that the decryption algorithm is known to the wire-tapper and that the decryption algorithm needs two inputs: a ciphertext and a decryption key. We stress that the existence of a secret key, not known to the wire-tapper, is merely a necessary condition for secret communication.

Evaluating the “security” of an encryption scheme is a very tricky business. A preliminary task is to understand what “security” is (i.e., to properly define what is meant by this intuitive term). Two approaches to defining security are known. The first (“classic”) approach is *information-theoretic*. It is concerned with the “information” about the plaintext that is “present” in the ciphertext. Loosely speaking, if the ciphertext contains information about the plaintext, then the encryption scheme is considered insecure. It has been shown that such a high (i.e., “perfect”) level of security can be achieved only if the key in use is at least as long as the *total* length of the messages sent via the encryption scheme. The fact that the key has to be longer than the information exchanged using it is indeed a drastic limitation on the applicability of such encryption

schemes. This is especially true when *huge* amounts of information need to be secretly communicated.

The second (“modern”) approach, as followed in this book, is based on *computational complexity*. This approach is based on the fact that *it does not matter whether or not the ciphertext contains information about the plaintext*, but rather *whether or not this information can be efficiently extracted*. In other words, instead of asking whether or not it is *possible* for the wire-tapper to extract specific information, we ask whether or not it is *feasible* for the wire-tapper to extract this information. It turns out that the new (i.e., “computational-complexity”) approach offers security even if the key is much shorter than the total length of the messages sent via the encryption scheme. For example, one can use “pseudorandom generators” (discussed later) that expand short keys into much longer “pseudo-keys,” so that the latter are as secure as “real keys” of comparable length.

In addition, the computational-complexity approach allows the introduction of concepts and primitives that cannot exist under the information-theoretic approach. A typical example is the concept of *public-key encryption schemes*. Note that in the preceding discussion we concentrated on the decryption algorithm and its key. It can be shown that the encryption algorithm must get, in addition to the message, an auxiliary input that depends on the decryption key. This auxiliary input is called the *encryption key*. Traditional encryption schemes, and in particular all the encryption schemes used over the millennia preceding the 1980s, operate with an encryption key equal to the decryption key. Hence, the wire-tapper in these schemes must be ignorant of the encryption key, and consequently the *key-distribution problem* arises (i.e., how two parties wishing to communicate over an insecure channel can agree on a secret encryption/decryption key).¹ The computational-complexity approach allows the introduction of encryption schemes in which the encryption key can be known to the wire-tapper without compromising the security of the scheme. Clearly, the decryption key in such schemes is different from the encryption key, and furthermore it is infeasible to compute the decryption key from the encryption key. Such encryption schemes, called *public-key schemes*, have the advantage of trivially resolving the key-distribution problem, because the encryption key can be publicized.

In Chapter 5, which will appear in the second volume of this work and will be devoted to encryption schemes, we shall discuss private-key and public-key encryption schemes. Much attention is devoted to defining the security of encryption schemes. Finally, constructions of secure encryption schemes based on various intractability assumptions are presented. Some of the constructions presented are based on pseudorandom generators, which are discussed in Chapter 3. Other constructions use specific one-way functions such as the RSA function and/or the operation of squaring modulo a composite number.

1.1.2. Pseudorandom Generators

It turns out that pseudorandom generators play a central role in the construction of encryption schemes (and related schemes). In particular, pseudorandom generators

¹The traditional solution is to exchange the key through an alternative channel that is secure, alas “more expensive to use,” for example, by a convoy.

yield simple constructions of private-key encryption schemes, and this observation is often used in practice (usually implicitly).

Although the term “pseudorandom generators” is commonly used *in practice*, both in the context of cryptography and in the much wider context of probabilistic procedures, it is seldom associated with a precise meaning. We believe that using a term without clearly stating what it means is dangerous in general and particularly so in a tricky business such as cryptography. Hence, a precise treatment of pseudorandom generators is central to cryptography.

Loosely speaking, a pseudorandom generator is a deterministic algorithm that expands short random seeds into much longer bit sequences that *appear* to be “random” (although they are not). In other words, although the output of a pseudorandom generator is not really random, it is *infeasible* to tell the difference. It turns out that pseudorandomness and computational difficulty are linked in an even more fundamental manner, as pseudorandom generators can be constructed based on various intractability assumptions. Furthermore, the main result in this area asserts that pseudorandom generators exist if and only if one-way functions exist.

Chapter 3, devoted to pseudorandom generators, starts with a treatment of the concept of computational indistinguishability. Pseudorandom generators are defined next and are constructed using special types of one-way functions (defined in Chapter 2). Pseudorandom *functions* are defined and constructed as well. The latter offer a host of additional applications.

1.1.3. Digital Signatures

A notion that did not exist in the pre-computerized world is that of a “digital signature.” The need to discuss digital signatures arose with the introduction of computer communication in the business environment in which parties need to commit themselves to proposals and/or declarations they make. Discussions of “unforgeable signatures” also took place in previous centuries, but the objects of discussion were handwritten signatures, not digital ones, and the discussion was not perceived as related to cryptography.

Relations between encryption and signature methods became possible with the “digitalization” of both and the introduction of the computational-complexity approach to security. Loosely speaking, a *scheme for unforgeable signatures* requires

- that each user be able *to efficiently generate his or her own signature* on documents of his or her choice,
- that each user be able *to efficiently verify* whether or not a given string is a signature of another (specific) user on a specific document, and
- that *no one be able to efficiently produce the signatures of other users* to documents that those users did not sign.

We stress that the formulation of unforgeable digital signatures also provides a clear statement of the essential ingredients of handwritten signatures. Indeed, the ingredients are each person’s ability to sign for himself or herself, a universally agreed verification procedure, and the belief (or assertion) that it is infeasible (or at least

difficult) to forge signatures in a manner that could pass the verification procedure. It is difficult to state to what extent handwritten signatures meet these requirements. In contrast, our discussion of digital signatures will supply precise statements concerning the extent to which digital signatures meet the foregoing requirements. Furthermore, schemes for unforgeable digital signatures can be constructed using the same computational assumptions as used in the construction of (private-key) encryption schemes.

In Chapter 6, which will appear in the second volume of this work and will be devoted to signature schemes, much attention will be focused on defining the security (i.e., unforgeability) of these schemes. Next, constructions of unforgeable signature schemes based on various intractability assumptions will be presented. In addition, we shall treat the related problem of message authentication.

Message Authentication

Message authentication is a task related to the setting considered for encryption schemes (i.e., communication over an insecure channel). This time, we consider the case of an active adversary who is monitoring the channel and may alter the messages sent on it. The parties communicating through this insecure channel wish to authenticate the messages they send so that the intended recipient can tell an original message (sent by the sender) from a modified one (i.e., modified by the adversary). Loosely speaking, a *scheme for message authentication* requires

- that each of the communicating parties be able *to efficiently generate an authentication tag* for any message of his or her choice,
- that each of the communicating parties be able *to efficiently verify* whether or not a given string is an authentication tag for a given message, and
- that *no external adversary* (i.e., a party other than the communicating parties) *be able to efficiently produce authentication tags* to messages not sent by the communicating parties.

In some sense, “message authentication” is similar to a digital signature. The difference between the two is that in the setting of message authentication it is not required that third parties (who may be dishonest) be able to verify the validity of authentication tags produced by the designated users, whereas in the setting of signature schemes it is required that such third parties be able to verify the validity of signatures produced by other users. Hence, digital signatures provide a solution to the message-authentication problem. On the other hand, a message-authentication scheme does not necessarily constitute a digital-signature scheme.

Signatures Widen the Scope of Cryptography

Considering the problem of digital signatures as belonging to cryptography widens the scope of this area from the specific secret-communication problem to a variety of problems concerned with limiting the “gain” that can be achieved by “dishonest” behavior of parties (who are either internal or external to the system). Specifically:

- In the secret-communication problem (solved by use of encryption schemes), one wishes to reduce, as much as possible, the information that a potential wire-tapper can extract from the communication between two designated users. In this case, the designated system consists of the two communicating parties, and the wire-tapper is considered as an external (“dishonest”) party.
- In the message-authentication problem, one aims at prohibiting any (external) wire-tapper from modifying the communication between two (designated) users.
- In the signature problem, one aims at providing all users of a system a way of making self-binding statements and of ensuring that one user cannot make statements that would bind another user. In this case, the designated system consists of the set of all users, and a potential forger is considered as an internal yet dishonest user.

Hence, in the wide sense, *cryptography is concerned with any problem in which one wishes to limit the effects of dishonest users*. A general treatment of such problems is captured by the treatment of “fault-tolerant” (or cryptographic) protocols.

1.1.4. Fault-Tolerant Protocols and Zero-Knowledge Proofs

A discussion of signature schemes naturally leads to a discussion of cryptographic protocols, because it is a natural concern to ask under what circumstances one party should provide its signature to another party. In particular, problems like mutual simultaneous commitment (e.g., contract signing) arise naturally. Another type of problem, motivated by the use of computer communication in the business environment, consists of “secure implementation” of protocols (e.g., implementing secret and incorruptible voting).

Simultaneity Problems

A typical example of a simultaneity problem is that of simultaneous exchange of secrets, of which contract signing is a special case. The setting for a simultaneous exchange of secrets consists of two parties, each holding a “secret.” The goal is to execute a protocol such that if both parties follow it correctly, then at termination each will hold its counterpart’s secret, and in any case (even if one party cheats) the first party will hold the second party’s secret if and only if the second party holds the first party’s secret. Perfectly simultaneous exchange of secrets can be achieved only if we assume the existence of third parties that are trusted to some extent. In fact, simultaneous exchange of secrets can easily be achieved using the active participation of a trusted third party: Each party sends its secret to the trusted third party (using a secure channel). The third party, on receiving both secrets, sends the first party’s secret to the second party and the second party’s secret to the first party. There are two problems with this solution:

1. The solution requires the *active* participation of an “external” party in all cases (i.e., also in case both parties are honest). We note that other solutions requiring milder forms of participation of external parties do exist.
2. The solution requires the existence of a *totally trusted* third entity. In some applications, such an entity does not exist. Nevertheless, in the sequel we shall discuss the problem

of implementing a trusted third party by a set of users with an honest majority (even if the identity of the honest users is not known).

Secure Implementation of Functionalities and Trusted Parties

A different type of protocol problem is concerned with the secure implementation of functionalities. To be more specific, we discuss the problem of evaluating a function of local inputs each of which is held by a different user. An illustrative and motivating example is *voting*, in which the function is majority, and the local input held by user A is a single bit representing the vote of user A (e.g., “pro” or “con”). Loosely speaking, a protocol for securely evaluating a specific function must satisfy the following:

- *Privacy*: No party can “gain information” on the input of other parties, beyond what is deduced from the value of the function.
- *Robustness*: No party can “influence” the value of the function, beyond the influence exerted by selecting its own input.

It is sometimes required that these conditions hold with respect to “small” (e.g., minority) coalitions of parties (instead of single parties).

Clearly, if one of the users is known to be totally trustworthy, then there exists a simple solution to the problem of secure evaluation of any function. Each user simply sends its input to the trusted party (using a secure channel), who, upon receiving all inputs, computes the function, sends the outcome to all users, and erases all intermediate computations (including the inputs received) from its memory. Certainly, it is unrealistic to assume that a party can be trusted to such an extent (e.g., that it will voluntarily erase what it has “learned”). Nevertheless, the problem of implementing secure function evaluation reduces to the problem of implementing a trusted party. It turns out that a trusted party can be implemented by a set of users with an honest majority (even if the identity of the honest users is not known). This is indeed a major result in this field, and much of Chapter 7, which will appear in the second volume of this work, will be devoted to formulating and proving it (as well as variants of it).

Zero-Knowledge as a Paradigm

A major tool in the construction of cryptographic protocols is the concept of *zero-knowledge* proof systems and the fact that zero-knowledge proof systems exist for all languages in \mathcal{NP} (provided that one-way functions exist). Loosely speaking, a zero-knowledge proof yields nothing but the validity of the assertion. Zero-knowledge proofs provide a tool for “forcing” parties to follow a given protocol properly.

To illustrate the role of zero-knowledge proofs, consider a setting in which a party, called Alice, upon receiving an encrypted message from Bob, is to send Carol the least significant bit of the message. Clearly, if Alice sends only the (least significant) bit (of the message), then there is no way for Carol to know Alice did not cheat. Alice could prove that she did not cheat by revealing to Carol the entire message as well as its decryption key, but that would yield information far beyond what had been

required. A much better idea is to let Alice augment the bit she sends Carol with a zero-knowledge proof that this bit is indeed the least significant bit of the message. We stress that the foregoing statement is of the “ \mathcal{NP} type” (since the proof specified earlier can be efficiently verified), and therefore the existence of zero-knowledge proofs for \mathcal{NP} -statements implies that the foregoing statement can be proved without revealing anything beyond its validity.

The focus of Chapter 4, devoted to zero-knowledge proofs, is on the foregoing result (i.e., the construction of zero-knowledge proofs for any \mathcal{NP} -statement). In addition, we shall consider numerous variants and aspects of the notion of zero-knowledge proofs and their effects on the applicability of this notion.

1.2. Some Background from Probability Theory

Probability plays a central role in cryptography. In particular, probability is essential in order to allow a discussion of information or lack of information (i.e., secrecy). We assume that the reader is familiar with the basic notions of probability theory. In this section, we merely present the probabilistic notations that are used throughout this book and three useful probabilistic inequalities.

1.2.1. Notational Conventions

Throughout this entire book we shall refer to only *discrete* probability distributions. Typically, the probability space consists of the set of all strings of a certain length ℓ , taken with uniform probability distribution. That is, the sample space is the set of all ℓ -bit-long strings, and each such string is assigned probability measure $2^{-\ell}$. Traditionally, functions from the sample space to the reals are called *random variables*. Abusing standard terminology, we allow ourselves to use the term *random variable* also when referring to functions mapping the sample space into the set of binary strings. We often do not specify the probability space, but rather talk directly about random variables. For example, we may say that X is a random variable assigned values in the set of all strings, so that $\Pr[X = 00] = \frac{1}{4}$ and $\Pr[X = 111] = \frac{3}{4}$. (Such a random variable can be defined over the sample space $\{0, 1\}^2$, so that $X(11) = 00$ and $X(00) = X(01) = X(10) = 111$.) In most cases the probability space consists of all strings of a particular length. Typically, these strings represent random choices made by some randomized process (see next section), and the random variable is the output of the process.

How to Read Probabilistic Statements. All our probabilistic statements refer to functions of random variables that are defined beforehand. Typically, we shall write $\Pr[f(X) = 1]$, where X is a random variable defined beforehand (and f is a function). An important convention is that *all occurrences of a given symbol in a probabilistic statement refer to the same (unique) random variable*. Hence, if $B(\cdot, \cdot)$ is a Boolean expression depending on two variables and X is a random variable, then $\Pr[B(X, X)]$ denotes the probability that $B(x, x)$ holds when x is chosen with probability $\Pr[X = x]$.

Namely,

$$\Pr[B(X, X)] = \sum_x \Pr[X = x] \cdot \chi(B(x, x))$$

where χ is an indicator function, so that $\chi(B) = 1$ if event B holds, and equals zero otherwise. For example, for every random variable X , we have $\Pr[X = X] = 1$. We stress that if one wishes to discuss the probability that $B(x, y)$ holds when x and y are chosen independently with the same probability distribution, then one needs to define *two* independent random variables, both with the same probability distribution. Hence, if X and Y are two independent random variables, then $\Pr[B(X, Y)]$ denotes the probability that $B(x, y)$ holds when the pair (x, y) is chosen with probability $\Pr[X = x] \cdot \Pr[Y = y]$. Namely,

$$\Pr[B(X, Y)] = \sum_{x, y} \Pr[X = x] \cdot \Pr[Y = y] \cdot \chi(B(x, y))$$

For example, for every two independent random variables, X and Y , we have $\Pr[X = Y] = 1$ only if both X and Y are trivial (i.e., assign the entire probability mass to a single string).

Typical Random Variables. Throughout this entire book, U_n denotes a random variable uniformly distributed over the set of strings of length n . Namely, $\Pr[U_n = \alpha]$ equals 2^{-n} if $\alpha \in \{0, 1\}^n$, and equals zero otherwise. In addition, we shall occasionally use random variables (arbitrarily) distributed over $\{0, 1\}^n$ or $\{0, 1\}^{l(n)}$ for some function $l: \mathbb{N} \rightarrow \mathbb{N}$. Such random variables are typically denoted by X_n, Y_n, Z_n , etc. We stress that in some cases X_n is distributed over $\{0, 1\}^n$, whereas in others it is distributed over $\{0, 1\}^{l(n)}$, for some function $l(\cdot)$, which is typically a polynomial. Another type of random variable, the output of a randomized algorithm on a fixed input, is discussed in Section 1.3.

1.2.2. Three Inequalities

The following probabilistic inequalities will be very useful in the course of this book. All inequalities refer to random variables that are assigned real values. The most basic inequality is the *Markov inequality*, which asserts that for random variables with bounded maximum or minimum values, some relation must exist between the deviation of a value from the expectation of the random variable and the probability that the random variable is assigned this value. Specifically, letting $E(X) \stackrel{\text{def}}{=} \sum_v \Pr[X = v] \cdot v$ denote the expectation of the random variable X , we have the following:

Markov Inequality: *Let X be a non-negative random variable and v a real number. Then*

$$\Pr[X \geq v] \leq \frac{E(X)}{v}$$

Equivalently, $\Pr[X \geq r \cdot E(X)] \leq \frac{1}{r}$.

Proof:

$$\begin{aligned} E(X) &= \sum_x \Pr[X = x] \cdot x \\ &\geq \sum_{x < v} \Pr[X = x] \cdot 0 + \sum_{x \geq v} \Pr[X = x] \cdot v \\ &= \Pr[X \geq v] \cdot v \end{aligned}$$

The claim follows. ■

The Markov inequality is typically used in cases in which one knows very little about the distribution of the random variable; it suffices to know its expectation and at least one bound on the range of its values. See Exercise 1.

Using Markov's inequality, one gets a “possibly stronger” bound for the deviation of a random variable from its expectation. This bound, called Chebyshev's inequality, is useful provided one has additional knowledge concerning the random variable (specifically, a good upper bound on its variance). For a random variable X of finite expectation, we denote by $\text{Var}(X) \stackrel{\text{def}}{=} E[(X - E(X))^2]$ the variance of X and observe that $\text{Var}(X) = E(X^2) - E(X)^2$.

Chebyshev's Inequality: *Let X be a random variable, and $\delta > 0$. Then*

$$\Pr[|X - E(X)| \geq \delta] \leq \frac{\text{Var}(X)}{\delta^2}$$

Proof: We define a random variable $Y \stackrel{\text{def}}{=} (X - E(X))^2$ and apply the Markov inequality. We get

$$\begin{aligned} \Pr[|X - E(X)| \geq \delta] &= \Pr[(X - E(X))^2 \geq \delta^2] \\ &\leq \frac{E[(X - E(X))^2]}{\delta^2} \end{aligned}$$

and the claim follows. ■

Chebyshev's inequality is particularly useful for analysis of the error probability of approximation via repeated sampling. It suffices to assume that the samples are picked in a pairwise-independent manner.

Corollary (Pairwise-Independent Sampling): *Let X_1, X_2, \dots, X_n be pairwise-independent random variables with the same expectation, denoted μ , and the same variance, denoted σ^2 . Then, for every $\varepsilon > 0$,*

$$\Pr \left[\left| \frac{\sum_{i=1}^n X_i}{n} - \mu \right| \geq \varepsilon \right] \leq \frac{\sigma^2}{\varepsilon^2 n}$$

The X_i 's are called *pairwise-independent* if for every $i \neq j$ and all a and b , it holds that $\Pr[X_i = a \wedge X_j = b]$ equals $\Pr[X_i = a] \cdot \Pr[X_j = b]$.

Proof: Define the random variables $\bar{X}_i \stackrel{\text{def}}{=} X_i - \mathbb{E}(X_i)$. Note that the \bar{X}_i 's are pairwise-independent and each has zero expectation. Applying Chebyshev's inequality to the random variable defined by the sum $\sum_{i=1}^n \frac{X_i}{n}$, and using the linearity of the expectation operator, we get

$$\begin{aligned} \Pr \left[\left| \sum_{i=1}^n \frac{X_i}{n} - \mu \right| \geq \varepsilon \right] &\leq \frac{\text{Var} \left[\sum_{i=1}^n \frac{X_i}{n} \right]}{\varepsilon^2} \\ &= \frac{\mathbb{E} \left[\left(\sum_{i=1}^n \bar{X}_i \right)^2 \right]}{\varepsilon^2 \cdot n^2} \end{aligned}$$

Now (again using the linearity of \mathbb{E})

$$\mathbb{E} \left[\left(\sum_{i=1}^n \bar{X}_i \right)^2 \right] = \sum_{i=1}^n \mathbb{E}[\bar{X}_i^2] + \sum_{1 \leq i \neq j \leq n} \mathbb{E}[\bar{X}_i \bar{X}_j]$$

By the pairwise independence of the \bar{X}_i 's, we get $\mathbb{E}[\bar{X}_i \bar{X}_j] = \mathbb{E}[\bar{X}_i] \cdot \mathbb{E}[\bar{X}_j]$, and using $\mathbb{E}[\bar{X}_i] = 0$, we get

$$\mathbb{E} \left[\left(\sum_{i=1}^n \bar{X}_i \right)^2 \right] = n \cdot \sigma^2$$

The corollary follows. ■

Using pairwise-independent sampling, the error probability in the approximation is decreasing linearly with the number of sample points. Using totally independent sampling points, the error probability in the approximation can be shown to decrease exponentially with the number of sample points. (The random variables X_1, X_2, \dots, X_n are said to be *totally independent* if for every sequence a_1, a_2, \dots, a_n it holds that $\Pr[\wedge_{i=1}^n X_i = a_i]$ equals $\prod_{i=1}^n \Pr[X_i = a_i]$.) Probability bounds supporting the foregoing statement are given next. The first bound, commonly referred to as the *Chernoff bound*, concerns 0-1 random variables (i.e., random variables that are assigned values of either 0 or 1).

Chernoff Bound: Let $p \leq \frac{1}{2}$, and let X_1, X_2, \dots, X_n be independent 0-1 random variables, so that $\Pr[X_i = 1] = p$ for each i . Then for all ε , $0 < \varepsilon \leq p(1 - p)$, we have

$$\Pr \left[\left| \frac{\sum_{i=1}^n X_i}{n} - p \right| > \varepsilon \right] < 2 \cdot e^{-\frac{\varepsilon^2}{2p(1-p)} \cdot n}$$

We shall usually apply the bound with a constant $p \approx \frac{1}{2}$. In this case, n independent samples give an approximation that deviates by ε from the expectation with probability δ that is exponentially decreasing with $\varepsilon^2 n$. Such an approximation is called an (ε, δ) -approximation and can be achieved using $n = O(\varepsilon^{-2} \cdot \log(1/\delta))$ sample points. It is important to remember that the sufficient number of sample points is polynomially related to ε^{-1} and logarithmically related to δ^{-1} . So using $\text{poly}(n)$ many samples, the

error probability (i.e., δ) can be made negligible (as a function in n), but the accuracy of the estimation (i.e., ε) can be bounded above only by any fixed polynomial fraction (but cannot be made negligible).² We stress that the dependence of the number of samples on ε is not better than in the case of pairwise-independent sampling; the advantage of totally independent samples lies only in the dependence of the number of samples on δ .

A more general bound, useful for approximation of the expectation of a general random variable (not necessarily 0-1), is given as follows:

Hoeffding Inequality:³ *Let X_1, X_2, \dots, X_n be n independent random variables with the same probability distribution, each ranging over the (real) interval $[a, b]$, and let μ denote the expected value of each of these variables. Then, for every $\varepsilon > 0$,*

$$\Pr \left[\left| \frac{\sum_{i=1}^n X_i}{n} - \mu \right| > \varepsilon \right] < 2 \cdot e^{-\frac{2\varepsilon^2}{(b-a)^2} \cdot n}$$

The Hoeffding inequality is useful for estimating the average value of a function defined over a large set of values, especially when the desired error probability needs to be negligible. It can be applied provided we can efficiently sample the set and have a bound on the possible values (of the function). See Exercise 2.

1.3. The Computational Model

Our approach to cryptography is heavily based on computational complexity. Thus, some background on computational complexity is required for our discussion of cryptography. In this section, we briefly recall the definitions of the complexity classes \mathcal{P} , \mathcal{NP} , \mathcal{BPP} , and “non-uniform \mathcal{P} ” (i.e., \mathcal{P}/poly) and the concept of oracle machines. In addition, we discuss the types of intractability assumptions used throughout the rest of this book.

1.3.1. \mathcal{P} , \mathcal{NP} , and \mathcal{NP} -Completeness

A conservative approach to computing devices associates efficient computations with the complexity class \mathcal{P} . Jumping ahead, we note that the approach taken in this book is a more liberal one in that it allows the computing devices to be randomized.

Definition 1.3.1 (Complexity Class \mathcal{P}): *A language L is **recognizable in** (deterministic) **polynomial time** if there exists a deterministic Turing machine M and a polynomial $p(\cdot)$ such that*

- *on input a string x , machine M halts after at most $p(|x|)$ steps, and*
- *$M(x) = 1$ if and only if $x \in L$.*

²Here and in the rest of this book, we denote by $\text{poly}()$ some fixed but unspecified polynomial.

³A more general form requires the X_i 's to be independent, but not necessarily identical, and uses $\mu \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n \mathbb{E}(X_i)$. See [6, app. A].

\mathcal{P} is the class of languages that can be recognized in (deterministic) polynomial time.

Likewise, the complexity class \mathcal{NP} is associated with computational problems having solutions that, once given, can be efficiently tested for validity. It is customary to define \mathcal{NP} as the class of languages that can be recognized by a non-deterministic polynomial-time Turing machine. A more fundamental formulation of \mathcal{NP} is given by the following equivalent definition.

Definition 1.3.2 (Complexity Class \mathcal{NP}): A language L is in \mathcal{NP} if there exists a Boolean relation $R_L \subseteq \{0, 1\}^* \times \{0, 1\}^*$ and a polynomial $p(\cdot)$ such that R_L can be recognized in (deterministic) polynomial time, and $x \in L$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in R_L$. Such a y is called a **witness for membership** of $x \in L$.

Thus, \mathcal{NP} consists of the set of languages for which there exist short proofs of membership that can be efficiently verified. It is widely believed that $\mathcal{P} \neq \mathcal{NP}$, and resolution of this issue is certainly the most intriguing open problem in computer science. If indeed $\mathcal{P} \neq \mathcal{NP}$, then there exists a language $L \in \mathcal{NP}$ such that every algorithm recognizing L will have a super-polynomial running time *in the worst case*. Certainly, all \mathcal{NP} -complete languages (defined next) will have super-polynomial-time complexity *in the worst case*.

Definition 1.3.3 (\mathcal{NP} -Completeness): A language is **\mathcal{NP} -complete** if it is in \mathcal{NP} and every language in \mathcal{NP} is polynomially reducible to it. A language L is **polynomially reducible** to a language L' if there exists a polynomial-time-computable function f such that $x \in L$ if and only if $f(x) \in L'$.

Among the languages known to be \mathcal{NP} -complete are *Satisfiability* (of propositional formulae), *Graph Colorability*, and *Graph Hamiltonicity*.

1.3.2. Probabilistic Polynomial Time

Randomized algorithms play a central role in cryptography. They are needed in order to allow the legitimate parties to generate secrets and are therefore allowed also to the adversaries. The reader is assumed to be familiar and comfortable with such algorithms.

1.3.2.1. Randomized Algorithms: An Example

To demonstrate the notion of a randomized algorithm, we present a simple randomized algorithm for deciding whether or not a given (undirected) graph is connected (i.e., there is a path between each pair of vertices in the graph). We comment that the following algorithm is interesting because it uses significantly less space than the standard (BFS or DFS-based) deterministic algorithms.

Testing whether or not a graph is connected is easily reduced to testing connectivity between any given pair of vertices.⁴ Thus, we focus on the task of determining whether or not two given vertices are connected in a given graph.

Algorithm. On input a graph $G = (V, E)$ and two vertices, s and t , we take a *random walk* of length $O(|V| \cdot |E|)$, starting at vertex s , and test at each step whether or not vertex t is encountered. If vertex t is ever encountered, then the algorithm will accept; otherwise, it will reject. By a random walk we mean that at each step we uniformly select one of the edges incident at the current vertex and traverse this edge to the other endpoint.

Analysis. Clearly, if s is not connected to t in the graph G , then the probability that the foregoing algorithm will accept will be zero. The harder part of the analysis is to prove that if s is connected to t in the graph G , then the algorithm will accept with probability at least $\frac{2}{3}$. (The proof is deferred to Exercise 3.) Thus, either way, the algorithm will err with probability at most $\frac{1}{3}$. The error probability can be further reduced by invoking the algorithm several times (using fresh random choices in each try).

随机算法两种观点

1.3.2.2. Randomized Algorithms: Two Points of View

Randomized algorithms (machines) can be viewed in two equivalent ways. One way of viewing **randomized algorithms** is to allow the algorithm to make random moves (i.e., “toss coins”). Formally, this can be modeled by a Turing machine in which the transition function maps pairs of the form $(\langle \text{state} \rangle, \langle \text{symbol} \rangle)$ to two possible triples of the form $(\langle \text{state} \rangle, \langle \text{symbol} \rangle, \langle \text{direction} \rangle)$. The next step for such a machine is determined by a random choice of one of these triples. Namely, to make a step, the machine chooses at random (with probability $\frac{1}{2}$ for each possibility) either the first triple or the second one and then acts accordingly. These random choices are called the *internal coin tosses* of the machine. The output of a probabilistic machine M on input x is not a string but rather a random variable that assumes strings as possible values. This random variable, denoted $M(x)$, is induced by the internal coin tosses of M . By $\Pr[M(x) = y]$ we mean the probability that machine M on input x will output y . The probability space is that of all possible outcomes for the internal coin tosses taken with uniform probability distribution.⁵ Because we consider only polynomial-time machines, we can assume, without loss of generality, that the number of coin tosses made by M on input x is independent of their outcome and is denoted by $t_M(x)$. We denote by $M_r(x)$ the output of M on input x when r is the outcome of its internal coin tosses. Then $\Pr[M(x) = y]$

⁴The space complexity of such a reduction is low; we merely need to store the names of two vertices (currently being tested). Alas, the time complexity is indeed relatively high; we need to invoke the two-vertex tester $\binom{n}{2}$ times, where n is the number of vertices in the graph.

⁵This sentence is slightly more problematic than it seems. The simple case is when, on input x , machine M always makes the same number of internal coin tosses (independent of their outcome). In general, the number of coin tosses may depend on the outcome of prior coin tosses. Still, for every r , the probability that the outcome of the sequence of internal coin tosses will be r equals $2^{-|r|}$ if the machine does not terminate when the sequence of outcomes is a strict prefix of r , and equals zero otherwise. Fortunately, because we consider polynomial-time machines, we can modify all machines so that they will satisfy the structure of the simple case (and thus avoid the foregoing complication).

什么概率?

is merely the fraction of $r \in \{0, 1\}^{t_M(x)}$ for which $M_r(x) = y$. Namely,

$$\Pr[M(x) = y] = \frac{|\{r \in \{0, 1\}^{t_M(x)} : M_r(x) = y\}|}{2^{t_M(x)}}$$

² The second way of looking at randomized algorithms is to view the outcome of the internal coin tosses of the machine as an auxiliary input. Namely, we consider deterministic machines with two inputs. The first input plays the role of the “real input” (i.e., x) of the first approach, while the second input plays the role of a possible outcome for a sequence of internal coin tosses. Thus, the notation $M(x, r)$ corresponds to the notation $M_r(x)$ used earlier. In the second approach, consider the probability distribution of $M(x, r)$ for any fixed x and a uniformly chosen $r \in \{0, 1\}^{t_M(x)}$. Pictorially, here the coin tosses are not “internal” but rather are supplied to the machine by an “external” coin-tossing device.

Before continuing, let it be noted that we should not confuse the fictitious model of “non-deterministic” machines with the model of probabilistic machines. The former is an unrealistic model that is useful for talking about search problems whose solutions can be efficiently verified (e.g., the definition of \mathcal{NP}), whereas the latter is a realistic model of computation.

Throughout this entire book, unless otherwise stated, a *probabilistic polynomial-time Turing machine* means a probabilistic machine that always (i.e., independently of the outcome of its internal coin tosses) halts after a polynomial (in the length of the input) number of steps. It follows that the number of coin tosses for a probabilistic polynomial-time machine M is bounded by a polynomial, denoted T_M , in its input length. Finally, without loss of generality, we assume that on input x the machine always makes $T_M(|x|)$ coin tosses.

将“有效”计算与BPP关联起来

1.3.2.3. Associating “Efficient” Computations with \mathcal{BPP}

The basic thesis underlying our discussion is the association of “efficient” computations with probabilistic polynomial-time computations. That is, we shall consider as efficient only randomized algorithms (i.e., probabilistic Turing machines) for which the running time is bounded by a polynomial in the length of the input.

Thesis: *Efficient computations correspond to computations that can be carried out by probabilistic polynomial-time Turing machines.*

A complexity class capturing these computations is the class denoted \mathcal{BPP} , of languages recognizable (with high probability) by probabilistic polynomial-time Turing machines. The probability refers to the event in which *the machine makes the correct verdict on string x* .

Definition 1.3.4 (Bounded-Probability Polynomial Time, \mathcal{BPP}): *We say that L is recognized by the probabilistic polynomial-time Turing machine M if*

- for every $x \in L$ it holds that $\Pr[M(x) = 1] \geq \frac{2}{3}$, and
- for every $x \notin L$ it holds that $\Pr[M(x) = 0] \geq \frac{2}{3}$.

BPP是可以被概率多项式时间图灵机识别的语言类

BPP is the class of languages that can be recognized by a probabilistic polynomial-time Turing machine (i.e., randomized algorithm).

The phrase “bounded-probability” indicates that the success probability is bounded away from $\frac{1}{2}$. In fact, in Definition 1.3.4, replacing the constant $\frac{2}{3}$ by any other constant greater than $\frac{1}{2}$ will not change the class defined; see Exercise 4. Likewise, the constant $\frac{2}{3}$ can be replaced by $1 - 2^{-|x|}$ and the class will remain invariant; see Exercise 5. We conclude that languages in BPP can be recognized by probabilistic polynomial-time algorithms with a negligible error probability. We use *negligible* to describe any function that decreases faster than the reciprocal of any polynomial:

错误概率可忽略不计

Negligible Functions

Definition 1.3.5 (Negligible): We call a function $\mu : \mathbb{N} \rightarrow \mathbb{R}$ **negligible** if for every positive polynomial $p(\cdot)$ there exists an N such that for all $n > N$,

$$\mu(n) < \frac{1}{p(n)}$$

For example, the functions $2^{-\sqrt{n}}$ and $n^{-\log_2 n}$ are negligible (as functions in n). Negligible functions stay that way when multiplied by any fixed polynomial. Namely, for every negligible function μ and any polynomial p , the function $\mu'(n) \stackrel{\text{def}}{=} p(n) \cdot \mu(n)$ is negligible. It follows that an event that occurs with negligible probability would be highly unlikely to occur even if we repeated the experiment polynomially many times.

Convention. In Definition 1.3.5 we used the phrase “there exists an N such that for all $n > N$.” In the future we shall use the shorter and less tedious phrase “for all sufficiently large n .” This makes one quantifier (i.e., the $\exists N$) implicit, and that is particularly beneficial in statements that contain several (more essential) quantifiers.

非均匀多项式时间算法

1.3.3. Non-Uniform Polynomial Time

A stronger (and actually unrealistic) model of efficient computation is that of non-uniform polynomial time. This model will be used only in the negative way, namely, for saying that even such machines cannot do something (specifically, even if the adversary employs such a machine, it cannot cause harm).

二元组

A *non-uniform polynomial-time “machine”* is a pair (M, \bar{a}) , where M is a two-input polynomial-time Turing machine and $\bar{a} = a_1, a_2, \dots$ is an infinite sequence of strings such that $|a_n| = \text{poly}(n)$.⁶ For every x , we consider the computation of machine M on the input pair $(x, a_{|x|})$. Intuitively, a_n can be thought of as extra “advice” supplied from the “outside” (together with the input $x \in \{0, 1\}^n$). We stress that machine M


⁶Recall that $\text{poly}()$ stands for some (unspecified) fixed polynomial; that is, we say that there exists some polynomial p such that $|a_n| = p(n)$ for all $n \in \mathbb{N}$.

gets the same advice (i.e., a_n) on all inputs of the same length (i.e., n). Intuitively, the advice a_n may be useful in some cases (i.e., for some computations on inputs of length n), but it is unlikely to encode enough information to be useful for all 2^n possible inputs.

Another way of looking at non-uniform polynomial-time “machines” is to consider an infinite sequence of Turing machines, M_1, M_2, \dots , such that both the length of the description of M_n and its running time on inputs of length n are bounded by polynomials in n (fixed for the entire sequence). Machine M_n is used only on inputs of length n . Note the correspondence between the two ways of looking at non-uniform polynomial time. The pair $(M, (a_1, a_2, \dots))$ (of the first definition) gives rise to an infinite sequence of machines M_{a_1}, M_{a_2}, \dots , where $M_{a_{|x|}}(x) \stackrel{\text{def}}{=} M(x, a_{|x|})$. On the other hand, a sequence M_1, M_2, \dots (as in the second definition) gives rise to a pair $(U, (\langle M_1 \rangle, \langle M_2 \rangle, \dots))$, where U is the universal Turing machine and $\langle M_n \rangle$ is the description of machine M_n (i.e., $U(x, \langle M_{|x|} \rangle) = M_{|x|}(x)$).

In the first sentence of this Section 1.3.3, non-uniform polynomial time was referred to as a stronger model than probabilistic polynomial time. That statement is valid in many contexts (e.g., language recognition, as seen later in Theorem 1.3.7). In particular, it will be valid in all contexts we discuss in this book. So we have the following informal “meta-theorem”:

Meta-theorem: *Whatever can be achieved by probabilistic polynomial-time machines can be achieved by non-uniform polynomial-time “machines.”*

The meta-theorem clearly is wrong if we think of the task of tossing  coins. So the meta-theorem should not be understood literally. It is merely an indication of real theorems that can be proved in reasonable cases. Let us consider, for example, the context of language recognition.

Definition 1.3.6: *The complexity class non-uniform polynomial time (denoted \mathcal{P}/poly) is the class of languages L that can be recognized by a non-uniform sequence of polynomial time “machines.” Namely, $L \in \mathcal{P}/\text{poly}$ if there exists an infinite sequence of machines M_1, M_2, \dots satisfying the following:*

1. *There exists a polynomial $p(\cdot)$ such that for every n , the description of machine M_n has length bounded above by $p(n)$.*
2. *There exists a polynomial $q(\cdot)$ such that for every n , the running time of machine M_n on each input of length n is bounded above by $q(n)$.*
3. *For every n and every $x \in \{0, 1\}^n$, machine M_n will accept x if and only if $x \in L$.*

Note that the non-uniformity is implicit in the absence of a requirement concerning the construction of the machines in the sequence. It is required only that these machines exist. In contrast, if we augment Definition 1.3.6 by requiring the existence of a polynomial-time algorithm that on input 1^n (n presented in unary) outputs the description of M_n , then we get a cumbersome way of defining \mathcal{P} . On the other hand, it

is obvious that $\mathcal{P} \subseteq \mathcal{P}/\text{poly}$ (in fact, strict containment can be proved by considering non-recursive unary languages). Furthermore:

Theorem 1.3.7: $\mathcal{BPP} \subseteq \mathcal{P}/\text{poly}$.

Proof: Let M be a probabilistic polynomial-time Turing machine recognizing $L \in \mathcal{BPP}$. Let $\chi_L(x) \stackrel{\text{def}}{=} 1$ if $x \in L$, and $\chi_L(x) \stackrel{\text{def}}{=} 0$ otherwise. Then, for every $x \in \{0, 1\}^*$,

$$\Pr[M(x) = \chi_L(x)] \geq \frac{2}{3}$$

Assume, without loss of generality, that on each input of length n , machine M uses the same number, denoted $m = \text{poly}(n)$, of coin tosses. Let $x \in \{0, 1\}^n$. Clearly, we can find for each $x \in \{0, 1\}^n$ a sequence of coin tosses $r \in \{0, 1\}^m$ such that $M_r(x) = \chi_L(x)$ (in fact, most sequences r have this property). But can one sequence $r \in \{0, 1\}^m$ fit all $x \in \{0, 1\}^n$? Probably not. (Provide an example!) Nevertheless, we can find a sequence $r \in \{0, 1\}^m$ that fits $\frac{2}{3}$ of all the x 's of length n . This is done by an averaging argument (which asserts that if $\frac{2}{3}$ of the r 's are good for each x , then there is an r that is good for at least $\frac{2}{3}$ of the x 's). However, this does not give us an r that is good for all $x \in \{0, 1\}^n$. To get such an r , we have to apply the preceding argument on a machine M' with exponentially vanishing error probability. Such a machine is guaranteed by Exercise 5. Namely, for every $x \in \{0, 1\}^*$,

$$\Pr[M'(x) = \chi_L(x)] > 1 - 2^{-|x|}$$

Applying the averaging argument, now we conclude that there exists an $r \in \{0, 1\}^m$, denoted r_n , that is good for *more than* a $1 - 2^{-n}$ fraction of the x 's in $\{0, 1\}^n$. It follows that r_n is good for all the 2^n inputs of length n . Machine M' (viewed as a deterministic two-input machine), together with the infinite sequence r_1, r_2, \dots constructed as before, demonstrates that L is in \mathcal{P}/poly . ■

Non-Uniform Circuit Families. A more convenient way of viewing non-uniform polynomial time, which is actually the way used in this book, is via (non-uniform) families of polynomial-size Boolean circuits. A *Boolean circuit* is a directed acyclic graph with internal nodes marked by elements of $\{\wedge, \vee, \neg\}$. Nodes with no in-going edges are called *input nodes*, and nodes with no out-going edges are called *output nodes*. A node marked \neg can have only one child. Computation in the circuit begins with placing input bits on the input nodes (one bit per node) and proceeds as follows. If the children of a node (of in-degree d) marked \wedge have values v_1, v_2, \dots, v_d , then the node gets the value $\wedge_{i=1}^d v_i$. Similarly for nodes marked \vee and \neg . The output of the circuit is read from its output nodes. The *size* of a circuit is the number of its edges. A *polynomial-size circuit family* is an infinite sequence of Boolean circuits C_1, C_2, \dots such that for every n , the circuit C_n has n input nodes and size $p(n)$, where $p(\cdot)$ is a polynomial (fixed for the entire family).

The computation of a Turing machine M on inputs of length n can be simulated by a single circuit (with n input nodes) having size $O((|M| + n + t(n))^2)$, where $t(n)$ is a bound on the running time of M on inputs of length n . Thus, a non-uniform sequence of polynomial-time machines can be simulated by a non-uniform family of polynomial-size circuits. The converse is also true, because machines with polynomial description lengths can incorporate polynomial-size circuits and simulate their computations in polynomial time. The thing that is nice about the circuit formulation is that there is no need to repeat the polynomiality requirement twice (once for size and once for time) as in the first formulation.

Convention. For the sake of simplicity, we often take the liberty of considering circuit families $\{C_n\}_{n \in \mathbb{N}}$, where each C_n has $\text{poly}(n)$ input bits rather than n .

1.3.4. Intractability Assumptions

We shall consider as *intractable* those tasks that cannot be performed by probabilistic polynomial-time machines. However, the adversarial tasks in which we shall be interested (“breaking an encryption scheme,” “forging signatures,” etc.) can be performed by non-deterministic polynomial-time machines (because the solutions, once found, can be easily tested for validity). Thus, the computational approach to cryptography (and, in particular, most of the material in this book) is *interesting* only if \mathcal{NP} is not contained in \mathcal{BPP} (which certainly implies $\mathcal{P} \neq \mathcal{NP}$).⁷ We use the phrase “not interesting” (rather than “not valid”) because all our statements will be of the form “if $\langle \text{intractability assumption} \rangle$ then $\langle \text{useful consequence} \rangle$,” Such a statement remains valid even if $\mathcal{P} = \mathcal{NP}$ (or if just $\langle \text{intractability assumption} \rangle$, which is never weaker than $\mathcal{P} \neq \mathcal{NP}$, is wrong); but in such a case the implication is of little interest (because everything is implied by a fallacy).

In most places where we state that “if $\langle \text{intractability assumption} \rangle$ then $\langle \text{useful consequence} \rangle$,” it will be the case that $\langle \text{useful consequence} \rangle$ either implies $\langle \text{intractability assumption} \rangle$ or implies some weaker form of it, which in turn implies $\mathcal{NP} \setminus \mathcal{BPP} \neq \emptyset$. Thus, in light of the current state of knowledge in complexity theory, we cannot hope to assert $\langle \text{useful consequence} \rangle$ without any intractability assumption.

In a few cases, an assumption concerning the limitations of probabilistic polynomial-time machines shall not suffice, and we shall use instead an assumption concerning the limitations of non-uniform polynomial-time machines. Such an assumption is of course stronger. But also the consequences in such a case will be stronger, since they will also be phrased in terms of non-uniform complexity. However, because all our proofs are obtained by reductions, an implication stated in terms of probabilistic polynomial time is stronger (than one stated in terms of non-uniform polynomial time) and will be preferred unless it is either not known or too complicated. This is the case because a probabilistic

⁷We remark that \mathcal{NP} is not known to contain \mathcal{BPP} . This is the reason we state the foregoing conjecture as \mathcal{NP} is not contained in \mathcal{BPP} , rather than $\mathcal{BPP} \neq \mathcal{NP}$. Likewise, although “sufficiently strong” one-way functions imply $\mathcal{BPP} = \mathcal{P}$, this equality is not known to hold unconditionally.

polynomial-time reduction (proving implication in its probabilistic formalization) always implies a non-uniform polynomial-time reduction (proving the statement in its non-uniform formalization), but the converse is not always true.⁸

Finally, we mention that intractability assumptions concerning worst-case complexity (e.g., $\mathcal{P} \neq \mathcal{NP}$) will not suffice, because we shall *not be satisfied* with their corresponding consequences. Cryptographic schemes that are guaranteed to be *hard to break only in the worst case* are useless. A cryptographic scheme must be unbreakable in “most cases” (i.e., “typical cases”), which implies that it will be *hard to break on the average*. It follows that because we are not able to prove that “worst-case intractability” implies analogous “intractability for the average case” (such a result would be considered a breakthrough in complexity theory), our intractability assumption must concern average-case complexity.

1.3.5. Oracle Machines

The original utility of oracle machines in complexity theory was to capture notions of reducibility. In this book (mainly in Chapters 5 and 6) we use oracle machines mainly for a different purpose altogether – to model an adversary that may use a cryptosystem in the course of its attempt to break the system. Other uses of oracle machines are discussed in Sections 3.6 and 4.7.

Loosely speaking, an oracle machine is a machine that is augmented so that it can ask questions to the outside. We consider the case in which these questions (called queries) are answered consistently by some function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, called the oracle. That is, if the machine makes a query q , then the answer it obtains is $f(q)$. In such a case, we say that the oracle machine is given access to the oracle f .

Definition 1.3.8 (Oracle Machines): *A (deterministic/probabilistic) oracle machine is a (deterministic/probabilistic) Turing machine with an additional tape, called the **oracle tape**, and two special states, called **oracle invocation** and **oracle appeared**. The computation of the deterministic oracle machine M on input x and with access to the oracle $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is defined by the successive-configuration relation. For configurations with states different from oracle invocation, the next configuration is defined as usual. Let γ be a configuration in which the state is oracle invocation and the content of the oracle tape is q . Then the configuration following γ is identical to γ , except that the state is oracle appeared, and the content of the oracle tape is $f(q)$. The string q is called M ’s **query**, and $f(q)$ is called the **oracle reply**. The computation of a probabilistic oracle machine is defined analogously. The output distribution of the oracle machine M , on input x and with access to the oracle f , is denoted $M^f(x)$.*

We stress that the running time of an oracle machine is the number of steps made during its computation and that the oracle’s reply to each query is obtained in a single step.

⁸The current paragraph may be better understood in the future, after seeing some concrete examples.

1.4. Motivation to the Rigorous Treatment

In this section we address three related issues:

1. the mere need for a rigorous treatment of the field,
2. the practical meaning and/or consequences of the rigorous treatment, and
3. the “conservative” tendencies of the treatment.

Parts of this section (corresponding to Items 2 and 3) are likely to become more clear after reading any of the following chapters.

1.4.1. The Need for a Rigorous Treatment

*If the truth of a proposition does not follow
from the fact that it is self-evident to us,
then its self-evidence in no way justifies our belief in its truth.*
Ludwig Wittgenstein, *Tractatus logico-philosophicus* (1921)

Cryptography is concerned with the construction of schemes that will be robust against malicious attempts to make these schemes deviate from their prescribed functionality. Given a desired functionality, a cryptographer should design a scheme that not only will satisfy the desired functionality under “normal operation” but also will maintain that functionality in face of adversarial attempts that will be devised after the cryptographer has completed the design. The fact that an adversary will devise its attack after the scheme has been specified makes the design of such schemes very hard. In particular, the adversary will try to take actions other than the ones the designer has envisioned. Thus, *the evaluation of cryptographic schemes* must take account of a practically infinite set of adversarial strategies. It is useless to make assumptions regarding the specific *strategy* that an adversary may use. The only assumptions that can be justified will concern the computational *abilities* of the adversary. To summarize, an evaluation of a cryptographic scheme is a study of an infinite set of potential strategies (which are not explicitly given). Such a highly complex study cannot be carried out properly without great care (i.e., rigor).

The design of cryptographic systems must be based on *firm foundations*, whereas ad hoc approaches and heuristics are a very dangerous way to go. Although always inferior to a rigorously analyzed solution, a heuristic may make sense when the designer has a very good idea about the environment in which a scheme is to operate. Yet a cryptographic scheme has to operate in a maliciously selected environment that typically will transcend the designer’s view. Under such circumstances, heuristics make little sense (if at all).

In addition to these straightforward considerations, we wish to stress two additional aspects.

On Trusting Unsound Intuitions. We believe that *one* of the roles of science is to formulate, examine, and refine our intuition about reality. A rigorous formulation is

required in order to allow a careful examination that may lead either to verification and justification of our intuition or to its rejection as false (or as something that is true only in certain cases or only under certain refinements). There are many cases in which our initial intuition turns out to be correct, as well as many cases in which our initial intuition turns out to be wrong. The more we understand the discipline, the better our intuition becomes.

At this stage in history, it would be very presumptuous to claim that we have good intuition about the *nature of efficient computation*. In particular, we do not even know the answers to such basic questions as whether or not \mathcal{P} is strictly contained in \mathcal{NP} , let alone have an understanding of what makes one computational problem hard while a seemingly related problem is easy. Consequently, we should be extremely careful when making assertions about what can or cannot be efficiently computed. Unfortunately, *making assertions about what can or cannot be efficiently computed is exactly what cryptography is all about*. Worse yet, many of the problems of cryptography have much more complex and cumbersome descriptions than are usually encountered in complexity theory. To summarize, cryptography deals with very complex computational notions and currently must do so without having a good understanding of much simpler computational notions. Hence, our current intuitions about cryptography must be considered highly unsound until they can be formalized and examined carefully. In other words, the general need to formalize and examine intuition becomes even more acute in a highly sensitive field such as cryptography that is intimately concerned with questions we hardly understand.

The Track Record. Cryptography, as a discipline, is well motivated. Consequently, cryptographic issues are being discussed by many researchers, engineers, and laypersons. Unfortunately, most such discussions are carried out without precise definitions of the subject matter. Instead, it is implicitly assumed that the basic concepts of cryptography (e.g., secure encryption) are self-evident (because they are so natural) and that there is no need to present adequate definitions. The fallacy of that assumption is demonstrated by the abandon of papers (not to mention private discussions) that derive and/or jump to wrong conclusions concerning security. In most cases these wrong conclusions can be traced back to implicit misconceptions regarding security that could not have escaped the eyes of the authors if they had been made explicit. We avoid listing all such cases here for several obvious reasons. Nevertheless, we shall mention one well-known example.

Around 1979, Ron Rivest claimed that no signature scheme that was “proven secure assuming the intractability of factoring” could resist a “chosen message attack.” His argument was based on an implicit (and unjustified) assumption concerning the nature of a “proof of security (which assumes the intractability of factoring).” Consequently, for several years it was believed that one had to choose between having a signature scheme “proven to be unforgeable under the intractability of factoring” and having a signature scheme that could resist a “chosen message attack.” However, in 1984, Goldwasser, Micali, and Rivest pointed out the fallacy on which Rivest’s 1979 argument had been based and furthermore presented signature schemes that could resist a “chosen message attack,” under general assumptions. In particular, the intractability of factoring suffices

to prove that there exists a signature scheme that can resist “forgery,” even under a “chosen message attack.”

To summarize, the basic concepts of cryptography are indeed very natural, but they are *not* self-evident nor well understood. Hence, we do not yet understand these concepts well enough to be able to discuss them *correctly* without using precise definitions and rigorously justifying every statement made.

1.4.2. Practical Consequences of the Rigorous Treatment

As customary in complexity theory, our treatment is presented in terms of asymptotic analysis of algorithms. (Actually, it would be more precise to use the term “functional analysis of running time.”) This makes the treatment less cumbersome, but it is *not* essential to the underlying ideas. In particular, the definitional approach taken in this book (e.g., the definitions of one-way functions, pseudorandom generators, zero-knowledge proofs, secure encryption schemes, unforgeable signature schemes, and secure protocols) is based on general paradigms that remain valid in any reasonable computational model. In particular, the definitions, although stated in an “abstract manner,” lend themselves to concrete interpolations. The same holds with respect to the results that typically relate several such definitions. To clarify the foregoing, we shall consider, as an example, the statement of a generic result as presented in this book.

A typical result presented in this book relates two computational problems. The first problem is a simple computational problem that is assumed to be intractable (e.g., intractability of factoring), whereas the second problem consists of “breaking” a specific implementation of a useful cryptographic primitive (e.g., a specific encryption scheme). The abstract statement may assert that if integer factoring cannot be performed in polynomial time, then the encryption scheme is secure in the sense that it cannot be “broken” in polynomial time. Typically, the statement is proved by a fixed polynomial-time reduction of integer factorization to the problem of breaking the encryption scheme. Hence, what is actually being proved is that if one can break the scheme in time $T(n)$, where n is the security parameter (e.g., key length), then one can factor integers of length m in time $T'(m) = f(m, T(g(m)))$, where f and g are fixed polynomials that are at least implicit in the proof. In order to determine the practicality of the result, one should first determine these polynomials (f and g). For most of the basic results presented in this book, these polynomials are reasonably small, in the sense that instantiating a scheme with a reasonable security parameter and making reasonable intractability assumptions (e.g., regarding factoring) will yield a scheme that it is infeasible to break in practice. (In the exceptional cases, we say so explicitly and view these results as merely claims of the plausibility of relating the two notions.) We actually distinguish three types of results:

1. *Plausibility results*: Here we refer to results that are aimed either at establishing a connection between two notions or at providing a generic way of solving a class of problems.

A result of the first type says that, in principle, X (e.g., a specific tool) can be used in order to construct Y (e.g., a useful utility), but the specific construction provided in the proof may be impractical. Still, such a result may be useful in practice because it suggests that one may be able to use *specific* implementations of X in order to provide a

practical construction of Y . At the very least, such a result can be viewed as a challenge to the researchers to either provide a practical construction of Y using X or explain why a practical construction cannot be provided.

A result of the second type says that any task that belongs to some class \mathcal{C} is solvable, but the generic construction provided in the proof may be impractical. Still, this is a very valuable piece of information: If we have a specific problem that falls into the foregoing class, then we know that the problem is solvable in principle. However, if we need to construct a real system, then we probably should construct a solution from scratch (rather than employing the preceding generic result).

To summarize, in both cases a plausibility result provides very useful information (even if it does not yield a practical solution). Furthermore, it is often the case that *some* tools developed toward proving a plausibility result may be useful in solving the specific problem at hand. This is typically the case for the next type of results.

2. *Introduction of paradigms and techniques that may be applicable in practice:* Here we refer to results that are aimed at introducing a new notion, model, tool, or technique. Such results (e.g., techniques) typically are applicable in practice, either as presented in the original work or, after further refinements, or at least as an inspiration.
3. *Presentation of schemes that are suitable for practical applications.*

Typically, it is quite easy to determine to which of the foregoing categories a specific result belongs. Unfortunately, the classification is not always stated in the original paper; however, typically it is evident from the construction. We stress that all results of which we are aware (in particular, all results mentioned in this book) come with an explicit construction. Furthermore, the security of the resulting construction is explicitly related to the complexity of certain intractable tasks. Contrary to some uninformed beliefs, for each of these results there is an explicit translation of concrete intractability assumptions (on which the scheme is based) into lower bounds on the amount of work required to violate the security of the resulting scheme.⁹ We stress that this translation can be invoked for any value of the security parameter. Doing so will determine whether a specific construction is adequate for a specific application under specific reasonable intractability assumptions. In many cases the answer is in the affirmative, but in general this does depend on the specific construction, as well as on the specific value of the security parameter and on what it is reasonable to assume for this value (of the security parameter).

1.4.3. The Tendency to Be Conservative

When reaching the chapters in which cryptographic primitives are defined, the reader may notice that we are unrealistically “conservative” in our definitions of security. In other words, we are unrealistically liberal in our definition of insecurity. Technically speaking, this tendency raises no problems, because our primitives that are secure in a very strong sense certainly are also secure in the (more restricted) reasonable sense. Furthermore, we are able to implement such (strongly secure) primitives using

⁹The only exception to the latter statement is Levin’s observation regarding the existence of a *universal one-way function* (see Section 2.4.1).

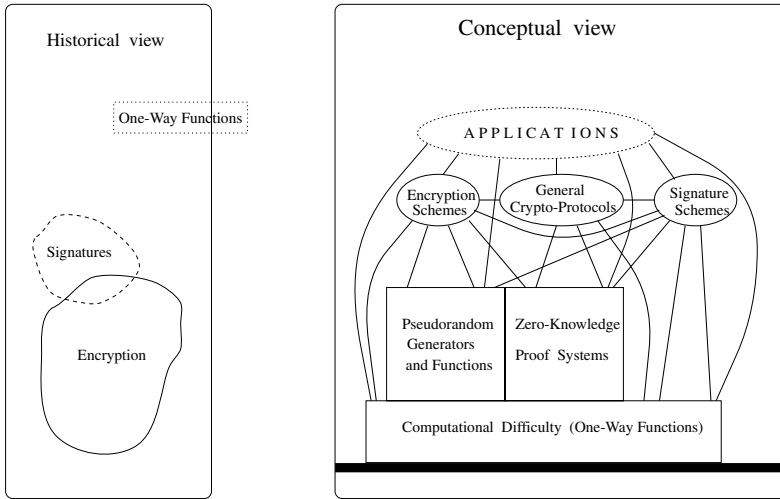


Figure 1.1: Cryptography: two points of view.

reasonable intractability assumptions, and in most cases we can show that such assumptions are necessary even for much weaker (and, in fact, less than minimal) notions of security. Yet the reader may wonder why we choose to present definitions that seem stronger than what is required in practice.

The reason for our tendency to be conservative when defining security is that it is extremely difficult to capture what is *exactly* required in a specific practical application. Furthermore, each practical application has different requirements, and it is undesirable to redesign the system for each new application. Thus, we actually need to address the security concerns of all future applications (which are unknown to us), not merely the security concerns of some known applications. It seems impossible to cover whatever can be required in all applications (or even in some wide set of applications) without taking our conservative approach.¹⁰ In the sequel, we shall see how our conservative approach leads to definitions of security that can cover all possible practical applications.

1.5. Miscellaneous

In Figure 1.1 we confront the “historical view” of cryptography (i.e., the view of the field in the mid-1970s) with the approach advocated in this text.

1.5.1. Historical Notes

Work done during the 1980s plays a dominant role in our exposition. That work, in turn, had been tremendously influenced by previous work, but those early influences are not stated explicitly in the historical notes to subsequent chapters. In this section we shall

¹⁰One may even argue that it seems impossible to cover whatever is required in one reasonable application without taking our conservative approach.

trace some of those influences. Generally speaking, those influences took the form of setting intuitive goals, providing basic techniques, and suggesting potential solutions that later served as a basis for constructive criticism (leading to robust approaches).

Classic Cryptography. Answering the fundamental question of classic cryptography in a gloomy way (i.e., it is *impossible* to design a code that cannot be broken), Shannon [200] also suggested a modification to the question: Rather than asking whether or not it is *possible* to break a code, one should ask whether or not it is *feasible* to break it. A code should be considered good if it cannot be broken by an investment of work that is in reasonable proportion to the work required of the legal parties using the code. Indeed, this is the approach followed by modern cryptography.

New Directions in Cryptography. The prospect for commercial applications was the trigger for the beginning of civil investigations of encryption schemes. The DES block-cipher [168], designed in the early 1970s, has adopted the new paradigm: It is clearly *possible*, but supposedly *infeasible*, to break it. Following the challenge of constructing and analyzing new (private-key) encryption schemes came new questions, such as how to exchange keys over an insecure channel [159]. New concepts were invented: *digital signatures* (cf., Diffie and Hellman [63] and Rabin [186]), *public-key cryptosystems* [63], and *one-way functions* [63]. First implementations of these concepts were suggested by Merkle and Hellman [163], Rivest, Shamir, and Adleman [191], and Rabin [187].

Cryptography was explicitly related to complexity theory in the late 1970s [39, 73, 147]: It was understood that problems related to breaking a 1-1 cryptographic mapping could not be \mathcal{NP} -complete and, more important, that \mathcal{NP} -hardness of the breaking task was poor evidence for cryptographic security. Techniques such as “*n*-out-of-*2n* verification” [186] and secret sharing [196] were introduced (and indeed were used extensively in subsequent research).

At the Dawn of a New Era. Early investigations of cryptographic protocols revealed the inadequacy of imprecise notions of security, as well as the subtleties involved in designing cryptographic protocols. In particular, problems such as *coin tossing over the telephone* [31], *exchange of secrets* [30], and *oblivious transfer* [188] (cf. [70]) were formulated. Doubts (raised by Lipton) concerning the security of the mental-poker protocol of [199] led to the current notion of secure encryption, due to Goldwasser and Micali [123], and to concepts such as computational indistinguishability [123, 210]. Doubts (raised by Fischer) concerning the oblivious-transfer protocol of [188] led to the concept of zero-knowledge (suggested by Goldwasser, Micali, and Rackoff [124], with early versions dating to March 1982).

A formal approach to the security of cryptographic protocols was suggested in [65]. That approach actually identified a *subclass* of insecure protocols (i.e., those that could be broken via a syntactically restricted type of attack). Furthermore, it turned out that it was much too difficult to test whether or not a given protocol was secure [69]. Recall that, in contrast, our current approach is to construct secure protocols (along with their proofs of security) and that this approach is *complete* (in the sense that it allows us to solve any solvable problem).

1.5.2. Suggestions for Further Reading

The background material concerning probability theory and computational complexity provided in Sections 1.2 and 1.3, respectively, should suffice for the purposes of this book. Still, a reader feeling uncomfortable with any of these areas may want to consult some standard textbooks on probability theory (e.g., [79]) and computational complexity (e.g., [86; 202]). The reader may also benefit from familiarity with randomized computation [167; 97, app. B].

Computational problems in number theory have provided popular candidates for one-way functions (and related intractability assumptions). However, because this book focuses on concepts, rather than on specific implementation of such concepts, those popular candidates will not play a major role in the exposition. Consequently, background in computational number theory is not really necessary for this book. Still, a brief description of relevant facts appears in Appendix A herein, and the interested reader is referred to other text (e.g., [10]).

This book focuses on the basic concepts, definitions, and results in cryptography. As argued earlier, these are of key importance for sound practice. However, practice needs much more than a sound theoretical framework, whereas this book makes no attempt to provide anything beyond the latter. For helpful suggestions concerning practice (i.e., applied cryptography), the reader may *critically* consult other texts (e.g., [158]).

Our treatment is presented in terms of asymptotic analysis of algorithms. Furthermore, for simplicity, we state results in terms of robustness against polynomial-time adversaries, rather than discussing general time-bounded adversaries. However, as explained in Section 1.4, none of these conventions is essential, and the results could be stated in general terms, as done in Section 2.6 and elsewhere (e.g., [155]). Our choice to present results in terms of robustness against polynomial-time adversaries makes the statement of the results somewhat less cumbersome, because it avoids stating the exact complexity of the reduction by which security is proved. This suffices for stating plausibility results, which is indeed our main objective in this book, but when using schemes in practice one needs to know the exact complexity of the reduction (for that will determine the actual security of the concrete scheme). We stress that typically it is easy to determine the complexity of the reductions presented in this book, and in some cases we also include comments referring to this aspect. We mention that the alternative choice, of presenting results in terms of robustness against general time-bounded adversaries, is taken in Luby's book [155].

1.5.3. Open Problems

As mentioned earlier (and further formalized in the next chapter), most of the content of this book relies on the existence of one-way functions. Currently, we assume the existence of such functions; it is to be hoped that the new century will witness a proof of this widely believed assumption/conjecture. We mention that the existence of one-way functions implies that \mathcal{NP} is not contained in \mathcal{BPP} , and thus would establish that $\mathcal{NP} \neq \mathcal{P}$ (which is the most famous open problem in computer science).

We mention that $\mathcal{NP} \neq \mathcal{P}$ is not known to imply any practical consequences. For the latter, it would be required that hard instances not only exist but also occur quite frequently (with respect to some easy-to-sample distribution). For further discussion, see Chapter 2.

1.5.4. Exercises

Exercise 1: *Applications of Markov's inequality:*

1. Let X be a random variable such that $E(X) = \mu$ and $X \leq 2\mu$. Give an upper bound on $\Pr[X \leq \frac{\mu}{2}]$.
2. Let $0 < \varepsilon$ and $\delta < 1$, and let Y be a random variable ranging in the interval $[0, 1]$ such that $E(Y) = \delta + \varepsilon$. Give a lower bound on $\Pr[Y \geq \delta + \frac{\varepsilon}{2}]$.

Guideline: In both cases, one can define auxiliary random variables and apply Markov's inequality. However, it is easier simply to apply directly the reasoning underlying the proof of Markov's inequality.

Exercise 2: *Applications of Chernoff/Hoeffding bounds:* Let $f : \{0, 1\}^* \rightarrow [0, 1]$ be a polynomial-time-computable function, and let $F(n)$ denote the average value of f over $\{0, 1\}^n$. Namely,

$$F(n) \stackrel{\text{def}}{=} \frac{\sum_{x \in \{0, 1\}^n} f(x)}{2^n}$$

Let $p(\cdot)$ be a polynomial. Present a probabilistic polynomial-time algorithm that on input 1^n will output an estimate to $F(n)$, denoted $A(n)$, such that

$$\Pr \left[|F(n) - A(n)| > \frac{1}{p(n)} \right] < 2^{-n}$$

Guideline: The algorithm selects at random polynomially many (how many?) sample points $s_i \in \{0, 1\}^n$. These points are selected independently and with uniform probability distribution. (Why?) The algorithm outputs the average value taken over this sample. Analyze the performance of the algorithm using the Hoeffding inequality. (Hint: Define random variables X_i such that $X_i = f(s_i)$.)

Exercise 3: *Analysis of the graph-connectivity algorithm:* Regarding the algorithm presented in Section 1.3.2.1, show that if s is connected to t in the graph G , then, with probability at least $\frac{2}{3}$, vertex t will be encountered in a random walk starting at s .

Guideline: Consider the connected component of vertex s , denoted $G' = (V', E')$. For any edge (u, v) in E' , let $T_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is first encountered. First, prove that $E[T_{u,v}] \leq 2|E'|$. (Hint: Consider the “frequency” with which this edge is traversed in a certain direction during an infinite random walk, and note that this frequency is independent of the identity of the edge and the direction.) Next, letting $\text{cover}(G')$ be the expected number of steps in a random walk starting at s and ending when the last of the vertices of V' is encountered, prove that $\text{cover}(G') \leq 4 \cdot |V'| \cdot |E'|$. (Hint: consider a cyclic tour C going through all vertices of G' , and show that $\text{cover}(G') \leq \sum_{(u,v) \in C} E[T_{u,v}]$.) Conclude by applying Markov's inequality.

Exercise 4: *Equivalent definition of \mathcal{BPP} . Part 1:* Prove that Definition 1.3.4 is robust when $\frac{2}{3}$ is replaced by $\frac{1}{2} + \frac{1}{p(|x|)}$ for every positive polynomial $p(\cdot)$. Namely, show that $L \in \mathcal{BPP}$ if there exists a polynomial $p(\cdot)$ and a probabilistic polynomial-time machine M such that

- for every $x \in L$ it holds that $\Pr[M(x)=1] \geq \frac{1}{2} + \frac{1}{p(|x|)}$, and
- for every $x \notin L$ it holds that $\Pr[M(x)=0] \geq \frac{1}{2} + \frac{1}{p(|x|)}$.

Guideline: Given a probabilistic polynomial-time machine M satisfying the foregoing condition, construct a probabilistic polynomial-time machine M' as follows. On input x , machine M' runs $O(p(|x|)^2)$ many copies of M , on the same input x , and rules by majority. Use Chebyshev's inequality (see Section 1.2) to show that M' is correct with probability at least $\frac{2}{3}$.

Exercise 5: *Equivalent definition of \mathcal{BPP} . Part 2:* Prove that Definition 1.3.4 is robust when $\frac{2}{3}$ is replaced by $1 - 2^{-p(|x|)}$ for every positive polynomial $p(\cdot)$. Namely, show that for every $L \in \mathcal{BPP}$ and every polynomial $p(\cdot)$, there exists a probabilistic polynomial-time machine M such that

- for every $x \in L$ it holds that $\Pr[M(x)=1] \geq 1 - 2^{-p(|x|)}$, and
- for every $x \notin L$ it holds that $\Pr[M(x)=0] \geq 1 - 2^{-p(|x|)}$.

Guideline: Similar to Exercise 4, except that you have to use a stronger probabilistic inequality (namely, Chernoff bound; see Section 1.2).

Computational Difficulty

In this chapter we define and study one-way functions. One-way functions capture our notion of “useful” computational difficulty and serve as a basis for most of the results presented in this book. Loosely speaking, a one-way function is a function that is easy to evaluate but hard to invert (in an average-case sense). (See the illustration in Figure 2.1.) In particular, we define strong and weak one-way functions and prove that the existence of weak one-way functions implies the existence of strong ones. The proof provides a good example of a *reducibility argument*, which is a strong type of “reduction” used to establish most of the results in the area. Furthermore, the proof provides a simple example of a case where a computational statement is much harder to prove than its “information-theoretic analogue.”

In addition, we define hard-core predicates and prove that every one-way function has a hard-core predicate. Hard-core predicates will play an important role in almost all subsequent chapters (the chapter on signature scheme being the exception).

Organization. In Section 2.1 we motivate the definition of one-way functions by arguing informally that it is implicit in various natural cryptographic primitives. The basic definitions are given in Section 2.2, and in Section 2.3 we show that weak one-way functions can be used to construct strong ones. A more efficient construction (for certain restricted cases) is postponed to Section 2.6. In Section 2.4 we view one-way functions as uniform collections of finite functions and consider various additional properties that such collections may have. In Section 2.5 we define hard-core predicates and show how to construct them from one-way functions.

Teaching Tip. As stated earlier, the proof that the existence of weak one-way functions implies the existence of strong ones (see Section 2.3) is instructive for the rest of the material. Thus, if you choose to skip this proof, do incorporate a discussion of the *reducibility argument* in the first place you use it (e.g., when showing how to construct hard-core predicates from one-way functions).

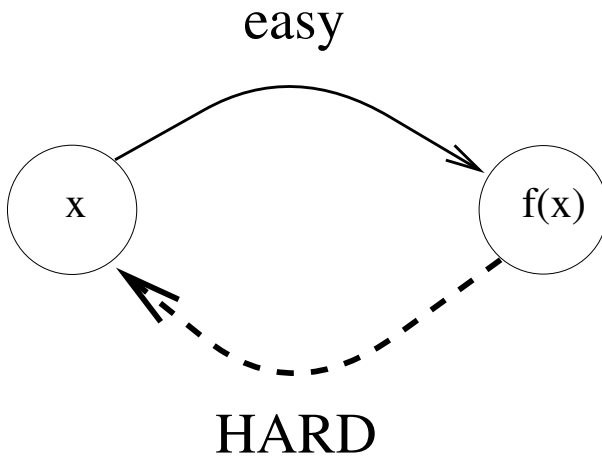


Figure 2.1: One-way functions: an illustration.

2.1. One-Way Functions: Motivation

As stated in the introductory chapter, modern cryptography is based on a gap between efficient algorithms provided for the legitimate users and the computational infeasibility of abusing or breaking these algorithms (via illegitimate adversarial actions). To illustrate this gap, we concentrate on the cryptographic task of secure data communication, namely, encryption schemes.

In secure encryption schemes, the legitimate users should be able to easily decipher the messages using some private information available to them, yet an adversary (not having this private information) should not be able to decrypt the ciphertext efficiently (i.e., in probabilistic polynomial time).¹ On the other hand, a non-deterministic machine can quickly decrypt the ciphertext (e.g., by guessing the private information). Hence, the existence of secure encryption schemes implies that there are tasks (e.g., “breaking” encryption schemes) that can be performed by non-deterministic polynomial-time machines, yet cannot be performed by deterministic (or even randomized) polynomial-time machines. In other words, a necessary condition for the existence of secure encryption schemes is that \mathcal{NP} not be contained in \mathcal{BPP} (and thus $\mathcal{P} \neq \mathcal{NP}$).

Although $\mathcal{P} \neq \mathcal{NP}$ is a necessary condition for modern cryptography, it is not a sufficient one. Suppose that the breaking of some encryption scheme is \mathcal{NP} -complete. Then, $\mathcal{P} \neq \mathcal{NP}$ implies that this encryption scheme is hard to break in the worst case, but it does not rule out the possibility that the encryption scheme is easy to break almost always. In fact, one can construct “encryption schemes” for which the breaking problem is \mathcal{NP} -complete and yet there exists an efficient breaking algorithm that succeeds 99% of the time. Hence, worst-case hardness is a poor measure of security. Security requires hardness in most cases, or at least “average-case hardness.” A necessary condition for the existence of secure encryption schemes is thus the existence of languages in \mathcal{NP}

¹This “private information” is called a key; see Chapter 5.

that are hard on the average. We mention that it is not known whether or not $\mathcal{P} \neq \mathcal{NP}$ implies the existence of languages in \mathcal{NP} that are hard on the average.

Furthermore, the mere existence of problems (in \mathcal{NP}) that are hard on the average does not suffice either. In order to be able to use such hard-on-the-average problems, we must be able to generate hard instances together with auxiliary information that will enable us to solve these instances fast. Otherwise, these hard instances will be hard also for the legitimate users, and consequently the legitimate users will gain no computational advantage over the adversary. Hence, the existence of secure encryption schemes implies the existence of an efficient way (i.e., probabilistic polynomial-time algorithm) to generate instances with corresponding auxiliary input such that

1. it is easy to solve these instances given the auxiliary input, but
2. it is hard, on the average, to solve these instances when not given the auxiliary input.

The foregoing requirement is reflected in the definition of one-way functions (as presented in the next section). Loosely speaking, a one-way function is a function that is easy to compute but hard (on the average) to invert. Thus, one-way functions capture the hardness of reversing the process of generating instances (and obtaining the auxiliary input from the instance alone), which is responsible for the discrepancy between the preceding two items. (For further discussion of this relationship, see Exercise 1.)

In assuming that one-way functions exist, we are postulating the existence of efficient processes (i.e., the computation of the function in the forward direction) that are hard to reverse. Note that such processes of daily life are known to us in abundance (e.g., the lighting of a match). The assumption that one-way functions exist is thus a complexity-theoretic analogue of daily experience.

2.2. One-Way Functions: Definitions

In this section, we present several definitions of one-way functions. The first version, hereafter referred to as a strong one-way function (or just one-way function), is the most popular one. We also present weak one-way functions, non-uniformly one-way functions, and plausible candidates for such functions.

2.2.1. Strong One-Way Functions

Loosely speaking, a one-way function is a function that is easy to compute but hard to invert. The first condition is quite clear: Saying that a function f is easy to compute means that there exists a polynomial-time algorithm that on input x outputs $f(x)$. The second condition requires more elaboration. What we mean by saying that a function f is hard to invert is that every probabilistic polynomial-time algorithm trying, on input y , to find an inverse of y under f may succeed only with negligible (in $|y|$) probability, where the probability is taken over the choices of y (as discussed later). A sequence $\{s_n\}_{n \in \mathbb{N}}$ (resp., a function $\mu : \mathbb{N} \rightarrow \mathbb{R}$) is called *negligible* in n if for every positive polynomial $p(\cdot)$ and all sufficiently large n 's, it holds that $s_n < \frac{1}{p(n)}$ (resp., $\mu(n) < \frac{1}{p(n)}$). Further discussion follows the definition.

Definition 2.2.1 (Strong One-Way Functions): A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is called **(strongly) one-way** if the following two conditions hold:

1. Easy to compute: There exists a (deterministic) polynomial-time algorithm A such that on input x algorithm A outputs $f(x)$ (i.e., $A(x) = f(x)$).
2. Hard to invert: For every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[A'(f(U_n), 1^n) \in f^{-1}(f(U_n))] < \frac{1}{p(n)}$$

Recall that U_n denotes a random variable uniformly distributed over $\{0, 1\}^n$. Hence, the probability in the second condition is taken over all the possible values assigned to U_n and all possible internal coin tosses of A' , with uniform probability distribution. Note that A' is not required to output a specific pre-image of $f(x)$; any pre-image (i.e., element in the set $f^{-1}(f(x))$) will do. (Indeed, in case f is 1-1, the string x is the only pre-image of $f(x)$ under f ; but in general there may be other pre-images.)

The Auxiliary Input 1^n . In addition to an input in the range of f , the inverting algorithm A' is also given the length of the desired output (in unary notation). The main reason for this convention is to rule out the possibility that a function will be considered one-way merely because it drastically shrinks its input, and so the inverting algorithm just does not have enough time to print the desired output (i.e., the corresponding pre-image). Consider, for example, the function f_{len} defined by $f_{\text{len}}(x) = y$ such that y is the binary representation of the length of x (i.e., $f_{\text{len}}(x) = |x|$). Since $|f_{\text{len}}(x)| = \log_2 |x|$, no algorithm can invert f_{len} on y in time polynomial in $|y|$; yet there exists an obvious algorithm that inverts f_{len} on $y = f_{\text{len}}(x)$ in time polynomial in $|x|$ (e.g., by $|x| \mapsto 0^{|x|}$). In general, the auxiliary input $1^{|x|}$, provided in conjunction with the input $f(x)$, allows the inverting algorithm to run in time polynomial in the total length of the main input and the desired output. Note that in the special case of length-preserving functions f (i.e., $|f(x)| = |x|$ for all x 's), this auxiliary input is redundant. More generally, the auxiliary input is redundant if, given only $f(x)$, one can generate $1^{|x|}$ in time polynomial in $|x|$. (See Exercise 4 and Section 2.2.3.2.)

Further Discussion

Hardness to invert is interpreted (by the foregoing definition) as an upper bound on the success probability of efficient inverting algorithms. The probability is measured with respect to both the random choices of the inverting algorithm and the distribution of the (main) input to this algorithm (i.e., $f(x)$). The input distribution to the inverting algorithm is obtained by applying f to a uniformly selected $x \in \{0, 1\}^n$. If f induces a permutation on $\{0, 1\}^n$, then the input to the inverting algorithm is uniformly distributed over $\{0, 1\}^n$. However, in the general case where f is not necessarily a one-to-one function, the input distribution to the inverting algorithm may differ substantially from the uniform one. In any case, it is required that the success probability, defined over the aforementioned probability space, be negligible (as a function of the length of x). To

further clarify the condition placed on the success probability, we consider two trivial algorithms.

Random-Guess Algorithm A_1 . On input $(y, 1^n)$, algorithm A_1 uniformly selects and outputs a string of length n . We note that the success probability of A_1 equals the collision probability of the random variable $f(U_n)$ (i.e., $\sum_y \Pr[f(U_n) = y]^2$). That is, letting U'_n denote a random variable uniformly distributed over $\{0, 1\}^n$ independently of U_n , we have

$$\begin{aligned} \Pr[A_1(f(U_n), 1^n) \in f^{-1}(f(U_n))] &= \Pr[f(U'_n) = f(U_n)] \\ &= \sum_y \Pr[f(U_n) = y]^2 \geq 2^{-n} \end{aligned}$$

where the inequality is due to the fact that, for non-negative x_i 's summing to 1, the sum $\sum_i x_i^2$ is minimized when all x_i 's are equal. Thus, the last inequality becomes an equality if and only if f is a 1-1 function. Consequently:

1. For any function f , the success probability of the trivial algorithm A_1 is strictly positive. Thus, one cannot require that any efficient algorithm will *always* fail to invert f .
2. For any 1-1 function f , the success probability of A_1 in inverting f is negligible. Of course, this does not indicate that f is one-way (but rather that A_1 is trivial).
3. If f is one-way, then the collision probability of the random variable $f(U_n)$ is negligible. This follows from the fact that A_1 falls within the scope of the definition, and its success probability equals the collision probability.

Fixed-Output Algorithm A_2 . Another trivial algorithm, denoted A_2 , is one that computes a function that is constant on all inputs of the same length (e.g., $A_2(y, 1^n) = 0^n$). For every function f , we have

$$\begin{aligned} \Pr[A_2(f(U_n), 1^n) \in f^{-1}(f(U_n))] &= \Pr[f(0^n) = f(U_n)] \\ &= \frac{|f^{-1}(f(0^n))|}{2^n} \geq 2^{-n} \end{aligned}$$

with equality holding in case $f(0^n)$ has a single pre-image (i.e., 0^n itself) under f . Again we observe analogous facts:

1. For any function f , the success probability of the trivial algorithm A_2 is strictly positive.
2. For any 1-1 function f , the success probability of A_2 in inverting f is negligible.
3. If f is one-way, then the fraction of x 's in $\{0, 1\}^n$ that are mapped by f to $f(0^n)$ is negligible.

Obviously, Definition 2.2.1 considers all probabilistic polynomial-time algorithms, not merely the trivial ones discussed earlier. In some sense this definition asserts that for one-way functions, no probabilistic polynomial-time algorithm can “significantly” outperform these trivial algorithms.

Negligible Probability

A few words concerning the notion of negligible probability are in order. The foregoing definition and discussion consider the success probability of an algorithm to be *negligible* if, as a function of the input length, the success probability is bounded above by every polynomial fraction. It follows that repeating the algorithm polynomially (in the input length) many times yields a new algorithm that also has negligible success probability. In other words, events that occur with negligible (in n) probability remain negligible even if the experiment is repeated for polynomially (in n) many times. Hence, defining a negligible success rate as “occurring with probability smaller than any polynomial fraction” is naturally coupled with defining feasible computation as “computed within polynomial time.”

A “strong negation” of the notion of a negligible fraction/probability is the notion of a noticeable fraction/probability. We say that a function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is *noticeable* if there exists a polynomial $p(\cdot)$ such that for all sufficiently large n ’s, it holds that $\nu(n) > \frac{1}{p(n)}$. We stress that functions may be neither negligible nor noticeable.

2.2.2. Weak One-Way Functions

One-way functions, as defined earlier, are one-way in a very strong sense. Namely, any efficient inverting algorithm has negligible success in inverting them. A much weaker definition, presented next, requires only that all efficient inverting algorithms fail with some noticeable probability.

Definition 2.2.2 (Weak One-Way Functions): A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is called **weakly one-way** if the following two conditions hold:

1. Easy to compute: As in the definition of a strong one-way function.
2. Slightly hard to invert: There exists a polynomial $p(\cdot)$ such that for every probabilistic polynomial-time algorithm A' and all sufficiently large n ’s,

$$\Pr[A'(f(U_n), 1^n) \notin f^{-1}(f(U_n))] > \frac{1}{p(n)}$$

We call the reader’s attention to the order of quantifiers: There exists a *single* polynomial $p(\cdot)$ such that $1/p(n)$ lower-bounds the failure probability of *all* probabilistic polynomial-time algorithms trying to invert f on $f(U_n)$.

Weak one-way functions fail to provide the kind of hardness alluded to in the earlier motivational discussions. Still, as we shall see later, they can be converted into strong one-way functions, which in turn do provide such hardness.

2.2.3. Two Useful Length Conventions

In the sequel it will be convenient to use the following two conventions regarding the *lengths* of the pre-images and images of one-way functions. In the current section we justify the use of these conventions.

2.2.3.1. Functions Defined Only for Some Lengths

In many cases it is more convenient to consider one-way functions with domains partial to the set of all strings. In particular, this facilitates the introduction of some structure in the domain of the function. A particularly important case, used throughout the rest of this section, is that of functions with domain $\cup_{n \in \mathbb{N}} \{0, 1\}^{l(n)}$, where $l(\cdot)$ is some polynomial. We provide a more general treatment of this case.

Let $I \subseteq \mathbb{N}$, and denote by $s_I(n)$ the successor of n with respect to I ; namely, $s_I(n)$ is the smallest integer that is both greater than n and in the set I (i.e., $s_I(n) \stackrel{\text{def}}{=} \min\{i \in I : i > n\}$). A set $I \subseteq \mathbb{N}$ is called *polynomial-time-enumerable* if there exists an algorithm that on input n halts within $\text{poly}(n)$ steps and outputs $1^{s_I(n)}$. (The unary output forces s_I to be polynomially bounded; i.e., $s_I(n) \leq \text{poly}(n)$.) Let I be a polynomial-time-enumerable set and f be a function with domain $\cup_{n \in I} \{0, 1\}^n$. We call f strongly (resp., weakly) *one-way on lengths in I* if f is polynomial-time-computable and is hard to invert over n 's in I . For example, the hardness condition for functions that are strongly one-way on lengths in I is stated as follows:

For every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's in I ,

$$\Pr[A'(f(U_n), 1^n) \in f^{-1}(f(U_n))] < \frac{1}{p(n)}$$

Ordinary one-way functions, as defined in previous subsections, can be viewed as being one-way on lengths in \mathbb{N} .

One-way functions on lengths in any polynomial-time-enumerable set can be easily transformed into ordinary one-way functions (i.e., defined over all of $\{0, 1\}^*$). In particular, for any function f with domain $\cup_{n \in I} \{0, 1\}^n$, we can construct a function $g : \{0, 1\}^* \rightarrow \{0, 1\}^*$ by letting

$$g(x) \stackrel{\text{def}}{=} f(x') \tag{2.1}$$

where x' is the longest prefix of x with length in I . In case the function f is length-preserving (i.e., $|f(x)| = |x|$ for all x), we can preserve this property by modifying the construction to obtain a length-preserving function $g' : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that

$$g'(x) \stackrel{\text{def}}{=} f(x')x'' \tag{2.2}$$

where $x = x'x''$, and x' is the longest prefix of x with length in I .

Proposition 2.2.3: *Let I be a polynomial-time-enumerable set, and let f be strongly (resp., weakly) one-way on lengths in I . Then g and g' (as defined in Eq. (2.1) and Eq. (2.2), respectively) are strongly (resp., weakly) one-way (in the ordinary sense).*

Although the validity of the foregoing proposition is very appealing, we urge the reader not to skip the following proof. The proof, which is indeed quite simple, uses (for the first time in this book) an argument that is used extensively in the sequel. The argument used to prove the hardness-to-invert property of the function g (resp., g') proceeds by

assuming, toward the contradiction, that g (resp., g') can be efficiently inverted with unallowable success probability. Contradiction is derived by deducing that f can be efficiently inverted with unallowable success probability. In other words, inverting f is “reduced” to inverting g (resp., g'). The term “reduction” is used here in a stronger-than-standard sense. Here a reduction needs to preserve the success probability of the algorithms. This kind of argument is called a *reducibility argument*.

Proof: We first prove that g and g' can be computed in polynomial time. To this end we use the fact that I is a polynomial-time-enumerable set, which implies that we can decide membership in I in polynomial time (e.g., by observing that $m \in I$ if and only if $s_I(m-1) = m$). It follows that on input x one can find in polynomial time the largest $m \leq |x|$ that satisfies $m \in I$. Computing $g(x)$ amounts to finding this m and applying the function f to the m -bit prefix of x . Similarly for g' .

We next prove that g maintains the hardness-to-invert property of f . A similar proof establishes the hardness-to-invert property of g' . For the sake of brevity, we present here only the proof for the case that f is strongly one-way. The proof for the case that f is weakly one-way is analogous.

The proof proceeds by contradiction. We assume, contrary to the claim (of the proposition), that there exists an efficient algorithm that inverts g with success probability that is not negligible. We use this inverting algorithm (for g) to construct an efficient algorithm that inverts f with success probability that is not negligible, hence deriving a contradiction (to the hypothesis of the proposition). In other words, we show that inverting f (with unallowable success probability) is efficiently reducible to inverting g (with unallowable success probability) and hence conclude that the latter is not feasible. The reduction is based on the observation that inverting g on images of arbitrary lengths yields inverting g also on images of lengths in I , and that on such lengths g collides with f .

Intuitively, any algorithm inverting g can be used to invert f as follows. On input $(y, 1^n)$, where y is supposedly in the image of $f(U_n) = g(U_m)$ for any $m \in \{n, \dots, s_I(n) - 1\}$, we can invoke the g -inverter on input $(y, 1^m)$ and output the longest prefix with length in I of the string that the g -inverter returns (e.g., if the g -inverter returns an m -bit-long string, then we output its n -bit-long prefix). Thus, our success probability in inverting f on $f(U_n)$ equals the success probability of the g -inverter on $g(U_m)$. The question is which $m \in \{n, \dots, s_I(n) - 1\}$ we should use, and the answer is to try them all (capitalizing on the fact that $s_I(n) = \text{poly}(n)$). Note that the integers are partitioned to intervals of the form $[n, \dots, s_I(n) - 1]$, each associated with a single $n \in I$. Thus, the success probability of any g -inverter on infinitely many lengths $m \in \mathbb{N}$ translates to the success probability of our f -inverter on infinitely many lengths $n \in I$. Details follow.

Given an algorithm B' for inverting g , we construct an algorithm A' for inverting f such that A' has complexity and success probability related to those for B' . (For simplicity, we shall assume that $B'(y, 1^m) \in \{0, 1\}^m$ holds for all $y \in \{0, 1\}^*$ and $m \in \mathbb{N}$; this assumption is immaterial, and later we comment about this aspect in two footnotes.) Algorithm A' uses algorithm B' as a subroutine and proceeds

as follows. On input y and 1^n (supposedly y is in the range of $f(U_n)$, and $n \in I$), algorithm A' proceeds as follows:

1. It computes $s_I(n)$ and sets $k \stackrel{\text{def}}{=} s_I(n) - n - 1 \geq 0$. (Thus, for every $i = 1, \dots, k$, we have $n + i \notin I$.)
2. For $i = 0, 1, \dots, k$, algorithm A' invokes algorithm B' on input $(y, 1^{n+i})$, obtaining $z_i \leftarrow B'(y, 1^{n+i})$; if $g(z_i) = y$, then A' outputs the n -bit-long prefix² of z_i .

Note that for all $x' \in \{0, 1\}^n$ and $|x''| \leq k$, we have $g(x'x'') = f(x')$, and so if $g(x'x'') = y$, then $f(x') = y$, which establishes the correctness of the output of A' . Using $s_I(n) = \text{poly}(n)$ and the fact that $s_I(n)$ is computable in polynomial time, it follows that if B' is a probabilistic polynomial-time algorithm, then so is A' . We next analyze the success probability of A' (showing that if B' inverts g with unallowable success probability, then A' inverts f with unallowable success probability).

Suppose that B' inverts g on $(g(U_m), 1^m)$ with probability $\varepsilon(m)$. Then there exists an n such that $m \in \{n, \dots, \text{poly}(n)\}$ and such that $A'(f(U_n), 1^n)$ invokes B' on input $(f(U_n), 1^m) = (g(U_m), 1^m)$. It follows that $A'(f(U_n), 1^n)$ inverts f with probability at least $\varepsilon(m) = \varepsilon(\text{poly}(n))$. Thus, $A'(f(U_n), 1^n)$ inherits the success of $B'(g(U_m), 1^m)$. A tedious analysis (which can be skipped) follows.³

Suppose, contrary to our claim, that g is not strongly one-way, and let B' be an algorithm demonstrating this contradiction hypothesis. Namely, there exists a polynomial $p(\cdot)$ such that for infinitely many m 's the probability that B' inverts g on $g(U_m)$ is at least $\frac{1}{p(m)}$. Let us denote the set of these m 's by M . Define a function $\ell_I : \mathbb{N} \rightarrow I$ such that $\ell_I(m)$ is the largest lower bound of m in I is both (i.e., $\ell_I(m) \stackrel{\text{def}}{=} \max\{i \in I : i \leq m\}$). Clearly, $m \geq \ell_I(m)$ and $m \leq s_I(\ell_I(m)) - 1$ for every m . The following two claims relate the success probability of algorithm A' with that of algorithm B' .

Claim 2.2.3.1: Let m be an integer and $n = \ell_I(m)$. Then

$$\Pr[A'(f(U_n), 1^n) \in f^{-1}(f(U_n))] \geq \Pr[B'(g(U_m), 1^m) \in g^{-1}(g(U_m))]$$

(Namely, the success probability of algorithm A' on $f(U_{\ell_I(m)})$ is bounded below by the success probability of algorithm B' on $g(U_m)$.)

Proof: By construction of A' , on input $(f(x'), 1^n)$, where $x' \in \{0, 1\}^n$, algorithm A' obtains the value $B'(f(x'), 1^t)$ for every $t \in \{n, \dots, s_I(n) - 1\}$. In particular, since $m \geq n$ and $m \leq s_I(\ell_I(m)) - 1 = s_I(n) - 1$, it follows that algorithm A' obtains the value $B'(f(x'), 1^m)$. By definition of g , for all $x'' \in \{0, 1\}^{m-n}$, it holds that $f(x') = g(x'x'')$. The claim follows. \square

Claim 2.2.3.2: There exists a polynomial $q(\cdot)$ such that $m < q(\ell_I(m))$ for all m 's.

²Here we use the assumption $z_i \in \{0, 1\}^{n+i}$, which implies that n is the largest integer that both is in I and is at most $n + i$. In general, A' outputs the longest prefix x' of z_i satisfying $|x'| \in I$. Note that it holds that $f(x') = g(z_i) = y$.

³The reader can verify that the following analysis does not refer to the length of the output of B' and so does not depend on the simplifying assumption made earlier.

Proof: Let q be a polynomial (as guaranteed by the polynomial-time enumerability of I) such that $s_I(n) < q(n)$. Then, for every m , we have $m < s_I(\ell_I(m)) < q(\ell_I(m))$. \square

By Claim 2.2.3.2, the set $S \stackrel{\text{def}}{=} \{\ell_I(m) : m \in M\}$ is infinite (as, otherwise, for u upper-bounding the elements in S we get $m < q(\ell_I(m)) \leq q(u)$ for every $m \in M$, which contradicts the hypothesis that M is infinite). Using Claim 2.2.3.1, it follows that for every $n = \ell_I(m) \in S$, the probability that A' inverts f on $f(U_n)$ is at least

$$\frac{1}{p(m)} > \frac{1}{p(q(\ell_I(m)))} = \frac{1}{p(q(n))} = \frac{1}{\text{poly}(n)}$$

where the inequality is due to Claim 2.2.3.2. It follows that f is not strongly one-way, in contradiction to the proposition's hypothesis. \blacksquare

2.2.3.2. Length-Regular and Length-Preserving Functions

A second useful convention regarding one-way functions is to assume that the function f is *length-regular* in the sense that for every $x, y \in \{0, 1\}^*$, if $|x| = |y|$, then $|f(x)| = |f(y)|$. We point out that the transformation presented earlier (i.e., both Eq. (2.1) and Eq. (2.2)) preserves length regularity. A special case of length regularity, preserved by Eq. (2.2), is that of *length-preserving* functions.

Definition 2.2.4 (Length-Preserving Functions): A function f is **length-preserving** if for every $x \in \{0, 1\}^*$ it holds that $|f(x)| = |x|$.

Given a strongly (resp., weakly) one-way function f , we can construct a strongly (resp., weakly) one-way function f'' that is length-preserving, as follows. Let p be a polynomial bounding the length expansion of f (i.e., $|f(x)| \leq p(|x|)$). Such a polynomial must exist because f is polynomial-time-computable. We first construct a length-regular function f' by defining

$$f'(x) \stackrel{\text{def}}{=} f(x)10^{p(|x|)-|f(x)|} \quad (2.3)$$

(We use a padding of the form 10^* in order to facilitate the parsing of $f'(x)$ into $f(x)$ and the “leftover” padding.) Next, we define f'' only on strings of length $p(n) + 1$, for $n \in \mathbb{N}$, by letting

$$f''(x'x'') \stackrel{\text{def}}{=} f'(x'), \text{ where } |x'x''| = p(|x'|) + 1 \quad (2.4)$$

Clearly, f'' is length-preserving.

Proposition 2.2.5: If f is a strongly (resp., weakly) one-way function, then so are f' and f'' (as defined in Eq. (2.3) and Eq. (2.4), respectively).

Proof Sketch: It is quite easy to see that both f' and f'' are polynomial-time-computable. Using “reducibility arguments” analogous to the one used in the preceding proof, we can establish the hardness-to-invert of both f' and f'' . For example, given an algorithm B' for inverting f' , we construct an algorithm A' for

inverting f as follows. On input y and 1^n (supposedly y is in the range of $f(U_n)$), algorithm A' halts with output $B'(y10^{p(n)-|y|}, 1^n)$. ■

On Dropping the Auxiliary Input $1^{|x|}$. The reader can easily verify that if f is length-preserving, then it is redundant to provide the inverting algorithm with the auxiliary input $1^{|x|}$ (in addition to $f(x)$). The same holds if f is length-regular and does not shrink its input by more than a polynomial amount (i.e., there exists a polynomial $p(\cdot)$ such that $p(|f(x)|) \geq |x|$ for all x). In the sequel, *we shall deal only with one-way functions that are length-regular and do not shrink their input by more than a polynomial amount.* Furthermore, we shall mostly deal with length-preserving functions. In all these cases, *we can assume, without loss of generality, that the inverting algorithm is given only $f(x)$ as input.*

On 1-1 One-Way Functions. If f is 1-1, then so is f' (as defined in Eq. (2.3)), but not f'' (as defined in Eq. (2.4)). Thus, when given a 1-1 one-way function, we can assume without loss of generality that it is length-regular, but we cannot assume that it is length-preserving. Furthermore, the assumption that 1-1 one-way functions exist seems stronger than the assumption that arbitrary (and hence length-preserving) one-way functions exist. For further discussion, see Section 2.4.

2.2.4. Candidates for One-Way Functions

Following are several candidates for one-way functions. Clearly, it is not known whether or not these functions are indeed one-way. These are only conjectures supported by extensive research that thus far has failed to produce an efficient inverting algorithm (one having noticeable success probability).

2.2.4.1. Integer Factorization

In spite of the extensive research directed toward the construction of feasible integer-factoring algorithms, the best algorithms known for factoring integers have sub-exponential running times. Hence it is reasonable to believe that the function f_{mult} that partitions its input string into two parts and returns the (binary representation of the) integer resulting by multiplying (the integers represented by) these parts is one-way. Namely, let

$$f_{\text{mult}}(x, y) = x \cdot y$$

where $|x| = |y|$, and $x \cdot y$ denotes (the string representing) the integer resulting by multiplying the integers (represented by the strings) x and y . Clearly, f_{mult} can be computed in polynomial time. Assuming the intractability of factoring (e.g., that given the product of two uniformly chosen n -bit-long primes, it is infeasible to find the prime factors), and using the density-of-primes theorem (which guarantees that at least $\frac{N}{\log_2 N}$ of the integers smaller than N are primes), it follows that f_{mult} is at least weakly one-way. (For further discussion, see Exercise 8.) Other popular functions related to integer factorization (e.g., the RSA function) are discussed in Section 2.4.3.

2.2.4.2. Decoding of Random Linear Codes

One of the most outstanding open problems in the area of error-correcting codes is that of presenting efficient decoding algorithms for random linear codes. Of particular interest are random linear codes with constant information rates that can correct a constant fraction of errors. An (n, k, d) linear code is a k -by- n binary matrix in which the vector sum (mod 2) of any non-empty subset of rows results in a vector with at least d entries of 1 (one-entries). (A k -bit-long message is encoded by multiplying it by the k -by- n matrix, and the resulting n -bit-long vector has a unique pre-image even when flipping up to $\frac{d}{2}$ of its entries.) The Gilbert-Varshamov bound for linear codes guarantees the existence of such a code provided that $\frac{k}{n} < 1 - H_2(\frac{d}{n})$, where $H_2(p) \stackrel{\text{def}}{=} -p \log_2 p - (1-p) \log_2 (1-p)$ if $p < \frac{1}{2}$ and $H_2(p) \stackrel{\text{def}}{=} 1$ otherwise (i.e., $H_2(\cdot)$ is a modification of the binary entropy function). Similarly, if for some $\varepsilon > 0$ it holds that $\frac{k}{n} < 1 - H_2(\frac{(1+\varepsilon)d}{n})$, then almost all k -by- n binary matrices will constitute (n, k, d) linear codes. Consider three constants $\kappa, \delta, \varepsilon > 0$ satisfying $\kappa < 1 - H_2((1+\varepsilon)\delta)$. The function f_{code} seems a plausible candidate for a one-way function:

$$f_{\text{code}}(C, x, i) \stackrel{\text{def}}{=} (C, xC + e(i))$$

where C is a κn -by- n binary matrix, x is a κn -dimensional binary vector, i is the index of an n -dimensional binary vector having at most $\frac{\delta n}{2}$ one-entries within a corresponding enumeration of such vectors (the vector itself is denoted $e(i)$), and the arithmetic is in the n -dimensional binary vector space. Clearly, f_{code} is polynomial-time-computable, provided we use an efficient enumeration of vectors. An efficient algorithm for inverting f_{code} would yield an efficient algorithm for decoding a non-negligible fraction of the constant-rate linear codes (which would constitute an earth-shaking result in coding theory).

2.2.4.3. The Subset-Sum Problem

Consider the function f_{ssum} defined as follows:

$$f_{\text{ssum}}(x_1, \dots, x_n, I) = \left(x_1, \dots, x_n, \sum_{i \in I} x_i \right)$$

where $|x_1| = \dots = |x_n| = n$, and $I \subseteq \{1, 2, \dots, n\}$. Clearly, f_{ssum} is polynomial-time-computable. The fact that the subset-sum problem is \mathcal{NP} -complete cannot serve as evidence to the one-wayness of f_{ssum} . On the other hand, the fact that the subset-sum problem is easy for special cases (such as having “hidden structure” and/or “low density”) does not rule out this proposal. The conjecture that f_{ssum} is one-way is based on the failure of known algorithms to handle random “high-density” instances (i.e., instances in which the length of the elements approximately equals their number, as in the definition of f_{ssum}).

2.2.5. Non-Uniformly One-Way Functions

In the foregoing two definitions of one-way functions the inverting algorithm is a probabilistic polynomial-time algorithm. Stronger versions of both definitions require

that the functions cannot be inverted even by non-uniform families of polynomial-size circuits. We stress that the easy-to-compute condition is still stated in terms of uniform algorithms. For example, the following is a non-uniform version of the definition of strong (length-preserving) one-way functions.

Definition 2.2.6 (Non-Uniformly Strong One-Way Functions): A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is called **non-uniformly one-way** if the following two conditions hold:

1. Easy to compute: There exists a polynomial-time algorithm A such that on input x algorithm A outputs $f(x)$.
2. Hard to invert: For every (even non-uniform) family of polynomial-size circuits $\{C_n\}_{n \in \mathbb{N}}$, every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[C_n(f(U_n)) \in f^{-1}(f(U_n))] < \frac{1}{p(n)}$$

The probability in the second condition is taken only over all the possible values of U_n . We note that any non-uniformly one-way function is one-way (i.e., in the uniform sense).

Proposition 2.2.7: If f is non-uniformly one-way, then it is one-way. That is, if f satisfies Definition 2.2.6, then it also satisfies Definition 2.2.1.

Proof: We convert any (uniform) probabilistic polynomial-time inverting algorithm into a non-uniform family of polynomial-size circuits, without decreasing the success probability. This is in accordance with our meta-theorem (see Section 1.3.3). Details follow.

Let A' be a probabilistic polynomial-time (inverting) algorithm. Let r_n denote a sequence of coin tosses for A' maximizing the success probability of A' (averaged over input $f(U_n)$). Namely, r_n satisfies

$$\Pr[A'_{r_n}(f(U_n)) \in f^{-1}(f(U_n))] \geq \Pr[A'(f(U_n)) \in f^{-1}(f(U_n))]$$

where the first probability is taken only over all possible values of U_n , and the second probability is also over all possible coin tosses for A' . (Recall that $A'_r(y)$ denotes the output of algorithm A' on input y and internal coin tosses r .) The desired circuit C_n incorporates the code of algorithm A' and the sequence r_n (which is of length polynomial in n). ■

We note that, typically, averaging arguments (of the form applied earlier) allow us to convert probabilistic polynomial-time algorithms into non-uniform polynomial-size circuits. Thus, in general, non-uniform notions of security (i.e., robustness against non-uniform polynomial-size circuits) imply uniform notions of security (i.e., robustness against probabilistic polynomial-time algorithms). The converse is not necessarily true. In particular, it is possible that one-way functions exist (in the uniform sense) and yet

there are no non-uniformly one-way functions. However, this situation (i.e., that one-way functions exist *only* in the uniform sense) seems unlikely, and it is widely believed that non-uniformly one-way functions exist. In fact, all candidates mentioned in the preceding subsection are believed to be non-uniformly one-way functions.

2.3. Weak One-Way Functions Imply Strong Ones

We first remark that not every weak one-way function is necessarily a strong one. Consider, for example, a one-way function f (which, without loss of generality, is length-preserving). Modify f into a function g so that $g(p, x) = (p, f(x))$ if p starts with $\log_2 |x|$ zeros, and $g(p, x) = (p, x)$ otherwise, where (in both cases) $|p| = |x|$.⁴ We claim that g is a weak one-way function but not a strong one. Clearly, g cannot be a strong one-way function (because for all but a $\frac{1}{n}$ fraction of the strings of length $2n$ the function g coincides with the identity function). To prove that g is weakly one-way, we use a “reducibility argument.”

Proposition 2.3.1: *Let f be a one-way function (even in the weak sense). Then g , constructed earlier, is a weakly one-way function.*

Proof: Intuitively, inverting g on inputs on which it does *not* coincide with the identity transformation is related to inverting f . Thus, if g is inverted, on inputs of length $2n$, with probability that is noticeably greater than $1 - \frac{1}{n}$, then g must be inverted with noticeable probability on inputs to which g applies f . Therefore, if g is not weakly one-way, then neither is f . The full, straightforward but tedious proof follows.

Given a probabilistic polynomial-time algorithm B' for inverting g , we construct a probabilistic polynomial-time algorithm A' that inverts f with “related” success probability. Following is the description of algorithm A' . On input y , algorithm A' sets $n \stackrel{\text{def}}{=} |y|$ and $l \stackrel{\text{def}}{=} \log_2 n$, selects p' uniformly in $\{0, 1\}^{n-l}$, computes $z \stackrel{\text{def}}{=} B'(0^l p', y)$, and halts with output of the n -bit suffix of z . Let S_{2n} denote the sets of all $2n$ -bit-long strings that start with $\log_2 n$ zeros (i.e., $S_{2n} \stackrel{\text{def}}{=} \{0^{\log_2 n} \alpha : \alpha \in \{0, 1\}^{2n - \log_2 n}\}$). Then, by construction of A' and g , we have

$$\begin{aligned}
 & \Pr[A'(f(U_n)) \in f^{-1}(f(U_n))] \\
 & \geq \Pr[B'(0^l U_{n-l}, f(U_n)) \in (0^l U_{n-l}, f^{-1}(f(U_n)))] \\
 & = \Pr[B'(g(U_{2n})) \in g^{-1}(g(U_{2n})) \mid U_{2n} \in S_{2n}] \\
 & \geq \frac{\Pr[B'(g(U_{2n})) \in g^{-1}(g(U_{2n}))] - \Pr[U_{2n} \notin S_{2n}]}{\Pr[U_{2n} \in S_{2n}]} \\
 & = n \cdot \left(\Pr[B'(g(U_{2n})) \in g^{-1}(g(U_{2n}))] - \left(1 - \frac{1}{n}\right) \right) \\
 & = 1 - n \cdot (1 - \Pr[B'(g(U_{2n})) \in g^{-1}(g(U_{2n}))])
 \end{aligned}$$

⁴Throughout the text, we treat $\log_2 |x|$ as if it were an integer. A precise argument can be derived by replacing $\log_2 |x|$ with $\lfloor \log_2 |x| \rfloor$ and some minor adjustments.

(For the second inequality, we used $\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]}$ and $\Pr[A \cap B] \geq \Pr[A] - \Pr[\neg B]$.) It should not come as a surprise that the above expression is meaningful only in case $\Pr[B'(g(U_{2n})) \in g^{-1}(g(U_{2n}))] > 1 - \frac{1}{n}$.

It follows that for every polynomial $p(\cdot)$ and every integer n , if B' inverts g on $g(U_{2n})$ with probability greater than $1 - \frac{1}{p(2n)}$, then A' inverts f on $f(U_n)$ with probability greater than $1 - \frac{n}{p(2n)}$. Hence, if g is not weakly one-way (i.e., for every polynomial $p(\cdot)$ there exist infinitely many m 's such that g can be inverted on $g(U_m)$ with probability $\geq 1 - 1/p(m)$), then also f is not weakly one-way (i.e., for every polynomial $q(\cdot)$ there exist infinitely many n 's such that f can be inverted on $f(U_n)$ with probability $\geq 1 - 1/q(n)$, where $q(n) = p(2n)/n$). This contradicts our hypothesis (that f is weakly one-way).

To summarize, given a probabilistic polynomial-time algorithm that inverts g on $g(U_{2n})$ with success probability $1 - \frac{1}{n} + \alpha(n)$, we obtain a probabilistic polynomial-time algorithm that inverts f on $f(U_n)$ with success probability $n \cdot \alpha(n)$. Thus, since f is (weakly) one-way, $n \cdot \alpha(n) < 1 - (1/q(n))$ must hold for some polynomial q , and so g must be weakly one-way (since each probabilistic polynomial-time algorithm trying to invert g on $g(U_{2n})$ must fail with probability at least $\frac{1}{n} - \alpha(n) > \frac{1}{n \cdot q(n)}$). ■

We have just shown that unless no one-way functions exist, there exist weak one-way functions that are not strong ones. This rules out the possibility that all one-way functions are strong ones. Fortunately, we can also rule out the possibility that all one-way functions are (only) weak ones. In particular, the existence of weak one-way functions implies the existence of strong ones.

Theorem 2.3.2: *Weak one-way functions exist if and only if strong one-way functions exist.*

We strongly recommend that the reader not skip the proof (given in Section 2.3.1), since we believe that the proof is very instructive to the rest of this book. Furthermore, the proof demonstrates that amplification of computational difficulty is much more involved than amplification of an analogous probabilistic event. Both aspects are further discussed in Section 2.3.3. An illustration of the proof in the context of a “toy” example is provided in Section 2.3.2. (It is possible to read Section 2.3.2 before Section 2.3.1; in fact, most readers may prefer to do so.)

2.3.1. Proof of Theorem 2.3.2

Let f be a weak one-way function, and let p be the polynomial guaranteed by the definition of a weak one-way function. Namely, every probabilistic polynomial-time algorithm fails to invert f on $f(U_n)$ with probability at least $\frac{1}{p(n)}$. We assume, for simplicity, that f is length-preserving (i.e. $|f(x)| = |x|$ for all x 's). This assumption, which is not really essential, is justified by Proposition 2.2.5. We define a function g

as follows:

$$g(x_1, \dots, x_{t(n)}) \stackrel{\text{def}}{=} f(x_1), \dots, f(x_{t(n)}) \quad (2.5)$$

where $|x_1| = \dots = |x_{t(n)}| = n$ and $t(n) \stackrel{\text{def}}{=} n \cdot p(n)$. Namely, the $n^2 p(n)$ -bit-long input of g is partitioned into $t(n)$ blocks, each of length n , and f is applied to each block.

Clearly, g can be computed in polynomial time (by an algorithm that breaks the input into blocks and applies f to each block). Furthermore, it is easy to see that inverting g on $g(x_1, \dots, x_{t(n)})$ requires finding a pre-image to each $f(x_i)$. One may be tempted to deduce that it is also clear that g is a strongly one-way function. A naive argument might proceed by assuming implicitly (with no justification) that the inverting algorithm worked separately on each $f(x_i)$. If that were indeed the case, then the probability that an inverting algorithm could successfully invert all $f(x_i)$ would be at most $(1 - \frac{1}{p(n)})^{n \cdot p(n)} < 2^{-n}$ (which is negligible also as a function of $n^2 p(n)$). However, the assumption that an algorithm trying to invert g works independently on each $f(x_i)$ cannot be justified. Hence, a more complex argument is required.

Following is an outline of our proof. The proof that g is strongly one-way proceeds by a contradiction argument. We assume, on the contrary, that g is not strongly one-way; namely, we assume that there exists a polynomial-time algorithm that inverts g with probability that is not negligible. We derive a contradiction by presenting a polynomial-time algorithm that, for infinitely many n 's, inverts f on $f(U_n)$ with probability greater than $1 - \frac{1}{p(n)}$ (in contradiction to our hypothesis). The inverting algorithm for f uses the inverting algorithm for g as a subroutine (without assuming anything about the manner in which the latter algorithm operates). (We stress that we do not assume that the g -inverter works in a particular way, but rather use any g -inverter to construct, in a generic way, an f -inverter.) Details follow.

Suppose that g is not strongly one-way. By definition, it follows that there exists a probabilistic polynomial-time algorithm B' and a polynomial $q(\cdot)$ such that for infinitely many m 's,

$$\Pr[B'(g(U_m)) \in g^{-1}(g(U_m))] > \frac{1}{q(m)} \quad (2.6)$$

Let us denote by M' the infinite set of integers for which this holds. Let N' denote the infinite set of n 's for which $n^2 \cdot p(n) \in M'$ (note that all m 's considered are of the form $n^2 \cdot p(n)$, for some integer n).

Using B' , we now present a probabilistic polynomial-time algorithm A' for inverting f . On input y (supposedly in the range of f), algorithm A' proceeds by applying the following probabilistic procedure, denoted I , on input y for $a(|y|)$ times, where $a(\cdot)$ is a polynomial that depends on the polynomials p and q (specifically, we set $a(n) \stackrel{\text{def}}{=} 2n^2 \cdot p(n) \cdot q(n^2 p(n))$).

Procedure I

Input: y (denote $n \stackrel{\text{def}}{=} |y|$).

For $i = 1$ to $t(n)$ do begin

1. Select uniformly and independently a sequence of strings $x_1, \dots, x_{t(n)} \in \{0, 1\}^n$.
 2. Compute $(z_1, \dots, z_{t(n)}) \leftarrow B'(f(x_1), \dots, f(x_{i-1}), y, f(x_{i+1}), \dots, f(x_{t(n)}))$.
(Note that y is placed in the i th position instead of $f(x_i)$.)
 3. If $f(z_i) = y$, then halt and output z_i .
(This is considered a *success*).
- end

Using Eq. (2.6), we now present a lower bound on the success probability of algorithm A' . To this end we define a set, denoted S_n , that contains all n -bit strings on which the procedure I succeeds with non-negligible probability (specifically, greater than $\frac{n}{a(n)}$). (The probability is taken only over the coin tosses of procedure I .) Namely,

$$S_n \stackrel{\text{def}}{=} \left\{ x : \Pr[I(f(x)) \in f^{-1}(f(x))] > \frac{n}{a(n)} \right\}$$

In the next two claims we shall show that S_n contains all but at most a $\frac{1}{2^{p(n)}}$ fraction of the strings of length $n \in N'$ and that for each string $x \in S_n$ the algorithm A' inverts f on $f(x)$ with probability exponentially close to 1. It will follow that A' inverts f on $f(U_n)$, for $n \in N'$, with probability greater than $1 - \frac{1}{p(n)}$, in contradiction to our hypothesis.

Claim 2.3.2.1: For every $x \in S_n$,

$$\Pr[A'(f(x)) \in f^{-1}(f(x))] > 1 - \frac{1}{2^n}$$

Proof: By definition of the set S_n , the procedure I inverts $f(x)$ with probability at least $\frac{n}{a(n)}$. Algorithm A' merely repeats I for $a(n)$ times, and hence

$$\Pr[A'(f(x)) \notin f^{-1}(f(x))] < \left(1 - \frac{n}{a(n)}\right)^{a(n)} < \frac{1}{2^n}$$

The claim follows. \square

Claim 2.3.2.2: For every $n \in N'$,

$$|S_n| > \left(1 - \frac{1}{2^{p(n)}}\right) \cdot 2^n$$

Proof: We assume, to the contrary, that $|S_n| \leq (1 - \frac{1}{2^{p(n)}}) \cdot 2^n$. We shall reach a contradiction to Eq. (2.6) (i.e., our hypothesis concerning the success probability of B'). Recall that by this hypothesis (for $n \in N_0$),

$$s(n) \stackrel{\text{def}}{=} \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)}))] > \frac{1}{q(n^2 p(n))} \quad (2.7)$$

Let $U_n^{(1)}, \dots, U_n^{(n \cdot p(n))}$ denote the n -bit-long blocks in the random variable $U_{n^2 p(n)}$ (i.e., these $U_n^{(i)}$'s are independent random variables each uniformly distributed in $\{0, 1\}^n$). We partition the event considered in Eq. (2.7) into two disjoint events corresponding to whether or not one of the $U_n^{(i)}$'s resides out of S_n . Intuitively, B' cannot perform well in such a case, since this case corresponds to the success

probability of I on pre-images out of S_n . On the other hand, the probability that all $U_n^{(i)}$'s reside in S_n is small. Specifically, we define

$$s_1(n) \stackrel{\text{def}}{=} \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \wedge (\exists i \text{ s.t. } U_n^{(i)} \notin S_n)]$$

and

$$s_2(n) \stackrel{\text{def}}{=} \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \wedge (\forall i : U_n^{(i)} \in S_n)]$$

Clearly, $s(n) = s_1(n) + s_2(n)$ (as the events considered in the s_i 's are disjoint). We derive a contradiction to the lower bound on $s(n)$ (given in Eq. (2.7)) by presenting upper bounds for both $s_1(n)$ and $s_2(n)$ (which sum up to less).

First, we present an upper bound on $s_1(n)$. The key observation is that algorithm I inverts f on input $f(x)$ with probability that is related to the success of B' to invert g on a sequence of random f -images containing $f(x)$. Specifically, for every $x \in \{0, 1\}^n$ and every $1 \leq i \leq n \cdot p(n)$, the probability that I inverts f on $f(x)$ is greater than or equal to the probability that B' inverts g on $g(U_{n^2 p(n)})$ conditioned on $U_n^{(i)} = x$ (since any success of B' to invert g means that f was inverted on the i th block, and thus contributes to the success probability of I). It follows that, for every $x \in \{0, 1\}^n$ and every $1 \leq i \leq n \cdot p(n)$,

$$\begin{aligned} & \Pr[I(f(x)) \in f^{-1}(f(x))] \\ & \geq \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \mid U_n^{(i)} = x] \end{aligned} \quad (2.8)$$

Since for $x \notin S_n$ the left-hand side (l.h.s.) cannot be large, we shall show that (the r.h.s. and so) $s_1(n)$ cannot be large. Specifically, using Eq. (2.8), it follows that

$$\begin{aligned} s_1(n) &= \Pr[\exists i \text{ s.t. } B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \wedge U_n^{(i)} \notin S_n] \\ &\leq \sum_{i=1}^{n \cdot p(n)} \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \wedge U_n^{(i)} \notin S_n] \\ &\leq \sum_{i=1}^{n \cdot p(n)} \sum_{x \notin S_n} \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \wedge U_n^{(i)} = x] \\ &= \sum_{i=1}^{n \cdot p(n)} \sum_{x \notin S_n} \Pr[U_n^{(i)} = x] \cdot \Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \mid U_n^{(i)} = x] \\ &\leq \sum_{i=1}^{n \cdot p(n)} \max_{x \notin S_n} \{\Pr[B'(g(U_{n^2 p(n)})) \in g^{-1}(g(U_{n^2 p(n)})) \mid U_n^{(i)} = x]\} \\ &\leq \sum_{i=1}^{n \cdot p(n)} \max_{x \notin S_n} \{\Pr[I(f(x)) \in f^{-1}(f(x))]\} \\ &\leq n \cdot p(n) \cdot \frac{n}{a(n)} = \frac{n^2 \cdot p(n)}{a(n)} \end{aligned}$$

(The last inequality uses the definition of S_n , and the one before it uses Eq. (2.8).)

We now present an upper bound on $s_2(n)$. Recall that by the contradiction hypothesis, $|S_n| \leq (1 - \frac{1}{2p(n)}) \cdot 2^n$. It follows that

$$\begin{aligned} s_2(n) &\leq \Pr[\forall i : U_n^{(i)} \in S_n] \\ &\leq \left(1 - \frac{1}{2p(n)}\right)^{n \cdot p(n)} \\ &< \frac{1}{2^{n/2}} < \frac{n^2 \cdot p(n)}{a(n)} \end{aligned}$$

(The last inequality holds for sufficiently large n .)

Combining the upper bounds on the s_i 's, we have $s_1(n) + s_2(n) < \frac{2n^2 \cdot p(n)}{a(n)} = \frac{1}{q(n^2 p(n))}$, where equality is by the definition of $a(n)$. Yet, on the other hand, $s_1(n) + s_2(n) = s(n) > \frac{1}{q(n^2 p(n))}$, where the inequality is due to Eq. (2.7). Contradiction is reached, and the claim follows. \square

Combining Claims 2.3.2.1 and 2.3.2.2, we obtain

$$\begin{aligned} &\Pr[A'(f(U_n)) \in f^{-1}(f(U_n))] \\ &\geq \Pr[A'(f(U_n)) \in f^{-1}(f(U_n)) \wedge U_n \in S_n] \\ &= \Pr[U_n \in S_n] \cdot \Pr[A'(f(U_n)) \in f^{-1}(f(U_n)) \mid U_n \in S_n] \\ &\geq \left(1 - \frac{1}{2p(n)}\right) \cdot (1 - 2^{-n}) > 1 - \frac{1}{p(n)} \end{aligned}$$

It follows that there exists a probabilistic polynomial-time algorithm (i.e., A') that inverts f on $f(U_n)$, for $n \in N'$, with probability greater than $1 - \frac{1}{p(n)}$. This conclusion, which follows from the hypothesis that g is not strongly one-way (i.e., Eq. (2.6)), stands in contradiction to the hypothesis that every probabilistic polynomial-time algorithm fails to invert f with probability at least $\frac{1}{p(n)}$, and the theorem follows. \blacksquare

2.3.2. Illustration by a Toy Example

Let us try to further clarify the algorithmic ideas underlying the proof of Theorem 2.3.2. To do so, consider the following quantitative notion of weak one-way functions. We say that (a polynomial-time-computable) f is ρ -one-way if for all probabilistic polynomial-time algorithms A' , for all but finitely many n 's, the probability that on input $f(U_n)$ algorithm A' fails to find a pre-image under f is at least $\rho(n)$. (Each weak one-way function is $1/p()$ -one-way for some polynomial p , whereas strong one-way functions are $(1 - \mu())$ -one-way, where μ is a negligible function.)

Proposition 2.3.3 (Toy Example): Suppose that f is $\frac{1}{3}$ -one-way, and let $g(x_1, x_2) \stackrel{\text{def}}{=} (f(x_1), f(x_2))$. Then g is 0.55-one-way (where $0.55 < 1 - (\frac{2}{3})^2$).

Proof Outline: Suppose, toward the contradiction, that there exists a polynomial-time algorithm A' that inverts $g(U_{2n})$ with success probability greater than

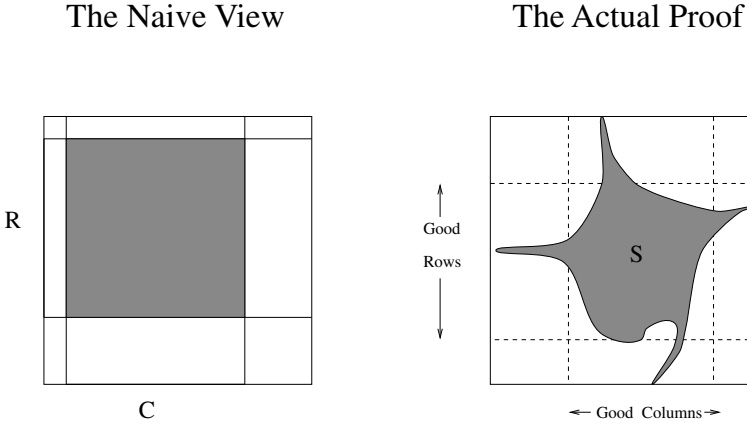


Figure 2.2: The naive view versus the actual proof of Proposition 2.3.3.

$1 - 0.55 = 0.45$, for infinitely many n 's. Consider any such n , and let $N \stackrel{\text{def}}{=} 2^n$. Assume for simplicity that A' is deterministic. Consider an N -by- N matrix with entries corresponding to pairs $(x_1, x_2) \in \{0, 1\}^n \times \{0, 1\}^n$ such that entry (x_1, x_2) is marked 1 if A' successfully inverts g on input $g(x_1, x_2) = (f(x_1), f(x_2))$ and is marked zero otherwise. Our contradiction hypothesis is that the fraction of 1-entries in the matrix is greater than 45%.

The naive (unjustified) assumption is that A' operates separately on each element of the pair $(f(x_1), f(x_2))$. If that were the case, then the success region of A' would have been a generalized rectangle $R \times C \subseteq \{0, 1\}^n \times \{0, 1\}^n$ (i.e., corresponding to all pairs (x_1, x_2) such that $x_1 \in R$ and $x_2 \in C$ for some sets $R \subseteq \{0, 1\}^n$ and $C \subseteq \{0, 1\}^n$). Using the hypothesis that f is $\frac{1}{3}$ -one-way, we have $|R|, |C| \leq \frac{2}{3} \cdot N$, and so $\frac{|R \times C|}{N^2} \leq \frac{4}{9} < 0.45$, in contradiction to our hypothesis regarding A' .

However, as stated earlier, the naive assumption cannot be justified, and so a more complex argument is required. In general, the success region of A' , denoted S , may be an arbitrary subset of $\{0, 1\}^n \times \{0, 1\}^n$ satisfying $|S| > 0.45 \cdot N^2$ (by the contradiction hypothesis). Let us call a row x_1 (resp., column x_2) *good* if it contains at least 0.1% of 1-entries; otherwise it is called *bad*. (See Figure 2.2.) The main algorithmic part of the proof is establishing the following claim.

Claim 2.3.3.1: *The fraction of good rows (resp., columns) is at most 66.8%.*

Once this claim is proved, all that is left is straightforward combinatorics (i.e., counting). That is, we upper-bound the size of S by counting separately the number of 1-entries in the intersection of good rows and good columns and the 1-entries in bad rows and bad columns: By Claim 2.3.3.1, there are at most $(0.668N)^2$ entries in the intersection of good rows and good columns, and by definition the number of 1-entries in each bad row (resp., bad column) is at most $0.001N$. Thus, $|S| \leq (0.668N)^2 + 2 \cdot N \cdot 0.001N < 0.449 \cdot N^2$, in contradiction to our hypothesis (i.e., $|S| > 0.45 \cdot N^2$).

Proof of Claim 2.3.3.1: Suppose, toward the contradiction, that the fraction of good rows is greater than 66.8% (the argument for columns is analogous). Then, to reach a contradiction, we construct an algorithm for inverting f as follows. On input y , the algorithm repeats the following steps 10,000 times:

1. Select x_2 uniformly in $\{0, 1\}^n$.
2. Invoke A' on input $(y, f(x_2))$, and obtain its output (x', x'') .
3. If $f(x') = y$, then halt with output x' .

Clearly, this algorithm works in polynomial time, and it is left to analyze its success in inverting f . For every good x_1 , the probability that the algorithm fails to invert f on input $y = f(x_1)$ is at most $(1 - 0.001)^{10,000} < 0.001$. Thus, the probability that the algorithm succeeds in inverting f on input $f(U_n)$ is at least $0.668 \cdot 0.999 > \frac{2}{3}$, in contradiction to the hypothesis that f is $\frac{1}{3}$ -one-way. \square

2.3.3. Discussion

2.3.3.1. Reducibility Arguments: A Digest

Let us recall the structure of the proof of Theorem 2.3.2. Given a weak one-way function f , we first constructed a polynomial-time-computable function g . This was done with the intention of later proving that g is strongly one-way. To prove that g is strongly one-way, we used a *reducibility argument*. The argument transforms efficient algorithms that supposedly contradict the strong one-wayness of g into efficient algorithms that contradict the hypothesis that f is weakly one-way. Hence g must be strongly one-way. We stress that our algorithmic transformation, which is in fact a randomized Cook reduction,⁵ makes no implicit or explicit assumptions about the structure of the prospective algorithms for inverting g . Assumptions such as the “natural” assumption that the inverter of g works independently on each block cannot be justified (at least not at our current state of understanding of the nature of efficient computations).

We use the term *reducibility argument*, rather than just saying a reduction, so as to emphasize that we do *not* refer here to standard (worst-case-complexity) reductions. Let us clarify the distinction: In both cases we refer to *reducing* the task of solving one problem to the task of solving another problem; that is, we use a procedure solving the second task in order to construct a procedure that solves the first task. However, in standard reductions one assumes that the second task has a perfect procedure solving it on all instances (i.e., on the worst case) and constructs such a procedure for the first task. Thus, the reduction may invoke the given procedure (for the second task) on very “non-typical” instances. This cannot be done in our reducibility arguments. Here, we are given a procedure that solves the second task *with certain probability with respect to a certain distribution*. Thus, in employing a reducibility argument, we cannot invoke this procedure on any instance. Instead, we must consider the probability distribution,

⁵A (randomized) Cook reduction of one computational problem Π_1 to another problem, denoted Π_2 , is a (probabilistic) polynomial-time oracle machine that solves Π_1 , while making queries to oracle Π_2 .

on instances of the second task, induced by our reduction. In many cases the latter distribution equals the distribution to which the hypothesis (regarding solvability of the second task) refers, but other cases can be handled too (e.g., these distributions may be “sufficiently close” for the specific purpose). In any case, a careful analysis of the distribution induced by the reducibility argument is due.

2.3.3.2. The Information-Theoretic Analogue

Theorem 2.3.2 has a natural information-theoretic (or “probabilistic”) analogue that asserts that repeating an experiment that has a noticeable failure probability sufficiently many times will yield some failure with very high probability. The reader is probably convinced at this stage that the proof of Theorem 2.3.2 is much more complex than the proof of the information-theoretic analogue. In the information-theoretic context, the repeated events are independent by definition, whereas in our computational context no such independence (which corresponds to the naive argument given at the beginning of the proof of Theorem 2.3.2) can be guaranteed. Another indication of the difference between the two settings follows. In the information-theoretic setting, the probability that none of the failure events will occur decreases exponentially with the number of repetitions. In contrast, in the computational setting we can reach only an unspecified negligible bound on the inverting probabilities of polynomial-time algorithms. Furthermore, it may be the case that g constructed in the proof of Theorem 2.3.2 can be efficiently inverted on $g(U_{n^2 p(n)})$ with success probability that is sub-exponentially decreasing (e.g., with probability $2^{-(\log_2 n)^3}$), whereas the analogous information-theoretic bound is exponentially decreasing (i.e., e^{-n}).

2.3.3.3. Weak One-Way Functions Versus Strong Ones: A Summary

By Theorem 2.3.2, whenever we assume the existence of one-way functions, there is no need to specify whether we refer to weak or strong ones. That is, as far as the mere existence of one-way function goes, the notions of weak and strong one-way functions are equivalent. However, as far as efficiency considerations are concerned, the two notions are not really equivalent, since the above transformation of weak one-way functions into strong ones is not practical. An alternative transformation, which is much more efficient, does exist for the case of one-way permutations and other specific classes of one-way functions. The interested reader is referred to Section 2.6.

2.4. One-Way Functions: Variations

In this section we discuss several issues concerning one-way functions. In the first subsection we present a function that is (strongly) one-way, provided that one-way functions exist. The construction of this function is of strict abstract interest. In contrast, the issues discussed in the other subsections are of practical importance. First, we present an alternative formulation of one-way functions. This formulation is better suited for describing many natural candidates for one-way functions, and indeed we use it in order to describe some popular candidates for one-way functions. Next, we use this

formulation to present one-way functions with additional properties; specifically, we consider (one-way) trapdoor permutations and claw-free function pairs. We remark that these additional properties are used in several constructions presented in other chapters of this book (e.g., trapdoor permutations are used in the construction of public-key encryption schemes, whereas claw-free permutations are used in the construction of collision-free hashing). We conclude this section with remarks concerning the “art” of proposing candidates for one-way functions.

2.4.1.*Universal One-Way Function

Using the notion of a universal machine and the result of the preceding section, it is possible to prove the existence of a *universal* one-way function; that is, we present a (fixed) function that is one-way, provided that one-way functions exist.

Proposition 2.4.1: *There exists a polynomial-time-computable function that is (strongly) one-way if and only if one-way functions exist.*

Proof Sketch: A key observation is that there exist one-way functions if and only if there exist one-way functions that can be evaluated by a quadratic-time algorithm. (The choice of the specific time bound is immaterial; what is important is that such a specific time bound exists.) This statement is proved using a padding argument. Details follow.

Let f be an arbitrary one-way function, and let $p(\cdot)$ be a polynomial bounding the time complexity of an algorithm for computing f . Define $g(x'x'') \stackrel{\text{def}}{=} f(x')x''$, where $|x'x''| = p(|x'|)$. An algorithm computing g first parses the input into x' and x'' so that $|x'x''| = p(|x'|)$ and then applies f to x' . The parsing and the other overhead operations can be implemented in quadratic time (in $|x'x''|$), whereas computing $f(x')$ is done within time $p(|x'|) = |x'x''|$ (which is linear in the input length). Hence, g can be computed (by a Turing machine) in quadratic time. The reader can verify that g is one-way using a “reducibility argument” (analogous to the one used in the proof of Proposition 2.2.5).

We now present a (universal one-way) function, denoted f_{uni} :

$$f_{\text{uni}}(\text{desc}(M), x) \stackrel{\text{def}}{=} (\text{desc}(M), M(x)) \quad (2.9)$$

where $\text{desc}(M)$ is a description of Turing machine M , and $M(x)$ is defined as the output of M on input x if M runs at most quadratic time on x , and $M(x)$ is defined as x otherwise. (Without loss of generality, we can view any string as the description of some Turing machine.) Clearly, f_{uni} can be computed in polynomial time by a universal machine that uses a step counter. To show that f_{uni} is weakly one-way (provided that one-way functions exist at all), we use a “reducibility argument.”

Assuming that one-way functions exist, and using the foregoing observation, it follows that there exists a one-way function g that is computed in quadratic time. Let M_g be the quadratic-time machine computing g . Clearly, an (efficient)

algorithm inverting f_{uni} on inputs of the form $f_{\text{uni}}(\text{desc}(M_g), U_n)$ with probability $p(n)$ can be easily modified into an (efficient) algorithm inverting g on inputs of the form $g(U_n)$ with probability $p(n)$. As in the proof of Proposition 2.3.1, it follows that an algorithm inverting f_{uni} with probability at least $1 - \varepsilon(n)$ on strings of length $|\text{desc}(M_g)| + n$ yields an algorithm inverting g with probability at least $1 - 2^{|\text{desc}(M_g)|} \cdot \varepsilon(n)$ on strings of length n . (We stress that $|\text{desc}(M_g)|$ is a constant, depending only on g .) Hence, if f_{uni} is not weakly one-way (i.e., the function ε is not noticeable), then also g cannot be (weakly) one-way (i.e., also $2^{|\text{desc}(M_g)|} \cdot \varepsilon$ is not noticeable).

Using Theorem 2.3.2 (to transform the weak one-way function f_{uni} into a strong one), the proposition follows. ■

Discussion. The observation by which it suffices to consider one-way functions that can be evaluated within a specific time bound is crucial to the construction of f_{uni} , the reason being that it is not possible to construct a polynomial-time machine that is universal for the class of all polynomial-time machines (i.e., a polynomial-time machine that can “simulate” all polynomial-time machines). It is, however, possible to construct, for every polynomial $p(\cdot)$, a polynomial-time machine that is universal for the class of machines with running time bounded by $p(\cdot)$.

The impracticality of the construction of f_{uni} stems from the fact that f_{uni} is likely to be hard to invert only on huge input lengths (i.e., lengths allowing the encoding of non-trivial algorithms as required for the evaluation of one-way functions). Furthermore, to obtain a strongly one-way function from f_{uni} , we need to apply the latter on a sequence of more than 2^L inputs, each of length $L + n$, where L is a lower bound on the length of the encoding of potential one-way functions, and n is our actual security parameter.

Still, Proposition 2.4.1 says that, in principle, the question of whether or not one-way functions exist “reduces” to the question of whether or not a specific function is one-way.

2.4.2. One-Way Functions as Collections

The formulation of one-way functions used thus far is suitable for an abstract discussion. However, for describing many natural candidates for one-way functions, the following formulation (although being more cumbersome) is more serviceable. Instead of viewing one-way functions as functions operating on an infinite domain (i.e., $\{0, 1\}^*$), we consider infinite collections of functions each operating on a finite domain. The functions in the collection share a single evaluating algorithm that when given as input a succinct representation of a function and an element in its domain returns the value of the specified function at the given point. The formulation of a collection of functions is also useful for the presentation of trapdoor permutations and claw-free functions (see Sections 2.4.4 and 2.4.5, respectively). We start with the following definition.

Definition 2.4.2 (Collection of Functions): A collection of functions consists of an infinite set of indices, denoted \bar{I} , and a corresponding set of finite functions,

denoted $\{f_i\}_{i \in \bar{I}}$. That is, for each $i \in \bar{I}$, the domain of the function f_i , denoted D_i , is a finite set.

Typically, the set of indices \bar{I} will be a “dense” subset of the set of all strings; that is, the fraction of n -bit-long strings in \bar{I} will be noticeable (i.e., $|\bar{I} \cap \{0, 1\}^n| \geq 2^n / \text{poly}(n)$).

We shall be interested only in collections of functions that can be used in cryptographic applications. As hinted earlier, a necessary condition for using a collection of functions is the existence of an efficient function-evaluating algorithm (denoted F) that on input $i \in \bar{I}$ and $x \in D_i$ returns $f_i(x)$. Yet this condition by itself does not suffice. We need to be able to (randomly) select an index specifying a function over a sufficiently large domain, as well as to be able to (randomly) select an element of the domain (when given the domain’s index). The sampling property of the index set is captured by an efficient algorithm (denoted I) that on input an integer n (presented in unary) randomly selects a $\text{poly}(n)$ -bit-long index specifying a function and its associated domain. (As usual, unary presentation is used so as to conform with the standard association of efficient algorithms with those running in times polynomial in the lengths of their inputs.) The sampling property of the domains is captured by an efficient algorithm (denoted D) that on input an index i randomly selects an element in D_i . The one-way property of the collection is captured by requiring that every efficient algorithm, when given an index of a function and an element in its range, fails to invert the function, except with negligible probability. The probability is taken over the distribution induced by the sampling algorithms I and D . All the preceding is captured by the following definition.

Definition 2.4.3 (Collection of One-Way Functions): A collection of functions $\{f_i : D_i \rightarrow \{0, 1\}^*\}_{i \in \bar{I}}$ is called strongly (resp., weakly) **one-way** if there exist three probabilistic polynomial-time algorithms I , D , and F such that the following two conditions hold:

1. **Easy to sample and compute:** The output distribution of algorithm I on input 1^n is a random variable assigned values in the set $\bar{I} \cap \{0, 1\}^n$. The output distribution of algorithm D on input $i \in \bar{I}$ is a random variable assigned values in D_i . On input $i \in \bar{I}$ and $x \in D_i$, algorithm F always outputs $f_i(x)$.

(Thus, $D_i \subseteq \bigcup_{m \leq \text{poly}(|i|)} \{0, 1\}^m$. Without loss of generality, we can assume that $D_i \subseteq \{0, 1\}^{\text{poly}(|i|)}$. Also without loss of generality, we can assume that algorithm F is deterministic.)

2. **Hard to invert (version for strongly one-way):** For every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n ’s,

$$\Pr[A'(I_n, f_{I_n}(X_n)) \in f_{I_n}^{-1}(f_{I_n}(X_n))] < \frac{1}{p(n)}$$

where I_n is a random variable describing the output distribution of algorithm I on input 1^n , and X_n is a random variable describing the output of algorithm D on input (random variable) I_n .

(The version for weakly one-way collections is analogous.)

We stress that the output of algorithm I on input 1^n is *not* necessarily distributed *uniformly* over $\bar{I} \cap \{0, 1\}^n$. Furthermore, it is not even required that $I(1^n)$ not be entirely concentrated on one single string. Likewise, the output of algorithm D on input i is *not* necessarily distributed *uniformly* over D_i . Yet the hardness-to-invert condition implies that $D(i)$ cannot be mainly concentrated on polynomially many (in $|i|$) strings. We stress that the collection is hard to invert with respect to the distribution induced by the algorithms I and D (in addition to depending, as usual, on the mapping induced by the function itself).

We can describe a collection of one-way functions by indicating the corresponding triplet of algorithms. Hence, we can say that a *triplet of probabilistic polynomial-time algorithms* (I, D, F) constitutes a collection of one-way functions if there exists a collection of functions for which these algorithms satisfy the foregoing two conditions.

Clearly, any collection of one-way functions can be represented as a one-way function, and vice versa (see Exercise 18), yet each formulation has its own advantages. In the sequel, we shall use the formulation of a collection of one-way functions in order to present popular candidates for one-way functions.

Relaxations. To allow a less cumbersome presentation of natural candidates for one-way collections (of functions), we relax Definition 2.4.3 in two ways. First, we allow the index-sampling algorithm to output, on input 1^n , indices of length $p(n)$ rather than n , where $p(\cdot)$ is some polynomial. Second, we allow all algorithms to fail with negligible probability. Most important, we allow the index sampler I to output strings not in \bar{I} so long as the probability that $I(1^n) \notin \bar{I} \cap \{0, 1\}^{p(n)}$ is a negligible function in n . (The same relaxations can be used when discussing trapdoor permutations and claw-free functions.)

Additional Properties: Efficiently Recognizable Indices and Domains. Several additional properties that hold for some candidate collections for one-way functions will be explicitly discussed in subsequent subsections. Here we mention two (useful) additional properties that hold in some candidate collections for one-way functions. The properties are (1) having an efficiently recognizable set of indices and (2) having efficiently recognizable collection of domains; that is, we refer to the existence of an efficient algorithm for deciding membership in \bar{I} and the existence of an efficient algorithm that given $i \in \bar{I}$ and x can determine whether or not $x \in D_i$. Note that for the non-relaxed Definition 2.4.3, the coins used to generate $i \in \bar{I}$ (resp., $x \in D_i$) constitute a certificate (i.e., an \mathcal{NP} -witness) for the corresponding claim; yet this certificate that $i \in \bar{I}$ (resp., $x \in D_i$) may assist in inverting the function f_i (resp., always yielding the pre-image x).

2.4.3. Examples of One-Way Collections

In this section we present several popular collections of one-way functions (e.g., RSA and discrete exponentiation) based on computational number theory.⁶ In the exposition

⁶Obviously these are merely candidate collections for one-way functions; their hardness-to-invert feature either is a (widely believed) conjecture or follows from a (widely believed) conjecture.

that follows, we assume some knowledge of elementary number theory and some familiarity with simple number-theoretic algorithms. Further discussion of the relevant number theoretic material is presented in Appendix A.

2.4.3.1. The RSA Function

The RSA collection of functions has an index set consisting of pairs (N, e) , where N is a product of two $(\frac{1}{2} \cdot \log_2 N)$ -bit primes, denoted P and Q , and e is an integer smaller than N and relatively prime to $(P - 1) \cdot (Q - 1)$. The function of index (N, e) has domain $\{1, \dots, N\}$ and maps the domain element x to $x^e \bmod N$. Using the fact that e is relatively prime to $(P - 1) \cdot (Q - 1)$, it can be shown that the function is in fact a permutation over its domain. Hence, the RSA collection is a collection of *permutations*.

We first substantiate the fact that the RSA collection satisfies the first condition for the definition of a one-way collection (i.e., that it is easy to sample and compute). To this end, we present the triplet of algorithms $(I_{\text{RSA}}, D_{\text{RSA}}, F_{\text{RSA}})$.

On input 1^n , algorithm I_{RSA} selects uniformly two primes, P and Q , such that $2^{n-1} \leq P < Q < 2^n$, and an integer e such that e is relatively prime to $(P - 1) \cdot (Q - 1)$. (Specifically, e is uniformly selected among the admissible possibilities.⁷) Algorithm I_{RSA} terminates with output (N, e) , where $N = P \cdot Q$. For an efficient implementation of I_{RSA} , we need a probabilistic polynomial-time algorithm for generating uniformly (or almost uniformly) distributed primes. For more details concerning the uniform generation of primes, see Appendix A.

As for algorithm D_{RSA} , on input (N, e) it selects (almost) uniformly an element in the set $D_{N,e} \stackrel{\text{def}}{=} \{1, \dots, N\}$. (The exponentially vanishing deviation is due to the fact that we implement an N -way selection via a sequence of unbiased coin tosses.) The output of F_{RSA} , on input $((N, e), x)$, is

$$\text{RSA}_{N,e}(x) \stackrel{\text{def}}{=} x^e \bmod N \quad (2.10)$$

It is not known whether or not factoring N can be reduced to inverting $\text{RSA}_{N,e}$, and in fact this is a well-known open problem. We remark that the best algorithms known for inverting $\text{RSA}_{N,e}$ proceed by (explicitly or implicitly) factoring N . In any case, it is widely believed that the RSA collection is hard to invert.

In the foregoing description, $D_{N,e}$ corresponds to the additive group mod N (and hence will contain N elements). Alternatively, the domain $D_{N,e}$ can be restricted to the elements of the multiplicative group modulo N (and hence will contain $(P - 1) \cdot (Q - 1) \approx N - 2\sqrt{N} \approx N$ elements). A modified domain sampler may work by selecting an element in $\{1, \dots, N\}$ and discarding the unlikely cases in which the selected element is not relatively prime to N . The function $\text{RSA}_{N,e}$ defined earlier induces a permutation on the multiplicative group modulo N . The resulting collection is as hard to invert as the original one. (A proof of this statement is left as an exercise to the reader.) The question of which formulation to prefer seems to be a matter of personal taste.

⁷In some sources, e is set to equal 3. In such a case, the primes $(P$ and $Q)$ are selected so that they are congruent to 2 mod 3. It is not known whether or not the assumption that one variant is one-way implies that the other also is.

2.4.3.2. The Rabin Function

The Rabin collection of functions is defined analogously to the RSA collection, except that the function is squaring modulo N (instead of raising to the e th power mod N). Namely,

$$\text{Rabin}_N(x) \stackrel{\text{def}}{=} x^2 \bmod N \quad (2.11)$$

This function, however, does not induce a permutation on the multiplicative group modulo N , but is rather a 4-to-1 mapping on this group.

It can be shown that extracting square roots modulo N is computationally equivalent to factoring N (i.e., the two tasks are reducible to one another via probabilistic polynomial-time reductions). For details, see Exercise 21. Hence, squaring modulo a composite is a collection of one-way functions if and only if factoring is intractable. We remind the reader that it is generally believed that integer factorization is intractable, and this holds also for the special case in which the integer is a product of two primes of the same length.⁸

2.4.3.3. The Factoring Permutations

For a special subclass of the integers, known by the name of *Blum integers*, the function $\text{Rabin}_N(\cdot)$ defined earlier induces a permutation on the quadratic residues modulo N . We say that r is a *quadratic residue mod N* if there exists an integer x such that $r \equiv x^2 \pmod{N}$. We denote by Q_N the set of quadratic residues in the multiplicative group mod N . For purposes of this paragraph, we say that N is a Blum integer if it is the product of two primes, each congruent to 3 mod 4. It can be shown that when N is a Blum integer, each element in Q_N has a unique square root that is also in Q_N , and it follows that in this case the function $\text{Rabin}_N(\cdot)$ induces a permutation over Q_N . This leads to the introduction of the collection $\text{SQR} \stackrel{\text{def}}{=} (I_{\text{BI}}, D_{\text{QR}}, F_{\text{SQR}})$ of permutations. On input 1^n , algorithm I_{BI} selects uniformly two primes, P and Q , such that $2^{n-1} \leq P < Q < 2^n$ and $P \equiv Q \equiv 3 \pmod{4}$, and outputs $N = P \cdot Q$. On input N , algorithm D_{QR} uniformly selects an element of Q_N by uniformly selecting an element of the multiplicative group modulo N and squaring it mod N . Algorithm F_{SQR} is defined exactly as in the Rabin collection. The resulting collection is one-way, provided that factoring is intractable.

2.4.3.4. Discrete Logarithms

Another computational number-theoretic problem that is widely believed to be intractable is that of extracting discrete logarithms in a finite field (and, in particular, of prime cardinality). The DLP collection of functions, which borrows its name (and its conjectured one-wayness) from the *discrete-logarithm problem*, is defined by the triplet of algorithms $(I_{\text{DLP}}, D_{\text{DLP}}, F_{\text{DLP}})$.

On input 1^n , algorithm I_{DLP} selects uniformly a prime P , such that $2^{n-1} \leq P < 2^n$, and a primitive element G in the multiplicative group modulo P (i.e., a generator

⁸In fact, the latter case is believed to be the hardest.

of this cyclic group), and outputs (P, G) . There exists a probabilistic polynomial-time algorithm for uniformly generating primes, together with the prime factorization of $P - 1$, where P is the prime generated (see Appendix A). Alternatively, one can uniformly generate a prime P of the form $2Q + 1$, where Q is also a prime. (In the latter case, however, one has to assume the intractability of DLP with respect to such primes. We remark that such primes are commonly believed to be the hardest for DLP.) Using the factorization of $P - 1$, we can find a primitive element by selecting an element of the group at random and checking whether or not it has order $P - 1$ (by raising the candidate to powers that non-trivially divide $P - 1$, and comparing the result to 1).

Regarding algorithm D_{DLP} , on input (P, G) it selects uniformly a residue modulo $P - 1$. Algorithm F_{DLP} , on input $((P, G), x)$, outputs

$$\text{DLP}_{P,G}(x) \stackrel{\text{def}}{=} G^x \bmod P \quad (2.12)$$

Hence, inverting $\text{DLP}_{P,G}$ amounts to extracting the discrete logarithm (to base G) modulo P . For every (P, G) of the foregoing form, the function $\text{DLP}_{P,G}$ induces a 1-1 and onto mapping from the additive group mod $P - 1$ to the multiplicative group mod P . Hence, $\text{DLP}_{P,G}$ induces a permutation on the set $\{1, \dots, P - 1\}$.

Exponentiation in other groups is also a reasonable candidate for a one-way function, provided that the discrete-logarithm problem for the group is believed to be hard. For example, it is believed that the logarithm problem is hard in the group of points on an elliptic curve.

2.4.4. Trapdoor One-Way Permutations

We shall define trapdoor (one-way) permutations and review a popular candidate (i.e., the RSA).

2.4.4.1. Definitions

The formulation of collections of one-way functions is convenient as a starting point to the definition of trapdoor permutations. Loosely speaking, these are collections of one-way permutations, $\{f_i\}$, with the extra property that f_i is efficiently inverted once it is given as auxiliary input a “trapdoor” for the index i . The trapdoor for index i , denoted by $t(i)$, *cannot* be efficiently computed from i , yet one can efficiently generate corresponding pairs $(i, t(i))$.

Definition 2.4.4 (Collection of Trapdoor Permutations): *Let $I : 1^* \rightarrow \{0, 1\}^* \times \{0, 1\}^*$ be a probabilistic algorithm, and let $I_1(1^n)$ denote the first element of the pair output by $I(1^n)$. A triple of algorithms, (I, D, F) , is called a **collection of strong (resp., weak) trapdoor permutations** if the following two conditions hold:*

- I. The algorithms induce a collection of one-way permutations: The triple (I_1, D, F) constitutes a collection of strong (resp., weak) one-way permutations.
(Recall that, in particular, $F(i, x) = f_i(x)$.)*

2. Easy to invert with trapdoor: *There exists a (deterministic) polynomial-time algorithm, denoted F^{-1} , such that for every (i, t) in the range of I and for every $x \in D_i$, it holds that $F^{-1}(t, f_i(x)) = x$.*

A useful relaxation of these conditions is to require that they be satisfied with overwhelmingly high probability. Namely, the index-generating algorithm I is allowed to output, with negligible probability, pairs (i, t) for which either f_i is not a permutation or $F^{-1}(t, f_i(x)) = x$ does not hold for all $x \in D_i$. On the other hand, one typically requires that the domain-sampling algorithm (i.e., D) produce an almost uniform distribution on the corresponding domain. Putting all these modifications together, we obtain the following version, which is incomparable to Definition 2.4.4. We take the opportunity to present a slightly different formulation, as well as to introduce a non-uniformly one-way version.

Definition 2.4.5 (Collection of Trapdoor Permutations, Revisited): *Let $\bar{I} \subseteq \{0, 1\}^*$ and $\bar{I}_n \stackrel{\text{def}}{=} \bar{I} \cap \{0, 1\}^n$. A **collection of permutations with indices** in \bar{I} is a set $\{f_i : D_i \rightarrow D_i\}_{i \in \bar{I}}$ such that each f_i is 1-1 on the corresponding D_i . Such a collection is called a **trapdoor permutation** if there exist four probabilistic polynomial-time algorithms I, D, F , and F^{-1} such that the following five conditions hold:*

1. Index and trapdoor selection: *For every n ,*

$$\Pr[I(1^n) \in \bar{I}_n \times \{0, 1\}^*] > 1 - 2^{-n}$$

2. Selection in domain: *For every $n \in \mathbb{N}$ and $i \in \bar{I}_n$,*

(a) $\Pr[D(i) \in D_i] > 1 - 2^{-n}$.

- (b) *Conditioned on $D(i) \in D_i$, the output is uniformly distributed in D_i . That is, for every $x \in D_i$,*

$$\Pr[D(i) = x \mid D(i) \in D_i] = \frac{1}{|D_i|}$$

Thus, $D_i \subseteq \cup_{m \leq \text{poly}(|i|)} \{0, 1\}^m$. Without loss of generality, $D_i \subseteq \{0, 1\}^{\text{poly}(|i|)}$.

3. Efficient evaluation: *For every $n \in \mathbb{N}$, $i \in \bar{I}_n$, and $x \in D_i$,*

$$\Pr[F(i, x) = f_i(x)] > 1 - 2^{-n}$$

4. Hard to invert: *Let I_n be a random variable describing the distribution of the first element in the output of $I(1^n)$, and $X_n \stackrel{\text{def}}{=} D(I_n)$. We consider two versions:*

Standard/uniform-complexity version: For every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[A'(I_n, f_{I_n}(X_n)) = X_n] < \frac{1}{p(n)}$$

Non-uniform-complexity version: For every family of polynomial-size circuits $\{C_n\}_{n \in \mathbb{N}}$, every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[C_n(I_n, f_{I_n}(X_n)) = X_n] < \frac{1}{p(n)}$$

5. Inverting with trapdoor: *For every $n \in \mathbb{N}$, any pair (i, t) in the range of $I(1^n)$ such that $i \in \tilde{I}_n$, and every $x \in D_i$,*

$$\Pr[F^{-1}(t, f_i(x)) = x] > 1 - 2^{-n}$$

We comment that an exponentially vanishing measure of indices for which any of Items 2, 3, and 5 does not hold can be omitted from \tilde{I} (and accounted for by the error allowed in Item 1). Items 3 and 5 can be relaxed by taking the probabilities also over all possible $x \in D_i$ with uniform distribution.

2.4.4.2. The RSA (and Factoring) Trapdoor

The RSA collection presented earlier can be easily modified to have the trapdoor property. To this end, algorithm I_{RSA} should be modified so that it outputs both the index (N, e) and the trapdoor (N, d) , where d is the multiplicative inverse of e modulo $(P - 1) \cdot (Q - 1)$ (note that e has such inverse because it has been chosen to be relatively prime to $(P - 1) \cdot (Q - 1)$). The inverting algorithm F_{RSA}^{-1} is identical to the algorithm F_{RSA} (i.e., $F_{\text{RSA}}^{-1}((N, d), y) = y^d \bmod N$). The reader can easily verify that

$$F_{\text{RSA}}^{-1}((N, d), F_{\text{RSA}}((N, e), x)) = x^{ed} \bmod N$$

indeed equals x , for every x in the multiplicative group modulo N . In fact, one can show that $x^{ed} \equiv x \pmod{N}$ for every x (even in case x is not relatively prime to N).

The Rabin collection presented earlier can be easily modified in a similar manner, enabling one to efficiently compute all four square roots of a given quadratic residue (mod N). The trapdoor in this case is the prime factorization of N . The square roots mod N can be computed by extracting a square root modulo each of the prime factors of N and combining the results using the Chinese Remainder Theorem. Efficient algorithms for extracting square roots modulo a given prime are known (see Appendix A). Furthermore, in case the prime P is congruent to 3 mod 4, the square roots of x mod P can be computed by raising x to the power $\frac{P+1}{4}$ (while reducing the intermediate results mod P). Furthermore, in case N is a Blum integer, the collection SQR, presented earlier, forms a collection of trapdoor permutations (provided, of course, that factoring is hard).

2.4.5.* Claw-Free Functions

The formulation of collections of one-way functions is also a convenient starting point for the definition of a collection of claw-free pairs of functions.

2.4.5.1. The Definition

Loosely speaking, a claw-free collection consists of a set of pairs of functions that are easy to evaluate, that have the same range for both members of each pair, and yet for which it is infeasible to find a range element together with a pre-image of it under each of these functions.

Definition 2.4.6 (Claw-Free Collection): A collection of pairs of functions consists of an infinite set of indices, denoted \bar{I} , two finite sets D_i^0 and D_i^1 for each $i \in \bar{I}$, and two functions f_i^0 and f_i^1 defined over D_i^0 and D_i^1 , respectively. Such a collection is called **claw-free** if there exist three probabilistic polynomial-time algorithms I , D , and F such that the following conditions hold:

1. Easy to sample and compute: The random variable $I(1^n)$ is assigned values in the set $\bar{I} \cap \{0, 1\}^n$. For each $i \in \bar{I}$ and $\sigma \in \{0, 1\}$, the random variable $D(\sigma, i)$ is distributed over D_i^σ , and $F(\sigma, i, x) = f_i^\sigma(x)$ for each $x \in D_i^\sigma$.
2. Identical range distribution: For every i in the index set \bar{I} , the random variables $f_i^0(D(0, i))$ and $f_i^1(D(1, i))$ are identically distributed.
3. Hard to form claws: A pair (x, y) satisfying $f_i^0(x) = f_i^1(y)$ is called a claw for index i . Let C_i denote the set of claws for index i . It is required that for every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[A'(I_n) \in C_{I_n}] < \frac{1}{p(n)}$$

where I_n is a random variable describing the output distribution of algorithm I on input 1^n .

The first requirement in Definition 2.4.6 is analogous to what appears in Definition 2.4.3. The other two requirements in Definition 2.4.6 are conflicting in nature. On one hand, it is required that claws do exist (to say the least), whereas on the other hand it is required that claws cannot be efficiently found. Clearly, a claw-free collection of functions yields a collection of strong one-way functions (see Exercise 22). A case of special interest arises when the two domains are identical (i.e., $D_i \stackrel{\text{def}}{=} D_i^0 = D_i^1$), the random variable $D(\sigma, i)$ is uniformly distributed over D_i , and the functions f_i^0 and f_i^1 are permutations over D_i . Such a collection is called a *collection of claw-free pairs of permutations*.

Again, a useful relaxation of the conditions of Definition 2.4.6 is obtained by allowing the algorithms (i.e., I , D , and F) to fail with negligible probability. An additional property that a (claw-free) collection may (or may not) have is an efficiently recognizable index set (i.e., a probabilistic polynomial-time algorithm for determining whether or not a given string is in \bar{I}).

2.4.5.2. The DLP Claw-Free Collection

We now seek to show that claw-free collections do exist under specific reasonable intractability assumptions. We start by presenting such a collection under the assumption that the discrete-logarithm problem (DLP) for fields of prime cardinality is intractable.

Following is the description of a collection of claw-free pairs of *permutations* (based on the foregoing assumption). The index set consists of triples, (P, G, Z) , where P is a prime, G is a primitive element mod P , and Z is an element in the field (of residues mod P). The index-sampling algorithm selects P and G as in the DLP collection presented in Section 2.4.3, and Z is selected uniformly among the residues mod P . The domain is the same for both functions with index (P, G, Z) and equals the set $\{1, \dots, P-1\}$,

and the domain-sampling algorithm selects uniformly from this set. As for the functions themselves, we set

$$f_{P,G,Z}^\sigma(x) \stackrel{\text{def}}{=} Z^\sigma \cdot G^x \bmod P \quad (2.13)$$

The reader can easily verify that both functions are permutations over $\{1, \dots, P-1\}$. In fact, the function $f_{P,G,Z}^0$ coincides with the function $\text{DLP}_{P,G}$ presented in Section 2.4.3. Furthermore, the ability to form a claw for the index (P, G, Z) yields the ability to find the discrete logarithm of $Z \bmod P$ to base G (since $G^x \equiv Z \cdot G^y \pmod{P}$ yields $G^{x-y} \equiv Z \pmod{P}$). Thus, the ability to form claws for a non-negligible fraction of the index set translates to the ability to invert the DLP collection presented in Section 2.4.3. Put in other words, if the DLP collection is one-way, then the collection of pairs of permutations defined in Eq. (2.13) is claw-free.

The foregoing collection *does not* have the additional property of having an efficiently recognizable index set, because it is not known how to efficiently recognize primitive elements modulo a prime. This can be remedied by making a slightly stronger assumption concerning the intractability of DLP. Specifically, we assume that DLP is intractable even if one is given the factorization of the size of the multiplicative group (i.e., the factorization of $P-1$) as additional input. Such an assumption allows one to add the factorization of $P-1$ into the description of the index. This makes the index set efficiently recognizable (since one can test whether or not G is a primitive element by raising it to powers of the form $(P-1)/Q$, where Q is a prime factor of $P-1$). If DLP is hard also for primes of the form $2Q+1$, where Q is also a prime, life is even easier: To test whether or not G is a primitive element mod P , one simply computes $G^2 \bmod P$ and $G^{(P-1)/2} \bmod P$ and checks whether or not both of them are different from 1.

2.4.5.3. Claw-Free Collections Based on Factoring

We now show that a claw-free collection (of functions) does exist under the assumption that integer factorization is infeasible. In the following description, we use the structural properties of Blum integers (i.e., products of two primes both congruent to 3 mod 4), which are further discussed in Appendix A. In particular, for a Blum integer N , it holds that

- the Jacobi symbol of $-1 \bmod N$ equals 1, and
- half of the square roots of each quadratic residue have Jacobi symbol 1.

Let J_N^{+1} (resp., J_N^{-1}) denote the set of residues in the multiplicative group modulo N with Jacobi symbol $+1$ (resp., -1).

The index set of the collection consists of all Blum integers that are composed of two primes of the same length. The index-selecting algorithm, on input 1^n , uniformly selects such an integer by uniformly selecting two (n -bit) primes, each congruent to 3 mod 4, and outputting their product, denoted N . Both functions of index N , denoted f_N^0 and f_N^1 , consist of squaring modulo N , but their corresponding domains are disjoint. The domain of function f_N^σ equals the set $J_N^{(-1)^\sigma}$. The domain-sampling algorithm, denoted D , uniformly selects an element of the corresponding domain in the natural

manner. Specifically, on input (σ, N) , algorithm D uniformly selects polynomially many residues mod N and outputs the first residue with Jacobi symbol $(-1)^\sigma$.

The reader can easily verify that both $f_N^0(D(0, N))$ and $f_N^1(D(1, N))$ are uniformly distributed over the set of quadratic residues mod N . The difficulty of forming claws follows from the fact that a claw yields two residues, $x \in J_N^{+1}$ and $y \in J_N^{-1}$, such that their squares modulo N are equal (i.e., $x^2 \equiv y^2 \pmod{N}$). Since $-1 \in J_N^{+1}$ (and the latter is a multiplicative subgroup), it follows that $y \not\equiv \pm x \pmod{N}$, and so the greatest common divisor (g.c.d.) of $y \pm x$ and N yields a factorization of N .

The foregoing collection consists of pairs of functions that are 2-to-1 (and are defined over disjoint domains). To obtain a collection of claw-free *permutations*, we slightly modify the collection as follows. The index set consists of Blum integers that are the products of two primes P and Q of the same length, so that $P \equiv 3 \pmod{8}$ and $Q \equiv 7 \pmod{8}$. For such composites, neither 2 nor -2 is a quadratic residue modulo $N = P \cdot Q$ (and in fact $\pm 2 \in J_N^{-1}$). Consider the functions f_N^0 and f_N^1 defined over the set, denoted Q_N , of quadratic residues modulo N :

$$f_N^\sigma(x) \stackrel{\text{def}}{=} 4^\sigma \cdot x^2 \pmod{N} \quad (2.14)$$

Clearly, both f_N^0 and f_N^1 are *permutations* over Q_N . The difficulty of forming claws follows from the fact that a claw yields two quadratic residues, x and y , so that $x^2 \equiv 4y^2 \pmod{N}$. Thus, $(x/y)^2 \equiv 4 \pmod{N}$, and so $(2 - (x/y)) \cdot (2 + (x/y)) \equiv 0 \pmod{N}$. Since $\pm 2 \notin Q_N$ (and the latter is a multiplicative subgroup), it follows that $(x/y) \not\equiv \pm 2 \pmod{N}$, and so the g.c.d. of $(2 \pm x \cdot y^{-1} \pmod{N})$ and N yields the factorization of N .

The foregoing collections are *not* known to possess the additional property of having an efficiently recognizable index set. In particular, it is not even known how to efficiently distinguish products of two primes from products of more than two primes.

2.4.6.*On Proposing Candidates

Although we do believe that one-way functions exist, their *mere* existence does not suffice for practical applications. Typically, an application that is based on one-way functions requires the specification of a concrete (candidate one-way) function.⁹ Hence, the problem of proposing reasonable candidates for one-way functions is of great practical importance. Everyone understands that such a reasonable candidate (for a one-way function) should have a very efficient algorithm for evaluating the function. In case the “function” is presented as a collection of one-way functions, the domain sampler and function-evaluation algorithm should be very efficient (whereas for index sampling, “moderate efficiency” may suffice). However, people seem less careful about *seriously considering* the difficulty of inverting the candidates that they propose. We stress that the candidate has to be difficult to invert on “the average” and not only in the worst case, and “the average” is taken with respect to the instance-distribution determined by the candidate function. Furthermore, “hardness on the average” (unlike

⁹As explained in Section 2.4.1, the observation concerning the existence of a universal one-way function is of little practical significance.

worst-case analysis) is extremely sensitive to the instance-distribution. Hence, one has to be extremely careful in deducing average-case complexity with respect to one distribution from the average-case complexity with respect to another distribution. The short history of the field contains several cases in which this point has been ignored, and consequently bad suggestions have been made.

Consider, for example, the following (bad) suggestion to base one-way functions on the conjectured difficulty of the Graph Isomorphism problem. Let $F_{GI}(G, \pi) = (G, \pi G)$, where G is an undirected graph, π is a permutation on its vertex set, and πG denotes the graph resulting by renaming the vertices of G using π (i.e., $(\pi(u), \pi(v))$ is an edge in πG if and only if (u, v) is an edge in G). Although it is indeed believed that Graph Isomorphism cannot be solved in polynomial time, it is easy to see that F_{GI} is easy to invert in most instances (e.g., use vertex-degree statistics to determine the isomorphism). That is, the conjectured worst-case hardness does not imply an average-case hardness for the uniform distribution. Furthermore, even if the problem is hard on the average with respect to *some* distribution, one has to specify this distribution and propose an efficient algorithm for sampling according to it.

2.5. Hard-Core Predicates

Loosely speaking, saying that a function f is one-way implies that given y , it is infeasible to find a pre-image of y under f . This does not mean that it is infeasible to find some partial information about the pre-image of y under f . Specifically, it may be easy to retrieve half of the bits of the pre-image (e.g., given a one-way function f , consider the function g defined by $g(x, r) \stackrel{\text{def}}{=} (f(x), r)$ for every $|x| = |r|$). The fact that one-way functions do not necessarily hide partial information about their pre-images limits their “direct applicability” to tasks such as secure encryption. Fortunately, assuming the existence of one-way functions, it is possible to construct one-way functions that hide specific partial information about their pre-images (which is easy to compute from the pre-image itself). This partial information can be considered as a “hard-core” of the difficulty of inverting f .

2.5.1. Definition

Loosely speaking, a *polynomial-time* predicate b is called a hard-core of a function f if every efficient algorithm, given $f(x)$, can guess $b(x)$ with success probability that is only negligibly better than one-half.

Definition 2.5.1 (Hard-Core Predicate): A polynomial-time-computable predicate $b : \{0, 1\}^* \rightarrow \{0, 1\}$ is called a **hard-core** of a function f if for every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[A'(f(U_n)) = b(U_n)] < \frac{1}{2} + \frac{1}{p(n)}$$

Note that for every $b : \{0, 1\}^* \rightarrow \{0, 1\}$ and $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ there exist obvious algorithms that guess $b(U_n)$ from $f(U_n)$ with success probability at least one-half (e.g., the algorithm that, obviously of its input, outputs a uniformly chosen bit). Also, if b is a hard-core predicate (for any function), then $b(U_n)$ must be almost unbiased (i.e., $|\Pr[b(U_n) = 0] - \Pr[b(U_n) = 1]|$ must be a negligible function in n).

Since b itself is polynomial-time-computable, the failure of efficient algorithms to approximate $b(x)$ from $f(x)$ (with success probability non-negligibly higher than one-half) must be due either to an information loss of f (i.e., f not being one-to-one) or to the difficulty of inverting f . For example, the predicate $b(\sigma\alpha) = \sigma$ is a hard-core of the function $f(\sigma\alpha) \stackrel{\text{def}}{=} 0\alpha$, where $\sigma \in \{0, 1\}$ and $\alpha \in \{0, 1\}^*$. Hence, in this case the fact that b is a hard-core of the function f is due to the fact that f loses information (specifically, the first bit σ). On the other hand, in case f loses no information (i.e., f is one-to-one), hard-cores for f exist only if f is one-way (see Exercise 25). We shall be interested in the case where the hardness of approximating $b(x)$ from $f(x)$ is due to computational reasons and not to information-theoretic ones (i.e., information loss).

Hard-core predicates for collections of one-way functions are defined in an analogous way. Typically, the predicate may depend on the index of the function, and both algorithms (i.e., the one for evaluating it, as well as the one for predicting it based on the function value) are also given this index. That is, a polynomial-time algorithm $B : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}$ is called a *hard-core of the one-way collection* (I, D, F) if for every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$\Pr[A'(I_n, f_{I_n}(X_n)) = B(I_n, X_n)] < \frac{1}{2} + \frac{1}{p(n)}$$

where $I_n \stackrel{\text{def}}{=} I(1^n)$ and $X_n \stackrel{\text{def}}{=} D(I_n)$.

Some Natural Candidates. Simple hard-core predicates are known for the RSA, Rabin, and DLP collections (presented in Section 2.4.3), provided that the corresponding collections are one-way. Specifically, the least significant bit is a hard-core for the RSA collection, provided that the RSA collection is one-way. Namely, assuming that the RSA collection is one-way, it is infeasible to guess (with success probability significantly greater than $\frac{1}{2}$) the least significant bit of x from $\text{RSA}_{N,e}(x) = x^e \bmod N$. Similarly, assuming the intractability of integer factorization, it is infeasible to guess the least significant bit of $x \in Q_N$ from $\text{Rabin}_N(x) = x^2 \bmod N$, where N is a Blum integer (and Q_N denotes the set of quadratic residues modulo N). Finally, assuming that the DLP collection is one-way, it is infeasible to guess whether or not $x < \frac{P}{2}$ when given $\text{DLP}_{P,G}(x) = G^x \bmod P$. In the next subsection we present a general result of this type.

2.5.2. Hard-Core Predicates for Any One-Way Function

Actually, the title is inaccurate: We are going to present hard-core predicates only for (strong) one-way functions of a special form. However, every (strong) one-way function can be easily transformed into a function of the required form, with no substantial loss in either “security” or “efficiency.”

Theorem 2.5.2: *Let f be an arbitrary strong one-way function, and let g be defined by $g(x, r) \stackrel{\text{def}}{=} (f(x), r)$, where $|x| = |r|$. Let $b(x, r)$ denote the inner product mod 2 of the binary vectors x and r . Then the predicate b is a hard-core of the function g .*

In other words, the theorem states that if f is strongly one-way, then it is infeasible to guess the exclusive-OR (XOR) of a random subset of the bits of x when given $f(x)$ and the subset itself. We stress that the theorem requires that f be strongly one-way and that the conclusion is false if f is only weakly one-way (see Exercise 25). Clearly, g is also strongly one-way. We point out that g maintains other properties of f , such as being length-preserving and being one-to-one. Furthermore, an analogous statement holds for collections of one-way functions with/without trapdoor, etc.

The rest of this section is devoted to proving Theorem 2.5.2. Again we use a reducibility argument: Here, inverting the function f is reduced to guessing $b(x, r)$ from $(f(x), r)$. Hence, we assume (for contradiction) the existence of an efficient algorithm guessing the inner product with an advantage that is non-negligible, and we derive an algorithm that inverts f with related (i.e., non-negligible) success probability. This contradicts the hypothesis that f is a one-way function.

We start with some preliminary observations and a motivating discussion and then turn to the main part of the actual proof. We conclude with more efficient implementations of the reducibility argument that assert “higher levels of security.”

2.5.2.1. Preliminaries

Let G be a (probabilistic polynomial-time) algorithm that on input $f(x)$ and r tries to guess the inner product (mod 2) of x and r . Denote by $\varepsilon_G(n)$ the (overall) advantage of algorithm G in guessing $b(x, r)$ from $f(x)$ and r , where x and r are uniformly chosen in $\{0, 1\}^n$. Namely,

$$\varepsilon_G(n) \stackrel{\text{def}}{=} \Pr[G(f(X_n), R_n) = b(X_n, R_n)] - \frac{1}{2} \quad (2.15)$$

where here and in the sequel X_n and R_n denote two independent random variables, each uniformly distributed over $\{0, 1\}^n$. Assuming, to the contrary, that b is not a hard-core of g means that there exists an efficient algorithm G , a polynomial $p(\cdot)$, and an infinite set N such that for every $n \in N$, it holds that $\varepsilon_G(n) > \frac{1}{p(n)}$. We restrict our attention to this algorithm G and to n 's in this set N . In the sequel, we shorthand ε_G by ε .

Our first observation is that on at least an $\frac{\varepsilon(n)}{2}$ fraction of the x 's of length n , algorithm G has at least an $\frac{\varepsilon(n)}{2}$ advantage in guessing $b(x, R_n)$ from $f(x)$ and R_n . Namely:

Claim 2.5.2.1: There exists a set $S_n \subseteq \{0, 1\}^n$ of cardinality at least $\frac{\varepsilon(n)}{2} \cdot 2^n$ such that for every $x \in S_n$, it holds that

$$s(x) \stackrel{\text{def}}{=} \Pr[G(f(x), R_n) = b(x, R_n)] \geq \frac{1}{2} + \frac{\varepsilon(n)}{2}$$

Here the probability is taken over all possible values of R_n and all internal coin tosses of algorithm G , whereas x is fixed.

Proof: The claim follows by an averaging argument. Namely, write $E(s(X_n)) = \frac{1}{2} + \varepsilon(n)$, and apply Markov's inequality. \square

In the sequel, we restrict our attention to x 's in S_n . We shall show an efficient algorithm that on every input y , with $y = f(x)$ and $x \in S_n$, finds x with very high probability. Contradiction to the (strong) one-wayness of f will follow by recalling that $\Pr[U_n \in S_n] \geq \frac{\varepsilon(n)}{2}$.

We start with a motivating discussion. The inverting algorithm that uses algorithm G as subroutine will be formally described and analyzed later.

2.5.2.2. A Motivating Discussion

Consider a fixed $x \in S_n$. By definition, $s(x) \geq \frac{1}{2} + \frac{\varepsilon(n)}{2} > \frac{1}{2} + \frac{1}{2p(n)}$. Suppose, for a moment, that $s(x) > \frac{3}{4} + \frac{1}{2p(n)}$. Of course there is no reason to believe that such is the case; we are just doing a mental experiment. Still, in this case (i.e., of $s(x) > \frac{3}{4} + \frac{1}{2p(n)}$), retrieving x from $f(x)$ is quite easy. To retrieve the i th bit of x , denoted x_i , we randomly select $r \in \{0, 1\}^n$ and compute $G(f(x), r)$ and $G(f(x), r \oplus e^i)$, where e^i is an n -dimensional binary vector with 1 in the i th component, and 0 in all the others, and $v \oplus u$ denotes the addition mod 2 of the binary vectors v and u . (The process is actually repeated polynomially many times, using independent random choices of such r 's, and x_i is determined by a majority vote.)

If both $G(f(x), r) = b(x, r)$ and $G(f(x), r \oplus e^i) = b(x, r \oplus e^i)$, then

$$\begin{aligned} G(f(x), r) \oplus G(f(x), r \oplus e^i) &= b(x, r) \oplus b(x, r \oplus e^i) \\ &= b(x, e^i) \\ &= x_i \end{aligned}$$

where the second equality uses

$$b(x, r) \oplus b(x, s) \equiv \sum_{i=1}^n x_i r_i + \sum_{i=1}^n x_i s_i \equiv \sum_{i=1}^n x_i (r_i + s_i) \equiv b(x, r \oplus s) \pmod{2}$$

The probability that both $G(f(x), r) = b(x, r)$ and $G(f(x), r \oplus e^i) = b(x, r \oplus e^i)$ hold, for a random r , is at least $1 - 2 \cdot (\frac{1}{4} - \frac{1}{2p(n)}) > \frac{1}{2} + \frac{1}{2p(n)}$. Hence, repeating the foregoing procedure sufficiently many times and ruling by majority, we retrieve x_i with very high probability. Similarly, we can retrieve all the bits of x and hence invert f on $f(x)$. However, the entire analysis was conducted under (the unjustifiable) assumption that $s(x) > \frac{3}{4} + \frac{1}{2p(n)}$, whereas we know only that $s(x) > \frac{1}{2} + \frac{1}{2p(n)}$.

The problem with the foregoing procedure is that it doubles the original error probability of algorithm G on inputs of the form $(f(x), \cdot)$. Under the unrealistic assumption that G 's average error on such inputs is non-negligibly smaller than $\frac{1}{4}$, the error-doubling phenomenon raises no problems. However, in general (and even in the special case

where G 's error is exactly $\frac{1}{4}$, the foregoing procedure is unlikely to invert f . Note that the *average* error probability of G (which is averaged over all possible inputs of the form $(f(x), \cdot)$) cannot be decreased by repeating G several times (e.g., G may always answer correctly on $\frac{3}{4}$ of the inputs and always err on the remaining $\frac{1}{4}$). What is required is an *alternative way of using* the algorithm G , a way that does not double the original error probability of G . The key idea is to generate the r 's in a way that requires applying algorithm G only once per each r (and i), instead of twice. Specifically, we shall use algorithm G to obtain a “guess” for $b(x, r \oplus e^i)$ and obtain $b(x, r)$ in a different way. The good news is that the error probability is no longer doubled, since we use G only to get a “guess” of $b(x, r \oplus e^i)$. The bad news is that we still need to know $b(x, r)$, and it is not clear how we can know $b(x, r)$ without applying G . The answer is that we can guess $b(x, r)$ by ourselves. This is fine if we need to guess $b(x, r)$ for only one r (or logarithmically in $|x|$ many r 's), but the problem is that we need to know (and hence guess) the values of $b(x, r)$ for polynomially many r 's. An obvious way of guessing these $b(x, r)$'s yields an exponentially vanishing success probability. Instead, we generate these polynomially many r 's such that, on one hand, they are “sufficiently random,” whereas, on the other hand, we can guess all the $b(x, r)$'s with noticeable success probability. Specifically, generating the r 's in a particular *pairwise-independent* manner will satisfy both (seemingly contradictory) requirements. We stress that in case we are successful (in our guesses for all the $b(x, r)$'s), we can retrieve x with high probability. Hence, we retrieve x with noticeable probability.

A word about the way in which the pairwise-independent r 's are generated (and the corresponding $b(x, r)$'s are guessed) is indeed in order. To generate $m = \text{poly}(n)$ many r 's, we uniformly (and independently) select $l \stackrel{\text{def}}{=} \log_2(m + 1)$ strings in $\{0, 1\}^n$. Let us denote these strings by s^1, \dots, s^l . We then guess $b(x, s^1)$ through $b(x, s^l)$. Let us denote these guesses, which are uniformly (and independently) chosen in $\{0, 1\}$, by σ^1 through σ^l . Hence, the probability that all our guesses for the $b(x, s^i)$'s are correct is $2^{-l} = \frac{1}{\text{poly}(n)}$. The different r 's correspond to the different *non-empty* subsets of $\{1, 2, \dots, l\}$. Specifically, we let $r^J \stackrel{\text{def}}{=} \bigoplus_{j \in J} s^j$. The reader can easily verify that the r^J 's are pairwise independent, and each is uniformly distributed in $\{0, 1\}^n$. The key observation is that

$$b(x, r^J) = b(x, \bigoplus_{j \in J} s^j) = \bigoplus_{j \in J} b(x, s^j)$$

Hence, our guess for the $b(x, r^J)$'s is $\bigoplus_{j \in J} \sigma^j$, and with noticeable probability all our guesses are correct.

2.5.2.3. Back to the Actual Proof

Following is a formal description of the inverting algorithm, denoted A . We assume, for simplicity, that f is length-preserving (yet this assumption is not essential). On input y (supposedly in the range of f), algorithm A sets $n \stackrel{\text{def}}{=} |y|$ and $l \stackrel{\text{def}}{=} \lceil \log_2(2n \cdot p(n)^2 + 1) \rceil$, where $p(\cdot)$ is the polynomial guaranteed earlier (i.e., $\varepsilon(n) > \frac{1}{p(n)}$ for the infinitely many n 's in N). Algorithm A proceeds as follows:

1. It uniformly and independently selects $s^1, \dots, s^l \in \{0, 1\}^n$ and $\sigma^1, \dots, \sigma^l \in \{0, 1\}$.
2. For every non-empty set $J \subseteq \{1, 2, \dots, l\}$, it computes a string $r^J \leftarrow \bigoplus_{j \in J} s^j$ and a bit $\rho^J \leftarrow \bigoplus_{j \in J} \sigma^j$.
3. For every $i \in \{1, \dots, n\}$ and every non-empty $J \subseteq \{1, \dots, l\}$, it computes

$$z_i^J \leftarrow \rho^J \oplus G(y, r^J \oplus e^i).$$
4. For every $i \in \{1, \dots, n\}$, it sets z_i to be the majority of the z_i^J values.
5. It outputs $z = z_1 \cdots z_n$.

Remark: An Alternative Implementation. In an alternative implementation of these ideas, the inverting algorithm tries all possible values for $\sigma^1, \dots, \sigma^l$, computes a string z for each of these 2^l possibilities, and outputs only one of the resulting z 's, with an obvious preference for a string z satisfying $f(z) = y$. For later reference, this alternative algorithm is denoted A' . (See further discussion in the next subsection.)

Following is a detailed analysis of the success probability of algorithm A on inputs of the form $f(x)$, for $x \in S_n$, where $n \in N$. One key observation, which is extensively used, is that for $x, \alpha, \beta \in \{0, 1\}^n$, it holds that

$$b(x, \alpha \oplus \beta) = b(x, \alpha) \oplus b(x, \beta)$$

It follows that $b(x, r^J) = b(x, \bigoplus_{j \in J} s^j) = \bigoplus_{j \in J} b(x, s^j)$. The main part of the analysis is showing that in case the σ^j 's are correct (i.e., $\sigma^j = b(x, s^j)$ for all $j \in \{1, \dots, l\}$), with constant probability, $z_i = x_i$ for all $i \in \{1, \dots, n\}$. This is proved by bounding from below the probability that the majority of the z_i^J 's equal x_i , where $z_i^J = b(x, r^J) \oplus G(f(x), r^J \oplus e^i)$ (due to the hypothesis that $\sigma^j = b(x, s^j)$ for all $j \in \{1, \dots, l\}$).

Claim 2.5.2.2: For every $x \in S_n$ and every $1 \leq i \leq n$,

$$\Pr \left[\left| \{J : b(x, r^J) \oplus G(f(x), r^J \oplus e^i) = x_i\} \right| > \frac{1}{2} \cdot (2^l - 1) \right] > 1 - \frac{1}{2^n}$$

where $r^J \stackrel{\text{def}}{=} \bigoplus_{j \in J} s^j$ and the s^j 's are independently and uniformly chosen in $\{0, 1\}^n$.

Proof: For every J , define a 0-1 random variable ζ^J such that ζ^J equals 1 if and only if $b(x, r^J) \oplus G(f(x), r^J \oplus e^i) = x_i$. Since $b(x, r^J) \oplus b(x, r^J \oplus e^i) = x_i$, it follows that $\zeta^J = 1$ if and only if $G(f(x), r^J \oplus e^i) = b(x, r^J \oplus e^i)$.

The reader can easily verify that each r^J is uniformly distributed in $\{0, 1\}^n$, and the same holds for each $r^J \oplus e^i$. It follows that each ζ^J equals 1 with probability $s(x)$, which by $x \in S_n$ is at least $\frac{1}{2} + \frac{1}{2p(n)}$. We show that the ζ^J 's are pairwise independent by showing that the r^J 's are pairwise independent. For every $J \neq K$, without loss of generality, there exist $j \in J$ and $k \in K - J$. Hence, for every $\alpha, \beta \in \{0, 1\}^n$, we have

$$\begin{aligned} \Pr[r^K = \beta \mid r^J = \alpha] &= \Pr[s^k = \beta \mid s^j = \alpha] \\ &= \Pr[s^k = \beta] \\ &= \Pr[r^K = \beta] \end{aligned}$$

and pairwise independence of the r^j 's follows. Let $m \stackrel{\text{def}}{=} 2^l - 1$, and let ζ represent a generic ζ^j (which are all identically distributed). Using Chebyshev's inequality (and $m \geq 2n \cdot p(n)^2$), we get

$$\begin{aligned}
 \Pr \left[\sum_j \zeta^j \leq \frac{1}{2} \cdot m \right] &\leq \Pr \left[\left| \sum_j \zeta^j - \left(\frac{1}{2} + \frac{1}{2p(n)} \right) \cdot m \right| \geq \frac{1}{2p(n)} \cdot m \right] \\
 &\leq \frac{m \cdot \text{Var}[\zeta]}{\left(\frac{1}{2p(n)} \cdot m \right)^2} \\
 &= \frac{\text{Var}[\zeta]}{\left(\frac{1}{2p(n)} \right)^2 \cdot (2n \cdot p(n)^2)} \\
 &< \frac{\frac{1}{4}}{\left(\frac{1}{2p(n)} \right)^2 \cdot (2n \cdot p(n)^2)} \\
 &= \frac{1}{2n}
 \end{aligned}$$

The claim follows. \square

Recall that if $\sigma^j = b(x, s^j)$ for all j 's, then $\rho^J = \oplus_{j \in J} \sigma^j = \oplus_{j \in J} b(x, s^j) = b(x, r^J)$ for all non-empty J 's. In this case, with probability at least $\frac{1}{2}$, the string z output by algorithm A equals x . However, the first event (i.e., $\sigma^j = b(x, s^j)$ for all j 's) happens with probability $2^{-l} = \frac{1}{2n \cdot p(n)^2 + 1}$ independently of the events analyzed in Claim 2.5.2.2. Hence, in case $x \in S_n$, algorithm A inverts f on $f(x)$ with probability at least $\frac{1}{2} \cdot 2^{-l} = \frac{1}{4n \cdot p(n)^2 + 2}$ (whereas the alternative algorithm A' succeeds with probability at least $\frac{1}{2}$). Recalling that (by Claim 2.5.2.1) $|S_n| > \frac{1}{2p(n)} \cdot 2^n$, we conclude that for every $n \in N$, algorithm A inverts f on $f(U_n)$ with probability at least $\frac{1}{8n \cdot p(n)^3 + 4p(n)}$. Noting that A is polynomial-time (i.e., it merely invokes G for $2n \cdot p(n)^2 = \text{poly}(n)$ times, in addition to making a polynomial amount of other computations), a contradiction to our hypothesis that f is strongly one-way follows. \blacksquare

2.5.2.4.* More Efficient Reductions

The preceding proof actually establishes the following:

Proposition 2.5.3: *Let G be a probabilistic algorithm with running time $t_G : \mathbb{N} \rightarrow \mathbb{N}$ and advantage $\varepsilon_G : \mathbb{N} \rightarrow [0, 1]$ in guessing b (see Eq. (2.15)). Then there exists an algorithm A that runs in time $O(n^2/\varepsilon_G(n)^2) \cdot t_G(n)$ such that*

$$\Pr[A(f(U_n)) = U_n] \geq \frac{\varepsilon_G(n)}{2} \cdot \frac{\varepsilon_G(n)^2}{4n}$$

The alternative implementation, A' , mentioned earlier (i.e., trying all possible values of the σ^j 's rather than guessing one of them), runs in time $O(n^3/\varepsilon_G(n)^4) \cdot t_G(n)$ and

satisfies

$$\Pr[A'(f(U_n)) = U_n] \geq \frac{\varepsilon_G(n)}{2} \cdot \frac{1}{2}$$

Below, we provide a more efficient implementation of A' . Combining it with a more refined averaging argument than the one used in Claim 2.5.2.1, we obtain the following:

Proposition 2.5.4: *Let $G, t_G : \mathbb{N} \rightarrow \mathbb{N}$, and $\varepsilon_G : \mathbb{N} \rightarrow [0, 1]$ be as before, and define $\ell(n) \stackrel{\text{def}}{=} \log_2(1/\varepsilon_G(n))$. Then there exists an algorithm A'' that runs in expected time $O(n^2 \cdot \ell(n)^3) \cdot t_G(n)$ and satisfies*

$$\Pr[A''(f(U_n)) = U_n] = \Omega(\varepsilon_G(n)^2)$$

Thus, the *time-versus-success ratio* of A'' is $\text{poly}(n)/\varepsilon_G(n)^2$, which (in some sense) is optimal up to a $\text{poly}(n)$ factor; see Exercise 30.

Proof Sketch: Let $\varepsilon(n) \stackrel{\text{def}}{=} \varepsilon_G(n)$, and $\ell \stackrel{\text{def}}{=} \log_2(1/\varepsilon(n))$. Recall that $E[s(X_n)] = 0.5 + \varepsilon(n)$, where $s(x) \stackrel{\text{def}}{=} \Pr[G(f(x), R_n) = b(x, R_n)]$ (as in Claim 2.5.2.1). We first replace Claim 2.5.2.1 by a more refined analysis.

Claim 2.5.4.1: There exists an $i \in \{1, \dots, \ell\}$ and a set $S_n \subseteq \{0, 1\}^n$ of cardinality at least $(2^{i-1} \cdot \varepsilon(n)) \cdot 2^n$ such that for every $x \in S_n$, it holds that

$$s(x) = \Pr[G(f(x), R_n) = b(x, R_n)] \geq \frac{1}{2} + \frac{1}{2^{i+1} \cdot \ell}$$

Proof: Let $A_i \stackrel{\text{def}}{=} \{x : s(x) \geq \frac{1}{2} + \frac{1}{2^{i+1} \cdot \ell}\}$. For any non-empty set $S \subseteq \{0, 1\}^n$, we let $a(S) \stackrel{\text{def}}{=} \max_{x \in S} \{s(x) - 0.5\}$, and $a(\emptyset) \stackrel{\text{def}}{=} 0$. Assuming, to the contrary, that the claim does not hold (i.e., $|A_i| < (2^{i-1} \cdot \varepsilon(n)) \cdot 2^n$ for $i = 1, \dots, \ell$), we get

$$\begin{aligned} E[s(X_n) - 0.5] &\leq \Pr[X_n \in A_1] \cdot a(A_1) \\ &\quad + \sum_{i=2}^{\ell} \Pr[X_n \in (A_i \setminus A_{i-1})] \cdot a(A_i \setminus A_{i-1}) \\ &\quad + \Pr[X_n \in (\{0, 1\}^n \setminus A_{\ell})] \cdot a(\{0, 1\}^n \setminus A_{\ell}) \\ &< \varepsilon(n) \cdot \frac{1}{2} + \sum_{i=2}^{\ell} (2^{i-1} \cdot \varepsilon(n)) \cdot \frac{1}{2^i \ell} + 1 \cdot \frac{1}{2^{\ell+1} \ell} \\ &= \frac{\varepsilon(n)}{2} + (\ell - 1) \cdot \frac{\varepsilon(n)}{2\ell} + \frac{2^{-\ell}}{2\ell} = \varepsilon(n) \end{aligned}$$

which contradicts $E[s(X_n) - 0.5] = \varepsilon(n)$. \square

Fixing any i that satisfies Claim 2.5.4.1, we let $\varepsilon \stackrel{\text{def}}{=} 2^{-i-1}/\ell$ and consider the corresponding set $S_n \stackrel{\text{def}}{=} \{x : s(x) \geq 0.5 + \varepsilon\}$. By suitable setting of parameters, we obtain that for every $x \in S_n$, algorithm A' runs in time $O(n^3/\varepsilon^4) \cdot t_G(n)$ and retrieves x from $f(x)$ with probability at least $\frac{1}{2}$. Our next goal is to provide a

more efficient implementation of A' , specifically, one running in time $O(n^2/\varepsilon^2) \cdot (t_G(n) + \log(n/\varepsilon))$.

The modified algorithm A' is given input $y = f(x)$ and a parameter ε and sets $l = \log((n/\varepsilon^2) + 1)$. In the actual description (presented later), it will be more convenient to use arithmetic of reals instead of Boolean. Hence, we denote $b'(x, r) = (-1)^{b(x,r)}$ and $G'(y, r) = (-1)^{G(y,r)}$. The verification of the following facts is left as an exercise:

Fact 1: For every x , it holds that $E[b'(x, U_n) \cdot G'(f(x), U_n + e^i)] = s'(x) \cdot (-1)^{x_i}$, where $s'(x) \stackrel{\text{def}}{=} 2 \cdot (s(x) - \frac{1}{2})$. (Note that for $x \in S_n$, we have $s'(x) \geq 2\varepsilon$.)

Fact 2: Let R be a uniformly chosen l -by- n Boolean matrix. Then for every $v \neq u \in \{0, 1\}^l \setminus \{0\}^l$, it holds that vR and uR are pairwise independent and uniformly distributed in $\{0, 1\}^n$.

Fact 3: For every $x \in \{0, 1\}^l$ and $v \in \{0, 1\}^l$, it holds that $b'(x, vR) = b'(xR^T, v)$.

Using these facts, we obtain the following:

Claim 2.5.4.2: For any $x \in S_n$ and a uniformly chosen l -by- n Boolean matrix R , there exists $\sigma \in \{0, 1\}^l$ such that, with probability at least $\frac{1}{2}$, for every $1 \leq i \leq n$, the sign of $\sum_{v \in \{0,1\}^l} b'(\sigma, v) \cdot G'(f(x), vR + e^i)$ equals the sign of $(-1)^{x_i}$.

Proof: Let $\sigma = xR^T$. Combining the foregoing facts, for every $v \in \{0, 1\}^l \setminus \{0\}^l$, we have $E[b'(xR^T, v) \cdot G'(f(x), vR + e^i)] = s'(x) \cdot (-1)^{x_i}$. Thus, for every such v , it holds that $\Pr[b'(xR^T, v) \cdot G'(f(x), vR + e^i) = (-1)^{x_i}] = \frac{1+s'(x)}{2} = s(x)$. Using Fact 2, $l = \log((2n/\varepsilon^2) + 1)$, and Chebyshev's inequality, the claim follows. \square

A last piece of notation: Let B be a 2^l -by- 2^l matrix, with the (σ, v) entry being $b'(\sigma, v)$, and let \bar{g}^i be a 2^l -dimensional vector, with the v th entry equal to $G'(f(x), vR + e^i)$. Thus, the σ th entry in the vector $B\bar{g}^i$ equals $\sum_{v \in \{0,1\}^l} b'(\sigma, v) \cdot G'(f(x), vR + e^i)$.

Efficient implementation of algorithm A' : On input $y = f(x)$ and a parameter ε , the inverting algorithm A' sets $l = \log((n/\varepsilon^2) + 1)$ and proceeds as follows:

1. For $i = 1, \dots, n$, it computes the 2^l -dimensional vector \bar{g}^i (as defined earlier).
2. For $i = 1, \dots, n$, it computes $\bar{z}_i \leftarrow B\bar{g}^i$.

Let Z be a 2^l -by- n real matrix in which the i th column equals \bar{z}_i .

Let Z' be a 2^l -by- n Boolean matrix representing the signs of the elements in Z : Specifically, the (i, j) th entry of Z' equals 1 if and only if the (i, j) th entry of Z is negative.

3. Scanning all rows of Z' , it outputs the first row z so that $f(z) = y$.

By Claim 2.5.4.2, for $x \in S_n$, with probability at least $\frac{1}{2}$, the foregoing algorithm retrieves x from $y = f(x)$. The running time of the algorithm is dominated by

Steps 1 and 2, which can be implemented in time $n \cdot 2^l \cdot O(t_G(n)) = O((n/\varepsilon)^2 \cdot t_G(n))$ and $n \cdot O(l \cdot 2^l) = O((n/\varepsilon)^2 \cdot \log(n/\varepsilon))$, respectively.¹⁰

Finally, we define algorithm A'' . On input $y = f(x)$, the algorithm selects $j \in \{1, \dots, \ell\}$ with probability 2^{-2j+1} (and halts with no output otherwise). It invokes the preceding implementation of algorithm A' on input y with parameter $\varepsilon \stackrel{\text{def}}{=} 2^{-j-1}/\ell$ and returns whatever A' does. The *expected* running time of A'' is

$$\sum_{j=1}^{\ell} 2^{-2j+1} \cdot O\left(\frac{n^2}{(2^{-j-1}/\ell)^2}\right) \cdot (t_G(n) + \log(n \cdot 2^{j+1}\ell)) = O(n^2 \cdot \ell^3) \cdot t_G(n)$$

(assuming $t_G(n) = \Omega(\ell \log n)$). Letting $i \leq \ell$ be an index satisfying Claim 2.5.4.1 (and letting S_n be the corresponding set), we consider the case in which j (selected by A'') is greater than or equal to i . By Claim 2.5.4.2, in such a case, and for $x \in S_n$, algorithm A' inverts f on $f(x)$ with probability at least $\frac{1}{2}$. Using $i \leq \ell$ ($= \log_2(1/\varepsilon(n))$), we get

$$\begin{aligned} \Pr[A''(f(U_n)) = U_n] &\geq \Pr[U_n \in S_n] \cdot \Pr[j \geq i] \cdot \frac{1}{2} \\ &\geq 2^{i-1}\varepsilon(n) \cdot 2^{-2i+1} \cdot \frac{1}{2} \\ &\geq \varepsilon(n) \cdot 2^{-\ell} \cdot \frac{1}{2} = \frac{\varepsilon(n)^2}{2} \end{aligned}$$

The proposition follows. ■

Comment. Using an additional trick,¹¹ one can save a factor of $\Theta(n)$ in the running time, resulting in an *expected* running time of $O(n \cdot \log^3(1/\varepsilon_G(n))) \cdot t_G(n)$.

¹⁰Using the special structure of matrix B , one can show that given a vector \bar{w} , the product $B\bar{w}$ can be computed in time $O(l \cdot 2^l)$. Hint: B (known as the Sylvester matrix) can be written recursively as

$$S_k = \begin{pmatrix} S_{k-1} & S_{k-1} \\ S_{k-1} & \bar{S}_{k-1} \end{pmatrix}$$

where $S_0 = +1$ and \bar{M} means flipping the $+1$ entries of M to -1 and vice versa. So

$$\begin{pmatrix} S_{k-1} & S_{k-1} \\ S_{k-1} & \bar{S}_{k-1} \end{pmatrix} \begin{bmatrix} w' \\ w'' \end{bmatrix} = \begin{bmatrix} S_{k-1}w' + S_{k-1}w'' \\ S_{k-1}w' - S_{k-1}w'' \end{bmatrix}$$

Thus, letting $T(k)$ denote the time used in multiplying S_k by a 2^k -dimensional vector, we have $T(k) = 2 \cdot T(k-1) + O(2^k)$, which solves to $T(k) = O(k2^k)$.

¹¹We further modify algorithm A' by setting $2^l = O(1/\varepsilon^2)$ (rather than $2^l = O(n/\varepsilon^2)$). Under the new setting, with constant probability, we recover correctly a constant fraction of the bits of x (rather than all of them). If x were a codeword under an asymptotically good error-correcting code (cf. [138]), this would suffice. To avoid this assumption, we modify algorithm A' so that it tries to recover certain XORs of bits of x (rather than individual bits of x). Specifically, we use an asymptotically good linear code (i.e., having constant rate, correcting a constant fraction of errors, and having efficient decoding algorithm). Thus, the modified A' recovers correctly a constant fraction of the bits in the encoding of x under such a code, and using the decoding algorithm it recovers x .

2.5.3.* Hard-Core Functions

We have just seen that every one-way function can be easily modified to have a hard-core predicate. In other words, the result establishes one bit of information about the pre-image that is hard to approximate from the value of the function. A stronger result may say that several bits of information about the pre-image are hard to approximate. For example, we may want to say that a specific pair of bits is hard to approximate, in the sense that it is infeasible to guess this pair with probability non-negligibly larger than $\frac{1}{4}$. Actually, in general, we take a slightly different approach and require that the true value of these bits be hard to distinguish from a random value. That is, a *polynomial-time* function h is called a hard-core of a function f if no efficient algorithm can distinguish $(f(x), h(x))$ from $(f(x), r)$, where r is a random string of length $|h(x)|$. For further discussion of the notion of efficient distinguishability, the reader is referred to Section 3.2. We assume for simplicity that h is length-regular (see next).

Definition 2.5.5 (Hard-Core Function): Let $h : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be a polynomial-time-computable function satisfying $|h(x)| = |h(y)|$ for all $|x| = |y|$, and let $l(n) \stackrel{\text{def}}{=} |h(1^n)|$. The function h is called a **hard-core** of a function f if for every probabilistic polynomial-time algorithm D' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,

$$|\Pr[D'(f(X_n), h(X_n)) = 1] - \Pr[D'(f(X_n), R_{l(n)}) = 1]| < \frac{1}{p(n)}$$

where X_n and $R_{l(n)}$ are two independent random variables, the first uniformly distributed over $\{0, 1\}^n$ and the second uniformly distributed over $\{0, 1\}^{l(n)}$.

For $l \equiv 1$, Definition 2.5.5 is equivalent to Definition 2.5.1; see the discussion following Lemma 2.5.8. See also Exercise 31.

Simple hard-core functions with logarithmic lengths (i.e., $l(n) = O(\log n)$) are known for the RSA, Rabin, and DLP collections, provided that the corresponding collections are one-way. For example, the function that outputs logarithmically many least significant bits is a hard-core function for the RSA collection, provided that the RSA collection is one-way. Namely, assuming that the RSA collection is one-way, it is infeasible to distinguish, given $\text{RSA}_{N,e}(x) = x^e \bmod N$, the $O(\log |N|)$ least significant bit of x from a uniformly distributed $O(\log |N|)$ -bit-long string. (Similar statements hold for the Rabin and DLP collections.) A general result of this type follows.

Theorem 2.5.6: Let f be an arbitrary strong one-way function, and let g_2 be defined by $g_2(x, s) \stackrel{\text{def}}{=} (f(x), s)$, where $|s| = 2|x|$.¹² Let $b_i(x, s)$ denote the inner product mod 2 of the binary vectors x and $(s_{i+1}, \dots, s_{i+n})$, where $s = (s_1, \dots, s_{2n})$. Then, for any constant $c > 0$, the function $h(x, s) \stackrel{\text{def}}{=} b_1(x, s) \cdots b_{l(|x|)}(x, s)$ is a hard-core of the function g_2 , where $l(n) \stackrel{\text{def}}{=} \min\{n, \lceil c \log_2 n \rceil\}$.

¹²In fact, we can use $|s| = |x| + l(|x|) - 1$, where $l(n) = O(\log n)$. In the current description, s_1 and $s_{n+l(n)+1}, \dots, s_{2n}$ are not used. However, the current formulation makes it unnecessary to specify l when defining g_2 .

The proof of the theorem follows by combining a *proposition that capitalizes on the structure of the specific function h* and a *general lemma concerning hard-core functions*. Loosely speaking, the proposition “reduces” the problem of approximating $b(x, r)$ given $g(x, r)$ to the problem of approximating the XOR of any non-empty set of the bits of $h(x, s)$ given $g_2(x, s)$, where b and g are the hard-core and the one-way function presented in the preceding subsection. Since we know that the predicate $b(x, r)$ cannot be approximated from $g(x, r)$, we conclude that no XOR of the bits of $h(x, s)$ can be approximated from $g_2(x, s)$. The general lemma implies that for every “logarithmically shrinking” function h' (i.e., h' satisfying $|h'(x)| = O(\log |x|)$), the function h' is a hard-core of a function f' if and only if the XOR of any non-empty subset of the bits of h' cannot be approximated from the value of f' . Following are the formal statements and proofs of both claims.

Proposition 2.5.7: *Let f , g_2 , l , and the b_i 's be as in Theorem 2.5.6. Let $\{I_n \subseteq \{1, 2, \dots, l(n)\}\}_{n \in \mathbb{N}}$ be an arbitrary sequence of non-empty sets, and let $b_{I_n}(x, s) \stackrel{\text{def}}{=} \bigoplus_{i \in I_n} b_i(x, s)$. Then for every probabilistic polynomial-time algorithm A' , every positive polynomial $p(\cdot)$, and all sufficiently large n 's,*

$$\Pr[A'(I_n, g_2(U_{3n})) = b_{I_n}(U_{3n})] < \frac{1}{2} + \frac{1}{p(n)}$$

where U_{3n} is a random variable uniformly distributed over $\{0, 1\}^{3n}$.

Proof: The proof is by a reducibility argument. Let X_n , R_n , and S_{2n} be independent random variables uniformly distributed over $\{0, 1\}^n$, $\{0, 1\}^n$, and $\{0, 1\}^{2n}$, respectively. We show that the problem of approximating $b(X_n, R_n)$ given $(f(X_n), R_n)$ is reducible to the problem of approximating $b_{I_n}(X_n, S_{2n})$ given $(f(X_n), S_{2n})$. The underlying observation is that for every $|s| = 2 \cdot |x|$ and every $I \subseteq \{1, \dots, l(n)\}$,

$$b_I(x, s) = \bigoplus_{i \in I} b_i(x, s) = b(x, \bigoplus_{i \in I} \text{sub}_i(s))$$

where $\text{sub}_i(s_1, \dots, s_{2n}) \stackrel{\text{def}}{=} (s_{i+1}, \dots, s_{i+n})$. Furthermore, the reader can verify that for every non-empty $I \subseteq \{1, \dots, l(n)\}$, the random variable $\bigoplus_{i \in I} \text{sub}_i(S_{2n})$ is uniformly distributed over $\{0, 1\}^n$, and that given a string $r \in \{0, 1\}^n$ and such a set I , one can efficiently select a string uniformly in the set $\{s : \bigoplus_{i \in I} \text{sub}_i(s) = r\}$. Verification of both claims is left as an exercise.¹³

Now assume, to the contrary, that there exists an efficient algorithm A' , a polynomial $p(\cdot)$, and an infinite sequence of sets (i.e., I_n 's) and n 's such that

$$\Pr[A'(I_n, g_2(U_{3n})) = b_{I_n}(U_{3n})] \geq \frac{1}{2} + \frac{1}{p(n)}$$

¹³Given any non-empty I and any $r = r_1 \dots r_n \in \{0, 1\}^n$, consider the following procedure, where k is the largest element in I . First, uniformly select $s_1, \dots, s_k, s_{k+n+1}, \dots, s_{2n} \in \{0, 1\}$. Next, going from $i = 1$ to $i = n$, determine s_{k+i} so that $\bigoplus_{j \in I, s_{i+j} = r_i} (i.e., s_{k+i} \leftarrow r_i \oplus (\bigoplus_{j \in I \setminus \{k\}} s_{j+i})$, where the relevant s_{i+j} 's are already determined, since $j < k$). This process determines a string $s_1 \dots s_{2n}$ uniformly among 2^n strings s that satisfy $\bigoplus_{i \in I} \text{sub}_i(s) = r$. Since there are 2^n possible r 's, both claims follow.

We first observe that for n 's satisfying the foregoing inequality we can easily find a set I satisfying

$$p_I \stackrel{\text{def}}{=} \Pr[A'(I, g_2(U_{3n})) = b_I(U_{3n})] \geq \frac{1}{2} + \frac{1}{2p(n)}$$

Specifically, we can try all possible I 's and estimate p_I for each of them (via random experiments), picking an I for which the estimate is highest. (Note that using $\text{poly}(n)$ many experiments, we can approximate each of the possible $2^{l(n)} - 1 = \text{poly}(n)$ different p_I 's up to an additive deviation of $1/4p(n)$ and error probability of 2^{-n} .)

We now present an algorithm for approximating $b(x, r)$ from $y \stackrel{\text{def}}{=} f(x)$ and r . On input y and r , the algorithm first finds a set I as described earlier (this stage depends only on $n \stackrel{\text{def}}{=} |x|$, which equals $|r|$). Once I is found, the algorithm uniformly selects a string s such that $\oplus_{i \in I} \text{sub}_i(s) = r$ and returns $A'(I, (y, s))$.

Note that for uniformly distributed $r \in \{0, 1\}^n$, the string s selected by our algorithm is uniformly distributed in $\{0, 1\}^{2n}$ and $b(x, r) = b_I(x, s)$. Evaluation of the success probability of this algorithm is left as an exercise. ■

The following lemma provides a generic transformation of algorithms distinguishing between $(f(X_n), h(X_n))$ and $(f(X_n), R_{l(n)})$ to algorithms that, given $f(X_n)$ and a random non-empty subset I of $\{1, \dots, l(n)\}$, predict the XOR of the bits of X_n at locations I .

Lemma 2.5.8 (Computational XOR Lemma): *Let f and h be arbitrary length-regular functions, and let $l(n) \stackrel{\text{def}}{=} |h(1^n)|$. Let D be any algorithm, and denote*

$$p \stackrel{\text{def}}{=} \Pr[D(f(X_n), h(X_n)) = 1] \quad \text{and} \quad q \stackrel{\text{def}}{=} \Pr[D(f(X_n), R_{l(n)}) = 1]$$

where X_n and $R_{l(n)}$ are independent random variables uniformly distributed over $\{0, 1\}^n$ and $\{0, 1\}^{l(n)}$, respectively. We consider a specific algorithm, denoted $G \stackrel{\text{def}}{=} G_D$, that uses D as a subroutine. Specifically, on input y , and $S \subseteq \{1, \dots, l(n)\}$ (and $l(n)$), algorithm G selects $r = r_1 \cdots r_{l(n)}$ uniformly in $\{0, 1\}^{l(n)}$ and outputs $D(y, r) \oplus 1 \oplus (\oplus_{i \in S} r_i)$. Then,

$$\Pr[G(f(X_n), I_l, l(n)) = \oplus_{i \in I_l} (h_i(X_n))] = \frac{1}{2} + \frac{p - q}{2^{l(n)} - 1}$$

where I_l is a randomly chosen non-empty subset of $\{1, \dots, l(n)\}$, and $h_i(x)$ denotes the i th bit of $h(x)$.

It follows that for logarithmically shrinking h 's, the existence of an efficient algorithm that distinguishes (with a gap that is not negligible in n) the random variables $(f(X_n), h(X_n))$ and $(f(X_n), R_{l(n)})$ implies the existence of an efficient algorithm that approximates the XOR of a random non-empty subset of the bits of $h(X_n)$ from the value of $f(X_n)$ with an advantage that is not negligible. On the other hand, it is clear that any efficient algorithm that approximates an XOR of a random non-empty subset of the

bits of h from the value of f can be easily modified to distinguish $(f(X_n), h(X_n))$ from $(f(X_n), R_{l(n)})$. Hence, for logarithmically shrinking h 's, the function h is a hard-core of a function f if and only if the XOR of any non-empty subset of the bits of h cannot be approximated from the value of f .

Proof: All that is required is to evaluate the success probability of algorithm G (as a function of $p - q$). We start by fixing an $x \in \{0, 1\}^n$ and evaluating $\Pr[G(f(x), I_l, l) = \oplus_{i \in I_l} (h_i(x))]$, where I_l is a uniformly chosen non-empty subset of $\{1, \dots, l\}$ and $l \stackrel{\text{def}}{=} l(n)$. The rest is an easy averaging (over the x 's).

Let \mathcal{C} denote the set (or class) of all non-empty subsets of $\{1, \dots, l\}$. Define, for every $S \in \mathcal{C}$, a relation \equiv_S such that $y \equiv_S z$ if and only if $\oplus_{i \in S} y_i = \oplus_{i \in S} z_i$, where $y = y_1 \cdots y_l$ and $z = z_1 \cdots z_l$. Note that for every $S \in \mathcal{C}$ and $z \in \{0, 1\}^l$, the relation $y \equiv_S z$ holds for exactly 2^{l-1} of the y 's. Recall that by definition of G , on input $(f(x), S, l)$ and random choice $r = r_1 \cdots r_l \in \{0, 1\}^l$, algorithm G outputs $D(f(x), r) \oplus 1 \oplus (\oplus_{i \in S} r_i)$. The latter equals $\oplus_{i \in S} (h_i(x))$ if and only if one of the following two disjoint events occurs:

event 1: $D(f(x), r) = 1$ and $r \equiv_S h(x)$.

event 2: $D(f(x), r) = 0$ and $r \not\equiv_S h(x)$.

By the preceding discussion and elementary manipulations, we get

$$\begin{aligned} s(x) &\stackrel{\text{def}}{=} \Pr[G(f(x), I_l, l) = \oplus_{i \in I_l} (h_i(x))] \\ &= \frac{1}{|\mathcal{C}|} \cdot \sum_{S \in \mathcal{C}} \Pr[G(f(x), S, l) = \oplus_{i \in S} (h_i(x))] \\ &= \frac{1}{|\mathcal{C}|} \cdot \sum_{S \in \mathcal{C}} (\Pr[\text{event 1}] + \Pr[\text{event 2}]) \\ &= \frac{1}{2 \cdot |\mathcal{C}|} \cdot \sum_{S \in \mathcal{C}} (\Pr[\Delta(R_l) = 1 \mid R_l \equiv_S h(x)] + \Pr[\Delta(R_l) = 0 \mid R_l \not\equiv_S h(x)]) \end{aligned}$$

where R_l is uniformly distributed over $\{0, 1\}^l$ (representing the random choice of algorithm G), and $\Delta(r)$ is shorthand for the random variable $D(f(x), r)$. The rest of the analysis is straightforward but tedious and can be skipped with little loss.

$$\begin{aligned} s(x) &= \frac{1}{2} + \frac{1}{2|\mathcal{C}|} \cdot \sum_{S \in \mathcal{C}} (\Pr[\Delta(R_l) = 1 \mid R_l \equiv_S h(x)] - \Pr[\Delta(R_l) \\ &= 1 \mid R_l \not\equiv_S h(x)]) \\ &= \frac{1}{2} + \frac{1}{2|\mathcal{C}|} \cdot \frac{1}{2^{l-1}} \cdot \left(\sum_{S \in \mathcal{C}} \sum_{r \equiv_S h(x)} \Pr[\Delta(r) = 1] - \sum_{S \in \mathcal{C}} \sum_{r \not\equiv_S h(x)} \Pr[\Delta(r) = 1] \right) \\ &= \frac{1}{2} + \frac{1}{2^l \cdot |\mathcal{C}|} \cdot \left(\sum_r \sum_{S \in \text{EQ}(r, h(x))} \Pr[\Delta(r) = 1] \right. \\ &\quad \left. - \sum_r \sum_{S \in \text{NE}(r, h(x))} \Pr[\Delta(r) = 1] \right) \end{aligned}$$

where $\text{EQ}(r, z) \stackrel{\text{def}}{=} \{S \in \mathcal{C} : r \equiv_S z\}$ and $\text{NE}(r, z) \stackrel{\text{def}}{=} \{S \in \mathcal{C} : r \not\equiv_S z\}$. Observe that for every $r \neq z$, it holds that $|\text{NE}(r, z)| = 2^{l-1}$ (and $|\text{EQ}(r, z)| = 2^{l-1} - 1$). On the other hand, $\text{EQ}(z, z) = \mathcal{C}$ (and $\text{NE}(z, z) = \emptyset$) holds for every z . Hence, we get

$$\begin{aligned} s(x) &= \frac{1}{2} + \frac{1}{2^l |\mathcal{C}|} \sum_{r \neq h(x)} ((2^{l-1} - 1) \cdot \Pr[\Delta(r) = 1] - 2^{l-1} \cdot \Pr[\Delta(r) = 1]) \\ &\quad + \frac{1}{2^l |\mathcal{C}|} \cdot |\mathcal{C}| \cdot \Pr[\Delta(h(x)) = 1] \\ &= \frac{1}{2} - \frac{1}{2^l |\mathcal{C}|} \sum_{r \neq h(x)} \Pr[\Delta(r) = 1] + \left(\frac{1}{|\mathcal{C}|} - \frac{1}{2^l |\mathcal{C}|} \right) \cdot \Pr[\Delta(h(x)) = 1] \end{aligned}$$

where the last equality uses $|\mathcal{C}| = 2^l - 1$ (i.e., $\frac{1}{2^l} = \frac{1}{|\mathcal{C}|} - \frac{1}{2^l |\mathcal{C}|}$). Rearranging the terms and substituting for Δ , we get

$$\begin{aligned} s(x) &= \frac{1}{2} + \frac{1}{|\mathcal{C}|} \cdot \Pr[\Delta(h(x)) = 1] - \frac{1}{2^l |\mathcal{C}|} \sum_r \Pr[\Delta(r) = 1] \\ &= \frac{1}{2} + \frac{1}{|\mathcal{C}|} \cdot (\Pr[D(f(x), h(x)) = 1] - \Pr[D(f(x), R_l) = 1]) \end{aligned}$$

Finally, taking the expectation over the x 's, we get

$$\begin{aligned} \mathbb{E}[s(X_n)] &= \frac{1}{2} + \frac{1}{|\mathcal{C}|} \cdot (\Pr[D(f(X_n), h(X_n)) = 1] - \Pr[D(f(X_n), R_l) = 1]) \\ &= \frac{1}{2} + \frac{1}{2^l - 1} \cdot (p - q) \end{aligned}$$

and the lemma follows. ■

2.6.* Efficient Amplification of One-Way Functions

The *amplification* of weak one-way functions into strong ones, presented in Theorem 2.3.2, has no practical value. Recall that this amplification transforms a function f that is hard to invert on a noticeable fraction (i.e., $\frac{1}{p(n)}$) of the strings of length n into a function g that is hard to invert on all but a negligible fraction of the strings of length $n^2 p(n)$. Specifically, it is shown that an algorithm running in time $T(n)$ that inverts g on a $\varepsilon(n)$ fraction of the strings of length $n^2 p(n)$ yields an algorithm running in time $\text{poly}(p(n), n, \frac{1}{\varepsilon(n)}) \cdot T(n)$ that inverts f on a $1 - \frac{1}{p(n)}$ fraction of the strings of length n . Hence, if f is hard to invert in practice on 1% of the strings of length 1000, then all we can say is that g is hard to invert in practice on almost all strings of length 100,000,000. In contrast, an efficient amplification of one-way functions, as given later, should relate the difficulty of inverting the (weak one-way) function f on strings of length n to the difficulty of inverting the (strong one-way) function g on the strings of length $O(n)$, rather than relating it to the difficulty of inverting the function g on the strings of length $\text{poly}(n)$. Consequently, we may get assertions such as this: If f is

hard to invert in practice on 1% of the strings of length 1000, then g is hard to invert in practice on almost all strings of length 5000. The following definition is natural for a general discussion of amplification of one-way functions.

Definition 2.6.1 (Quantitative One-Wayness): Let $T : \mathbb{N} \rightarrow \mathbb{N}$ and $\varepsilon : \mathbb{N} \rightarrow \mathbb{R}$ be polynomial-time-computable functions. A polynomial-time-computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is called $\varepsilon(\cdot)$ -one-way with respect to time $T(\cdot)$ if for every algorithm A' , with running time bounded by $T(\cdot)$ and all sufficiently large n 's,

$$\Pr[A'(f(U_n)) \neq f^{-1}(f(U_n))] > \varepsilon(n)$$

Using this terminology, we review what we already know about amplification of one-way functions. A function f is weakly one-way if there exists a polynomial $p(\cdot)$ such that f is $\frac{1}{p(\cdot)}$ -one-way with respect to polynomial time.¹⁴ A function f is strongly one-way if for every polynomial $q(\cdot)$, the function f is $(1 - \frac{1}{q(\cdot)})$ -one-way with respect to polynomial time. (The identity function is only 0-one-way with respect to linear time, whereas no function is $(1 - \exp(\cdot))$ -one-way with respect to linear time.¹⁵) The amplification result of Theorem 2.3.2 can be generalized and restated as follows: *If there exist a polynomial p and a (polynomial-time-computable) function f that is $\frac{1}{p(\cdot)}$ -one-way with respect to time $T(\cdot)$, then there exists a (polynomial-time-computable) function g that is strongly one-way with respect to respect to time $T'(\cdot)$, where $T'(n^2 \cdot p(n)) = T(n)$, or, in other words, $T'(n) = T(n^\varepsilon)$ for some $\varepsilon > 0$ satisfying $(n^2 \cdot p(n))^\varepsilon \leq n$.* In contrast, an efficient amplification of one-way functions, as given later, should state that the foregoing holds with respect to $T'(O(n)) = T(n)$ (in other words, $T'(n) = T(\varepsilon \cdot n)$ for some $\varepsilon > 0$). Such a result can be obtained for *regular* one-way functions. A function f is called *regular* if there exists a polynomial-time-computable function $m : \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial $p(\cdot)$ such that for every y in the range of f , the number of pre-images (of length n) of y under f is between $\frac{m(n)}{p(n)}$ and $m(n) \cdot p(n)$. In this book we review the result only for one-way permutations (i.e., length-preserving 1-1 functions).

Theorem 2.6.2 (Efficient Amplification of One-Way Permutations): Let $p(\cdot)$ be a polynomial, and $T : \mathbb{N} \rightarrow \mathbb{N}$ function. Suppose that f is a polynomial-time-computable permutation that is $\frac{1}{p(\cdot)}$ -one-way with respect to time $T(\cdot)$. Then there exists a constant $\gamma > 1$, a polynomial q , and a polynomial-time-computable permutation F such that for every polynomial-time-computable function $\varepsilon : \mathbb{N} \rightarrow [0, 1]$, the function F is $(1 - \varepsilon(\cdot))$ -one-way with respect to time $T'_\varepsilon(\cdot)$, where $T'_\varepsilon(\gamma \cdot n) \stackrel{\text{def}}{=} \frac{\varepsilon(n)^2}{q(n)} \cdot T(n)$.

The constant γ depends only on the polynomial $p(\cdot)$.

¹⁴Here and later, *with respect to polynomial time* means with respect to time T , for every polynomial T .

¹⁵The identity function can be “inverted” with failure probability zero in linear time. On the other hand, for every function f , the algorithm that, given y , outputs $0^{|y|}$ inverts f on $f(U_n)$ with failure probability of at most $1 - 2^{-n} < 1 - \exp(-n)$.