Introduction to Modern Cryptography

Mihir Bellare[1] **Phillip Rogaway**[2]

2005

[1] Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093, USA. mihir@cs.ucsd.edu, http://www-cse.ucsd.edu/users/mihir

[2] Department of Computer Science, Kemper Hall of Engineering, University of California at Davis, Davis, CA 95616, USA; and Department of Computer Science, Faculty of Science, Chiang Mai University, Chiang Mai, 50200 Thailand. rogaway@cs.ucdavis.edu, http://www.cs.ucdavis.edu/∼rogaway

Preface

This is a set of class notes that we have been developing jointly for some years. We use them for cryptography courses that we teach at our respective institutions. Each time one of us teaches the class, he takes the token and updates the notes a bit. The process has resulted in an evolving document that has lots of gaps, as well as plenty of "unharmonized" parts. One day it will, with luck, be complete and cogent.

The viewpoint taken throughout these notes is to emphasize the *theory of cryptography as it can be applied to practice.* This is an approach that the two of us have pursued in our research, and it seems to be a pedagogically desirable approach as well.

We would like to thank the following students of past versions of our courses who have pointed out errors and made suggestions for changes: Andre Barroso, Keith Bell, Kostas Bimpikis, Alexandra Boldyreva, Dustin Boswell, Brian Buesker, Michael Burton, Chris Calabro, Sashka Davis, Alex Gantman, Bradley Huffaker, Hyun Min Kang, Vivek Manpuria, Chanathip Namprempre, Adriana Palacio, Wenjing Rao, Fritz Schneider, Juliana Wong. We welcome further corrections, comments and suggestions.

Mihir Bellare San Diego, California USA
Phillip Rogaway Davis, California USA

Contents

Chapter 1

Introduction

Historically, cryptography arose as a means to enable parties to maintain privacy of the information they send to each other, even in the presence of an adversary with access to the communication channel. While providing privacy remains a central goal, the field has expandeded to encompass many others, including not just other goals of communication security, such as guaranteeing integrity and authenticity of communications, but many more sophisticated and fascinating goals.

Once largely the domain of the military, cryptography is now in widespread use, and you are likely to have used it even if you don't know it. When you shop on the Internet, for example to buy a book at `www.amazon.com`, cryptography is used to ensure privacy of your credit card number as it travels from you to the shop's server. Or, in electronic banking, cryptography is used to ensure that your checks cannot be forged.

Cryptography has been used almost since writing was invented. For the larger part of its history, cryptography remained an art, a game of ad hoc designs and attacks. Although the field retains some of this flavor, the last twenty-five years have brought in something new. The art of cryptography has now been supplemented with a legitimate science. In this course we shall focus on that science, which is modern cryptography.

Modern cryptography is a remarkable discipline. It is a cornerstone of computer and communications security, with end products that are imminently practical. Yet its study touches on branches of mathematics that may have been considered esoteric, and it brings together fields like number theory, computational-complexity theory, and probabiltity theory. This course is your invitation to this fascinating field.

1.1 Goals and settings

Modern cryptography addresses a wide range of problems. But the most basic problem remains the classical one of ensuring security of communication across an insecure medium. To describe it, let's introduce the first two members of our cast of characters: our sender, S, and our receiver, R. (Sometimes people call these characters Alice, A, and Bob, B. Alice and Bob figure in many works on cryptography. But we're going to want the letter A for someone else, anyway.) The sender and receiver want to communicate with each other.

The ideal channel. Imagine our two parties are provided with a dedicated, untappable, impenetrable pipe or tube into which the sender can whisper a message and the receiver will hear

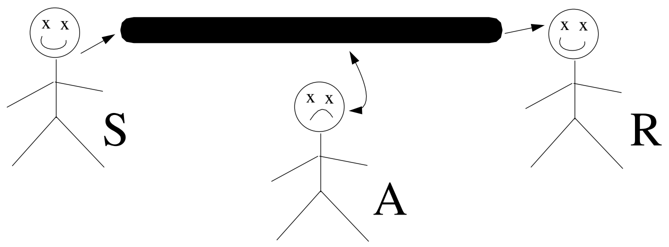

Figure 1.1: Several cryptographic goals aim to imitate some aspect of an ideal channel connecting a sender S to a receiver R.

it. Nobody else can look inside the pipe or change what's there. This pipe provides the perfect medium, available only to the sender and receiver, as though they were alone in the world. It is an "ideal" communication channel from the security point of view. See Fig. 1.1.

Unfortunately, in real life, there are no ideal channels connecting the pairs of parties that might like to communicate with each other. Usually such parties are communicating over some public network like the Internet.

The most basic goal of cryptography is to provide such parties with a means to imbue their communications with security properties akin to those provided by the ideal channel.

At this point we should introduce the third member of our cast. This is our *adversary*, denoted A. An adversary models the source of all possible threats. We imagine the adversary as having access to the network and wanting to compromise the security of the parties communications in some way.

Not all aspects of an ideal channel can be emulated. Instead, cryptographers distill a few central security goals and try to achieve them. The first such goal is *privacy*. Providing privacy means hiding the content of a transmission from the adversary. The second goal is *authenticity* or *integrity*. We want the receiver, upon receiving a communication pertaining to be from the sender, to have a way of assuring itself that it really did originate with the sender, and was not sent by the adversary, or modified en route from the sender to the receiver.

Protocols. In order to achieve security goals such as privacy or authenticity, cryptography supplies the sender and receiver with a *protocol*. A protocol is just a collection of programs (equivalently, algorithms, software), one for each party involved. In our case, there would be some program for the sender to run, and another for the receiver to run. The sender's program tells her how to package, or encapsulate, her data for transmission. The receiver's program tells him how to decapsulate the received package to recover the data together possibly with associated information telling her whether or not to regard it as authentic. Both programs are a function of some *cryptographic keys* as we discuss next.

Trust models. It is not hard to convince yourself that in order to communicate securely, there must be something that a party knows, or can do, that the adversary does not know, or cannot do. There has to be some "asymmetry" between the situation in which the parties finds themselves and situation in which the adversary finds itself.

The *trust model* specifies who, initially, has what keys. There are two central trust models: the symmetric (or shared-key) trust model and the asymmetric (or public-key) trust model. We look at them, and the cryptographic problems they give rise to, in turn.

We will sometimes use words from the theory of "formal languages." Here is the vocabulary you should know.

An *alphabet* is a finite nonempty set. We usually use the Greek letter Σ to denote an alphabet. The elements in an alphabet are called *characters*. So, for example, $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is an alphabet having ten characters, and $\Sigma = \{0, 1\}$ is an alphabet, called the *binary alphabet*, which has two characters. A *string* is finite sequence of characters. The number of characters in a string is called its *length*, and the length of a string X is denoted $|X|$. So $X = 1011$ is a string of length four over the binary alphabet, and $Y = \texttt{cryptography}$ is a string of length 12 over the alphabet of English letters. The string of length zero is called the *empty string* and is denoted ε. If X and Y are strings then the concatenation of X and Y, denoted $X \| Y$, is the characters of X followed by the characters of Y. So, for example, $1011 \| 0 = 10110$. We can encode almost anything into a string. We like to do this because it is as (binary) strings that objects are represented in computers. Usually the details of how one does this are irrelevant, and so we use the notation $\langle \textit{something} \rangle$ for any fixed, natural way to encode *something* as a string. For example, if n is a number and X is a string then $Y = \langle n, X \rangle$ is some string which encodes n and X. It is easy to go from n and X to $Y = \langle n, X \rangle$, and it is also easy to go from $Y = \langle n, X \rangle$ back to n and X. A *language* is a set of strings, all of the strings being drawn from the same alphabet, Σ. If Σ is an alphabet then Σ^* denotes the set of all strings whose characters are drawn from Σ. For example, $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$.

Figure 1.2: Elementary notation from formal-language theory.

1.1.1 The symmetric setting

In practice, the simplest and also most common setting is that the sender and receiver share a *key* that the adversary does not know. This is called the *symmetric setting* or symmetric trust model. The encapsulation and decapsulation procedures above would both depend on this same shared key. The shared key is usually a uniformly distributed random string having some number of bits, k. Recall that a *string* is just a sequence of bits. (For language-theoretic background, see Fig. 1.2.) The sender and receiver must somehow use the key K to overcome the presence of the adversary.

One might ask how the symmetric setting is realized. Meaning, how do a sender and receiver initially come into possession of a key unknown to the adversary? We will discuss this later. The symmetric model is not concerned with how the parties got the key, but with how to use it.

In cryptography we assume that the secret key is kept securely by the party using it. If it is kept on a computer, we assume that the adversary cannot penetrate these machines and recover the key. Ensuring that this assumption is true is the domain of computer systems security.

Let us now take a closer look at some specific problems in the symmetric setting. We'll describe these problems quite informally, but we'll be returning to them later in our studies, when they'll get a much more thorough treatment.

Symmetric encryption schemes. A protocol used to provide privacy in the symmetric setting is called a *symmetric encryption scheme*. When we specify such a scheme Π, we must specify three algorithms, so that the scheme is a triple of algorithms, $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The encapsulation algorithm we discussed above is, in this context, called an *encryption* algorithm, and is the algorithm $\mathcal{E}$. The message M that the sender wishes to transmit is usually referrred to as a *plaintext*. The sender

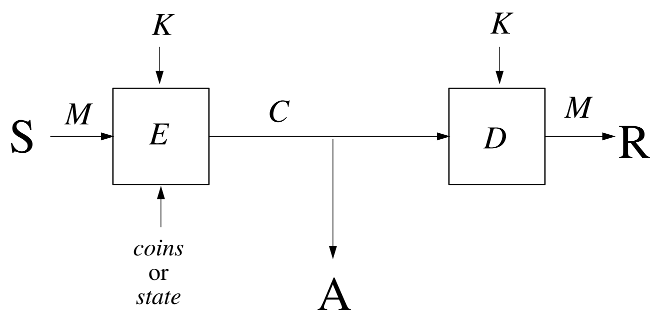

Figure 1.3: Symmetric encryption. The sender and the receiver share a secret key, K. The adversary lacks this key. The message M is the plaintext; the message C is the ciphertext.

encrypts the plaintext under the shared key K by applying $\mathcal{E}$ to K and M to obtain a *ciphertext* C. The ciphertext is transmitted to the receiver. The above-mentioned decapsulation procedure, in this context, is called a *decryption* algorithm, and is the algorithm $\mathcal{D}$. The receiver applies $\mathcal{D}$ to K and C. The decryption process might be unsuccessful, indicated by its returning a special symbol $\perp$, but, if successful, it ought to return the message that was originally encrypted. The first algorithm in Π is the *key generation* algorithm which specifies the manner in which the key is to be chosen. In most cases this algorithm simply returns a random string of length the key length. The encryption algorithm $\mathcal{E}$ may be randomized, or it might keep some state around. A picture for symmetric encryption can be found in Figure 1.3.

The encryption scheme does not tell the adversary what to do. It does not say how the key, once generated, winds its way into the hands of the two parties. And it does not say how messages are transmitted. It only says how keys are generated and how the data is processed.

What is privacy? The goal of a symmetric encryption scheme is that an adversary who obtains the ciphertext be unable to learn anything about the plaintext. What exactly this means, however, is not clear, and obtaining a *definition of privacy* will be an important objective in later chapters.

One thing encryption does not do is hide the length of a plaintext string. This is usually recoverable from the length of the ciphertext string.

As an example of the issues involved in defining privacy, let us ask ourselves whether we could hope to say that it is impossible for the adversary to figure out M given C. But this cannot be true, because the adversary could just guess M, by outputting a random sequence of $|M|$ bits. (As indicated above, the length of the plaintext is usually computable from the length of the ciphertext.) She would be right with probability 2^{-n}. Not bad, if, say $n = 1$! Does that make the scheme bad? No. But it tells us that security is a probabilistic thing. The scheme is not secure or insecure, there is just some probability of breaking it.

Another issue is a priori knowledge. Before M is transmitted, the adversary might know something about it. For example, that M is either 0^n or 1^n. Why? Because she knows Alice and Bob are talking about buying or selling a fixed stock, and this is just a buy or sell message. Now, she can always get the message right with probability 1/2. How is this factored in?

So far one might imagine that an adversary attacking the privacy of an encryption scheme is passive, merely obtaining and examining ciphertexts. In fact, this might not be the case at all. We will consider adversaries that are much more powerful than that.

Message Authenticity. In the message-authentication problem the receiver gets some message which is claimed to have originated with a particular sender. The channel on which this message

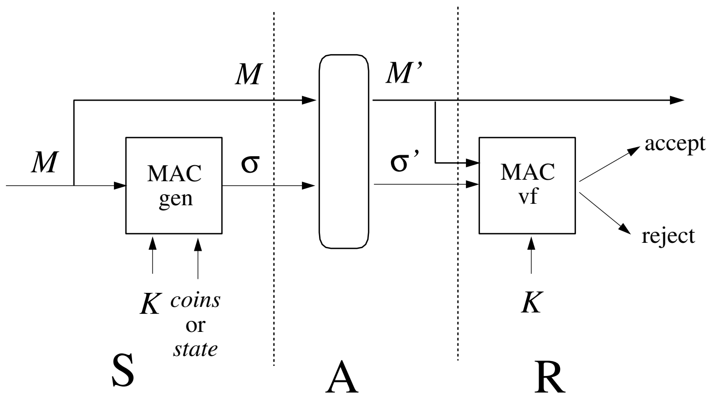

Figure 1.4: A message authentication code. The tag σ accompanies the message M. The receiver R uses it to decide if the message really did originate with the sender S with whom he shares the key K.

flows is insecure. Thus the receiver R wants to distinguish the case in which the message really did originate with the claimed sender S from the case in which the message originated with some imposter, A. In such a case we consider the design of an encapsulation mechanism with the property that un-authentic transmissions lead to the decapsulation algorithm outputting the special symbol $\perp$.

The most common tool for solving the message-authentication problem in the symmetric setting is a *message authentication scheme*, also called a *message authentication code* (MAC). Such a scheme is specified by a triple of algorithms, $\Pi = (\mathcal{K}, \mathcal{T}, \mathcal{V})$. When the sender wants to send a message M to the receiver she computes a "tag," σ, by applying $\mathcal{T}$ to the shared key K and the message M, and then transmits the pair (M, σ). (The encapsulation procedure referred to above thus consists of taking M and returning this pair. The tag is also called a MAC.) The computation of the MAC might be probabilistic or use state, just as with encryption. Or it may well be deterministic. The receiver, on receipt of M and σ, uses the key K to check if the tag is OK by applying the *verification algorithm* $\mathcal{V}$ to K, M and σ. If this algorithms returns 1, he accepts M as authentic; otherwise, he regards M as a forgery. An appropriate reaction might range from ignoring the bogus message to tearing down the connection to alerting a responsible party about the possible mischief. See Figure 1.4.

1.1.2 The asymmetric setting

A shared key K between the sender and the receiver is not the only way to create the information asymmetry that we need between the parties and the adversary. In the *asymmetric setting*, also called the *public-key setting*, a party possesses a *pair* of keys—a *public key*, *pk*, and an associated *secret key, sk*. A party's public key is made publicly known and bound to its identity. For example, a party's public key might be published in a phone book.

The problems that arise are the same as before, but the difference in the setting leads to the development of different kinds of tools.

ASYMMETRIC ENCRYPTION. The sender is assumed to be able to obtain an authentic copy pk_R of the receiver's public key. (The adversary is assumed to know pk_R too.) To send a secret message M to the receiver the sender computes a ciphertext $C \leftarrow \mathcal{E}_{pk_R}(M)$ and sends C to the receiver. When the receiver receives a ciphertext C he computes $M \leftarrow \mathcal{D}_{sk_R}(C)$. The asymmetric encryption

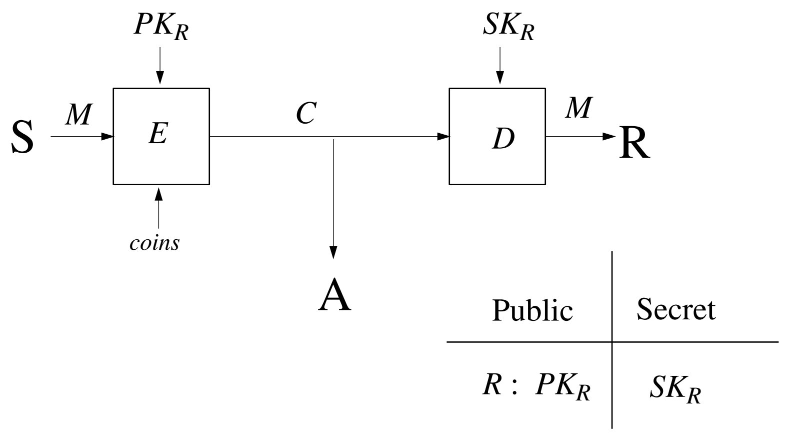

Figure 1.5: Asymmetric encryption. The receiver R has a public key, pk_R, which the sender knows belongs to R. The receiver also has a corresponding secret key, sk_R.

scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ is specified by the algorithms for key generation, encryption and decryption. For a picture of encryption in the public-key setting, see Fig. 1.5.

The idea of public-key cryptography, and the fact that we can actually realize this goal, is remarkable. You've never met the receiver before. But you can send him a secret message by looking up some information in a phone book and then using this information to help you garble up the message you want to send. The intended receiver will be able to understand the content of your message, but nobody else will. The idea of public-key cryptography is due to Whitfield Diffie and Martin Hellman and was published in 1976 [10].

Digital signatures. The tool for solving the message-authentication problem in the asymmetric setting is a *digital signature*. Here the sender has a public key pk_S and a corresponding secret key sk_S. The receiver is assumed to know the key pk_S and that it belongs to party S. (The adversary is assumed to know pk_S too.) When the sender wants to send a message M she attaches to it some extra bits, σ, which is called a *signature* for the message and is computed as a function of M and sk_S by applying to them a *signing* algorithm Sig. The receiver, on receipt of M and σ, checks if it is OK using the public key of the sender, pk_S, by applying a *verification* algorithm $\mathcal{V}$. If this algorithm accepts, the receiver regards M as authentic; otherwise, he regards M as an attempted forgery. The digital signature scheme $\Pi = (\mathcal{K}, \mathrm{Sig}, \mathcal{V})$ is specified by the algorithms for key generation, signing and verifying. A picture is given in Fig. 1.6.

One difference between a MAC and a digital signature concerns what is called *non-repudiation*. With a MAC anyone who can verify a tagged message can also produce one, and so a tagged message would seem to be of little use in proving authenticity in a court of law. But with a digitally-signed message the *only* party who should be able to produce a message that verifies under public key pk_S is the party S herself. Thus if the signature scheme is good, party S cannot just maintain that the receiver, or the one presenting the evidence, concocted it. If signature σ authenticates M with respect to public key pk_S, then it is only S that should have been able to devise σ. The sender cannot refute that. Probably the sender S can claim that the key sk_S was stolen from her. Perhaps this, if true, might still be construed the sender's fault.

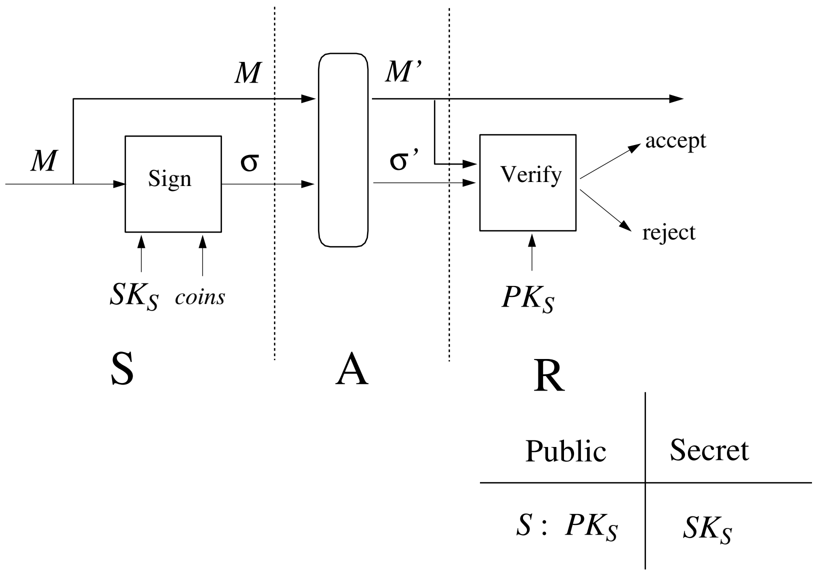

Figure 1.6: A digital signature scheme. The signature σ accompanies the message M. The receiver R uses it to decide if the message really did originate with the sender S with has public key pk_S.

	symmetric trust model	**asymmetric trust model**
message privacy	symmetric (a.k.a. private-key) encryption	asymmetric (a.k.a. public-key) encryption
message authenticity	message authentication code (MAC)	digital signature scheme

Figure 1.7: Summary of main goals and trust models.

1.1.3 Summary

To summarize, there are two common aims concerned with mimicking an ideal channel: achieving message privacy and achieving message authenticity. There are two main trust models in which we are interested in achieving these goals: the symmetric trust model and the asymmetric trust model. The tools used to achieve these four goals are named as shown in Fig. 1.7.

1.2 Other goals

Cryptography has numerous other goals, some related to the ones above, some not. Let us discuss a few of them.

1.2.1 Pseudorandom Number Generation

Lots of applications require "random" numbers or bits. These applications involve simulation, efficient algorithms, and cryptography itself. In particular, randomness is essential to key generation, and, additionally, many cryptographic algorithms, such as encryption algorithms, are randomized.

A pseudorandom number generator is a deterministic algorithm that takes as input a short random string called a *seed* and stretches it to output a longer sequence of bits that is "pseudorandom."

In some applications, people use Linear Congruential Generators (LCGs) for pseudorandom number generation. But LCGs do not have good properties with regard to the quality of pseudorandomness of the bits output. With the ideas and techniques of modern cryptography, one can do much better. We will say what it means for a pseudorandom number generator to be "good" and then how to design one that is good in this sense. Our notion of "good" is such that our generators provably suffice for typical applications.

It should be clarified that pseudorandom generators do not generate pseudorandom bits from scratch. They need as input a random seed, and their job is to stretch this. Thus, they reduce the task of random number generation to the task of generating a short random seed. As to how to do the latter, we must step outside the domain of cryptography. We might wire to our computer a Geiger counter that generates a "random" bit every second, and run the computer for, say, 200 seconds, to get a 200 bit random seed, which we can then stretch via the pseudorandom number generator. Sometimes, more ad hoc methods are used; a computer might obtain a "random" seed by computing some function of various variable system parameters such as the time and system load.

We won't worry about the "philosophical" question as to whether the bits that form the seed are random in any *real* sense. We'll simply assume that these bits are completely unpredictable to anything "beyond" the computer which has gathered this data—mathematically, we'll treat these bits as random. We will then study pseudorandom number generation under the assumption that a random seed is available.

1.2.2 Authenticated key exchange

It is common for a pair of communicating parties to wish to establish a *secure session*. This is a communication session in which they exchange information with the conviction that each is indeed speaking to the other, and the content of the information remains hidden to any third party. One example is a login session in which Alice wishes to remotely logon to her computer. Another example is a web-browsing session in which a client wants to communicate securely with a server for some period.

Parties who already either share a secret key or are in possession of authentic copies of each other's public keys could use these keys directly to provide privacy and integrity of communicated data, via symmetric or asymmetric cryptography. However, this is not what is commonly done. Rather, the parties will use their existing keys —called *long-lived keys* in this context— to derive a session key. This is done via an *authenticated key exchange* protocol. This is a message exchange whose goal is to provide the parties a "fresh" and authentic shared key that will then be used to encrypt and authenticate traffic in the session using symmetric cryptography. Once the session is over, the session key is discarded.

Authenticated key exchange is one of the more subtle goals in cryptography, and will spend some time later applying the paradigms of modern cryptography to see how to define this goal and provide high-assurance solutions.

1.2.3 Coin Flipping

Alice and Bob are getting divorced, and want to decide who gets to keep the car. Alice calls Bob on the telephone and offers a simple solution. "Bob," she says, "I've got a penny in my pocket. I'm going to toss it in the air right now. You call *heads* or *tails.* If you get it right, you get the car. If you get it wrong, I get the car."

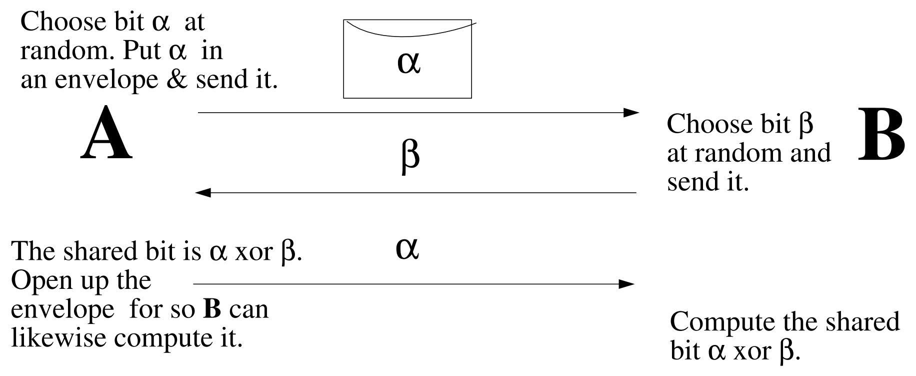
Choose bit α at random. Put α in an envelope & send it.
α
A
Choose bit β at random and send it.
B
β
The shared bit is α xor β. Open up the envelope for so B can likewise compute it.
α
Compute the shared bit α xor β.

Figure 1.8: Envelope solution to the telephone-coin-flipping 5problem.

Bob is not as bright as Alice, but something troubles him about this arrangement.

The *telephone-coin-flip* problem is to come up with a protocol so that, to the maximal extent possible, neither Alice nor Bob can cheat the other and, at the same time, each of them learn the outcome of a fair coin toss.

Here is a solution—sort of. Alice puts a random bit α inside an envelope and sends it to Bob. Bob announces a random bit β. Now Alice opens the envelope for Bob to see. The shared bit is defined as $\alpha \oplus \beta$. See Figure 1.8.

To do this over the telephone we need some sort of "electronic envelope" (in cryptography, this called a *commitment scheme*). Alice can put a value in the envelope and Bob can't see what the envelope contains. Later, Alice can open the envelope so that Bob can see what the envelope contains. Alice can't change her mind about an envelope's contents—it can only be opened up in one way.

Here is a simple technique to implement an electronic envelope. To put a "0" inside an envelope Alice chooses two random 500-bit primes p and q subject to the constraints that $p < q$ and $p \equiv 1 \pmod 4$ and $q \equiv 3 \pmod 4$. The product of p and q, say $N = pq$, is the commitment to zero; that is what Alice would send to commit to 0. To put a "1" inside an envelope Alice chooses too random 500-bit primes p and q subject to the constraints that $p < q$ and $p \equiv 3 \pmod 4$ and $q \equiv 1 \pmod 4$. The product of these, $N = pq$, is the commitment to 1. Poor Bob, seeing N, would like to figure out if the smaller of its two prime factors is congruent to 1 or to 3 modulo 4. We have no idea how to make that determination short of factoring N—and we don't know how to factor 1000 digit numbers which are the product of random 500-digit primes. Our best algorithms would, take way too long to run. When Alice wants to decommit (open the envelope) N she announces p and q. Bob verifies that they are prime (this is easy to do) and multiply to N, and then he looks to see if the smaller factor is congruent to 1 or to 3 modulo 4.

1.3 What cryptography is about

Let us now move away from the particular examples we have given and ask what, in general, is cryptography about?

1.3.1 Protocols, parties and adversaries

Briefly, cryptography is about constructing and analyzing *protocols* which overcome the influence of *adversaries*. In the last sections we gave examples of several different protocol problems, and a couple of different protocols.

Suppose that you are trying to solve some cryptographic problem. The problem will usually involve some number of *parties*. Us cryptographers often like to anthropomorphize our parties, giving them names like "Alice" and "Bob" and referring to them as though they are actual people. We do this because it's convenient and fun. But you shouldn't think that it means that the parties are *really* human beings. They might be—but they could be lots of other things, too. Like a cell phone, a computer, a processes running on a computer, an institution, or maybe a little gadget sitting on the top of your television set.

We usually think of the parties as the "good guys," and we want to help them accomplish their goal. We do this by making a protocol for the parties to use.

A protocol tells each party how to behave. A protocol is essentially a program, but it's a distributed program. Here are some features of protocols for you to understand.

A protocol instructs the parties what to do. It doesn't tell the adversary what to do. That is up to her.

A protocol can be *probabilistic*. This means that it can make random choices. To formalize this we usually assume that the model of computation that allows a party to specify a number $n \geq 2$ and then obtain a random value $i \overset{\$}{\leftarrow} \{0, 1, \ldots, n-1\}$. This notation means that i is a random value from the indicated set, all values being equally likely.

A protocol can be *stateful*. This means that when a party finishes what he is doing he can retain some information for the next time that he is active. When that party runs again he will remember the state that he was last in. So, for example, you could have a party that knows "this is the first time I've been run," "this is the second time I've been run," and so on.

When we formalize protocols, they are usually tuples of algorithms. But the actual formalization will vary from problem to problem. For example, a protocol for symmetric encryption isn't the same "type" of thing as a protocol for a telephone coin flip.

Another word for a protocol is a *scheme*. We'll use the two words interchangeably. So an encryption scheme is a protocol for encryption, and a message-authentication scheme is a protocol for message authentication. For us, a function, computed by a deterministic, sequential algorithm, is also a protocol. It's a particularly simple kind of protocol.

How can we devise and analyze protocols? The first step is to try to understand the *threats* and the *goals* for our particular problem. Once we have a good idea about these, we can try to find a protocol solution.

The *adversary* is the agent that embodies the "source" of the threat. Adversaries aim to defeat our protocol's goals. Protocols, in turn, are designed to to surmount the behavior of adversaries. It is a game—a question of who is more clever, protocol designer or adversary.

The adversary is usually what we focus on. In rigorous formalizations of cryptographic problems, the parties may actually vanish, being "absorbed" into the formalization. But the adversary will never vanish. She will be at center stage.

Cryptography is largely about thinking about the adversary. What can she do, and what can't she do? What is she trying to accomplish? We have to answer these questions before we can get very far.

Just as we warned that one shouldn't literally regard our parties as people, so too for the adversary. The adversary *might* represent an actual person, but it might just as well be an automated attack program, a competitor's company, a criminal organization, a government institution, one or more of the protocol's legitimate parties, a group of friendly hackers, or merely some unlucky circumstances conspiring together, not controlled by any intelligence at all.

By imagining a powerful adversary we take a pessimistic view about what might go wrong. We aim to succeed even if someone is out to get us. Maybe nobody is out to get us. In that case, we should at least be achieving high *reliability*. After all, if a powerful adversary can't succeed in disrupting our endeavors, then neither will noisy lines, transmission errors due to software bugs, unlucky message delivery times, careless programmers sending improperly formatted messages, and so forth.

When we formalize adversaries they will be random access machines (RAMs) with access to an oracle.

1.3.2 Cryptography and computer security

Good protocols are an essential tool for making secure computing systems. Badly designed protocols are easily exploited to break into computer systems, to eavesdrop on phone calls, to steal services, and so forth. Good protocol design is also hard. It is easy to under-estimate the task and quickly come up with *ad hoc* protocols that later turn out to be wrong. In industry, the necessary time and expertise for proper protocol design is typically under-estimated, often at future cost. It takes knowledge, effort and ingenuity to do the job right.

Security has many facets. For a system to be secure, many factors must combine. For example, it should not be possible for hackers to exploit bugs, break into your system, and use your account. They shouldn't be able to buy off your system administrator. They shouldn't be able to steal your back-up tapes. These things lie in the realm of system security.

The cryptographic protocol is just one piece of the puzzle. If it is poorly designed, the attacker will exploit that. For example, suppose the protocol transmits your password in the clear (that is, in a way that anyone watching can understand what it is). That's a protocol problem, not a system problem. And it will certainly be exploited.

The security of the system is only as strong as its weakest link. This is a big part of the difficulty of building a secure system. To get security we need to address all the problems: how do we secure our machines against intruders, how do we administer machines to maintain security, how do we design good protocols, and so on. All of these problems are important, but we will not address all of these problems here. This course is about the design of secure protocols. We usually have to assume that the rest of the system is competent at doing its job.

We make this assumption because it provides a natural abstraction boundary in dealing with the enormous task of providing security. Computer system security is a domain of a different nature, requiring different tools and expertise. Security can be best addressed by splitting it into more manageable components.

1.3.3 The rules of the game

Cryptography has rules. The first rule is that we may only try to overcome the adversary by means of protocols. We aren't allowed to overcome the adversary by intimidating her, arresting her, or putting poison in her coffee. These methods might be effective, but they are not cryptography.

Another rule that most cryptographers insist on is to make the protocols *public*. That which must be secret should be embodied in **keys**. Keys are data, not algorithms. Why do we insist that our protocols be public? There are several reasons. A resourceful adversary will likely find out what the protocol is anyway, since it usually has to be embodied in many programs or machines; trying to hide the protocol description is likely to be costly or infeasible. More than that, the attempt to hide the protocol makes one wonder if you've achieved security or just obfuscation. Peer review and academic work cannot progress in the absence of known mechanisms, so keeping cryptographic methods secret is often seen as anti-intellectual and a sign that ones work will not hold up to serious scrutiny.

Government organizations that deal in cryptography often do not make their mechanisms public. For them, learning the cryptographic mechanism is one more hoop that that the adversary must jump through. Why give anything away? Some organizations may have other reasons for not wanting mechanisms to be public, like a fear of disseminating cryptographic know-how, or a fear that the organization's abilities, or inabilities, will become better understood.

1.4 Approaches to the study of cryptography

Here we very briefly discuss the history of cryptography, and then at two development paradigms, namely *cryptanalysis-driven* design and *proof-driven* design.

1.4.1 Phases in cryptography's development

The history of cryptography can roughly be divided into three stages. In the first, early stage, algorithms had to be implementable with paper and ink. Julius Caesar used cryptograms. His and other early symmetric encryption schemes often took the form of *substitution ciphers*. In such a scheme, a key is a permutation π: $\Sigma \to \Sigma$ (meaning, a one-to-one, onto map from the alphabet to itself). A symbol $\sigma \in \Sigma$ is encrypted as $\pi(\sigma)$, and a piece of text is encrypted by encrypting each symbol in it. Decryption is done using the map π^{-1}. As we will see, however, such schemes are not very secure. The system can be strengthened in various ways, but none too effective.

The second age of cryptography was that of cryptographic engines. This is associated to the period of the World War II, and the most famous crypto engine was the German Enigma machine. How its codes were broken is a fascinating story.

The last stage is modern cryptography. Its central feature is the reliance on mathematics and electronic computers. Computers enabled the use of much more sophisticated encryption algorithms, and mathematics told us how to design them. It is during this most recent stage that cryptography becomes much more a science.

1.4.2 Cryptanalysis-driven design

Traditionally, cryptographic mechanisms have been designed by focusing on concrete attacks and how to defeat them. The approach has worked something like this.

(1) A cryptographic goal is recognized.

(2) A solution is offered.

(3) One searches for an attack on the proposed solution.

(4) When one is found, if it is deemed damaging or indicative of a potential weakness, you go back to Step 2 and try to come up with a better solution. The process then continues.

Sometimes one finds protocol problems in the form of subtle mathematical relationships that allow one to subvert the protocol's aims. Sometimes, instead, one "jumps out of the system," showing that some essential cryptographic issue was overlooked in the design, application, or implementation of the cryptography.

Some people like to use the word *cryptography* to refer to the making of cryptographic mechanisms, *cryptanalysis* to refer to the attacking of cryptographic mechanisms, and *cryptology* to refer to union. Under this usage, we've been saying "cryptography" in many contexts where "cryptology" would be more accurate. Most cryptographers don't observe this distinction between the words "cryptography" and "cryptology," so neither will we.

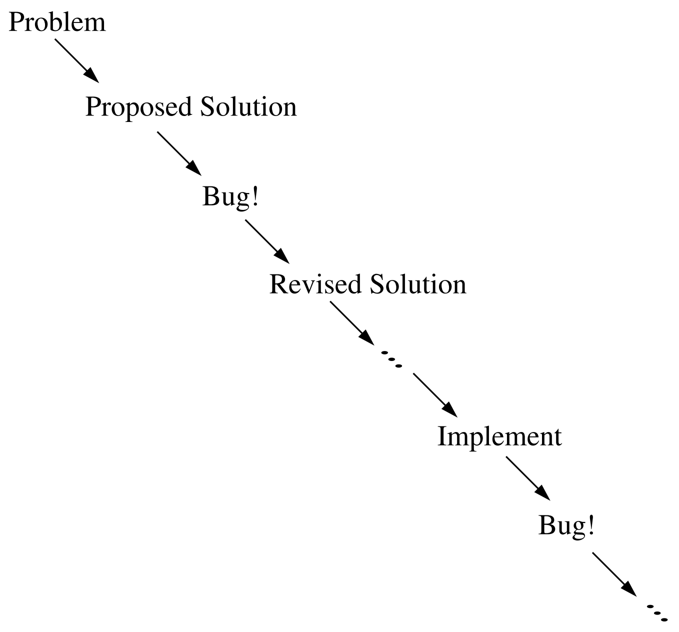

Figure 1.9: The classical-cryptography approach.

There are some difficulties with the approach of cryptanalysis-drive design. The obvious problem is that one never knows if things are right, nor when one is finished! The process should iterate until one feels "confident" that the solution is adequate. But one has to accept that design errors might come to light at any time. If one is making a commercial product one must eventually say that enough is enough, ship the product, and hope for the best. With luck, no damaging attacks will subsequently emerge. But sometimes they do, and when this happens the company that owns the product may find it difficult or impossible to effectively fix the fielded solution. They might try to keep secret that there is a good attack, but it is not easy to keep secret such a thing. See Figure 1.9.

Doing cryptanalysis well takes a lot of cleverness, and it is not clear that insightful cryptanalysis is a skill that can be effectively taught. Sure, one can study the most famous attacks—but will they really allow you to produce a new, equally insightful one? Great cleverness and mathematical prowess seem to be the requisite skills, not any specific piece of knowledge. Perhaps for these reasons, good cryptanalysts are very valuable. Maybe you have heard of Adi Shamir or Don Coppersmith, both renowned cryptanalysts.

Sadly, it is hard to base a science on an area where assurance is obtained by knowing that Coppersmith thought about a mechanism and couldn't find an attack. We need to pursue things differently.

1.4.3 Shannon security for symmetric encryption

The "systematic" approach to cryptography, where proofs and definitions play a visible role, begins in the work of Claude Shannon. Shannon was not only the father of information theory, but he might also be said to be the father of the modern-era of cryptography.

Let's return to the problem of symmetric encryption. Security, we have said, means defeating an adversary, so we have to specify what is it the adversary wants to do. As we have mentioned before, we need some formal way of saying what it means for the scheme to be secure. The idea of

Shannon, which we consider in more depth later, is to say that a scheme is perfectly secure if, for any two messages M_1, M_2, and any ciphertext C, the latter is just as likely to show up when M_1 is encrypted as when M_2 is encrypted. Here, likelihood means the probability, taken over the choice of key, and coins tossed by the encryption algorithm, if any.

Perfect security is a very powerful guarantee; indeed, in some sense, the best one can hope for. However, it has an important limitation, namely that, to achieve it, the number of message bits that one can encrypt cannot exceed the number of bits in the key. But if we want to do practical cryptography, we must be able to use a single short key to encrypt lots of bits. This means that we will not be able to achieve Shannon's perfect security. We must seek a different paradigm and a different notion of security that although "imperfect" is good enough.

1.4.4 Computational-complexity theory

Modern cryptography introduces a new dimension: the amount of computing power available to an adversary. It seeks to have security as long as adversaries don't have "too much" computing time. Schemes are breakable "in principle," but not in practice. Attacks are infeasible, not impossible.

This is a radical shift from many points of view. It takes cryptography from the realm of information theory into the realm of computer science, and complexity theory in particular, since that is where we study how hard problems are to solve as a function of the computational resources invested. And it changes what we can efficiently achieve.

We will want to be making statements like this:

> Assuming the adversary uses no more than t computing cycles, her probability of breaking the scheme is at most $t/2^{200}$.

Notice again the statement is probabilistic. Almost all of our statements will be.

Notice another important thing. Nobody said anything about *how* the adversary operates. What algorithm, or technique, does she use? We do not know anything about that. The statement holds nonetheless. So it is a very strong statement.

It should be clear that, in practice, a statement like the one above would be good enough. As the adversary works harder, her chance of breaking the scheme increases, and if the adversary had 2^{200} computing cycles at her disposal, we'd have no security left at all. But nobody has that much computing power.

Now we must ask ourselves how we can hope to get protocols with such properties. The legitimate parties must be able to efficiently execute the protocol instructions: their effort should be reasonable. But somehow, the task for the adversary must be harder.

1.4.5 Atomic primitives

We want to make a distinction between the protocols that that we use and those that we are designing. At the lowest level are what we call *atomic primitives*. Higher level protocols are built on top of these.

$$\begin{tikzcd} \boxed{\text{Atomic Primitives}} \arrow[d] \\ \boxed{\text{Protocols}} \end{tikzcd}$$

What's the distinction? Perhaps the easiest way to think of it is that the protocols we build address a cryptographic problem of interest. They say how to encrypt, how to authenticate, how to distribute a key. We build our protocols out of atomic primitives. Atomic primitives are protocols in their own right, but they are simpler protocols. Atomic primitives have some sort of "hardness" or "security" properties, but by themselves they don't solve any problem of interest. They must be properly used to achieve some useful end.

In the early days nobody bothered to make such a distinction between protocols and the primitives that used them. And if you think of the one-time pad encryption method, there is really just one object, the protocol itself.

Atomic primitives are drawn from two sources: engineered constructs and mathematical problems. In the first class fall standard *blockciphers* such as the well-known DES algorithm. In the second class falls the RSA function. We'll be looking at both types of primitives later.

The computational nature of modern cryptography means that one must find, and base cryptography on, computationally hard problems. Suitable ones are not so commonplace. Perhaps the first thought one might have for a source of computationally hard problems is **NP**-complete problems. Indeed, early cryptosystems tried to use these, particularly the Knapsack problem. However, these efforts have mostly failed. One reason is that **NP**-complete problems, although apparently hard to solve in the worst-case, may be easy on the average.

An example of a more suitable primitive is a *one-way function*. This is a function $f\colon D \to R$ mapping some domain D to some range R with two properties:

(1) f is easy to compute: there is an efficient algorithm that given $x \in D$ outputs $y = f(x) \in R$.

(2) f is hard to invert: an adversary I given a random $y \in R$ has a hard time figuring out a point x such that $f(x) = y$, as long as her computing time is restricted.

The above is not a formal definition. The latter, which we will see later, will talk about probabilities. The input x will be chosen at random, and we will then talk of the probability an adversary can invert the function at $y = f(x)$, as a function of the time for which she is allowed to compute.

Can we find objects with this strange asymmetry? It is sometimes said that one-way functions are obvious from real life: it is easier to break a glass than to put it together again. But we want concrete mathematical functions that we can implement in systems.

One source of examples is number theory, and this illustrates the important interplay between number theory and cryptography. A lot of cryptography has been done using number theory. And there is a very simple one-way function based on number theory—something you already know quite well. Multiplication! The function f takes as input two numbers, a and b, and multiplies them together to get $N = ab$. There is no known algorithm that given a random $N = ab$, always and quickly recovers a pair of numbers (not 1 and N, of course!) that are factors of N. This "backwards direction" is the *factoring* problem, and it has remained unsolved for hundreds of years.

Here is another example. Let p be a prime. The set $Z_p^* = \{1, \ldots, p-1\}$ turns out to be a group under multiplication modulo p. We fix an element $g \in Z_p^*$ which generates the group (that is, $\{g^0, g^1, g^2, \ldots, g^{p-2}\}$ is all of Z_p^*) and consider the function $f\colon \{0, \ldots, p-2\} \to Z_p^*$ defined by $f(x) = g^x \bmod p$. This is called the *discrete exponentiation* function, and its inverse is called the *discrete logarithm* function: $\log_g(y)$ is the value x such that $y = g^x$. It turns out there is no known fast algorithm that computes discrete logarithms, either. This means that for large enough p (say 1000 bits) the task is infeasible, given current computing power, even in thousands of years. So this is another one-way function.

It should be emphasized though that these functions have not been *proven* to be hard functions to invert. Like **P** versus **NP**, whether or not there is a good one-way function out there is an open question. We have some candidate examples, and we work with them. Thus, cryptography is build on assumptions. If the assumptions are wrong, a lot of protocols might fail. In the meantime we live with them.

1.4.6 The provable-security approach

While there are several different ways in which proofs can be effective tools in cryptography, we will generally follow the proof-using tradition which has come to be known as "provable security." Provable security emerged in 1982, with the work of Shafi Goldwasser and Silvio Micali. At that time, Goldwasser and Micali were graduate students at UC Berkeley. They, and their advisor Manuel Blum, wanted to put public-key encryption on a scientifically firm basis. And they did that, effectively creating a new viewpoint on what cryptography is really about.

We have explained above that we like to start from atomic primitives and transform them into protocols. Now good atomic primitives are rare, as are the people who are good at making and attacking them. Certainly, an important effort in cryptography is to design new atomic primitives, and to analyze the old ones. This, however, is not the part of cryptography that this course will focus on. One reason is that the weak link in real-world cryptography seems to be between atomic primitives and protocols. It is in this transformation that the bulk of security flaws arise. And there is a science that can do something about it, namely, provable security.

We will view a cryptographer as an engine for turning atomic primitives into protocols. That is, we focus on protocol design under the assumption that good atomic primitives exist. Some examples of the kinds of questions we are interested in are these. What is the best way to encrypt a large text file using DES, assuming DES is secure? What is the best way to design a signature scheme using multiplication, assuming that multiplication is one-way? How "secure" are known methods for these tasks? What do such questions even mean, and can we find a good framework in which to ask and answer them?

A poorly designed protocol can be insecure *even though the underlying atomic primitive is good.* The fault is not of the underlying atomic primitive, but that primitive was somehow misused.

Indeed, lots of protocols have been broken, yet the good atomic primitives, like DES and multiplication and RSA, have never been convincingly broken. We would like to build on the strength of such primitives in such a way that protocols can "inherit" this strength, not lose it. The provable-security paradigm lets us do that.

The provable-security paradigm is as follows. Take some goal, like achieving privacy via symmetric encryption. The first step is to make a formal adversarial *model* and *define* what it *means* for an encryption scheme to be secure. The definition explains exactly when—on which runs—the adversary is successful.

With a definition in hand, a particular protocol, based on some particular atomic primitive, can be put forward. It is then analyzed from the point of view of meeting the definition. The plan is now show security via a *reduction.* A reduction shows that the *only* way to defeat the protocol is to break the underlying atomic primitive. Thus we will also need a formal definition of what the atomic primitive is supposed to do.

A reduction is a proof that if the atomic primitive does the job it is supposed to do, then the protocol we have made does the job that it is supposed to do. Believing this, it is no longer necessary to directly cryptanalyze the protocol: if you were to find a weakness in it, you would have unearthed one in the underlying atomic primitive. So if one is going to do cryptanalysis, one might as well focus on the atomic primitive. And if we believe the latter is secure, then we *know*, without further cryptanalysis of the protocol, that the protocol is secure, too.

A picture for the provable-security paradigm might look like Fig. 1.10.

In order to do a reduction one must have a formal notion of what is meant by the security of the underlying atomic primitive: what attacks, exactly, does it withstand? For example, we might assume that RSA is a one-way function.

Here is another way of looking at what reductions do. When I give you a reduction from the

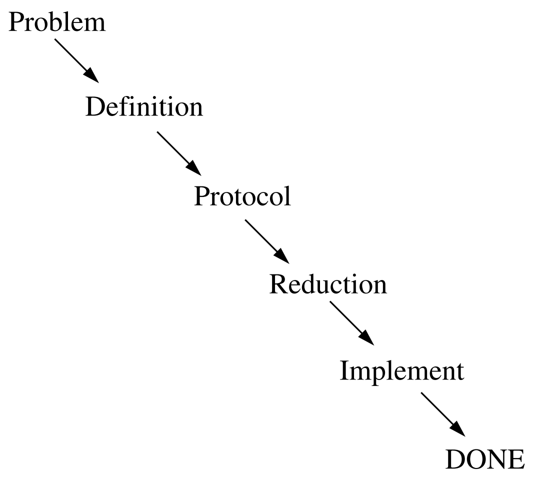

Figure 1.10: The provable-security paradigm.

We think that computational problem Ξ can't be solved in polynomial time.	We think that cryptographic protocol Π can't be effectively attacked.
We believe this because if Ξ could be solved in polynomial time, then so could SAT (say).	We believe this because if Π could be effectively attacked, then so could RSA (say).
To show this we *reduce* SAT to Ξ: we show that *if* somebody could solve Ξ in polynomial time, then they could solve SAT in polynomial time, too.	To show this we *reduce* RSA to Π: we show that *if* somebody could break Π by effective means, then they could break RSA by effective means, too.

Figure 1.11: The analogy between reductionist-cryptography and NP-Completeness.

onewayness of RSA to the security of my protocol, I am giving you a *transformation* with the following property. Suppose you claim to be able to break my protocol P. Let A be the adversary that you have that does this. My transformation takes A and turns it into another adversary, A', that breaks RSA. Conclusion: as long as we believe you can't break RSA, there could be no such adversary A. In other words, my protocol is secure.

Those familiar with the theory of **NP**-completeness will recognize that the basic idea of reductions is the same. When we provide a reduction from SAT to some computational problem Ξ we are saying our Ξ is hard unless SAT is easy; when we provide a reduction from RSA to our protocol Π, we are saying that Π is secure unless RSA is easy to invert. The analogy is further spelled out in Fig. 1.11, for the benefit of those of you familiar with the notion of NP-Completeness.

Experience has taught us that the particulars of reductions in cryptography are a little harder to comprehend than they were in elementary complexity theory. Part of the difficulty lies in the fact that every problem domain will have it's own unique notion of what is an "effective attack." It's rather like having a different "version" of the notion of NP-Completeness as you move from one problem to another. We will also be concerned with the *quality* of reductions. One could have concerned oneself with this in complexity theory, but it's not usually done. For doing practical work in cryptography, however, paying attention to the quality of reductions is important. Given these difficulties, we will proceed rather slowly through the ideas. Don't worry; you will get it (even if you never heard of NP-Completeness).

The concept of using reductions in cryptography is a beautiful and powerful idea. Some of us by now are so used to it that we can forget how innovative it was! And for those not used to it, it can be hard to understand (or, perhaps, believe) at first hearing—perhaps because it delivers so much. Protocols designed this way truly have superior security guarantees.

In some ways the term "provable security" is misleading. As the above indicates, what is probably the central step is providing a model and definition, which does not involve proving anything. And then, one does not "prove a scheme secure:" one provides a reduction of the security of the scheme to the security of some underlying atomic primitive. For that reason, we sometimes use the term "reductionist security" instead of "provable security" to refer to this genre of work.

1.4.7 Theory for practice

As you have by now inferred, this course emphasizes general principles, not specific systems. We will not be talking about the latest holes in *sendmail* or *Netscape*, how to configure *PGP*, or the latest attack against the ISO 9796 signature standard. This kind of stuff is interesting and useful, but it is also pretty transitory. Our focus is to understand the fundamentals, so that we know how to deal with new problems as they arise.

We want to make this clear because cryptography and security are now quite hyped topic. There are many buzzwords floating around. Maybe someone will ask you if, having taken a course, you know one of them, and you will not have heard of it. Don't be alarmed. Often these buzzwords don't mean much.

This is a theory course. Make no mistake about that! Not in the sense that we don't care about practice, but in the sense that we approach practice by trying to understand the fundamentals and how to apply them. Thus the main goal is to understand the theory of protocol design, and how to apply it. We firmly believe it is via an understanding of the theory that good design comes. If you know the theory you can apply it anywhere; if you only know the latest technology your knowledge will soon by obsolete. We will see how the theory and the practice can contribute to each other, refining our understanding of both.

In assignments you will be asked to prove theorems. There may be a bit of mathematics for you to pick up. But more than that, there is "mathematical thinking."

Don't be alarmed if what you find in these pages contradicts "conventional wisdom." Conventional wisdom is often wrong! And often the standard texts give an impression that the field is the domain of experts, where to know whether something works or not, you must consult an expert or the recent papers to see if an attack has appeared. The difference in our approach is that you will be given reasoning tools, and you can then think for yourself.

Cryptography is fun. Devising definitions, designing protocols, and proving them correct is a highly creative endeavor. We hope you come to enjoy thinking about this stuff, and that you come to appreciate the elegance in this domain.

1.5 What background do I need?

Now that you have had some introduction to the material and themes of the class, you need to decide whether you should take it. Here are some things to consider in making this decision.

A student taking this course is expected to be comfortable with the following kinds of things, which are covered in various other courses.

The first is probability theory. Probability is everywhere in cryptography. You should be comfortable with ideas like sample spaces, events, experiments, conditional probability, random variables and their expectations. We won't use anything deep from probability theory, but we will draw heavily on the language and basic concepts of this field.

You should know about alphabets, strings and formal languages, in the style of an undergraduate course in the theory of computation.

You should know about algorithms and how to measure their complexity. In particular, you should have taken and understood at least an undergraduate algorithms class.

Most of all you should have general mathematical maturity, meaning, especially, you need to be able to understand what is (and what is not) a proper definition.

1.6 Problems

Problem 1 Besides the symmetric and the asymmetric trust models, think of a couple more ways to "create asymmetry" between the receiver and the adversary. Show how you would encrypt a bit in your model. ▮

Problem 2 In the telephone coin-flipping protocol, what should happen if Alice refuses to send her second message? Is this potentially damaging? ▮

Problem 3 Argue that what we have said about keeping the algorithm public but the key secret is fundamentally meaningless. ▮

Problem 4 *A limitation on fixed-time fair-coin-flipping TMs.* Consider the model of computation in which we augment a Turing machine so that it can obtain the output of a random coin flip: by going into a distinguished state $Q_\$$, the next state will be Q_H with probability 1/2, and the next state will be Q_T with probability 1/2. Show that, in this model of computation, there is no constant-time algorithm to perfectly deal out five cards to each of two players.

(A deck of cards consists of 52 cards, and a perfect deal means that all hands should be equally likely. Saying that the algorithm is constant-time means that there is some number T such that the algorithm is guaranteed to stop within T steps.) ▮

Problem 5 *Composition of EPT Algorithms.* John designs an EPT (expected polynomial time) algorithm to solve some computational problem Π—but he assumes that he has in hand a black-box (ie., a unit-time subroutine) which solves some other computational problem, Π'. Ted soon discovers an EPT algorithm to solve Π'. True or false: putting these two pieces together, John and Ted now have an EPT algorithm for Π. Give a proof or counterexample.

(When we speak of the worst-case running time of machine M we are looking at the function $T(n)$ which gives, for each n, the maximal time which M might spend on an input of size n: $T(n) = \max_{x,\,|x|=n}[\#\mathrm{Steps}_M(x)]$. When we speak of the expected running time of M we are instead looking at the function $T(n)$ which gives, for each n, the maximal value among inputs of length n of the expected value of the running time of M on this input—that is, $T(n) = \max_{x,\,|x|=n} \mathbf{E}[\#\mathrm{Steps}_M(x)]$, where the expectation is over the random choices made by M.) ▮

Bibliography

[DH] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Trans. Info. Theory*, Vol. IT-22, No. 6, November 1976, pp. 644–654.

Chapter 2

Classical Encryption

In this chapter we take a quick look at some classical encryption techniques, illustrating their weakness and using these examples to initiate questions about how to define privacy. We then discuss Shannon's notion of perfect security.

2.1 Substitution ciphers

One of the earliest approaches to symmetric encryption is what is called a substitution cipher. Say the plaintext is English text. We can view this as a sequence of symbols, each symbol being either a letter, a blank or a punctuation mark. Encryption substitutes each symbol σ with another symbol $\pi(\sigma)$. The function π is the key, and has to be a permutation (meaning, one-to-one and onto) so that decryption is possible.

Encryption of this form is quite natural and well known, and, indeed, to many people it defines how encryption is done. We will later see many other (and better) ways to encrypt, but it is worth beginning by exploring this one.

Let's begin by specifying the scheme a little more mathematically. It may be valuable at this time to review the box in the Introduction that recalls the vocabulary of formal languages; we will be talking of things like alphabets, symbols, and strings.

Let Σ be a finite alphabet, whose members are called symbols. (In our examples, Σ would contain the 26 letters of the English alphabet, the blank symbol $\sqcup$, and punctuation symbols. Let us refer to this henceforth as the *English alphabet.*) If x is a string over Σ then we denote by $x[i]$ its i-th symbol.

Recall that if x is a string then $|x|$ denotes the length of x, meaning the number of symbols in it. Let us also adopt the convention that if X is a set then $|X|$ denotes its size. The double use of the "$|\cdot|$" notation should not cause much problem since the type of object to which it is applied, namely a set or a string, will usually be quite clear.

A *permutation* on a set S is a map π: $S \to S$ that is one-to-one and onto. Such a map is invertible, and we denote its inverse by π^{-1}. The inverse is also a permutation, and the map and its inverse are related by the fact that $\pi^{-1}(\pi(x)) = x$ and $\pi(\pi^{-1}(y)) = y$ for all $x, y \in S$. We let $\mathsf{Perm}(S)$ denote the set of all permutations on set S. Note that this set has size $|S|!$.

In the introduction, we had discussed symmetric encryption schemes, and said that any such scheme is specified as a triple $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ consisting of a key-generation algorithm, an encryption algorithm, and a decryption algorithm. A *substitution cipher over alphabet* Σ is a special kind of symmetric encryption scheme in which the output of the key-generation algorithm $\mathcal{K}$ is always a permutation over Σ and the encryption and decryption algorithms are as follows:

```
Algorithm $\mathcal{E}_\pi(M)$
    For $i = 1, \ldots, |M|$ do
        $C[i] \leftarrow \pi(M[i])$
Return $C$

Algorithm $\mathcal{D}_\pi(C)$
    For $i = 1, \ldots, |C|$ do
        $M[i] \leftarrow \pi^{-1}(C[i])$
Return $M$
```

Above, the plaintext M is a string over Σ, as is the ciphertext C. The key is denoted π and is a permutation over Σ. We will let $\mathsf{Keys}(\mathcal{SE})$ denote the set of all keys that might be output by $\mathcal{K}$.

There are many possible substitution ciphers over Σ, depending on the set $\mathsf{Keys}(\mathcal{SE})$. In the simplest case, this is the set of all permutations over Σ, and $\mathcal{K}$ is picking a permutation at random. But one might consider schemes in which permutations are chosen from a much smaller set.

In our examples, unless otherwise indicated, the alphabet will be the English one defined above, namely Σ contains the 26 English letters, the blank symbol $\sqcup$, and punctuation symbols. We will, for simplicity, restrict attention to substitution ciphers that are *punctuation respecting*. By this we mean that any key (permutation) $\pi \in \mathsf{Keys}(\mathcal{SE})$ leaves blanks and punctuation marks unchanged. In specifying such a key, we need only say how it transforms each of the 26 English letters.

Example 2.1.1 This is an example of how encryption is performed with a (punctuation respecting) substitution cipher. An example key (permutation) π is depicted below:

σ	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\pi(\sigma)$	D	B	U	P	W	I	Z	L	A	F	N	S	G	K	H	T	J	X	C	M	Y	O	V	E	Q	R

Note every English letter appears once and exactly once in the second row of the table. That's why π is called a permutation. The inverse π^{-1} permutation is obtained by reading the table backwards. Thus $\pi^{-1}(\mathtt{D}) = \mathtt{A}$ and so on. The encryption of the plaintext

$$M = \mathtt{HI\ THERE}$$

is

$$C = \pi(\mathtt{H})\pi(\mathtt{H})\pi(\mathtt{I})\pi(\sqcup)\pi(\mathtt{T})\pi(\mathtt{H})\pi(\mathtt{E})\pi(\mathtt{R})\pi(\mathtt{E}) = \mathtt{LA\ MLWXW} \quad \blacksquare$$

Now let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an arbitrary substitution cipher. We are interested in its security. To assess this we think about what the adversary has and what it might want to do.

The adversary begins with the disadvantage of not being given the key π. It is assumed however to come in possession of a ciphertext C. The most basic goal that we can consider for it is that it wants to recover the plaintext $M = \mathcal{D}(\pi, C)$ underlying C.

The adversary is always assumed to know the "rules of the game." Meaning, it knows the algorithms $\mathcal{K}, \mathcal{E}, \mathcal{D}$. It knows that a substitution cipher is being used, and that it is punctuation respecting in our case. The *only* thing it does not know a priori is the key, for that is assumed to have been shared secretly and privately between the sender and receiver.

So the adversary sees some gibberish, such as the text `LA MLWXW`. One might imagine that in the absence of the key π it would have a tough time figuring out that the message was `HI THERE`. But in fact, substitution ciphers are not so hard to cryptanalyze. Indeed, breaking a substitution cipher is a popular exercise in a Sunday newspaper or magazine, and many of you may have done it. The adversary can use its knowledge of the structure of English text to its advantage. Often a good way to begin is by making what is called a *frequency table*. This table shows, for ever letter τ, how often τ occurs in the ciphertext. Now it turns out that the most common letter in English

τ	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\pi^{-1}(\tau)$		R	T												H					A				E		
$\pi^{-1}(\tau)$		R	T	I					N						H	C				A		W		E		
$\pi^{-1}(\tau)$	L	R	T	I			M	F	N		O				H	C		S		A		W		E		
$\pi^{-1}(\tau)$	L	R	T	I			M	F	N		O			P	H	C	U	S		A	D	W		E		

Figure 2.1: Cryptanalysis of Example 2.1.2.

text is typically E. The next most common are the group T, A, O, I, N, S, H, R. (These letters have roughly the same frequency, somewhat lower than that of E, but higher than other letters.) So if X is the most frequent ciphertext symbol, a good guess would be that it represents E. (The guess is not necessarily true, but one attempts to validate or refute it in further stages.) Another thing to do is look for words that have few letters. Thus, if the letter T occurs by itself in the ciphertext, we conclude that it must represent A or I. Two letter words give similar information. And so on, it is remarkable how quickly you actually (usually) can figure out the key.

Example 2.1.2 Let us try to decrypt the following ciphertext:

```
COXBX TBX CVK CDGXR DI T GTI'R ADHX VOXI OX ROKQAU IKC RNXPQATCX: VOXI OX
PTI'C THHKBU DC, TIU VOXI OX PTI.
```

Here is our frequency table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
3	3	7	4	0	0	2	3	9	0	4	0	0	1	8	3	2	4	0	8	3	4	0	13	0	0

The most common symbol being X, we guess that $\pi^{-1}(\texttt{X}) = \texttt{E}$. Now we see the word OX, and, assuming X represents E, O must represent one of B, H, M, W. We also note that O has a pretty high frequency count, namely 8. So my guess is that O falls in the second group of letters mentioned above. But of the letters B, H, M and W, only H is in this group, so let's guess that $\pi^{-1}(\texttt{O}) = \texttt{H}$. Now, consider the first word in the ciphertext, namely COXBX. We read it as $*$HE$*$E. This could be THERE or THESE. I will guess that $\pi^{-1}(\texttt{C}) = \texttt{T}$, keeping in mind that $\pi^{-1}(\texttt{B})$ should be either R or S. The letter T occurs on its own in the ciphertext, so must represent A or I. But the second ciphertext word can now be read as $*$RE or $*$SE, depending on our guess for B discussed above. We know that the $*$ (which stands for the letter T in the ciphertext) decodes to either A or I. Even though a few choices yield English words, my bet is the word is ARE, so I will guess $\pi^{-1}(\texttt{T}) = \texttt{A}$ and $\pi^{-1}(\texttt{B}) = \texttt{R}$. The second row of the table in Fig. 2.1 shows where we are. Now let us write the ciphertext again, this time indicating above different letters what we believe them to represent:

```
THERE ARE T   T  E     A  A '     E  HE  HE  H        T   E   ATE:  HE  HE
COXBX TBX CVK CDGXR DI T GTI'R ADHX VOXI OX ROKQAU IKC RNXPQATCX: VOXI OX

 A 'T A   R   T, A    HE  HE  A .
PTI'C THHKBU DC, TIU VOXI OX PTI.
```

Since the last letter of the ciphertext word DC represents T, the first letter must represent A or I. But we already have something representing A, so we guess that $\pi^{-1}(\texttt{D}) = \texttt{I}$. From the ciphertext word DI it follows that I must be either N, T or S. It can't be T because C already represents T. But

I is also the last letter of the ciphertext word VOXI, and ∗HEN is a more likely ending than ∗HES so I will guess $\pi^{-1}(\texttt{I}) = \texttt{N}$. To make sense of the ciphertext word VOXI, I then guess that $\pi^{-1}(\texttt{V}) = \texttt{W}$. The ciphertext word PTI'C is now ∗AN'T and so surely $\pi^{-1}(\texttt{P}) = \texttt{C}$. The second row of the table of Fig. 2.1 shows where we are now, and our text looks like:

```
THERE ARE TW  TI E  IN A  AN'   I E WHEN HE  H     N T   EC  ATE: WHEN HE
COXBX TBX CVK CDGXR DI T GTI'R ADHX VOXI OX ROKQAU IKC RNXPQATCX: VOXI OX

CAN'T A   R  IT, AN  WHEN HE CAN.
PTI'C THHKBU DC, TIU VOXI OX PTI.
```

At this point I can decrypt the first 8 words of the ciphertext pretty easily: THERE ARE TWO TIMES IN A MAN'S LIFE. The third row of the table of Fig. 2.1 shows where we are after I put in the corresponding guesses. Applying them, our status is:

```
THERE ARE TWO TIMES IN A MAN'S LIFE WHEN HE SHO L  NOT S EC LATE: WHEN HE
COXBX TBX CVK CDGXR DI T GTI'R ADHX VOXI OX ROKQAU IKC RNXPQATCX: VOXI OX

CAN'T AFFOR  IT, AN  WHEN HE CAN.
PTI'C THHKBU DC, TIU VOXI OX PTI.
```

The rest is easy. The decryption is:

```
THERE ARE TWO TIMES IN A MAN'S LIFE WHEN HE SHOULD NOT SPECULATE: WHEN HE
COXBX TBX CVK CDGXR DI T GTI'R ADHX VOXI OX ROKQAU IKC RNXPQATCX: VOXI OX

CAN'T AFFORD IT, AND WHEN HE CAN.
PTI'C THHKBU DC, TIU VOXI OX PTI.
```

The third row of the table of Fig. 2.1 shows our final knowledge of the key π. The text, by the way, is a quotation from Mark Twain. ▌

Some people argue that this type of cryptanalysis is not possible if the ciphertext is short, and thus that substitution ciphers work fine if, say, one changes the key quite frequently. Other people argue for other kinds of variants and extensions. And in fact, these types of systems have been the basis for encryption under relatively modern times. We could spend a lot of time on this subject, and many books do, but we won't. The reason is that, as we will explain, the idea of a substitution cipher is flawed at a quite fundamental level, and the flaw remains in the various variations and enhancements proposed. It will take some quite different ideas to get systems that deliver quality privacy.

To illustrate why the idea of a substitution cipher is flawed at a fundamental level, consider the following example usage of the scheme. A polling station has a list of voters, call them $V_1, V_2, \ldots, V_n$. Each voter casts a (secret) ballot which is a choice between two values. You could think of them as YES or NO, being votes on some Proposition, or BUSH and KERRY. In any case, we represent them as letters: the two choices are Y and N. At the end of the day, the polling station has a list $v_1, \ldots, v_n$ of n votes, where v_i is V_i's vote. Each vote being a letter, either Y or N, we can think of the list of votes as a string $v = v_1 \ldots v_n$ over the alphabet of English letters. The polling station wants to transmit this string to a tally center, encrypted in order to preserve anonymity of votes. The polling station and tally center have agreed on a key π for a substitution cipher. The polling station encrypts the message string v to get a ciphertext string $c = \pi(v_1) \ldots \pi(v_n)$ and transmits this to the tally center. Our question is, is this secure?

It quickly becomes apparent that it is not. There are only two letters in v, namely Y and N. This means that c also contains only two letters. Let's give them names, say A and B. One of these is $\pi(\texttt{Y})$ and the other is $\pi(\texttt{N})$. If the adversary knew which is which, it would, from the ciphertext, know the votes of all voters. But we claim that it is quite easy for the adversary to know which is which. Consider for example that the adversary is one of the voters, say V_1. So it knows its own vote v_1. Say this is $\texttt{Y}$. It now looks at the first symbol in the ciphertext. If this is A, then it knows that $A = \pi(\texttt{Y})$ and thus that $B = \texttt{N}$, and can now immediately recover all of $v_2, \ldots, v_n$ from the ciphertext. (If the first symbol is B, it is the other way around, but again it recovers all votes.)

This attack works even when the ciphertext is short (that is, when n is small). The weakness is exhibits is in the very nature of the cipher, namely that a particular letter is always encrypted in the same way, and thus repetitions can be detected.

Pinpointing this weakness illustrates something of the types of mode of thought we need to develop in cryptography. We need to be able to think about usage of application scenarios in which a scheme that otherwise seems good will be seen to be bad. For example, above, we considered not only that the encrypted text is votes, but that the adversary could be one of the voters. We need to always ask, "what if?"

We want symmetric encryption schemes that are not subject to the types of attacks above and, in particular, would provide security in an application such as the voting one. Towards this end we now consider one-time-pad encryption.

2.2 One-time-pad encryption and perfect security

The One-Time-Pad (OTP) scheme with key-length m is the symmetric encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ whose algorithms are as follows. The code for the key-generation algorithm is

$$K \xleftarrow{\$} \{0,1\}^m \text{ ; return } K \ ,$$

meaning a key is a random m-bit string. The encryption algorithm is defined by $\mathcal{E}_K(M) = K \oplus M$, where the message M is an m-bit binary string and $\oplus$ denotes bitwise XOR. The decryption algorithm is defined by $\mathcal{D}_K(C) = K \oplus C$, where $C \in \{0,1\}^m$. The correctness condition is met because $\mathcal{D}_K(\mathcal{E}_K(M)) = K \oplus (K \oplus M) = M$ for all $M \in \{0,1\}^m$.

Let us go back to our voting example. Represent $\texttt{Y}$ by 1 and $\texttt{N}$ by 0, and now encrypt the vote string $v = v_1 \ldots v_n$ with the OTP scheme with key-length n. Thus the ciphertext is $C = K \oplus v$. This time, weaknesses such as those we saw above are not present. Say the adversary has the ciphertext $C = C_1 \ldots C_n$, where C_i is the i-th bit of C. Say the adversary knows that $v_1 = 1$. It can deduce that K_1, the first bit of K, is $1 \oplus C_1$. But having K_1 tells it nothing about the other bits of K, and hence $v_2, \ldots, v_n$ remain hidden.

It turns out that the higher quality of privacy we see here is not confined to this one application setting. The scheme has a property called perfect security, which we will define below, and effectively provides the best possible security. We will also see that substitution ciphers do not have this property, providing a more formal interpretation of the concrete weaknesses we have seen in them.

Before going on, we remark that the perfect security of the OTP with key-length m relies crucially on our encrypting only a single message of length m. Were we to encrypt two or more messages, privacy would be lost. To see this, suppose we encrypt M_1, then M_2. The adversary obtains $C_1 = K \oplus M_1$ and $C_2 = K \oplus M_2$. XORing them together, it obtains $M_1 \oplus M_2$. This however provides partial information about the data, and, in particular, if the adversary would now happen to learn M_1, it could deduce M_2, which is not desirable.

The idea behind perfect security is to consider that one of two messages M_1 or M_2 is being encrypted. The adversary is aware of this, and all it wants to know is which of the two it is. It has in hand the ciphertext C, and now asks itself whether, given C, it can tell whether it was M_1 or M_2 that gave rise to it. Perfect security says that the adversary cannot tell. It asks that the probability that C arises as the ciphertext is the same whether M_1 or M_2 was chosen to be encrypted.

Definition 2.2.1 (Perfect Security of a Symmetric Encryption Scheme) Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, and assume we use it to encrypt just one message, drawn from a set $\mathsf{Plaintexts}$. We say that $\mathcal{SE}$ is *perfectly secure* if for any two messages $M_1, M_2 \in \mathsf{Plaintexts}$ and any C

$$\Pr\left[\mathcal{E}_K(M_1) = C\right] \;=\; \Pr\left[\mathcal{E}_K(M_2) = C\right] \;. \tag{2.1}$$

In both cases, the probability is over the random choice $K \xleftarrow{\$} \mathcal{K}$ and over the coins tossed by $\mathcal{E}$ if any. ▌

Let us now show that a substitution cipher fails to have this property, even if the ciphertext encrypted is very short, say three letters.

Claim 2.2.2 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a substitution cipher over the alphabet Σ consisting of the 26 English letters. Assume that $\mathcal{K}$ picks a random permutation over Σ as the key. (That is, its code is $\pi \xleftarrow{\$} \mathsf{Perm}(\Sigma)$; return π.) Let $\mathsf{Plaintexts}$ be the set of all three letter English words. Assume we use $\mathcal{SE}$ to encrypt a single message from $\mathsf{Plaintexts}$. Then $\mathcal{SE}$ is *not* perfectly secure. ▌

Intuitively, this is due to the weakness we saw above, namely that if a letter appears twice in a plaintext, it is encrypted the same way in the ciphertext, and thus repetitions can be detected. If M_1, M_2 are messages such that M_1 contains repeated letters but M_2 does not, then, if the adversary sees a ciphertext with repeated letters it knows that M_1 was encrypted. This means that this particular ciphertext has different probabilities of showing up in the two cases. We now make all this formal.

Proof of Claim 2.2.2: We are asked to show that the condition of Definition 2.2.1 does not hold, so the first thing to do is refer to the definition and right down what it means for the condition to not hold. It is important here to be careful with the logic. The contrapositive of "for all M_1, M_2, C some condition holds" is "there exist M_1, M_2, C such that the condition does not hold." Accordingly, we need to show there exist $M_1, M_2 \in \mathsf{Plaintexts}$, and there exists C, such that

$$\Pr\left[\mathcal{E}_\pi(M_1) = C\right] \neq \Pr\left[\mathcal{E}_\pi(M_2) = C\right] \;. \tag{2.2}$$

We have replaced K with π because the key here is a permutation.

We establish the above by picking M_1, M_2, C in a clever way. Namely we set M_1 to some three letter word that contains a repeated letter; specifically, let us set it to `FEE`. We set M_2 to a three letter word that does not contain any repeated letter; specifically, let us set it to `FAR`. We set C to `XYY`, a ciphertext that has the same "pattern" as `FEE` in the sense that the last two letters are the same and the first is different from these. Now we evaluate the probabilities in question:

$$\begin{aligned}
\Pr\left[\mathcal{E}_\pi(M_1) = C\right] &= \Pr\left[\mathcal{E}_\pi(\texttt{FEE}) = \texttt{XYY}\right] \\
&= \frac{|\{\, \pi \in \mathsf{Perm}(\Sigma) \;:\; \mathcal{E}_\pi(\texttt{FEE}) = \texttt{XYY} \,\}|}{|\mathsf{Perm}(\Sigma)|} \\
&= \frac{|\{\, \pi \in \mathsf{Perm}(\Sigma) \;:\; \pi(\texttt{F})\pi(\texttt{E})\pi(\texttt{E}) = \texttt{XYY} \,\}|}{|\mathsf{Perm}(\Sigma)|} \\
&= \frac{24!}{26!} \\
&= \frac{1}{650} \;.
\end{aligned}$$

Recall that the probability is over the choice of key, here π, which is chosen at random from $\mathsf{Perm}(\Sigma)$, and over the coins of $\mathcal{E}$, if any. In this case, $\mathcal{E}$ does not toss coins, so the probability is over π alone. The probability can be expressed as the ratio of the number of choices of π for which the stated event, namely that $\mathcal{E}_\pi(\texttt{FEE}) = \texttt{XYY}$, is true, divided by the total number of possible choices of π, namely the size of the set $\mathsf{Perm}(\Sigma)$ from which π is drawn. The second term is 26!. For the first, we note that the condition means that $\pi(\texttt{F}) = \texttt{X}$ and $\pi(\texttt{E}) = \texttt{Y}$, but the value of π on any of the other 24 input letters may still be any value other than $\texttt{X}$ or $\texttt{Y}$. There are 24! different ways to assign distinct values to the remaining 24 inputs to π, so this is the numerator above. Now, we proceed similarly for M_2:

$$\begin{aligned}
\Pr\left[\mathcal{E}_\pi(M_2) = C\right] &= \Pr\left[\mathcal{E}_\pi(\texttt{FAR}) = \texttt{XYY}\right] \\
&= \frac{|\{\, \pi \in \mathsf{Perm}(\Sigma) \;:\; \mathcal{E}_\pi(\texttt{FAR}) = \texttt{XYY} \,\}|}{|\mathsf{Perm}(\Sigma)|} \\
&= \frac{|\{\, \pi \in \mathsf{Perm}(\Sigma) \;:\; \pi(\texttt{F})\pi(\texttt{A})\pi(\texttt{R}) = \texttt{XYY} \,\}|}{|\mathsf{Perm}(\Sigma)|} \\
&= \frac{0}{26!} \\
&= 0\,.
\end{aligned}$$

In this case, the numerator asks us to count the number of permutations π with the property that $\pi(\texttt{F}) = \texttt{X}$, $\pi(\texttt{A}) = \texttt{Y}$ and $\pi(\texttt{R}) = \texttt{Y}$. But no permutation can have the same output $\texttt{Y}$ on two different inputs. So the number of permutations meeting this condition is zero.

In conclusion, we have Equation (2.2) because the two probabilities we computed above are different. ▌

Let us now show that the OTP scheme with key-length m does have the perfect security property. Intuitively, the reason is as follows. Say $m = 3$, and consider two messages, say $M_1 = 010$ and $M_2 = 001$. Say the adversary receives the ciphertext $C = 101$. It asks itself whether it was M_1 or M_2 that was encrypted. Well, it reasons, if it was M_1, then the key must have been $K = M_1 \oplus C = 111$, while if M_2 was encrypted then the key must have been $K = M_2 \oplus C = 100$. But either of these two was equally likely as the key, so how do I know which of the two it was? Here now is the formal statement and proof.

Claim 2.2.3 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the OTP scheme with key-length $m \geq 1$. Assume we use it to encrypt a single message drawn from $\{0,1\}^m$. Then $\mathcal{SE}$ is perfectly secure. ▌

Proof of Claim 2.2.3: As per Definition 2.2.1, for any $M_1, M_2 \in \{0,1\}^m$ and any C we need to show that Equation (2.1) is true. So let M_1, M_2 be any m-bit strings. We can assume C is also an m-bit string, since otherwise both sides of Equation (2.1) are zero and thus equal. Now

$$\begin{aligned}
\Pr\left[\mathcal{E}_K(M_1) = C\right] &= \Pr\left[K \oplus M_1 = C\right] \\
&= \frac{|\{\, K \in \{0,1\}^m \;:\; K \oplus M_1 = C \,\}|}{|\{0,1\}^m|} \\
&= \frac{1}{2^m}\,.
\end{aligned}$$

Above, the probability is over the random choice of K from $\{0,1\}^m$, with M_1, C fixed. We write the probability as the ratio of two terms: the first is the number of keys K for which $K \oplus M_1 = C$, and the second is the total possible number of keys. The first term is one, because K can only be the string $M_1 \oplus C$, while the second term is 2^m. Similarly we have

$$\begin{aligned} \Pr\left[\mathcal{E}_K(M_2) = C\right] &= \Pr\left[K \oplus M_2 = C\right] \\ &= \frac{|\{\, K \in \{0,1\}^m \;:\; K \oplus M_2 = C \,\}|}{|\{0,1\}^m|} \\ &= \frac{1}{2^m}\,. \end{aligned}$$

In this case the numerator of the fraction is one because only the key $K = M_2 \oplus C$ has the property that $K \oplus M_2 = C$. Now, since the two probabilities we have computed above are equal, Equation (2.1) is true, and thus our proof is complete. ■

Perfect security is great in terms of security, but comes at a hefty price. It turns out that in any perfectly secure scheme, the length of the key must be at least the length of the (single) message encrypted. This means that in practice a perfectly secure scheme (like the OTP) is prohibitively expensive, requiring parties to exchange very long keys before they can communicate securely.

In practice we want parties to be able to hold a short key, for example 128 bits, and then be able to securely encrypt essentially any amount of data. To achieve this, we need to make a switch regarding what kinds of security attributes we seek. As we discussed in the Introduction, we will ask for security that is not perfect but good enough, the latter interpreted in a computational sense. Visualizing an adversary as someone running programs to break our system, we will say something like, yes, in principle you can break my scheme, but it would take more than 100 years running on the world's fastest computers to break it with a probability greater than 2^{-60}. In practice, this is good enough.

To get schemes like that we need some tools, and this is what we turn to next.

2.3 Problems

Problem 6 Suppose that you want to encrypt a single message $M \in \{0,1,2\}$ using a random shared key $K \in \{0,1,2\}$. Suppose you do this by representing K and M using two bits (00, 01, or 10), and then XORing the two representations. Does this seem like a good protocol to you? Explain. ▮

Problem 7 Suppose that you want to encrypt a single message $M \in \{0,1,2\}$ using a random shared key $K \in \{0,1,2\}$. Explain a good way to do this. ▮

Problem 8 *Symmetric encryption with a deck of cards.* Alice shuffles a deck of cards and deals it all out to herself and Bob (each of them gets half of the 52 cards). Alice now wishes to send a secret message M to Bob by saying something aloud. Eavesdropper Eve is listening in: she hears everything Alice says (but Eve can't see the cards).

Part A. Suppose Alice's message M is a string of 48-bits. Describe how Alice can communicate M to Bob in such a way that Eve will have *no* information about what is M.

Part B. Now suppose Alice's message M is 49 bits. Prove that there exists no protocol which allows Alice to communicate M to Bob in such a way that Eve will have no information about M.

Chapter 3

Blockciphers

Blockciphers are the central tool in the design of protocols for shared-key cryptography (aka. symmetric) cryptography. They are the main available "technology" we have at our disposal. This chapter will take a look at these objects and describe the state of the art in their construction.

It is important to stress that blockciphers are just tools—raw ingredients for cooking up something more useful. Blockciphers don't, by themselves, do something that an end-user would care about. As with any powerful tool, one has to learn to use this one. Even an excellent blockcipher won't give you security if you use don't use it right. But used well, these are powerful tools indeed. Accordingly, an important theme in several upcoming chapters will be on how to use blockciphers well. We won't be emphasizing how to design or analyze blockciphers, as this remains very much an art.

This chapter gets you acquainted with some typical blockciphers, and discusses attacks on them. In particular we'll look at two examples, DES and AES. DES is the "old standby." It is currently the most widely-used blockcipher in existence, and it is of sufficient historical significance that every trained cryptographer needs to have seen its description. AES is a modern blockcipher, and it is expected to supplant DES in the years to come.

3.1 What is a blockcipher?

A blockcipher is a function $E\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. This notation means that E takes two inputs, one being a k-bit string and the other an n-bit string, and returns an n-bit string. The first input is the key. The second might be called the plaintext, and the output might be called a ciphertext. The *key-length* k and the *block-length* n are parameters associated to the blockcipher. They vary from blockcipher to blockcipher, as of course does the design of the algorithm itself.

For each key $K \in \{0,1\}^k$ we let $E_K\colon \{0,1\}^n \to \{0,1\}^n$ be the function defined by $E_K(M) = E(K,M)$. For any blockcipher, and any key K, it is required that the function E_K be a *permutation* on $\{0,1\}^n$. This means that it is a bijection (ie., a one-to-one and onto function) of $\{0,1\}^n$ to $\{0,1\}^n$. (For every $C \in \{0,1\}^n$ there is exactly one $M \in \{0,1\}^n$ such that $E_K(M) = C$.) Accordingly E_K has an inverse, and we denote it E_K^{-1}. This function also maps $\{0,1\}^n$ to $\{0,1\}^n$, and of course we have $E_K^{-1}(E_K(M)) = M$ and $E_K(E_K^{-1}(C)) = C$ for all $M, C \in \{0,1\}^n$. We let $E^{-1}\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be defined by $E^{-1}(K,C) = E_K^{-1}(C)$. This is the inverse blockcipher to E.

Preferably, the blockcipher E is a public specified algorithm. Both the cipher E and its inverse E^{-1} should be easily computable, meaning given K, M we can readily compute $E(K,M)$, and given

K, C we can readily compute $E^{-1}(K, C)$. By "readily compute" we mean that there are public and relatively efficient programs available for these tasks.

In typical usage, a random key K is chosen and kept secret between a pair of users. The function E_K is then used by the two parties to process data in some way before they send it to each other. Typically, we will assume the adversary will be able to obtain some input-output examples for E_K, meaning pairs of the form (M, C) where $C = E_K(M)$. But, ordinarily, the adversary will not be shown the key K. Security relies on the secrecy of the key. So, as a first cut, you might think of the adversary's goal as recovering the key K given some input-output examples of E_K. The blockcipher should be designed to make this task computationally difficult. (Later we will refine the view that the adversary's goal is key-recovery, seeing that security against key-recovery is a necessary but not sufficient condition for the security of a blockcipher.)

We emphasize that we've said absolutely nothing about what properties a blockcipher should have. A function like $E_K(M) = M$ is a blockcipher (the "identity blockcipher"), but we shall not regard it as a "good" one.

How do real blockciphers work? Lets take a look at some of them to get a sense of this.

3.2 Data Encryption Standard (DES)

The Data Encryption Standard (DES) is the quintessential blockcipher. Even though it is now quite old, and on the way out, no discussion of blockciphers can really omit mention of this construction. DES is a remarkably well-engineered algorithm which has had a powerful influence on cryptography. It is in very widespread use, and probably will be for some years to come. Every time you use an ATM machine, you are using DES.

3.2.1 A brief history

In 1972 the NBS (National Bureau of Standards, now NIST, the National Institute of Standards and Technology) initiated a program for data protection and wanted as part of it an encryption algorithm that could be standardized. They put out a request for such an algorithm. In 1974, IBM responded with a design based on their "Lucifer" algorithm. This design would eventually evolve into the DES.

DES has a key-length of $k = 56$ bits and a block-length of $n = 64$ bits. It consists of 16 rounds of what is called a "Feistel network." We will describe more details shortly.

After NBS, several other bodies adopted DES as a standard, including ANSI (the American National Standards Institute) and the American Bankers Association.

The standard was to be reviewed every five years to see whether or not it should be re-adopted. Although there were claims that it would not be re-certified, the algorithm was re-certified again and again. Only recently did the work for finding a replacement begin in earnest, in the form of the AES (Advanced Encryption Standard) effort.

3.2.2 Construction

The DES algorithm is depicted in Fig. 3.1. It takes input a 56-bit key K and a 64 bit plaintext M. The key-schedule *KeySchedule* produces from the 56-bit key K a sequence of 16 subkeys, one for each of the rounds that follows. Each subkey is 48-bits long. We postpone the discussion of the *KeySchedule* algorithm.

The initial permutation *IP* simply permutes the bits of M, as described by the table of Fig. 3.2. The table says that bit 1 of the output is bit 58 of the input; bit 2 of the output is bit 50 of the

```
function $\mathsf{DES}_K(M)$    // $|K| = 56$ and $|M| = 64$
    $(K_1, \ldots, K_{16}) \leftarrow KeySchedule(K)$    // $|K_i| = 48$ for $1 \leq i \leq 16$
    $M \leftarrow IP(M)$
    Parse $M$ as $L_0 \parallel R_0$    // $|L_0| = |R_0| = 32$
    for $r = 1$ to 16 do
        $L_r \leftarrow R_{r-1}$ ; $R_r \leftarrow f(K_r, R_{r-1}) \oplus L_{r-1}$
    $C \leftarrow IP^{-1}(L_{16} \parallel R_{16})$
    return $C$
```

Figure 3.1: The DES blockcipher. The text and other figures describe the subroutines $KeySchedule, f, IP, IP^{-1}$.

IP

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

IP^{-1}

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

Figure 3.2: Tables describing the DES initial permutation IP and its inverse IP^{-1}.

input; ... ; bit 64 of the output is bit 7 of the input. Note that the key is not involved in this permutation. The initial permutation does not appear to affect the cryptographic strength of the algorithm. It might have been included to slow-down software implementations.

The permuted plaintext is now input to a loop, which operates on it in 16 rounds. Each round takes a 64-bit input, viewed as consisting of a 32-bit left half and a 32-bit right half, and, under the influence of the sub-key K_r, produces a 64-bit output. The input to round r is $L_{r-1} \parallel R_{r-1}$, and the output of round r is $L_r \parallel R_r$. Each round is what is called a Feistel round, named after Horst Feistel, one the IBM designers of a precursor of DES. Fig. 3.1 shows how it works, meaning how $L_r \parallel R_r$ is computed as a function of $L_{r-1} \parallel R_{r-1}$, by way of the function f, the latter depending on the sub-key K_r associated to the r-th round.

One of the reasons to use this round structure is that it is reversible, important to ensure that DES_K is a permutation for each key K, as it should be to qualify as a blockcipher. Indeed, given $L_r \parallel R_r$ (and K_r) we can recover $L_{r-1} \parallel R_{r-1}$ via $R_{r-1} \leftarrow L_r$ and $L_{r-1} \leftarrow f(K_r, L_r) \oplus R_r$.

Following the 16 rounds, the inverse of the permutation IP, also depicted in Fig. 3.2, is applied to the 64-bit output of the 16-th round, and the result of this is the output ciphertext.

A sequence of Feistel rounds is a common high-level design for a blockcipher. For a closer look we need to see how the function $f(\cdot, \cdot)$ works. It is shown in Fig. 3.3. It takes a 48-bit subkey J and a 32-bit input R to return a 32-bit output. The 32-bit R is first expanded into a 48-bit via the function E described by the table of Fig. 3.4. This says that bit 1 of the output is bit 32 of the input; bit 2 of the output is bit 1 of the input; ... ; bit 48 of the output is bit 1 of the input.

Note the E function is quite structured. In fact barring that 1 and 32 have been swapped (see top left and bottom right) it looks almost sequential. Why did they do this? Who knows. That's the answer to most things about DES.

```
function $f(J, R)$    // $|J| = 48$ and $|R| = 32$
    $R \leftarrow E(R)$ ; $R \leftarrow R \oplus J$
    Parse $R$ as $R_1 \| R_2 \| R_3 \| R_4 \| R_5 \| R_6 \| R_7 \| R_8$    // $|R_i| = 6$ for $1 \le i \le 8$
    for $i = 1, \ldots, 8$ do
        $R_i \leftarrow \mathbf{S}_i(R_i)$    // Each S-box returns 4 bits
    $R \leftarrow R_1 \| R_2 \| R_3 \| R_4 \| R_5 \| R_6 \| R_7 \| R_8$    // $|R| = 32$ bits
    $R \leftarrow P(R)$
    return $R$
```

Figure 3.3: The f-function of DES. The text and other figures describe the subroutines used.

E

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

P

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

Figure 3.4: Tables describing the expansion function E and final permutation P of the DES f-function.

Now the sub-key J is XORed with the output of the E function to yield a 48-bit result that we continue to denote by R. This is split into 8 blocks, each 6-bits long. To the i-th block we apply the function $\mathbf{S}_i$ called the i-th S-box. Each S-box is a function taking 6 bits and returning 4 bits. The result is that the 48-bit R is compressed to 32 bits. These 32 bits are permuted according to the P permutation described in the usual way by the table of Fig. 3.4, and the result is the output of the f function. Let us now discuss the S-boxes.

Each S-box is described by a table as shown in Fig. 3.5. Read these tables as follows. $\mathbf{S}_i$ takes a 6-bit input. Write it as $b_1b_2b_3b_4b_5b_6$. Read $b_3b_4b_5b_6$ as an integer in the range $0, \ldots, 15$, naming a column in the table describing $\mathbf{S}_i$. Let b_1b_2 name a row in the table describing $\mathbf{S}_i$. Take the row b_1b_2, column $b_3b_4b_5b_6$ entry of the table of $\mathbf{S}_i$ to get an integer in the range $0, \ldots, 15$. The output of $\mathbf{S}_i$ on input $b_1b_2b_3b_4b_5b_6$ is the 4-bit string corresponding to this table entry.

The S-boxes are the heart of the algorithm, and much effort was put into designing them to achieve various security goals and resistance to certain attacks.

Finally, we discuss the key schedule. It is shown in Fig. 3.6. Each round sub-key K_r is formed by taking some 48 bits of K. Specifically, a permutation called PC-1 is first applied to the 56-bit key to yield a permuted version of it. This is then divided into two 28-bit halves and denoted $C_0 \| D_0$. The algorithm now goes through 16 rounds. The r-th round takes input $C_{r-1} \| D_{r-1}$, computes $C_r \| D_r$, and applies a function PC-2 that extracts 48 bits from this 56-bit quantity. This is the sub-key K_r for the r-th round. The computation of $C_r \| D_r$ is quite simple. The bits of C_{r-1} are rotated to the left j positions to get C_r, and D_r is obtained similarly from D_{r-1}, where j is either 1 or 2, depending on r.

The functions PC-1 and PC-2 are tabulated in Fig. 3.7. The first table needs to be read in a strange way. It contains 56 integers, these being all integers in the range $1, \ldots, 64$ barring multiples

$\mathbf{S}_1$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
1	0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
1	1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

$\mathbf{S}_2$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
0	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
1	0	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
1	1	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

$\mathbf{S}_3$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
0	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
1	0	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	1	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

$\mathbf{S}_4$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
0	1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
1	0	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
1	1	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

$\mathbf{S}_5$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
0	1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
1	0	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
1	1	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

$\mathbf{S}_6$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
0	1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
1	0	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
1	1	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

$\mathbf{S}_7$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
0	1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	0	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
1	1	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

$\mathbf{S}_8$:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
0	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
1	0	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
1	1	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 3.5: The DES S-boxes.

---

of 8. Given a 56-bit string $K = K[1] \ldots K[56]$ as input, the corresponding function returns the 56-bit string $L = L[1] \ldots L[56]$ computed as follows. Suppose $1 \leq i \leq 56$, and let a be the i-th entry of the table. Write $a = 8q + r$ where $1 \leq r \leq 7$. Then let $L[i] = K[a - q]$. As an example, let us determine the first bit, $L[1]$, of the output of the function on input K. We look at the first entry in the table, which is 57. We divide it by 8 to get $57 = 8(7) + 1$. So $L[1]$ equals $K[57 - 7] = K[50]$, meaning the 1st bit of the output is the 50-th bit of the input. On the other hand PC-2 is read in the usual way as a map taking a 56-bit input to a 48 bit output: bit 1 of the output is bit 14 of the input; bit 2 of the output is bit 17 of the input; … ; bit 56 of the output is bit 32 of the input.

```
Algorithm $KeySchedule(K)$     // $|K| = 56$
    $K \leftarrow PC\text{-}1(K)$
    Parse $K$ as $C_0 \,\|\, D_0$
    for $r = 1, \ldots, 16$ do
        if $r \in \{1, 2, 9, 16\}$ then $j \leftarrow 1$ else $j \leftarrow 2$ fi
        $C_r \leftarrow leftshift_j(C_{r-1})$ ; $D_r \leftarrow leftshift_j(D_{r-1})$
        $K_r \leftarrow PC\text{-}2(C_r \,\|\, D_r)$
    return $(K_1, \ldots, K_{16})$
```

Figure 3.6: The key schedule of DES. Here $leftshift_j$ denotes the function that rotates its input to the left by j positions.

PC-1

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

PC-2

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

Figure 3.7: Tables describing the *PC*-1 and *PC*-2 functions used by the DES key schedule of Fig. 3.6.

Well now you know how DES works. Of course, the main questions about the design are: why, why and why? What motivated these design choices? We don't know too much about this, although we can guess a little. And one of the designers of DES, Don Coppersmith, has written a short paper which provides some information.

3.2.3 Speed

One of the design goals of DES was that it would have fast implementations relative to the technology of its time. How fast can you compute DES? In roughly current technology (well, nothing is current by the time one writes it down!) one can get well over 1 Gbit/sec on high-end VLSI. Specifically at least 1.6 Gbits/sec, maybe more. That's pretty fast. Perhaps a more interesting figure is that one can implement each DES S-box with at most 50 two-input gates, where the circuit has depth of only 3. Thus one can compute DES by a combinatorial circuit of about $8 \cdot 16 \cdot 50 = 640$ gates and depth of $3 \cdot 16 = 48$ gates.

In software, on a fairly modern processor, DES takes something like 80 cycles per byte. This is disappointingly slow—not surprisingly, since DES was optimized for hardware and was designed before the days in which software implementations were considered feasible or desirable.

3.3 Key recovery attacks on blockciphers

Now that we know what a blockcipher looks like, let us consider attacking one. This is called cryptanalysis of the blockcipher.

We fix a blockcipher E: $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ having key-size k and block size n. It is assumed that the attacker knows the description of E and can compute it. For concreteness, you can think of E as being DES.

Historically, cryptanalysis of blockciphers has focused on key-recovery. The cryptanalyst may think of the problem to be solved as something like this. A k-bit key T, called the target key, is chosen at random. Let $q \geq 0$ be some integer parameter.

GIVEN: The adversary has a sequence of q input-output examples of E_T, say

$$(M_1, C_1), \ldots, (M_q, C_q)$$

where $C_i = E_T(M_i)$ for $i = 1, \ldots, q$ and $M_1, \ldots, M_q$ are all distinct n-bit strings.

FIND: The adversary wants to find the target key T.

Let us say that a key K is *consistent with the input-output examples* $(M_1, C_1), \ldots, (M_q, C_q)$ if $E_K(M_i) = C_i$ for all $1 \leq i \leq q$. We let

$$\mathsf{Cons}_E((M_1, C_1), \ldots, (M_q, C_q))$$

be the set of all keys consistent with the input-output examples $(M_1, C_1), \ldots, (M_q, C_q)$. Of course the target key T is in this set. But the set might be larger, containing other keys. Without asking further queries, a key-recovery attack cannot hope to differentiate the target key from other members of $\mathsf{Cons}_E((M_1, C_1), \ldots, (M_q, C_q))$. Thus, the goal is sometimes viewed as simply being to find some key in this set. For practical blockciphers we expect that, if a few input-output examples are used, the size of the above set will be one, so the adversary can indeed find the target key. We will exemplify this when we consider specific attacks.

Some typical kinds of "attack" that are considered within this framework:

KNOWN-MESSAGE ATTACK: $M_1, \ldots, M_q$ are any distinct points; the adversary has no control over them, and must work with whatever it gets.

CHOSEN-MESSAGE ATTACK: $M_1, \ldots, M_q$ are chosen by the adversary, perhaps even adaptively. That is, imagine it has access to an "oracle" for the function E_K. It can feed the oracle M_1 and get back $C_1 = E_K(M_1)$. It can then decide on a value M_2, feed the oracle this, and get back C_2, and so on.

Clearly a chosen-message attack gives the adversary more power, but it may be less realistic in practice.

The most obvious attack strategy is exhaustive key search. The adversary goes through all possible keys $K' \in \{0,1\}^k$ until it finds one that explains the input-output pairs. Here is the attack in detail, using $q = 1$, meaning one input-output example. For $i = 1, \ldots, 2^k$ let T_i denote the i-th k-bit string (in lexicographic order).

```
algorithm $A_E^{\text{eks}}(M_1, C_1)$
    for $i = 1, \ldots, 2^k$ do
        if $E_{T_i}(M_1) = C_1$ then return $T_i$
```

This attack always returns a key consistent with the given input-output example (M_1, C_1). Whether or not it is the target key depends on the blockcipher. If one imagines the blockcipher to be random, then the blockcipher's key length and block length are relevant in assessing if the above attack will find the "right" key. , The likelihood of the attack returning the target key can be increased by testing against more input-output examples:

```
algorithm $A_E^{\text{eks}}((M_1, C_1), \ldots, (M_q, C_q))$
    for $i = 1, \ldots, 2^k$ do
        if $E(T_i, M_1) = C_1$ then
            if ( $E(T_i, M_2) = C_2$ and $\cdots$ and $E(T_i, M_q) = C_q$ ) then return $T_i$
```

A fairly small vaue of q, say somewhat more than k/n, is enough that this attack will usually return the target key itself. For DES, $q = 1$ or $q = 2$ seems to be enough.

Thus, no blockcipher is perfectly secure. It is always possible for an attacker to recover a consistent key. A good blockcipher, however, is designed to make this task computationally prohibitive.

How long does exhaustive key-search take? Since we will choose q to be small we can neglect the difference in running time between the two versions of the attack above, and focus for simplicity on the first attack. In the worst case, it uses 2^k computations of the blockcipher. However it could be less since one could get lucky. For example if the target key is in the first half of the search space, only 2^{k-1} computations would be used. So a better measure is how long it takes on the average. This is

$$\sum_{i=1}^{2^k} i \cdot \Pr[K = T_i] \;=\; \sum_{i=1}^{2^k} \frac{i}{2^k} \;=\; \frac{1}{2^k} \cdot \sum_{i=1}^{2^k} i \;=\; \frac{1}{2^k} \cdot \frac{2^k(2^k+1)}{2} \;=\; \frac{2^k+1}{2} \approx 2^{k-1}$$

computations of the blockcipher. This is because the target key is chosen at random, so with probability $1/2^k$ equals T_i, and in that case the attack uses i E-computations to find it.

Thus to make key-recovery by exhaustive search computationally prohibitive, one must make the key-length k of the blockcipher large enough.

Let's look at DES. We noted above that there is VLSI chip that can compute it at the rate of 1.6 Gbits/sec. How long would key-recovery via exhaustive search take using this chip? Since a DES plaintext is 64 bits, the chip enables us to perform $(1.6 \cdot 10^9)/64 = 2.5 \cdot 10^7$ DES computations per second. To perform 2^{55} computations (here $k = 56$) we thus need $2^{55}/(2.5 \cdot 10^7) \approx 1.44 \cdot 10^9$ seconds, which is about 45.7 years. This is clearly prohibitive.

It turns out that that DES has a property called key-complementation that one can exploit to reduce the size of the search space by one-half, so that the time to find a key by exhaustive search comes down to 22.8 years. But this is still prohibitive.

Yet, the conclusion that DES is secure against exhaustive key search is actually too hasty. We will return to this later and see why.

Exhaustive key search is a generic attack in the sense that it works against any blockcipher. It only involves computing the blockcipher and makes no attempt to analyze the cipher and find and exploit weaknesses. Cryptanalysts also need to ask themselves if there is some weakness in the structure of the blockcipher they can exploit to obtain an attack performing better than exhaustive key search.

For DES, the discovery of such attacks waited until 1990. Differential cryptanalysis is capable of finding a DES key using about 2^{47} input-output examples (that is, $q = 2^{47}$) in a chosen-message attack [1, 2]. Linear cryptanalysis [4] improved differential in two ways. The number of input-output examples required is reduced to 2^{44}, and only a known-message attack is required. (An alternative version uses 2^{42} chosen plaintexts [24].)

These were major breakthroughs in cryptanalysis that required careful analysis of the DES construction to find and exploit weaknesses. Yet, the practical impact of these attacks is small. Why? Ordinarily it would be impossible to obtain 2^{44} input-output examples. Furthermore, the storage requirement for these examples is prohibitive. A single input-output pair, consisting of a 64-bit plaintext and 64-bit ciphertext, takes 16 bytes of storage. When there are 2^{44} such pairs, we need $16 \cdot 2^{44} = 2.81 \cdot 10^{14}$ bits, or about 281 terabytes of storage, which is enormous.

Linear and differential cryptanalysis were however more devastating when applied to other ciphers, some of which succumbed completely to the attack.

So what's the best possible attack against DES? The answer is exhaustive key search. What we ignored above is that the DES computations in this attack can be performed in parallel. In 1993, Weiner argued that one can design a $1 million machine that does the exhaustive key search for DES in about 3.5 hours on the average [7]. His machine would have about 57,000 chips, each performing numerous DES computations. More recently, a DES key search machine was actually built by the Electronic Frontier Foundation, at a cost of $250,000 [5]. It finds the key in 56 hours, or about 2.5 days on the average. The builders say it will be cheaper to build more machines now that this one is built.

Thus DES is feeling its age. Yet, it would be a mistake to take away from this discussion the impression that DES is a weak algorithm. Rather, what the above says is that it is an impressively strong algorithm. After all these years, the best practical attack known is still exhaustive key search. That says a lot for its design and its designers.

Later we will see that we would like security properties from a blockcipher that go beyond resistance to key-recovery attacks. It turns out that from that point of view, a limitation of DES is its block size. Birthday attacks "break" DES with about $q = 2^{32}$ input output examples. (The meaning of "break" here is very different from above.) Here 2^{32} is the square root of 2^{64}, meaning to resist these attacks we must have bigger block size. The next generation of ciphers—things like AES—took this into account.

3.4 Iterated-DES and DESX

The emergence of the above-discussed key-search engines lead to the view that in practice DES should be considered broken. Its shortcoming was its key-length of 56, not long enough to resist exhaustive key search.

People looked for cheap ways to strengthen DES, turning it, in some simple way, into a cipher with a larger key length. One paradigm towards this end is iteration.

3.4.1 Double-DES

Let K_1, K_2 be 56-bit DES keys and let M be a 64-bit plaintext. Let

$$\mathsf{2DES}(K_1 \,\|\, K_2, M) \;=\; \mathsf{DES}(K_2, \mathsf{DES}(K_1, M)) \;.$$

This defines a blockcipher 2DES: $\{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}$ that we call *Double-DES*. It has a 112-bit key, viewed as consisting of two 56-bit DES keys. Note that it is reversible, as required to be a blockcipher:

$$\mathsf{2DES}^{-1}(K_1 \,\|\, K_2, C) \;=\; \mathsf{DES}^{-1}(K_1, \mathsf{DES}^{-1}(K_2, C)) \;.$$

for any 64-bit C.

The key length of 112 is large enough that there seems little danger of 2DES succumbing to an exhaustive key search attack, even while exploiting the potential for parallelism and special-purpose hardware. On the other hand, 2DES also seems secure against the best known cryptanalytic techniques, namely differential and linear cryptanalysis, since the iteration effectively increases the number of Feistel rounds. This would indicate that 2DES is a good way to obtain a DES-based cipher more secure than DES itself.

However, although 2DES has a key-length of 112, it turns out that it can be broken using about 2^{57} DES and DES^{-1} computations by what is called a meet-in-the-middle attack, as we now illustrate. Let $K_1 \| K_2$ denote the target key and let $C_1 = \mathsf{2DES}(K_1 \| K_2, M_1)$. The attacker, given M_1, C_1, is attempting to find $K_1 \| K_2$. We observe that

$$C_1 = \mathsf{DES}(K_2, \mathsf{DES}(K_1, M_1)) \quad \Rightarrow \quad \mathsf{DES}^{-1}(K_2, C_1) = \mathsf{DES}(K_1, M_1) \ .$$

This leads to the following attack. Below, for $i = 1, \ldots, 2^{56}$ we let T_i denote the i-th 56-bit string (in lexicographic order):

```
$A^{\rm MinM}_{\mathsf{2DES}}(M_1, C_1)$
    for $i = 1, \ldots, 2^{56}$ do $L[i] \leftarrow \mathsf{DES}(T_i, M_1)$
    for $j = 1, \ldots, 2^{56}$ do $R[j] \leftarrow \mathsf{DES}^{-1}(T_j, C_1)$
    $S \leftarrow \{ (i,j) \ : \ L[i] = R[j] \}$
    Pick some $(l, r) \in S$ and return $T_l \| T_r$
```

For any $(i,j) \in S$ we have

$$\mathsf{DES}(T_i, M_1) = L[i] = R[j] = \mathsf{DES}^{-1}(T_j, C_1)$$

and as a consequence $\mathsf{DES}(T_j, \mathsf{DES}(T_i, M_1)) = C_1$. So the key $T_i \| T_j$ is consistent with the input-output example (M_1, C_1). Thus,

$$\{ T_l \| T_r \ : \ (l,r) \in S \} \ = \ \mathsf{Cons}_E((M_1, C_1)) \ .$$

The attack picks some pair (l, r) from S and outputs $T_l \| T_r$, thus returning a key consistent with the input-output example (M_1, C_1).

The set S above is likely to be quite large, of size about $2^{56+56}/2^{64} = 2^{48}$, meaning the attack as written is not likely to return the target key itself. However, by using a few more input-output examples, it is easy to whittle down the choices in the set S until it is likely that only the target key remains.

The attack makes $2^{56} + 2^{56} = 2^{57}$ DES or DES^{-1} computations. The step of forming the set S can be implemented in linear time in the size of the arrays involved, say using hashing. (A naive strategy takes time quadratic in the size of the arrays.) Thus the running time is dominated by the $\mathsf{DES}, \mathsf{DES}^{-1}$ computations.

The meet-in-the-middle attack shows that $\mathsf{2DES}$ is quite far from the ideal of a cipher where the best attack is exhaustive key search. However, this attack is not particularly practical, even if special purpose machines are designed to implement it. The machines could do the $\mathsf{DES}, \mathsf{DES}^{-1}$ computations quickly in parallel, but to form the set S the attack needs to store the arrays L, R, each of which has 2^{56} entries, each entry being 64 bits. The amount of storage required is $8 \cdot 2^{57} \approx 1.15 \cdot 10^{18}$ bytes, or about $1.15 \cdot 10^6$ terabytes, which is so large that implementing the attack is impractical.

There are some strategies that modify the attack to reduce the storage overhead at the cost of some added time, but still the attack does not appear to be practical.

Since a 112-bit $\mathsf{2DES}$ key can be found using 2^{57} DES or DES^{-1} computations, we sometimes say that $\mathsf{2DES}$ has an *effective key length* of 57.

3.4.2 Triple-DES

The triple-DES ciphers use three iterations of DES or DES^{-1}. The three-key variant is defined by

$$\mathsf{3DES3}(K_1 \| K_2 \| K_3, M) \ = \ \mathsf{DES}(K_3, \mathsf{DES}^{-1}(K_2, \mathsf{DES}(K_1, M)) \ ,$$

so that $\mathsf{3DES3}\colon \ \{0,1\}^{168} \times \{0,1\}^{64} \to \{0,1\}^{64}$. The two-key variant is defined by

$$\mathsf{3DES2}(K_1 \| K_2, M) \ = \ \mathsf{DES}(K_2, \mathsf{DES}^{-1}(K_1, \mathsf{DES}(K_2, M)) \ ,$$

so that $\mathsf{3DES2}\colon \{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}$. You should check that these functions are reversible so that they do qualify as blockciphers. The term "triple" refers to there being three applications of DES or DES^{-1}. The rationale for the middle application being DES^{-1} rather than DES is that DES is easily recovered via

$$\mathsf{DES}(K, M) = \mathsf{3DES3}(K \,\|\, K \,\|\, K, M) \tag{3.1}$$

$$\mathsf{DES}(K, M) = \mathsf{3DES2}(K \,\|\, K, M)\ . \tag{3.2}$$

As with $\mathsf{2DES}$, the key length of these ciphers appears long enough to make exhaustive key search prohibitive, even with the best possible engines, and, additionally, differential and linear cryptanalysis are not particularly effective because iteration effectively increases the number of Feistel rounds.

$\mathsf{3DES3}$ is subject to a meet-in-the-middle attack that finds the 168-bit key using about 2^{112} computations of DES or DES^{-1}, so that it has an effective key length of 112. There does not appear to be a meet-in-the-middle attack on $\mathsf{3DES2}$ however, so that its key length of 112 is also its effective key length.

The $\mathsf{3DES2}$ cipher is popular in practice and functions as a canonical and standard replacement for DES. $\mathsf{2DES}$, although having the same effective key length as $\mathsf{3DES2}$ and offering what appears to be the same or at least adequate security, is not popular in practice. It is not entirely apparent why $\mathsf{3DES2}$ is preferred over $\mathsf{2DES}$, but the reason might be Equation (3.2).

3.4.3 DESX

Although $\mathsf{2DES}, \mathsf{3DES3}$ and $\mathsf{3DES2}$ appear to provide adequate security, they are slow. The first is twice as slow as DES and the other two are three times as slow. It would be nice to have a DES based blockcipher that had a longer key than DES but was not significantly more costly. Interestingly, there is a simple design that does just this. Let K be a 56-bit DES key, let K_1, K_2 be 64-bit strings, and let M be a 64-bit plaintext. Let

$$\mathsf{DESX}(K \,\|\, K_1 \,\|\, K_2, M) = K_2 \oplus \mathsf{DES}(K, K_1 \oplus M)\ .$$

This defines a blockcipher $\mathsf{DESX}\colon \{0,1\}^{184} \times \{0,1\}^{64} \to \{0,1\}^{64}$. It has a 184-bit key, viewed as consisting of a 56-bit DES key plus two auxiliary keys, each 64 bits long. Note that it is reversible, as required to be a blockcipher:

$$\mathsf{DESX}^{-1}(K \,\|\, K_1 \,\|\, K_2, C) = K_1 \oplus \mathsf{DES}^{-1}(K, K_2 \oplus C)\ .$$

The key length of 184 is certainly enough to preclude exhaustive key search attacks. DESX is no more secure than DES against linear of differential cryptanalysis, but we already saw that these are not really practical attacks.

There is a meet-in-the-middle attack on DESX. It finds a 184-bit DESX key using 2^{120} DES and DES^{-1} computations. So the effective key length of DESX seems to be 120, which is large enough for security.

DESX is less secure than Double or Triple DES because the latter are more more resistant than DES to linear and differential cryptanalysis while DESX is only as good as DES itself in this regard. However, this is good enough; we saw that in practice the weakness of DES was not these attacks but rather the short key length leading to successful exhaustive search attacks. DESX fixes this, and very cheaply. In summary, DESX is popular because it is much cheaper than Double of Triple DES while providing adequate security.

3.4.4 Why a new cipher?

DESX is arguably a fine cipher. Nonetheless, there were important reasons to find and standardize a new cipher.

We will see later that the security provided by a blockcipher depends not only on its key length and resistance to key-search attacks but on its block length. A blockcipher with block length n can be "broken" in time around $2^{n/2}$. When $n = 64$, this is 2^{32}, which is quite small. Although $\mathsf{2DES}, \mathsf{3DES3}, \mathsf{3DES2}, \mathsf{DESX}$ have a higher (effective) key length than DES, they preserve its block size and thus are no more secure than DES from this point of view. It was seen as important to have a blockcipher with a block length n large enough that a $2^{n/2}$ time attack was not practical. This was one motivation for a new cipher.

Perhaps the larger motivation was speed. Desired was a blockcipher that ran faster than DES in software.

3.5 Advanced Encryption Standard (AES)

In 1998 the National Institute of Standards and Technology (NIST/USA) announced a "competition" for a new blockcipher. The new blockcipher would, in time, replace DES. The relatively short key length of DES was the main problem that motivated the effort: with the advances in computing power, a key space of 2^{56} keys was just too small. With the development of a new algorithm one could also take the opportunity to address the modest software speed of DES, making something substantially faster, and to increase the block size from 64 to 128 bits (the choice of 64 bits for the block size can lead to security difficulties, as we shall later see. Unlike the design of DES, the new algorithm would be designed in the open and by the public.

Fifteen algorithms were submitted to NIST. They came from around the world. A second round narrowed the choice to five of these algorithms. In the summer of 2001 NIST announced their choice: an algorithm called Rijndael. The algorithm should be embodied in a NIST FIPS (Federal Information Processing Standard) any day now; right now, there is a draft FIPS. Rijndael was designed by Joan Daemen and Vincent Rijmen (from which the algorithm gets its name), both from Belgium. It is descendent of an algorithm called Square.

In this section we shall describe AES.

A word about notation. Purists would prefer to reserve the term "AES" to refer to the standard, using the word "Rijndael" or the phrase "the AES algorithm" to refer to the algorithm itself. (The same naming pundits would have us use the acronym DEA, Data Encryption Algorithm, to refer to the algorithm of the DES, the Data Encryption Standard.) We choose to follow common convention and refer to both the standard and the algorithm as AES. Such an abuse of terminology never seems to lead to any misunderstandings. (Strictly speaking, AES is a special case of Rijndael. The latter includes more options for block lengths than AES does.)

The AES has a block length of $n = 128$ bits, and a key length k that is variable: it may be 128, 192 or 256 bits. So the standard actually specifies three different blockciphers: AES128, AES192, AES256. These three blockciphers are all very similar, so we will stick to describing just one of them, AES128. For simplicity, in the remainder of this section, AES means the algorithm AES128. We'll write $C = \mathsf{AES}_K(M)$ where $|K| = 128$ and $|M| = |C| = 128$.

We're going to describe AES in terms of four additional mappings: *expand*, S, *shift-rows*, and *mix-cols*. The function *expand* takes a 128-bit string and produces a vector of eleven keys, $(K_0, \ldots, K_{10})$. The remaining three functions bijectively map 128-bits to 128-bits. Actually, we'll be more general for S, letting it be a map on $((\{0,1\})^8)^+$. Let's postpone describing all of these maps and start off with the high-level structure of AES, which is given in Fig. 3.8.

```
function $\mathsf{AES}_K(M)$
    $(K_0, \ldots, K_{10}) \leftarrow \mathit{expand}(K)$
    $s \leftarrow M \oplus K_0$
    for $r = 1$ to $10$ do
        $s \leftarrow S(s)$
        $s \leftarrow \mathit{shift\text{-}rows}(s)$
        if $r \leq 9$ then $s \leftarrow \mathit{mix\text{-}cols}(s)$ fi
        $s \leftarrow s \oplus K_r$
    end for
    return $s$
```

Figure 3.8: The function AES128. See the accompanying text and figures for definitions of the maps *expand*, S, *shift-rows*, *mix-cols*.

Refer to Fig. 3.8. The value s is called the *state*. One initizlizes the state to M and the final state is the ciphertext C one gets by enciphering M. What happens in each iteration of the for loop is called a *round*. So AES consists of ten rounds. The rounds are identical except that each uses a different subkey K_i and, also, round 10 omits the call to *mix-cols*.

To understand what goes on in S and *mix-cols* we will need to review a bit of algebra. Let us make a pause to do that. We describe a way to do arithmetic on bytes. Identify each byte $a = a_7a_6a_5a_4a_3a_2a_1a_0$ with the formal polynomial $a_7\mathtt{x}^7 + a_6\mathtt{x}^6 + a_5\mathtt{x}^5 + a_4\mathtt{x}^4 + a_3\mathtt{x}^3 + a_2\mathtt{x}^2 + a_1\mathtt{x} + a_0$. We can add two bytes by taking their bitwise xor (which is the same as the mod-2 sum the corresponding polynomials). We can multiply two bytes to get a degree 14 (or less) polynomial, and then take the remainder of this polynomial by the fixed irreducible polynomial

$$m(\mathtt{x}) = \mathtt{x}^8 + \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1\ .$$

This remainder polynomial is a polynomial of degree at most seven which, as before, can be regarded as a byte. In this way, we can add and multiply any two bytes. The resulting algebraic structure has all the properties necessary to be called a *finite field*. In particular, this is one representation of the finite field known as GF(2^8)—the Galois field on $2^8 = 256$ points. As a finite field, you can find the inverse of any nonzero field point (the zero-element is the zero byte) and you can distribute addition over multiplication, for example.

There are some useful tricks when you want to multiply two bytes. Since $m(\mathtt{x})$ is another name for zero, $\mathtt{x}^8 = \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1 = \{1\mathrm{b}\}$. (Here the curly brackets simply indicate a hexadecimal number.) So it is easy to multiply a byte a by the byte $\mathtt{x} = \{02\}$: namely, shift the 8-bit byte a one position to the left, letting the first bit "fall off" (but remember it!) and shifting a zero into the last bit position. We write this operation $a \lll 1$. If that first bit of a was a 0, we are done. If the first bit was a 1, we need to add in (that is, xor in) $\mathtt{x}^8 = \{1\mathrm{b}\}$. In summary, for a a byte, $a \cdot \mathtt{x} = a \cdot \{02\}$ is $a \lll 1$ if the first bit of a is 0, and it is $(a \lll 1) \oplus \{1\mathrm{b}\}$ if the first bit of a is 1.

Knowing how to multiply by $\mathtt{x} = \{02\}$ let's you conveniently multiply by other quantities. For example, to compute $\{\mathrm{a}1\} \cdot \{03\}$ compute $\{\mathrm{a}1\} \cdot (\{02\} \oplus \{01\}) = \{\mathrm{a}1\} \cdot \{02\} \oplus \{\mathrm{a}1\} \cdot \{01\} = \{42\} \oplus \{1\mathrm{b}\} \oplus a1 = \{\mathrm{f}8\}$. Try some more examples on your own.

As we said, each nonzero byte a has a multiplicative inverse, $\mathit{inv}(a) = a^{-1}$, The mapping we will denote $S : \{0,1\}^8 \to \{0,1\}^8$ is obtained from the map $\mathit{inv} : a \mapsto a^{-1}$. First, patch this map to make it total on $\{0,1\}^8$ by setting $\mathit{inv}(\{00\}) = \{00\}$. Then, to compute $S(a)$, first replace a by $\mathit{inv}(a)$,

63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.9: The AES S-box, which is a function $S : \{0,1\}^8 \to \{0,1\}^8$ specified by the following list. All values in hexadecimal. The meaning is: $S(00) = 63$, $S(01) = 7c$, …, $S(\mathrm{ff}) = 16$.

number the bits of a by $a = a_7a_6a_5a_4a_3a_2a_1a_0$, and return the value a', where $a' = a'_7a'_6a'_5a'_4a'_3a'_2a'_1a'_0$ where

$$\begin{pmatrix} a'_7 \\ a'_6 \\ a'_5 \\ a'_4 \\ a'_3 \\ a'_2 \\ a'_1 \\ a'_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

All arithmetic is in GF(2), meaning that addition of bits is their xor and multiplication of bits is the conjunction (and).

All together, the map S is give by Fig. 3.9, which lists the values of

$$S(0), S(1), \ldots, S(255)\ .$$

In fact, one could forget how this table is produced, and just take it for granted. But the fact is that it is made in the simple way we have said.

Now that we have the function S, let us extend it (without bothering to change the name) to a function with domain $\{\{0,1\}^8\}^+$. Namely, given an m-byte string $A = A[1] \ldots A[m]$, set $S(A)$ to be $S(A[1])\ \ldots S(A[m])$. In other words, just apply S bytewise.

Now we're ready to understand the first map, $S(s)$. One takes the 16-byte state s and applies the 8-bit lookup table to each of its bytes to get the modified state s.

Moving on, the *shift-rows* operation works like this. Imagine plastering the 16 bytes of s =

```
function $expand(K)$
    $K_0 \leftarrow K$
    for $i \leftarrow 1$ to 10 do
        $K_i[0] \leftarrow K_{i-1}[0] \oplus S(K_{i-1}[3] \lll 8) \oplus \mathrm{C}_i$
        $K_i[1] \leftarrow K_{i-1}[1] \oplus K_i[0]$
        $K_i[2] \leftarrow K_{i-1}[2] \oplus K_i[1]$
        $K_i[3] \leftarrow K_{i-1}[3] \oplus K_i[2]$
    od
    return $(K_0, \ldots, K_{10})$
```

Figure 3.10: The AES128 key-expansion algorithm maps a 128-bit key K into eleven 128-bit subkeys, $K_0, \ldots, K_{10}$. Constants $(\mathrm{C}_1, \ldots, \mathrm{C}_{10})$ are ({02000000}, {04000000}, {08000000}, {10000000}, {20000000}, {40000000}, {80000000}, {1B000000}, {36000000}, {6C000000}). All other notation is described in the accompanying text.

$s_0 s_1 \ldots s_{15}$ going top-to-bottom, then left-to-right, to make a 4×4 table:

$$\begin{array}{cccc} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{array}$$

For the *shift-rows* step, left circularly shift the second row by one position; the third row by two positions; and the the fourth row by three positions. The first row is not shifted at all. Somewhat less colorfully, the mapping is simply

$$\textit{shift-rows}(s_0 s_1 s_2 \cdots s_{15}) \quad = \quad s_0 s_5 s_{10} s_{15} s_4 s_9 s_{14} s_3 s_8 s_{13} s_2 s_7 s_{12} s_1 s_6 s_{11}$$

Using the same convention as before, the *mix-cols* step takes each of the four columns in the 4×4 table and applies the (same) transformation to it. Thus we define *mix-cols*(s) on 4-byte words, and then extend this to a 16-byte quantity wordwise. The value of *mix-cols*$(a_0 a_1 a_2 a_3) = a_0' a_1' a_2' a_3'$ is defined by:

$$\begin{pmatrix} a_0' \\ a_1' \\ a_2' \\ a_3' \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 02 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

An equivalent way to explain this step is to say that we are multiplying $a(\mathtt{x}) = a_3\mathtt{x}^3 + a_2\mathtt{x}^2 + a_1\mathtt{x}^1 + a_0$ by the fixed polynomial $c(\mathtt{x}) = \{03\}\mathtt{x}^3 + \{01\}\mathtt{x}^2 + \{01\}\mathtt{x} + \{02\}$ and taking the result modulo $\mathtt{x}^4 + 1$.

At this point we have described everything but the key-expansion map, *expand*. That map is given in Fig. 3.10.

We have now completed the definition of AES. One key property is that AES *is* a blockcipher: the map is invertible. This follows because every round is invertible. That a round is invertible follows from each of its steps being invertible, which is a consequence of S being a permutation and the matrix used in *mix-cols* having an inverse.

In the case of DES, the rationale for the design were not made public. Some explanation for different aspects of the design have become more apparent over time as we have watched the effects on DES of new attack strategies, but fundamentally, the question of why the design is as it is has not received a satisfying cipher. In the case of AES there was significantly more documentation of the rationale for design choices. (See the book *The design of Rijndael* by the designers [9]).

Nonetheless, the security of blockciphers, including DES and AES, eventually comes down to the statement that "we have been unable to find effective attacks, and we have tried attacks along the following lines ...." If people with enough smarts and experience utter this statement, then it suggests that the blockcipher is good. Beyond this, it's hard to say much. Yet, by now, our community has become reasonably experienced designing these things. It wouldn't even be that hard a game were it not for the fact we tend to be agressive in optimizing the block-cipher's speed. (Some may come to the opposite opinion, that it's a very hard game, seeing just how many reasonable-looking blockciphers have been broken.) Later we give some vague sense of the sort of cleverness that people muster against blockciphers.

3.6 Limitations of key-recovery based security

As discussed above, classically, the security of blockciphers has been looked at with regard to key recovery. That is, analysis of a blockcipher E has focused primarily on the following question: given some number q of input-output examples $(M_1, C_1), \ldots, (M_q, C_q)$, where T is a random, unknown key and $C_i = E_T(M_i)$, how hard is it for an attacker to find T? A blockcipher is viewed as "secure" if the best key-recovery attack is computationally infeasible, meaning requires a value of q or a running time t that is too large to make the attack practical. In the sequel, we refer to this as *security against key-recovery.*

However, as a notion of security, security against key-recovery is quite limited. A good notion should be sufficiently strong to be useful. This means that if a blockcipher is secure, then it should be possible to use the blockcipher to make worthwhile constructions and be able to have some guarantee of the security of these constructions. But even a cursory glance at common blockcipher usages shows that good security in the sense of key recovery is not sufficient for security of the usages of blockciphers.

As an example, consider that we typically want to think of $C = E_K(M)$ as an "encryption" of plaintext M under key K. An adversary in possession of C but not knowing K should find it computationally infeasible to recover M, or even some part of M such as its first half. Security against key-recovery is certainly necessary for this, because if the adversary could find K it could certainly compute M, via $M = E_K^{-1}(M)$. But security against key-recovery is not sufficient to ensure that M cannot be recovered given K alone. As an example, consider the blockcipher $E\colon \{0,1\}^{128} \times \{0,1\}^{256} \to \{0,1\}^{256}$ defined by $E_K(M) = \mathsf{AES}_K(M[1]) \parallel M[2]$ where $M[1]$ is the first 128 bits of M and $M[2]$ is the last 128 bits of M. Key recovery is as hard as for AES, but a ciphertext reveals the second half of the plaintext.

This might seem like an artificial example. Many people, on seeing this, respond by saying: "But, clearly, DES and AES are *not* designed like this." True. But that is missing the point. The point is that security against key-recovery *alone* does not make a "good" blockcipher.

But then what does make a good blockcipher? This questions turns out to not be so easy to answer. Certainly one can list various desirable properties. For example, the ciphertext should not reveal half the bits of the plaintext. But that is not enough either. As we see more usages of ciphers, we build up a longer and longer list of security properties SP1, SP2, SP3, ... that are necessary for the security of some blockcipher based application.

Such a long list of necessary but not sufficient properties is no way to treat security. What we need is a single "mater" property of a blockcipher which, if met, *guarantees* security of *lots of* natural usages of the cipher.

Such a property is that the blockcipher be a pseudorandom permutation (PRF), a notion explored in another chapter.

3.7 Problems

Problem 9 Show that for all $K \in \{0,1\}^{56}$ and all $x \in \{0,1\}^{64}$

$$\mathsf{DES}_K(x) = \overline{\mathsf{DES}_{\overline{K}}(\overline{x})}\ .$$

This is called the key-complementation property of DES. ▮

Problem 10 Explain how to use the key-complementation property of DES to speed up exhaustive key search by about a factor of two. Explain any assumptions that you make. ▮

Problem 11 Find a key K such that $\mathsf{DES}_K(\cdot) = \mathsf{DES}_K^{-1}(\cdot)$. Such a key is sometimes called a "weak" key. ▮

Problem 12 As with AES, suppose we are working in the finite field with 2^8 elements, representing field points using the irreducible polynomial $m(\mathtt{x}) = \mathtt{x}^8 + \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1$. Compute the byte that is the result of multiplying bytes:

$$\{\text{e1}\}\ \cdot\ \{05\}$$

▮

Problem 13 For AES, we have given two different descriptions of *mix-cols*: one using matric multiplication (in GF(2^8)) and one based on multiplying by a fixed polynomial $c(\mathtt{x})$ modulo a second fixed polynomial, $d(\mathtt{x}) = \mathtt{x}^4 + 1$. Show that these two methods are equivalent. ▮

Problem 14 Verify that the matrix used for *mix-cols* has as its inverse the matrix

$$\begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix}$$

Explain why it is that all of the entries in this matrix begin with a zero. ▮

Problem 15 How many different permutations are there from 128 bits to 128 bits? How many different functions are then from 128 bits to 128 bits? ▮

Problem 16 Upper and lower bound, as best you can, the probability that a random function from 128 bits to 128 bits is actually a permutation. ▮

Problem 17 Without consulting any of the numerous public-domain implementations available, implement AES, on your own, from the spec or from the description provided by this chapter. Then test your implementation according to the test vectors provided in the AES documentation. ▮

Problem 18 Justify and then refute the following proposition: enciphering under AES can be implemented faster than deciphering. ▮

Problem 19 Choose a random DES key $K \in \{0,1\}^{56}$. Let (M, C), where $C = \mathsf{DES}_K(M)$, be a single plaintext/ciphertext pair that an adversary knows. Suppose the adversary does an exhaustive key search to locate the lexicographically first key T such that $C = \mathsf{DES}_T(M)$. Estimate the probablity that $T = K$. Discuss any assumptions you must make to answer this question.

Bibliography

[1] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *J. of Cryptology*, Vol. 4, No. 1, pp. 3–72, 1991.

[2] E. Biham and A. Shamir. Differential cryptanalysis of the Full 16-round DES. *Advances in Cryptology – CRYPTO '92*, Lecture Notes in Computer Science Vol. 740, E. Brickell ed., Springer-Verlag, 1992.

[3] J. Daemen and V. Rijmen. The Design of Rijndael. Springer, 2001.

[4] M. Matsui. Linear cryptanalysis method for DES cipher. *Advances in Cryptology – EUROCRYPT '93*, Lecture Notes in Computer Science Vol. 765, T. Helleseth ed., Springer-Verlag, 1993.

[5] EFF DES Cracker Project. `http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/`.

[6] L. Knudsen and J. E. Mathiassen. A Chosen-Plaintext Linear Attack on DES. *Fast Software Encryption '00*, Lecture Notes in Computer Science Vol. 1978, B. Schneier ed., Springer-Verlag, 2000.

[7] M. Wiener. Efficient DES Key Search. In Practical Cryptography for Data Internetworks, W. Stallings editor, IEEE Computer Society Press, 1996, pp. 31–79. `http://www3.sympatico.ca/wienerfamily/Michael/MichaelPapers/dessearch.pdf`.

Chapter 4

Pseudorandom Functions

Pseudorandom functions (PRFs) and their cousins, pseudorandom permutations (PRPs), figure as central tools in the design of protocols, especially those for shared-key cryptography. At one level, PRFs and PRPs can be used to model blockciphers, and they thereby enable the security analysis of protocols based on blockciphers. But PRFs and PRPs are also a useful conceptual starting point in contexts where blockciphers don't quite fit the bill because of their fixed block-length. So in this chapter we will introduce PRFs and PRPs and investigate their basic properties.

4.1 Function families

A *function family* is a map $F\colon \mathcal{K} \times D \to R$. Here $\mathcal{K}$ is the set of keys of F and D is the domain of F and R is the range of F. The set of keys and the range are finite, and all of the sets are nonempty. The two-input function F takes a key K and an input X to return a point Y we denote by $F(K, X)$. For any key $K \in \mathcal{K}$ we define the map $F_K\colon D \to R$ by $F_K(X) = F(K, Y)$. We call the function F_K an *instance* of function family F. Thus F specifies a collection of maps, one for each key. That's why we call F a function *family* or *family of functions*.

Sometimes we write $\mathsf{Keys}(F)$ for $\mathcal{K}$, $\mathsf{D}(F)$ for D, and $\mathsf{R}(F)$ for R. Usually $\mathcal{K} = \{0,1\}^k$ for some integer k, the *key length*. Often $D = \{0,1\}^\ell$ for some integer ℓ called the *input length*, and $R = \{0,1\}^L$ for some integers L called the *output length*. But sometimes the domain or range could be sets containing strings of varying lengths.

There is some probability distribution on the (finite) set of keys $\mathcal{K}$. Unless otherwise indicated, this distribution will be the uniform one. We denote by $K \overset{\$}{\leftarrow} \mathcal{K}$ the operation of selecting a random string from $\mathcal{K}$ and naming it K. We denote by $f \overset{\$}{\leftarrow} F$ the operation: $K \overset{\$}{\leftarrow} \mathcal{K}$; $f \leftarrow F_K$. In other words, let f be the function F_K where K is a randomly chosen key. We are interested in the input-output behavior of this randomly chosen instance of the family.

A *permutation* is a bijection (i.e. a one-to-one onto map) whose domain and range are the same set. That is, a map $\pi\colon D \to D$ is a permutation if for every $y \in D$ there is exactly one $x \in D$ such that $\pi(x) = y$. We say that F is a family of permutations if $\mathsf{D}(F) = \mathsf{R}(F)$ and each F_K is a permutation on this common set.

Example 4.1.1 A blockcipher is a family of permutations. In particular DES is a family of permutations $\mathsf{DES}\colon \mathcal{K} \times D \to R$ with

$$\mathcal{K} = \{0,1\}^{56} \text{ and } D = \{0,1\}^{64} \text{ and } R = \{0,1\}^{64} .$$

Here the key length is $k = 56$ and the input length and output length are $\ell = L = 64$. Similarly AES (when "AES" refers to "AES128") is a family of permutations $\mathsf{AES}\colon \mathcal{K} \times D \to R$ with

$$\mathcal{K} = \{0,1\}^{128} \text{ and } D = \{0,1\}^{128} \text{ and } R = \{0,1\}^{128} .$$

Here the key length is $k = 128$ and the input length and output length are $\ell = L = 128$. ▌

4.2 Games

We will use code-based games [4] in definitions and some proofs. We recall some background here. A game —see Fig. 4.1 for an example— has an **Initialize** procedure, procedures to respond to adversary oracle queries, and a **Finalize** procedure. A game G is executed with an adversary A as follows. First, **Initialize** executes and its outputs are the inputs to A. Then, A executes, its oracle queries being answered by the corresponding procedures of G. When A terminates, its output becomes the input to the **Finalize** procedure. The output of the latter, denoted G^A, is called the output of the game, and we let "$G^A \Rightarrow y$" denote the event that this game output takes value y. Variables not explicitly initialized or assigned are assumed to have value $\perp$, except for booleans which are assumed initialized to false. Games G_i, G_j are *identical until* bad if their code differs only in statements that follow the setting of the boolean flag bad to true. The following is the Fundamental Lemmas of game-playing:

Lemma 4.2.1 [4] Let G_i, G_j be identical until bad games, and A an adversary. Let BAD_i (resp. BAD_j) denote the event that the execution of G_i (resp. G_j) with A sets bad. Then

$$\Pr\left[G_i^A \wedge \mathsf{BAD}_i\right] = \Pr\left[G_j^A \wedge \mathsf{BAD}_j\right] \text{ and } \Pr\left[G_i^A\right] - \Pr\left[G_j^A\right] \le \Pr\left[\mathsf{BAD}_j\right] .$$

When the **Finalize** is absent, it is understood to be the identity function.

```
Finalize(d)
Return d.
```

In this case the output G^A of the game is the same as the output of the adversary.

4.3 Random functions and permutations

A particular game that we will consider frequently is the game Rand_R described on the right hand side of Fig. 4.1. Here R is a finite set, for example $\{0,1\}^{128}$. The game provides the adversary access to an oracle **Fn** that implements a random function. This means that on any query the oracle returns a random point from R as response subject to the restriction that if twice queried on the same point, the response is the same both time. The game maintains the function in the form of a table T where $T[X]$ holds the value of the function at X. Initially, the table is everywhere undefined, meaning holds $\perp$ in every entry.

One must remember that the term "random function" is misleading. It might lead one to think that certain functions are "random" and others are not. (For example, maybe the constant function that always returns 0^L on any input is not random, but a function with many different range values is random.) This is not right. The randomness of the function refers to the way it was chosen, not to an attribute of the selected function itself. When you choose a function at random, the constant function is just as likely to appear as any other function. It makes no sense to talk of the randomness of an individual function; the term "random function" just means a function chosen at random.

Example 4.3.1 Let's do some simple probabilistic computations to understand random functions. In all of the following, we refer to Rand_R where $R = \{0,1\}^L$.

1. Fix $X \in \{0,1\}^\ell$ and $Y \in \{0,1\}^L$. Let A be

```
Adversary A
$Z \leftarrow \mathbf{Fn}(X)$
Return $(Y = Z)$
```

Then:

$$\Pr\left[\mathrm{Rand}_R^A \Rightarrow \mathsf{true}\right] = 2^{-L} .$$

Notice that the probability doesn't depend on ℓ. Nor does it depend on the values of X, Y.

2. Fix $X_1, X_2 \in \{0,1\}^\ell$ and $Y \in \{0,1\}^L$. Let A be

```
Adversary A
$Z_1 \leftarrow \mathbf{Fn}(X_1)$
$Z_2 \leftarrow \mathbf{Fn}(X_2)$
Return $(Y = Z_1 \wedge Y = Z_2)$
```

Then:

$$\Pr\left[\mathrm{Rand}_R^A \Rightarrow \mathsf{true}\right] = \begin{cases} 2^{-2L} & \text{if } X_1 \neq X_2 \\ 2^{-L} & \text{if } X_1 = X_2 \end{cases}$$

3. Fix $X_1, X_2 \in \{0,1\}^\ell$ and $Y \in \{0,1\}^L$. Let A be

```
Adversary A
$Z_1 \leftarrow \mathbf{Fn}(X_1)$
$Z_2 \leftarrow \mathbf{Fn}(X_2)$
Return $(Y = Z_1 \oplus Z_2)$
```

Then:

$$\Pr\left[\mathrm{Rand}_R^A \Rightarrow \mathsf{true}\right] = \begin{cases} 2^{-L} & \text{if } X_1 \neq X_2 \\ 0 & \text{if } X_1 = X_2 \text{ and } Y \neq 0^L \\ 1 & \text{if } X_1 = X_2 \text{ and } Y = 0^L \end{cases}$$

4. Suppose $l \leq L$ and let $\tau\colon \{0,1\}^L \to \{0,1\}^l$ denote the function that on input $Y \in \{0,1\}^L$ returns the first l bits of Y. Fix $X_1 \in \{0,1\}^\ell$ and $Y_1 \in \{0,1\}^l$. Let A be

```
Adversary A
$Z_1 \leftarrow \mathbf{Fn}(X_1)$
Return $(\tau(Z_1) = Y_1)$
```

Then:

$$\Pr\left[\mathrm{Rand}_R^A \Rightarrow \mathsf{true}\right] = 2^{-l} . \blacksquare$$

4.3.1 Random permutations

The game Perm_D shown on the right hand side of Fig. 4.2 provides the adversary access to an oracle that implements a random permutation over the finite set D. Random permutations are somewhat harder to work with than random functions, due to the lack of independence between values on different points. Let's look at some probabilistic computations involving them.

Example 4.3.2 In all of the following we refer to game Perm_D where $D = \{0,1\}^\ell$.

1. Fix $X, Y \in \{0,1\}^\ell$. Let's A be

```
Adversary A
$Z \leftarrow \mathbf{Fn}(X)$
Return $(Y = Z)$
```

Then

$$\Pr\left[\mathrm{Perm}_D^A \Rightarrow \mathsf{true}\right] = 2^{-\ell} .$$

2. Fix $X_1, X_2 \in \{0,1\}^\ell$ and $Y_1, Y_2 \in \{0,1\}^L$, and assume $X_1 \neq X_2$. Let A be

```
Adversary A
$Z_1 \leftarrow \mathbf{Fn}(X_1)$
$Z_2 \leftarrow \mathbf{Fn}(X_2)$
Return $(Y_1 = Z_1 \wedge Y_2 = Z_2)$
```

Then

$$\Pr\left[\mathrm{Perm}_D^A \Rightarrow \mathsf{true}\right] = \begin{cases} \dfrac{1}{2^\ell(2^\ell - 1)} & \text{if } Y_1 \neq Y_2 \\ 0 & \text{if } Y_1 = Y_2 \end{cases}$$

3. Fix $X_1, X_2 \in \{0,1\}^\ell$ and $Y \in \{0,1\}^\ell$. Let A be

```
Adversary A
$Z_1 \leftarrow \mathbf{Fn}(X_1)$
$Z_2 \leftarrow \mathbf{Fn}(X_2)$
Return $(Y = Z_1 \oplus Z_2)$
```

Then:

$$\Pr\left[\mathrm{Perm}_D^A \Rightarrow \mathsf{true}\right] = \begin{cases} \dfrac{1}{2^\ell - 1} & \text{if } X_1 \neq X_2 \text{ and } Y \neq 0^\ell \\ 0 & \text{if } X_1 \neq X_2 \text{ and } Y = 0^\ell \\ 0 & \text{if } X_1 = X_2 \text{ and } Y \neq 0^\ell \\ 1 & \text{if } X_1 = X_2 \text{ and } Y = 0^\ell \end{cases}$$

In the case $X_1 \neq X_2$ and $Y \neq 0^\ell$ this is computed as follows:

$$\begin{aligned} &\Pr\left[\mathbf{Fn}(X_1) \oplus \mathbf{Fn}(X_2) = Y\right] \\ &= \sum_{Y_1} \Pr\left[\mathbf{Fn}(X_1) = Y_1 \wedge \mathbf{Fn}(X_2) = Y_1 \oplus Y\right] \\ &= \sum_{Y_1} \frac{1}{2^\ell - 1} \cdot \frac{1}{2^\ell} \\ &= 2^\ell \cdot \frac{1}{2^\ell - 1} \cdot \frac{1}{2^\ell} \\ &= \frac{1}{2^\ell - 1} . \end{aligned}$$

Above, the sum is over all $Y_1 \in \{0,1\}^\ell$. In obtaining the second equality, we used item 2 above and the assumption that $Y \neq 0^\ell$.

4.4 Pseudorandom functions

A pseudorandom function is a family of functions with the property that the input-output behavior of a random instance of the family is "computationally indistinguishable" from that of a random function. Someone who has only black-box access to a function, meaning can only feed it inputs and get outputs, has a hard time telling whether the function in question is a random instance of the family in question or a random function. The purpose of this section is to arrive at a suitable formalization of this notion. Later we will look at motivation and applications.

We fix a family of functions $F\colon \mathcal{K} \times D \to R$. (You may want to think $\mathcal{K} = \{0,1\}^k$, $D = \{0,1\}^\ell$ and $R = \{0,1\}^L$ for some integers $k, \ell, L \geq 1$.) Imagine that you are in a room which contains a terminal connected to a computer outside your room. You can type something into your terminal and send it out, and an answer will come back. The allowed questions you can type must be elements of the domain D, and the answers you get back will be elements of the range R. The computer outside your room implements a function $\mathbf{Fn}\colon D \to R$, so that whenever you type a value X you get back $\mathbf{Fn}(X)$. However, your only access to $\mathbf{Fn}$ is via this interface, so the only thing you can see is the input-output behavior of $\mathbf{Fn}$.

We consider two different ways in which $\mathbf{Fn}$ will be chosen, giving rise to two different "worlds." In the "real" world, $\mathbf{Fn}$ is a random instance of F, meaning is F_K for a random K. In the "random" world, $\mathbf{Fn}$ is a random function with range R.

You are not told which of the two worlds was chosen. The choice of world, and of the corresponding function $\mathbf{Fn}$, is made before you enter the room, meaning before you start typing questions. Once made, however, these choices are fixed until your "session" is over. Your job is to discover which world you are in. To do this, the only resource available to you is your link enabling you to provide values X and get back $\mathbf{Fn}(X)$. After trying some number of values of your choice, you must make a decision regarding which world you are in. The quality of pseudorandom family F can be thought of as measured by the difficulty of telling, in the above game, whether you are in the real world or in the random world.

In the formalization, the entity referred to as "you" above is an algorithm called the adversary. The adversary algorithm A may be randomized. We formalize the ability to query $\mathbf{Fn}$ as giving A an *oracle* which takes input any string $X \in D$ and returns $\mathbf{Fn}(X)$. A can only interact with the function by giving it inputs and examining the outputs for those inputs; it cannot examine the function directly in any way. Algorithm A can decide which queries to make, perhaps based on answers received to previous queries. Eventually, it outputs a bit b which is its decision as to which world it is in. Outputting the bit "1" means that A "thinks" it is in the real world; outputting the bit "0" means that A thinks it is in the random world.

The worlds are formalized via the game of Fig. 4.1. The following definition associates to any adversary a number between 0 and 1 that is called its prf-advantage, and is a measure of how well the adversary is doing at determining which world it is in. Further explanations follow the definition.

Definition 4.4.1 Let $F\colon \mathcal{K} \times D \to R$ be a family of functions, and let A be an algorithm that takes an oracle and returns a bit. We consider two games as described in Fig. 4.1. The *prf-advantage* of A is defined as

$$\mathbf{Adv}_F^{\mathrm{prf}}(A) = \Pr\left[\mathrm{Real}_F^A \Rightarrow 1\right] - \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right]$$

It should be noted that the family F is public. The adversary A, and anyone else, knows the description of the family and is capable, given values K, X, of computing $F(K, X)$.

```
Game Real$_F$
procedure Initialize
$K \xleftarrow{\$} \mathsf{Keys}(F)$
procedure Fn($x$)
Return $F_K(x)$
```

```
Game Rand$_R$
procedure Fn($x$)
If $\mathsf{T}[x] = \perp$ then
    $\mathsf{T}[x] \xleftarrow{\$} R$
Return $\mathsf{T}[x]$
```

Figure 4.1: Games used to define PRFs.

Game Real$_F$ picks a random instance F_K of family F and then runs adversary A with oracle $\mathbf{Fn} = F_K$. Adversary A interacts with its oracle, querying it and getting back answers, and eventually outputs a "guess" bit. The game returns the same bit. Game Rand$_R$ implements **Fn** as a random function with range R. Again, adversary A interacts with the oracle, eventually returning a bit that is the output of the game. Each game has a certain probability of returning 1. The probability is taken over the random choices made in the game. Thus, for the first game, the probability is over the choice of K and any random choices that A might make, for A is allowed to be a randomized algorithm. In the second game, the probability is over the random choice made by the game in implementing **Fn** and any random choices that A makes. These two probabilities should be evaluated separately; the two games are completely distinct.

To see how well A does at determining which world it is in, we look at the difference in the probabilities that the two games return 1. If A is doing a good job at telling which world it is in, it would return 1 more often in the first game than in the second. So the difference is a measure of how well A is doing. We call this measure the prf-advantage of A. Think of it as the probability that A "breaks" the scheme F, with "break" interpreted in a specific, technical way based on the definition.

Different adversaries will have different advantages. There are two reasons why one adversary may achieve a greater advantage than another. One is that it is more "clever" in the questions it asks and the way it processes the replies to determine its output. The other is simply that it asks more questions, or spends more time processing the replies. Indeed, we expect that as an adversary sees more and more input-output examples of **Fn**, or spends more computing time, its ability to tell which world it is in should go up.

The "security" of family F as a pseudorandom function must thus be thought of as depending on the resources allowed to the attacker. We may want to know, for any given resource limitations, what is the prf-advantage achieved by the most "clever" adversary amongst all those who are restricted to the given resource limits.

The choice of resources to consider can vary. One resource of interest is the time-complexity t of A. Another resource of interest is the number of queries q that A asks of its oracle. Another resource of interest is the total length μ of all of A's queries. When we state results, we will pay attention to such resources, showing how they influence maximal adversarial advantage.

Let us explain more about the resources we have mentioned, giving some important conventions underlying their measurement. The first resource is the time-complexity of A. To make sense of this we first need to fix a model of computation. We fix some RAM model, as discussed in Chapter 1. Think of the model used in your algorithms courses, often implicitly, so that you could measure the running time. However, we adopt the convention that the *time-complexity* of A refers not just to the running time of A, but to the maximum of the running times of the two games in the definition, plus the size of the code of A. In measuring the running time of the first game, we must count the time to choose the key K at random, and the time to compute the value $F_K(x)$ for any query x

```
Game Real$_F$
procedure Initialize
$K \xleftarrow{\$} \mathsf{Keys}(F)$
procedure Fn($x$)
Return $F_K(x)$
```

```
Game Perm$_D$
procedure Initialize
$\mathsf{UR} \leftarrow \emptyset$
procedure Fn($x$)
If $\mathsf{T}[x] = \perp$ then
    $\mathsf{T}[x] \xleftarrow{\$} D \setminus \mathsf{UR}$ ; $\mathsf{UR} \leftarrow \mathsf{UR} \cup \{\mathsf{T}[x]\}$
Return $\mathsf{T}[x]$
```

Figure 4.2: Games used to define PRP under CPA.

made by A to its oracle. In measuring the running time of the second game, we count the execution time of **Fn** over the call made to it by A.

The number of queries made by A captures the number of input-output examples it sees. In general, not all strings in the domain must have the same length, and hence we also measure the sum of the lengths of all queries made.

The strength of this definition lies in the fact that it does not specify anything about the kinds of strategies that can be used by a adversary; it only limits its resources. A adversary can use whatever means desired to distinguish the function as long as it stays within the specified resource bounds.

What do we mean by a "secure" PRF? Definition 4.4.1 does not have any explicit condition or statement regarding when F should be considered "secure." It only associates to any adversary A attacking F a prf-advantage function. Intuitively, F is "secure" if the value of the advantage function is "low" for all adversaries whose resources are "practical."

This is, of course, not formal. However, we wish to keep it this way because it better reflects reality. In real life, security is not some absolute or boolean attribute; security is a function of the resources invested by an attacker. All modern cryptographic systems are breakable in principle; it is just a question of how long it takes.

This is our first example of a cryptographic definition, and it is worth spending time to study and understand it. We will encounter many more as we go along. Towards this end let us summarize the main features of the definitional framework as we will see them arise later. First, there are *games*, involving an adversary. Then, there is some *advantage* function associated to an adversary which returns the probability that the adversary in question "breaks" the scheme. These two components will be present in all definitions. What varies is the games; this is where we pin down how we measure security.

4.5 Pseudorandom permutations

A family of functions $F\colon \mathcal{K} \times D \to D$ is a pseudorandom permutation if the input-output behavior of a random instance of the family is "computationally indistinguishable" from that of a random permutation on D.

In this setting, there are two kinds of attacks that one can consider. One, as before, is that the adversary gets an oracle for the function **Fn** being tested. However when F is a family of permutations, one can also consider the case where the adversary gets, in addition, an oracle for $\mathbf{Fn}^{-1}$. We consider these settings in turn. The first is the setting of chosen-plaintext attacks while the second is the setting of chosen-ciphertext attacks.

```
Game Real_F
procedure Initialize
$K \stackrel{\$}{\leftarrow} \mathsf{Keys}(F)$
procedure Fn($x$)
Return $F_K(x)$
procedure Fn$^{-1}$($x$)
Return $F_K^{-1}(x)$
```

```
Game Perm_D
procedure Initialize
$\mathsf{UR} \leftarrow \emptyset$ ; $\mathsf{UD} \leftarrow \emptyset$
procedure Fn($x$)
If $\mathsf{T}[x] = \perp$ then
    $\mathsf{T}[x] \stackrel{\$}{\leftarrow} D \setminus \mathsf{UR}$
    $\mathsf{S}[\mathsf{T}[x]] \leftarrow x$
    $\mathsf{UR} \leftarrow \mathsf{UR} \cup \{\mathsf{T}[x]\}$ ; $\mathsf{UD} \leftarrow \mathsf{UD} \cup \{x\}$
Return $\mathsf{T}[x]$
procedure Fn$^{-1}$($y$)
If $\mathsf{S}[y] = \perp$ then
    $\mathsf{S}[y] \stackrel{\$}{\leftarrow} D \setminus \mathsf{UD}$
    $\mathsf{T}[\mathsf{S}[y]] \leftarrow y$
    $\mathsf{UD} \leftarrow \mathsf{UD} \cup \{\mathsf{S}[y]\}$ ; $\mathsf{UR} \leftarrow \mathsf{UR} \cup \{y\}$
Return $\mathsf{S}[y]$
```

Figure 4.3: Games used to define PRP under CCA.

4.5.1 PRP under CPA

We fix a family of functions $F\colon \mathcal{K} \times D \to D$. (You may want to think $\mathcal{K} = \{0,1\}^k$ and $D = \{0,1\}^\ell$, since this is the most common case. We do not mandate that F be a family of permutations although again this is the most common case.) As before, we consider an adversary A that is placed in a room where it has oracle access to a function **Fn** chosen in one of two ways.

In the "real" world, **Fn** is a random instance of F, meaning is F_K for a random K. In the "random" world, **Fn** is a random permutation on D.

Notice that the real world is the same in the PRF setting, but the random world has changed. As before the task facing the adversary A is to determine in which world it was placed based on the input-output behavior of **Fn**.

Definition 4.5.1 Let $F\colon \mathcal{K} \times D \to D$ be a family of functions, and let A be an algorithm that takes an oracle **Fn** for a function **Fn**: $D \to D$, and returns a bit. We consider two games as described in Fig. 4.2. The *prp-cpa-advantage* of A is defined as

$$\mathbf{Adv}_F^{\text{prp-cpa}}(A) = \Pr\left[\text{Real}_F^A \Rightarrow 1\right] - \Pr\left[\text{Perm}_D^A \Rightarrow 1\right]$$

The intuition is similar to that for Definition 4.4.1. The difference is that here the "ideal" object that F is being compared with is no longer a random function, but rather a random permutation.

In game Real_F, the probability is over the random choice of key K and also over the coin tosses of A if the latter happens to be randomized. The game returns the same bit that A returns. In game Perm_D, a permutation **Fn**: $D \to D$ is chosen at random, and the result bit of A's computation with oracle **Fn** is returned. The probability is over the choice of **Fn** and the coins of A if any. As before, the measure of how well A did at telling the two worlds apart, which we call the prp-cpa-advantage of A, is the difference between the probabilities that the games return 1.

Conventions regarding resource measures also remain the same as before. Informally, a family F is a secure PRP under CPA if $\mathbf{Adv}_F^{\text{prp-cpa}}(A)$ is "small" for all adversaries using a "practical" amount of resources.

4.5.2 PRP under CCA

We fix a family of permutations $F: \mathcal{K} \times D \to D$. (You may want to think $\mathcal{K} = \{0,1\}^k$ and $D = \{0,1\}^\ell$, since this is the most common case. This time, we do mandate that F be a family of permutations.) As before, we consider an adversary A that is placed in a room, but now it has oracle access to two functions, **Fn** and its inverse $\mathbf{Fn}^{-1}$. The manner in which **Fn** is chosen is the same as in the CPA case, and once **Fn** is chosen, $\mathbf{Fn}^{-1}$ is automatically defined, so we do not have to say how it is chosen.

In the "real" world, **Fn** is a random instance of F, meaning is F_K for a random K. In the "random" world, **Fn** is a random permutation on D. In either case, $\mathbf{Fn}^{-1}$ is the inverse of **Fn**. As before the task facing the adversary A is to determine in which world it was placed based on the input-output behavior of its oracles.

Definition 4.5.2 Let $F: \mathcal{K} \times D \to D$ be a family of permutations, and let A be an algorithm that takes an oracle **Fn** for a function $\mathbf{Fn}: D \to D$, and also an oracle $\mathbf{Fn}^{-1}$ for the function $\mathbf{Fn}^{-1}: D \to D$, and returns a bit. We consider two games as described in Fig. 4.3. The *prp-cca-advantage* of A is defined as

$$\mathbf{Adv}_F^{\text{prp-cca}}(A) = \Pr\left[\text{Real}_F^A \Rightarrow 1\right] - \Pr\left[\text{Perm}_D^A \Rightarrow 1\right]$$

The intuition is similar to that for Definition 4.4.1. The difference is that here the adversary has more power: not only can it query **Fn**, but it can directly query $\mathbf{Fn}^{-1}$. Conventions regarding resource measures also remain the same as before. However, we will be interested in some additional resource parameters. Specifically, since there are now two oracles, we can count separately the number of queries, and total length of these queries, for each. As usual, informally, a family F is a secure PRP under CCA if $\mathbf{Adv}_F^{\text{prp-cca}}(A)$ is "small" for all adversaries using a "practical" amount of resources.

4.5.3 Relations between the notions

If an adversary does not query $\mathbf{Fn}^{-1}$ the oracle might as well not be there, and the adversary is effectively mounting a chosen-plaintext attack. Thus we have the following:

Proposition 4.5.3 [PRP-CCA implies PRP-CPA] Let $F: \mathcal{K} \times D \to D$ be a family of permutations and let A be a prp-cpa adversary. Suppose that A runs in time t, asks q queries, and these queries total μ bits. Then there exists a prp-cca adversary B that runs in time t, asks q chosen-plaintext queries, these queries totaling μ bits, and asks no chosen-ciphertext queries, such that

$$\mathbf{Adv}_F^{\text{prp-cpa}}(A) \leq \ \mathbf{Adv}_F^{\text{prp-cca}}(B)\ . \blacksquare$$

Though the technical result is easy, it is worth stepping back to explain its interpretation. The theorem says that if you have an adversary A that breaks F in the PRP-CPA sense, then you have some *other* adversary B that breaks F in the PRP-CCA sense. Furthermore, the adversary B will be just as efficient as the adversary A was. As a consequence, if you think there is *no* reasonable adversary B that breaks F in the PRP-CCA sense, then you have no choice but to believe that there is *no* reasonable adversary A that breaks F in the PRP-CPA sense. The inexistence of a reasonable adversary B that breaks F in the PRP-CCA sense means that F is PRP-CCA secure, while the inexistence of a reasonable adversary A that breaks F in the PRP-CPA sense means that F is PRP-CPA secure. So PRP-CCA security implies PRP-CPA security, and a statement like the proposition above is how, precisely, one makes such a statement.

4.6 Modeling blockciphers

One of the primary motivations for the notions of pseudorandom functions (PRFs) and pseudorandom permutations (PRPs) is to model blockciphers and thereby enable the security analysis of protocols that use blockciphers.

As discussed in the chapter on blockciphers, classically the security of DES or other blockciphers has been looked at only with regard to key recovery. That is, analysis of a blockcipher F has focused on the following question: Given some number of input-output examples

$$(X_1, F_K(X_1)), \ldots, (X_q, F_K(X_q))$$

where K is a random, unknown key, how hard is it to find K? The blockcipher is taken as "secure" if the resources required to recover the key are prohibitive. Yet, as we saw, even a cursory glance at common blockcipher usages shows that hardness of key recovery is not *sufficient* for security. We had discussed wanting a *master* security property of blockciphers under which natural usages of blockciphers could be proven secure. We suggest that this *master* property is that the blockcipher be a secure PRP, under either CPA or CCA.

We cannot prove that specific blockciphers have this property. The best we can do is assume they do, and then go on to use them. For quantitative security assessments, we would make specific conjectures about the advantage functions of various blockciphers. For example we might conjecture something like:

$$\mathbf{Adv}^{\text{prp-cpa}}_{\mathsf{DES}}(A_{t,q}) \quad \leq \quad c_1 \cdot \frac{t/T_{\mathsf{DES}}}{2^{55}} + c_2 \cdot \frac{q}{2^{40}}$$

for any adversary $A_{t,q}$ that runs in time at most t and asks at most q 64-bit oracle queries. Here T_{DES} is the time to do one DES computation on our fixed RAM model of computation, and c_1, c_2 are some constants depending only on this model. In other words, we are conjecturing that the best attacks are either exhaustive key search or linear cryptanalysis. We might be bolder with regard to AES and conjecture something like

$$\mathbf{Adv}^{\text{prp-cpa}}_{\mathsf{AES}}(B_{t,q}) \quad \leq \quad c_1 \cdot \frac{t/T_{\mathsf{AES}}}{2^{128}} + c_2 \cdot \frac{q}{2^{128}} \ .$$

for any adversary $B_{t,q}$ that runs in time at most t and asks at most q 128-bit oracle queries. We could also make similar conjectures regarding the strength of blockciphers as PRPs under CCA rather than CPA.

More interesting is the PRF security of blockciphers. Here we cannot do better than assume that

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{DES}}(A_{t,q}) \quad \leq \quad c_1 \cdot \frac{t/T_{\mathsf{DES}}}{2^{55}} + \frac{q^2}{2^{64}}$$

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{AES}}(B_{t,q}) \quad \leq \quad c_1 \cdot \frac{t/T_{\mathsf{AES}}}{2^{128}} + \frac{q^2}{2^{128}} \ .$$

for any adversaries $A_{t,q}, B_{t,q}$ running in time at most t and making at most q oracle queries. This is due to the birthday attack discussed later. The second term in each formula arises simply because the object under consideration is a family of permutations.

We stress that these are all conjectures. There could exist highly effective attacks that break DES or AES as a PRF without recovering the key. So far, we do not know of any such attacks, but the amount of cryptanalytic effort that has focused on this goal is small. Certainly, to assume that a blockcipher is a PRF is a much stronger assumption than that it is secure against key recovery.

Nonetheless, the motivation and arguments we have outlined in favor of the PRF assumption stay, and our view is that if a blockcipher is broken as a PRF then it should be considered insecure, and a replacement should be sought.

4.7 Example attacks

Let us illustrate the models by providing adversaries that attack different function families in these models.

Example 4.7.1 We define a family of functions F: $\{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^L$ as follows. We let $k = L\ell$ and view a k-bit key K as specifying an L row by ℓ column matrix of bits. (To be concrete, assume the first L bits of K specify the first column of the matrix, the next L bits of K specify the second column of the matrix, and so on.) The input string $X = X[1] \ldots X[\ell]$ is viewed as a sequence of bits, and the value of $F(K, x)$ is the corresponding matrix vector product. That is

$$F_K(X) \;=\; \begin{bmatrix} K[1,1] & K[1,2] & \cdots & K[1,\ell] \\ K[2,1] & K[2,2] & \cdots & K[2,\ell] \\ \vdots & & & \vdots \\ K[L,1] & K[L,2] & \cdots & K[L,\ell] \end{bmatrix} \cdot \begin{bmatrix} X[1] \\ X[2] \\ \vdots \\ X[l] \end{bmatrix} \;=\; \begin{bmatrix} Y[1] \\ Y[2] \\ \vdots \\ Y[L] \end{bmatrix}$$

where

$$\begin{aligned} Y[1] &= K[1,1]\cdot x[1] \oplus K[1,2]\cdot x[2] \oplus \ldots \oplus K[1,\ell]\cdot x[\ell] \\ Y[2] &= K[2,1]\cdot x[1] \oplus K[2,2]\cdot x[2] \oplus \ldots \oplus K[2,\ell]\cdot x[\ell] \\ \vdots &= \vdots \\ Y[L] &= K[L,1]\cdot x[1] \oplus K[L,2]\cdot x[2] \oplus \ldots \oplus K[L,\ell]\cdot x[\ell] \,. \end{aligned}$$

Here the bits in the matrix are the bits in the key, and arithmetic is modulo two. The question we ask is whether F is a "secure" PRF. We claim that the answer is no. The reason is that one can design an adversary algorithm A that achieves a high advantage (close to 1) in distinguishing between the two worlds.

We observe that for any key K we have $F_K(0^\ell) = 0^L$. This is a weakness since a random function of ℓ-bits to L-bits is very unlikely to return 0^L on input 0^ℓ, and thus this fact can be the basis of a distinguishing adversary. Let us now show how the adversary works. Remember that as per our model it is given an oracle **Fn** for **Fn**: $\{0,1\}^\ell \to \{0,1\}^L$ and will output a bit. Our adversary A works as follows:

```
Adversary $A$
    $Y \leftarrow \mathbf{Fn}(0^\ell)$
    if $Y = 0^L$ then return 1 else return 0
```

This adversary queries its oracle at the point 0^ℓ, and denotes by Y the ℓ-bit string that is returned. If $y = 0^L$ it bets that **Fn** was an instance of the family F, and if $y \neq 0^L$ it bets that **Fn** was a random function. Let us now see how well this adversary does. Let $R = \{0,1\}^L$. We claim that

$$\begin{aligned} \Pr\left[\mathrm{Real}_F^A \Rightarrow 1\right] &= 1 \\ \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] &= 2^{-L} \,. \end{aligned}$$

Why? Look at Game Real_F as defined in Definition 4.4.1. Here $\mathbf{Fn} = F_K$ for some K. In that case it is certainly true that $\mathbf{Fn}(0^\ell) = 0^L$ so by the code we wrote for A the latter will return 1. On the other hand look at Game Rand_R as defined in Definition 4.4.1. Here $\mathbf{Fn}$ is a random function. As we saw in Example 4.3.1, the probability that $\mathbf{Fn}(0^\ell) = 0^L$ will be 2^{-L}, and hence this is the probability that A will return 1. Now as per Definition 4.4.1 we subtract to get

$$\begin{aligned}\mathbf{Adv}_F^{\mathrm{prf}}(A) &= \Pr\left[\mathrm{Real}_F^A \Rightarrow 1\right] - \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] \\ &= 1 - 2^{-L} .\end{aligned}$$

Now let t be the time complexity of F. This is $O(\ell + L)$ plus the time for one computation of F, coming to $O(\ell^2 L)$. The number of queries made by A is just one, and the total length of all queries is l. Our conclusion is that there exists an extremely efficient adversary whose prf-advantage is very high (almost one). Thus, F is not a secure PRF. ▌

Example 4.7.2 . Suppose we are given a secure PRF F: $\{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^L$. We want to use F to design a PRF G: $\{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^{2L}$. The input length of G is the same as that of F but the output length of G is twice that of F. We suggest the following candidate construction: for every k-bit key K and every ℓ-bit input x

$$G_K(x) = F_K(x) \,\|\, F_K(\overline{x}) .$$

Here " $\|$ " denotes concatenation of strings, and $\overline{x}$ denotes the bitwise complement of the string x. We ask whether this is a "good" construction. "Good" means that under the assumption that F is a secure PRF, G should be too. However, this is not true. Regardless of the quality of F, the construct G is insecure. Let us demonstrate this.

We want to specify an adversary attacking G. Since an instance of G maps ℓ bits to $2L$ bits, the adversary D will get an oracle for a function $\mathbf{Fn}$ that maps ℓ bits to $2L$ bits. In the random world, $\mathbf{Fn}$ will be chosen as a random function of ℓ bits to $2L$ bits, while in the real world, $\mathbf{Fn}$ will be set to G_K where K is a random k-bit key. The adversary must determine in which world it is placed. Our adversary works as follows:

```
Adversary $A$
    $y_1 \leftarrow \mathbf{Fn}(1^\ell)$
    $y_2 \leftarrow \mathbf{Fn}(0^\ell)$
    Parse $y_1$ as $y_1 = y_{1,1} \| y_{1,2}$ with $|y_{1,1}| = |y_{1,2}| = L$
    Parse $y_2$ as $y_2 = y_{2,1} \| y_{2,2}$ with $|y_{2,1}| = |y_{2,2}| = L$
    if $y_{1,1} = y_{2,2}$ then return 1 else return 0
```

This adversary queries its oracle at the point 1^ℓ to get back y_1 and then queries its oracle at the point 0^ℓ to get back y_2. Notice that 1^ℓ is the bitwise complement of 0^ℓ. The adversary checks whether the first half of y_1 equals the second half of y_2, and if so bets that it is in the real world. Let us now see how well this adversary does. Let $R = \{0,1\}^{2L}$. We claim that

$$\begin{aligned}\Pr\left[\mathrm{Real}_G^A \Rightarrow 1\right] &= 1 \\ \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] &= 2^{-L} .\end{aligned}$$

Why? Look at Game Real_G as defined in Definition 4.4.1. Here $g = G_K$ for some K. In that case we have

$$\begin{aligned}G_K(1^\ell) &= F_K(1^\ell) \,\|\, F_K(0^\ell) \\ G_K(0^\ell) &= F_K(0^\ell) \,\|\, F_K(1^\ell)\end{aligned}$$

by definition of the family G. Notice that the first half of $G_K(1^\ell)$ is the same as the second half of $G_K(0^\ell)$. So A will return 1. On the other hand look at Game Rand_R as defined in Definition 4.4.1. Here $\mathbf{Fn}$ is a random function. So the values $\mathbf{Fn}(1^\ell)$ and $\mathbf{Fn}(0^\ell)$ are both random and independent $2L$ bit strings. What is the probability that the first half of the first string equals the second half of the second string? It is exactly the probability that two randomly chosen L-bit strings are equal, and this is 2^{-L}. So this is the probability that A will return 1. Now as per Definition 4.4.1 we subtract to get

$$\begin{aligned}\mathbf{Adv}_G^{\mathrm{prf}}(A) &= \Pr\left[\mathrm{Real}_G^A{\Rightarrow}1\right] - \Pr\left[\mathrm{Rand}_R^A{\Rightarrow}1\right] \\ &= 1 - 2^{-L}\,.\end{aligned}$$

Now let t be the time complexity of A. This is $O(\ell + L)$ plus the time for two computations of G, coming to $O(\ell + L)$ plus the time for four computations of F. The number of queries made by D is two, and the total length of all queries is 2ℓ. Thus we have exhibited an efficient adversary with a very high prf-advantage, showing that G is not a secure PRF. ▮

4.8 Security against key recovery

We have mentioned several times that security against key recovery is not sufficient as a notion of security for a blockcipher. However it is certainly necessary: if key recovery is easy, the blockcipher should be declared insecure. We have indicated that we want to adopt as notion of security for a blockcipher the notion of a PRF or a PRP. If this is to be viable, it should be the case that any function family that is insecure under key recovery is also insecure as a PRF or PRP. In this section we verify this simple fact. Doing so will enable us to exercise the method of reductions.

We begin by formalizing security against key recovery. We consider an adversary that, based on input-output examples of an instance F_K of family F, tries to find K. Its advantage is defined as the probability that it succeeds in finding K. The probability is over the random choice of K, and any random choices of the adversary itself.

We give the adversary oracle access to F_K so that it can obtain input-output examples of its choice. We do not constrain the adversary with regard to the method it uses. This leads to the following definition.

Definition 4.8.1 Let $F\colon \mathcal{K} \times D \to R$ be a family of functions, and let B be an algorithm that takes an oracle $\mathbf{Fn}$ for a function $\mathbf{Fn}\colon D \to R$ and outputs a string. We consider the game as described in Fig. 4.4. The *kr-advantage* of B is defined as

$$\mathbf{Adv}_F^{\mathrm{kr}}(B) = \Pr\left[\mathrm{KR}_F^B{\Rightarrow}1\right]$$

This definition has been made general enough to capture all types of key-recovery attacks. Any of the classical attacks such as exhaustive key search, differential cryptanalysis or linear cryptanalysis correspond to different, specific choices of adversary B. They fall in this framework because all have the goal of finding the key K based on some number of input-output examples of an instance F_K of the cipher. To illustrate let us see what are the implications of the classical key-recovery attacks on DES for the value of the key-recovery advantage function of DES. Assuming the exhaustive key-search attack is always successful based on testing two input-output examples leads to the fact that there exists an adversary B such that $\mathbf{Adv}_{\mathsf{DES}}^{\mathrm{kr}}(B) = 1$ and B makes two oracle queries and

```
Game KR_F

procedure Initialize
$K \xleftarrow{\$} \mathsf{Keys}(F)$

procedure Fn($x$)
return $F_K(x)$

procedure Finalize($K'$)
return $(K = K')$
```

Figure 4.4: Game used to define KR.

has running time about 2^{55} times the time T_{DES} for one computation of DES. On the other hand, linear cryptanalysis implies that there exists an adversary B such that $\mathbf{Adv}^{\mathrm{kr}}_{\mathsf{DES}}(B) \geq 1/2$ and B makes 2^{44} oracle queries and has running time about 2^{44} times the time T_{DES} for one computation of DES.

For a more concrete example, let us look at the key-recovery advantage of the family of Example 4.7.1.

Example 4.8.2 Let F: $\{0,1\}^k \times \{0,1\}^l \to \{0,1\}^L$ be the family of functions from Example 4.7.1. We saw that its prf-advantage was very high. Let us now compute its kr-advantage. The following adversary B recovers the key. We let e_j be the l-bit binary string having a 1 in position j and zeros everywhere else. We assume that the manner in which the key K defines the matrix is that the first L bits of K form the first column of the matrix, the next L bits of K form the second column of the matrix, and so on.

```
Adversary $B$
    $K' \leftarrow \varepsilon$    // $\varepsilon$ is the empty string
    for $j = 1, \ldots, l$ do
        $y_j \leftarrow \mathbf{Fn}(e_j)$
        $K' \leftarrow K' \parallel y_j$
    return $K'$
```

The adversary B invokes its oracle to compute the output of the function on input e_j. The result, y_j, is exactly the j-th column of the matrix associated to the key K. The matrix entries are concatenated to yield K', which is returned as the key. Since the adversary always finds the key we have

$$\mathbf{Adv}^{\mathrm{kr}}_{F}(B) \;=\; 1\,.$$

The time-complexity of this adversary is $t = O(l^2 L)$ since it makes $q = l$ calls to its oracle and each computation of $\mathbf{Fn}$ takes $O(lL)$ time. The parameters here should still be considered small: l is 64 or 128, which is small for the number of queries. So F is insecure against key-recovery. ▌

Note that the F of the above example is less secure as a PRF than against key-recovery: its advantage function as a PRF had a value close to 1 for parameter values much smaller than those above. This leads into our next claim, which says that for any given parameter values, the kr-advantage of a family cannot be significantly more than its prf or prp-cpa advantage.

Proposition 4.8.3 Let $F\colon \mathcal{K} \times D \to R$ be a family of functions, and let B be a key-recovery adversary against F. Assume B's running time is at most t and it makes at most $q < |D|$ oracle queries. Then there exists a PRF adversary A against F such that A has running time at most t plus the time for one computation of F, makes at most $q+1$ oracle queries, and

$$\mathbf{Adv}^{\mathrm{kr}}_F(B) \quad \leq \quad \mathbf{Adv}^{\mathrm{prf}}_F(A) + \frac{1}{|R|} \, . \tag{4.1}$$

Furthermore if $D = R$ then there also exists a PRP CPA adversary A against F such that A has running time at most t plus the time for one computation of F, makes at most $q+1$ oracle queries, and

$$\mathbf{Adv}^{\mathrm{kr}}_F(B) \quad \leq \quad \mathbf{Adv}^{\mathrm{prp\text{-}cpa}}_F(A) + \frac{1}{|D|-q} \, . \blacksquare \tag{4.2}$$

The Proposition implies that if a family of functions is a secure PRF or PRP then it is also secure against all key-recovery attacks. In particular, if a blockcipher is modeled as a PRP or PRF, we are implicitly assuming it to be secure against key-recovery attacks.

Before proceeding to a formal proof let us discuss the underlying ideas. The problem that adversary A is trying to solve is to determine whether its given oracle $\mathbf{Fn}$ is a random instance of F or a random function of D to R. A will run B as a subroutine and use B's output to solve its own problem.

B is an algorithm that expects to be in a world where it gets an oracle $\mathbf{Fn}$ for some random key $K \in \mathcal{K}$, and it tries to find K via queries to its oracle. For simplicity, first assume that B makes no oracle queries. Now, when A runs B, it produces some key K'. A can test K' by checking whether $F(K', x)$ agrees with $\mathbf{Fn}(x)$ for some value x. If so, it bets that $\mathbf{Fn}$ was an instance of F, and if not it bets that $\mathbf{Fn}$ was random.

If B does make oracle queries, we must ask how A can run B at all. The oracle that B wants is not available. However, B is a piece of code, communicating with its oracle via a prescribed interface. If you start running B, at some point it will output an oracle query, say by writing this to some prescribed memory location, and stop. It awaits an answer, to be provided in another prescribed memory location. When that appears, it continues its execution. When it is done making oracle queries, it will return its output. Now when A runs B, it will itself supply the answers to B's oracle queries. When B stops, having made some query, A will fill in the reply in the prescribed memory location, and let B continue its execution. B does not know the difference between this "simulated" oracle and the real oracle except in so far as it can glean this from the values returned.

The value that B expects in reply to query x is $F_K(x)$ where K is a random key from $\mathcal{K}$. However, A returns to it as the answer to query x the value $\mathbf{Fn}(x)$, where $\mathbf{Fn}$ is A's oracle. When A is in the real world, $\mathbf{Fn}(x)$ is an instance of F and so B is functioning as it would in its usual environment, and will return the key K with a probability equal to its kr-advantage. However when A is in the random world, $\mathbf{Fn}$ is a random function, and B is getting back values that bear little relation to the ones it is expecting. That does not matter. B is a piece of code that will run to completion and produce some output. When we are in the random world, we have no idea what properties this output will have. But it is some key in $\mathcal{K}$, and A will test it as indicated above. It will fail the test with high probability as long as the test point x was not one that B queried, and A will make sure the latter is true via its choice of x. Let us now proceed to the actual proof.

Proof of Proposition 4.8.3: We prove the first equation and then briefly indicate how to alter the proof to prove the second equation.

As per Definition 4.4.1, adversary A will be provided an oracle $\mathbf{Fn}$ for a function $\mathbf{Fn}\colon D \to R$, and will try to determine in which World it is. To do so, it will run adversary B as a subroutine. We provide the description followed by an explanation and analysis.

```
Adversary A
  $i \leftarrow 0$
  Run adversary $B$, replying to its oracle queries as follows
  When $B$ makes an oracle query $x$ do
    $i \leftarrow i+1$ ; $x_i \leftarrow x$
    $y_i \leftarrow \mathbf{Fn}(x_i)$
    Return $y_i$ to $B$ as the answer
  Until $B$ stops and outputs a key $K'$
  Let $x$ be some point in $D - \{x_1, \ldots, x_q\}$
  $y \leftarrow \mathbf{Fn}(x)$
  if $F(K', x) = y$ then return 1 else return 0
```

As indicated in the discussion preceding the proof, A is running B and itself providing answers to B's oracle queries via the oracle $\mathbf{Fn}$. When B has run to completion it returns some $K' \in \mathcal{K}$, which A tests by checking whether $F(K', x)$ agrees with $\mathbf{Fn}(x)$. Here x is a value different from any that B queried, and it is to ensure that such a value can be found that we require $q < |D|$ in the statement of the Proposition. Now we claim that

$$\Pr\left[\mathrm{Real}_F^A \Rightarrow 1\right] \geq \mathbf{Adv}_F^{\mathrm{kr}}(B) \tag{4.3}$$

$$\Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] = \frac{1}{|R|} . \tag{4.4}$$

We will justify these claims shortly, but first let us use them to conclude. Subtracting, as per Definition 4.4.1, we get

$$\begin{aligned}\mathbf{Adv}_F^{\mathrm{prf}}(A) &= \Pr\left[\mathrm{Real}_F^A \Rightarrow 1\right] - \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] \\ &\geq \mathbf{Adv}_F^{\mathrm{kr}}(B) - \frac{1}{|R|}\end{aligned}$$

as desired. It remains to justify Equations (4.3) and (4.4).

Equation (4.3) is true because in Real_F the oracle $\mathbf{Fn}$ is a random instance of F, which is the oracle that B expects, and thus B functions as it does in KR_F^B. If B is successful, meaning the key K' it outputs equals K, then certainly A returns 1. (It is possible that A might return 1 even though B was not successful. This would happen if $K' \neq K$ but $F(K', x) = F(K, x)$. It is for this reason that Equation (4.3) is in inequality rather than an equality.) Equation (4.4) is true because in Rand_R the function $\mathbf{Fn}$ is random, and since x was never queried by B, the value $\mathbf{Fn}(x)$ is unpredictable to B. Imagine that $\mathbf{Fn}(x)$ is chosen only when x is queried to $\mathbf{Fn}$. At that point, K', and thus $F(K', x)$, is already defined. So $\mathbf{Fn}(x)$ has a $1/|R|$ chance of hitting this fixed point. Note this is true regardless of how hard B tries to make $F(K', x)$ be the same as $\mathbf{Fn}(x)$.

For the proof of Equation (4.2), the adversary A is the same. For the analysis we see that

$$\begin{aligned}\Pr\left[\mathrm{Real}_F^A \Rightarrow 1\right] &\geq \mathbf{Adv}_F^{\mathrm{kr}}(B) \\ \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] &\leq \frac{1}{|D| - q} .\end{aligned}$$

Subtracting yields Equation (4.2). The first equation above is true for the same reason as before. The second equation is true because in World 0 the map **Fn** is now a random permutation of D to D. So $\mathbf{Fn}(x)$ assumes, with equal probability, any value in D except $y_1, \ldots, y_q$, meaning there are at least $|D| - q$ things it could be. (Remember $R = D$ in this case.) ▌

The following example illustrates that the converse of the above claim is far from true. The kr-advantage of a family can be significantly smaller than its prf or prp-cpa advantage, meaning that a family might be very secure against key recovery yet very insecure as a prf or prp, and thus not useful for protocol design.

Example 4.8.4 Define the blockcipher E: $\{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^\ell$ by $E_K(x) = x$ for all k-bit keys K and all ℓ-bit inputs x. We claim that it is very secure against key-recovery but very insecure as a PRP under CPA. More precisely, we claim that for any adversary B,

$$\mathbf{Adv}_E^{\mathrm{kr}}(B) \;=\; 2^{-k} ,$$

regardless of the running time and number of queries made by B. On the other hand there is an adversary A, making only one oracle query and having a very small running time, such that

$$\mathbf{Adv}_E^{\mathrm{prp\text{-}cpa}}(A) \;\geq\; 1 - 2^{-\ell} .$$

In other words, given an oracle for E_K, you may make as many queries as you want, and spend as much time as you like, before outputting your guess as to the value of K, yet your chance of getting it right is only 2^{-k}. On the other hand, using only a single query to a given oracle **Fn**: $\{0,1\}^\ell \to \{0,1\}^\ell$, and very little time, you can tell almost with certainty whether **Fn** is an instance of E or is a random function of ℓ bits to ℓ bits. Why are these claims true? Since E_K does not depend on K, an adversary with oracle E_K gets no information about K by querying it, and hence its guess as to the value of K can be correct only with probability 2^{-k}. On the other hand, an adversary can test whether $\mathbf{Fn}(0^\ell) = 0^\ell$, and by returning 1 if and only if this is true, attain a prp-advantage of $1 - 2^{-\ell}$. ▌

4.9 The birthday attack

Suppose E: $\{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^\ell$ is a family of permutations, meaning a blockcipher. If we are given an oracle **Fn**: $\{0,1\}^\ell \to \{0,1\}^\ell$ which is either an instance of E or a random function, there is a simple test to determine which of these it is. Query the oracle at distinct points $x_1, x_2, \ldots, x_q$, and get back values $y_1, y_2, \ldots, y_q$. You know that if **Fn** were a permutation, the values $y_1, y_2, \ldots, y_q$ must be distinct. If **Fn** was a random function, they may or may not be distinct. So, if they are distinct, bet on a permutation.

Surprisingly, this is pretty good adversary, as we will argue below. Roughly, it takes $q = \sqrt{2^\ell}$ queries to get an advantage that is quite close to 1. The reason is the birthday paradox. If you are not familiar with this, you may want to look at the appendix on the birthday problem and then come back to the following.

This tells us that an instance of a blockcipher can be distinguished from a random function based on seeing a number of input-output examples which is approximately $2^{\ell/2}$. This has important consequences for the security of blockcipher based protocols.

Proposition 4.9.1 Let E: $\{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^\ell$ be a family of permutations. Suppose q satisfies $2 \leq q \leq 2^{(\ell+1)/2}$. Then there is an adversary A, making q oracle queries and having running time about that to do q computations of E, such that

$$\mathbf{Adv}_E^{\mathrm{prf}}(A) \;\;\geq\;\; 0.3 \cdot \frac{q(q-1)}{2^\ell} \; . \; ▌ \qquad (4.5)$$

Proof of Proposition 4.9.1: Adversary A is given an oracle $\mathbf{Fn}$: $\{0,1\}^\ell \to \{0,1\}^\ell$ and works like this:

```
Adversary A
for $i = 1, \ldots, q$ do
    Let $x_i$ be the $i$-th $\ell$-bit string in lexicographic order
    $y_i \leftarrow \mathbf{Fn}(x_i)$
if $y_1, \ldots, y_q$ are all distinct then return 1, else return 0
```

Let us now justify Equation (4.5). Letting $N = 2^\ell$, we claim that

$$\Pr\left[\mathrm{Real}_E^A \Rightarrow 1\right] = 1 \tag{4.6}$$

$$\Pr\left[\mathrm{Rand}_E^A \Rightarrow 1\right] = 1 - C(N,q) \;. \tag{4.7}$$

Here $C(N,q)$, as defined in the appendix on the birthday problem, is the probability that some bin gets two or more balls in the experiment of randomly throwing q balls into N bins. We will justify these claims shortly, but first let us use them to conclude. Subtracting, we get

$$\begin{aligned} \mathbf{Adv}_E^{\mathrm{prf}}(A) &= \Pr\left[\mathrm{Real}_E^A \Rightarrow 1\right] - \Pr\left[\mathrm{Rand}_E^A \Rightarrow 1\right] \\ &= 1 - [1 - C(N,q)] \\ &= C(N,q) \\ &\geq 0.3 \cdot \frac{q(q-1)}{2^\ell} \;. \end{aligned}$$

The last line is by Theorem A.1 in the appendix on the birthday problem. It remains to justify Equations (4.6) and (4.7).

Equation (4.6) is clear because in the real world, $\mathbf{Fn} = E_K$ for some key K, and since E is a family of permutations, $\mathbf{Fn}$ is a permutation, and thus $y_1, \ldots, y_q$ are all distinct. Now, suppose A is in the random world, so that $\mathbf{Fn}$ is a random function of ℓ bits to ℓ bits. What is the probability that $y_1, \ldots, y_q$ are all distinct? Since $\mathbf{Fn}$ is a random function and $x_1, \ldots, x_q$ are distinct, $y_1, \ldots, y_q$ are random, independently distributed values in $\{0,1\}^\ell$. Thus we are looking at the birthday problem. We are throwing q balls into $N = 2^\ell$ bins and asking what is the probability of there being no collisions, meaning no bin contains two or more balls. This is $1 - C(N,q)$, justifying Equation (4.7). ▌

4.10 The PRP/PRF switching lemma

When we analyse blockcipher-based constructions, we find a curious dichotomy: PRPs are what most naturally model blockciphers, but analyses are often considerably simpler and more natural assuming the blockcipher is a PRF. To bridge the gap, we relate the prp-security of a blockcipher to its prf-security. The following says, roughly, these two measures are always close—they don't differ by more than the amount given by the birthday attack. Thus a particular family of permutations E may have prf-advantage that exceeds its prp-advantage, but not by more than $0.5\, q^2/2^n$.

Lemma 4.10.1 [PRP/PRF Switching Lemma] Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a function family. Let $R = \{0,1\}^n$. Let A be an adversary that asks at most q oracle queries. Then

$$\left|\Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] - \Pr\left[\mathrm{Perm}_R^A \Rightarrow 1\right]\right| \;\leq\; \frac{q(q-1)}{2^{n+1}} \,. \tag{4.8}$$

As a consequence, we have that

$$\left|\mathbf{Adv}_E^{\mathrm{prf}}(A) - \mathbf{Adv}_E^{\mathrm{prp}}(A)\right| \;\leq\; \frac{q(q-1)}{2^{n+1}} \,. \blacksquare \tag{4.9}$$

The proof introduces a technique that we shall use repeatedly: a *game-playing argument.* We are trying to compare what happens when an adversary A interacts with one kind of object—a random permutation oracle—to what happens when the adversary interacts with a different kind of object—a random function oracle. So we set up each of these two interactions as a kind of game, writing out the game in pseudocode. The two games are written in a way that highlights when they have differing behaviors. In particular, any time that the behavior in the two games differ, we set a flag bad. The probability that the flag bad gets set in one of the two games is then used to bound the difference between the probability that the adversary outputs 1 in one game and the the probability that the adversary outputs 1 in the other game.

Proof: Let's begin with Equation (4.8), as Equation (4.9) follows from that. We need to establish that

$$-\frac{q(q-1)}{2^{n+1}} \;\leq\; \Pr\left[\mathrm{Rand}_R^A \Rightarrow 1\right] - \Pr\left[\mathrm{Perm}_R^A \Rightarrow 1\right] \;\leq\; \frac{q(q-1)}{2^{n+1}}$$

Let's show the right-hand inequality, since the left-hand inequality works in exactly the same way. So we are trying to establish that

$$\Pr[A^\rho \Rightarrow 1] - \Pr[A^\pi \Rightarrow 1] \;\leq\; \frac{q(q-1)}{2^{n+1}} \,. \tag{4.10}$$

We can assume that A never asks an oracle query that is not an n-bit string. You can assume that such an *invalid* oracle query would generate an error message. The same error message would be generated on any invalid query, regardless of A's oracle, so asking invalid queries is pointless for A.

We can also assume that A never *repeats* an oracle query: if it asks a question X it won't later ask the same question X. It's not interesting for A to repeat a question, because it's going to get the same answer as before, independent of the type of oracle to which A is speaking to. More precisely, with a little bit of bookkeeping the adversary can remember what was its answer to each oracle query it already asked, and it doesn't have to repeat an oracle query because the adversary can just as well look up the prior answer.

Let's look at Games G_0 and G_1 of Fig. 4.5. Notice that the adversary never sees the flag bad. The flag bad will play a central part in our analysis, but it is not something that the adversary A can get hold of. It's only for our bookkeeping.

Suppose that the adversary asks a query X. By our assumptions about A, the string X is an n-bit string that the adversary has not yet asked about. In line 10, we choose a random n-bit string Y. Lines 11,12, next, are the most interesting. If the point Y that we just chose is already in the range of the function we are defining then we set a flag bad. In such a case, if we are playing game G_0, then we now make a *fresh* choice of Y, this time from the co-range of the function. If we are playing game G_1 then we stick with our original choice of Y. Either way, we return Y, effectively growing the domain of our function.

```
procedure Initialize // $\boxed{G_0}$, $G_1$
UR ← ∅

procedure Fn($x$)
10  $Y \xleftarrow{\$} R$
11  if $Y \in$ UR then
12      bad ← true; $\boxed{Y \xleftarrow{\$} R \setminus \mathsf{UR}}$
13  UR ← UR ∪ $\{Y\}$
14  return $Y$
```

Figure 4.5: Games used in the proof of the Switching Lemma. Game G_0 includes the boxed code while game G_1 does not.

Now let's think about what A sees as it plays Game G_1. Whatever query X is asked, we just return a random n-bit string Y. So game G_1 perfectly simulates a random function. Remember that the adversary isn't allowed to repeat a query, so what the adversary would get if it had a random function oracle is a random n-bit string in response to each query—just what we are giving it. Hence

$$\Pr[\text{Rand}_R^A \Rightarrow 1] \quad = \quad \Pr[G_1 \Rightarrow 1] \tag{4.11}$$

Now if we're in game G_0 then what the adversary gets in response to each query X is a random point Y that has not already been returned to A. Thus

$$\Pr[\text{Perm}_R^A \Rightarrow 1] \quad = \quad \Pr[G_0^A \Rightarrow 1]\ . \tag{4.12}$$

But game G_0, G_1 are identical until bad and hence the Fundamental Lemma of game playing implies that

$$\Pr[G_0^A \Rightarrow 1] - \Pr[G_1^A \Rightarrow 1] \quad \le \quad \Pr[G_1^A \text{ sets } \mathsf{bad}]\ . \tag{4.13}$$

To bound $\Pr[G_1^A$ sets bad$]$ is simple. Line 11 is executed q times. The first time it is executed UR contains 0 points; the second time it is executed UR contains 1 point; the third time it is executed Range(π) contains at most 2 points; and so forth. Each time line 11 is executed we have just selected a random value Y that is independent of the contents of UR. By the sum bound, the probability that a Y will ever be in UR at line 11 is therefore at most $0/2^n + 1/2^n + 2/2^n + \cdots + (q-1)/2^n = (1 + 2 + \cdots + (q-1))/2^n = q(q-1)/2^{n+1}$. This completes the proof of Equation (4.10). To go on and show that $\mathbf{Adv}_E^{\mathrm{prf}}(A) - \mathbf{Adv}_E^{\mathrm{prp}}(A) \le q(q-1)/2^{n+1}$ note that

$$\begin{aligned}
\mathbf{Adv}_E^{\mathrm{prf}}(A) - \mathbf{Adv}_E^{\mathrm{prp}}(A) \quad &= \quad \Pr\left[\text{Real}_F^A \Rightarrow 1\right] - \Pr\left[\text{Rand}_R^A \Rightarrow 1\right] - \left(\Pr\left[\text{Real}_F^A \Rightarrow 1\right] - \Pr\left[\text{Perm}_R^A \Rightarrow 1\right]\right) \\
&= \quad \Pr\left[\text{Perm}_R^A \Rightarrow 1\right] - \Pr\left[\text{Rand}_R^A \Rightarrow 1\right] \\
&\le \quad q(q-1)/2^{n+1}
\end{aligned}$$

This completes the proof. ▌

The PRP/PRF switching lemma is one of the central tools for understanding block-cipher based protocols, and the game-playing method will be one of our central techniques for doing proofs.

4.11 Historical notes

The concept of pseudorandom functions is due to Goldreich, Goldwasser and Micali [17], while that of pseudorandom permutation is due to Luby and Rackoff [25]. These works are however in the complexity-theoretic or "asymptotic" setting, where one considers an infinite sequence of families rather than just one family, and defines security by saying that polynomial-time adversaries have "negligible" advantage. In contrast our approach is motivated by the desire to model blockciphers and is called the "concrete security" approach. It originates with [2]. Definitions 4.4.1 and 4.5.1 are from [2], as are Propositions 4.9.1 and 4.10.1.

4.12 Problems

Problem 20 Let E: $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a secure PRP. Consider the family of permutations E': $\{0,1\}^k \times \{0,1\}^{2n} \to \{0,1\}^{2n}$ defined by for all $x, x' \in \{0,1\}^n$ by

$$E'_K(x \,\|\, x') = E_K(x) \,\|\, E_K(x \oplus x') \ .$$

Show that E' is not a secure PRP. ▌

Problem 21 Consider the following blockcipher $E : \{0,1\}^3 \times \{0,1\}^2 \to \{0,1\}^2$:

key	0	1	2	3
0	0	1	2	3
1	3	0	1	2
2	2	3	0	1
3	1	2	3	0
4	0	3	2	1
5	1	0	3	2
6	2	1	0	3
7	3	2	1	0

(The eight possible keys are the eight rows, and each row shows where the points to which 0, 1, 2, and 3 map.) Compute the maximal prp-advantage an adversary can get (a) with one query, (b) with four queries, and (c) with two queries. ▌

Problem 22 Present a secure construction for the problem of Example 4.7.2. That is, given a PRF F: $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, construct a PRF G: $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^{2n}$ which is a secure PRF as long as F is secure. ▌

Problem 23 Design a blockcipher $E : \{0,1\}^k \times \{0,1\}^{128} \to \{0,1\}^{128}$ that is secure (up to a large number of queries) against non-adaptive adversaries, but is completely insecure (even for two queries) against an adaptive adversary. (A non-adaptive adversary readies all her questions $M_1, \ldots, M_q$, in advance, getting back $E_K(M_1), ..., E_K(M_q)$. An adaptive adversary is the sort we have dealt with throughout: each query may depend on prior answers.) ▌

Problem 24 Let $a[i]$ denote the i-th bit of a binary string i, where $1 \le i \le |a|$. The *inner product* of n-bit binary strings a, b is

$$\langle\, a, b \,\rangle \;=\; a[1]b[1] \oplus a[2]b[2] \oplus \cdots \oplus a[n]b[n] \ .$$

```
Game $G$

procedure Initialize
$K \xleftarrow{\$} \mathsf{Keys}(F)$

procedure $f(x)$
Return $F_K(x)$

procedure $g(x)$
Return $F_K(x)$
```

```
Game $H$

procedure Initialize
$K_1 \xleftarrow{\$} \mathsf{Keys}(F)$ ; $K_2 \xleftarrow{\$} \mathsf{Keys}(F)$

procedure $f(x)$
Return $F_{K_1}(x)$

procedure $g(x)$
Return $F_{K_2}(x)$
```

Figure 4.6: Game used to in Problem 26.

A family of functions $F\colon \{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^L$ is said to be *inner-product preserving* if for every $K \in \{0,1\}^k$ and every distinct $x_1, x_2 \in \{0,1\}^\ell - \{0^\ell\}$ we have

$$\langle\, F(K,x_1), F(K,x_2)\,\rangle \;=\; \langle\, x_1, x_2\,\rangle\;.$$

Prove that if F is inner-product preserving then there exists an adversary A, making at most two oracle queries and having running time $2 \cdot T_F + O(\ell)$, where T_F denotes the time to perform one computation of F, such that

$$\mathbf{Adv}^{\mathrm{prf}}_F(A) \geq \frac{1}{2}\cdot\left(1+\frac{1}{2^L}\right)\;.$$

Explain in a sentence why this shows that if F is inner-product preserving then F is not a secure PRF. ▮

Problem 25 Let $E\colon \{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^\ell$ be a blockcipher. The *two-fold cascade* of E is the blockcipher $E^{(2)}\colon \{0,1\}^{2k} \times \{0,1\}^\ell \to \{0,1\}^\ell$ defined by

$$E^{(2)}_{K_1 \,\|\, K_2}(x) = E_{K_1}(E_{K_2}(x))$$

for all $K_1, K_2 \in \{0,1\}^k$ and all $x \in \{0,1\}^\ell$. Prove that if E is a secure PRP then so is $E^{(2)}$. ▮

Problem 26 Let A be a adversary that makes at most q total queries to its two oracles, f and g, where $f, g : \{0,1\}^n \to \{0,1\}^n$. Assume that A never asks the same query X to both of its oracles. Define

$$\mathbf{Adv}(A) = \Pr[G^A = 1] - \Pr[H^A = 1]$$

where games G, H are defined in Fig. 4.6. Prove a good upper bound for $\mathbf{Adv}(A)$, say $\mathbf{Adv}(A) \leq q^2/2^n$. ▮

Problem 27 Let $F\colon \{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^\ell$ be a family of functions and $r \geq 1$ an integer. The *r-round Feistel cipher associated to F* is the family of permutations $F^{(r)}\colon \{0,1\}^{rk} \times \{0,1\}^{2\ell} \to \{0,1\}^{2\ell}$ defined as follows for any $K_1, \ldots, K_r \in \{0,1\}^k$ and input $x \in \{0,1\}^{2\ell}$:

```
Function $F^{(r)}(K_1 \,\|\, \cdots \,\|\, K_r, x)$
    Parse $x$ as $L_0 \,\|\, R_0$ with $|L_0| = |R_0| = \ell$
    For $i = 1, \ldots, r$ do
```

```
        $L_i \leftarrow R_{i-1}$ ; $R_i \leftarrow F(K_i, R_{i-1}) \oplus L_{i-1}$
    EndFor
Return $L_r \,\|\, R_r$
```

(a) Prove that there exists an adversary A, making at most two oracle queries and having running time about that to do two computations of F, such that

$$\mathbf{Adv}^{\mathrm{prf}}_{F^{(2)}}(A) \geq 1 - 2^{-\ell} \, .$$

(b) Prove that there exists an adversary A, making at most two queries to its first oracle and one to its second oracle, and having running time about that to do three computations of F or F^{-1}, such that

$$\mathbf{Adv}^{\mathrm{prp\text{-}cca}}_{F^{(3)}}(A) \geq 1 - 3 \cdot 2^{-\ell} \, . \blacksquare$$

Problem 28 Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a function family and let A be an adversary that asks at most q queries. In trying to construct a proof that $|\mathbf{Adv}^{\mathrm{prp}}_E(A) - \mathbf{Adv}^{\mathrm{prf}}_E(A)| \leq q^2/2^{n+1}$, Michael and Peter put forward an argument a fragment of which is as follows:

> Consider an adversary A that asks at most q oracle queries to an oracle $\mathbf{Fn}$ for a function from R to R, where $R = \{0,1\}^n$. Let C (for "collision") be the event that A asks some two distinct queries X and X' and the oracle returns the same answer. Then clearly
>
> $$\Pr[\mathrm{Perm}^A_R \Rightarrow 1] = \Pr[\mathrm{Rand}^A_R \Rightarrow 1 \mid \overline{\mathsf{C}}].$$

Show that Michael and Peter have it all wrong: prove that the quantities above are not necessarily equal. Do this by selecting a number n and constructing an adversary A for which the left and right sides of the equation above are unequal. $\blacksquare$

Bibliography

[1] M. Bellare and P. Rogaway. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. *Advances in Cryptology – EUROCRYPT '06*, Lecture Notes in Computer Science Vol. , ed., Springer-Verlag, 2006

[2] M. Bellare, J. Kilian and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences* , Vol. 61, No. 3, Dec 2000, pp. 362–399.

[3] O. Goldreich, S. Goldwasser and S. Micali. How to construct random functions. *Journal of the ACM,* Vol. 33, No. 4, 1986, pp. 210–217.

[4] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput,* Vol. 17, No. 2, April 1988.

Chapter 5

Symmetric Encryption

The symmetric setting considers two parties who share a key and will use this key to imbue communicated data with various security attributes. The main security goals are privacy and authenticity of the communicated data. The present chapter looks at privacy. A later chapter looks at authenticity. Chapters 3 and 4 describe tools we shall use here.

5.1 Symmetric encryption schemes

The primitive we will consider is called an *encryption scheme.* Such a scheme specifies an *encryption algorithm*, which tells the sender how to process the plaintext using the key, thereby producing the ciphertext that is actually transmitted. An encryption scheme also specifies a *decryption algorithm*, which tells the receiver how to retrieve the original plaintext from the transmission while possibly performing some verification, too. Finally, there is a *key-generation algorithm*, which produces a key that the parties need to share. The formal description follows.

Definition 5.1.1 A *symmetric encryption scheme* $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ consists of three algorithms, as follows:

- The randomized *key generation* algorithm $\mathcal{K}$ returns a string K. We let $\mathsf{Keys}(\mathcal{SE})$ denote the set of all strings that have non-zero probability of being output by $\mathcal{K}$. The members of this set are called *keys*. We write $K \overset{\$}{\leftarrow} \mathcal{K}$ for the operation of executing $\mathcal{K}$ and letting K denote the key returned.
- The *encryption* algorithm $\mathcal{E}$, which might be randomized or stateful, takes a key $K \in \mathsf{Keys}(\mathcal{SE})$ and a *plaintext* $M \in \{0,1\}^*$ to return a *ciphertext* $C \in \{0,1\}^* \cup \{\perp\}$. We write $C \overset{\$}{\leftarrow} \mathcal{E}_K(M)$ for the operation of executing $\mathcal{E}$ on K and M and letting C denote the ciphertext returned.
- The deterministic *decryption* algorithm $\mathcal{D}$ takes a key $K \in \mathsf{Keys}(\mathcal{SE})$ and a ciphertext $C \in \{0,1\}^*$ to return some $M \in \{0,1\}^* \cup \{\perp\}$. We write $M \leftarrow \mathcal{D}_K(C)$ for the operation of executing $\mathcal{D}$ on K and C and letting M denote the message returned.

The scheme is said to provide *correct decryption* if for any key $K \in \mathsf{Keys}(\mathcal{SE})$, any sequence of messages $M_1, \ldots, M_q \in \{0,1\}^*$, and any sequence of ciphertexts $C_1 \overset{\$}{\leftarrow} \mathcal{E}_K(M_1), C_2 \overset{\$}{\leftarrow} \mathcal{E}_K(M_2), \ldots,$ $C_q \overset{\$}{\leftarrow} \mathcal{E}_K(M_q)$ that may arise in encrypting $M_1, \ldots, M_q$, it is the case that $\mathcal{D}_K(C_i) = M_i$ for each $C_i \neq \perp$.

The key-generation algorithm, as the definition indicates, is randomized. It takes no inputs. When it is run, it flips coins internally and uses these to select a key K. Typically, the key is just a random string of some length, in which case this length is called the *key length* of the scheme. When two parties want to use the scheme, it is assumed they are in possession of a key K generated via $\mathcal{K}$.

How they came into joint possession of this key K in such a way that the adversary did not get to know K is not our concern here, and will be addressed later. For now we assume the key has been shared.

Once in possession of a shared key, the sender can run the encryption algorithm with key K and input message M to get back a string we call the ciphertext. The latter can then be transmitted to the receiver.

The encryption algorithm may be either randomized or stateful. If randomized, it flips coins and uses those to compute its output on a given input K, M. Each time the algorithm is invoked, it flips coins anew. In particular, invoking the encryption algorithm twice on the same inputs may not yield the same response both times.

We say the encryption algorithm is *stateful* if its operation depends on a quantity called the *state* that is initialized in some pre-specified way. When the encryption algorithm is invoked on inputs K, M, it computes a ciphertext based on K, M and the current state. It then updates the state, and the new state value is stored. (The receiver does not maintain matching state and, in particular, decryption does not require access to any global variable or call for any synchronization between parties.) Usually, when there is state to be maintained, the state is just a counter. If there is no state maintained by the encryption algorithm the encryption scheme is said to be *stateless*.

The encryption algorithm might be both randomized and stateful, but in practice this is rare: it is usually one or the other but not both.

When we talk of a *randomized symmetric encryption scheme* we mean that the encryption algorithm is randomized. When we talk of a *stateful symmetric encryption scheme* we mean that the encryption algorithm is stateful.

The receiver, upon receiving a ciphertext C, will run the decryption algorithm with the same key used to create the ciphertext, namely compute $\mathcal{D}_K(C)$. The decryption algorithm is neither randomized nor stateful.

Many encryption schemes restrict the set of strings that they are willing to encrypt. (For example, perhaps the algorithm can only encrypt plaintexts of length a positive multiple of some block length n, and can only encrypt plaintexts of length up to some maximum length.) These kinds of restrictions are captured by having the encryption algorithm return the special symbol $\perp$ when fed a message not meeting the required restriction. In a stateless scheme, there is typically a set of strings $\mathcal{M}$, called the *plaintext space*, such that

$$M \in \mathcal{M} \text{ iff } \Pr[K \stackrel{\$}{\leftarrow} \mathcal{K};\ C \stackrel{\$}{\leftarrow} \mathcal{E}_K(M) :\ C \neq \perp] \ = \ 1$$

In a stateful scheme, whether or not $\mathcal{E}_K(M)$ returns $\perp$ depends not only on M but also possibly on the value of the state variable. For example, when a counter is being used, it is typical that there is a limit to the number of encryptions performed, and when the counter reaches a certain value the encryption algorithm returns $\perp$ no matter what message is fed to it.

The correct decryption requirement simply says that decryption works: if a message M is encrypted under a key K to yield a ciphertext C, then one can recover M by decrypting C under K. This holds, however, only if $C \neq \perp$. The condition thus says that, for each key $K \in \mathsf{Keys}(\mathcal{SE})$ and message $M \in \{0,1\}^*$, with probability one over the coins of the encryption algorithm, either the latter outputs $\perp$ or it outputs a ciphertext C which upon decryption yields M. If the scheme is stateful, this condition is required to hold for every value of the state.

Correct decryption is, naturally, a requirement before one can use a symmetric encryption scheme in practice, for if this condition is not met, the scheme fails to communicate information accurately. In analyzing the security of symmetric encryption schemes, however, we will see that it is sometimes useful to be able to consider ones that do not meet this condition.

5.2 Some symmetric encryption schemes

We now provide a few examples of encryption schemes. We stress that not all of the schemes that follow are *secure* encryption schemes. Some are secure and some are not, as we will see later. All the schemes here satisfy the correct decryption requirement.

5.2.1 The one-time-pad encryption scheme

We begin with the classical one-time-pad.

Scheme 5.2.1 [One-time-pad encryption] The one-time-pad encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ is stateful and deterministic. The key-generation algorithm simply returns a random k-bit string K, where the key-length k is a parameter of the scheme, so that the key space is $\mathsf{Keys}(\mathcal{SE}) = \{0,1\}^k$. The encryptor maintains a counter *ctr* which is initially zero. The encryption and decryption algorithms operate as follows:

```
algorithm $\mathcal{E}_K(M)$
    Let static $ctr \leftarrow 0$
    Let $m \leftarrow |M|$
    if $ctr + m > k$ then return $\perp$
    $C \leftarrow M \oplus K[ctr+1 \,..\, ctr+m]$
    $ctr \leftarrow ctr + m$
    return $\langle ctr - m, C \rangle$
```

```
algorithm $\mathcal{D}_K(\langle ctr, C \rangle)$
    Let $m \leftarrow |M|$
    if $ctr + m > k$ then return $\perp$
    $M \leftarrow C \oplus K[ctr+1 \,..\, ctr+m]$
    return $M$
```

Here $X[i \,..\, j]$ denotes the i-th through j-th bit of the binary string X. By $\langle ctr, C \rangle$ we mean a string that encodes the number *ctr* and the string C. The most natural encoding is to encode *ctr* using some fixed number of bits, at least $\lg k$, and to prepend this to C. Conventions are established so that every string Y is regarded as encoding some ctr, C for some ctr, C. The encryption algorithm XORs the message bits with key bits, starting with the key bit indicated by one plus the current counter value. The counter is then incremented by the length of the message. Key bits are not reused, and thus if not enough key bits are available to encrypt a message, the encryption algorithm returns $\perp$. Note that the ciphertext returned includes the value of the counter. This is to enable decryption. (Recall that the decryption algorithm, as per Definition 5.1.1, must be stateless and deterministic, so we do not want it to have to maintain a counter as well.) ▮

5.2.2 Some modes of operation

The following schemes rely either on a family of permutations (i.e., a blockcipher) or a family of functions. Effectively, the mechanisms spell out how to use the blockcipher to encrypt. We call such a mechanism a *mode of operation* of the blockcipher. For these schemes it is convenient to assume that the length of the message to be encrypted is a positive multiple of a block length associated to the family. In practice, one could pad the message appropriately so that the padded message always had length a positive multiple of the block length, and apply the encryption algorithm to the padded message. The padding function should be injective and easily invertible. In this way you would create a new encryption scheme.

```
algorithm $\mathcal{E}_K(M)$
    $M[1] \cdots M[m] \leftarrow M$
    for $i \leftarrow 1$ to $m$ do
        $C[i] \leftarrow E_K(M[i])$
    $C \leftarrow C[1] \cdots C[m]$
    return $C$
```

```
algorithm $\mathcal{D}_K(C)$
    $C[1] \cdots C[m] \leftarrow C$
    for $i \leftarrow 1$ to $m$ do
        $M[i] \leftarrow E_K^{-1}(C[i])$
    $M \leftarrow M[1] \cdots M[m]$
    return $M$
```

Figure 5.1: ECB mode.

```
algorithm $\mathcal{E}_K(M)$
$M[1] \cdots M[m] \leftarrow M$
$C[0] \stackrel{\$}{\leftarrow} \{0,1\}^n$
for $i = 1, \ldots, m$ do
    $C[i] \leftarrow E_K(M[i] \oplus C[i-1])$
return $C$
```

```
algorithm $\mathcal{D}_K(C)$
$C[0] \cdots C[m] \leftarrow C$
for $i = 1, \ldots, m$ do
    $M[i] \leftarrow E_K^{-1}(C[i]) \oplus C[i-1]$
return $M$
```

Figure 5.2: CBC$ mode.

With a block length n understood, we will denote by

$$X[i] \ldots X[m] \leftarrow X$$

the operation of parsing string X into $m-i+1$ blocks, each block of length n. Here $i \le m$ and X is assumed to have length $(m-i+1) \cdot n$. Thus, $X[j]$ consists of bits $(j-i)n+1$ to $(j-i+1)n$ of X, for $i \le j \le m$.

The first scheme we consider is ECB (Electronic Codebook Mode), whose security is considered in Section 5.5.1.

Scheme 5.2.2 [ECB mode] Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher. Operating it in ECB (Electronic Code Book) mode yields a stateless symmetric encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The key-generation algorithm simply returns a random key for the blockcipher, meaning it picks a random string $K \stackrel{\$}{\leftarrow} \mathcal{K}$ and returns it. The encryption and decryption algorithms are depicted in Fig. 5.1. Notice that this time the encryption algorithm did not make any random choices. (That does not mean it is not, technically, a randomized algorithm; it is simply a randomized algorithm that happened not to make any random choices.) ▌

The next scheme, cipher-block chaining (CBC) with random initial vector, is the most popular block-cipher mode of operation, used pervasively in practice.

```
algorithm $\mathcal{E}_K(M)$
$M[1] \cdots M[m] \leftarrow M$
$C[0] \leftarrow ctr$
for $i = 1, \ldots, m$ do
    $C[i] \leftarrow E_K(M[i] \oplus C[i-1])$
return $C$
```

```
algorithm $\mathcal{D}_K(C)$
$C[0] \cdots C[m] \leftarrow C$
for $i = 1, \ldots, m$ do
    $M[i] \leftarrow E_K^{-1}(C[i]) \oplus C[i-1]$
return $M$
```

Figure 5.3: CBCC mode.

Scheme 5.2.3 [CBC\$ mode] Let E: $\mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a blockcipher. Operating it in CBC mode with random IV yields a stateless symmetric encryption scheme, $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The key generation algorithm simply returns a random key for the blockcipher, $K \xleftarrow{\$} \mathcal{K}$. The encryption and decryption algorithms are depicted in Fig. 5.2. The IV ("initialization vector") is $C[0]$, which is chosen at random by the encryption algorithm. This choice is made independently each time the algorithm is invoked. ▮

For the following schemes it is useful to introduce some notation. With n fixed, we let $\langle i \rangle$ denote the n-bit string that is the binary representation of integer $i \bmod 2^n$. If we use a number $i \geq 0$ in a context for which a string $I \in \{0,1\}^n$ is required, it is understood that we mean to replace i by $I = [i]_n$. The following is a counter-based version of CBC mode, whose security is considered in Section 5.5.3.

Scheme 5.2.4 [CBCC mode] Let E: $\mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a blockcipher. Operating it in CBC mode with counter IV yields a stateful symmetric encryption scheme, $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The key generation algorithm simply returns a random key for the blockcipher, $K \xleftarrow{\$} \mathcal{K}$. The encryptor maintains a counter *ctr* which is initially zero. The encryption and decryption algorithms are depicted in Fig. 5.3. The IV ("initialization vector") is $C[0]$, which is set to the current value of the counter. The counter is then incremented each time a message is encrypted. The counter is a static variable, meaning that its value is preserved across invocations of the encryption algorithm. ▮

The CTR (counter) modes that follow are not much used, to the best of our knowledge, but perhaps wrongly so. We will see later that they have good privacy properties. In contrast to CBC, the encryption procedure is parallelizable, which can be exploited to speed up the process in the presence of hardware support. It is also the case that the methods work for strings of arbitrary bit lengths, without doing anything "special" to achieve this end. There are two variants of CTR mode, one random and the other stateful, and, as we will see later, their security properties are different. For security analyses see Section 5.7 and Section 5.10.1.

Scheme 5.2.5 [CTR\$ mode] Let F: $\mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a family of functions. (Possibly a blockcipher, but not necessarily.) Then CTR mode over F with a random starting point is a probabilistic, stateless symmetric encryption scheme, $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The key-generation algorithm simply returns a random key for E. The encryption and decryption algorithms are depicted in Fig. 5.4. The starting point $C[0]$ is used to define a sequence of values on which F_K is applied to produce a "pseudo one-time pad" to which the plaintext is XORed. The starting point $C[0]$ chosen by the encryption algorithm is a random n-bit string. To add an n-bit string $C[0]$ to an integer i—when we write $F_K(R + i)$—convert the n-bit string $C[0]$ into an integer in the range

```
algorithm $\mathcal{E}_K(M)$
$M[1] \cdots M[m] \leftarrow M$
$C[0] \xleftarrow{\$} \{0,1\}^n$
for $i = 1, \ldots, m$ do
    $P[i] \leftarrow F_K(C[0] + i)$
    $C[i] \leftarrow P[i] \oplus M[i]$
return $C$

algorithm $\mathcal{D}_K(C)$
$C[0] \cdots C[m] \leftarrow C$
for $i = 1, \ldots, m$ do
    $P[i] \leftarrow F_K(C[0] + i)$
    $M[i] \leftarrow P[i] \oplus C[i]$
return $M$
```

Figure 5.4: CTR\$ mode using a family of functions $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. This version of counter mode is randomized and stateless.

```
algorithm $\mathcal{E}_K(M)$
$M[1] \cdots M[m] \leftarrow M$
$C[0] \leftarrow$ ctr
for $i = 1, \ldots, m$ do
    $P[i] \leftarrow F_K(ctr + i)$
    $C[i] \leftarrow P[i] \oplus M[i]$
ctr $\leftarrow$ ctr + m
return $C$

algorithm $\mathcal{D}_K(C)$
$C[0] \cdots C[m] \leftarrow C$
ctr $\leftarrow C[0]$
for $i = 1, \ldots, m$ do
    $P[i] \leftarrow F_K(ctr + i)$
    $M[i] \leftarrow P[i] \oplus C[i]$
return $M$
```

Figure 5.5: CTRC mode using a family of functions $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. This version of counter mode uses stateful (but deterministic) encryption.

$[0 \mathinner{..} 2^n - 1]$ in the usual way, add this number to i, take the result modulo 2^n, and then convert this back into an n-bit string. Note that the starting point $C[0]$ is included in the ciphertext, to enable decryption. ▌

We now give the counter-based version of CTR mode.

Scheme 5.2.6 [CTRC mode] Let $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a family of functions (possibly a blockcipher, but not necessarily). Operating it in CTR mode with a counter starting point is a stateful symmetric encryption scheme, $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, which we call CTRC. The key-generation algorithm simply returns a random key for F. The encryptor maintains a counter *ctr* which is initially zero. The encryption and decryption algorithms are depicted in Fig. 5.5. Position index *ctr* is not allowed to wrap around: the encryption algorithm returns $\perp$ if this would happen. The position index is included in the ciphertext in order to enable decryption. The encryption algorithm updates the position index upon each invocation, and begins with this updated value the next time it is invoked. ▌

We will return to the security of these schemes after we have developed the appropriate notions.

5.3 Issues in privacy

Let us fix a symmetric encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. Two parties share a key K for this scheme, this key having being generated as $K \xleftarrow{\$} \mathcal{K}$. The adversary does not a priori know K. We now want to explore the issue of what the privacy of the scheme might mean. For this chapter, security *is* privacy, and we are trying to get to the heart of what security is about.

The adversary is assumed able to capture any ciphertext that flows on the channel between the two parties. It can thus collect ciphertexts, and try to glean something from them. Our first question is: what exactly does "glean" mean? What tasks, were the adversary to accomplish them, would make us declare the scheme insecure? And, correspondingly, what tasks, were the adversary unable to accomplish them, would make us declare the scheme secure?

It is easier to think about *insecurity* than security, because we can certainly identify adversary actions that indubitably imply the scheme is insecure. So let us begin here.

For example, if the adversary can, from a few ciphertexts, derive the underlying key K, it can later decrypt anything it sees, so if the scheme allowed easy key recovery from a few ciphertexts it is definitely insecure.

Now, the mistake that is often made is to go on to reverse this, saying that if key recovery is hard, then the scheme is secure. This is certainly not true, for there are other possible weaknesses. For example, what if, given the ciphertext, the adversary could easily recover the plaintext M without finding the key? Certainly the scheme is insecure then too.

So should we now declare a scheme secure if it is hard to recover a plaintext from the ciphertext? Many people would say yes. Yet, this would be wrong too.

One reason is that the adversary might be able to figure out *partial information* about M. For example, even though it might not be able to recover M, the adversary might, given C, be able to recover the first bit of M, or the sum of all the bits of M. This is not good, because these bits might carry valuable information.

For a concrete example, say I am communicating to my broker a message which is a sequence of "buy" or "sell" decisions for a pre-specified sequence of stocks. That is, we have certain stocks, numbered 1 through m, and bit i of the message is 1 if I want to buy stock i and 0 otherwise. The message is sent encrypted. But if the first bit leaks, the adversary knows whether I want to buy or sell stock 1, which may be something I don't want to reveal. If the sum of the bits leaks, the adversary knows how many stocks I am buying.

Granted, this might not be a problem at all if the data were in a different format. However, making assumptions, or requirements, on how users format data, or how they use it, is a bad and dangerous approach to secure protocol design. An important principle of good cryptographic design is that the encryption scheme should provide security regardless of the format of the plaintext. Users should not have to worry about the how they format their data: they format it as they like, and encryption should provide privacy nonetheless.

Put another way, as designers of security protocols, we should not make assumptions about data content or formats. Our protocols must protect any data, no matter how formatted. We view it as the job of the protocol designer to ensure this is true.

At this point it should start becoming obvious that there is an infinite list of insecurity properties, and we can hardly attempt to characterize security as their absence. We need to think about security in a different and more direct way and arrive at some definition of it.

This important task is surprisingly neglected in many treatments of cryptography, which will provide you with many schemes and attacks, but never actually define the goal by saying what an encryption scheme is actually trying to achieve and when it should be considered secure rather than merely not known to be insecure. This is the task that we want to address.

One might want to say something like: the encryption scheme is secure if given C, the adversary has no idea what M is. This however cannot be true, because of what is called *a priori* information. Often, something about the message is known. For example, it might be a packet with known headers. Or, it might be an English word. So the adversary, and everyone else, has some information about the message even before it is encrypted.

We want schemes that are secure in the strongest possible natural sense. What is the best we could hope for? It is useful to make a thought experiment. What would an "ideal" encryption be like? Well, it would be as though some angel took the message M from the sender and delivered it to the receiver, in some magic way. The adversary would see nothing at all. Intuitively, our goal is to approximate this as best as possible. We would like encryption to have the properties of ideal encryption. In particular, no partial information would leak.

We do deviate from the ideal in one way, though. Encryption is not asked to hide the length of the plaintext string. This information not only can leak but is usually supposed to be known to the adversary a priori.

As an example, consider the ECB encryption scheme of Scheme 5.2.2. Given the ciphertext, can an eavesdropping adversary figure out the message? It is hard to see how, since it does not know K, and if F is a "good" blockcipher, then it ought to have a hard time inverting F_K without knowledge of the underlying key. Nonetheless this is not a good scheme. Consider just the case $n = 1$ of a single block message. Suppose a missile command center has just two messages, 1^n for *fire* and 0^n for *don't fire*. It keeps sending data, but always one of these two. What happens? When the first ciphertext C_1 goes by, the adversary may not know what is the plaintext. But then, let us say it sees a missile taking off. Now, it knows the message M_1 underlying C_1 was 1^n. But then it can easily decrypt all subsequent messages, for if it sees a ciphertext C, the message is 1^n if $C = C_1$ and 0^n if $C \neq C_1$.

In a secure encryption scheme, it should not be possible to relate ciphertexts of different messages of the same length in such a way that information is leaked.

Not allowing message-equalities to be leaked has a dramatic implication. Namely, *encryption must be probabilistic* or *depend on state information*. If not, you can always tell if the same message was sent twice. Each encryption must use fresh coin tosses, or, say, a counter, and an encryption of a particular message may be different each time. In terms of our setup it means $\mathcal{E}$ is a *probabilistic* or *stateful* algorithm. That's why we defined symmetric encryption schemes, above, to allow these types of algorithms.

The reason this is dramatic is that it goes in many ways against the historical or popular notion of encryption. Encryption was once thought of as a code, a fixed mapping of plaintexts to ciphertexts. But this is not the contemporary viewpoint. A single plaintext should have many possible ciphertexts (depending on the random choices or the state of the encryption algorithm). Yet it must be possible to decrypt. How is this possible? We have seen several examples above.

One formalization of privacy is what is called *perfect security*, an information-theoretic notion introduced by Shannon and showed by him to be met by the one-time pad scheme, and covered in Chapter 2. Perfect security asks that regardless of the computing power available to the adversary, the ciphertext provides it no information about the plaintext beyond the a priori information it had prior to seeing the ciphertext. Perfect security is a very strong attribute, but achieving it requires a key as long as the total amount of data encrypted, and this is not usually practical. So here we look at a notion of *computational security*. The security will only hold with respect to adversaries of limited computing power. If the adversary works harder, she can figure out more, but a "feasible" amount of effort yields no noticeable information. This is the important notion for us and will be used to analyze the security of schemes such as those presented above.

5.4 Indistinguishability under chosen-plaintext attack

Having discussed the issues in Section 5.3 above, we will now distill a formal definition of security.

5.4.1 Definition

The basic idea behind indistinguishability (or, more fully, *left-or-right indistinguishability under a chosen-plaintext attack*) is to consider an adversary (not in possession of the secret key) who chooses two messages of the same length. Then one of the two messages is encrypted, and the ciphertext is given to the adversary. The scheme is considered secure if the adversary has a hard time telling which of the two messages was the one encrypted.

We will actually give the adversary a little more power, letting her choose a whole sequence of pairs of equal-length messages. Let us now detail the game.

The adversary chooses a sequence of pairs of messages, $(M_{0,1}, M_{1,1}), \ldots, (M_{0,q}, M_{1,q})$, where, in each pair, the two messages have the same length. We give to the adversary a sequence of ciphertexts $C_1, \ldots, C_q$ where either (1) C_i is an encryption of $M_{0,i}$ for all $1 \leq i \leq q$ or, (2) C_i is an encryption of $M_{1,i}$ for all $1 \leq i \leq q$. In doing the encryptions, the encryption algorithm uses the same key but fresh coins, or an updated state, each time. The adversary gets the sequence of ciphertexts and now it must guess whether $M_{0,1}, \ldots, M_{0,q}$ were encrypted or $M_{1,1}, \ldots, M_{1,q}$ were encrypted.

To further empower the adversary, we let it choose the sequence of message pairs via a *chosen plaintext attack*. This means that the adversary chooses the first pair, then receives C_1, then chooses the second pair, receives C_2, and so on. (Sometimes this is called an *adaptive* chosen-plaintext attack, because the adversary can adaptively choose each query in a way responsive to the earlier answers.)

Let us now formalize this. We fix some encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. It could be either stateless or stateful. We consider an adversary A. It is a program which has access to an oracle that we call LR (left or right) oracle. A can provide as input any pair of equal-length messages. The oracle will return a ciphertext. We will consider two possible ways in which this ciphertext is computed by the oracle, corresponding to two possible "worlds" in which the adversary "lives". In the "right" world, the oracle, given query M_0, M_1, runs $\mathcal{E}$ with key K and input M_1 to get a ciphertext C which it returns. In the "left" world, the oracle, given M_0, M_1, runs $\mathcal{E}$ with key K and input M_0 to get a ciphertext C which it returns. The problem for the adversary is, after talking to its oracle for some time, to tell which of the two oracles it was given. The formalization uses the game $\mathrm{Left}_{\mathcal{SE}}$ and $\mathrm{Right}_{\mathcal{SE}}$ of Fig. 5.6. The game begins by picking at random encryption key K. (The key is not returned to the adversary.) Each game then defines a **LR** oracle, to which the adversary may make multiple queries, each query being a pair of equal-length strings. The adversary outputs a bit which become the output of the game.

First assume the given symmetric encryption scheme $\mathcal{SE}$ is stateless. The oracle, in either world, is probabilistic, because it calls the encryption algorithm. Recall that this algorithm is probabilistic. Above, when we say $C \xleftarrow{\$} \mathcal{E}_K(M_b)$, it is implicit that the oracle picks its own random coins and uses them to compute ciphertext C.

The random choices of the encryption function are somewhat "under the rug" here, not being explicitly represented in the notation. But these random bits should not be forgotten. They are central to the meaningfulness of the notion and the security of the schemes.

If the given symmetric encryption scheme $\mathcal{SE}$ is stateful, the oracles, in either world, become stateful, too. (Think of a subroutine that maintains a "static" variable across successive calls.) An oracle begins with a state value initialized to a value specified by the encryption scheme. For example, in CTRC mode, the state is an integer *ctr* that is initialized to 0. Now, each time the oracle is invoked, it computes $\mathcal{E}_K(M_b)$ according to the specification of algorithm $\mathcal{E}$. The algorithm may, as a side-effect, update the state, and upon the next invocation of the oracle, the new state value will be used.

```
Game Left$_{\mathcal{SE}}$
procedure Initialize
$K \xleftarrow{\$} \mathcal{K}$
procedure LR$(M_0, M_1)$
Return $C \xleftarrow{\$} \mathcal{E}_K(M_0)$
```

```
Game Right$_{\mathcal{SE}}$
procedure Initialize
$K \xleftarrow{\$} \mathcal{K}$
procedure LR$(M_0, M_1)$
Return $C \xleftarrow{\$} \mathcal{E}_K(M_1)$
```

Figure 5.6: Games used to define IND-CPA.

The following definition associates to a symmetric encryption scheme $\mathcal{SE}$ and an adversary A a pair of experiments, one capturing each of the worlds described above. The adversary's advantage, which measures its success in breaking the scheme, is the difference in probabilities of the two games returning the bit one.

Definition 5.4.1 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, and let A be an algorithm that has access to an oracle. We consider two games as described in Fig. 5.6. The *ind-cpa advantage* of A is defined as

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) = \Pr\left[\text{Right}^A_{\mathcal{SE}} \Rightarrow 1\right] - \Pr\left[\text{Left}^A_{\mathcal{SE}} \Rightarrow 1\right]$$

As the above indicates, the choice of which world we are in is made just once, at the beginning, before the adversary starts to interact with the oracle. In the left world, *all* message pairs sent to the oracle are answered by the oracle encrypting the left message in the pair, while in the right world, all message pairs are answered by the oracle encrypting the right message in the pair. The choice of which does not flip-flop from oracle query to oracle query.

If $\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A)$ is small (meaning close to zero), it means that A is outputting 1 about as often in world 0 as in world 1, meaning it is not doing a good job of telling which world it is in. If this quantity is large (meaning close to one—or at least far from zero) then the adversary A is doing well, meaning our scheme $\mathcal{SE}$ is not secure, at least to the extent that we regard A as "reasonable."

Informally, for symmetric encryption scheme $\mathcal{SE}$ to be secure against chosen plaintext attack, the ind-cpa advantage of an adversary must be small, no matter what strategy the adversary tries. However, we have to be realistic in our expectations, understanding that the advantage may grow as the adversary invests more effort in its attack. Security is a measure of how large the advantage of the adversary might when compared against the adversary's resources.

We consider an encryption scheme to be "secure against chosen-plaintext attack" if an adversary restricted to using "practical" amount of resources (computing time, number of queries) cannot obtain "significant" advantage. The technical notion is called left-or-right indistinguishability under chosen-plaintext attack, denoted IND-CPA.

We discuss some important conventions regarding the resources of adversary A. The *running time* of an adversary A is the worst case execution time of A over all possible coins of A and all conceivable oracle return values (including return values that could never arise in the experiments used to define the advantage). Oracle queries are understood to return a value in unit time, but it takes the adversary one unit of time to read any bit that it chooses to read. By convention, the running time of A also includes the size of the code of the adversary A, in some fixed RAM model of computation. This convention for measuring time complexity is the same as used in other parts of these notes, for all kinds of adversaries.

Other resource conventions are specific to the IND-CPA notion. When the adversary asks its left-or-right encryption oracle a query (M_0, M_1) we say that length of this query is $\max(|M_0|, |M_1|)$.

```
Game IND-CPA_SE                    procedure LR(M_0, M_1)
                                   return $C \xleftarrow{\$} \mathcal{E}_K(M_b)$
procedure Initialize               procedure Finalize(b')
$K \xleftarrow{\$} \mathcal{K}$ ; $b \xleftarrow{\$} \{0,1\}$       return $(b = b')$
```

Figure 5.7: Game used to redefine IND-CPA.

(This will equal $|M_0|$ since we require that any query consist of equal-length messages.) The total length of queries is the sum of the length of each query. We can measure query lengths in bits or in blocks, with block having some understood number of bits n.

The resources of the adversary we will typically care about are three. First, its time-complexity, measured according to the convention above. Second, the number of oracle queries, meaning the number of message pairs the adversary asks of its oracle. These messages may have different lengths, and our third resource measure is the sum of all these lengths, denoted μ, again measured according to the convention above.

5.4.2 Alternative interpretation

Let us move on to describe a somewhat different interpretation of indistinguishability. Why is $\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A)$ called the "advantage" of the adversary? We can view the task of the adversary as trying to guess which world it is in. A trivial guess is for the adversary to return a random bit. In that case, it has probability 1/2 of being right. Clearly, it has not done anything damaging in this case. The advantage of the adversary measures how much better than this it does at guessing which world it is in, namely the excess over 1/2 of the adversary's probability of guessing correctly. In this subsection we will see how the above definition corresponds to this alternative view, a view that lends some extra intuition to the definition and is also useful in later usages of the definition.

Proposition 5.4.2 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, and let A be an algorithm that has access to an oracle that takes input a pair of strings and returns a string. We consider the game as described in Fig. 5.7. Then

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) = 2 \cdot \Pr\left[\text{IND-CPA}^{A}_{\mathcal{SE}} \Rightarrow \mathsf{true}\right] - 1 \, .$$

In the above game, the **LR** oracle provided to an adversary A is as in $\text{Left}_{\mathcal{SE}}$ if $b = 0$ and as in $\text{Right}_{\mathcal{SE}}$ if $b = 1$, where the bit b is chosen at random. A eventually outputs a bit b', its guess as to the value of b. Thus,

$$\Pr\left[\text{IND-CPA}^{A}_{\mathcal{SE}} \Rightarrow \mathsf{true}\right]$$

is the probability that A correctly guesses which world it is in. The probability is over the initial choice of world as given by the bit b, the choice of K, the random choices of $\mathcal{E}_K(\cdot)$ if any, and the coins of A if any. This value is 1/2 when the adversary deserves no advantage, since one can guess b correctly by a strategy as simple as "always answer zero" or "answer with a random bit." The "advantage" of A can thus be viewed as the excess of this probability over 1/2, which, re-scaled, is

$$2 \cdot \Pr\left[\mathbf{Exp}^{\text{ind-cpa-cg}}_{\mathcal{SE}}(A) = 1\right] - 1 \, .$$

The Proposition says that this rescaled advantage is exactly the same measure as before.

Proof of Proposition 5.4.2: We let $\Pr[\cdot]$ be the probability of event "·" in the game $\text{IND-CPA}^{A}_{\mathcal{SE}}$, and refer below to quantities in this game. The claim of the Proposition follows by a straightforward calculation:

$$\begin{aligned}
&\Pr\left[\text{IND-CPA}_{\mathcal{SE}}^{A}\Rightarrow\mathsf{true}\right] \\
&\quad= \Pr\left[b=b'\right] \\
&\quad= \Pr\left[b=b' \mid b=1\right]\cdot\Pr\left[b=1\right]+\Pr\left[b=b' \mid b=0\right]\cdot\Pr\left[b=0\right] \\
&\quad= \Pr\left[b=b' \mid b=1\right]\cdot\frac{1}{2}+\Pr\left[b=b' \mid b=0\right]\cdot\frac{1}{2} \\
&\quad= \Pr\left[b'=1 \mid b=1\right]\cdot\frac{1}{2}+\Pr\left[b'=0 \mid b=0\right]\cdot\frac{1}{2} \\
&\quad= \Pr\left[b'=1 \mid b=1\right]\cdot\frac{1}{2}+\left(1-\Pr\left[b'=1 \mid b=0\right]\right)\cdot\frac{1}{2} \\
&\quad= \frac{1}{2}+\frac{1}{2}\cdot\left(\Pr\left[b'=1 \mid b=1\right]-\Pr\left[b'=1 \mid b=0\right]\right) \\
&\quad= \frac{1}{2}+\frac{1}{2}\cdot\left(\Pr\left[\text{Right}_{\mathcal{SE}}^{A}\Rightarrow 1\right]-\Pr\left[\text{Left}_{\mathcal{SE}}^{A}\Rightarrow 1\right]\right) \\
&\quad= \frac{1}{2}+\frac{1}{2}\cdot\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(A)\ .
\end{aligned}$$

We began by expanding the quantity of interest via standard conditioning. The term of 1/2 in the third line emerged because the choice of b is made at random. In the fourth line we noted that if we are asking whether $b = b'$ given that we know $b = 1$, it is the same as asking whether $b' = 1$ given $b = 1$, and analogously for $b = 0$. In the fifth line and sixth lines we just manipulated the probabilities and simplified. The next line is important; here we observed that the conditional probabilities in question are exactly the probabilities that A returns 1 in the games of Definition 5.4.1. ▌

5.4.3 Why is this a good definition?

Our thesis is that we should consider an encryption scheme to be "secure" if and only if it is IND-CPA secure, meaning that the above formalization captures our intuitive sense of privacy, and the security requirements that one might put on an encryption scheme can be boiled down to this one.

But why? Why does IND-CPA capture "privacy"? This is an important question to address and answer.

In particular, here is one concern. In Section 5.3 we noted a number of security properties that are necessary but not sufficient for security. For example, it should be computationally infeasible for an adversary to recover the key from a few plaintext-ciphertext pairs, or to recover a plaintext from a ciphertext.

A test of our definition is that it implies the necessary properties that we have discussed, and others. For example, a scheme that is secure in the IND-CPA sense of our definition should also be, automatically, secure against key-recovery or plaintext-recovery. Later, we will prove such things, and even stronger things. For now, let us continue to get a better sense of how to work with the definition by using it to show that certain schemes are insecure.

5.5 Example chosen-plaintext attacks

We illustrate the use of our IND-CPA definition in finding attacks by providing an attack on ECB mode, and also a general attack on deterministic, stateless schemes.

5.5.1 Attack on ECB

Let us fix a blockcipher E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. The ECB symmetric encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ was described as Scheme 5.2.2. Suppose an adversary sees a ciphertext $C = \mathcal{E}_K(M)$ corresponding to some random plaintext M, encrypted under the key K also unknown to the adversary. Can the adversary recover M? Not easily, if E is a "good" blockcipher. For example if E is AES, it seems quite infeasible. Yet, we have already discussed how infeasibility of recovering plaintext from ciphertext is not an indication of security. ECB has other weaknesses. Notice that if two plaintexts M and M' agree in the first block, then so do the corresponding ciphertexts. So an adversary, given the ciphertexts, can tell whether or not the first blocks of the corresponding plaintexts are the same. This is loss of partial information about the plaintexts, and is not permissible in a secure encryption scheme.

It is a test of our definition to see that it captures these weaknesses and also finds the scheme insecure. It does. To show this, we want to show that there is an adversary that has a high IND-CPA advantage while using a small amount of resources. We now construct such an adversary A. Remember that A is given a lr-encryption oracle $\mathbf{LR}(\cdot,\cdot)$ that takes as input a pair of messages and that returns an encryption of either the left or the right message in the pair, depending on the value of the bit b. The goal of A is to determine the value of b. Our adversary works like this:

```
Adversary $A$
    $M_1 \leftarrow 0^{2n}$ ; $M_0 \leftarrow 0^n \| 1^n$
    $C[1]C[2] \leftarrow \mathbf{LR}(M_0, M_1)$
    If $C[1] = C[2]$ then return 1 else return 0
```

Above, $X[i]$ denotes the i-th block of a string X, a block being a sequence of n bits. The adversary's single oracle query is the pair of messages M_0, M_1. Since each of them is two blocks long, so is the ciphertext computed according to the ECB scheme. Now, we claim that

$$\Pr\left[\mathrm{Right}_{\mathcal{SE}}^{A} \Rightarrow 1\right] = 1 \text{ and}$$

$$\Pr\left[\mathrm{Left}_{\mathcal{SE}}^{A} \Rightarrow 1\right] = 0 .$$

Why? You have to return to the definitions of the quantities in question. In the right world, the oracle returns $C[1]C[2] = E_K(0^n) \| E_K(0^n)$, so $C[1] = C[2]$ and A returns 1. In the left world, the oracle returns $C[1]C[2] = E_K(0^n)E_K(1^n)$. Since E_K is a permutation, $C[1] \neq C[2]$. So A returns 0 in this case.

Subtracting, we get $\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) = 1 - 0 = 1$. And A achieved this advantage by making just one oracle query, whose length, which as per our conventions is just the length of M_0, is $2n$ bits. This means that the ECB encryption scheme is insecure.

As an exercise, try to analyze the same adversary as an adversary against CBC\$ or CTR modes, and convince yourself that the adversary will not get a high advantage.

There is an important feature of this attack that must be emphasized. Namely, ECB is an insecure encryption scheme *even if the underlying blockcipher E is highly secure*. The weakness is not in the tool being used (here the blockcipher) but in the manner we are using it. It is the ECB mechanism that is at fault. Even the best of tools are useless if you don't know how to properly use them.

This is the kind of design flaw that we want to be able to spot and eradicate. Our goal is to find symmetric encryption schemes that are secure as long as the underlying blockcipher is secure. In other words, the scheme has no inherent flaw; as long as you use good ingredients, the recipe will produce a good meal.

If you don't use good ingredients? Well, that is your problem. All bets are off.

5.5.2 Any deterministic, stateless schemes is insecure

ECB mode is deterministic and stateless, so that if the same message is encrypted twice, the same ciphertext is returned. It turns out that this property, in general, results in an insecure scheme, and provides perhaps a better understanding of why ECB fails. Let us state the general fact more precisely.

Proposition 5.5.1 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a deterministic, stateless symmetric encryption scheme. Assume there is an integer m such that the plaintext space of the scheme contains two distinct strings of length m. Then there is an adversary A such that

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) \;=\; 1 \,.$$

Adversary A runs in time $O(m)$ and asks just two queries, each of length m. ▌

The requirement being made on the message space is minimal; typical schemes have messages spaces containing all strings of lengths between some minimum and maximum length, possibly restricted to strings of some given multiples. Note that this Proposition applies to ECB and is enough to show the latter is insecure.

Proof of Proposition 5.5.1: We must describe the adversary A. Remember that A is given an lr-encryption oracle $\mathbf{LR}(\cdot, \cdot)$ that takes input a pair of messages and returns an encryption of either the left or the right message in the pair. The goal of A is to determine which. Our adversary works like this:

```
Adversary $A$
    Let $X, Y$ be distinct, $m$-bit strings in the plaintext space
    $C_1 \leftarrow \mathbf{LR}(X, Y)$
    $C_2 \leftarrow \mathbf{LR}(Y, Y)$
    If $C_1 = C_2$ then return 1 else return 0
```

Now, we claim that

$$\begin{aligned} \Pr\left[\text{Right}^{A}_{\mathcal{SE}} \Rightarrow 1\right] &= 1 \text{ and} \\ \Pr\left[\text{Left}^{A}_{\mathcal{SE}} \Rightarrow 1\right] &= 0 \,. \end{aligned}$$

Why? In the right world, the oracle returns $C_1 = \mathcal{E}_K(Y)$ and $C_2 = \mathcal{E}_K(Y)$, and since the encryption function is deterministic and stateless, $C_1 = C_2$, so A returns 1. In the left world, the oracle returns $C_1 = \mathcal{E}_K(X)$ and $C_2 = \mathcal{E}_K(Y)$, and since it is required that decryption be able to recover the message, it must be that $C_1 \neq C_2$. So A returns 0.

Subtracting, we get $\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) = 1 - 0 = 1$. And A achieved this advantage by making two oracle queries, each of whose length, which as per our conventions is just the length of the first message, is m bits. ▌

5.5.3 Attack on CBC encryption with counter IV

Let us fix a blockcipher E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding counter-based version of the CBC encryption mode described in Scheme 5.2.4. We show that this scheme is insecure. The reason is that the adversary can predict the counter value.

To justify our claim of insecurity, we present an adversary A. As usual it is given an lr-encryption oracle $\mathbf{LR}(\cdot,\cdot)$ and wants to determine whether the left or right message is being encrypted. Our adversary works like this:

```
Adversary $A$
    $M_{0,1} \leftarrow 0^n$ ; $M_{1,1} \leftarrow 0^n$
    $M_{0,2} \leftarrow 0^n$ ; $M_{1,2} \leftarrow 0^{n-1}1$
    $C_1 \stackrel{\$}{\leftarrow} \mathbf{LR}(M_{0,1}, M_{1,1})$
    $C_2 \stackrel{\$}{\leftarrow} \mathbf{LR}(M_{0,2}, M_{1,2})$
    If $C_1 = C_2$ then return 1 else return 0
```

We claim that

$$\begin{aligned} \Pr\left[\text{Right}_{\mathcal{SE}}^{A} \Rightarrow 1\right] &= 1 \text{ and} \\ \Pr\left[\text{Left}_{\mathcal{SE}}^{A} \Rightarrow 1\right] &= 0 \,. \end{aligned}$$

Why? First consider the left world,. In that case $C_1[0] = 0$ and $C_2[0] = 1$ and $C_1[1] = E_K(0)$ and $C_2[1] = E_K(1)$ and so $C_1 \neq C_2$ and A returns 0. On the other hand, if we are in the right world, then $C_1[0] = 0$ and $C_2[0] = 1$ and $C_1[1] = E_K(0)$ and $C_2[1] = E_K(0)$, so A returns 1.

Subtracting, we get $\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(A) = 1 - 0 = 1$, showing that A has a very high advantage. Moreover, A is practical, using very few resources. So the scheme is insecure.

5.6 Semantic security

In this section we describe an alternative notion of encryption-scheme security, *semantic security*, again under a chosen-plaintext attack. We will abbreviate this notion as SEM-CPA. It captures the idea that a secure encryption scheme should hide all information about an unknown plaintext. This definition may match our intuition about what secure encryption ought to achieve better than does IND-CPA. We then show that IND-CPA implies SEM-CPA. (In fact, they are equivalent.) By showing that our IND-CPA notion implies SEM-CPA we gain confidence that our definition appropriately models privacy.

Semantic security, which was introduced by Goldwasser and Micali for public-key encryption, transfers the intuition of Shannon's notion of security to a setting where security is not absolute but dependent on the computational effort made by an adversary. Shannon says that an encryption scheme is secure if *that which can be determined about a plaintext from its ciphertext can be determined in the absence of the ciphertext.* Semantic security asks that *that which can be efficiently computed about some plaintexts from their ciphertexts can be computed, just as easily, in the absence of those ciphertexts.*

Our formalization allows an adversary to choose a message space $\mathcal{M}$ from which messages may be drawn, and to specify a function f on messages. Messages M and M' are drawn independently and at random from the message space $\mathcal{M}$. We consider two worlds. In the first, the adversary will attempt to compute $f(M)$ given an encryption of M. In the second, the adversary will attempt to compute $f(M)$ given an encryption of M' (that is, the adversary is given no information related to M). The scheme is secure if it succeeds about as often in the second game as the first, no matter what (reasonable) f and $\mathcal{M}$ the adversary selects.

To make our definition as general as possible, we will actually let the adversary choose, in sequence, message spaces $\mathcal{M}_1, \ldots, \mathcal{M}_q$. From each message space $\mathcal{M}_i$ draw the message M_i at random, and then let C_i be a random encryption of M_i. Adaptively querying, the adversary obtains the vector of ciphertexts $(C_1, \ldots, C_q)$. Now the adversary tries to find a function f such that it can do a good job at predicting $f(M_1, \ldots, M_q)$. Doing a good job means predicting this value significantly better than how well the adversary would predict it had it been given *no* information about $M_1, \ldots, M_q$: each C_i was not the encryption of M_i but the encryption of a random point M_i' from $\mathcal{M}_i$. The formal definition now follows.

Definition 5.6.1 [Semantic security] Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, and let A be an algorithm that has access to an oracle. We consider the following experiments:

```
Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{ss-cpa-1}}(A)$
  $K \stackrel{\$}{\leftarrow} \mathcal{K}$,  $s \stackrel{\$}{\leftarrow} \varepsilon$
  for $i \leftarrow 1$ to $q$ do
    $(\mathcal{M}_i, s) \stackrel{\$}{\leftarrow} A(s)$
    $M_i, M_i' \stackrel{\$}{\leftarrow} \mathcal{M}_i$
    if $|M_i| \neq |M_i'|$ then $M_i \leftarrow M_i' \leftarrow \varepsilon$
    $C_i \stackrel{\$}{\leftarrow} \mathcal{E}_K(M_i)$;  $s \leftarrow \langle s, C_i \rangle$
  $(f, Y) \stackrel{\$}{\leftarrow} A(s)$
  return $f(M_1, \ldots, M_q) = Y$
```

```
Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{ss-cpa-0}}(A)$
  $K \stackrel{\$}{\leftarrow} \mathcal{K}$,  $s \stackrel{\$}{\leftarrow} \varepsilon$
  for $i \leftarrow 1$ to $q$ do
    $(\mathcal{M}_i, s) \stackrel{\$}{\leftarrow} A(s)$
    $M_i, M_i' \stackrel{\$}{\leftarrow} \mathcal{M}_i$
    if $|M_i| \neq |M_i'|$ then $M_i \leftarrow M_i' \leftarrow \varepsilon$
    $C_i \stackrel{\$}{\leftarrow} \mathcal{E}_K(M_i')$;  $s \leftarrow \langle s, C_i \rangle$
  $(f, Y) \stackrel{\$}{\leftarrow} A(s)$
  return $f(M_1, \ldots, M_q) = Y$
```

The SEM-CPA advantage of A is defined as

$$\mathbf{Adv}_{\mathcal{SE}}^{\text{sem-cpa}}(A) \quad = \quad \Pr\left[\mathbf{Exp}_{\mathcal{SE}}^{\text{ss-cpa-1}}(A) \Rightarrow 1\right] - \Pr\left[\mathbf{Exp}_{\mathcal{SE}}^{\text{ss-cpa-0}}(A) \Rightarrow 1\right] \; . \blacksquare$$

In the definition above, each experiment initializes its oracle by choosing a random key K. A total of q times, the adversary chooses a message space $\mathcal{M}_i$. The message space is specified by an always-halting probabilistic algorithm, written in some fixed programming language. The code for this algorithm is what the adversary actually outputs. Each time the message space is output, two random samples are drawn from this message space, M_i and M_i'. We expect that M_i and M_i' to have the same length, and if they don't we "erase" both strings. The encryption of one of these messages will be returned to the adversary. Which string gets encrypted depends on the experiment: M_i for experiment 1 and M_i' for experiment 0. By f we denote a deterministic function. It is described by an always-halting program and, as before, it actually the program for f that the adversary outputs. By Y we denote a string. The string s represents saved state that the adversary may wish to retain.

In speaking of the running time of A, we include, beyond the actual running time, the maximal time to draw two samples from each message space $\mathcal{M}$ that A outputs, and we include the maximal time to compute $f(M_1, \ldots, M_q)$ over any vector of strings. In speaking of the length of A's queries we sum, over all the message spaces output by A, the maximal length of a string M output with nonzero probability by $\mathcal{M}$, and we sum also over the lengths of the encodings of each messages space, function f, and string Y output by A.

We emphasize that the above would seem to be an exceptionally strong notion of security. We have given the adversary the ability to choose the message spaces from which each message will be drawn. We have let the adversary choose the partial information about the messages that it finds convenient to predict. We have let the adversary be fully adaptive. We have built in the ability to perform a chosen-message attack (simply by producing an algorithm $\mathcal{M}$ that samples one and only one point). Despite all this, we now show that security in the indistinguishability sense implies semantic security.

Theorem 5.6.2 ***[IND-CPA $\Rightarrow$ SEM-CPA]*** *Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme and let A be an adversary (for attacking the SEM-CPA security of $\mathcal{SE}$) that runs in time at most t and asks at most q queries, these queries totaling at most μ bits. Then there exists and adversary B (for attacking the IND-CPA security of $\mathcal{SE}$) that achieves advantage*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(B) \quad \geq \quad \mathbf{Adv}^{\text{sem-cpa}}_{\mathcal{SE}}(A)$$

and where B runs in time $t + O(\mu)$ and asks at most q queries, these queries totaling μ bits.

Proof: The adversary B, which has oracle g, is constructed as follows.

```
algorithm $B^g$
    $s \xleftarrow{\$} \varepsilon$
    for $i \leftarrow 1$ to $q$ do
        $(\mathcal{M}_i, s) \xleftarrow{\$} A(s)$
        $M_i, M'_i \xleftarrow{\$} \mathcal{M}_i$
        if $|M_i| \neq |M'_i|$ then $M_i \leftarrow M'_i \leftarrow \varepsilon$
        $C_i \leftarrow g(M'_i, M_i)$,  $s \leftarrow \langle s, C_i \rangle$
    $(f, Y) \xleftarrow{\$} A(s)$
    if $f(M_1, \ldots, M_q) = Y$ then return 1 else return 0
```

Suppose first that g is instantiated by a right encryption oracle—an oracle that returns $C \xleftarrow{\$} \mathcal{E}_K(M)$ in response to a query (M', M). Then the algorithm above coincides with experiment $\mathbf{Exp}^{\text{ss-cpa-1}}_{\mathcal{SE}}(A)$. Similarly, if g is instantiated by a left encryption oracle—the oracle it returns $C \xleftarrow{\$} \mathcal{E}_K(M')$ in response to a query (M', M)—then the algorithm above coincides with experiment $\mathbf{Exp}^{\text{ss-cpa-0}}_{\mathcal{SE}}(A)$. It follows that $\mathbf{Adv}^{\text{sem-cpa}}_{\mathcal{SE}}(B) = \mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A)$. To complete the theorem, note that B's running time is A's running time plus $O(\mu)$ and B asks a total of q queries, these having total length at most the total length of A's queries, under our convention. ▌

5.7 Security of CTR modes

Recall that the CTR (counter) mode of operation of a family of functions comes in two variants: the randomized (stateless) version CTRC of Scheme 5.2.5, and the counter-based (stateful) mechanism CTR\$ of Scheme 5.2.6. Both modes achieve indistinguishability under a chosen-plaintext attack, but, interestingly, the quantitative security is a little different. The difference springs from the fact that CTRC achieves *perfect* indistinguishability if one uses a random function in the role of the underlying F_K—but CTR\$ would not achieve perfect indistinguishability even then, because of the possibility that collisions would produce "overlaps" in the pseudo-one-time pad.

We will state the main theorems about the schemes, discuss them, and then prove them. For the counter version we have:

Theorem 5.7.1 ***[Security of CTRC mode]*** *Let $F\colon\ \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a family of functions and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding CTRC symmetric encryption scheme as described in Scheme 5.2.6. Let A be an adversary (for attacking the IND-CPA security of $\mathcal{SE}$) that runs in time at most t and asks at most q queries, these totaling at most σ n-bit blocks. Then there exists an adversary B (attacking the PRF security of F) such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) \quad \leq \quad 2 \cdot \mathbf{Adv}^{\text{prf}}_{F}(B)\ .$$

Furthermore B runs in time at most $t' = t + O(q + n\sigma)$ and asks at most $q' = \sigma$ oracle queries. ▌

Theorem 5.7.2 ***[Security of CTR\$ mode]*** *Let $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding CTR\$ symmetric encryption scheme as described in Scheme 5.2.5. Let A be an adversary (for attacking the IND-CPA security of $\mathcal{SE}$) that runs in time at most t and asks at most q queries, these totaling at most σ n-bit blocks. Then there exists an adversary B (attacking the PRF security of F) such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) \quad \leq \quad \mathbf{Adv}^{\text{prf}}_{F}(B) + \frac{0.5\,\sigma^2}{2^n} \, .$$

Furthermore B runs in time at most $t' = t + O(q + n\sigma)$ and asks at most $q' = \sigma$ oracle queries. ▌

The above theorems exemplify the kinds of results that the provable-security approach is about. Namely, we are able to provide provable guarantees of security of some higher level cryptographic construct (in this case, a symmetric encryption scheme) based on the assumption that some building block (in this case an underlying block) is secure. The above results are the first example of the "punch-line" we have been building towards. So it is worth pausing at this point and trying to make sure we really understand what these theorems are saying and what are their implications.

If we want to entrust our data to some encryption mechanism, we want to know that this encryption mechanism really provides privacy. If it is ill-designed, it may not. We saw this happen with ECB. Even if we used a secure blockcipher, the flaws of ECB mode make it an insecure encryption scheme.

Flaws are not apparent in CTR at first glance. But maybe they exist. It is very hard to see how one can be convinced they do *not* exist, when one cannot possible exhaust the space of all possible attacks that could be tried. Yet this is exactly the difficulty that the above theorems circumvent. They are saying that CTR mode *does not have design flaws.* They are saying that as long as you use a good blockcipher, you are *assured* that nobody will break your encryption scheme. One cannot ask for more, since if one does not use a good blockcipher, there is no reason to expect security of your encryption scheme anyway. We are thus getting a conviction that *all attacks fail* even though we do not even know exactly how these attacks might operate. That is the power of the approach.

Now, one might appreciate that the ability to make such a powerful statement takes work. It is for this that we have put so much work and time into developing the definitions: the formal notions of security that make such results meaningful. For readers who have less experience with definitions, it is worth knowing, at least, that the effort is worth it. It takes time and work to understand the notions, but the payoffs are big: you get significant guarantees of security.

How, exactly, are the theorems saying this? The above discussion has pushed under the rug the quantitative aspect that is an important part of the results. It may help to look at a concrete example.

Example 5.7.3 Let us suppose that F is the blockcipher AES, so that $n = 128$. Suppose I want to encrypt $q = 2^{30}$ messages, each being one kilobyte (2^{13} bits) long. I am thus encrypting a total of 2^{43} bits, which is to say $\sigma = 2^{36}$ blocks. (This is about one terabyte). Can I do this securely using CTR\$? Let A be an adversary attacking the privacy of my encryption. Theorem 5.7.2 says that there exists a B satisfying the stated conditions. How large can $\mathbf{Adv}^{\text{prf}}_{\mathsf{AES}}(B)$ be? It makes $q = 2^{36}$ queries, and it is consistent with our state of knowledge of the security of AES to assume that such an adversary cannot do better than mount a birthday attack, meaning its advantage is no more than $q^2/2^{128}$. Under such an assumption, the theorem tells us that $\mathbf{Adv}^{\text{rnd-cpa}}_{\mathcal{SE}}(A)$ is at most $\sigma^2/2^{128} + 0.5\,\sigma^2/2^{128} = 1.5\,2^{72}/2^{128} \leq 1/2^{55}$. This is a very small number indeed, saying that

```
Game $G_0$

procedure Initialize
$K \xleftarrow{\$} \{0,1\}^k; b \xleftarrow{\$} \{0,1\}$
$ctr \leftarrow 0$

procedure LR$(M_0, M_1)$
$C[0] \leftarrow ctr; m \leftarrow \|M_b\|_n$
for $i = 1, ..., m$ do
    $T[ctr + i] \leftarrow F_K(ctr + i)$
    $C[i] \leftarrow T[ctr + i] \oplus M_b[i]$
$ctr \leftarrow ctr + m$
return $C$

procedure Finalize$(b')$
return $(b = b')$
```

```
Game $G_1$

procedure Initialize
$b \xleftarrow{\$} \{0,1\}; ctr \leftarrow 0$

procedure LR$(M_0, M_1)$
$C[0] \leftarrow ctr; m \leftarrow \|M_b\|_n$
for $i = 1, ..., m$ do
    $T[ctr + i] \xleftarrow{\$} \{0,1\}^n$
    $C[i] \leftarrow T[ctr + i] \oplus M_b[i]$
$ctr \leftarrow ctr + m$
return $C$

procedure Finalize$(b')$
return $(b = b')$
```

```
Game $G_2$

procedure Initialize
$b \xleftarrow{\$} \{0,1\}; ctr \leftarrow 0$

procedure LR$(M_0, M_1)$
$C[0] \leftarrow ctr; m \leftarrow \|M_0\|_n$
for $i = 1, ..., m$ do
    $C[i] \xleftarrow{\$} \{0,1\}^n$
return C

procedure Finalize$(b')$
return $(b = b')$
```

Figure 5.8: Games used to prove Theorem 5.7.1.

our encryption is secure, at least under the assumption that the best attack on the PRF security of AES is a birthday attack. Note however that if we encrypt 2^{64} blocks of data, our provable-security bound becomes meaningless. ▌

The example illustrates how to use the theorems to figure out how much security you will get from the CTR encryption scheme in a given application.

Note that as per the above theorems, encrypting more than $\sigma = 2^{n/2}$ blocks of data with CTR\$ is not secure regardless of the quality of F as a PRF. On the other hand, with CTRC, it might be secure, as long as F can withstand σ queries. This is an interesting and possibly useful distinction. Yet, in the setting in which such modes are usually employed, the distinction all but vanishes. For usually F is a blockcipher and in that case, we know from the birthday attack that the prf-advantage of B may itself be as large as $\Theta(\sigma^2/2^n)$, and thus, again, encrypting more than $\sigma = 2^{n/2}$ blocks of data is not secure. However, we might be able to find or build function families F that are not families of permutations and preserve PRF security against adversaries making more than $2^{n/2}$ queries.

Proof of Theorem 5.7.1: We consider the games G_0, G_1, G_2 of Fig. 5.8. We now build prf-adversary B so that

$$\Pr\left[G_0^A \Rightarrow 1\right] - \Pr\left[G_1^A \Rightarrow 1\right] = \mathbf{Adv}_F^{\mathrm{prf}}(B). \tag{5.1}$$

Additionally, B's resource usage (running time and query count) is as claimed in the theorem.

Remember that B has access to an oracle **Fn**: $\{0,1\}^n \to \{0,1\}^n$. B runs A, answering any queries that A makes to its **LR** oracle via the subroutine LRSim. This subroutine uses B's own oracle **Fn**.

```
Adversary $B$
    $b \xleftarrow{\$} \{0,1\}$; $ctr \leftarrow 0$;
    $b' \xleftarrow{\$} A^{\mathrm{LRSim}}$
    If $(b = b')$ then return 1
    Else return 0

subroutine LRSim$(M_0, M_1)$
    $C[0] \leftarrow ctr; m \leftarrow \|M_b\|_n$
    for $i = 1, ..., m$ do
        $T[ctr + i] \leftarrow \mathbf{Fn}(ctr + i)$
        $C[i] \leftarrow T[ctr + i] \oplus M_b[i]$
    $ctr \leftarrow ctr + m$
    return C
```

If $\mathbf{Fn} = F_K$ then B is providing A the environment of game G_0 so

$$\Pr\left[\mathrm{Real}_F^B{\Rightarrow}1\right] = \Pr\left[G_0^A{\Rightarrow}1\right]$$

If $\mathbf{Fn}$ is random then B is providing A the environment of game G_1 so

$$\Pr\left[\mathrm{Rand}_F^B{\Rightarrow}1\right] = \Pr G_1^A{\Rightarrow}1$$

Thus

$$\begin{aligned}\mathbf{Adv}_F^{\mathrm{prf}}(B) &= \Pr\left[\mathrm{Real}_F^B{\Rightarrow}1\right] - \Pr\left[\mathrm{Rand}_F^B{\Rightarrow}1\right] \\ &= \Pr\left[G_0^A{\Rightarrow}1\right] - \Pr\left[G_1^A{\Rightarrow}1\right]\end{aligned}$$

which proves equation (5.1). Obviously,

$$\Pr[G_0^A{\Rightarrow}1] = \Pr[G_1^A{\Rightarrow}1] + \Pr[G_0^A{\Rightarrow}1] - \Pr[G_1^A{\Rightarrow}1]$$

So, using equation (5.1) we have

$$\begin{aligned}\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(A) &= 2 \cdot \Pr[G_0^A{\Rightarrow}1] - 1 \\ &= 2 \cdot \Big(\Pr[G_1^A{\Rightarrow}1] + \mathbf{Adv}_F^{\mathrm{prf}}(B) \Big) - 1 \\ &= 2 \cdot \mathbf{Adv}_F^{\mathrm{prf}}(B) + 2\Pr[G_1^A{\Rightarrow}1] - 1\end{aligned}$$

We now claim that

$$\Pr[G_1^A{\Rightarrow}1] = \tfrac{1}{2}.$$

Combining this with the above we have

$$\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(A) \;\leq\; 2 \cdot \mathbf{Adv}_F^{\mathrm{prf}}(B).$$

The above claim is justified by observing that

$$\Pr\left[G_1^A{\Rightarrow}1\right] = \Pr\left[G_2^A{\Rightarrow}1\right] \text{ and } \Pr\left[G_2^A{\Rightarrow}1\right] = \tfrac{1}{2}.$$

▮

```
Game $G_0$

procedure Initialize
$K \xleftarrow{\$} \{0,1\}^k; b \xleftarrow{\$} \{0,1\}; S \leftarrow \emptyset$

procedure LR$(M_0, M_1)$
$m \leftarrow \|M_b\|_n$; $C[0] \xleftarrow{\$} \{0,1\}^n$
for $i = 1, ..., n$ do
   $P \leftarrow C[i-1] \oplus M_b[i]$
   if $P \in S$ then
      $\mathsf{T}[P] \leftarrow E_K(P)$
   $C[i] \leftarrow T[P_{j,i}]$
$S \leftarrow S \cup \{P\}$
return $C$

procedure Finalize$(b')$
return $(b = b')$
```

```
Game $G_1$

procedure Initialize
$b \xleftarrow{\$} \{0,1\}$ ; $S \leftarrow \emptyset$

procedure LR$(M_0, M_1)$
$m \leftarrow \|M_b\|_n$; $C[0] \xleftarrow{\$} \{0,1\}^n$
for $i = 1, ..., n$ do
   $P \leftarrow C[i-1] \oplus M_b[i]$
   If $P \notin S$ then $\mathsf{T}[P] \xleftarrow{\$} \{0,1\}^n$
   $C[i] \leftarrow \mathsf{T}[P]$
$S \leftarrow S \cup \{P\}$
return $C$

procedure Finalize$(b')$
return $(b = b')$
```

Figure 5.9: Games used to prove Theorem 5.8.1.

5.8 Security of CBC with a random IV

In this section we show that CBC encryption using a random IV is IND-CPA secure as long as E is a blockcipher that is a secure PRF or PRP. Namely we show:

Theorem 5.8.1 ***[Security of CBC\$ mode]*** *Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding CBC\$ symmetric encryption scheme as described in Scheme 5.2.5. Let A be an adversary (for attacking the IND-CPA security of $\mathcal{SE}$) that runs in time at most t and asks at most q queries, these totaling at most σ n-bit blocks. Then there exists an adversary B (attacking the PRF security of E) such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) \quad \leq \quad 2 \cdot \mathbf{Adv}^{\text{prf}}_{E}(B) + \frac{\sigma^2}{2^n} \ .$$

Furthermore B runs in time at most $t' = t + O(q + n\sigma)$ and asks at most $q' = \sigma$ oracle queries. ▌

Proof: The proof uses games G_0, G_1 of Fig. 5.9 and games G_2, G_3, G_4 of Fig. 5.10. Then we have

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) = 2 \cdot \Pr\left[G_0^A \Rightarrow \mathsf{true}\right] - 1.$$

But

$$\Pr\left[G_0^A \Rightarrow \mathsf{true}\right] = \Pr\left[G_1^A \Rightarrow \mathsf{true}\right] + \left(\Pr\left[G_0^A \Rightarrow \mathsf{true}\right] - \Pr\left[G_1^A \Rightarrow \mathsf{true}\right]\right)$$

We now build prf-adversary B so that

$$\Pr\left[G_0^A \Rightarrow \mathsf{true}\right] - \Pr\left[G_1^A \Rightarrow \mathsf{true}\right] \leq \mathbf{Adv}^{\text{prf}}_{E}(B).$$

Additionally, B's resource usage (query count and running time) is as claimed in the theorem.

The prf-adversary B works as follows:

```
Adversary $B$
$b \xleftarrow{\$} \{0,1\}$ ; $S \leftarrow \emptyset$
$b' \xleftarrow{\$} A^{\mathbf{LR}}$
if $(b = b')$ then return 1
else return 0
```

```
subroutine LRSim($M_0, M_1$)
$m \leftarrow \|M_b\|_n$; $C[0] \xleftarrow{\$} \{0,1\}^n$
for $i = 1, ..., m$ do
    $P \leftarrow C[i-1] \oplus M_b[i]$
    if $P \notin S$ then $T[P] \leftarrow \mathbf{Fn}(P)$
    $C[i] \leftarrow T[P]$
$S \leftarrow S \cup \{P\}$
return $C$
```

It is easy to see that

$$\begin{aligned} \Pr\left[\text{Real}_E^B \Rightarrow 1\right] &= \Pr\left[G_0^A \Rightarrow \mathsf{true}\right] \\ \Pr\left[\text{Rand}_E^B \Rightarrow 1\right] &= \Pr\left[G_1^A \Rightarrow \mathsf{true}\right] \end{aligned}$$

So we have

$$\Pr\left[G_0^A \Rightarrow \mathsf{true}\right] - \Pr\left[G_1^A \Rightarrow \mathsf{true}\right] = \Pr\left[\text{Real}_E^B \Rightarrow 1\right] - \Pr\left[\text{Rand}_E^B \Rightarrow 1\right] \leq \mathbf{Adv}_E^{\text{prf}}(B).$$

We are going to prove that $\Pr\left[G_1^A \Rightarrow \mathsf{true}\right] \leq \frac{1}{2} + \sigma^2 \cdot 2^{-n-1}$.

Assuming this for now, we will have

$$\begin{aligned} \mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(A) &\leq 2 \cdot \left(\frac{1}{2} + \frac{\sigma^2}{2^{n+1}}\right) - 1 + 2\mathbf{Adv}_E^{\text{prf}}(B) \\ &= \frac{\sigma^2}{2^n} + 2\mathbf{Adv}_E^{\text{prf}}(B) \end{aligned}$$

In the following, we use games G_2, G_3, G_4 of Fig. 5.10.

$$\begin{aligned} \Pr[G_1^A \Rightarrow \mathsf{true}] &= \Pr[G_2^A \Rightarrow \mathsf{true}] \\ &= \Pr[G_3^A \Rightarrow \mathsf{true}] + (\Pr[G_2^A \Rightarrow \mathsf{true}] - \Pr[G_3^A \Rightarrow \mathsf{true}]) \end{aligned}$$

We will show that

- $\Pr[G_3^A \Rightarrow \mathsf{true}] = \frac{1}{2}$
- $\Pr[G_2^A \Rightarrow \mathsf{true}] - \Pr[G_3^A \Rightarrow \mathsf{true}] \leq \frac{\sigma^2}{2^{n+1}}$

Ciphertext C in G_3 is always random, independently of b, so

$$\Pr\left[G_3^A \Rightarrow \mathsf{true}\right] = \frac{1}{2}.$$

G_2 and G_3 are identical-until-bad, so Fundamental Lemma implies

$$\Pr\left[G_2^A \Rightarrow \mathsf{true}\right] - \Pr\left[G_3^A \Rightarrow \mathsf{true}\right] \leq \Pr\left[G_3^A \text{ sets } \mathsf{bad}\right].$$

```
Game $\boxed{G_2}$, $G_3$

procedure Initialize
$b \xleftarrow{\$} \{0,1\}$ ; $S \leftarrow \emptyset$

procedure LR$(M_0, M_1)$
$m \leftarrow \|M_b\|_n$; $C[0] \xleftarrow{\$} \{0,1\}^n$
for $i = 1, ..., n$ do
   $P \leftarrow C[i-1] \oplus M_b[i]$
   $C[i] \xleftarrow{\$} \{0,1\}^n$
   If $P \in S$ then
      $\mathsf{bad} \leftarrow \mathsf{true}$ ; $\boxed{C[i] \leftarrow \mathsf{T}[P]}$
   $\mathsf{T}[P] \leftarrow C[i]$
$S \leftarrow S \cup \{P\}$
return $C$

procedure Finalize$(b')$
return $(b = b')$
```

```
Game $G_4$

procedure Initialize
$j \leftarrow 0$; $S \leftarrow \emptyset$

procedure LR$(M_0, M_1)$
$m \leftarrow \|M_0\|_n$
for $i = 1, ..., m$ do
   $P \xleftarrow{\$} \{0,1\}^n$
   $C[i] \xleftarrow{\$} \{0,1\}^n$
   If $P \in S$ then $\mathsf{bad} \leftarrow \mathsf{true}$
$S \leftarrow S \cup \{P\}$

procedure Finalize$(b')$
return $(b = b')$
```

Figure 5.10: Games used to prove Theorem 5.8.1.

$$\Pr\left[G_3^A \text{ sets } \mathsf{bad}\right] = \Pr\left[G_4^A \text{ sets } \mathsf{bad}\right]$$

The ℓ-th time the if-statement is executed, it has probability

$$\frac{\ell - 1}{2^n}$$

of setting bad. Thus

$$\begin{aligned}\Pr\left[G_4^A \text{ sets } \mathsf{bad}\right] &\le \sum_{\ell=1}^{\sigma} \frac{\ell-1}{2^n} \\ &= \frac{\sigma(\sigma-1)}{2^{n+1}} \\ &\le \frac{\sigma^2}{2^{n+1}} .\end{aligned}$$

The theorem follows. ▮

We have so far shown that the IND-CPA advantage of CBC\$ falls off by an amount that is at most quadratic in the number of blocks, σ, asked by the adversary. We can also give a matching attack, showing that there actually *is* an adversary that obtains advantage of about $\sigma^2/2^n$. This tells us that our security result is *tight*—there is no possibility of making the bound significantly better. It means that we have arrived at reasonably precise understanding of the security of CBC encryption with a random IV.

Proposition 5.8.2 Let $n \geq 1$, let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a function family, and let $\sigma \in [0 \,..\, \sqrt{2}\, 2^{n/2} - 1]$. Then there is an adversary A that asks a single query, the query consisting of σ blocks, runs in time $O(n\sigma \lg(\sigma))$, and achieves advantage $\mathbf{Adv}^{\text{ind-cpa}}_{\text{CBC}[E]}(A) \geq 0.15\, \sigma^2/2^n$ and

Proof: The adversary A sets $L \leftarrow 0^{n\sigma}$, chooses $R \stackrel{\$}{\leftarrow} \{0,1\}^{n\sigma}$, and asks its oracle the query (L, R), receiving in response a ciphertext C that it partitions into $\sigma + 1$ n-bit blocks, $C_0 C_1 \ldots C_\sigma$. If there is an $i, I \in [0 \,..\, \sigma]$ such that $i < I$ and $C_i = C_I$ then A selects the lexicographically first such (i, I) and answers 1 (for "right oracle") if $C_{i+1} \neq C_{I+1}$. (In this case the adversary has found a "proof" that the oracle is a right oracle.) In all other cases the adversary outputs 0 (for "left oracle").

The adversary described asks asks a single query of σ blocks and, using standard data structure techniques, it runs in time $O(n\sigma \lg(\sigma))$. It remains to calculate the adversary's advantage, $\mathbf{Adv}^{\text{prp}}_E(A) = \Pr[A^{\text{Right}(\cdot,\cdot)} \Rightarrow 1] - \Pr[A^{\text{Left}(\cdot,\cdot)} \Rightarrow 1]$. The second summand is zero since when A is given a left encryption-oracle that oracle is encrypting the zero-string and any time $C_i = C_I$ we *must* have that $C_{i+1} = C_{I+1}$ as well. Thus

$$\begin{aligned}
\mathbf{Adv}^{\text{prp}}_E(A) &= \Pr[A^{\text{Right}(\cdot,\cdot)} \Rightarrow 1] \\
&= \Pr[R \stackrel{\$}{\leftarrow} \{0,1\}^{n\sigma};\ K \stackrel{\$}{\leftarrow} \mathcal{K};\ \text{IV} \stackrel{\$}{\leftarrow} \{0,1\}^n;\ C \stackrel{\$}{\leftarrow} \text{CBC}^{\text{IV}}_K(R) : \\
&\qquad \exists i < I \text{ s.t. } C_i = C_I \text{ and } C_{i+1} \neq C_{I+1} \text{ on the first such } (i, I)]
\end{aligned}$$

By the structure of CBC mode with a random IV it is easy to see that that when you encrypt a random string $R \in \{0,1\}^{n\sigma}$ you get a random string $C \in \{0,1\}^{n(\sigma+1)}$. To see this, note that to make block C_i, for $i \geq 1$, you xor the random block R_i with C_i and apply the blockcipher. The random block R_i is independent of C_i—it wasn't even consulted in making C_i—and it is independent of all of $C_0, \ldots, C_{i-1}$, too. The image of a uniformly selected value under a permutation is uniform. The very first block of ciphertext, C_0, is uniform. This makes the entire string $C_0 C_1 \cdots C_\sigma$ uniform. So the probability in question is

$$\begin{aligned}
\mathbf{Adv}^{\text{prp}}_E(A) &= \Pr[C \stackrel{\$}{\leftarrow} \{0,1\}^{n(\sigma+1)} : \\
&\qquad \exists i < I \text{ s.t. } C_i = C_I \text{ and } C_{i+1} \neq C_{I+1} \text{ on the first such } (i, I)]
\end{aligned}$$

Now the birthday bound (Appendix A, Theorem A.0.1) tells us that the probability there will be an $i < I$ such that $C_i = C_I$ is at least $C(2^n, \sigma + 1) \geq 0.3\sigma^2/2^n$. When there is such an i, I and we fix the lexicographically first such i, I, note that C_{I+1} is still uniform and independent of C_{i+1}. Independence is assured because C_{I+1} is obtained as $E_K(R_{I+1} \oplus C_I)$ for a permutation E_K and a uniform random value R_{I+1} that is independent of C_I and C_{i+1}. Because of this probabilistic independence, the probability of the conjunct is just the product of the probabilities and we have that

$$\mathbf{Adv}^{\text{prp}}_E(A) \;\geq\; 0.3\, \sigma^2/2^n \cdot (1 - 2^{-n}) \;\geq\; 0.15\, \sigma^2/2^n$$

completing the proof. ▌

5.9 Indistinguishability under chosen-ciphertext attack

So far we have considered privacy under chosen-plaintext attack. Sometimes we want to consider privacy when the adversary is capable of mounting a stronger type of attack, namely a chosen-ciphertext attack. In this type of attack, an adversary has access to a decryption oracle. It can feed this oracle a ciphertext and get back the corresponding plaintext.

How might such a situation arise? One situation one could imagine is that an adversary at some point gains temporary access to the equipment performing decryption. It can feed the equipment ciphertexts and see what plaintexts emerge. (We assume it cannot directly extract the key from the equipment, however.)

If an adversary has access to a decryption oracle, security at first seems moot, since after all it can decrypt anything it wants. To create a meaningful notion of security, we put a restriction on the use of the decryption oracle. To see what this is, let us look closer at the formalization. As in the case of chosen-plaintext attacks, we consider two worlds. In the left world, *all* message pairs sent to the oracle are answered by the oracle encrypting the left message in the pair, while in the right world, all message pairs are answered by the oracle encrypting the right message in the pair. The choice of which does not flip-flop from oracle query to oracle query.

The adversary's goal is the same as in the case of chosen-plaintext attacks: it wants to figure out which world it is in. There is one easy way to do this. Namely, query the lr-encryption oracle on two distinct, equal length messages M_0, M_1 to get back a ciphertext C, and now call the decryption oracle on C. If the message returned by the decryption oracle is M_0 then the adversary is in world 0, and if the message returned by the decryption oracle is M_1 then the adversary is in world 1. The restriction we impose is simply that this call to the decryption oracle is not allowed. More generally, call a query C to the decryption oracle *illegitimate* if C was previously returned by the lr-encryption oracle; otherwise a query is *legitimate.* We insist that only legitimate queries are allowed. In the formalization below, the experiment simply returns 0 if the adversary makes an illegitimate query. (We clarify that a query C is legitimate if C is returned by the lr-encryption oracle *after* C was queried to the decryption oracle.)

This restriction still leaves the adversary with a lot of power. Typically, a successful chosen-ciphertext attack proceeds by taking a ciphertext C returned by the lr-encryption oracle, modifying it into a related ciphertext C', and querying the decryption oracle with C'. The attacker seeks to create C' in such a way that its decryption tells the attacker what the underlying message M was. We will see this illustrated in Section 5.10 below.

The model we are considering here might seem quite artificial. If an adversary has access to a decryption oracle, how can we prevent it from calling the decryption oracle on certain messages? The restriction might arise due to the adversary's having access to the decryption equipment for a limited period of time. We imagine that after it has lost access to the decryption equipment, it sees some ciphertexts, and we are capturing the security of these ciphertexts in the face of previous access to the decryption oracle. Further motivation for the model will emerge when we see how encryption schemes are used in protocols. We will see that when an encryption scheme is used in many authenticated key-exchange protocols the adversary effectively has the ability to mount chosen-ciphertext attacks of the type we are discussing. For now let us just provide the definition and exercise it.

Definition 5.9.1 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, let A be an algorithm that has access to two oracles, and let b be a bit. We consider the following experiment:

```
Game Left_SE                        | Game Right_SE
                                    |
procedure Initialize                | procedure Initialize
$K \xleftarrow{\$} \mathcal{K}$     | $K \xleftarrow{\$} \mathcal{K}$
                                    |
procedure LR($M_0, M_1$)            | procedure LR($M_0, M_1$)
Return $C \xleftarrow{\$} \mathcal{E}_K(M_0)$ | Return $C \xleftarrow{\$} \mathcal{E}_K(M_0)$
                                    |
procedure Dec($C$)                  | procedure Dec($C$)
return $M \leftarrow \mathcal{D}_K(C)$ | return $M \leftarrow \mathcal{D}_K(C)$
```

The *IND-CCA advantage* of A is defined as

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(A) \;=\; \Pr\left[\text{Right}^A_{\mathcal{SE}} \Rightarrow 1\right] - \Pr\left[\text{Left}^A_{\mathcal{SE}} \Rightarrow 1\right] \;. \blacksquare$$

The conventions with regard to resource measures are the same as those used in the case of chosen-plaintext attacks. In particular, the length of a query M_0, M_1 to the lr-encryption oracle is defined as the length of M_0.

We consider an encryption scheme to be "secure against chosen-ciphertext attack" if a "reasonable" adversary cannot obtain "significant" advantage in distinguishing the cases $b = 0$ and $b = 1$ given access to the oracles, where reasonable reflects its resource usage. The technical notion is called indistinguishability under chosen-ciphertext attack, denoted IND-CCA.

5.10 Example chosen-ciphertext attacks

Chosen-ciphertext attacks are powerful enough to break all the standard modes of operation, even those like CTR and CBC that are secure against chosen-plaintext attack. The one-time pad scheme is also vulnerable to a chosen-ciphertext attack: our notion of perfect security only took into account chosen-plaintext attacks. Let us now illustrate a few chosen-ciphertext attacks.

5.10.1 Attacks on the CTR schemes

Let $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a family of functions and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the associated CTR\$ symmetric encryption scheme as described in Scheme 5.2.5. The weakness of the scheme that makes it susceptible to a chosen-ciphertext attack is the following. Say $\langle r, C\rangle$ is a ciphertext of some n-bit message M, and we flip bit i of C, resulting in a new ciphertext $\langle r, C'\rangle$. Let M' be the message obtained by decrypting the new ciphertext. Then M' equals M with the i-th bit flipped. (You should check that you understand why.) Thus, by making a decryption oracle query of $\langle r, C'\rangle$ one can learn M' and thus M. In the following, we show how this idea can be applied to break the scheme in our model by figuring out in which world an adversary has been placed.

Proposition 5.10.1 Let $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a family of functions and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding CTR\$ symmetric encryption scheme as described in Scheme 5.2.5. Then

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(t, 1, n, 1, 2n) \;=\; 1$$

for $t = O(n)$ plus the time for one application of F.

The advantage of this adversary is 1 even though it uses hardly any resources: just one query to each oracle. That is clearly an indication that the scheme is insecure.

Proof of Proposition 5.10.1: We will present an adversary algorithm A, having time-complexity t, making 1 query to its lr-encryption oracle, this query being of length n, making 1 query to its decryption oracle, this query being of length $2n$, and having

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(A) \;=\; 1\,.$$

The Proposition follows.

Remember that the lr-encryption oracle $\mathcal{E}_K(\mathbf{LR}(\cdot,\cdot,b))$ takes input a pair of messages, and returns an encryption of either the left or the right message in the pair, depending on the value of b. The goal of A is to determine the value of b. Our adversary works like this:

```
Adversary $A^{\mathbf{LR}(\cdot,\cdot),\mathbf{Dec}(\cdot)}$
    $M_0 \leftarrow 0^n$ ; $M_1 \leftarrow 1^n$
    $\langle r, C\rangle \leftarrow \mathbf{LR}(M_0, M_1)$
    $C' \leftarrow C \oplus 1^n$
    $M \leftarrow \mathbf{Dec}(\langle r, C'\rangle)$
    If $M = M_0$ then return 1 else return 0
```

The adversary's single lr-encryption oracle query is the pair of distinct messages M_0, M_1, each one block long. It is returned a ciphertext $\langle r, C\rangle$. It flips the bits of C to get C' and then feeds the ciphertext $\langle r, C\rangle$ to the decryption oracle. It bets on world 1 if it gets back M_0, and otherwise on world 0. Notice that $\langle r, C'\rangle \neq \langle r, C\rangle$, so the decryption query is legitimate. Now, we claim that

$$\begin{aligned}\Pr\left[\text{Right}^A_{\mathcal{SE}}\Rightarrow 1\right] &= 1\\ \Pr\left[\text{Left}^A_{\mathcal{SE}}\Rightarrow 1\right] &= 0\,.\end{aligned}$$

Hence $\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(A) = 1-0 = 1$. And A achieved this advantage by making just one lr-encryption oracle query, whose length, which as per our conventions is just the length of M_0, is n bits, and just one decryption oracle query, whose length is $2n$ bits (assuming an encoding of $\langle r, X\rangle$ as $n+|X|$-bits). So $\mathbf{Adv}^{\text{pr-cpa}}_{\mathcal{SE}}(t, 1, n, 1, 2n) = 1$.

Why are the two equations claimed above true? You have to return to the definitions of the quantities in question, as well as the description of the scheme itself, and walk it through. In world 1, meaning $b = 1$, let $\langle r, C\rangle$ denote the ciphertext returned by the lr-encryption oracle. Then

$$C \;=\; F_K(r+1) \oplus M_1 \;=\; F_K(r+1) \oplus 1^n\,.$$

Now notice that

$$\begin{aligned}M &= \mathcal{D}_K(\langle r, C'\rangle)\\ &= F_K(r+1) \oplus C'\\ &= F_K(r+1) \oplus C \oplus 1^n\\ &= F_K(r+1) \oplus (F_K(r+1) \oplus 1^n) \oplus 1^n\\ &= 0^n\\ &= M_0\,.\end{aligned}$$

Thus, the decryption oracle will return M_0, and A will return 1. In world 0, meaning $b = 0$, let $\langle r, C[1]\rangle$ denote the ciphertext returned by the lr-encryption oracle. Then

$$C \;=\; F_K(r+1) \oplus M_0 \;=\; F_K(r+1) \oplus 0^n \;.$$

Now notice that

$$\begin{aligned} M &= \mathcal{D}_K(\langle r, C'\rangle) \\ &= F_K(r+1) \oplus C' \\ &= F_K(r+1) \oplus C \oplus 1^n \\ &= F_K(r+1) \oplus (F_K(r+1) \oplus 0^n) \oplus 1^n \\ &= 1^n \\ &= M_1 \;. \end{aligned}$$

Thus, the decryption oracle will return M_1, and A will return 0, meaning will return 1 with probability zero. ▌

An attack on CTRC (cf. Scheme 5.2.6) is similar, and is left to the reader.

5.10.2 Attack on CBC$

Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the associated CBC$ symmetric encryption scheme as described in Scheme 5.2.3. The weakness of the scheme that makes it susceptible to a chosen-ciphertext attack is the following. Say $\langle \mathrm{IV}, C[1]\rangle$ is a ciphertext of some n-bit message M, and we flip bit i of the IV, resulting in a new ciphertext $\langle \mathrm{IV}', C[1]\rangle$. Let M' be the message obtained by decrypting the new ciphertext. Then M' equals M with the i-th bit flipped. (You should check that you understand why by looking at Scheme 5.2.3.) Thus, by making a decryption oracle query of $\langle \mathrm{IV}', C[1]\rangle$ one can learn M' and thus M. In the following, we show how this idea can be applied to break the scheme in our model by figuring out in which world an adversary has been placed.

Proposition 5.10.2 Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the corresponding CBC$ encryption scheme as described in Scheme 5.2.3. Then

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(t, 1, n, 1, 2n) \;=\; 1$$

for $t = O(n)$ plus the time for one application of F.

The advantage of this adversary is 1 even though it uses hardly any resources: just one query to each oracle. That is clearly an indication that the scheme is insecure.

Proof of Proposition 5.10.2: We will present an adversary A, having time-complexity t, making 1 query to its lr-encryption oracle, this query being of length n, making 1 query to its decryption oracle, this query being of length $2n$, and having

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(A) \;=\; 1 \;.$$

The proposition follows.

Remember that the lr-encryption oracle $\mathcal{E}_K(\mathbf{LR}(\cdot,\cdot,b))$ takes input a pair of messages, and returns an encryption of either the left or the right message in the pair. The goal of A is to determine which. Our adversary works like this:

```
Adversary $A^{\mathbf{LR}(\cdot,\cdot),\mathbf{Dec}(\cdot)}$
    $M_0 \leftarrow 0^n$ ; $M_1 \leftarrow 1^n$
    $\langle \mathrm{IV}, C[1] \rangle \leftarrow \mathbf{LR}(M_0, M_1)$
    $\mathrm{IV}' \leftarrow \mathrm{IV} \oplus 1^n$
    $M \leftarrow \mathbf{Dec}(\langle \mathrm{IV}', C[1] \rangle)$
    If $M = M_0$ then return 1 else return 0
```

The adversary's single lr-encryption oracle query is the pair of distinct messages M_0, M_1, each one block long. It is returned a ciphertext $\langle \mathrm{IV}, C[1] \rangle$. It flips the bits of the IV to get a new IV, IV', and then feeds the ciphertext $\langle \mathrm{IV}', C[1] \rangle$ to the decryption oracle. It bets on world 1 if it gets back M_0, and otherwise on world 0. It is important that $\langle \mathrm{IV}', C[1] \rangle \neq \langle \mathrm{IV}, C[1] \rangle$ so the decryption oracle query is legitimate. Now, we claim that

$$\begin{aligned}\Pr\left[\mathrm{Right}_{\mathcal{SE}}^{A} \Rightarrow 1\right] &= 1 \\ \Pr\left[\mathrm{Left}_{\mathcal{SE}}^{A} \Rightarrow 1\right] &= 0 \,.\end{aligned}$$

Hence $\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(A) = 1 - 0 = 1$. And A achieved this advantage by making just one lr-encryption oracle query, whose length, which as per our conventions is just the length of M_0, is n bits, and just one decryption oracle query, whose length is $2n$ bits. So $\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(t, 1, n, 1, 2n) = 1$.

Why are the two equations claimed above true? You have to return to the definitions of the quantities in question, as well as the description of the scheme itself, and walk it through. In world 1, meaning $b = 1$, the lr-encryption oracle returns $\langle \mathrm{IV}, C[1] \rangle$ with

$$C[1] \;=\; E_K(\mathrm{IV} \oplus M_1) \;=\; E_K(\mathrm{IV} \oplus 1^n) \,.$$

Now notice that

$$\begin{aligned} M &= \mathcal{D}_K(\langle \mathrm{IV}', C[1] \rangle) \\ &= E_K^{-1}(C[1]) \oplus \mathrm{IV}' \\ &= E_K^{-1}(E_K(\mathrm{IV} \oplus 1^n)) \oplus \mathrm{IV}' \\ &= (\mathrm{IV} \oplus 1^n) \oplus \mathrm{IV}'[0] \\ &= (\mathrm{IV} \oplus 1^n) \oplus (\mathrm{IV} \oplus 1^n) \\ &= 0^n \\ &= M_0 \,. \end{aligned}$$

Thus, the decryption oracle will return M_0, and A will return 1. In world 0, meaning $b = 0$, the lr-encryption oracle returns $\langle \mathrm{IV}, C[1] \rangle$ with

$$C[1] \;=\; E_K(\mathrm{IV} \oplus M_0) \;=\; E_K(\mathrm{IV} \oplus 0^l) \,.$$

Now notice that

$$\begin{aligned} M &= \mathcal{D}_K(\langle \mathrm{IV}', C[1] \rangle) \\ &= E_K^{-1}(C[1]) \oplus \mathrm{IV}' \\ &= E_K^{-1}(E_K(\mathrm{IV} \oplus 0^n)) \oplus \mathrm{IV}' \\ &= (\mathrm{IV} \oplus 0^n) \oplus \mathrm{IV}'[0] \end{aligned}$$

$$
\begin{aligned}
&= (\mathrm{IV} \oplus 0^n) \oplus (\mathrm{IV} \oplus 1^n) \\
&= 1^n \\
&= M_1 \, .
\end{aligned}
$$

Thus, the decryption oracle will return M_1, and A will return 0, meaning will return 1 with probability zero. ▮

5.11 Historical notes

The pioneering work on the theory of encryption is that of Goldwasser and Micali [18], with refinements by [28, 13]. This body of work is however in the asymmetric (i.e., public key) setting, and uses the asymptotic framework of polynomial-time adversaries and negligible success probabilities. The treatment of symmetric encryption we are using is from [3]. In particular Definition 5.1.1 and the concrete security framework are from [3]. The analysis of the CTR and CBC mode encryption schemes, as given in Theorems 5.7.1, 5.7.2 and 5.8.1 is also from [3]. The analysis of the CBC mode here is new.

5.12 Problems

Problem 29 Formalize a notion of security against key-recovery for symmetric encryption schemes. Then prove that IND-CPA security implies key-recovery security.

Problem 30 Consider the following notion of indistinguishability of an encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$:

$$\mathbf{Adv}^{\text{ind0-cpa}}_{\mathcal{SE}}(A) \;=\; \Pr[K \xleftarrow{\$} \mathcal{K} : A^{\mathcal{E}_K(\cdot)} \Rightarrow 1] - \Pr[K \xleftarrow{\$} \mathcal{K} : A^{\mathcal{E}_K(0^{|\cdot|})} \Rightarrow 1] \, .$$

That is, a scheme in IND0-CPA secure if the encryption of every string looks like the encryption of an equal number of zeros. Here we assume that whenever M is in the message space, so is $0^{|M|}$. Prove that this notion of security is equivalent to IND-CPA security, carefully stating a pair of theorems and proving them.

Problem 31 The definition above for IND0-CPA provides the adversary with no method to get, with certitude, the encryption of a given message: when the adversary asks a query M, it might get answered with $C \xleftarrow{\$} \mathcal{E}_K(M)$ or it might get answered with $C \xleftarrow{\$} \mathcal{E}_K(0^{|M|})$. Consider providing the adversary an additional, "reference" oracle that *always* encrypts the queried string. Consider defining the corresponding advantage notion in the natural way: for an encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, let

$$\mathbf{Adv}^{\text{ind0-cpa*}}_{\mathcal{SE}}(A) \;=\; \Pr[K \xleftarrow{\$} \mathcal{K} : A^{\mathcal{E}_K(\cdot),\, \mathcal{E}_K(\cdot)} \Rightarrow 1] - \Pr[K \xleftarrow{\$} \mathcal{K} : A^{0^{|M|},\, \mathcal{E}_K(\cdot)} \Rightarrow 1]$$

State and prove a theorem that shows that this notion of security is equivalent to our original IND0-CPA notion (and therefore to IND-CPA).

Problem 32 Let $l \geq 1$ and $m \geq 2$ be integers, and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a given symmetric encryption scheme whose associated plaintext space is $\{0,1\}^n$, meaning one can only encrypt messages of length n. In order to be able to encrypt longer messages, say ones of mn bits for some

```
algorithm $\mathcal{E}_K^{(m)}(M)$
    $M[1] \ldots M[m] \leftarrow M$
    for $i \leftarrow 1$ to $m$ do
        $C[i] \leftarrow \mathcal{E}_K(M[i])$
    return $C \leftarrow \langle C[1], \ldots, C[m] \rangle$
```

```
algorithm $\mathcal{D}_K^{(m)}(C)$
    $C[1] \cdots C[m] \leftarrow C$
    for $i \leftarrow 1$ to $m$ do
        $M[i] \leftarrow \mathcal{D}_K(C[i])$
        if $M[i] = \perp$ then return $\perp$
    return $M \leftarrow \langle M[1], \ldots, M[m] \rangle$
```

Figure 5.11: Encryption scheme for Problem 32.

$m \geq 1$, we define a new symmetric encryption scheme $\mathcal{SE}^{(m)} = (\mathcal{K}, \mathcal{E}^{(m)}, \mathcal{D}^{(m)})$ having the same key-generation algorithm as that of $\mathcal{SE}$, plaintext space $\{0,1\}^{mn}$, and encryption and decryption algorithms as depicted in Fig. 5.11.

(a) Show that

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}^{(m)}}(t, 1, mn, 1, mn) \quad = \quad 1$$

for some small t.

(b) Show that

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}^{(m)}}(t, q, mnq) \quad \leq \quad \mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(t, mq, mnq)$$

for any t, q.

Part **(a)** says that $\mathcal{SE}^{(m)}$ is insecure against chosen-ciphertext attack. Note this is true regardless of the security properties of $\mathcal{SE}$, which may itself be secure against chosen-ciphertext attack. Part **(b)** says that if $\mathcal{SE}$ is secure against chosen-plaintext attack, then so is $\mathcal{SE}^{(m)}$.

Problem 33 The CBC-Chain mode of operation is a CBC variant in which the IV that is used for the very first message to be encrypted is random, while the IV used for each subsequent encrypted message is the last block of ciphertext that was generated. The scheme is probabilistic and stateful. Show that CBC-Chain is insecure by giving a simple and efficient adversary that breaks it in the IND-CPA sense.

Problem 34 Using the proof of Theorem 5.7.1 as a template, prove Theorem 5.7.2 assuming Lemma **??**.

Problem 35 Define a notion for indistinguishability from random bits, IND\$-CPA. Your notion should capture the idea that the encryption of each message M looks like a string of random bits. Pay careful attention to the *number* of random bits that one outputs. Then formalize and prove that IND\$-CPA security implies IND-CPA security—but that IND-CPA security does not imply IND\$-CPA security.

Problem 36 Using a game-based argument, prove that CBC\$[$\mathsf{Func}(n,n)$] achieves IND\$-CPA security. Assume that one encodes $\langle R, C\rangle$ as $R \,\|\, C$.

Problem 37 Devise a secure extension to CBC\$ mode that allows messages of any bit length to be encrypted. Clearly state your encryption and decryption algorithm. Your algorithm should resemble CBC mode as much as possible, and should coincide with CBC mode when the message being encrypted is a multiple of the blocklength. It should increase the length of the message being encrypted by exactly n bits, where n is the length of the underlying blockcipher. How would you prove your algorithm secure?

Problem 38 An IND-CPA secure encryption scheme might not conceal identities, in the following sense: given a pair of ciphertexts C, C' for equal-length messages, it might be "obvious" if the ciphertexts were encrypted using the same random key or were encrypted using two different random keys. Give an example of a (plausibly) IND-CPA secure encryption scheme that has this is identity-revealing. Then give a definition for "identity-concealing" encryption. Your definition should imply IND-CPA security but a scheme meeting your definition can't be identity-revealing.

Bibliography

[1] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. *Proceedings of the 38th Symposium on Foundations of Computer Science*, IEEE, 1997.

[2] O. Goldreich. A uniform complexity treatment of encryption and zero-knowledge. *Journal of Cryptology*, Vol. 6, 1993, pp. 21-53.

[3] S. Goldwasser and S. Micali. Probabilistic encryption. *J. of Computer and System Sciences*, Vol. 28, April 1984, pp. 270–299.

[4] S. Micali, C. Rackoff and R. Sloan. The notion of security for probabilistic cryptosystems. *SIAM J. of Computing*, April 1988.

[5] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. *Proceedings of the 22nd Annual Symposium on the Theory of Computing*, ACM, 1990.

[6] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. *Advances in Cryptology – CRYPTO '91*, Lecture Notes in Computer Science Vol. 576, J. Feigenbaum ed., Springer-Verlag, 1991.

Chapter 6

Hash Functions

A hash function usually means a function that compresses, meaning the output is shorter than the input. Often, such a function takes an input of arbitrary or almost arbitrary length to one whose length is a fixed number, like 160 bits. Hash functions are used in many parts of cryptography, and there are many different types of hash functions, with differing security properties. We will consider them in this chapter.

6.1 The hash function SHA1

The hash function known as SHA1 is a simple but strange function from strings of almost arbitrary length to strings of 160 bits. The function was finalized in 1995, when a FIPS (Federal Information Processing Standard) came out from the US National Institute of Standards that specified SHA1.

Let $\{0,1\}^{<\ell}$ denote the set of all strings of length strictly less than ℓ. The function SHA1: $\{0,1\}^{<2^{64}} \rightarrow \{0,1\}^{160}$ is shown in Fig. 6.1. (Since 2^{64} is a very large length, we think of SHA1 as taking inputs of almost arbitrary length.) It begins by padding the message via the function shapad, and then iterates the *compression function* sha1 to get its output. The operations used in the algorithms of Fig. 6.1 are described in Fig. 6.2. (The first input in the call to SHF1 in code for SHA1 is a 128 bit string written as a sequence of four 32-bit words, each word being consisting of 8 hexadecimal characters. The same convention holds for the initialization of the variable V in the code of SHF1.)

SHA1 is derived from a function called MD4 that was proposed by Ron Rivest in 1990, and the key ideas behind SHA1 are already in MD4. Besides SHA1, another well-known "child" of MD4 is MD5, which was likewise proposed by Rivest. The MD4, MD5, and SHA11 algorithms are all quite similar in structure. The first two produce a 128-bit output, and work by "chaining" a compression function that goes from $512+128$ bits to 128 bits, while SHA1 produces a 160 bit output and works by chaining a compression function from $512+160$ bits to 160 bits.

So what is SHA1 supposed to do? First and foremost, it is supposed to be the case that nobody can find distinct strings M and M' such that $\mathsf{SHA1}(M) = \mathsf{SHA1}(M')$. This property is called *collision resistance*.

Stop for a moment and think about the collision-resistance requirement, for it is really quite amazing to think that such a thing could be possible. The function SHA1 maps strings of (almost) any length to strings of 160 bits. So even if you restricted the domain of SHA1 just to "short" strings—let us say strings of length 256 bits—then there must be an *enormous* number of pairs of

```
algorithm $\mathsf{SHA1}(M)$    // $|M| < 2^{64}$
    $V \leftarrow \mathsf{SHF1}($ 5A827999 $\|$ 6ED9EBA1 $\|$ 8F1BBCDC $\|$ CA62C1D6 , $M$ $)$
return $V$
```

```
algorithm $\mathsf{SHF1}(K, M)$    // $|K| = 128$ and $|M| < 2^{64}$
    $y \leftarrow \mathsf{shapad}(M)$
    Parse $y$ as $M_1 \| M_2 \| \cdots \| M_n$ where $|M_i| = 512$ $(1 \le i \le n)$
    $V \leftarrow$ 67452301 $\|$ EFCDAB89 $\|$ 98BADCFE $\|$ 10325476 $\|$ C3D2E1F0
    for $i = 1, \ldots, n$ do
        $V \leftarrow \mathsf{shf1}(K, M_i \| V)$
return $V$
```

```
algorithm $\mathsf{shapad}(M)$    // $|M| < 2^{64}$
    $d \leftarrow (447 - |M|) \bmod 512$
    Let $\ell$ be the 64-bit binary representation of $|M|$
    $y \leftarrow M \| 1 \| 0^d \| \ell$    // $|y|$ is a multiple of 512
return $y$
```

```
algorithm $\mathsf{shf1}(K, B \| V)$    // $|K| = 128$, $|B| = 512$ and $|V| = 160$
    Parse $B$ as $W_0 \| W_1 \| \cdots \| W_{15}$ where $|W_i| = 32$ $(0 \le i \le 15)$
    Parse $V$ as $V_0 \| V_1 \| \cdots \| V_4$ where $|V_i| = 32$ $(0 \le i \le 4)$
    Parse $K$ as $K_0 \| K_1 \| K_2 \| K_3$ where $|K_i| = 32$ $(0 \le i \le 3)$
    for $t = 16$ to 79 do
        $W_t \leftarrow \mathsf{ROTL}^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$
    $A \leftarrow V_0$ ; $B \leftarrow V_1$ ; $C \leftarrow V_2$ ; $D \leftarrow V_3$ ; $E \leftarrow V_4$
    for $t = 0$ to 19 do
        $L_t \leftarrow K_0$ ; $L_{t+20} \leftarrow K_1$ ; $L_{t+40} \leftarrow K_2$ ; $L_{t+60} \leftarrow K_3$
    for $t = 0$ to 79 do
        if $(0 \le t \le 19)$ then $f \leftarrow (B \wedge C) \vee ((\neg B) \wedge D)$
        if $(20 \le t \le 39$ OR $60 \le t \le 79)$ then $f \leftarrow B \oplus C \oplus D$
        if $(40 \le t \le 59)$ then $f \leftarrow (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
        $temp \leftarrow \mathsf{ROTL}^5(A) + f + E + W_t + L_t$
        $E \leftarrow D$ ; $D \leftarrow C$ ; $C \leftarrow \mathsf{ROTL}^{30}(B)$ ; $B \leftarrow A$ ; $A \leftarrow temp$
    $V_0 \leftarrow V_0 + A$ ; $V_1 \leftarrow V_1 + B$ ; $V_2 \leftarrow V_2 + C$ ; $V_3 \leftarrow V_3 + D$ ; $V_4 \leftarrow V_4 + E$
    $V \leftarrow V_0 \| V_1 \| V_2 \| V_3 \| V_4$
return $V$
```

Figure 6.1: The SHA1 hash function and the underlying SHF1 family.

strings M and M' that hash to the same value. This is just by the pigeonhole principle: if 2^{256} pigeons (the 256-bit messages) roost in 2^{160} holes (the 160-bit hash values) then some two pigeons (two distinct strings) roost in the same hole (have the same hash). Indeed countless pigeons must share the same hole. The difficult is only that nobody has as yet *identified* (meaning, explicitly provided) even two such pigeons (strings).

$X \wedge Y$	bitwise AND of X and Y
$X \vee Y$	bitwise OR of X and Y
$X \oplus Y$	bitwise XOR of X and Y
$\neg X$	bitwise complement of X
$X + Y$	integer sum modulo 2^{32} of X and Y
$\mathsf{ROTL}^l(X)$	circular left shift of bits of X by l positions $(0 \le l \le 31)$

Figure 6.2: Operations on 32-bit words used in sha1.

In trying to define this collision-resistance property of SHA1 we immediately run into "foundational" problems. We would like to say that it is computationally infeasible to output a pair of distinct strings M and M' that collide under SHA1. But in what sense could it be infeasible? There *is* a program—indeed a very short an simple one, having just two "print" statements—whose output specifies a collision. It's not computationally hard to output a collision; it can't be. The only difficulty is our *human* problem of not knowing what this program is.

It seems very hard to make a mathematical definition that captures the idea that human beings can't find collisions in SHA1. In order to reach a mathematically precise definition we are going to have to change the very nature of what we conceive to be a hash function. Namely, rather than it being a single function, it will be a family of functions. This is unfortunate in some ways, because it distances us from concrete hash functions like SHA1. But no alternative is known.

6.2 Collision-resistant hash functions

A hash function for us is a family of functions $H\colon \mathcal{K} \times D \to R$. Here D is the domain of H and R is the range of H. As usual, if $K \in \mathcal{K}$ is a particular key then $H_K\colon D \to R$ is defined for all $M \in D$ by $H_K(M) = H(K, M)$. This is the instance of H defined by key K.

An example is SHF1: $\{0,1\}^{128} \times \{0,1\}^{<2^{64}} \to \{0,1\}^{160}$, as described in Fig. 6.1. This hash function takes a 128-bit key and an input M of at most 2^{64} bits and returns a 160-bit output. The function SHA1 is an instance of this family, namely the one whose associated key is

$$\texttt{5A827999} \,\|\, \texttt{6ED9EBA1} \,\|\, \texttt{8F1BBCDC} \,\|\, \texttt{CA62C1D6}\ .$$

Let $H\colon \mathcal{K} \times D \to R$ be a hash function. Here is some notation we use in this chapter. For any key K and $y \in R$ we let

$$H_K^{-1}(y) \;=\; \{\, x \in D \;:\; H_K(x) = y \,\}$$

denote the pre-image set of y under H_K. Let

$$\mathsf{Image}(H_K) \;=\; \{\, H_K(x) \;:\; x \in D \,\}$$

denote the image of H_K.

A *collision* for a function $h\colon D \to R$ is a pair $x_1, x_2 \in D$ of points such that (1) $H_K(x_1) = H_K(x_2)$ and (2) $x_1 \neq x_2$. The most basic security property of a hash function is collision-resistance, which measures the ability of an adversary to find a collision for an instance of a family H. There are different notions of collision-resistance, varying in restrictions put on the adversary in its quest for a collision.

To introduce the different notions, we imagine a game, parameterized by an integer $s \in \{0, 1, 2\}$,

Pre-key attack phase	A selects $2-s$ points
Key selection phase	A key K is selected at random from $\mathcal{K}$
Post-key attack phase	A is given K and returns s points
Winning condition	The 2 points selected by A form a collision for H_K

Figure 6.3: Framework for security notions for collision-resistant hash functions. The three choices of $s \in \{0,1,2\}$ give rise to three notions of security.

---

and involving an adversary A. It consists of a *pre-key attack* phase, followed by a *key-selection phase*, followed by a *post-key attack phase*. The adversary is attempting to find a collision for H_K, where key K is selected at random from $\mathcal{K}$ in the key-selection phase. Recall that a collision consists of a pair x_1, x_2 of (distinct) points in D. The adversary is required to specify $2-s$ points in the pre-key attack phase, before it has any information about the key. (The latter has yet to be selected.) Once the adversary has specified these points and the key has been selected, the adversary is given the key, and will choose the remaining s points as a function of the key, in the post-key attack phase. It wins if the $2 = (2-s)+s$ points it has selected form a collision for H_K.

Fig. 6.3 summarizes the framework. The three choices of the parameter s give rise to three notions of security. The higher the value of s the more power the adversary has, and hence the more stringent is the corresponding notion of security. Fig. 6.4 provides in more detail the experiments underlying the three attacks arising from the above framework. We represent by *st* information that the adversary wishes to maintain across its attack phases. It will output this information in the pre-key attack phase, and be provided it at the start of the post-key attack phase.

In a variant of this model that we consider in Section **??**, the adversary is not given the key K in the post-key attack phase, but instead is given an oracle for $H_K(\cdot)$. To disambiguate, we refer to our current notions as capturing collision-resistance under *known-key* attack, and the notions of Section **??** as capturing collision-resistance under *hidden-key* attack. The notation in the experiments of Fig. 6.4 and Definition 6.2.1 reflects this via the use of "kk", except that for CR0, known and hidden key attacks coincide, and hence we just say cr0.

The three types of hash functions we are considering are known by other names in the literature, as indicated in Fig. 6.5.

Definition 6.2.1 Let $H\colon \mathcal{K} \times D \to R$ be a hash function and let A be an algorithm. We let

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{cr2}}_{H}(A) &= \Pr\left[\mathrm{CR2}^{A}_{H} \Rightarrow \mathsf{true}\right] \\
\mathbf{Adv}^{\mathrm{cr1}}_{H}(A) &= \Pr\left[\mathrm{CR1}^{A}_{H} \Rightarrow \mathsf{true}\right] \\
\mathbf{Adv}^{\mathrm{cr0}}_{H}(A) &= \Pr\left[\mathrm{CR0}^{A}_{H} \Rightarrow \mathsf{true}\right] . \blacksquare
\end{aligned}$$

In measuring resource usage of an adversary we use our usual conventions. Although there is formally no definition of a "secure" hash function, we will talk of a hash function being CR2, CR1 or CR0 with the intended meaning that its associated advantage function is small for all adversaries of practical running time.

Note that the running time of the adversary is not really relevant for CR0, because we can always imagine that hardwired into its code is a "best" choice of distinct points x_1, x_2, meaning a

```
Game CR2_H
procedure Initialize
    $K \xleftarrow{\$} \{0,1\}^k$
    Return $K$

procedure Finalize($x_1, x_2$)
    Return $(x_1 \neq x_2 \wedge H_K(x_1) = H_K(x_2))$
```

```
Game CR1_H
procedure Initialize

procedure Submit($x_1$)
    $K \xleftarrow{\$} \{0,1\}^k$
    Return $K$

procedure Finalize($x_2$)
    Return $(x_1 \neq x_2 \wedge H_K(x_1) = H_K(x_2))$
```

```
Game CR0_H
procedure Initialize

procedure Submit($x_1, x_2$)
    $K \xleftarrow{\$} \{0,1\}^k$
    Return $K$

procedure Finalize()
    Return $(x_1 \neq x_2 \wedge H_K(x_1) = H_K(x_2))$
```

Figure 6.4: Games defining security notions for three kinds of collision-resistant hash functions under known-key attack.

Type	**Name(s) in literature**
CR2-KK	collision-free, collision-resistant, collision-intractable
CR1-KK	universal one-way [29] (aka. target-collision resistant [1])
CR0	universal, almost universal

Figure 6.5: Types of hash functions, with names in our framework and corresponding names found in the literature.

choice for which

$$\begin{aligned} &\Pr\left[\, K \xleftarrow{\$} \mathcal{K} \;:\; H_K(x_1) = H_K(x_2) \,\right] \\ &= \max_{y_1 \neq y_2} \Pr\left[\, K \xleftarrow{\$} \mathcal{K} \;:\; H_K(y_1) = H_K(y_2) \,\right] . \end{aligned}$$

The above value equals $\mathbf{Adv}_H^{\mathrm{cr0}}(A)$ and is the maximum advantage attainable.

Clearly, a CR2 hash function is also CR1 and a CR1 hash function is also CR0. The following states the corresponding relations formally. The proof is trivial and is omitted.

Proposition 6.2.2 Let $H\colon \mathcal{K} \times D \to R$ be a hash function. Then for any adversary A_0 there exists an adversary A_1 having the same running time as A_0 and

$$\mathbf{Adv}_H^{\mathrm{cr0}}(A_0) \;\leq\; \mathbf{Adv}_H^{\mathrm{cr1\text{-}kk}}(A_1)\ .$$

Also for any adversary A_1 there exists an adversary A_2 having the same running time as A_1 and

$$\mathbf{Adv}_H^{\mathrm{cr1\text{-}kk}}(A_1) \;\leq\; \mathbf{Adv}_H^{\mathrm{cr2\text{-}kk}}(A_2)\ . \blacksquare$$

We believe that SHF1 is CR2, meaning that there is no practical algorithm A for which $\mathbf{Adv}_H^{\mathrm{cr2\text{-}kk}}(A)$ is appreciably large. This is, however, purely a belief, based on the current inability to find such an algorithm. Perhaps, later, such an algorithm will emerge.

It is useful, for any integer n, to get $\mathsf{SHF1}^n\colon \{0,1\}^n \to \{0,1\}^{160}$ denote the restriction of SHF1 to the domain $\{0,1\}^n$. Note that a collision for $\mathsf{SHF1}^n_K$ is also a collision for $\mathsf{SHF1}_K$, and it is often convenient to think of attacking $\mathsf{SHF1}^n$ for some fixed n rather than SHF1 itself.

6.3 Collision-finding attacks

Let us focus on CR2, which is the most important property for the applications we will see later. We consider different types of CR2-type collision-finding attacks on a family $H\colon \mathcal{K} \times D \to R$ where D, R are finite sets. We assume the family performs some reasonable compression, say $|D| \geq 2|R|$. Canonical example families to keep in mind are $H = \mathsf{SHF1}^n$ for $n \geq 161$ and shf1, the compression function of SHF1.

Collision-resistance does not mean it is impossible to find a collision. Analogous to the case of one-wayness, there is an obvious collision-finding strategy. Let us enumerate the elements of D in some way, so that $D = \{D_1, D_2, \ldots, D_d\}$ where $d = |D|$. The following adversary A implements an exhaustive search collision-finding attack:

```
Adversary $A(K)$
    $x_1 \xleftarrow{\$} D$ ; $y \leftarrow H_K(x_1)$
    for $i = 1, \ldots, q$ do
        if ($H_K(D_i) = y$ and $x_1 \neq D_i$) then return $x_1, D_i$
    return FAIL
```

We call q the number of *trials*. Each trial involves one computation of H_K, so the number of trials is a measure of the time taken by the attack. To succeed, the attack requires that $H_K^{-1}(y)$ has size at least two, which happens at least half the time if $|D| \geq 2|R|$. However, we would still expect that it would take about $q = |D|$ trials to find a collision, which is prohibitive, for D is usually large. For example, for $F = $ shf1, the domain has size 2^{672}, far too large. For $\mathsf{SHF1}^n$, we would choose n as small as possible, but need $n \geq 161$ to ensure collisions exist, so the attack uses 2^{161} computations of H_K, which is not practical.

Now here's another idea. We pick points at random, hoping that their image under H_K equals the image under H_K of an initial target point. Call this the random-input collision-finding attack. It is implemented like this:

```
Adversary $A(K)$
    $x_1 \xleftarrow{\$} D$ ; $y \leftarrow H_K(x_1)$
    for $i = 1, \ldots, q$ do
        $x_2 \xleftarrow{\$} D$
        if ($H_K(x_2) = y$ and $x_1 \neq x_2$) then return $x_1, x_2$
    return FAIL
```

```
for $i = 1, \ldots, q$ do    // $q$ is the number of trials
    $x_i \xleftarrow{\$} D$ ; $y_i \leftarrow H_K(x_i)$
    if (there exists $j < i$ such that $y_i = y_j$ but $x_i \neq x_j$)    // collision found
        then return $x_i, x_j$
    return FAIL    // No collision found
```

Figure 6.6: Birthday attack on a hash function $H\colon \mathcal{K} \times D \to R$. The attack is successful in finding a collision if it does not return FAIL.

A particular trial finds a collision with probability (about) 1 in $|R|$, so we expect to find a collision in about $q = |R|$ trials. This is much better than the $|D|$ trials used by our first attempt. In particular, a collision for $\mathsf{shf1}$ would be found in time around 2^{160} rather than 2^{672}. But this is still far from practical. Our conclusion is that as long as the range size of the hash function is large enough, this attack is not a threat.

We now consider another strategy, called a *birthday attack*, that turns out to be much better than the above. It is illustrated in Fig. 6.6. It picks at random q points from the domain, and applies H_K to each of them. If it finds two distinct points yielding the same output, it has found a collision for H_K. The question is how large q need be to find a collision. The answer may seem surprising at first. Namely, $q = O(\sqrt{|R|})$ trials suffices.

We will justify this later, but first let us note the impact. Consider $\mathsf{SHA1}^n$ with $n \geq 161$. As we indicated, the random-input collision-finding attack takes about 2^{160} trials to find a collision. The birthday attack on the other hand takes around $\sqrt{2^{160}} = 2^{80}$ trials. This is MUCH less than 2^{160}. Similarly, the birthday attack finds a collision in $\mathsf{shf1}$ in around 2^{80} trials while while random-input collision-finding takes about 2^{160} trials.

To see why the birthday attack performs as well as we claimed, we recall the following game. Suppose we have q balls. View them as numbered, $1, \ldots, q$. We also have N bins, where $N \geq q$. We throw the balls at random into the bins, one by one, beginning with ball 1. At random means that each ball is equally likely to land in any of the N bins, and the probabilities for all the balls are independent. A collision is said to occur if some bin ends up containing at least two balls. We are interested in $C(N, q)$, the probability of a collision. As shown in the Appendix,

$$C(N, q) \;\approx\; \frac{q^2}{2N} \tag{6.1}$$

for $1 \leq q \leq \sqrt{2N}$. Thus $C(N, q) \approx 1$ for $q \approx \sqrt{2N}$.

The relation to birthdays arises from the question of how many people need be in a room before the probability of there being two people with the same birthday is close to one. We imagine each person has a birthday that is a random one of the 365 days in a year. This means we can think of a person as a ball being thrown at random into one of 365 bins, where the i-th bin represents having birthday the i-th day of the year. So we can apply the Proposition from the Appendix with $N = 365$ and q the number of people in the room. The Proposition says that when the room contains $q \approx \sqrt{2 \cdot 365} \approx 27$ people, the probability that there are two people with the same birthday is close to one. This number (27) is quite small and may be hard to believe at first hearing, which is why this is sometimes called the birthday paradox.

To see how this applies to the birthday attack of Fig. 6.6, let us enumerate the points in the range as $R_1, \ldots, R_N$, where $N = |R|$. Each such point defines a bin. We view x_i as a ball, and imagine that it is thrown into bin y_i, where $y_i = H_K(x_i)$. Thus, a collision of balls (two balls in the same bin) occurs precisely when two values x_i, x_j have the same output under H_K. We are interested in the probability that this happens as a function of q. (We ignore the probability that $x_i = x_j$, counting a collision only when $H_K(x_i) = H_K(x_j)$. It can be argued that since D is larger than R, the probability that $x_i = x_j$ is small enough to neglect.)

However, we cannot apply the birthday analysis directly, because the latter assumes that each ball is equally likely to land in each bin. This is not, in general, true for our attack. Let $P(R_j)$ denote the probability that a ball lands in bin R_j, namely the probability that $H_K(x) = R_j$ taken over a random choice of x from D. Then

$$P(y) \;=\; \frac{|H_K^{-1}(R_j)|}{|D|} \;.$$

In order for $P(R_1) = P(R_2) = \cdots = P(R_N)$ to be true, as required to apply the birthday analysis, it must be the case that

$$|H_K^{-1}(R_1)| = |H_K^{-1}(R_2)| = \cdots = |H_K^{-1}(R_N)| \;.$$

A function H_K with this property is called *regular*, and H is called regular if H_K is regular for every K. Our conclusion is that if H is regular, then the probability that the attack succeeds is roughly $C(N, q)$. So the above says that in this case we need about $q \approx \sqrt{2N} = \sqrt{2 \cdot |R|}$ trials to find a collision with probability close to one.

If H is not regular, it turns out the attack succeeds even faster, telling us that we ought to design hash functions to be as "close" to regular as possible [2].

In summary, there is a $2^{l/2}$ or better time attack to find collisions in any hash function outputting l bits. This leads designers to choose l large enough that $2^{l/2}$ is prohibitive. In the case of SHF1 and shf1, the choice is $l = 160$ because 2^{80} is indeed a prohibitive number of trials. These functions cannot thus be considered vulnerable to birthday attack. (Unless they turn out to be extremely non-regular, for which there is no evidence so far.)

Ensuring, by appropriate choice of output length, that a function is not vulnerable to a birthday attack does not, of course, guarantee it is collision resistant. Consider the family H: $\mathcal{K} \times \{0,1\}^{161} \rightarrow \{0,1\}^{160}$ defined as follows. For any K and any x, function $H_K(x)$ returns the first 160 bits of x. The output length is 160, so a birthday attack takes 2^{80} time and is not feasible, but it is still easy to find collisions. Namely, on input K, an adversary can just pick some 160-bit y and output $y0, y1$. This tells us that to ensure collision-resistance it is not only important to have a long enough output but also design the hash function so that there no clever "shortcuts" to finding a collision, meaning no attacks that exploit some weakness in the structure of the function to quickly find collisions.

We believe that shf1 is well-designed in this regard. Nobody has yet found an adversary that finds a collision in shf1 using less than 2^{80} trials. Even if a somewhat better adversary, say one finding a collision for shf1 in 2^{65} trials, were found, it would not be devastating, since this is still a very large number of trials, and we would still consider shf1 to be collision-resistant.

If we believe shf1 is collision-resistant, Theorem 6.5.2 tells us that SHF1, as well as $\mathsf{SHF1}_n$, can also be considered collision-resistant, for all n.

6.4 One-wayness of collision-resistant hash functions

Intuitively, a family H is one-way if it is computationally infeasible, given H_K and a range point $y = H_K(x)$, where x was chosen at random from the domain, to find a pre-image of y (whether x or some other) under H_K. Since this definition too has a hidden-key version, we indicate the known-key in the notation below.

Definition 6.4.1 Let H: $\mathcal{K} \times D \to R$ be a family of functions and let A be an algorithm. We consider the following experiment:

```
Game OW_H

procedure Initialize
$K \xleftarrow{\$} \{0,1\}^k$
$x \xleftarrow{\$} D$ ; $y \leftarrow H_K(x)$
Return $K, y$

procedure Finalize($x'$)
Return $(H_K(x') = y)$
```

We let

$$\mathbf{Adv}_H^{\mathrm{ow}}(A) \;=\; \Pr[\mathrm{OW}_H^A \Rightarrow \mathsf{true}] \;. \blacksquare$$

We now ask ourselves whether collision-resistance implies one-wayness. It is easy to see, however, that, in the absence of additional assumptions about the hash function than collision-resistance, the answer is "no." For example, let H be a family of functions every instance of which is the identity function. Then H is highly collision-resistant (the advantage of an adversary in finding a collision is zero regardless of its time-complexity since collisions simply don't exist) but is not one-way.

However, we would expect that "genuine" hash functions, meaning ones that perform some non-trivial compression of their data (ie. the size of the range is more than the size of the domain) are one-way. This turns out to be true, but needs to be carefully quantified. To understand the issues, it may help to begin by considering the natural argument one would attempt to use to show that collision-resistance implies one-wayness.

Suppose we have an adversary A that has a significant advantage in attacking the one-wayness of hash function H. We could try to use A to find a collision via the following strategy. In the pre-key phase (we consider a type-1 attack) we pick and return a random point x_1 from D. In the post-key phase, having received the key K, we compute $y = H_K(x_1)$ and give K, y to A. The latter returns some x_2, and, if it was successful, we know that $H_K(x_2) = y$. So $H_K(x_2) = H_K(x_1)$ and we have a collision.

Not quite. The catch is that we only have a collision if $x_2 \neq x_1$. The probability that this happens turns out to depend on the quantity:

$$\mathbf{PreIm}_H(1) \;=\; \Pr\left[\, K \xleftarrow{\$} \mathcal{K} \,;\, x \xleftarrow{\$} D \,;\, y \leftarrow H_K(x) \;:\; |H_K^{-1}(y)| = 1 \,\right] \;.$$

This is the probability that the size of the pre-image set of y is exactly 1, taken over y generated as shown. The following Proposition says that a collision-resistant function H is one-way as long as $\mathbf{PreIm}_H(1)$ is small.

Proposition 6.4.2 Let H: $\mathcal{K} \times D \to R$ be a hash function. Then for any A there exists a B such that

$$\mathbf{Adv}_H^{\text{ow-kk}}(A) \;\leq\; 2 \cdot \mathbf{Adv}_H^{\text{cr1-kk}}(B) + \mathbf{PreIm}_H(1) \;.$$

Furthermore the running time of B is that of A plus the time to sample a domain point and compute H once. $\blacksquare$

The result is about the CR1 type of collision-resistance. However Proposition 6.2.2 implies that the same is true for CR2.

A general and widely-applicable corollary of the above Proposition is that collision-resistance implies one-wayness as long as the domain of the hash function is significantly larger than its range. The following quantifies this.

Corollary 6.4.3 Let $H\colon \mathcal{K} \times D \to R$ be a hash function. Then for any A there exists a B such that

$$\mathbf{Adv}_H^{\text{ow-kk}}(A) \;\; \leq \;\; 2 \cdot \mathbf{Adv}_H^{\text{cr1-kk}}(B) + \frac{|R|}{|D|} \; .$$

Furthermore the running time of B is that of A plus the time to sample a domain point and compute H once. ▮

Proof of Corollary 6.4.3: For any key K, the number of points in the range of H_K that have exactly one pre-image certainly cannot exceed $|R|$. This implies that

$$\mathbf{PreIm}_H(1) \;\; \leq \;\; \frac{|R|}{|D|} \; .$$

The corollary follows from Proposition 6.4.2. ▮

Corollary 6.4.3 says that if H is collision-resistant, and performs enough compression that $|R|$ is much smaller than $|D|$, then it is also one-way. Why? Let A be a practical adversary that attacks the one-wayness of H. Then B is also practical, and since H is collision-resistant we know $\mathbf{Adv}_H^{\text{cr1-kk}}(B)$ is low. Equation (6.2) then tells us that as long as $|R|/|D|$ is small, $\mathbf{Adv}_H^{\text{ow-kk}}(A)$ is low, meaning H is one-way.

As an example, let H be the compression function shf1. In that case $R = \{0,1\}^{160}$ and $D = \{0,1\}^{672}$ so $|R|/|D| = 2^{-512}$, which is tiny. We believe shf1 is collision-resistant, and the above thus says it is also one-way.

There are some natural hash functions, however, for which Corollary 6.4.3 does not apply. Consider a hash function H every instance of which is two-to-one. The ratio of range size to domain size is $1/2$, so the right hand side of the equation of Corollary 6.4.3 is 1, meaning the bound is vacuous. However, such a function is a special case of the one considered in the following Proposition.

Corollary 6.4.4 Suppose $1 \leq r < d$ and let $H\colon \mathcal{K} \times \{0,1\}^d \to \{0,1\}^r$ be a hash function which is regular, meaning $|H_K^{-1}(y)| = 2^{d-r}$ for every $y \in \{0,1\}^r$ and every $K \in \mathcal{K}$. Then for any A there exists a B such that

$$\mathbf{Adv}_H^{\text{ow-kk}}(A) \;\; \leq \;\; 2 \cdot \mathbf{Adv}_H^{\text{cr1-kk}}(B) \; .$$

Furthermore the running time of B is that of A plus the time to sample a domain point and compute H once. ▮

Proof of Corollary 6.4.4: The assumption $d > r$ implies that $\mathbf{PreIm}_H(1) = 0$. Now apply Proposition 6.4.2. ▮

We now turn to the proof of Proposition 6.4.2.

Proof of Proposition 6.4.2: Here's how B works:

```
Pre-key phase                      | Post-key phase
                                   |
Adversary $B()$                    | Adversary $B(K, st)$
    $x_1 \xleftarrow{\$} D$ ; $st \leftarrow x_1$      |     Retrieve $x_1$ from $st$
    return $(x_1, st)$             |     $y \leftarrow H_K(x_1)$ ; $x_2 \xleftarrow{\$} B(K, y)$
                                   |     return $x_2$
```

Let $\Pr[\cdot]$ denote the probability of event "$\cdot$" in experiment $\mathbf{Exp}_H^{\text{cr1-kk}}(B)$. For any $K \in \mathcal{K}$ let

$$S_K \;=\; \{\, x \in D \;:\; |H_K^{-1}(H_K(x))| = 1 \,\} \,.$$

$$\begin{aligned}
&\mathbf{Adv}_H^{\text{cr1-kk}}(B) && (6.2)\\
&\quad= \Pr[H_K(x_2) = y \wedge x_1 \neq x_2] && (6.3)\\
&\quad\geq \Pr[H_K(x_2) = y \wedge x_1 \neq x_2 \wedge x_1 \notin S_K] && (6.4)\\
&\quad= \Pr[x_1 \neq x_2 \mid H_K(x_2) = y \wedge x_1 \notin S_K] \cdot \Pr[H_K(x_2) = y \wedge x_1 \notin S_K] && (6.5)\\
&\quad\geq \frac{1}{2} \cdot \Pr[H_K(x_2) = y \wedge x_1 \notin S_K] && (6.6)\\
&\quad\geq \frac{1}{2} \cdot (\Pr[H_K(x_2) = y] - \Pr[x_1 \in S_K]) && (6.7)\\
&\quad= \frac{1}{2} \cdot \Big(\mathbf{Adv}_H^{\text{ow-kk}}(A) - \mathbf{PreIm}_H(1)\Big) \,. && (6.8)
\end{aligned}$$

Re-arranging terms yields Equation (6.2). Let us now justify the steps above. Equation (6.3) is by definition of $\mathbf{Adv}_H^{\text{cr1-kk}}(B)$ and B. Equation (6.4) is true because $\Pr[E] \geq \Pr[E \wedge F]$ for any events E, F. Equation (6.5) uses the standard formula $\Pr[E \wedge F] = Pr[E|F] \cdot \Pr[F]$. Equation (6.6) is justified as follows. Adversary A has no information about x_1 other than that it is a random point in the set $H_K^{-1}(y)$. However if $x_1 \notin S_K$ then $|H_K^{-1}(y)| \geq 2$. So the probability that $x_2 \neq x_1$ is at least 1/2 in this case. Equation (6.7) applies another standard probabilistic inequality, namely that $\Pr[E \wedge \overline{F}] \geq \Pr[E] - \Pr[F]$. Equation (6.8) uses the definitions of the quantities involved. ▌

6.5 The MD transform

We saw above that SHF1 worked by iterating applications of its compression function shf1. The latter, under any key, compresses 672 bits to 160 bits. SHF1 works by compressing its input 512 bits at a time using shf1.

The iteration method has been chosen carefully. It turns out that if shf1 is collision-resistant, then SHF1 is guaranteed to be collision-resistant. In other words, the harder task of designing a collision-resistant hash function taking long and variable-length inputs has been reduced to the easier task of designing a collision-resistant compression function that only takes inputs of some fixed length.

This has clear benefits. We need no longer seek attacks on SHF1. To validate it, and be assured it is collision-resistant, we need only concentrate on validating shf1 and showing the latter is collision-resistant.

This is one case of an important hash-function design principle called the MD paradigm [10, 3]. This paradigm shows how to transform a compression function into a hash function in such a way that collision-resistance of the former implies collision-resistance of the latter. We are now going to take a closer look at this paradigm.

```
$H(K, M)$
    $y \leftarrow \mathsf{pad}(M)$
    Parse $y$ as $M_1 \| M_2 \| \cdots \| M_n$ where $|M_i| = b$ $(1 \le i \le n)$
    $V \leftarrow \mathtt{IV}$
    for $i = 1, \ldots, n$ do
        $V \leftarrow h(K, M_i \| V)$
Return $V$
```

```
Adversary $A_h(K)$
    Run $A_H(K)$ to get its output $(x_1, x_2)$
    $y_1 \leftarrow \mathsf{pad}(x_1)$ ; $y_2 \leftarrow \mathsf{pad}(x_2)$
    Parse $y_1$ as $M_{1,1} \| M_{1,2} \| \cdots \| M_{1,n[1]}$ where $|M_{1,i}| = b$ $(1 \le i \le n[1])$
    Parse $y_2$ as $M_{2,1} \| M_{2,2} \| \cdots \| M_{2,n[2]}$ where $|M_{2,i}| = b$ $(1 \le i \le n[2])$
    $V_{1,0} \leftarrow \mathtt{IV}$ ; $V_{2,0} \leftarrow \mathtt{IV}$
    for $i = 1, \ldots, n[1]$ do $V_{1,i} \leftarrow h(K, M_{1,i} \| V_{1,i-1})$
    for $i = 1, \ldots, n[2]$ do $V_{2,i} \leftarrow h(K, M_{2,i} \| V_{2,i-1})$
    if $(V_{1,n[1]} \ne V_{2,n[2]}$ OR $x_1 = x_2)$ return FAIL
    if $|x_1| \ne |x_2|$ then return $(M_{1,n[1]} \| V_{1,n[1]-1}, M_{2,n[2]} \| V_{2,n[2]-1})$
    $n \leftarrow n[1]$    // $n = n[1] = n[2]$ since $|x_1| = |x_2|$
    for $i = n$ downto 1 do
        if $M_{1,i} \| V_{1,i-1} \ne M_{2,i} \| V_{2,i-1}$ then return $(M_{1,i} \| V_{1,i-1}, M_{2,i} \| V_{2,i-1})$
```

Figure 6.7: Hash function H defined from compression function h via the MD paradigm, and adversary A_h for the proof of Theorem 6.5.2.

Let b be an integer parameter called the block length, and v another integer parameter called the chaining-variable length. Let h: $\mathcal{K} \times \{0,1\}^{b+v} \to \{0,1\}^v$ be a family of functions that we call the compression function. We assume it is collision-resistant.

Let B denote the set of all strings whose length is a positive multiple of b bits, and let D be some subset of $\{0,1\}^{<2^b}$.

Definition 6.5.1 A function pad: $D \to B$ is called a *MD-compliant padding function* if it has the following properties for all $M, M_1, M_2 \in D$:
(1) M is a prefix of $\mathsf{pad}(M)$
(2) If $|M_1| = |M_2|$ then $|\mathsf{pad}(M_1)| = |\mathsf{pad}(M_2)|$
(3) If $M_1 \ne M_2$ then the last block of $\mathsf{pad}(M_1)$ is different from the last block of $\mathsf{pad}(M_2)$. ▮

A block, above, consists of b bits. Remember that the output of pad is in B, meaning is a sequence of b-bit blocks. Condition (3) of the definition is saying that if two messages are different then, when we apply pad to them, we end up with strings that differ in their final blocks.

An example of a MD-compliant padding function is shapad. However, there are other examples as well.

Now let $\mathtt{IV}$ be a v-bit value called the initial vector. We build a family H: $\mathcal{K} \times D \to \{0,1\}^v$ from h and pad as illustrated in Fig. 6.7. Notice that $\mathsf{SHF1}$ is such a family, built from $h = \mathsf{shf1}$ and $\mathsf{pad} = \mathsf{shapad}$. The main fact about this method is the following.

Theorem 6.5.2 *Let h: $\mathcal{K}\times\{0,1\}^{b+v} \rightarrow \{0,1\}^v$ be a family of functions and let H: $\mathcal{K}\times D \rightarrow \{0,1\}^v$ be built from h as described above. Suppose we are given an adversary A_H that attempts to find collisions in H. Then we can construct an adversary A_h that attempts to find collisions in h, and*

$$\mathbf{Adv}_H^{\text{cr2-kk}}(A_H) \;\leq\; \mathbf{Adv}_h^{\text{cr2-kk}}(A_h)\;. \tag{6.9}$$

Furthermore, the running time of A_h is that of A_H plus the time to perform $(|\mathsf{pad}(x_1)|+|\mathsf{pad}(x_2)|)/b$ computations of h where (x_1, x_2) is the collision output by A_H. ▌

This theorem says that if h is collision-resistant then so is H. Why? Let A_H be a practical adversary attacking H. Then A_h is also practical, because its running time is that of A_H plus the time to do some extra computations of h. But since h is collision-resistant we know that $\mathbf{Adv}_h^{\text{cr2-kk}}(A_h)$ is low. Equation (6.9) then tells us that $\mathbf{Adv}_H^{\text{cr2-kk}}(A_H)$ is low, meaning H is collision-resistant as well.

Proof of Theorem 6.5.2: Adversary A_h, taking input a key $K \in \mathcal{K}$, is depicted in Fig. 6.7. It runs A_H on input K to get a pair (x_1, x_2) of messages in D. We claim that if x_1, x_2 is a collision for H_K then A_h will return a collision for h_K.

Adversary A_h computes $V_{1,n[1]} = H_K(x_1)$ and $V_{2,n[2]} = H_K(x_2)$. If x_1, x_2 is a collision for H_K then we know that $V_{1,n[1]} = V_{2,n[2]}$. Let us assume this. Now, let us look at the inputs to the application of h_K that yielded these outputs. If these inputs are different, they form a collision for h_K.

The inputs in question are $M_{1,n[1]} \,\|\, V_{1,n[1]-1}$ and $M_{2,n[2]} \,\|\, V_{2,n[2]-1}$. We now consider two cases. The first case is that x_1, x_2 have different lengths. Item (3) of Definition 6.5.1 tells us that $M_{1,n[1]} \neq M_{2,n[2]}$. This means that $M_{1,n[1]} \,\|\, V_{1,n[1]-1} \neq M_{2,n[2]} \,\|\, V_{2,n[2]-1}$, and thus these two points form a collision for h_K that can be output by A_h.

The second case is that x_1, x_2 have the same length. Item (2) of Definition 6.5.1 tells us that y_1, y_2 have the same length as well. We know this length is a positive multiple of b since the range of pad is the set B, so we let n be the number of b-bit blocks that comprise y_1 and y_2. Let V_n denote the value $V_{1,n}$, which by assumption equals $V_{2,n}$. We compare the inputs $M_{1,n} \,\|\, V_{1,n-1}$ and $M_{2,n} \,\|\, V_{2,n-1}$ that under h_K yielded V_n. If they are different, they form a collision for h_K and can be returned by A_h. If, however, they are the same, then we know that $V_{1,n-1} = V_{2,n-1}$. Denoting this value by V_{n-1}, we now consider the inputs $M_{1,n-1} \,\|\, V_{1,n-2}$ and $M_{2,n-1} \,\|\, V_{2,n-2}$ that under h_K yield V_{n-1}. The argument repeats itself: if these inputs are different we have a collision for h_K, else we can step back one more time.

Can we get stuck, continually stepping back and not finding our collision? No, because $y_1 \neq y_2$. Why is the latter true? We know that $x_1 \neq x_2$. But item (1) of Definition 6.5.1 says that x_1 is a prefix of y_1 and x_2 is a prefix of y_2. So $y_1 \neq y_2$.

We have argued that on any input K, adversary A_h finds a collision in h_K exactly when A_H finds a collision in H_K. This justifies Equation (6.9). We now justify the claim about the running time of A_h. The main component of the running time of A_h is the time to run A_H. In addition, it performs a number of computations of h equal to the number of blocks in y_1 plus the number of blocks in y_2. There is some more overhead, but small enough to neglect. ■

Bibliography

[1] M. Bellare and P. Rogaway. Collision-Resistant Hashing: Towards Making UOWHFs Practical. *Advances in Cryptology – CRYPTO '97*, Lecture Notes in Computer Science Vol. 1294, B. Kaliski ed., Springer-Verlag, 1997.

[2] M. Bellare and T. Kohno. Hash function balance and its impact on birthday attacks. *Advances in Cryptology – EUROCRYPT '04*, Lecture Notes in Computer Science Vol. 3027, C. Cachin and J. Camenisch ed., Springer-Verlag, 2004.

[3] I. Damgård. A Design Principle for Hash Functions. *Advances in Cryptology – CRYPTO '89*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989.

[4] B. den Boer and A. Bosselaers, Collisions for the compression function of MD5. *Advances in Cryptology – EUROCRYPT '93*, Lecture Notes in Computer Science Vol. 765, T. Helleseth ed., Springer-Verlag, 1993.

[5] H. Dobbertin, Cryptanalysis of MD4. *Fast Software Encryption—Cambridge Workshop*, Lecture Notes in Computer Science, vol. 1039, D. Gollman, ed., Springer-Verlag, 1996.

[6] H. Dobbertin, Cryptanalysis of MD5. Rump Session of Eurocrypt 96, May 1996, `http://www.iacr.org/conferences/ec96/rump/index.html`.

[7] H. Dobbertin, A. Bosselaers, B. Preneel. RIPEMD-160, a strengthened version of RIPEMD. *Fast Software Encryption '96*, Lecture Notes in Computer Science Vol. 1039, D. Gollmann ed., Springer-Verlag, 1996.

[8] M. Naor and M. Yung, Universal one-way hash functions and their cryptographic applications. *Proceedings of the 21st Annual Symposium on the Theory of Computing*, ACM, 1989.

[9] National Institute of Standards. FIPS 180-2, Secure hash standard, August 2000. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`.

[10] R. Merkle. One way hash functions and DES. *Advances in Cryptology – CRYPTO '89*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989.

[11] R. Rivest, The MD4 message-digest algorithm, *Advances in Cryptology – CRYPTO '90*, Lecture Notes in Computer Science Vol. 537, A. J. Menezes and S. Vanstone ed., Springer-Verlag, 1990, pp. 303–311. Also IETF RFC 1320 (April 1992).

[12] R. Rivest, The MD5 message-digest algorithm. IETF RFC 1321 (April 1992).

Chapter 7

Message Authentication

In most people's minds, privacy is the goal most strongly associated to cryptography. But message authentication is arguably even more important. Indeed you may or may not care if some particular message you send out stays private, but you almost certainly do want to be sure of the originator of each message that you act on. Message authentication is what buys you that guarantee.

Message authentication allows one party—the sender—to send a message to another party—the receiver—in such a way that if the message is modified en route, then the receiver will almost certainly detect this. Message authentication is also called *data-origin authentication.* Message authentication is said to protect the *integrity* of a message, ensuring that each message that it is received and deemed acceptable is arriving in the same condition that it was sent out—with no bits inserted, missing, or modified.

Here we'll be looking at the shared-key setting for message authentication (remember that message authentication in the public-key setting is the problem addressed by *digital signatures*). In this case the sender and the receiver share a secret key, K, which they'll use to authenticate their transmissions. We'll define the message authentication goal and we'll describe some different ways to achieve it. As usual, we'll be careful to pin down the problem we're working to solve.

7.1 The setting

It is often crucial for an agent who receives a message to be sure who sent it. If a hacker can call into his bank's central computer and produce deposit transactions that *appears* to be coming from a branch office, easy wealth is just around the corner. If an unprivileged user can interact over the network with his company's mainframe in such a way that the machine *thinks* that the packets it is receiving are coming from the system administrator, then all the machine's access-control mechanisms are for naught. In such cases the risk is that an adversary A, the *forger*, will create messages that look like they come from some other party, S, the (legitimate) *sender.* The attacker will send a message M to R, the *receiver* (or *verifier*), under S's identity. The receiver R will be tricked into believing that M originates with S. Because of this wrong belief, R may inappropriately act on M.

The rightful sender S could be one of many different kinds of entities, like a person, a corporation, a network address, or a particular process running on a particular machine. As the receiver R, you might know that it is S that supposedly sent you the message M for a variety of reasons. For

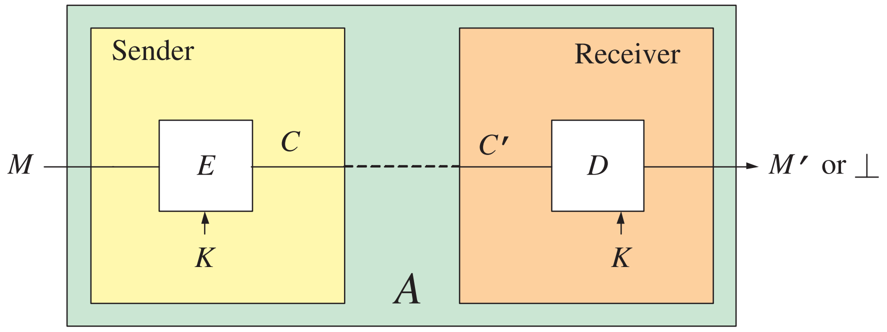

Figure 7.1: An authenticated-encryption scheme. Here we are authenticating messages with what is, syntactically, just an encryption scheme. The sender transmits a transformed version C of M and the receiver is able to recover $M' = M$ or else obtain indication of failure. Adversary A controls the communication channel and may even influence messages sent by the sender.

example, the message M might be tagged by an identifier which somehow names S. Or it might be that the manner in which M arrives is a route dedicated to servicing traffic from S.

Here we're going to be looking at the case when S and R already share some secret key, K. How S and R came to get this shared secret key is a separate question, one that we deal with later.

There are several high-level approaches for authenticating transmissions.

1. The most general approach works like this. To authenticate a message M using the key K, the sender will apply some encryption algorithm $\mathcal{E}$ to K, giving rise to a ciphertext C. When we speak of encrypting M in this context, we are using the word in the broadest possible sense, as any sort of keyed transformation on the message that obeys are earlier definition for the syntax of an encryption scheme; in particular, we are *not* suggesting that C conceals M. The sender S will transmit C to the receiver R. Maybe the receiver will receive C, or maybe it will not. The problem is that an adversary A may control the channel on which messages are being sent. Let C' be the message that the receiver actually gets. The receiver R, on receipt of C', will apply some decryption algorithm $\mathcal{D}$ to K and C'. We want that this should yield one of two things: (1) a message M' that is the original message M; or (2) an indication $\perp$ that C' be regarded as inauthentic. Viewed in this way, message authentication is accomplished by an encryption scheme. We are no longer interested in the privacy of the encryption scheme but, functionally, it is still an encryption scheme. See Fig. 7.1. We sometimes use the term *authenticated encryption* to indicate that we are using an encryption scheme to achieve authenticity.

2. Since our authenticity goal is not about privacy, most often the ciphertext C that the sender transmits is simply the original message M together with a tag T; that is, $C = \langle M, T \rangle$. When the ciphertext is of this form, we call the mechanism a *message-authentication scheme*. A message-authentication scheme will be specified by a tag-generation algorithm TG and a tag-verification algorithm VF. The former may be probabilistic or stateful; the latter is neither. The tag-generation algorithm TG produces a tag $T \xleftarrow{\$} \mathrm{TG}_K(M)$ from a key K and the message. The tag-verification algorithm $\mathrm{VF} \leftarrow \mathrm{VF}_K(M', T')$ produces a bit from a key K, a message M', and a tag T'. The intent is that the bit 1 tells the receiver to accept M', while the bit 0 tells the receiver to reject M'. See Fig. 7.5

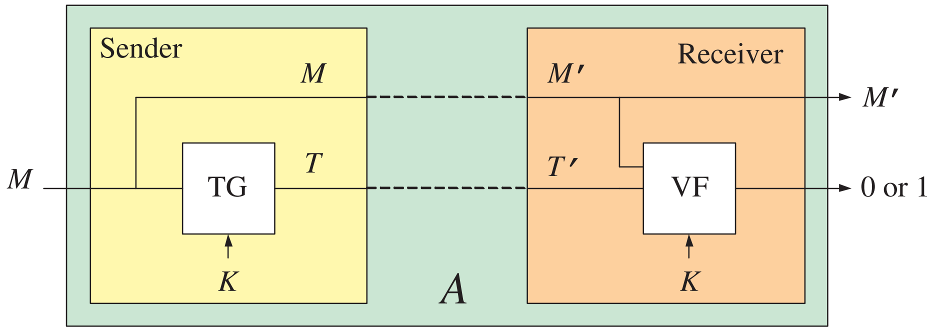

Figure 7.2: A message authentication scheme. This is a special case of the more general framework from the prior diagram. The authenticated message C is now understood to be the original message M together with a tag T. Separate algorithms generate the tag and verify the pair.

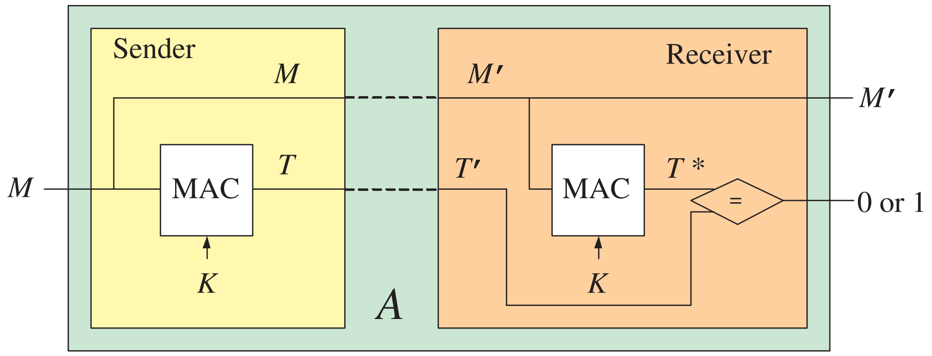

Figure 7.3: A message authentication code. This is a special case of a message authentication scheme. The authenticated message C is now understood to be the original message M together with a tag T that is computed as a deterministic and stateless function of M and K. The receiver verifies the authenticity of messages using the same MACing algorithm.

3. The most common possibility of all occurs when the tag-generation algorithm TG is deterministic and stateless. In this case we call the tag-generation algorithm, and the scheme itself, a *message authentication code*, or MAC. When authentication is accomplished using a MAC, we do not need to specify a separate tag-verification algorithm, for tag-verification always works he same way: the receiver, having received $\langle M', T' \rangle$, computes $T^* = \mathrm{MAC}_K(M')$. If this computed-tag T^* is identical to the received tag T' then the receiver regards the message M' as authentic; otherwise, the receiver regards M' as inauthentic. We write $T = \mathrm{MAC}_K(M)$ for the tag generated by the specified MAC. See Fig. 7.5

When the receiver decides that a message he has received is inauthentic what should he do? The receiver might want to just ignore the bogus message. Perhaps it was just noise on the channel; or perhaps taking action will do more harm than good, opening up new possibilities for denial-of-service attacks. Alternatively, the receiver may want to take more decisive actions, like tearing down the channel on which the message was received and informing some human being of apparent mischief. The proper course of action is dictated by the circumstances and the security policy of the receiver.

We point out that adversarial success in violating authenticity demands an active attack: to succeed, the adversary has to do more than listen—it has to get some bogus message to the receiver. In some communication scenarios it may be difficult for the adversary to get its messages to the receiver. For example, it may be tricky for an adversary to drop its own messages onto a physically secure phone line or fiber-optic channel. In other environments it may be trivial for the adversary to put messages onto the channel. Since we don't know what are the characteristics of the sender—receiver channel it is best to assume the worst and think that the adversary has plenty of power over this channel. We will actually assume even more than that, giving the adversary the power of creating legitimately authenticated messages.

We wish to emphasize that the message-authentication problem is very different from the privacy problem. We are not worried about secrecy of the message M; our concern is in whether the adversary can profit by injecting new messages into the communications stream. Not only is the problem conceptually different but, as we shall now see, privacy-providing encryption does nothing to ensure message authenticity.

7.2 Privacy does not imply authenticity

We know how to encrypt data so as to provide privacy, and something often suggested—and even done—is to encrypt as a way to provide authenticity. Fix a symmetric encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, and let parties S and R share a key K for this scheme. When S wants to send a message M to R, she encrypts it, transferring a ciphertext $M' = C$ generated via $C \xleftarrow{\$} \mathcal{E}_K(M)$. The receiver B decrypts it and, if it "makes sense", he regards the recovered message $M = \mathcal{D}_K(C)$ as authentic.

The argument that this works is as follows. Suppose, for example, that S transmits an ASCII message M_{100} which indicates that R should please transfer \$100 from the checking account of S to the checking account of some other party A. The adversary A wants to change the amount from the \$100 to \$900. Now if M_{100} had been sent in the clear, A can easily modify it. But if M_{100} is encrypted so that ciphertext C_{100} is sent, how is A to modify C_{100} so as to make S recover the different message M_{900}? The adversary A does not know the key K, so she cannot just encrypt M_{900} on her own. The privacy of C_{100} already rules out that C_{100} can be profitably tampered with.

The above argument is completely wrong. To see the flaws let's first look at a counter-example. If we encrypt M_{100} using a one time pad, then all the adversary has to do is to xor the byte of the ciphertext C_{100} that encodes the character "1" with the xor of the bytes for 1 and 9. That is, when we one-time pad encrypt, the privacy of the transmission does *not* make it difficult for the adversary to tamper with ciphertext so as to produce related ciphertexts.

How should one react to this counter-example? What you should *not* conclude is that one-time pad encryption is unsound. Our goal for the one-time pad was to provide privacy, and nothing we have said suggests that one-time pad encryption does not. Faulting the one-time pad encryption scheme for not providing authenticity is like faulting a car for not being able to fly; there is no reason to expect a tool designed to solve one problem to be effective at solving another.

You should *not* conclude that the example is contrived, and that you'd fare far better with some other encryption method. One-time-pad encryption is not at all contrived. And other methods of encryption, like CBC encryption, are only marginally better at protecting message integrity. This will be explored in the exercises.

You should *not* conclude that the failure stemmed from a failure to add "redundancy" before the message was encrypted. Adding redundancy is something like this: before the sender S encrypts his data he pads it with some known, fixed string, like 128 bits of zeros. When the receiver decrypts the ciphertext he checks whether the decrypted string ends in 128 zeros. He rejects the transmission if it does not. Such an approach can, and almost always will, fail. For example, the added redundancy does absolutely nothing for our one-time-pad example.

What you *should* conclude is that privacy-providing encryption was never an appropriate approach for protecting its authenticity. With hindsight, this is pretty clear. The fact that data is encrypted need not prevent an adversary from being able to make the receiver recover data different from that which the sender had intended. Indeed with most encryption schemes *any* ciphertext will decrypt to *something*, so even a random transmission will cause the receiver to receive something different from what the sender intended, which was not to send any message at all. Now perhaps the random ciphertext will look like garbage to the receiver, or perhaps not. Since we do not know what the receiver intends to do with his data it is impossible to say.

Since the encryption schemes we have discussed were not designed for authenticating messages, they don't. We emphasize this because the belief that good encryption, perhaps after adding redundancy, already provides authenticity, is not only voiced, but even printed in books or embedded into security systems.

Good cryptographic design is goal-oriented. One must understand and formalize our goal. Only then do we have the basis on which to design and evaluate potential solutions. Accordingly, our next step is to come up with a definition for a message-authentication scheme and its security.

7.3 Syntax for message authentication

In Section 7.1 we sketched three approaches, each more narrow than then the next, for providing authenticity. The first, which we called authenticated encryption, one provides authenticity by using what is a symmetric encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The imagined purpose shifts from providing privacy to providing authenticity, but the syntax of does not change. Recall that we already built into our definition of a symmetric encryption scheme the possibility that decryption would output a distinguished value $\perp$. We didn't use that capability in defining privacy—but we will need it for authenticity. Intuitively, the decryption mechanism outputting $\perp$ is interpreted as meaning that the ciphertext received (that is, the authenticated message) should be regarded as invalid.

We also singled out two more specific ways to provide authenticity. special cases of the above encryption schemes designed The first was a *message-authentication scheme*. Formally, this is a pair of algorithms (TG, VF). The first of these may be probabilistic or stateful, while the second is deterministic. Algorithm TG (for "tag generation") takes as input a string $K \in \mathcal{K}$, for some associated set $\mathcal{K}$, and a string $M \in \{0,1\}^*$. The set $\mathcal{K}$ is either finite or otherwise has an associated probability distribution (we must be able to choose a random point K from $\mathcal{K}$). The tag-generation algorithm TG produces a tag $T \xleftarrow{\$} \mathrm{TG}_K(M) \in \{0,1\}^* \cup \{\perp\}$. Algorithm VF (for "verification") takes as input strings $K \in$, $M \in \{0,1\}^*$, and $T \in \{0,1\}^*$. It outputs a bit $\mathrm{VF}_K(M,T) \in \{0,1\}$. The intended semantics is 1 for *accept* and 0 for *reject*. We insist that if $T \xleftarrow{\$} \mathrm{TG}_K(M)$ and $T \neq \perp$ then $\mathrm{VF}_K(M,T) = 1$. Every message-authentication scheme gives rise to an encryption scheme where $\mathcal{E}_K(M)$ computes $T \xleftarrow{\$} \mathrm{TG}_K(M)$ and returns $\langle M, T\rangle$, and $\mathcal{D}_K(\langle M,T\rangle) = M$ if $\mathrm{VF}_K(M,T) = 1$ while $\mathcal{D}_K(\langle M,T\rangle) = \perp$ otherwise. Of course this encryption scheme does nothing to provide privacy.

A *message authentication code* (MAC) corresponds to the special case of a message-authentication scheme in which tag-generation is deterministic and stateful. Formally, a message authentication code is a deterministic algorithm MAC: $\mathcal{K} \times \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ where $\mathcal{K}$ is a finite set, or is otherwise endowed with a probability distribution. The tag for a message M is $T = \mathrm{MAC}_K(M)$.

To verify $\langle M,T\rangle$ the receiver checks if $T = \mathrm{MAC}_K(M)$. If so, message M is viewed as authentic; otherwise, the message is viewed as being a forgery.

Note that our definitions don't permit stateful message-recovery / verification. Stateful functions for the receiver can be problematic because of the possibility of messages not reaching their destination—it is too easy for the receiver to be in a state different from the one that we'd like. All the same, stateful MAC verification functions are essential for detecting "replay attacks."

Recall that it was essential for the IND-CPA security of an encryption scheme that the encryption algorithm be probabilistic or stateful—you couldn't achieve IND-CPA security with a deterministic encryption algorithm. But we will see that probabilism and state are not necessary for achieving secure message authentication. This realization is built into the fact that we deal with MACs.

7.4 Definitions of security

Let us concentrate first on message authentication codes. We begin with a discussion of the issues and then state a formal definition.

The goal that we seek to achieve with a MAC is to be able to detect any attempt by the adversary to modify the transmitted data. We don't want the adversary to be able to produce messages that the receiver will deem authentic—only the sender should be able to do this. That is, we don't want that the adversary A to be able to create a pair (M, Tag) such that $\mathrm{VF}_K(M, \mathit{Tag}) = 1$, but M did not originate with the sender S. Such a pair (M, Tag) is called a *forgery.* If the adversary can make such a pair, she is said to have forged.

In some discussions of security people assume that the adversary's goal is to recover the secret key K. Certainly if it could do this, it would be a disaster, since it could then forge anything. It is important to understand, however, that an adversary might be able to forge without being able to recover the key, and if all we asked was for the adversary to be unable to recover the key, we'd be asking too little. Forgery is what counts, not key recovery.

Now it should be admitted right away that some forgeries might be useless to the adversary. For example, maybe the adversary can forge, but it can only forge strings that look random; meanwhile, suppose that all "good" messages are supposed to have a certain format. Should this really be viewed as a forgery? The answer is *yes.* If checking that the message is of a certain format was really a part of validating the message, then that should have been considered as part of the message-authentication code. In the absence of this, it is not for us to make assumptions about how the messages are formatted or interpreted; we really have no idea. Good protocol design means the security is guaranteed no matter what is the application.

In our adversary's attempt to forge a message we could consider various attacks. The simplest setting is that the adversary wants to forge a message even though it has never seen any transmission sent by the sender. In this case the adversary must concoct a pair (M, T) that is valid, even though it hasn't obtained any information to help. This is called a *no-message attack.* It often falls short of capturing the capabilities of realistic adversaries, since an adversary who can inject bogus messages onto the communications media can probably see valid messages as well. We should let the adversary use this information.

Suppose the sender sends the transmission (M, T) consisting of some message M and its legitimate tag T. The receiver will certainly accept this—that is built into our definition. Now at once a simple attack comes to mind: the adversary can just repeat this transmission, (M, T), and get the receiver to accept it once again. This attack is unavoidable, for our MAC is a deterministic function that the receiver recomputes. If the receiver accepted (M, T) once, he's bound to do it again.

What we have just described is called a *replay attack*. The adversary sees a valid (M, T) from the sender, and at some later point in time it re-transmits it. Since the receiver accepted it the first time, he'll do so again.

Should a replay attack count as a valid forgery? In real life it usually should. Say the first message was "Transfer \$1000 from my account to the account of party A." Then party A may have a simple way to enriching herself: it just keeps replaying this same authenticated message, happily watching her bank balance grow.

It is important to protect against replay attacks. But for the moment we will not try to do this. We will say that a replay is *not* a valid forgery; to be valid a forgery must be of a message M which was *not* already produced by the sender. We will see later that we can always achieve security against replay attacks by simple means; that is, we can take any message authentication mechanism which is not secure against replay attacks and modify it—after making the receiver stateful—so that it will be secure against replay attacks. At this point, not worrying about replay attacks results in a cleaner problem definition. And it leads us to a more modular protocol-design approach—that is, we cut up the problem into sensible parts ("basic security" and then "replay security") solving them one by one.

Of course there is no reason to think that the adversary will be limited to seeing only one example message. Realistic adversaries may see millions of authenticated messages, and still it should be hard for them to forge.

For some message authentication schemes the adversary's ability to forge will grow with the number q_{s} of legitimate message-tag pairs it sees. Likewise, in some security systems the number of valid (M, T) pairs that the adversary can obtain may be architecturally limited. (For example, a stateful Signer may be unwilling to MAC more than a certain number of messages.) So when we give our quantitative treatment of security we will treat q_{s} as an important adversarial resource.

How exactly do all these tagged messages arise? We could think of there being some distribution on messages that the sender will authenticate, but in some settings it is even possible for the adversary to influence which messages are tagged. In the worst case, imagine that the adversary *itself* chooses which messages get authenticated. That is, the adversary chooses a message, gets its tag, chooses another message, gets its tag, and so forth. Then it tries to forge. This is called an *adaptive chosen-message attack*. It wins if it succeeds in forging the MAC of a message which it has not queried to the sender.

At first glance it may seem like an adaptive chosen-message attack is unrealistically generous to our adversary; after all, if an adversary could really obtain a valid tag for *any* message it wanted, wouldn't that make moot the whole point of authenticating messages? In fact, there are several good arguments for allowing the adversary such a strong capability. First, we will see examples—higher-level protocols that use MACs—where adaptive chosen-message attacks are quite realistic. Second, recall our general principles. We want to design schemes which are secure in *any* usage. This requires that we make worst-case notions of security, so that when we err in realistically modeling adversarial capabilities, we err on the side of caution, allowing the adversary more power than it might really have. Since eventually we will design schemes that meet our stringent notions of security, we only gain when we assume our adversary to be strong.

As an example of a simple scenario in which an adaptive chosen-message attack is realistic, imagine that the sender S is forwarding messages to a receiver R. The sender receives messages from any number of third parties, $A_1, \ldots, A_n$. The sender gets a piece of data M from party A_i along a secure channel, and then the sender transmits to the receiver $\langle i \rangle \parallel M \parallel MAC_K(\langle i \rangle \parallel M)$. This is the sender's way of attesting to the fact that he has received message M from party A_i. Now if one of these third parties, say A_1, wants to play an adversarial role, it will ask the sender

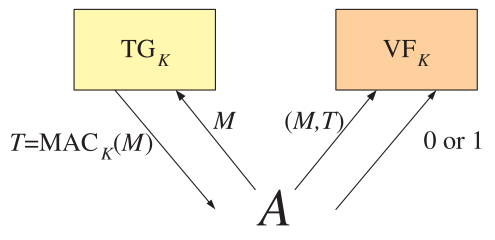

Figure 7.4: *The model for a message authentication code. Adversary A has access to a tag-generation oracle and a tag-verification oracle. The adversary wants to get the verification oracle to answer 1 to some (M,T) for which it didn't earlier ask the signing oracle M. The verification oracle returns 1 if $T = \mathrm{MAC}_K(M)$ and 0 if $T \neq \mathrm{MAC}_K(M)$.*

to forward its adaptively-chosen messages $M_1, M_2, \ldots$ to the receiver. If, based on what it sees, it can learn the key K, or even if it can learn to forge message of the form $\langle 2\rangle \parallel M$, so as to produce a valid $\langle 2\rangle \parallel M \parallel MAC_K(\langle 2\rangle \parallel M)$, then the intent of the protocol will have been defeated.

So far we have said that we want to give our adversary the ability to obtain MACs for messages of its choosing, and then we want to look at whether or not it can forge: produce a valid (M,T) pair where it never asked the sender to MAC M. But we should recognize that a realistic adversary might be able to produce lots of candidate forgeries, and it may be content if any of these turn out to be valid. We can model this possibility by giving the adversary the capability to tell if a prospective (M,T) pair is valid, and saying that the adversary forges if it ever finds an (M,T) pair that is but M was not MACed by the sender.

Whether or not a real adversary can try lots of possible forgeries depends on the context. Suppose the receiver is going to tear down a connection the moment he detects an invalid tag. Then it is unrealistic to try to use this receiver to help you determine if a candidate pair (M,T) is valid—one mistake, and you're done for. In this case, thinking of there being a single attempt to forge a message is quite adequate.

On the other hand, suppose that a receiver just ignores any improperly tagged message, while it responds in some noticeably different way if it receives a properly authenticated message. In this case a quite reasonable adversarial strategy may be ask the verifier about the validity of a large number of candidate (M,T) pairs. The adversary hopes to find at least one that is valid. When the adversary finds such an (M,T) pair, we'll say that it has won.

Let us summarize. To be fully general, we will give our adversary two different capabilities. The first adversarial capability is to obtain a MAC M for any message that it chooses. We will call this a signing query. The adversary will make some number of them, q_{s}. The second adversarial capability is to find out if a particular pair (M,T) is valid. We will call this a verification query. The adversary will make some number of them, q_{v}. Our adversary is said to succeed—to forge—if it ever makes a verification query (M,T) and gets a return value of 1 (ACCEPT) even though the message M is not a message that the adversary already knew a tag for by virtue of an earlier signing query. Let us now proceed more formally.

Let MAC: $\mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ be an arbitrary message authentication code. We will formalize a quantitative notion of security against adaptive chosen-message attack. We begin by describing the model.

We distill the model from the intuition we have described above. There is no need, in the model, to think of the sender and the verifier as animate entities. The purpose of the sender, from the adversary's point of view, is to authenticate messages. So we will embody the sender as an oracle that the adversary can use to authenticate any message M. This *tag-generation oracle*, as we will call it, is our way to provide the adversary black-box access to the function $\mathrm{MAC}_K(\cdot)$. Likewise, the purpose of the verifier, from the adversary's point of view, is to have that will test attempted forgeries. So we will embody the verifier as an oracle that the adversary can use to see if a candidate pair (M, T) is valid. This *verification oracle*, as we will call it, is our way to provide the adversary black-box access to the function $\mathrm{VF}_K(\cdot)$ which is 1 if $T = \mathrm{MAC}_K(M)$ and 0 otherwise. Thus, when we become formal, the cast of characters—the sender, receiver, and the adversary—gets reduced to just the adversary, running with its oracles.

Definition 7.4.1 [MAC security] Let MAC: $\mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ be a message authentication code and let A be an adversary. We consider the following experiment:

```
Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathrm{MAC}}(A)$
    $K \xleftarrow{\$} \mathcal{K}$
    Run $A^{\mathrm{MAC}_K(\cdot),\mathrm{VF}_K(\cdot,\cdot)}$ where $\mathrm{VF}_K(M,T)$ is 1 if $\mathrm{MAC}_K(M) = T$ and 0 otherwise
    if $A$ made a $\mathrm{VF}_K$ query $(M,T)$ such that
    –  The oracle returned 1, and
    –  $A$ did not, prior to making verification query $(M,T)$,
       make tag-generation query $M$
    then return 1 else return 0
```

The *uf-cma advantage* of A is defined as

$$\mathbf{Adv}^{\text{uf-cma}}_{\mathrm{MAC}}(A) \quad = \quad \Pr\left[\mathbf{Exp}^{\text{uf-cma}}_{\mathrm{MAC}}(A) \Rightarrow 1\right] \ . \ \blacksquare$$

Let us discuss the above definition. Fix a message authentication code MAC. Then we associate to any adversary A its "advantage," or "success probability." We denote this value as $\mathbf{Adv}^{\text{uf-cma}}_{\mathrm{MAC}}(A)$. It's just the chance that A manages to forge. The probability is over the choice of key K and the probabilistic choices, if any, that the adversary A makes.

As usual, the advantage that can be achieved depends both on the adversary strategy and the resources it uses. Informally, Π is secure if the advantage of a practical adversary is low.

As usual, there is a certain amount of arbitrariness as to which resources we measure. Certainly it is important to separate the oracle queries (q_{s} and q_{v}) from the time. In practice, signing queries correspond to messages sent by the legitimate sender, and obtaining these is probably more difficult than just computing on one's own. Verification queries correspond to messages the adversary hopes the verifier will accept, so finding out if it does accept these queries again requires interaction. Some system architectures may effectively limit q_{s} and q_{v}. No system architecture can limit t; that is limited primarily by the adversary's budget.

We emphasize that there are contexts in which you are happy with a MAC that makes forgery impractical when $q_{\mathrm{v}} = 1$ and $q_{\mathrm{s}} = 0$ (an "impersonation attack") and there are contexts in which you are happy when forgery is impractical when $q_{\mathrm{v}} = 1$ and $q_{\mathrm{s}} = 1$ (a "substitution attack"). But it is perhaps more common that you'd like for forgery to be impractical even when q_{s} is large, like 2^{50}, and when q_{v} is large, too.

Naturally the key K is not directly given to the adversary, and neither are any random choices or counter used by the MAC-generation algorithm. The adversary sees these things only to the extent that they are reflected in the answers to her oracle queries.

With a definition for MAC security in hand, it is not hard for us to similarly define authenticity for encryption schemes and message-authentication schemes. Let us do the former; we will explore the latter in exercises. We have an encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and we want to measure how effective an adversary is at attacking its authenticity.

Definition 7.4.2 [Authenticity of an encryption scheme] Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an encryption scheme and let A be an adversary. We consider the following experiment:

```
Experiment $\mathbf{Exp}_{\Pi}^{\mathrm{auth}}(A)$
    $K \xleftarrow{\$} \mathcal{K}$
    Run $A^{\mathcal{E}_K(\cdot), \mathrm{VF}_K(\cdot)}$ where $\mathrm{VF}_K(C)$ is 1 if $\mathcal{D}_K(C) \in \{0,1\}^*$ and 0 if $\mathcal{D}_K(C) = \perp$
    if $A$ made a $\mathrm{VF}_K$ query $C$ such that
    – The oracle returned 1, and
    – $A$ did not, prior to making verification query $C$,
        make an encryption query that returned $C$
    then return 1 else return 0
```

The *authenticity advantage* of A is defined as

$$\mathbf{Adv}_{\Pi}^{\mathrm{auth}}(A) \quad = \quad \Pr[\mathbf{Exp}_{\Pi}^{\mathrm{auth}}(A) \Rightarrow 1] \ . \ \blacksquare$$

We note that we could just as well have provided A with a decryption oracle $\mathcal{D}_K(\cdot)$ instead of a verification oracle $\mathrm{VF}_K(\cdot)$, giving the adversary credit if it ever manages to ask a this oracle a query C that decrypts to something other than $\perp$ and where C was not already returned by the encryption oracle.

7.5 Examples

Let us examine some example message authentication codes and use the definition to assess their strengths and weaknesses. We fix a PRF $F\colon\ \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. Our first scheme MAC1: $\mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ works as follows:

```
algorithm $\mathrm{MAC1}_K(M)$
    if ($|M| \bmod n \neq 0$ or $|M| = 0$) then return $\perp$
    Break $M$ into $n$-bit blocks $M = M_1 \ldots M_m$
    for $i \leftarrow 1$ to $m$ do $Y_i \leftarrow F_K(M_i)$
    $T \leftarrow Y_1 \oplus \cdots \oplus Y_n$
    return $T$
```

Now let us try to assess the security of this message authentication code.

Suppose the adversary wants to forge the tag of a certain given message M. A priori it is unclear this can be done. The adversary is not in possession of the secret key K, so cannot compute F_K and use it to compute T. But remember that the notion of security we have defined says that the adversary is successful as long as it can produce a correct tag for *some* message, not necessarily a given one. We now note that even without a chosen-message attack (in fact without seeing any examples of correctly tagged data) the adversary can do this. It can choose a message M consisting of two equal blocks, say $M = X \,\|\, X$ where X is some n-bit string, set $T \leftarrow 0^n$, and make verification query (M, T). Notice that $\mathrm{VF}_K(M, \mathit{Tag}) = 1$ because $F_K(x) \oplus F_K(x) = 0^n = T$. In more detail, the adversary is as follows.

```
algorithm $A_1^{\mathrm{MAC}_K(\cdot), \mathrm{VF}_K(\cdot,\cdot)}$
    Let $X$ be any $n$-bit string
    $M \leftarrow X \,\|\, X$
    $T \leftarrow 0^n$
    $d \leftarrow \mathrm{VF}_K(M, T)$
```

Then $\mathbf{Adv}^{\text{uf-cma}}_{\text{MAC}}(A_1) = 1$. Furthermore A_1 makes no signing oracle queries, uses $t = O(n)$ time, and its verification query has length $2n$-bits, so it is very practical.

There are many other attacks. For example we note that

$$T = F_K(M_1) \oplus F_K(M_2)$$

is not only the tag of M_1M_2 but also the tag of M_2M_1. So it is possible, given the tag of a message, to forge the tag of a new message formed by permuting the blocks of the old message. We leave it to the reader to specify the corresponding adversary and compute its advantage.

Let us now try to strengthen the scheme to avoid these attacks. Instead of applying F_K to a data block, we will first prefix the data block with its index. To do this, first pick some parameter ι with $1 \leq \iota \leq n-1$. We will write each block's index as an ι-bit string. The MAC-generation algorithm is the following:

```
algorithm MAC2$_K(M)$
    $\eta \leftarrow n - \iota$
    if ($|M| \bmod \eta \neq 0$ or $|M| = 0$ or $|M|/\eta \geq 2^\iota$) then return $\perp$
    Break $M$ into $\eta$-bit blocks $M = M_1 \ldots M_m$
    for $i \leftarrow 1$ to $m$ do $Y_i \leftarrow F_K([i]_\iota \,\|\, M_i)$
    $T \leftarrow Y_1 \oplus \cdots \oplus Y_m$
    return $Tag$
```

As the code indicates, we divide M into blocks, but the size of each block is smaller than in our previous scheme: it is now only $\eta = n - \iota$ bits. Then we prefix the i-th message block with the value i itself, the block index, written in binary as a string of length exactly m bits. It is to this padded block that we apply F_K before taking the xor.

Note that encoding of the block index i as an *iota*-bit string is only possible if $i < 2^\iota$. This means that we cannot authenticate a message M having more 2^ι blocks. This explains the conditions under which the MAC returns $\perp$. However this is a feasible restriction in practice, since a reasonable value of ι, like $\iota = 32$, is large enough that very long messages will be in the message space.

Anyway, the question we are really concerned with is the security. Has this improved from scheme MAC1? Begin by noticing that the attacks we found on MAC1 no longer work. For example if X is an η-bit string and we let $M = X \,\|\, X$ then its tag is *not* likely to be 0^n. Similarly, the second attack discussed above, namely that based on permuting of message blocks, also has low chance of success against the new scheme. Why? In the new scheme, if M_1, M_2 are strings of length η, then

$$\begin{aligned} \text{MAC2}_K(M_1M_2) &= F_K([1]_\iota \,\|\, M_1) \oplus F_K([2]_\iota \,\|\, M_2) \\ \text{MAC2}_K(M_2M_1) &= F_K([1]_m \,\|\, M_2) \oplus F_K([2]_\iota \,\|\, M_1) \,. \end{aligned}$$

These are unlikely to be equal. As an exercise, a reader might upper bound the probability that these values are equal in terms of the value of the advantage of F at appropriate parameter values.

All the same, MAC2 is still insecure. The attack however require a more non-trivial usage of the chosen-message attacking ability. The adversary will query the tagging oracle at several related points and combine the responses into the tag of a new message. We call it A_2–

```
algorithm $A_2^{\text{MAC}_K(\cdot),\text{VF}_K(\cdot)}$
    Let $A_1, B_1$ be distinct, $\eta$-bit strings
    Let $A_2, B_2$ be distinct $\eta$-bit strings
    $T_1 \leftarrow \text{MAC}_K(A_1A_2)$ ; $T_2 \leftarrow \text{MAC}_K(A_1B_2)$ ; $T_3 \leftarrow \text{MAC}_K(B_1A_2)$
    $T \leftarrow T_1 \oplus T_2 \oplus T_3$
    $d \leftarrow \mathrm{VF}_K(B_1B_2, T)$
```

We claim that $\mathbf{Adv}^{\text{uf-cma}}_{\text{MAC2}}(A_2) = 1$. Why? This requires two things. First that $\mathrm{VF}_K(B_1B_2, T) = 1$, and second that B_1B_2 was never a query to $\mathrm{MAC}_K(\cdot)$ in the above code. The latter is true because we insisted above that $a_1 \neq b_1$ and $a_2 \neq b_2$, which together mean that $B_1B_2 \notin \{A_1A_2, A_1B_2, B_1A_2\}$. So now let us check the first claim. We use the definition of the tagging algorithm to see that

$$\begin{aligned} T_1 &= F_K([1]_\iota \,\|\, A_1) \oplus F_K([2]_\iota \,\|\, A_2) \\ T_2 &= F_K([1]_\iota \,\|\, A_1) \oplus F_K([2]_\iota \,\|\, B_2) \\ T_3 &= F_K([1]_\iota \,\|\, B_1) \oplus F_K([2]_\iota \,\|\, A_2)\ . \end{aligned}$$

Now look how A_2 defined T and do the computation; due to cancellations we get

$$\begin{aligned} T &= T_1 \oplus T_2 \oplus T_3 \\ &= F_K([1]_\iota \,\|\, B_1) \oplus F_K([2]_\iota \,\|\, B_2)\ . \end{aligned}$$

This is indeed the correct tag of B_1B_2, meaning the value T' that $\mathrm{VF}_K(B_1B_2, T)$ would compute, so the latter algorithm returns 1, as claimed. In summary we have shown that this scheme is insecure.

It turns out that a slight modification of the above, based on use of a counter or random number chosen by the MAC algorithm, actually yields a secure scheme. For the moment however we want to stress a feature of the above attacks. Namely that these attacks did not cryptanalyze the PRF F. The attacks did not care anything about the structure of F; whether it was DES, AES, or anything else. They found weaknesses in the message authentication schemes themselves. In particular, the attacks work just as well when F_K is a random function, or a "perfect" cipher. This illustrates again the point we have been making, about the distinction between a tool (here the PRF) and its usage. We need to make better usage of the tool, and in fact to tie the security of the scheme to that of the underlying tool in such a way that attacks like those illustrated here are provably impossible under the assumption that the tool is secure.

7.6 The PRF-as-a-MAC paradigm

Pseudorandom functions make good MACs, and constructing a MAC in this way is an excellent approach. Here we show why PRFs are good MACs, and determine the concrete security of the underlying reduction. The following shows that the reduction is almost tight—security hardly degrades at all.

Let F: $\mathcal{K} \times D \to \{0,1\}^\tau$ be a family of functions. We associate to F a message authentication code MAC: $\mathcal{K} \times D \to \{0,1\}^\tau$ via

```
algorithm $\mathrm{MAC}_K(M)$
    if $(M \notin D)$ then return $\perp$
    $T \leftarrow F_K(M)$
    return $T$
```

Note that when we think of a PRF as a MAC it is important that the domain of the PRF be whatever one wants as the domain of the MAC. So such a PRF probably won't be realized as a blockcipher. It may have to be realized by a PRF that allows for inputs of many different lengths, since you might want to MAC messages of many different lengths. As yet we haven't demonstrated that we can make such PRFs. But we will. Let us first relate the security of the above MAC to that of the PRF.

Proposition 7.6.1 Let $F\colon\ \mathcal{K}\times D \to \{0,1\}^\tau$ be a family of functions and let MAC be the associated message authentication code as defined above. Let A by any adversary attacking Π, making q_{s} MAC-generation queries of total length μ_s, q_{v} MAC-verification queries of total length μ_v, and having running time t. Then there exists an adversary B attacking F such that

$$\mathbf{Adv}_{\Pi}^{\text{uf-cma}}(A) \quad\leq\quad \mathbf{Adv}_{F}^{\text{prf}}(B) + \frac{q_{\mathrm{v}}}{2^\tau}\,. \tag{7.1}$$

Furthermore B makes $q_{\mathrm{s}} + q_{\mathrm{v}}$ oracle queries of total length $\mu_s + \mu_v$ and has running time t.

Proof: Remember that B is given an oracle for a function $f\colon\ D \to \{0,1\}^\tau$. It will run A, providing it an environment in which A's oracle queries are answered by B.

```
algorithm $B^f$
    $d \leftarrow 0$ ; $S \leftarrow \emptyset$
    Run $A$
        When $A$ asks its signing oracle some query $M$:
            Answer $f(M)$ to $A$ ; $S \leftarrow S \cup \{M\}$
        When $A$ asks its verification oracle some query $(M, Tag)$:
            if $f(M) = Tag$ then
                answer 1 to $A$ ; if $M \notin S$ then $d \leftarrow 1$
                else answer 0 to $A$
    Until $A$ halts
    return $d$
```

We now proceed to the analysis. We claim that

$$\Pr\left[\mathbf{Exp}_{F}^{\text{prf-1}}(B)\Rightarrow 1\right] \quad=\quad \mathbf{Adv}_{\Pi}^{\text{uf-cma}}(A) \tag{7.2}$$

$$\Pr\left[\mathbf{Exp}_{F}^{\text{prf-0}}(B)\Rightarrow 1\right] \quad\leq\quad \frac{q_{\mathrm{v}}}{2^\tau}\,. \tag{7.3}$$

Subtracting, we get Equation (7.1). Let us now justify the two equations above.

In the first case f is an instance of F, so that the simulated environment that B is providing for A is exactly that of experiment $\mathbf{Exp}_{\Pi}^{\text{uf-cma}}(A)$. Since B returns 1 exactly when A makes a successful verification query, we have Equation (7.2).

In the second case, A is running in an environment that is alien to it, namely one where a random function is being used to compute MACs. We have no idea what A will do in this environment, but no matter what, we know that the probability that any particular verification query (M, Tag) with $M \notin S$ will be answered by 1 is at most $2^{-\tau}$, because that is the probability that $Tag = f(M)$. Since there are at most q_{v} verification queries, Equation (7.3) follows. ▌

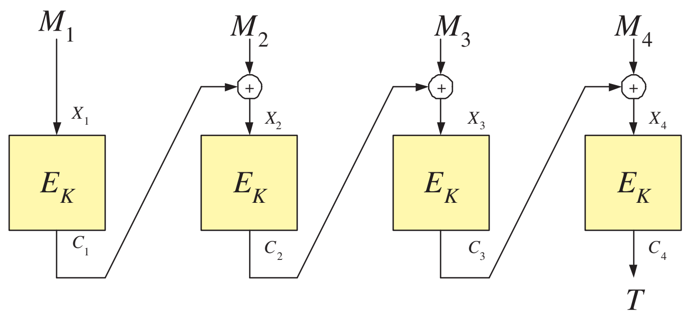

Figure 7.5: *The CBC MAC, here illustrated with a message M of four blocks, $M = M_1M_2M_3M_4$.*

7.7 The CBC MAC

A very popular class of MACs is obtained via cipher-block chaining of a given blockcipher. The method is as follows:

Scheme 7.7.1 CBC MAC] Let E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher. The CBC MAC over blockcipher E has key space $\mathcal{K}$ and is given by the following algorithm:

```
algorithm MAC$_K(M)$
    if $M \notin (\{0,1\}^n)^+$ then return $\perp$
    Break $M$ into $n$-bit blocks $M_1 \cdots M_m$
    $C_0 \leftarrow 0^n$
    for $i = 1$ to $m$ do $C_i \leftarrow E_K(C_{i-1} \oplus M_i)$
    return $C_m$
```

See Fig. 7.5 for an illustration with $m = 4$. ▌

As we will see below, the CBC MAC is secure only if you restrict attention to strings of some one particular length: the domain is restricted to $\{0,1\}^{mn}$ for some constant m. If we apply the CBC MAC across messages of varying lengths, it will be easy to distinguish this object from a random function.

Theorem 7.7.2 *[2] Fix $n \geq 1$, $m \geq 1$, and $q \geq 2$. Let A be an adversary that asks at most q queries, each of mn bits. Then that*

$$\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{CBC}[\mathsf{Func}(mn,n)]}(A) \;\leq\; \frac{m^2q^2}{2^n} \,. \blacksquare$$

Proof: Let A be an adversary that asks exactly q queries and assume without loss of generality that it never repeats a query. Refer to games C0–C9 in Fig. 7.6. Let us begin by explaining the notation used there. Each query M^s in the games is required to be a string of blocks, and we silently parse M^s to $M^s = M_1^s M_2^s \cdots M_m^s$ where each M_i is a block. Recall that $M_{1\to i}^s = M_1^s \cdots M_i^s$. The function π: $\{0,1\}^n \to \{0,1\}^n$ is initially undefined at each point. The set Domain(π) grows as we define points $\pi(X)$, while $\overline{\text{Range}}(\pi)$, initially $\{0,1\}^n$, correspondingly shrinks. The table $\mathbb{Y}$ stores blocks and is indexed by strings of blocks P having at most m blocks. A random block will come to occupy selected entries $\mathbb{Y}[X]$ except for $\mathbb{Y}[\varepsilon]$, which is initialized to the constant block 0^n and

```
On the $s^{\text{th}}$ query $F(M^s)$                          Game C1
100 $P \leftarrow \mathsf{Prefix}(M^1, \ldots, M^s)$
101 $C \leftarrow \mathbb{Y}[P]$
102 for $j \leftarrow \|P\|_n + 1$ to $m$ do
103   $X \leftarrow C \oplus M^s_j$
104   $C \xleftarrow{\$} \{0,1\}^n$
105   if $C \in \mathrm{Range}(\pi)$ then $bad \leftarrow \mathsf{true}$, $C \xleftarrow{\$} \overline{\mathrm{Range}(\pi)}$
106   if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$, $C \leftarrow \pi(X)$
107   $\pi(X) \leftarrow C$
108   $\mathbb{Y}[M^s_{1\to j}] \leftarrow C$                      omit for Game C0
109 return $C$
```

```
On the $s^{\text{th}}$ query $F(M^s)$                          Game C2
200 $P \leftarrow \mathsf{Prefix}(M^1, \ldots, M^s)$
201 $C \leftarrow \mathbb{Y}[P]$
202 for $j \leftarrow \|P\|_n + 1$ to $m$ do
203     $X \leftarrow C \oplus M^s_j$
204     $C \xleftarrow{\$} \{0,1\}^n$
205     if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
206     $\pi(X) \leftarrow C$
207     $\mathbb{Y}[M^s_{1\to j}] \leftarrow C$
208 return $C$
```

```
On the $s^{\text{th}}$ query $F(M^s)$                          Game C3
300 $P \leftarrow \mathsf{Prefix}(M^1, \ldots, M^s)$
301 $C \leftarrow \mathbb{Y}[P]$
302 for $j \leftarrow \|P\|_n + 1$ to $m$ do
303     $X \leftarrow C \oplus M^s_j$
304     $C \xleftarrow{\$} \{0,1\}^n$
305     if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
306     $\pi(X) \leftarrow \mathsf{defined}$
307     $\mathbb{Y}[M^s_{1\to j}] \leftarrow C$
308 return $C$
```

```
On the $s^{\text{th}}$ query $F(M^s)$                          Game C4
400 $P \leftarrow \mathsf{Prefix}(M^1, \ldots, M^s)$
401 $C \leftarrow \mathbb{Y}[P]$
402 for $j \leftarrow \|P\|_n + 1$ to $m$ do
403     $X \leftarrow C \oplus M^s_j$
404     if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
405     $\pi(X) \leftarrow \mathsf{defined}$
406     $C \leftarrow \mathbb{Y}[M^s_{1\to j}] \xleftarrow{\$} \{0,1\}^n$
407 $Z^s \xleftarrow{\$} \{0,1\}^n$
408 return $Z^s$
```

```
500 for $s \leftarrow 1$ to $q$ do                             Game C5
501     $\mathtt{P}^s \leftarrow \mathsf{Prefix}(\mathtt{M}^1, \ldots, \mathtt{M}^s)$
502     $C \leftarrow \mathbb{Y}[\mathtt{P}^s]$
503     for $j \leftarrow \|\mathtt{P}^s\|_n + 1$ to $m$ do
504         $X \leftarrow C \oplus \mathtt{M}^s_j$
505         if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
506         $\pi(X) \leftarrow \mathsf{defined}$
507         $C \leftarrow \mathbb{Y}[\mathtt{M}^s_{1\to j}] \xleftarrow{\$} \{0,1\}^n$
```

```
600 for $s \leftarrow 1$ to $q$ do                             Game C6
601     $\mathtt{P}^s \leftarrow \mathsf{Prefix}(\mathtt{M}^1, \ldots, \mathtt{M}^s)$
602     $C \leftarrow \mathbb{Y}[\mathtt{P}^s]$
603     $X \leftarrow C \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}$
604     if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
605     $\pi(X) \leftarrow \mathsf{defined}$
606     $C \leftarrow \mathbb{Y}[\mathtt{M}^s_{1\to\|\mathtt{P}^s\|_n+1}] \xleftarrow{\$} \{0,1\}^n$
607     for $j \leftarrow \|\mathtt{P}^s\|_n + 2$ to $m$ do
608         $X \leftarrow C \oplus \mathtt{M}^s_j$
609         if $X \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
610         $\pi(X) \leftarrow \mathsf{defined}$
611         $C \leftarrow \mathbb{Y}[\mathtt{M}^s_{1\to j}] \xleftarrow{\$} \{0,1\}^n$
```

```
700 for $X \in \{0,1\}^+$ do $\mathbb{Y}[X] \xleftarrow{\$} \{0,1\}^n$          Game C7
701 for $s \leftarrow 1$ to $q$ do
702   $\mathtt{P}^s \leftarrow \mathsf{Prefix}(\mathtt{M}^1, \ldots, \mathtt{M}^s)$
703   if $\mathbb{Y}[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} \in \mathrm{Domain}(\pi)$ then $bad \leftarrow \mathsf{true}$
704   $\pi(\mathbb{Y}[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}) \leftarrow \mathsf{defined}$
705   for $j \leftarrow \|\mathtt{P}^s\|_n + 2$ to $m$ do
706     if $\mathbb{Y}[\mathtt{M}^s_{1\to j-1}] \oplus \mathtt{M}^s_j \in \mathrm{Domain}(\pi)$ then $\mathsf{bad} \leftarrow \mathsf{true}$
707     $\pi(\mathbb{Y}[\mathtt{M}^s_{1\to j-1}] \oplus \mathtt{M}^s_j) \leftarrow \mathsf{defined}$
```

```
800 for $X \in \{0,1\}^+$ do $\mathbb{Y}[X] \xleftarrow{\$} \{0,1\}^n$          Game C8
801 for $s \leftarrow 1$ to $q$ do
802   $\mathtt{P}^s \leftarrow \mathsf{Prefix}(\mathtt{M}^1, \ldots, \mathtt{M}^s)$
803   if $\mathbb{Y}[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} \in \mathrm{Domain}(\pi)$ then$\mathsf{bad} \leftarrow \mathsf{true}$
804   $\pi(\mathbb{Y}[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}) \leftarrow \mathsf{defined}$
805   for $j \leftarrow \|\mathtt{P}^s\|_n + 1$ to $m - 1$ do
806    if $\mathbb{Y}[\mathtt{M}^s_{1\to j}] \oplus \mathtt{M}^s_{j+1} \in \mathrm{Domain}(\pi)$ then $\mathsf{bad} \leftarrow \mathsf{true}$
807    $\pi(\mathbb{Y}[\mathtt{M}^s_{1\to j}] \oplus \mathtt{M}^s_{j+1}) \leftarrow \mathsf{defined}$
```

```
900 for $X \in \{0,1\}^+$ do $\mathbb{Y}[X] \xleftarrow{\$} \{0,1\}^n$          Game C9
901 for $s \leftarrow 1$ to $q$ do $\mathtt{P}^s \leftarrow \mathsf{Prefix}(\mathtt{M}^1, \ldots, \mathtt{M}^s)$
902 $\mathsf{bad} \leftarrow \exists (r,i) \neq (s,j)(r \le s)(i \ge \|\mathtt{P}^r\|_n + 1)(j \ge \|\mathtt{P}^s\|_n + 1)$
903      $\mathbb{Y}[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}$ and $r < s$ or
904      $\mathbb{Y}[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1} = \mathbb{Y}[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}$ or
905      $\mathbb{Y}[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1} = \mathbb{Y}[\mathtt{M}^s_{1\to j}] \oplus \mathtt{M}^s_{j+1}$ or
906      $\mathbb{Y}[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}[\mathtt{M}^s_{1\to j}] \oplus \mathtt{M}^s_{j+1}$
```

Figure 7.6: Games used in the CBC MAC analysis. Let $\mathsf{Prefix}(M^1, \ldots, M^s)$ be ε if $s = 1$, else the longest string $P \in (\{0,1\}^n)^*$ s.t. P is a prefix of M^s and M^r for some $r < s$. In each game, **Initialize** sets $\mathbb{Y}[\varepsilon] \leftarrow 0^n$.

is never changed. The value defined (introduced at line 306) is an arbitrary point of $\{0,1\}^n$, say 0^n. Finally, $\mathsf{Prefix}(M^1, \ldots, M^s)$ is the longest string of blocks $P = P_1 \cdots P_p$ that is a prefix of M^s and is also a prefix of M^r for some $r < s$. If Prefix is applied to a single string the result is the empty string, $\mathsf{Prefix}(P^1) = \varepsilon$. As an example, letting A, B, and C be distinct blocks, Prefix(ABC) $= \varepsilon$, Prefix(ACC, ACB, ABB, ABA) $=$ AB, and Prefix(ACC, ACB, BBB) $= \varepsilon$.

We briefly explain the game chain up until the terminal game. Game C0 is obtained from game C1 by dropping the assignment statements that immediately follow the setting of bad. Game **C1** is a realization of $\mathrm{CBC}^m[\mathsf{Perm}(n)]$ and game **C0** is a realization of $\mathsf{Func}(mn,n)$. Games C1 and C0 are designed so that the fundamental lemma applies, so the advantage of A in attacking the CBC construction is at most $\Pr[A^{\mathrm{C0}}$ sets bad]. **C0→C2**: The C0 → C2 transition is a lossy transition that takes care of bad getting set at line 105, which clearly happens with probability at most $(0+1+\cdots+(mq-1))/2^n \le 0.5\, m^2q^2/2^n$, so $\Pr[A^{\mathrm{C0}}$ sets bad$] \le \Pr[A^{\mathrm{C2}}$ sets bad$] + 0.5\, m^2q^2/2^n$. **C2→C3**: Next notice that in game C2 we never actually use the values assigned to π, all that matters is that we *record* that a value had been placed in the domain of π, and so game C3 does just that, dropping a fixed value defined $= 0^n$ into $\pi(X)$ when we want X to join the domain of π. **C3→C4**: Now notice that in game C3 the value returned to the adversary, although dropped into $\mathbb{Y}\,[M_1^s \cdots M_m^s]$, is never subsequently used in the game so we could as well choose a random value Z^s and return it to the adversary, doing nothing else with Z^s. This is the change made for game C4. The transition is conservative. **C4→C5**: Changing game C4 to C5 is by the "coin-fixing" technique. Coin-fixing in this case amounts to letting the adversary choose the sequence of queries $\mathtt{M}^1, \ldots, \mathtt{M}^m$ it asks and the sequence of answers returned to it. The queries still have to be valid: each M^s is an mn-bit string different from all prior ones: that is the query/response set. For the worst $\mathtt{M}^1, \ldots, \mathtt{M}^m$, which the coin-fixing technique fixes, $\Pr[A^{\mathrm{C4}}$ sets bad$] \le \Pr[$C5 sets bad]. Remember that, when applicable, coin-fixing is safe. **C5→C6**: Game C6 unrolls the first iteration of the loop at lines 503–507. This transformation is conservative. **C6→C7**: Game C7 is a rewriting of game C6 that omits mention of the variables C and X, directly using values from the $\mathbb{Y}$-table instead, whose values are now chosen at the beginning of the game. The change is conservative. **C7→C8**: Game C8 simply re-indexes the for loop at line 705. The change is conservative. **C8→C9**: Game C9 restructures the setting of bad inside the loop at 802–807 to set bad in a single statement. Points were into the domain of π at lines 804 and 807 and we checked if any of these points coincide with specified other points at lines 803 and 806. The change is conservative.

At this point, we have only to bound $\Pr[A^{\mathrm{C9}}$ sets bad]. We do this using the sum bound and a case analysis. Fix any $r, i,\ s, j$ as specified in line 902. Consider the following ways that bad can get set to true.

Line 903. We first bound $\Pr[\mathbb{Y}\,[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}]$. If $\mathtt{P}^r = \mathtt{P}^s = \varepsilon$ then $\Pr[\mathbb{Y}\,[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}] = \Pr[\mathtt{M}^r_1 = \mathtt{M}^s_1] = 0$ because $\mathtt{M}^r$ and $\mathtt{M}^s$, having only ε as a common block prefix, must differ in their first block. If $\mathtt{P}^r = \varepsilon$ but $\mathtt{P}^s \ne \varepsilon$ then $\Pr[\mathbb{Y}\,[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}] = \Pr[\mathtt{M}^r_1 = \mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}] = 2^{-n}$ since the probability expression involves the single random variable $\mathbb{Y}\,[\mathtt{P}^s]$ that is uniformly distributed in $\{0,1\}^n$. If $\mathtt{P}^r \ne \varepsilon$ and $\mathtt{P}^s = \varepsilon$ the same reasoning applies. If $\mathtt{P}^r \ne \varepsilon$ and $\mathtt{P}^s \ne \varepsilon$ then $\Pr[\mathbb{Y}\,[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}] = 2^{-n}$ unless $\mathtt{P}^r = \mathtt{P}^s$, so assume that to be the case. Then $\Pr[\mathbb{Y}\,[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}] = \Pr[\mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1}] = 0$ because $\mathtt{P}^r = \mathtt{P}^s$ is the *longest* block prefix that coincides in $\mathtt{M}^r$ and $\mathtt{M}^s$.

Line 904. We want to bound $\Pr[\mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} = \mathbb{Y}\,[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1}]$. If $\mathtt{P}^s = \varepsilon$ then $\Pr[\mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} = \mathbb{Y}\,[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1}] = \Pr[\mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} = \mathbb{Y}\,[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1}] = 2^{-n}$ because it involves a single random value $\mathbb{Y}\,[\mathtt{M}^r_{1\to i}]$. So assume that $\mathtt{P}^s \ne \varepsilon$. Then $\Pr[\mathbb{Y}\,[\mathtt{P}^s] \oplus \mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} = \mathbb{Y}\,[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1}] = 2^{-n}$ unless

$\mathtt{P}^s = \mathtt{M}^r_{1\to i}$ in which case we are looking at $\Pr[\mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} = \mathtt{M}^r_{\|\mathtt{P}^s\|_n+1}]$. But this is 0 because $\mathtt{P}^s = \mathtt{M}^r_{1\to i}$ means that the longest prefix that $\mathtt{M}^s$ shares with $\mathtt{M}^r$ is $\mathtt{P}^s$ and so $\mathtt{M}^s_{\|\mathtt{P}^s\|_n+1} \neq \mathtt{M}^r_{\|\mathtt{P}^s\|_n+1}$.

Line 905. What is $\mathbb{Y}[\mathtt{M}^s_{1\to j}] \oplus \mathtt{M}^s_{j+1} = \mathbb{Y}[\mathtt{M}^r_{1\to i}] \oplus \mathtt{M}^r_{i+1}$. It is 2^{-n} unless $i = j$ and $\mathtt{M}^s_{1\to j} = \mathtt{M}^r_{1\to i}$. In that case $\|\mathtt{P}^s\|_n \geq j$ and $\|\mathtt{P}^r\|_n \geq i$, contradicting our choice of allowed values for i and j at line 902.

Line 906. We must bound $\Pr[\mathbb{Y}[\mathtt{P}^r] \oplus \mathtt{M}^r_{\|\mathtt{P}^r\|_n+1} = \mathbb{Y}[\mathtt{M}^s_{1\to j}] \oplus \mathtt{M}^s_{j+1}]$. As before, this is 2^{-n} unless $\mathtt{P}^r = \mathtt{M}^s_{1\to j}$ but we can not have that $\mathtt{P}^r = \mathtt{M}^s_{1\to j}$ because $j \geq \|\mathtt{P}^s\|_n + 1$.

There are at most $0.5m^2q^2$ tuples (r, i, s, j) considered at line 902 and we now know that for each of them bad gets set with probability at most 2^{-n}. So Pr[Game C9 sets bad] $\leq 0.5m^2q^2/2^n$. Combining with the loss from the C0→C2 transition we have that Pr[Game C0 setsbad] $\leq m^2q^2/2^n$, completing the proof. ▌

7.8 The universal-hashing approach

We have shown that one paradigm for making a good MAC is to make something stronger: a good PRF. Unfortunately, out-of-the-box PRFs usually operate on strings of some fixed length, like 128 bits. That's almost certainly not the domain that we want for our MAC's message space. In this section we describe a simple paradigm for extending the domain of a PRF by using a universal hash-function family. Several MACs can be seen as instances of this approach.

Definition 7.8.1 Let $H\colon \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ and let δ be a real number. We say that H is δ-AU (read this as δ almost-universal) if for all distinct $M, M' \in \mathcal{M}$, $\Pr[K \xleftarrow{\$} \mathcal{K} : H_K(M) = H_K(M')] \leq \delta$.

Definition 7.8.2 Let $H\colon \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ and $F\colon \mathcal{K}' \times \{0,1\}^n \to \{0,1\}^\tau$ be function families. Then $F \circ H$ is the function family $F \circ H\colon (\mathcal{K} \times \mathcal{K}') \times \mathcal{M} \to \{0,1\}^\tau$ defined by $F \circ H_{(K,K')}(M) = F_{K'}(H_K(M))$.

Theorem 7.8.3 *Let $H\colon \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ be a δ-AU function family and let $F = \mathsf{Func}(n,\tau)$ be the family of all functions from n bits to τ bits. Let A be an adversary that asks at most q queries. Then $\mathbf{Adv}^{\mathrm{prf}}_{F\circ H}(A) \leq \binom{q}{2}\delta^2$.*

To be continued. Give a CW mod-p arithmetic MAC. Then describe EMAC and CMAC, and HMAC, probably in different sections.

7.9 Problems

Problem 39 Consider the following variant of the CBC MAC, intended to allow one to MAC messages of arbitrary length. The construction uses a blockcipher $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, which you should assume to be secure. The domain for the MAC is $(\{0,1\}^n)^+$. To MAC M under key K compute $\mathrm{CBC}_K(M \,\|\, |M|)$, where $|M|$ is the length of M, written in n bits. Of course K has k bits. Show that this MAC is completely insecure: break it with a constant number of queries.

Problem 40 Consider the following variant of the CBC MAC, intended to allow one to MAC messages of arbitrary length. The construction uses a blockcipher $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, which you should assume to be secure. The domain for the MAC is $(\{0,1\}^n)^+$. To MAC M under key (K, K') compute $\mathrm{CBC}_K(M) \oplus K'$. Of course K has k bits and K' has n bits. Show that this MAC is completely insecure: break it with a constant number of queries.

Problem 41 Let $\mathrm{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme and let $\mathrm{MA} = (\mathcal{K}', \mathrm{MAC}, \mathrm{VF})$ be a message authentication code. Alice (A) and Bob (B) share a secret key $K = (K1, K2)$ where $K1 \leftarrow \mathcal{K}$ and $K2 \leftarrow \mathcal{K}'$. Alice wants to send messages to Bob in a private and authenticated way. Consider her sending each of the following as a means to this end. For each, say whether it is a secure way or not, and briefly justify your answer. (In the cases where the method is good, you don't have to give a proof, just the intuition.)

- **(a)** $M, \mathrm{MAC}_{K2}(\mathcal{E}_{K1}(M))$
- **(b)** $\mathcal{E}_{K1}(M, \mathrm{MAC}_{K2}(M))$
- **(c)** $\mathrm{MAC}_{K2}(\mathcal{E}_{K1}(M))$
- **(d)** $\mathcal{E}_{K1}(M), \mathrm{MAC}_{K2}(M)$
- **(e)** $\mathcal{E}_{K1}(M), \mathcal{E}_{K1}(\mathrm{MAC}_{K2}(M))$
- **(f)** $C, \mathrm{MAC}_{K2}(C)$ where $C = \mathcal{E}_{K1}(M)$
- **(g)** $\mathcal{E}_{K1}(M, A)$ where A encodes the identity of Alice; B decrypts the received ciphertext C and checks that the second half of the plaintext is "A".

In analyzing these schemes, you should assume that the primitives have the properties guaranteed by their definitions, but no more; for an option to be good it must work for *any* choice of a secure encryption scheme and a secure message authentication scheme.

Now, out of all the ways you deemed secure, suppose you had to choose one to implement for a network security application. Taking performance issues into account, do all the schemes look pretty much the same, or is there one you would prefer?

Problem 42 Refer to problem 4.3. Given a blockcipher E: $\mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ construct a cipher E': $\mathcal{K}' \times \{0,1\}^{2n} \to \{0,1\}^{2n}$. Formalize and prove a theorem that shows that E' is a secure PRP if E is.

Problem 43 Let H: $\{0,1\}^k \times D \to \{0,1\}^L$ be a hash function, and let $\Pi = (\mathcal{K}, \mathrm{MAC}, \mathrm{VF})$ be the message authentication code defined as follows. The key-generation algorithm $\mathcal{K}$ takes no inputs and returns a random k-bit key K, and the tagging and verifying algorithms are:

```
algorithm $\mathrm{MAC}_K(M)$      | algorithm $\mathrm{VF}_K(M, Tag')$
    $Tag \leftarrow H(K, M)$       |     $Tag \leftarrow H(K, M)$
    return $Tag$                   |     if $Tag = Tag'$ then return 1
                                   |     else return 0
```

Show that

$$\mathbf{Adv}_H^{\text{cr2-hk}}(t, q, \mu) \quad \leq \quad (q-1) \cdot \mathbf{Adv}_\Pi^{\text{uf-cma}}(t', q-1, \mu, q-1, \mu)$$

for any t, q, μ with $q \geq 2$, where t' is $t + O(\log(q))$. (This says that if Π is a secure message authentication code then H was a CR2-HK secure hash function.)

Bibliography

[1] M. Bellare, J. Kilian and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences* , Vol. 61, No. 3, Dec 2000, pp. 362–399.

[2] M. Bellare, R. Canetti and H. Krawczyk. Keying hash functions for message authentication. *Advances in Cryptology – CRYPTO '96*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. *Advances in Cryptology – CRYPTO '99*, Lecture Notes in Computer Science Vol. 1666, M. Wiener ed., Springer-Verlag, 1999.

[4] J. Black and P. Rogaway. CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions. *Advances in Cryptology – CRYPTO '00*, Lecture Notes in Computer Science Vol. 1880, M. Bellare ed., Springer-Verlag, 2000.

Chapter 8

Authenticated Encryption

Authenticated encryption is the problem of achieving both privacy *and* authenticity in the shared-key setting. Historically, this goal was given very little attention by cryptographers. Perhaps people assumed that there was nothing to be said: combine what we did on encryption in Chapter 5 and what we did on message authentication in Chapter 7—end of story, no?

The answer is indeed *no*. First, from a theoretical point of view, achieving privacy and authenticity together is a *new* cryptographic goal—something different from achieving privacy and different from achieving authenticity. We need to look at what this goal actually means. Second, even if we do plan to achieve authenticated encryption using the tools we've already looked at, say an encryption scheme and a MAC, we still have to figure out *how* to combine these primitives in a way that is guaranteed to achieve security. Finally, if our goal is to achieve both privacy and authenticity then we may be able to achieve efficiency that would not be achievable if treating these goals separately.

Rest of Chapter to be written.

Chapter 9

Computational Number Theory

9.1 The basic groups

We let $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ denote the set of integers. We let $\mathbf{Z}_+ = \{1, 2, \ldots\}$ denote the set of positive integers and $\mathbf{N} = \{0, 1, 2, \ldots\}$ the set of non-negative integers.

9.1.1 Integers mod N

If a, b are integers, not both zero, then their greatest common divisor, denoted $\gcd(a, b)$, is the largest integer d such that d divides a and d divides b. If $\gcd(a, b) = 1$ then we say that a and b are relatively prime. If a, N are integers with $N > 0$ then there are unique integers r, q such that $a = Nq + r$ and $0 \leq r < N$. We call r the remainder upon division of a by N, and denote it by $a \bmod N$. We note that the operation $a \bmod N$ is defined for both negative and non-negative values of a, but only for positive values of N. (When a is negative, the quotient q will also be negative, but the remainder r must always be in the indicated range $0 \leq r < N$.) If a, b are any integers and N is a positive integer, we write $a \equiv b \pmod{N}$ if $a \bmod N = b \bmod N$. We associate to any positive integer N the following two sets:

$$\begin{aligned} \mathbf{Z}_N &= \{0, 1, \ldots, N-1\} \\ \mathbf{Z}_N^* &= \{\, i \in \mathbf{Z} \,:\, 1 \leq i \leq N-1 \text{ and } \gcd(i, N) = 1 \,\} \end{aligned}$$

The first set is called the set of integers mod N. Its size is N, and it contains exactly the integers that are possible values of $a \bmod N$ as a ranges over $\mathbf{Z}$. We define the Euler Phi (or totient) function φ: $\mathbf{Z}_+ \to \mathbf{N}$ by $\varphi(N) = |\mathbf{Z}_N^*|$ for all $N \in \mathbf{Z}_+$. That is, $\varphi(N)$ is the size of the set $\mathbf{Z}_N^*$.

9.1.2 Groups

Let G be a non-empty set, and let $\cdot$ be a binary operation on G. This means that for every two points $a, b \in G$, a value $a \cdot b$ is defined.

Definition 9.1.1 Let G be a non-empty set and let $\cdot$ denote a binary operation on G. We say that G is a *group* if it has the following properties:

1. Closure: For every $a, b \in G$ it is the case that $a \cdot b$ is also in G.

2. Associativity: For every $a, b, c \in G$ it is the case that $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

3. Identity: There exists an element $\mathbf{1} \in G$ such that $a \cdot \mathbf{1} = \mathbf{1} \cdot a = a$ for all $a \in G$.

4. Invertibility: For every $a \in G$ there exists a unique $b \in G$ such that $a \cdot b = b \cdot a = \mathbf{1}$.

The element b in the invertibility condition is referred to as the inverse of the element a, and is denoted a^{-1}. ▌

We now return to the sets we defined above and remark on their group structure. Let N be a positive integer. The operation of addition modulo N takes input any two integers a, b and returns $(a+b) \bmod N$. The operation of multiplication modulo N takes input any two integers a, b and returns $ab \bmod N$.

Fact 9.1.2 Let N be a positive integer. Then $\mathbf{Z}_N$ is a group under addition modulo N, and $\mathbf{Z}_N^*$ is a group under multiplication modulo N. ▌

In $\mathbf{Z}_N$, the identity element is 0 and the inverse of a is $-a \bmod N = N - a$. In $\mathbf{Z}_N^*$, the identity element is 1 and the inverse of a is a $b \in \mathbf{Z}_N^*$ such that $ab \equiv 1 \pmod{N}$. In may not be obvious why such a b even exists, but it does. We do not prove the above fact here.

In any group, we can define an exponentiation operation which associates to any $a \in G$ and any integer i a group element we denote a^i, defined as follows. If $i = 0$ then a^i is defined to be $\mathbf{1}$, the identity element of the group. If $i > 0$ then

$$a^i = \underbrace{a \cdot a \cdots a}_{i} .$$

If i is negative, then we define $a^i = (a^{-1})^{-i}$. Put another way, let $j = -i$, which is positive, and set

$$a^i = \underbrace{a^{-1} \cdot a^{-1} \cdots a^{-1}}_{j} .$$

With these definitions in place, we can manipulate exponents in the way in which we are accustomed with ordinary numbers. Namely, identities such as the following hold for all $a \in G$ and all $i, j \in \mathbf{Z}$:

$$\begin{aligned} a^{i+j} &= a^i \cdot a^j \\ (a^i)^j &= a^{ij} \\ a^{-i} &= (a^i)^{-1} \\ a^{-i} &= (a^{-1})^i . \end{aligned}$$

We will use this type of manipulation frequently without explicit explanation.

It is customary in group theory to call the size of a group G its *order*. That is, the order of a group G is $|G|$, the number of elements in it. We will often make use of the following basic fact. It says that if any group element is raised to the power the order of the group, the result is the identity element of the group.

Fact 9.1.3 Let G be a group and let $m = |G|$ be its order. Then $a^m = \mathbf{1}$ for all $a \in G$. ▌

This means that computation in the group indices can be done modulo m:

Proposition 9.1.4 Let G be a group and let $m = |G|$ be its order. Then $a^i = a^{i \bmod m}$ for all $a \in G$ and all $i \in \mathbf{Z}$. ▌

We leave it to the reader to prove that this follows from Fact 9.1.3.

Example 9.1.5 Let us work in the group $\mathbf{Z}_{21}^*$ under the operation of multiplication modulo 21. The members of this group are $1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20,$ so the order of the group is $m = 12$. Suppose we want to compute 5^{86} in this group. Applying the above we have

$$5^{86} \bmod 21 \;=\; 5^{86 \bmod 12} \bmod 21 \;=\; 5^2 \bmod 21 \;=\; 25 \bmod 21 \;=\; 4 \,. \blacksquare$$

If G is a group, a set $S \subseteq G$ is called a subgroup if it is a group in its own right, under the same operation as that under which G is a group. If we already know that G is a group, there is a simple way to test whether S is a subgroup: it is one if and only if $x \cdot y^{-1} \in S$ for all $x, y \in S$. Here y^{-1} is the inverse of y in G.

Fact 9.1.6 Let G be a group and let S be a subgroup of G. Then the order of S divides the order of G. $\blacksquare$

9.2 Algorithms

Fig. 9.1 summarizes some basic algorithms involving numbers. These algorithms are used to implement public-key cryptosystems, and thus their running time is an important concern. We begin with a discussion about the manner in which running time is measured, and then go on to discuss the algorithms, some very briefly, some in more depth.

9.2.1 Bit operations and binary length

In a course or text on algorithms, we learn to analyze the running time of an algorithm as a function of the size of its input. The inputs are typically things like graphs, or arrays, and the measure of input size might be the number of nodes in the graph or the length of the array. Within the algorithm we often need to perform arithmetic operations, like addition or multiplication of array indices. We typically assume these have $O(1)$ cost. The reason this assumption is reasonable is that the numbers in question are small and the cost of manipulating them is negligible compared to costs proportional to the size of the array or graph on which we are working.

In contrast, the numbers arising in cryptographic algorithms are large, having magnitudes like 2^{512} or 2^{1024}. The arithmetic operations on these numbers are the main cost of the algorithm, and the costs grow as the numbers get bigger.

The numbers are provided to the algorithm in binary, and the size of the input number is thus the number of bits in its binary representation. We call this the length, or binary length, of the number, and we measure the running time of the algorithm as a function of the binary lengths of its input numbers. In computing the running time, we count the number of bit operations performed.

Let $b_{k-1} \ldots b_1 b_0$ be the binary representation of a positive integer a, meaning $b_0, \ldots, b_{k-1}$ are bits such that $b_{k-1} = 1$ and $a = 2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \cdots + 2^1 b_1 + 2^0 b_0$. Then the binary length of a is k, and is denoted $|a|$. Notice that $|a| = k$ if and only if $2^{k-1} \leq a < 2^k$. If a is negative, we let $|a| = |-a|$, and assume that an additional bit or two is used to indicate to the algorithm that the input is negative.

9.2.2 Integer division and mod algorithms

We define the integer division function as taking input two integers a, N, with $N > 0$, and returning the quotient and remainder obtained by dividing a by N. That is, the function returns (q, r) such

Algorithm	**Input**		**Output**	**Running Time**
INT-DIV	a, N	$(N > 0)$	(q, r) with $a = Nq + r$ and $0 \le r < N$	$O(\|a\| \cdot \|N\|)$
MOD	a, N	$(N > 0)$	$a \bmod N$	$O(\|a\| \cdot \|N\|)$
EXT-GCD	a, b	$((a, b) \neq (0, 0))$	$(d, \overline{a}, \overline{b})$ with $d = \gcd(a, b) = a\overline{a} + b\overline{b}$	$O(\|a\| \cdot \|b\|)$
MOD-ADD	a, b, N	$(a, b \in \mathbf{Z}_N)$	$(a + b) \bmod N$	$O(\|N\|)$
MOD-MULT	a, b, N	$(a, b \in \mathbf{Z}_N)$	$ab \bmod N$	$O(\|N\|^2)$
MOD-INV	a, N	$(a \in \mathbf{Z}_N^*)$	$b \in \mathbf{Z}_N^*$ with $ab \equiv 1 \pmod N$	$O(\|N\|^2)$
MOD-EXP	a, n, N	$(a \in \mathbf{Z}_N)$	$a^n \bmod N$	$O(\|n\| \cdot \|N\|^2)$
EXP$_G$	a, n	$(a \in G)$	$a^n \in G$	$2\|n\|$ G-operations

Figure 9.1: **Some basic algorithms and their running time.** Unless otherwise indicated, an input value is an integer and the running time is the number of bit operations. G denotes a group.

that $a = qN + r$ with $0 \le r < N$. We denote by INT-DIV an algorithm implementing this function. The algorithm uses the standard division method we learned way back in school, which turns out to run in time proportional to the product of the binary lengths of a and N.

We also want an algorithm that implements the mod function, taking integer inputs a, N with $N > 0$ and returning $a \bmod N$. This algorithm, denoted MOD, can be implemented simply by calling INT-DIV(a, N) to get (q, r), and then returning just the remainder r.

9.2.3 Extended GCD algorithm

Suppose a, b are integers, not both 0. A basic fact about the greatest common divisor of a and b is that it is the smallest positive element of the set

$$\{ a\overline{a} + b\overline{b} \ : \ \overline{a}, \overline{b} \in \mathbf{Z} \}$$

of all integer linear combinations of a and b. In particular, if $d = \gcd(a, b)$ then there exist integers $\overline{a}, \overline{b}$ such that $d = a\overline{a} + b\overline{b}$. (Note that either $\overline{a}$ or $\overline{b}$ could be negative.)

Example 9.2.1 The gcd of 20 and 12 is $d = \gcd(20, 12) = 4$. We note that $4 = 20(2) + (12)(-3)$, so in this case $\overline{a} = 2$ and $\overline{b} = -3$. ▮

Besides the gcd itself, we will find it useful to be able to compute these weights $\overline{a}, \overline{b}$. This is what the extended-gcd algorithm EXT-GCD does: given a, b as input, it returns $(d, \overline{a}, \overline{b})$ such that $d = \gcd(a, b) = a\overline{a} + b\overline{b}$. The algorithm itself is an extension of Euclid's classic algorithm for computing the gcd, and the simplest description is a recursive one. We now provide it, and then discuss the correctness and running time. The algorithm takes input any integers a, b, not both zero.

```
Algorithm EXT-GCD($a, b$)
If $b = 0$ then return $(a, 1, 0)$
Else
    $(q, r) \leftarrow$ INT-DIV$(a, b)$
    $(d, x, y) \leftarrow$ EXT-GCD$(b, r)$
    $\overline{a} \leftarrow y$
    $\overline{b} \leftarrow x - qy$
    Return $(d, \overline{a}, \overline{b})$
EndIf
```

The base case is when $b = 0$. If $b = 0$ then we know by assumption that $a \neq 0$, so $\gcd(a, b) = a$, and since $a = a(1) + b(0)$, the weights are 1 and 0. If $b \neq 0$ then we can divide by it, and we divide a by it to get a quotient q and remainder r. For the recursion, we use the fact that $\gcd(a, b) = \gcd(b, r)$. The recursive call thus yields $d = \gcd(a, b)$ together with weights x, y such that $d = bx + ry$. Noting that $a = bq + r$ we have

$$d \ = \ bx + ry \ = \ bx + (a - bq)y \ = \ ay + b(x - qy) \ = \ a\overline{a} + b\overline{b} \, ,$$

confirming that the values assigned to $\overline{a}, \overline{b}$ are correct.

The running time of this algorithm is $O(|a| \cdot |b|)$, or, put a little more simply, the running time is quadratic in the length of the longer number. This is not so obvious, and proving it takes some work. We do not provide this proof here.

We also want to know an upper bound on the lengths of the weights $\overline{a}, \overline{b}$ output by EXT-GCD(a, b). The running time bound tells us that $|\overline{a}|, |\overline{b}| = O(|a| \cdot |b|)$, but this is not good enough for some of what follows. I would expect that $|\overline{a}|, |\overline{b}| = O(|a| + |b|)$. Is this true? If so, can it be proved by induction based on the recursive algorithm above?

9.2.4 Algorithms for modular addition and multiplication

The next two algorithms in Fig. 9.1 are the ones for modular addition and multiplication. To compute $(a + b) \bmod N$, we first compute $c = a + b$ using the usual algorithm we learned way back in school, which runs in time linear in the binary representations of the numbers. We might imagine that it now takes quadratic time to do the mod operation, but in fact if $c > N$, the mod operation can be simply executed by subtracting N from c, which takes only linear time, which is why the algorithm as a whole takes linear time. For multiplication mod N, the process is much the same. First compute $c = ab$ using the usual algorithm, which is quadratic time. This time we do the mod by invoking MOD(c, N). (The length of c is the sum of the lengths of a and b, and so c is not small as in the addition case, so a shortcut to the mod as we saw there does not seem possible.)

9.2.5 Algorithm for modular inverse

The next algorithm in Fig. 9.1 is for computation of the multiplicative inverse of a in the group $\mathbf{Z}_N^*$. Namely, on input $N > 0$ and $a \in \mathbf{Z}_N^*$, algorithm MOD-INV returns b such that $ab \equiv 1 \pmod{N}$. The method is quite simple:

```
Algorithm MOD-INV$(a, N)$
$(d, \overline{a}, \overline{N}) \leftarrow$ EXT-GCD$(a, N)$
$b \leftarrow \overline{a} \bmod N$
Return $b$
```

Correctness is easy to see. Since $a \in \mathbf{Z}_N^*$ we know that $\gcd(a, N) = 1$. The EXT-GCD algorithm thus guarantees that $d = 1$ and $1 = a\overline{a} + N\overline{N}$. Since $N \bmod N = 0$, we have $1 \equiv a\overline{a} \pmod{N}$, and thus $b = \overline{a} \bmod N$ is the right value to return.

The cost of the first step is $O(|a| \cdot |N|)$. The cost of the second step is $O(|\overline{a}| \cdot |N|)$. If we assume that $|\overline{a}| = O(|a| + |N|)$ then the overall cost is $O(|a| \cdot |N|)$. See discussion of the EXT-GCD algorithm regarding this assumption on the length of $\overline{a}$.

9.2.6 Exponentiation algorithm

We will be using exponentiation in various different groups, so it is useful to look at it at the group level. Let G be a group and let $a \in G$. Given an integer $n \in \mathbf{Z}$ we want to compute the group element a^n as defined in Section 9.1.2. The naive method, assuming for simplicity $n \geq 0$, is to execute

```
$y \leftarrow \mathbf{1}$
For $i = 1, \ldots, n$ do $y \leftarrow y \cdot a$ EndFor
Return $y$
```

This might at first seem like a satisfactory algorithm, but actually it is very slow. The number of group operations required is n, and the latter can be as large as the order of the group. Since we are often looking at groups containing about 2^{512} elements, exponentiation by this method is not feasible. In the language of complexity theory, the problem is that we are looking at an exponential time algorithm. This is because the running time is exponential in the binary length $|n|$ of the input n. So we seek a better algorithm. We illustrate the idea of fast exponentiation with an example.

Example 9.2.2 Suppose the binary length of n is 5, meaning the binary representation of n has the form $b_4b_3b_2b_1b_0$. Then

$$\begin{aligned} n &= 2^4b_4 + 2^3b_3 + 2^2b_2 + 2^1b_1 + 2^0b_0 \\ &= 16b_4 + 8b_3 + 4b_2 + 2b_1 + b_0 \ . \end{aligned}$$

Our exponentiation algorithm will proceed to compute the values $y_5, y_4, y_3, y_2, y_1, y_0$ in turn, as follows:

$$\begin{aligned} y_5 &= \mathbf{1} \\ y_4 &= y_5^2 \cdot a^{b_4} &&= a^{b_4} \\ y_3 &= y_4^2 \cdot a^{b_3} &&= a^{2b_4+b_3} \\ y_2 &= y_3^2 \cdot a^{b_2} &&= a^{4b_4+2b_3+b_2} \\ y_1 &= y_2^2 \cdot a^{b_1} &&= a^{8b_4+4b_3+2b_2+b_1} \\ y_0 &= y_1^2 \cdot a^{b_0} &&= a^{16b_4+8b_3+4b_2+2b_1+b_0} \ . \end{aligned}$$

Two group operations are required to compute y_i from y_{i+1}, and the number of steps equals the binary length of n, so the algorithm is fast. ▌

In general, we let $b_{k-1} \ldots b_1b_0$ be the binary representation of n, meaning $b_0, \ldots, b_{k-1}$ are bits such that $n = 2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \cdots + 2^1b_1 + 2^0b_0$. The algorithm proceeds as follows given any input $a \in G$ and $n \in \mathbf{Z}$:

```
Algorithm $\text{EXP}_G(a, n)$
If $n < 0$ then $a \leftarrow a^{-1}$ and $n \leftarrow -n$ EndIf
Let $b_{k-1} \ldots b_1b_0$ be the binary representation of $n$
$y \leftarrow \mathbf{1}$
For $i = k - 1$ downto 0 do
    $y \leftarrow y^2 \cdot a^{b_i}$
End For
Output $y$
```

The algorithm uses two group operations per iteration of the loop: one to multiply y by itself, another to multiply the result by a^{b_i}. (The computation of a^{b_i} is without cost, since this is just a if $b_i = 1$ and $\mathbf{1}$ if $b_i = 0$.) So its total cost is $2k = 2|n|$ group operations. (We are ignoring the cost of the one possible inversion in the case $n < 0$.) (This is the worst case cost. We observe that it actually takes $|n| + W_H(n)$ group operations, where $W_H(n)$ is the number of ones in the binary representation of n.)

We will typically use this algorithm when the group G is $\mathbf{Z}_N^*$ and the group operation is multiplication modulo N, for some positive integer N. We have denoted this algorithm by MOD-EXP in Fig. 9.1. (The input a is not required to be relatively prime to N even though it usually will be, so is listed as coming from $\mathbf{Z}_N$.) In that case, each group operation is implemented via MOD-MULT and takes $O(|N|^2)$ time, so the running time of the algorithm is $O(|n| \cdot |N|^2)$. Since n is usually in $\mathbf{Z}_N$, this comes to $O(|N|^3)$. The salient fact to remember is that modular exponentiation is a cubic time algorithm.

9.3 Cyclic groups and generators

Let G be a group, let $\mathbf{1}$ denote its identity element, and let $m = |G|$ be the order of G. If $g \in G$ is any member of the group, the *order* of g is defined to be the least positive integer n such that $g^n = \mathbf{1}$. We let

$$\langle g \rangle \;=\; \{\, g^i \;:\; i \in \mathbf{Z}_n \,\} \;=\; \{g^0, g^1, \ldots, g^{n-1}\}$$

denote the set of group elements generated by g. A fact we do not prove, but is easy to verify, is that this set is a subgroup of G. The order of this subgroup (which, by definition, is its size) is just the order of g. Fact 9.1.6 tells us that the order n of g divides the order m of the group. An element g of the group is called a *generator* of G if $\langle g \rangle = G$, or, equivalently, if its order is m. If g is a generator of G then for every $a \in G$ there is a unique integer $i \in \mathbf{Z}_m$ such that $g^i = a$. This i is called the discrete logarithm of a to base g, and we denote it by $\mathrm{DLog}_{G,g}(a)$. Thus, $\mathrm{DLog}_{G,g}(\cdot)$ is a function that maps G to $\mathbf{Z}_m$, and moreover this function is a bijection, meaning one-to-one and onto. The function of $\mathbf{Z}_m$ to G defined by $i \mapsto g^i$ is called the discrete exponentiation function, and the discrete logarithm function is the inverse of the discrete exponentiation function.

Example 9.3.1 Let $p = 11$, which is prime. Then $Z_{11}^* = \{1,2,3,4,5,6,7,8,9,10\}$ has order $p-1 = 10$. Let us find the subgroups generated by group elements 2 and 5. We raise them to the powers $i = 0,\ldots,9$. We get:

i	0	1	2	3	4	5	6	7	8	9
2^i mod 11	1	2	4	8	5	10	9	7	3	6
5^i mod 11	1	5	3	4	9	1	5	3	4	9

Looking at which elements appear in the row corresponding to 2 and 5, respectively, we can determine the subgroups these group elements generate:

$$\begin{aligned} \langle 2 \rangle &= \{1,2,3,4,5,6,7,8,9,10\} \\ \langle 5 \rangle &= \{1,3,4,5,9\}\ . \end{aligned}$$

Since $\langle 2 \rangle$ equals $\mathbf{Z}_{11}^*$, the element 2 is a generator. Since a generator exists, $\mathbf{Z}_{11}^*$ is cyclic. On the other hand, $\langle 5 \rangle \neq \mathbf{Z}_{11}^*$, so 5 is not a generator. The order of 2 is 10, while the order of 5 is 5. Note that these orders divide the order 10 of the group. The table also enables us to determine the discrete logarithms to base 2 of the different group elements:

a	1	2	3	4	5	6	7	8	9	10
$\mathrm{DLog}_{\mathbf{Z}_{11}^*,2}(a)$	0	1	8	2	4	9	7	3	6	5

Later we will see a way of identifying all the generators given that we know one of them. ▌

The discrete exponentiation function is conjectured to be one-way (meaning the discrete logarithm function is hard to compute) for some cyclic groups G. Due to this fact we often seek cyclic groups for cryptographic usage. Here are three sources of such groups. We will not prove any of the facts below; their proofs can be found in books on algebra.

Fact 9.3.2 Let p be a prime. Then the group $\mathbf{Z}_p^*$ is cyclic. ▌

The operation here is multiplication modulo p, and the size of this group is $\varphi(p) = p-1$. This is the most common choice of group in cryptography.

Fact 9.3.3 Let G be a group and let $m = |G|$ be its order. If m is a prime number, then G is cyclic. ▌

In other words, any group having a prime number of elements is cyclic. Note that it is not for this reason that Fact 9.3.2 is true, since the order of $\mathbf{Z}_p^*$ (where p is prime) is $p-1$, which is even if $p \geq 3$ and 1 if $p = 2$, and is thus never a prime number.

The following is worth knowing if you have some acquaintance with finite fields. Recall that a field is a set F equipped with two operations, an addition and a multiplication. The identity element of the addition is denoted 0. When this is removed from the field, what remains is a group under multiplication. This group is always cyclic.

Fact 9.3.4 Let F be a finite field, and let $F^* = F - \{0\}$. Then F^* is a cyclic group under the multiplication operation of F. ▌

A finite field of order m exists if and only if $m = p^n$ for some prime p and integer $n \geq 1$. The finite field of order p is exactly $\mathbf{Z}_p$, so the case $n = 1$ of Fact 9.3.4 implies Fact 9.3.2. Another interesting special case of Fact 9.3.4 is when the order of the field is 2^n, meaning $p = 2$, yielding a cyclic group of order $2^n - 1$.

When we want to use a cyclic group G in cryptography, we will often want to find a generator for it. The process used is to pick group elements in some appropriate way, and then test each chosen element to see whether it is a generator. One thus has to solve two problems. One is how to test whether a given group element is a generator, and the other is what process to use to choose the candidate generators to be tested.

Let $m = |G|$ and let $\mathbf{1}$ be the identity element of G. The obvious way to test whether a given $g \in G$ is a generator is to compute the values $g^1, g^2, g^3, \ldots$, stopping at the first j such that $g^j = \mathbf{1}$. If $j = m$ then g is a generator. This test however can require up to m group operations, which is not efficient, given that the groups of interest are large, so we need better tests.

The obvious way to choose candidate generators is to cycle through the entire group in some way, testing each element in turn. Even with a fast test, this can take a long time, since the group is large. So we would also like better ways of picking candidates.

We address these problems in turn. Let us first look at testing whether a given $g \in G$ is a generator. One sees quickly that computing all powers of g as in $g^1, g^2, g^3, \ldots$ is not necessary. For example if we computed g^8 and found that this is not $\mathbf{1}$, then we know that $g^4 \neq \mathbf{1}$ and $g^2 \neq \mathbf{1}$ and $g \neq \mathbf{1}$. More generally, if we know that $g^j \neq \mathbf{1}$ then we know that $g^i \neq \mathbf{1}$ for all i dividing j. This tells us that it is better to first compute high powers of g, and use that to cut down the space of exponents that need further testing. The following Proposition pinpoints the optimal way to do this. It identifies a set of exponents $m_1, \ldots, m_n$ such that one need only test whether $g^{m_i} \neq \mathbf{1}$ for $i = 1, \ldots, n$. As we will argue later, this set is quite small.

Proposition 9.3.5 Let G be a cyclic group and let $m = |G|$ be the size of G. Let $p_1^{\alpha_1} \cdots p_n^{\alpha_n}$ be the prime factorization of m and let $m_i = m/p_i$ for $i = 1, \ldots, n$. Let $g \in G$. Then g is a generator of G if and only if

$$\text{For all } i = 1, \ldots, n: \quad g^{m_i} \neq \mathbf{1}\ , \tag{9.1}$$

where $\mathbf{1}$ is the identity element of G. ▌

Proof of Proposition 9.3.5: First suppose that g is a generator of G. Then we know that the smallest positive integer j such that $g^j = \mathbf{1}$ is $j = m$. Since $0 < m_i < m$, it must be that $g^{m_i} \neq \mathbf{1}$ for all $i = 1, \ldots, m$.

Conversely, suppose g satisfies the condition of Equation (9.1). We want to show that g is a generator. Let j be the order of g, meaning the smallest positive integer such that $g^j = \mathbf{1}$. Then we know that j must divide the order m of the group, meaning $m = dj$ for some integer $d \geq 1$. This implies that $j = p_1^{\beta_1} \cdots p_n^{\beta_n}$ for some integers $\beta_1, \ldots, \beta_n$ satisfying $0 \leq \beta_i \leq \alpha_i$ for all $i = 1, \ldots, n$. If $j < m$ then there must be some i such that $\beta_i < \alpha_i$, and in that case j divides m_i, which in turn implies $g^{m_i} = \mathbf{1}$ (because $g^j = \mathbf{1}$). So the assumption that Equation (9.1) is true implies that j cannot be strictly less than m, so the only possibility is $j = m$, meaning g is a generator. ▮

The number n of terms in the prime factorization of m cannot be more than $\lg(m)$, the binary logarithm of m. (This is because $p_i \geq 2$ and $\alpha_i \geq 1$ for all $i = 1, \ldots, n$.) So, for example, if the group has size about 2^{512}, then at most 512 tests are needed. So testing is quite efficient. One should note however that it requires knowing the prime factorization of m.

Let us now consider the second problem we discussed above, namely how to choose candidate group elements for testing. There seems little reason to think that trying all group elements in turn will yield a generator in a reasonable amount of time. Instead, we consider picking group elements at random, and then testing them. The probability of success in any trial is $|\mathsf{Gen}(G)|/|G|$. So the expected number of trials before we find a generator is $|G|/|\mathsf{Gen}(G)|$. To estimate the efficacy of this method, we thus need to know the number of generators in the group. The following Proposition gives a characterization of the generator set which in turn tells us its size.

Proposition 9.3.6 Let G be a cyclic group of order m, and let g be a generator of G. Then $\mathsf{Gen}(G) = \{\, g^i \in G \,:\, i \in \mathbf{Z}_m^* \,\}$ and $|\mathsf{Gen}(G)| = \varphi(m)$. ▮

That is, having fixed one generator g, a group element h is a generator if and only if its discrete logarithm to base g is relatively prime to the order m of the group. As a consequence, the number of generators is the number of integers in the range $1, \ldots, m-1$ that are relatively prime to m.

Proof of Proposition 9.3.6: Given that $\mathsf{Gen}(G) = \{\, g^i \in G \,:\, i \in \mathbf{Z}_m^* \,\}$, the claim about its size follows easily:

$$|\mathsf{Gen}(G)| \;=\; \left|\{\, g^i \in G \,:\, i \in \mathbf{Z}_m^* \,\}\right| \;=\; |\mathbf{Z}_m^*| \;=\; \varphi(m) \;.$$

We now prove that $\mathsf{Gen}(G) = \{\, g^i \in G \,:\, i \in \mathbf{Z}_m^* \,\}$. First, we show that if $i \in \mathbf{Z}_m^*$ then $g^i \in \mathsf{Gen}(G)$. Second, we show that if $i \in \mathbf{Z}_m - \mathbf{Z}_m^*$ then $g^i \notin \mathsf{Gen}(G)$.

So first suppose $i \in \mathbf{Z}_m^*$, and let $h = g^i$. We want to show that h is a generator of G. It suffices to show that the only possible value of $j \in \mathbf{Z}_m$ such that $h^j = \mathbf{1}$ is $j = 0$, so let us now show this. Let $j \in \mathbf{Z}_m$ be such that $h^j = \mathbf{1}$. Since $h = g^i$ we have

$$\mathbf{1} \;=\; h^j \;=\; g^{ij \bmod m} \;.$$

Since g is a generator, it must be that $ij \equiv 0 \pmod{m}$, meaning m divides ij. But $i \in \mathbf{Z}_m^*$ so $\gcd(i, m) = 1$. So it must be that m divides j. But $j \in \mathbf{Z}_m$ and the only member of this set divisible by m is 0, so $j = 0$ as desired.

Next, suppose $i \in \mathbf{Z}_m - \mathbf{Z}_m^*$ and let $h = g^i$. To show that h is not a generator it suffices to show that there is some non-zero $j \in \mathbf{Z}_m$ such that $h^j = \mathbf{1}$. Let $d = \gcd(i, m)$. Our assumption $i \in \mathbf{Z}_m - \mathbf{Z}_m^*$ implies that $d > 1$. Let $j = m/d$, which is a non-zero integer in $\mathbf{Z}_m$ because $d > 1$. Then the following shows that $h^j = \mathbf{1}$, completing the proof:

$$h^j \;=\; g^{ij} \;=\; g^{i \cdot m/d} \;=\; g^{m \cdot i/d} \;=\; (g^m)^{i/d} \;=\; \mathbf{1}^{i/d} \;=\; \mathbf{1}.$$

We used here the fact that d divides i and that $g^m = \mathbf{1}$. ▮

Example 9.3.7 Let us determine all the generators of the group $\mathbf{Z}_{11}^*$. Let us first use Proposition 9.3.5. The size of $\mathbf{Z}_{11}^*$ is $m = \varphi(11) = 10$, and the prime factorization of 10 is $2^1 \cdot 5^1$. Thus, the test for whether a given $a \in \mathbf{Z}_{11}^*$ is a generator is that $a^2 \not\equiv 1 \pmod{11}$ and $a^5 \not\equiv 1 \pmod{11}$. Let us compute a^2 mod 11 and a^5 mod 11 for all group elements a. We get:

a	1	2	3	4	5	6	7	8	9	10
a^2 mod 11	1	4	9	5	3	3	5	9	4	1
a^5 mod 11	1	10	1	1	1	10	10	10	1	10

The generators are those a for which the corresponding column has no entry equal to 1, meaning in both rows, the entry for this column is different from 1. So

$$\mathsf{Gen}(\mathbf{Z}_{11}^*) \;=\; \{2, 6, 7, 8\}\ .$$

Now, let us use Proposition 9.3.6 and double-check that we get the same thing. We saw in Example 9.3.1 that 2 was a generator of $\mathbf{Z}_{11}^*$. As per Proposition 9.3.6, the set of generators is

$$\mathsf{Gen}(\mathbf{Z}_{11}^*) \;=\; \{\, 2^i \bmod 11 \;:\; i \in \mathbf{Z}_{10}^* \,\}\ .$$

This is because the size of the group is $m = 10$. Now, $\mathbf{Z}_{10}^* = \{1, 3, 7, 9\}$. The values of 2^i mod 11 as i ranges over this set can be obtained from the table in Example 9.3.1 where we computed all the powers of 2. So

$$\begin{aligned} \{\, 2^i \bmod 11 \;:\; i \in \mathbf{Z}_{10}^* \,\} \;&=\; \{2^1 \bmod 11, 2^3 \bmod 11, 2^7 \bmod 11, 2^9 \bmod 11\} \\ &=\; \{2, 6, 7, 8\}\ . \end{aligned}$$

This is the same set we obtained above via Proposition 9.3.5. If we try to find a generator by picking group elements at random and then testing using Proposition 9.3.5, each trial has probability of success $\varphi(10)/10 = 4/10$, so we would expect to find a generator in 10/4 trials. We can optimize slightly by noting that 1 and -1 can never be generators, and thus we only need pick candidates randomly from $\mathbf{Z}_{11}^* - \{1, 10\}$. In that case, each trial has probability of success $\varphi(10)/8 = 4/8 = 1/2$, so we would expect to find a generator in 2 trials. ▌

When we want to work in a cyclic group in cryptography, the most common choice is to work over $\mathbf{Z}_p^*$ for a suitable prime p. The algorithm for finding a generator would be to repeat the process of picking a random group element and testing it, halting when a generator is found. In order to make this possible we choose p in such a way that the prime factorization of the order $p - 1$ of $\mathbf{Z}_p^*$ is known. In order to make the testing fast, we choose p so that $p - 1$ has few prime factors. Accordingly, it is common to choose p to equal $2q + 1$ for some prime q. In this case, the prime factorization of $p - 1$ is $2^1 q^1$, so we need raise a candidate to only two powers to test whether or not it is a generator. In choosing candidates, we optimize slightly by noting that 1 and -1 are never generators, and accordingly pick the candidates from $\mathbf{Z}_p^* - \{1, p - 1\}$ rather than from $\mathbf{Z}_p^*$. So the algorithm is as follows:

```
Algorithm FIND-GEN($p$)
$q \leftarrow (p - 1)/2$
$\mathsf{found} \leftarrow 0$
While ($\mathsf{found} \neq 1$) do
    $g \xleftarrow{\$} \mathbf{Z}_p^* - \{1, p - 1\}$
    If ($g^2 \bmod p \neq 1$) and ($g^q \bmod p \neq 1$) then $\mathsf{found} \leftarrow 1$
```

```
EndWhile
Return $g$
```

Proposition 9.3.5 tells us that the group element g returned by this algorithm is always a generator of $\mathbf{Z}_p^*$. By Proposition 9.3.6, the probability that an iteration of the algorithm is successful in finding a generator is

$$\frac{|\mathsf{Gen}(\mathbf{Z}_p^*)|}{|\mathbf{Z}_p^*|-2} = \frac{\varphi(p-1)}{p-3} = \frac{\varphi(2q)}{2q-2} = \frac{q-1}{2q-2} = \frac{1}{2} .$$

Thus the expected number of iterations of the while loop is 2. Above, we used that fact that $\varphi(2q) = q-1$ which is true because q is prime.

9.4 Squares and non-squares

An element a of a group G is called a *square*, or *quadratic residue* if it has a square root, meaning there is some $b \in G$ such that $b^2 = a$ in G. We let

$$\mathsf{QR}(G) = \{ g \in G : g \text{ is quadratic residue in } G \}$$

denote the set of all squares in the group G. We leave to the reader to check that this set is a subgroup of G.

We are mostly interested in the case where the group G is $\mathbf{Z}_N^*$ for some integer N. An integer a is called a *square mod N* or *quadratic residue mod N* if a mod N is a member of $\mathsf{QR}(\mathbf{Z}_N^*)$. If $b^2 \equiv a \pmod{N}$ then b is called a square-root of a mod N. An integer a is called a *non-square mod N* or *quadratic non-residue mod N* if a mod N is a member of $\mathbf{Z}_N^* - \mathsf{QR}(\mathbf{Z}_N^*)$. We will begin by looking at the case where $N = p$ is a prime. In this case we define a function J_p: $\mathbf{Z} \to \{-1, 1\}$ by

$$J_p(a) = \begin{cases} 1 & \text{if } a \text{ is a square mod } p \\ 0 & \text{if } a \bmod p = 0 \\ -1 & \text{otherwise.} \end{cases}$$

for all $a \in \mathbf{Z}$. We call $J_p(a)$ the *Legendre symbol* of a. Thus, the Legendre symbol is simply a compact notation for telling us whether or not its argument is a square modulo p.

Before we move to developing the theory, it may be useful to look at an example.

Example 9.4.1 Let $p = 11$, which is prime. Then $\mathbf{Z}_{11}^* = \{1,2,3,4,5,6,7,8,9,10\}$ has order $p-1 = 10$. A simple way to determine $\mathsf{QR}(\mathbf{Z}_{11}^*)$ is to square all the group elements in turn:

a	1	2	3	4	5	6	7	8	9	10
a^2 mod 11	1	4	9	5	3	3	5	9	4	1

The squares are exactly those elements that appear in the second row, so

$$\mathsf{QR}(\mathbf{Z}_{11}^*) = \{1,3,4,5,9\} .$$

The number of squares is 5, which we notice equals $(p-1)/2$. This is not a coincidence, as we will see. Also notice that each square has exactly two different square roots. (The square roots of 1 are 1 and 10; the square roots of 3 are 5 and 6; the square roots of 4 are 2 and 9; the square roots of 5 are 4 and 7; the square roots of 9 are 3 and 8.)

Since 11 is prime, we know that $\mathbf{Z}_{11}^*$ is cyclic, and as we saw in Example 9.3.1, 2 is a generator. (As a side remark, we note that a generator must be a non-square. Indeed, if $a = b^2$ is a square, then $a^5 = b^{10} = 1$ modulo 11 because 10 is the order of the group. So $a^j = 1$ modulo 11 for some positive $j < 10$, which means a is not a generator. However, not all non-squares need be generators.) Below, we reproduce from that example the table of discrete logarithms of the group elements. We also add below it a row providing the Legendre symbols, which we know because, above, we identified the squares. We get:

a	1	2	3	4	5	6	7	8	9	10
$\mathrm{DLog}_{\mathbf{Z}_{11}^*,2}(a)$	0	1	8	2	4	9	7	3	6	5
$J_{11}(a)$	1	-1	1	1	1	-1	-1	-1	1	-1

We observe that the Legendre symbol of a is 1 if its discrete logarithm is even, and -1 if the discrete logarithm is odd, meaning the squares are exactly those group elements whose discrete logarithm is even. It turns out that this fact is true regardless of the choice of generator. ▌

As we saw in the above example, the fact that $\mathbf{Z}_p^*$ is cyclic is useful in understanding the structure of the subgroup of quadratic residues $\mathsf{QR}(\mathbf{Z}_p^*)$. The following Proposition summarizes some important elements of this connection.

Proposition 9.4.2 Let $p \geq 3$ be a prime and let g be a generator of $\mathbf{Z}_p^*$. Then

$$\mathsf{QR}(\mathbf{Z}_p^*) \;=\; \{\, g^i \;:\; i \in \mathbf{Z}_{p-1} \text{ and } i \text{ is even} \,\}\;, \tag{9.2}$$

and the number of squares mod p is

$$\left|\mathsf{QR}(\mathbf{Z}_p^*)\right| \;=\; \frac{p-1}{2}\;.$$

Furthermore, every square mod p has exactly two different square roots mod p. ▌

Proof of Proposition 9.4.2: Let

$$E \;=\; \{\, g^i \;:\; i \in \mathbf{Z}_{p-1} \text{ and } i \text{ is even} \,\}\;.$$

We will prove that $E = \mathsf{QR}(\mathbf{Z}_p^*)$ by showing first that $E \subseteq \mathsf{QR}(\mathbf{Z}_p^*)$ and second that $\mathsf{QR}(\mathbf{Z}_p^*) \subseteq E$.

To show that $E \subseteq \mathsf{QR}(\mathbf{Z}_p^*)$, let $a \in E$. We will show that $a \in \mathsf{QR}(\mathbf{Z}_p^*)$. Let $i = \mathrm{DLog}_{\mathbf{Z}_p^*,g}(a)$. Since $a \in E$ we know that i is even. Let $j = i/2$ and note that $j \in \mathbf{Z}_{p-1}$. Clearly

$$(g^j)^2 \equiv g^{2j \bmod p-1} \equiv g^{2j} \equiv g^i \pmod p\;,$$

so g^j is a square root of $a = g^i$. So a is a square.

To show that $\mathsf{QR}(\mathbf{Z}_p^*) \subseteq E$, let b be any element of $\mathbf{Z}_p^*$. We will show that $b^2 \in E$. Let $j = \mathrm{DLog}_{\mathbf{Z}_p^*,g}(b)$. Then

$$b^2 \equiv (g^j)^2 \equiv g^{2j \bmod p-1} \equiv g^{2j} \pmod p\;,$$

the last equivalence being true because the order of the group $\mathbf{Z}_p^*$ is $p-1$. This shows that $b^2 \in E$.

The number of even integers in $\mathbf{Z}_{p-1}$ is exactly $(p-1)/2$ since $p-1$ is even. The claim about the size of $\mathsf{QR}(\mathbf{Z}_p^*)$ thus follows from Equation (9.2). It remains to justify the claim that every square mod p has exactly two square roots mod p. This can be seen by a counting argument, as follows.

Suppose a is a square mod p. Let $i = \mathrm{DLog}_{\mathbf{Z}_p^*,g}(a)$. We know from the above that i is even. Let $x = i/2$ and let $y = x + (p-1)/2 \bmod (p-1)$. Then g^x is a square root of a. Furthermore

$$(g^y)^2 \equiv g^{2y} \equiv g^{2x+(p-1)} \equiv g^{2x}g^{p-1} \equiv a \cdot 1 \equiv a \pmod p\;,$$

so g^y is also a square root of a. Since i is an even number in $\mathbf{Z}_{p-1}$ and $p-1$ is even, it must be that $0 \le x < (p-1)/2$. It follows that $(p-1)/2 \le y < p-1$. Thus $x \ne y$. This means that a has as least two square roots. This is true for each of the $(p-1)/2$ squares mod p. So the only possibility is that each of these squares has exactly two square roots. ▌

Suppose we are interested in knowing whether or not a given $a \in \mathbf{Z}_p^*$ is a square mod p, meaning we want to know the value of the Legendre symbol $J_p(a)$. Proposition 9.4.2 tells us that

$$J_p(a) \;=\; (-1)^{\mathrm{DLog}_{\mathbf{Z}_p^*,g}(a)} \,,$$

where g is any generator of $\mathbf{Z}_p^*$. This however is not very useful in computing $J_p(a)$, because it requires knowing the discrete logarithm of a, which is hard to compute. The following Proposition says that the Legendre symbols of a modulo an odd prime p can be obtained by raising a to the power $(p-1)/2$, and helps us compute the Legendre symbol.

Proposition 9.4.3 Let $p \ge 3$ be a prime. Then

$$J_p(a) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

for any $a \in \mathbf{Z}_p^*$. ▌

Now one can determine whether or not a is a square mod p by running the algorithm MOD-EXP on inputs $a, (p-1)/2, p$. If the algorithm returns 1 then a is a square mod p, and if it returns $p-1$ (which is the same as -1 mod p) then a is a non-square mod p. Thus, the Legendre symbol can be computed in time cubic in the length of p.

Towards the proof of Proposition 9.4.3, we begin with the following lemma which is often useful in its own right.

Lemma 9.4.4 Let $p \ge 3$ be a prime. Then

$$g^{\frac{p-1}{2}} \equiv -1 \pmod{p}$$

for any generator g of $\mathbf{Z}_p^*$.

Proof of Lemma 9.4.4: We begin by observing that 1 and -1 are both square roots of 1 mod p, and are distinct. (It is clear that squaring either of these yields 1, so they are square roots of 1. They are distinct because -1 equals $p-1$ mod p, and $p-1 \ne 1$ because $p \ge 3$.) By Proposition 9.4.2, these are the only square roots of 1. Now let

$$b \;=\; g^{\frac{p-1}{2}} \bmod p \,.$$

Then $b^2 \equiv 1 \pmod{p}$, so b is a square root of 1. By the above b can only be 1 or -1. However, since g is a generator, b cannot be 1. (The smallest positive value of i such that g^i is 1 mod p is $i = p-1$.) So the only choice is that $b \equiv -1 \pmod{p}$, as claimed. ▌

Proof of Proposition 9.4.3: By definition of the Legendre symbol, we need to show that

$$a^{\frac{p-1}{2}} \equiv \begin{cases} \;\;1 \pmod{p} & \text{if } a \text{ is a square mod } p \\ -1 \pmod{p} & \text{otherwise.} \end{cases}$$

Let g be a generator of $\mathbf{Z}_p^*$ and let $i = \mathrm{DLog}_{\mathbf{Z}_p^*,g}(a)$. We consider separately the cases of a being a square and a being a non-square.

Suppose a is a square mod p. Then Proposition 9.4.2 tells us that i is even. In that case

$$a^{\frac{p-1}{2}} \equiv (g^i)^{\frac{p-1}{2}} \equiv g^{i\cdot\frac{p-1}{2}} \equiv (g^{p-1})^{i/2} \equiv 1 \pmod{p} ,$$

as desired.

Now suppose a is a non-square mod p. Then Proposition 9.4.2 tells us that i is odd. In that case

$$a^{\frac{p-1}{2}} \equiv (g^i)^{\frac{p-1}{2}} \equiv g^{i\cdot\frac{p-1}{2}} \equiv g^{(i-1)\cdot\frac{p-1}{2}+\frac{p-1}{2}} \equiv (g^{p-1})^{(i-1)/2} \cdot g^{\frac{p-1}{2}} \equiv g^{\frac{p-1}{2}} \pmod{p} .$$

However Lemma 9.4.4 tells us that the last quantity is -1 modulo p, as desired. ▌

The following Proposition says that $ab \bmod p$ is a square if and only if either both a and b are squares, or if both are non-squares. But if one is a square and the other is not, then $ab \bmod p$ is a non-square. This can be proved by using either Proposition 9.4.2 or Proposition 9.4.3. We use the latter in the proof. You might try, as an exercise, to reprove the result using Proposition 9.4.2 instead.

Proposition 9.4.5 Let $p \geq 3$ be prime. Then

$$J_p(ab \bmod p) = J_p(a) \cdot J_p(b)$$

for all $a, b \in \mathbf{Z}_p^*$. ▌

Proof of Proposition 9.4.5: Using Proposition 9.4.3 we get

$$J_p(ab \bmod p) \equiv (ab)^{\frac{p-1}{2}} \equiv a^{\frac{p-1}{2}} b^{\frac{p-1}{2}} \equiv J_p(a) \cdot J_p(b) \pmod{p} .$$

The two quantities we are considering both being either 1 or -1, and equal modulo p, must then be actually equal. ▌

A quantity of cryptographic interest is the Diffie-Hellman (DH) key. Having fixed a cyclic group G and generator g for it, the DH key associated to elements $X = g^x$ and $Y = g^y$ of the group is the group element g^{xy}. The following Proposition tells us that the DH key is a square if either X *or* Y is a square, and otherwise is a non-square.

Proposition 9.4.6 Let $p \geq 3$ be a prime and let g be a generator of $\mathbf{Z}_p^*$. Then

$$J_p(g^{xy} \bmod p) = 1 \quad \text{if and only if} \quad J_p(g^x \bmod p) = 1 \text{ or } J_p(g^y \bmod p) = 1 ,$$

for all $x, y \in \mathbf{Z}_{p-1}$. ▌

Proof of Proposition 9.4.6: By Proposition 9.4.2, it suffices to show that

$$xy \bmod (p-1) \text{ is even} \quad \text{if and only if} \quad x \text{ is even or } y \text{ is even} .$$

But since $p-1$ is even, $xy \bmod (p-1)$ is even exactly when xy is even, and clearly xy is even exactly if either x or y is even. ▌

With a cyclic group G and generator g of G fixed, we will be interested in the distribution of the DH key g^{xy} in G, under random choices of x, y from $\mathbf{Z}_m$, where $m = |G|$. One might at first think that in this case the DH key is a random group element. The following proposition tells us that in the group $\mathbf{Z}_p^*$ of integers modulo a prime, this is certainly not true. The DH key is significantly more likely to be a square than a non-square, and in particular is thus not even almost uniformly distributed over the group.

Proposition 9.4.7 Let $p \geq 3$ be a prime and let g be a generator of $\mathbf{Z}_p^*$. Then

$$\Pr\left[x \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \,;\, y \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \;:\; J_p(g^{xy}) = 1 \right]$$

equals 3/4. ▌

Proof of Proposition 9.4.7: By Proposition 9.4.7 we need only show that

$$\Pr\left[x \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \,;\, y \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \;:\; J_p(g^{x}) = 1 \text{ or } J_p(g^{y}) = 1 \right]$$

equals 3/4. The probability in question is $1 - \alpha$ where

$$\begin{aligned}
\alpha &= \Pr\left[x \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \,;\, y \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \;:\; J_p(g^{x}) = -1 \text{ and } J_p(g^{y}) = -1 \right] \\
&= \Pr\left[x \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \;:\; J_p(g^{x}) = -1 \right] \cdot \Pr\left[y \stackrel{\$}{\leftarrow} \mathbf{Z}_{p-1} \;:\; J_p(g^{y}) = -1 \right] \\
&= \frac{|\mathsf{QR}(\mathbf{Z}_p^*)|}{|\mathbf{Z}_p^*|} \cdot \frac{|\mathsf{QR}(\mathbf{Z}_p^*)|}{|\mathbf{Z}_p^*|} \\
&= \frac{(p-1)/2}{p-1} \cdot \frac{(p-1)/2}{p-1} \\
&= \frac{1}{2} \cdot \frac{1}{2} \\
&= \frac{1}{4}\,.
\end{aligned}$$

Thus $1-\alpha = 3/4$ as desired. Here we used Proposition 9.4.2 which told us that $|\mathsf{QR}(\mathbf{Z}_p^*)| = (p-1)/2$. ▌

The above Propositions, combined with Proposition 9.4.3 (which tells us that quadratic residuosity modulo a prime can be efficiently tested), will later lead us to pinpoint weaknesses in certain cryptographic schemes in $\mathbf{Z}_p^*$.

9.5 Groups of prime order

A group of prime order is a group G whose order $m = |G|$ is a prime number. Such a group is always cyclic. These groups turn out to be quite useful in cryptography, so let us take a brief look at them and some of their properties.

An element h of a group G is called *non-trivial* if it is not equal to the identity element of the group.

Proposition 9.5.1 Suppose G is a group of order q where q is a prime, and h is any non-trivial member of G. Then h is a generator of G. ▌

Proof of Proposition 9.5.1: It suffices to show that the order of h is q. We know that the order of any group element must divide the order of the group. Since the group has prime order q, the only possible values for the order of h are 1 and q. But h does not have order 1 since it is non-trivial, so it must have order q. ▌

A common way to obtain a group of prime order for cryptographic schemes is as a subgroup of a group of integers modulo a prime. We pick a prime p having the property that $q = (p-1)/2$ is also prime. It turns out that the subgroup of quadratic residues modulo p then has order q, and hence is a group of prime order. The following proposition summarizes the facts for future reference.

Proposition 9.5.2 Let $q \geq 3$ be a prime such that $p = 2q + 1$ is also prime. Then $\mathsf{QR}(\mathbf{Z}_p^*)$ is a group of prime order q. Furthermore, if g is any generator of $\mathbf{Z}_p^*$, then $g^2 \bmod p$ is a generator of $\mathsf{QR}(\mathbf{Z}_p^*)$. ▌

Note that the operation under which $\mathsf{QR}(\mathbf{Z}_p^*)$ is a group is multiplication modulo p, the same operation under which $\mathbf{Z}_p^*$ is a group.

Proof of Proposition 9.5.2: We know that $\mathsf{QR}(\mathbf{Z}_p^*)$ is a subgroup, hence a group in its own right. Proposition 9.4.2 tells us that $|\mathsf{QR}(\mathbf{Z}_p^*)|$ is $(p-1)/2$, which equals q in this case. Now let g be a generator of $\mathbf{Z}_p^*$ and let $h = g^2 \bmod p$. We want to show that h is a generator of $\mathsf{QR}(\mathbf{Z}_p^*)$. As per Proposition 9.5.1, we need only show that h is non-trivial, meaning $h \neq 1$. Indeed, we know that $g^2 \not\equiv 1 \pmod{p}$, because g, being a generator, has order p and our assumptions imply $p > 2$. ▌

Example 9.5.3 Let $q = 5$ and $p = 2q + 1 = 11$. Both p and q are primes. We know from Example 9.4.1 that

$$\mathsf{QR}(\mathbf{Z}_{11}^*) \;=\; \{1, 3, 4, 5, 9\} \; .$$

This is a group of prime order 5. We know from Example 9.3.1 that 2 is a generator of $\mathbf{Z}_p^*$. Proposition 9.5.2 tells us that $4 = 2^2$ is a generator of $\mathsf{QR}(\mathbf{Z}_{11}^*)$. We can verify this by raising 4 to the powers $i = 0, \ldots, 4$:

i	0	1	2	3	4
$4^i \bmod 11$	1	4	5	9	3

We see that the elements of the last row are exactly those of the set $\mathsf{QR}(\mathbf{Z}_{11}^*)$. ▌

Let us now explain what we perceive to be the advantage conferred by working in a group of prime order. Let G be a cyclic group, and g a generator. We know that the discrete logarithms to base g range in the set $\mathbf{Z}_m$ where $m = |G|$ is the order of G. This means that arithmetic in these exponents is modulo m. If G has prime order, then m is prime. This means that *any non-zero exponent has a multiplicative inverse modulo m.* In other words, in working in the exponents, we can divide. It is this that turns out to be useful.

As an example illustrating how we use this, let us return to the problem of the distribution of the DH key that we looked at in Section 9.4. Recall the question is that we draw x, y independently at random from $\mathbf{Z}_m$ and then ask how g^{xy} is distributed over G. We saw that when $G = \mathbf{Z}_p^*$ for a prime $p \geq 3$, this distribution was noticebly different from uniform. In a group of prime order, the distribution of the DH key, in contrast, is very close to uniform over G. It is not quite uniform, because the identity element of the group has a slightly higher probability of being the DH key than other group elements, but the deviation is small enough to be negligible for groups of reasonably large size. The following proposition summarizes the result.

Proposition 9.5.4 Suppose G is a group of order q where q is a prime, and let g be a generator of G. Then for any $Z \in G$ we have

$$\Pr\left[\, x \xleftarrow{\$} \mathbf{Z}_q \,;\, y \xleftarrow{\$} \mathbf{Z}_q \;:\; g^{xy} = Z \,\right] \;=\; \begin{cases} \dfrac{1}{q}\left(1 - \dfrac{1}{q}\right) & \text{if } Z \neq \mathbf{1} \\[2ex] \dfrac{1}{q}\left(2 - \dfrac{1}{q}\right) & \text{if } Z = \mathbf{1}, \end{cases}$$

where $\mathbf{1}$ denotes the identity element of G. ▌

Proof of Proposition 9.5.4: First suppose $Z = \mathbf{1}$. The DH key g^{xy} is $\mathbf{1}$ if and only if either x or y is 0 modulo q. Each is 0 with probability $1/q$ and these probabilities are independent, so the probability that either x or y is 0 is $2/q - 1/q^2$, as claimed.

Now suppose $Z \neq \mathbf{1}$. Let $z = \text{DLog}_{G,g}(Z)$, meaning $z \in \mathbf{Z}_q^*$ and $g^z = Z$. We will have $g^{xy} \equiv Z \pmod{p}$ if and only if $xy \equiv z \pmod{q}$, by the uniqueness of the discrete logarithm. For any fixed $x \in \mathbf{Z}_q^*$, there is exactly one $y \in \mathbf{Z}_q$ for which $xy \equiv z \pmod{q}$, namely $y = x^{-1}z \bmod q$, where x^{-1} is the multiplicative inverse of x in the group $\mathbf{Z}_q^*$. (Here we are making use of the fact that q is prime, since otherwise the inverse of x modulo q may not exist.) Now, suppose we choose x at random from $\mathbf{Z}_q$. If $x = 0$ then, regardless of the choice of $y \in \mathbf{Z}_q$, we will not have $xy \equiv z \pmod{q}$, because $z \not\equiv 0 \pmod{q}$. On the other hand, if $x \neq 0$ then there is exactly $1/q$ probability that the randomly chosen y is such that $xy \equiv z \pmod{q}$. So the probability that $xy \equiv z \pmod{q}$ when both x and y are chosen at random in $\mathbf{Z}_q$ is

$$\frac{q-1}{q} \cdot \frac{1}{q} \;=\; \frac{1}{q}\left(1 - \frac{1}{q}\right)$$

as desired. Here, the first term is because when we choose x at random from $\mathbf{Z}_q$, it has probability $(q-1)/q$ of landing in $\mathbf{Z}_q^*$. ▮

9.6 Historical Notes

9.7 Exercises and Problems

Chapter 10

Number-Theoretic Primitives

Number theory is a source of several computational problems that serve as primitives in the design of cryptographic schemes. Asymmetric cryptography in particular relies on these primitives. As with other beasts that we have been calling "primitives," these computational problems exhibit some intractability features, but by themselves do not solve any cryptographic problem directly relevant to a user security goal. But appropriately applied, they become useful to this end. In order to later effectively exploit them it is useful to first spend some time understanding them.

This understanding has two parts. The first is to provide precise definitions of the various problems and their measures of intractability. The second is to look at what is known or conjectured about the computational complexity of these problems.

There are two main classes of primitives. The first class relates to the discrete logarithm problem over appropriate groups, and the second to the factoring of composite integers. We look at them in turn.

This chapter assumes some knowledge of computational number theory as covered in the chapter on Computational Number Theory.

10.1 Discrete logarithm related problems

Let G be a cyclic group and let g be a generator of G. Recall this means that $G = \{g^0, g^1, \ldots, g^{m-1}\}$, where $m = |G|$ is the order of G. The discrete logarithm function $\mathrm{DLog}_{G,g}\colon\ G \to \mathbf{Z}_m$ takes input a group element a and returns the unique $i \in \mathbf{Z}_m$ such that $a = g^i$. There are several computational problems related to this function that are used as primitives.

10.1.1 Informal descriptions of the problems

The computational problems we consider in this setting are summarized in Fig. 10.1. In all cases, we are considering an attacker that knows the group G and the generator g. It is given the quantities listed in the column labeled "given," and is trying to compute the quantities, or answer the question, listed in the column labeled "figure out."

The most basic problem is the discrete logarithm (DL) problem. Informally stated, the attacker is given as input some group element X, and must compute $\mathrm{DLog}_{G,g}(X)$. This problem is conjectured to be computationally intractable in suitable groups G.

Problem	**Given**	**Figure out**
Discrete logarithm (DL)	g^x	x
Computational Diffie-Hellman (CDH)	g^x, g^y	g^{xy}
Decisional Diffie-Hellman (DDH)	g^x, g^y, g^z	Is $z \equiv xy \pmod{\|G\|}$?

Figure 10.1: An informal description of three discrete logarithm related problems over a cyclic group G with generator g. For each problem we indicate the input to the attacker, and what the attacker must figure out to "win." The formal definitions are in the text.

One might imagine "encrypting" a message $x \in \mathbf{Z}_m$ by letting g^x be the ciphertext. An adversary wanting to recover x is then faced with solving the discrete logarithm problem to do so. However, as a form of encryption, this has the disadvantage of being non-functional, because an intended recipient, namely the person to whom the sender is trying to communicate x, is faced with the same task as the adversary in attempting to recover x.

The Diffie-Hellman (DH) problems first appeared in the context of secret key exchange. Suppose two parties want to agree on a key which should remain unknown to an eavesdropping adversary. The first party picks $x \stackrel{\$}{\leftarrow} \mathbf{Z}_m$ and sends $X = g^x$ to the second party; the second party correspondingly picks $y \stackrel{\$}{\leftarrow} \mathbf{Z}_m$ and sends $Y = g^y$ to the first party. The quantity g^{xy} is called the DH-key corresponding to X, Y. We note that

$$Y^x = g^{xy} = X^y . \tag{10.1}$$

Thus the first party, knowing Y, x, can compute the DH key, as can the second party, knowing X, y. The adversary sees X, Y, so to recover the DH-key the adversary must solve the Computational Diffie-Hellman (CDH) problem, namely compute g^{xy} given $X = g^x$ and $Y = g^y$. Similarly, we will see later a simple asymmetric encryption scheme, based on Equation (10.1), where recovery of the encrypted message corresponds to solving the CDH problem.

The obvious route to solving the CDH problem is to try to compute the discrete logarithm of either X or Y and then use Equation (10.1) to obtain the DH key. However, there might be other routes that do not involve computing discrete logarithms, which is why CDH is singled out as a computational problem in its own right. This problem appears to be computationally intractable in a variety of groups.

We have seen before that security of a cryptographic scheme typically demands much more than merely the computational intractability of recovery of some underlying key. The computational intractability of the CDH problem turns out to be insufficient to guarantee the security of many schemes based on DH keys, including the secret key exchange protocol and encryption scheme mentioned above. The Decisional Diffie-Hellman (DDH) problem provides the adversary with a task that can be no harder, but possibly easier, than solving the CDH problem, namely to tell whether or not a given group element Z is the DH key corresponding to given group elements X, Y. This problem too appears to be computationally intractable in appropriate groups.

We now proceed to define the problems more formally. Having done that we will provide more specific discussions about their hardness in various different groups and their relations to each other.

10.1.2 The discrete logarithm problem

The description of the discrete logarithm problem given above was that the adversary is given as input some group element X, and is considered successful if it can output $\mathrm{DLog}_{G,g}(X)$. We would like to associate to a specific adversary A some advantage function measuring how well it does in solving this problem. The measure adopted is to look at the fraction of group elements for which the adversary is able to compute the discrete logarithm. In other words, we imagine the group element X given to the adversary as being drawn at random.

Definition 10.1.1 Let G be a cyclic group of order m, let g be a generator of G, and let A be an algorithm that returns an integer in $\mathbf{Z}_m$. We consider the following experiment:

```
Experiment $\mathrm{DL}_{G,g}^{A}$
    $x \stackrel{\$}{\leftarrow} \mathbf{Z}_m$ ; $X \leftarrow g^x$
    $\overline{x} \leftarrow A(X)$
    If $g^{\overline{x}} = X$ then return 1 else return 0
```

The *dl-advantage* of A is defined as

$$\mathbf{Adv}_{G,g}^{\mathrm{dl}}(A) \quad = \quad \Pr\left[\mathrm{DL}_{G,g}^{A} = 1\right] . \quad \blacksquare$$

Recall that the discrete exponentiation function takes input $i \in \mathbf{Z}_m$ and returns the group element g^i. The discrete logarithm function is the inverse of the discrete exponentiation function. The definition above simply measures the one-wayness of the discrete exponentiation function according to the standard definition of one-way function. It is to emphasize this that certain parts of the experiment are written the way they are.

The discrete logarithm problem is said to hard in G if the dl-advantage of any adversary of reasonable resources is small. Resources here means the time-complexity of the adversary, which includes its code size as usual.

10.1.3 The Computational Diffie-Hellman problem

As above, the transition from the informal description to the formal definition involves considering the group elements X, Y to be drawn at random.

Definition 10.1.2 Let G be a cyclic group of order m, let g be a generator of G, and let A be an algorithm that returns an element of G. We consider the following experiment:

```
Experiment $\mathrm{CDH}_{G,g}^{A}$
    $x \stackrel{\$}{\leftarrow} \mathbf{Z}_m$ ; $y \stackrel{\$}{\leftarrow} \mathbf{Z}_m$
    $X \leftarrow g^x$ ; $Y \leftarrow g^y$
    $Z \leftarrow A(X, Y)$
    If $Z = g^{xy}$ then return 1 else return 0
```

The *cdh-advantage* of A is defined as

$$\mathbf{Adv}_{G,g}^{\mathrm{cdh}}(A) \quad = \quad \Pr\left[\mathrm{CDH}_{G,g}^{A} = 1\right] . \quad \blacksquare$$

Again, the CDH problem is said to be hard in G if the cdh-advantage of any adversary of reasonable resources is small, where the resource in question is the adversary's time complexity.

10.1.4 The Decisional Diffie-Hellman problem

The formalization considers a "two worlds" setting. The adversary gets input X, Y, Z. In either world, X, Y are random group elements, but the manner in which Z is chosen depends on the world. In World 1, $Z = g^{xy}$ where $x = \mathrm{DLog}_{G,g}(X)$ and $y = \mathrm{DLog}_{G,g}(Y)$. In World 0, Z is chosen at random from the group, independently of X, Y. The adversary must decide in which world it is. (Notice that this is a little different from the informal description of Fig. 10.1 which said that the adversary is trying to determine whether or not $Z = g^{xy}$, because if by chance $Z = g^{xy}$ in World 0, we will declare the adversary unsuccessful if it answers 1.)

Definition 10.1.3 Let G be a cyclic group of order m, let g be a generator of G, let A be an algorithm that returns a bit, and let b be a bit. We consider the following experiments:

```
Experiment $\mathrm{DDH1}^A_{G,g}$
    $x \xleftarrow{\$} \mathbf{Z}_m$
    $y \xleftarrow{\$} \mathbf{Z}_m$
    $z \leftarrow xy \bmod m$
    $X \leftarrow g^x$ ; $Y \leftarrow g^y$ ; $Z \leftarrow g^z$
    $d \leftarrow A(X, Y, Z)$
    Return $d$
```

```
Experiment $\mathrm{DDH0}^A_{G,g}$
    $x \xleftarrow{\$} \mathbf{Z}_m$
    $y \xleftarrow{\$} \mathbf{Z}_m$
    $z \xleftarrow{\$} \mathbf{Z}_m$
    $X \leftarrow g^x$ ; $Y \leftarrow g^y$ ; $Z \leftarrow g^z$
    $d \leftarrow A(X, Y, Z)$
    Return $d$
```

The *ddh-advantage* of A is defined as

$$\mathbf{Adv}^{\mathrm{ddh}}_{G,g}(A) \;=\; \Pr\left[\mathrm{DDH1}^A_{G,g} = 1\right] - \Pr\left[\mathrm{DDH0}^A_{G,g} = 1\right] \; . \quad \blacksquare$$

Again, the DDH problem is said to be hard in G if the ddh-advantage of any adversary of reasonable resources is small, where the resource in question is the adversary's time complexity.

10.1.5 Relations between the problems

Relative to a fixed group G and generator g for G, if you can solve the DL problem then you can solve the CDH problem, and if you can solve the CDH problem then you can solve the DDH problem. So if DL is easy then CDH is easy, and if CDH is easy then DDH is easy. Equivalently, if DDH is hard then CDH is hard, and if CDH is hard then DL is hard.

We note that the converses of these statements are not known to be true. There are groups where DDH is easy, while CDH and DL appear to be hard. (We will see examples of such groups later.) Correspondingly, there could be groups where CDH is easy but DL is hard.

The following Proposition provides the formal statement and proof corresponding to the above claim that if you can solve the DL problem then you can solve the CDH problem, and if you can solve the CDH problem then you can solve the DDH problem.

Proposition 10.1.4 Let G be a cyclic group and let g be a generator of G. Let A_{dl} be an adversary (against the DL problem). Then there exists an adversary A_{cdh} (against the CDH problem) such that

$$\mathbf{Adv}^{\mathrm{dl}}_{G,g}(A_{\mathrm{dl}}) \;\leq\; \mathbf{Adv}^{\mathrm{cdh}}_{G,g}(A_{\mathrm{cdh}}) \; . \tag{10.2}$$

Furthermore the running time of A_{cdh} is the that of A_{dl} plus the time to do one exponentiation in G. Similarly let A_{cdh} be an adversary (against the CDH problem). Then there exists an adversary A_{ddh} (against the DDH problem) such that

$$\mathbf{Adv}^{\mathrm{cdh}}_{G,g}(A_{\mathrm{cdh}}) \;\leq\; \mathbf{Adv}^{\mathrm{ddh}}_{G,g}(A_{\mathrm{ddh}}) + \frac{1}{|G|} \; . \tag{10.3}$$

Furthermore the running time of $A_{\rm ddh}$ is the same as that of $A_{\rm cdh}$. ▌

Proof of Proposition 10.1.4: Adversary $A_{\rm cdh}$ works as follows:

```
Adversary $A_{\rm cdh}(X,Y)$
    $\overline{x} \leftarrow A(X)$
    $Z \leftarrow Y^{\overline{x}}$
    Return $Z$
```

Let $x = \mathrm{DLog}_{G,g}(X)$ and $y = \mathrm{DLog}_{G,g}(y)$. If $A_{\rm dl}$ is successful then its output $\overline{x}$ equals x. In that case

$$Y^{\overline{x}} \;=\; Y^x = (g^y)^x = g^{yx} = g^{xy}$$

is the correct output for $A_{\rm cdh}$. This justifies Equation (10.2).

We now turn to the second inequality in the proposition. Adversary $A_{\rm ddh}$ works as follows:

```
Adversary $A_{\rm ddh}(X,Y,Z)$
    $\overline{Z} \leftarrow B(X,Y)$
    If $\overline{Z} = Z$ then return 1 else return 0
```

We claim that

$$\begin{aligned}\Pr\left[\mathrm{DDH1}_{G,g}^{A_{\rm ddh}} = 1\right] &= \mathbf{Adv}_{G,g}^{\rm cdh}(A_{\rm cdh}) \\ \Pr\left[\mathrm{DDH0}_{G,g}^{A_{\rm ddh}} = 1\right] &= \frac{1}{|G|}\ ,\end{aligned}$$

which implies Equation (10.3). To justify the above, let $x = \mathrm{DLog}_{G,g}(X)$ and $y = \mathrm{DLog}_{G,g}(y)$. If $A_{\rm cdh}$ is successful then its output $\overline{Z}$ equals g^{xy}, so in world 1, $A_{\rm ddh}$ returns 1. On the other hand in world 0, Z is uniformly distributed over G and hence has probability $1/|G|$ of equalling $\overline{Z}$. ▌

10.2 The choice of group

The computational complexity of the above problems depends of course on the choice of group G. (But not perceptibly on the choice of generator g.) The issues are the type of group, and also its size. Let us look at some possibilities.

10.2.1 General groups

For any "reasonable" group G, there is an algorithm that can solve the discrete logarithm problem in time $|G|^{1/2} \cdot O(|p|^3)$. (The exceptions are groups lacking succinct representations of group elements, and we will not encounter such groups here.) In thinking about this running time we neglect the $|p|^3$ factor since it is very small compared to $|G|^{1/2}$, so that we view this as a $O(|G|^{1/2})$ algorithm.

There are several different algorithms with this running time. Shank's baby-step giant-step algorithm is the simplest, and is deterministic. Pollard's algorithm is randomized, and, although taking time on the same order as that taken by Shank's algorithm, is more space efficient, and preferred in practice.

Let us present Shank's baby-step giant-step algorithm. Let $m = |G|$ and let $n = \lceil\sqrt{m}\rceil$. Given $X = g^x$ we seek x. We note that there exist integers x_0, x_1 such that $0 \le x_0, x_1 \le n$ and

$x = nx_1 + x_0$. This means that $g^{nx_1+x_0} = X$, or $Xg^{-x_0} = (g^n)^{x_1}$. The idea of the algorithm is to compute two lists:

$$\begin{array}{lll} Xg^{-b} & \text{for} & b = 0, 1, \ldots, n \\ (g^n)^a & \text{for} & a = 0, 1, \ldots, n \end{array}$$

and then find a group element that is contained in both lists. The corresponding values of a, b satisfy $Xg^{-b} = (g^n)^a$, and thus $\mathrm{DLog}_{G,g}(X) = an + b$. The details follow.

```
Algorithm $A_{\mathrm{bsgs}}(X)$
    $n \leftarrow \lceil \sqrt{m} \rceil$ ; $N \leftarrow g^n$
    For $b = 0, \ldots, n$ do $B[Xg^{-b}] \leftarrow b$
    For $a = 0, \ldots, n$ do
        $Y \leftarrow N^a$
        If $B[Y]$ is defined then $x_0 \leftarrow B[Y]$ ; $x_1 \leftarrow a$
    Return $ax_1 + x_0$
```

This algorithm is interesting because it shows that there is a better way to compute the discrete logarithm of X than to do an exhaustive search for it. However, it does not yield a practical discrete logarithm computation method, because one can work in groups large enough that an $O(|G|^{1/2})$ algorithm is not really feasible. There are however better algorithms in some specific groups.

10.2.2 Integers modulo a prime

Naturally, the first specific group to consider is the integers modulo a prime, which we know is cyclic. So let $G = \mathbf{Z}_p^*$ for some prime p and let g be a generator of g. We consider the different problems in turn.

We begin by noting that the Decisional Diffie-Hellman problem is easy in this group. Some indication of this already appeared in the chapter on Computational Number Theory. In particular we saw there that the DH key g^{xy} is a square with probability 3/4 and a non-square with probability 1/4 if x, y are chosen at random from $\mathbf{Z}_{p-1}$. However, we know that a random group element is a square with probability 1/2. Thus, a strategy to tell which world we are in when given a triple X, Y, Z is to test whether or not Z is a square mod p. If so, bet on World 1, else on World 0. (We also know that the Jacobi symbol can be computed via an exponentiation mod p, so testing for squares can be done efficiently, specifically in cubic time.) A computation shows that this adversary has advantage 1/4, enough to show that the DDH problem is easy. The Proposition below presents a slightly better attack that achieves advantage 1/2, and provides the details of the analysis.

Proposition 10.2.1 Let $p \geq 3$ be a prime, let $G = \mathbf{Z}_p^*$, and let g be a generator of G. Then there is an adversary A, with running time $O(|p|^3)$ such that

$$\mathbf{Adv}^{\mathrm{ddh}}_{G,g}(A) = \frac{1}{2} . \quad \blacksquare$$

Proof of Proposition 10.2.1: The input to our adversary A is a triple X, Y, Z of group elements, and the adversary is trying to determine whether Z was chosen as g^{xy} or as a random group element, where x, y are the discrete logarithms of X and Y, respectively. We know that if we know $J_p(g^x)$ and $J_p(g^y)$, we can predict $J_p(g^{xy})$. Our adversary's strategy is to compute $J_p(g^x)$ and $J_p(g^y)$ and then see whether or not the challenge value Z has the Jacobi symbol value that g^{xy} ought to have. In more detail, it works as follows:

```
Adversary $A(X, Y, Z)$
    If $J_p(X) = 1$ or $J_p(Y) = 1$
        Then $s \leftarrow 1$ Else $s \leftarrow -1$
    If $J_p(Z) = s$ then return 1 else return 0
```

We know that the Jacobi symbol can be computed via an exponentiation modulo p, which we know takes $O(|p|^3)$ time. Thus, the time-complexity of the above adversary is $O(|p|^3)$. We now claim that

$$\begin{aligned}\Pr\left[\mathrm{DDH1}_{G,g}^{A} = 1\right] &= 1\\ \Pr\left[\mathrm{DDH0}_{G,g}^{A} = 1\right] &= \frac{1}{2}\,.\end{aligned}$$

Subtracting, we get

$$\mathbf{Adv}_{G,g}^{\mathrm{ddh}}(A) \;=\; \Pr\left[\mathrm{DDH1}_{G,g}^{A} = 1\right] - \Pr\left[\mathrm{DDH0}_{G,g}^{A} = 1\right] \;=\; 1 - \frac{1}{2} = \frac{1}{2}$$

as desired. Let us now see why the two equations above are true.

Let $x = \mathrm{DLog}_{G,g}(X)$ and $y = \mathrm{DLog}_{G,g}(Y)$. We know that the value s computed by our adversary A equals $J_p(g^{xy} \bmod p)$. But in World 1, $Z = g^{xy} \bmod p$, so our adversary will always return 1. In World 0, Z is distributed uniformly over G, so

$$\Pr\left[J_p(Z) = 1\right] \;=\; \Pr\left[J_p(Z) = -1\right] \;=\; \frac{(p-1)/2}{p-1} \;=\; \frac{1}{2}\,.$$

Since s is distributed independently of Z, the probability that $J_p(Z) = s$ is 1/2. ▌

Now we consider the CDH and DL problems. It appears that the best approach to solving the CDH in problem in $\mathbf{Z}_p^*$ is via the computation of discrete logarithms. (This has not been proved in general, but there are proofs for some special classes of primes.) Thus, the main question is how hard is the computation of discrete logarithms. This depends both on the size and structure of p.

The currently best algorithm is the GNFS (General Number Field Sieve) which has a running time of the form

$$O(e^{(C+o(1))\cdot\ln(p)^{1/3}\cdot(\ln\ln(p))^{2/3}}) \qquad (10.4)$$

where $C \approx 1.92$. For certain classes of primes, the value of C is even smaller. These algorithms are heuristic, in the sense that the run time bounds are not proven, but appear to hold in practice.

If the prime factorization of the order of the group is known, the discrete logarithm problem over the group can be decomposed into a set of discrete logarithm problems over subgroups. As a result, if $p - 1 = p_1^{\alpha_1} \cdots p_n^{\alpha_n}$ is the prime factorization of $p-1$, then the discrete logarithm problem in $\mathbf{Z}_p^*$ can be solved in time on the order of

$$\sum_{i=1}^{n} \alpha_i \cdot (\sqrt{p_i} + |p|)\ .$$

If we want the discrete logarithm problem in $\mathbf{Z}_p^*$ to be hard, this means that it must be the case that at least one of the prime factors p_i of $p-1$ is large enough that $\sqrt{p_i}$ is large.

The prime factorization of $p-1$ might be hard to compute given only p, but in fact we usually choose p in such a way that we know the prime factorization of $p-1$, because it is this that gives us a way to find a generator of the group $\mathbf{Z}_p^*$, as discussed in the chapter on Computational Number Theory So the above algorithm is quite relevant.

From the above, if we want to make the DL problem in $\mathbf{Z}_p^*$ hard, it is necessary to choose p so that it is large and has at least one large prime factor. A common choice is $p = sq + 1$ where $s \geq 2$ is some small integer (like $s = 2$) and q is a prime. In this case, $p - 1$ has the factor q, which is large.

Precise estimates of the size of a prime necessary to make a discrete logarithm algorithm infeasible are hard to make based on asymptotic running times of the form given above. Ultimately, what actual implementations can accomplish is the most useful data. In April 2001, it was announced that discrete logarithms had been computed modulo a 120 digit (ie. about 400 bit) prime (Joux and Lercier, 2001). The computation took 10 weeks and was done on a 525MHz quadri-processor Digital Alpha Server 8400 computer. The prime p did not have any special structure that was exploited, and the algorithm used was the GNFS. A little earlier, discrete logarithms had been computed modulo a slightly larger prime, namely a 129 digit one, but this had a special structure that was exploited [35].

Faster discrete logarithm computation can come from many sources. One is exploiting parallelism and the paradigm of distributing work across available machines on the Internet. Another is algorithmic improvements. A reduction in the constant C of Equation (10.4) has important impact on the running time. A reduction in the exponents from $1/3, 2/3$ to $1/4, 3/4$ would have an even greater impact. There are also threats from hardware approaches such as the design of special purpose discrete logarithm computation devices. Finally, the discrete logarithm probably can be solved in polynomial time with a quantum computer. Whether a quantum computer can be built is not known.

Predictions are hard to make. In choosing a prime p for cryptography over $\mathbf{Z}_p^*$, the security risks must be weighed against the increase in the cost of computations over $\mathbf{Z}_p^*$ as a function of the size of p.

10.2.3 Other groups

In elliptic curve groups, the best known algorithm is the $O(\sqrt{|G|})$ one mentioned above. Thus, it is possible to use elliptic curve groups of smaller size than groups of integers modulo a prime for the same level of security, leading to improved efficiency for implementing discrete log based cryptosystem.

10.3 The RSA system

The RSA system is the basis of the most popular public-key cryptography solutions. Here we provide the basic mathematical and computational background that will be used later.

10.3.1 The basic mathematics

We begin with a piece of notation:

Definition 10.3.1 Let $N, f \geq 1$ be integers. The *RSA function associated to* N, f is the function $\mathsf{RSA}_{N,f}\colon\ \mathbf{Z}_N^* \to \mathbf{Z}_N^*$ defined by $\mathsf{RSA}_{N,f}(w) = w^f \bmod N$ for all $w \in \mathbf{Z}_N^*$. ▮

The RSA function associated to N, f is thus simply exponentiation with exponent f in the group $\mathbf{Z}_N^*$, but it is useful in the current context to give it a new name. The following summarizes a basic property of this function. Recall that $\varphi(N)$ is the order of the group $\mathbf{Z}_N^*$.

Proposition 10.3.2 Let $N \geq 2$ and $e, d \in \mathbf{Z}^*_{\varphi(N)}$ be integers such that $ed \equiv 1 \pmod{\varphi(N)}$. Then the RSA functions $\mathsf{RSA}_{N,e}$ and $\mathsf{RSA}_{N,d}$ are both permutations on $\mathbf{Z}^*_N$ and, moreover, are inverses of each other, ie. $\mathsf{RSA}^{-1}_{N,e} = \mathsf{RSA}_{N,d}$ and $\mathsf{RSA}^{-1}_{N,d} = \mathsf{RSA}_{N,e}$. ▌

A permutation, above, simply means a bijection from $\mathbf{Z}^*_N$ to $\mathbf{Z}^*_N$, or, in other words, a one-to-one, onto map. The condition $ed \equiv 1 \pmod{\varphi(N)}$ says that d is the inverse of e in the group $\mathbf{Z}^*_{\varphi(N)}$.

Proof of Proposition 10.3.2: For any $x \in \mathbf{Z}^*_N$, the following hold modulo N:

$$\mathsf{RSA}_{N,d}(\mathsf{RSA}_{N,e}(x)) \equiv (x^e)^d \equiv x^{ed} \equiv x^{ed \bmod \varphi(N)} \equiv x^1 \equiv x \; .$$

The third equivalence used the fact that $\varphi(N)$ is the order of the group $\mathbf{Z}^*_N$. The fourth used the assumed condition on e, d. Similarly, we can show that for any $y \in \mathbf{Z}^*_N$,

$$\mathsf{RSA}_{N,e}(\mathsf{RSA}_{N,d}(y)) \equiv y$$

modulo N. These two facts justify all the claims of the Proposition. ▌

With N, e, d as in Proposition 10.3.2 we remark that

- For any $x \in \mathbf{Z}^*_N$: $\mathsf{RSA}_{N,e}(x) = \text{MOD-EXP}(x, e, N)$ and so one can efficiently compute $\mathsf{RSA}_{N,e}(x)$ given N, e, x.
- For any $y \in \mathbf{Z}^*_N$: $\mathsf{RSA}_{N,d}(y) = \text{MOD-EXP}(y, d, N)$ and so one can efficiently compute $\mathsf{RSA}_{N,d}(y)$ given N, d, y.

We now consider an adversary that is given N, e, y and asked to compute $\mathsf{RSA}^{-1}_{N,e}(y)$. If it had d, this could be done efficiently by the above, but we do not give it d. It turns out that when the paremeters N, e are properly chosen, this adversarial task appears to be computationally infeasible, and this property will form the basis of both asymmetric encryption schemes and digital signature schemes based on RSA. Our goal in this section is to lay the groundwork for these later applications by showing how RSA parameters can be chosen so as to make the above claim of computational difficulty true, and formalizing the sense in which it is true.

10.3.2 Generation of RSA parameters

We begin with a computational fact.

Proposition 10.3.3 There is an $O(k^2)$ time algorithm that on inputs $\varphi(N), e$ where $e \in \mathbf{Z}^*_{\varphi(N)}$ and $N < 2^k$, returns $d \in \mathbf{Z}^*_{\varphi(N)}$ satisfying $ed \equiv 1 \pmod{\varphi(N)}$. ▌

Proof of Proposition 10.3.3: Since d is the inverse of e in the group $\mathbf{Z}^*_{\varphi(N)}$, the algorithm consists simply of running MOD-INV$(e, \varphi(N))$ and returning the outcome. Recall that the modular inversion algorithm invokes the extended-gcd algorithm as a subroutine and has running time quadratic in the bit-length of its inputs. ▌

To choose RSA parameters, one runs a generator. We consider a few types of geneators:

Definition 10.3.4 A *modulus generator* with *associated security parameter* k (where $k \geq 2$ is an integer) is a randomized algorithm that takes no inputs and returns integers N, p, q satisfying:

1. p, q are distinct, odd primes

2. $N = pq$

3. $2^{k-1} \leq N < 2^k$ (ie. N has bit-length k).

An *RSA generator* with *associated security parameter* k is a randomized algorithm that takes no inputs and returns a pair $((N, e), (N, p, q, d))$ such that the three conditions above are true, and, in addition,

4. $e, d \in \mathbf{Z}^*_{(p-1)(q-1)}$

5. $ed \equiv 1 \pmod{(p-1)(q-1)}$

We call N an *RSA modulus*, or just modulus. We call e the *encryption exponent* and d the *decryption exponent.* ▌

Note that $(p-1)(q-1) = \varphi(N)$ is the size of the group $\mathbf{Z}^*_N$. So above, e, d are relatively prime to the order of the group $\mathbf{Z}^*_N$. As the above indicates, we are going to restrict attention to numbers N that are the product of two distinct odd primes. Condition (4) for the RSA generator translates to $1 \le e, d < (p-1)(q-1)$ and $\gcd(e, (p-1)(q-1)) = \gcd(d, (p-1)(q-1)) = 1$.

For parameter generation to be feasible, the generation algorithm must be efficient. There are many different possible efficient generators. We illustrate a few.

In modulus generation, we usually pick the primes p, q at random, with each being about $k/2$ bits long. The corresponding modulus generator $\mathcal{K}^\$_{\mathrm{mod}}$ with associated security parameter k works as follows:

```
Algorithm $\mathcal{K}^\$_{\mathrm{mod}}$
    $\ell_1 \leftarrow \lfloor k/2 \rfloor$ ; $\ell_2 \leftarrow \lceil k/2 \rceil$
    Repeat
        $p \xleftarrow{\$} \{2^{\ell_1 - 1}, \ldots, 2^{\ell_1} - 1\}$ ; $q \xleftarrow{\$} \{2^{\ell_2 - 1}, \ldots, 2^{\ell_2} - 1\}$
    Until the following conditions are all true:
    –   TEST-PRIME($p$) = 1 and TEST-PRIME($q$) = 1
    –   $p \neq q$
    –   $2^{k-1} \le N$
    $N \leftarrow pq$
    Return $(N, e), (N, p, q, d)$
```

Above, TEST-PRIME denotes an algorithm that takes input an integer and returns 1 or 0. It is designed so that, with high probability, the former happens when the input is prime and the latter when the input is composite.

Sometimes, we may want modulii product of primes having a special form, for example primes p, q such that $(p-1)/2$ and $(q-1)/2$ are both prime. This corresponds to a different modulus generator, which works as above but simply adds, to the list of conditions tested to exit the loop, the conditions TEST-PRIME($(p-1)/2$)) = 1 and TEST-PRIME($(q-1)/2$)) = 1. There are numerous other possible modulus generators too.

An RSA generator, in addition to N, p, q, needs to generate the exponents e, d. There are several options for this. One is to first choose N, p, q, then pick e at random subject to $\gcd(N, \varphi(N)) = 1$, and compute d via the algorithm of Proposition 10.3.3. This *random-exponent RSA generator*, denoted $\mathcal{K}^\$_{\mathrm{rsa}}$, is detailed below:

```
Algorithm $\mathcal{K}^\$_{\mathrm{rsa}}$
    $(N, p, q) \xleftarrow{\$} \mathcal{K}^\$_{\mathrm{mod}}$
    $M \leftarrow (p-1)(q-1)$
    $e \xleftarrow{\$} \mathbf{Z}^*_M$
    Compute $d$ by running the algorithm of Proposition 10.3.3 on inputs $M, e$
    Return $((N, e), (N, p, q, d))$
```

In order to speed-up computation of $\mathsf{RSA}_{N,e}$, however, we often like e to be small. To enable this, we begin by setting e to some small prime number like 3, and then picking the other parameters appropriately. In particular we associate to any odd prime number e the following *exponent-e RSA generator*:

```
Algorithm $\mathcal{K}_{\rm rsa}^{e}$
    Repeat
        $(N, p, q) \xleftarrow{\$} \mathcal{K}_{\rm mod}^{\$}(k)$
    Until
    –  $e < (p-1)$ and $e < (q-1)$
    –  $\gcd(e, (p-1)) = \gcd(e, (q-1)) = 1$
    $M \leftarrow (p-1)(q-1)$
    Compute $d$ by running the algorithm of Proposition 10.3.3 on inputs $M, e$
    Return $((N, e), (N, p, q, d))$
```

10.3.3 One-wayness problems

The basic assumed security property of the RSA functions is one-wayness, meaning given N, e, y it is hard to compute $\mathsf{RSA}_{N,e}^{-1}(y)$. One must be careful to formalize this properly though. The formalization chooses y at random.

Definition 10.3.5 Let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k, and let A be an algorithm. We consider the following experiment:

```
Experiment $\mathbf{Exp}_{\mathcal{K}_{\rm rsa}}^{\text{ow-kea}}(A)$
    $((N, e), (N, p, q, d)) \xleftarrow{\$} \mathcal{K}_{\rm rsa}$
    $x \xleftarrow{\$} \mathbf{Z}_N^*$ ; $y \leftarrow x^e \bmod N$
    $x' \xleftarrow{\$} A(N, e, y)$
    If $x' = x$ then return 1 else return 0
```

The *ow-kea-advantage* of A is defined as

$$\mathbf{Adv}_{\mathcal{K}_{\rm rsa}}^{\text{ow-kea}}(A) \quad = \quad \Pr\left[\mathbf{Exp}_{\mathcal{K}_{\rm rsa}}^{\text{ow-kea}}(A) = 1\right] \ . \ \blacksquare$$

Above, "kea" stands for "known-exponent attack." We might also allow a chosen-exponent attack, abbreviated "cea," in which, rather than having the encryption exponent specified by the instance of the problem, one allows the adversary to choose it. The only condition imposed is that the adversary not choose $e = 1$.

Definition 10.3.6 Let $\mathcal{K}_{\rm mod}$ be a modulus generator with associated security parameter k, and let A be an algorithm. We consider the following experiment:

```
Experiment $\mathbf{Exp}_{\mathcal{K}_{\rm rsa}}^{\text{ow-cea}}(A)$
    $(N, p, q) \xleftarrow{\$} \mathcal{K}_{\rm mod}$
    $y \xleftarrow{\$} \mathbf{Z}_N^*$
    $(x, e) \xleftarrow{\$} A(N, y)$
    If $x^e \equiv y \pmod{N}$ and $e > 1$
        then return 1 else return 0.
```

The *ow-cea-advantage* of A is defined as

$$\mathbf{Adv}^{\text{ow-cea}}_{\mathcal{K}_{\text{mod}}}(A) \quad = \quad \Pr\left[\mathbf{Exp}^{\text{ow-cea}}_{\mathcal{K}_{\text{mod}}}(A) = 1\right] \; . \quad \blacksquare$$

10.4 Historical notes

10.5 Exercises and Problems

Bibliography

[1] T. Denny and D. Weber The solution of Mccurley's discrete logchallenge. *Advances in Cryptology – CRYPTO '98*, Lecture Notes in Computer Science Vol. 1462, H. Krawczyk ed., Springer-Verlag, 1998.

Chapter 11

Digital signatures

In the public key setting, the primitive used to provide data integrity is a digital signature scheme. In this chapter we look at security notions and constructions for this primitive.

11.1 Digital signature schemes

A digital signature scheme is just like a message authentication scheme except for an asymmetry in the key structure. The key sk used to generate signatures (in this setting the tags are often called signatures) is different from the key pk used to verify signatures. Furthermore pk is public, in the sense that the adversary knows it too. So while only a signer in possession of the secret key can generate signatures, anyone in possession of the corresponding public key can verify the signatures.

Definition 11.1.1 A *digital signature scheme* $\mathcal{DS} = (\mathcal{K}, \mathrm{Sig}, \mathrm{VF})$ consists of three algorithms, as follows:

- The randomized *key generation* algorithm $\mathcal{K}$ (takes no inputs and) returns a pair (pk, sk) of keys, the public key and matching secret key, respectively. We write $(pk, sk) \overset{\$}{\leftarrow} \mathcal{K}$ for the operation of executing $\mathcal{K}$ and letting (pk, sk) be the pair of keys returned.
- The *signing* algorithm Sig takes the secret key sk and a message M to return a *signature* (also sometimes called a *tag*) $\sigma \in \{0,1\}^* \cup \{\perp\}$. The algorithm may be randomized or stateful. We write $\sigma \overset{\$}{\leftarrow} \mathrm{Sig}_{sk}(M)$ or $\sigma \overset{\$}{\leftarrow} \mathrm{Sig}(sk, M)$ for the operation of running Sig on inputs sk, M and letting σ be the signature returned.
- The deterministic *verification* algorithm VF takes a public key pk, a message M, and a candidate signature σ for M to return a bit. We write $d \leftarrow \mathrm{VF}_{pk}(M, \sigma)$ or $d \leftarrow \mathrm{VF}(pk, M, \sigma)$ to denote the operation of running VF on inputs pk, M, σ and letting d be the bit returned.

We require that $\mathrm{VF}_{pk}(M, \sigma) = 1$ for any key-pair (pk, sk) that might be output by $\mathcal{K}$, any message M, and any $\sigma \neq \perp$ that might be output by $\mathrm{Sig}_{sk}(M)$. If Sig is stateless then we associate to each public key a *message space* $\mathsf{Messages}(pk)$ which is the set of all M for which $\mathrm{Sig}_{sk}(M)$ never returns $\perp$. ▮

Let S be an entity that wants to have a digital signature capability. The first step is key generation: S runs $\mathcal{K}$ to generate a pair of keys (pk, sk) for itself. Note the key generation algorithm is run locally by S. Now, S can produce a digital signature on some document $M \in \mathsf{Messages}(pk)$ by running

$\mathrm{Sig}_{sk}(M)$ to return a signature σ. The pair (M, σ) is then the authenticated version of the document. Upon receiving a document M' and tag σ' purporting to be from S, a receiver B in possession of pk verifies the authenticity of the signature by using the specified verification procedure, which depends on the message, signature, and public key. Namely he computes $\mathrm{VF}_{pk}(M', \sigma')$, whose value is a bit. If this value is 1, it is read as saying the data is authentic, and so B accepts it as coming from S. Else it discards the data as unauthentic.

Note that an entity wishing to verify S's signatures must be in possession of S's public key pk, and must be assured that the public key is authentic, meaning really is S's key and not someone else's key. We will look later into mechanisms for assuring this state of knowledge. But the key management processes are not part of the digital signature scheme itself. In constructing and analyzing the security of digital signature schemes, we make the assumption that any prospective verifier is in possession of an authentic copy of the public key of the signer. This assumption is made in what follows.

A viable scheme of course requires some security properties. But these are not our concern now. First we want to pin down what constitutes a specification of a scheme, so that we know what are the kinds of objects whose security we want to assess.

The key usage is the "mirror-image" of the key usage in an asymmetric encryption scheme. In a digital signature scheme, the holder of the secret key is a sender, using the secret key to tag its own messages so that the tags can be verified by others. In an asymmetric encryption scheme, the holder of the secret key is a receiver, using the secret key to decrypt ciphertexts sent to it by others.

The signature algorithm might be randomized, meaning internally flip coins and use these coins to determine its output. In this case, there may be many correct tags associated to a single message M. The algorithm might also be stateful, for example making use of a counter that is maintained by the sender. In that case the signature algorithm will access the counter as a global variable, updating it as necessary. The algorithm might even be both randomized and stateful. However, unlike encryption schemes, whose encryption algorithms must be either randomized or stateful for the scheme to be secure, a deterministic, stateless signature algorithm is not only possible, but common.

The signing algorithm might only be willing to sign certain messages and not others. It indicates its unwillingness to sign a message by returning $\perp$. If the scheme is stateless, the message space, which can depend on the public key, is the set of all messages for which the probability that the signing algorithm returns $\perp$ is zero. If the scheme is stateful we do not talk of such a space since whether or not the signing algorithm returns $\perp$ can depend not only on the message but on its state.

The last part of the definition says that signatures that were correctly generated will pass the verification test. This simply ensures that authentic data will be accepted by the receiver. In the case of a sateful scheme, the requirement holds for any state of the signing algorithm.

11.2 A notion of security

Digital signatures aim to provide the same security property as message authentication schemes; the only change is the more flexible key structure. Accordingly, we can build on our past work in understanding and pinning down a notion of security for message authentication; the one for digital signatures differs only in that the adversary has access to the public key.

The goal of the adversary F is forgery: It wants to produce document M and tag σ such that $\mathrm{VF}_{pk}(M, \sigma) = 1$, but M did not originate with the sender S. The adversary is allowed a chosen-message attack in the process of trying to produce forgeries, and the scheme is secure if even after such an attack the adversary has low probability of producing forgeries.

Let $\mathcal{DS} = (\mathcal{K}, \mathrm{Sig}, \mathrm{VF})$ be an arbitrary digital signature scheme. Our goal is to formalize a measure of insecurity against forgery under chosen-message attack for this scheme. The adversary's actions are viewed as divided into two phases. The first is a "learning" phase in which it is given oracle access to $\mathrm{Sig}_{sk}(\cdot)$, where (pk, sk) was a priori chosen at random according to $\mathcal{K}$. It can query this oracle up to q times, in any manner it pleases, as long as all the queries are messages in the underlying message space $\mathsf{Messages}(pk)$ associated to this key. Once this phase is over, it enters a "forgery" phases, in which it outputs a pair (M, σ). The adversary is declared successful if $M \in \mathsf{Messages}(pk)$, $\mathrm{VF}_{pk}(M, \sigma) = 1$ and M was not a query made by the adversary to the signing oracle. Associated to any adversary F is thus a success probability called its advantage. (The probability is over the choice of keys, any probabilistic choices that Sig might make, and the probabilistic choices, if any, that F makes.) The advantage of the scheme is the success probability of the "cleverest" possible adversary, amongst all adversaries restricted in their resources to some fixed amount. We choose as resources the running time of the adversary, the number of queries it makes, and the total bit-length of all queries combined plus the bit-length of the output message M in the forgery.

Definition 11.2.1 Let $\mathcal{DS} = (\mathcal{K}, \mathrm{Sig}, \mathrm{VF})$ be a digital signature scheme, and let A be an algorithm that has access to an oracle and returns a pair of strings. We consider the following experiment:

```
Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(A)$
    $(pk, sk) \xleftarrow{\$} \mathcal{K}$
    $(M, \sigma) \leftarrow A^{\mathrm{Sig}_{sk}(\cdot)}(pk)$
    If the following are true return 1 else return 0:
    – $\mathrm{VF}_{pk}(M, \sigma) = 1$
    – $M \in \mathsf{Messages}(pk)$
    – $M$ was not a query of $A$ to its oracle
```

The *uf-cma-advantage* of A is defined as

$$\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(A) \quad = \quad \Pr\left[\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(A) = 1\right] \; . \quad \blacksquare$$

In the case of message authentication schemes, we provided the adversary not only with an oracle for producing tags, but also with an oracle for verifying them. Above, there is no verification oracle. This is because verification of a digital signature does not depend on any quantity that is secret from the adversary. Since the adversary has the public key and knows the algorithm VF, it can verify as much as it pleases by running the latter.

When we talk of the time-complexity of an adversary, we mean the worst case total execution time of the entire experiment. This means the adversary complexity, defined as the worst case execution time of A plus the size of the code of the adversary A, in some fixed RAM model of computation (worst case means the maximum over A's coins or the answers returned in response to A's oracle queries), plus the time for other operations in the experiment, including the time for key generation and the computation of answers to oracle queries via execution of the encryption algorithm.

As adversary resources, we will consider this time complexity, the message length μ, and the number of queries q to the sign oracle. We define μ as the sum of the lengths of the oracle queries plus the length of the message in the forgery output by the adversary. In practice, the queries correspond to messages signed by the legitimate sender, and it would make sense that getting these examples is more expensive than just computing on one's own. That is, we would expect q to be smaller than t. That is why q, μ are resources separate from t.

11.3 RSA based signatures

The RSA trapdoor permutation is widely used as the basis for digital signature schemes. Let us see how.

11.3.1 Key generation for RSA systems

We will consider various methods for generating digital signatures using the RSA functions. While these methods differ in how the signature and verification algorithms operate, they share a common key-setup. Namely the public key of a user is a modulus N and an encryption exponent e, where $N = pq$ is the product of two distinct primes, and $e \in \mathbf{Z}^*_{\varphi(N)}$. The corresponding secret contains the decryption exponent $d \in \mathbf{Z}^*_{\varphi(N)}$ (and possibly other stuff too) where $ed \equiv 1 \pmod{\varphi(N)}$.

How are these parameters generated? We refer back to Definition 10.9 where we had introduced the notion of an RSA generator. This is a randomized algorithm having an associated security parameter and returning a pair $((N, e), (N, p, q, d))$ satisfying the various conditions listed in the definition. The key-generation algorithm of the digital signature scheme is simply such a generator, meaning the user's public key is (N, e) and its secret key is (N, p, q, d).

Note N is not really secret. Still, it turns out to be convenient to put it in the secret key. Also, the descriptions we provide of the signing process will usually depend only on N, d and not p, q, so it may not be clear why p, q are in the secret key. But in practice it is good to keep them there because their use speeds up signing via the Chinese Remainder theorem and algorithm.

Recall that the map $\mathsf{RSA}_{N,e}(\cdot) = (\cdot)^e \bmod N$ is a permutation on Z^*_N with inverse $\mathsf{RSA}_{N,d}(\cdot) = (\cdot)^d \bmod N$.

Below we will consider various signature schemes all of which use the above key generation algorithm and try to build in different ways on the one-wayness of RSA in order to securely sign.

11.3.2 Trapdoor signatures

Trapdoor signatures represent the most direct way in which to attempt to build on the one-wayness of RSA in order to sign. We believe that the signer, being in possession of the secret key N, d, is the only one who can compute the inverse RSA function $\mathsf{RSA}^{-1}_{N,e} = \mathsf{RSA}_{N,d}$. For anyone else, knowing only the public key N, e, this task is computationally infeasible. Accordingly, the signer signs a message by performing on it this "hard" operation. This requires that the message be a member of Z^*_N, which, for convenience, is assumed. It is possible to verify a signature by performing the "easy" operation of computing $\mathsf{RSA}_{N,e}$ on the claimed signature and seeing if we get back the message.

More precisely, let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k, as per Definition 10.9. We consider the digital signature scheme $\mathcal{DS} = (\mathcal{K}_{\rm rsa}, \mathrm{Sig}, \mathrm{VF})$ whose signing and verifying algorithms are as follows:

```
Algorithm $\mathrm{Sig}_{N,p,q,d}(M)$
    If $M \notin \mathbf{Z}^*_N$ then return $\perp$
    $x \leftarrow M^d \bmod N$
    Return $x$
```

```
Algorithm $\mathrm{VF}_{N,e}(M, x)$
    If $(M \notin \mathbf{Z}^*_N$ or $x \notin \mathbf{Z}^*_N)$ then return 0
    If $M = x^e \bmod N$ then return 1 else return 0
```

This is a deterministic stateless scheme, and the message space for public key (N, e) is $\mathsf{Messages}(N, e) = Z^*_N$, meaning the only messages that the signer signs are those which are elements of the group Z^*_N. In this scheme we have denoted the signature of M by x. The signing algorithm simply applies $\mathsf{RSA}_{N,d}$ to the message to get the signature, and the verifying algorithm applies $\mathsf{RSA}_{N,e}$ to the signature and tests whether the result equals the message.

The first thing to check is that signatures generated by the signing algorithm pass the verification test. This is true because of Proposition 10.7 which tells us that if $x = M^d \bmod N$ then $x^e = M \bmod N$.

Now, how secure is this scheme? As we said above, the intuition behind it is that the signing operation should be something only the signer can perform, since computing $\mathsf{RSA}_{N,e}^{-1}(M)$ is hard without knowledge of d. However, what one should remember is that the formal assumed hardness property of RSA, namely one-wayness under known-exponent attack (we call it just one-wayness henceforth) as specified in Definition 10.10, is under a very different model and setting than that of security for signatures. One-wayness tells us that if we select M at random and then feed it to an adversary (who knows N, e but not d) and ask the latter to find $x = \mathsf{RSA}_{N,e}^{-1}(M)$, then the adversary will have a hard time succeeding. But the adversary in a signature scheme is not given a random message M on which to forge a signature. Rather, its goal is to create a pair (M, x) such that $\mathrm{VF}_{N,e}(M, x) = 1$. It does not have to try to imitate the signing algorithm; it must only do something that satisfies the verification algorithm. In particular it is allowed to choose M rather than having to sign a given or random M. It is also allowed to obtain a valid signature on any message other than the M it eventually outputs, via the signing oracle, corresponding in this case to having an oracle for $\mathsf{RSA}_{N,e}^{-1}(\cdot)$. These features make it easy for an adversary to forge signatures.

A couple of simple forging strategies are illustrated below. The first is to simply output the forgery in which the message and signature are both set to 1. The second is to first pick at random a value that will play the role of the signature, and then compute the message based on it:

```
Forger $F_1^{\mathrm{Sig}_{N,p,q,d}(\cdot)}(N, e)$
    Return $(1, 1)$

Forger $F_2^{\mathrm{Sig}_{N,p,q,d}(\cdot)}(N, e)$
    $x \xleftarrow{\$} Z_N^*$ ; $M \leftarrow x^e \bmod N$
    Return $(M, x)$
```

These forgers makes no queries to their signing oracles. We note that $1^e \equiv 1 \pmod{N}$, and hence the uf-cma-advantage of F_1 is 1. Similarly, the value (M, x) returned by the second forger satisfies $x^e \bmod N = M$ and hence it has uf-cma-advantage 1 too. The time-complexity in both cases is very low. (In the second case, the forger uses the $O(k^3)$ time to do its exponentiation modulo N.) So these attacks indicate the scheme is totally insecure.

The message M whose signature the above forger managed to forge is random. This is enough to break the scheme as per our definition of security, because we made a very strong definition of security. Actually for this scheme it is possible to even forge the signature of a given message M, but this time one has to use the signing oracle. The attack relies on the multiplicativity of the RSA function.

```
Forger $F^{\mathrm{Sig}_{N,e}(\cdot)}(N, e)$
    $M_1 \xleftarrow{\$} Z_N^* - \{1, M\}$ ; $M_2 \leftarrow M M_1^{-1} \bmod N$
    $x_1 \leftarrow \mathrm{Sig}_{N,e}(M_1)$ ; $x_2 \leftarrow \mathrm{Sig}_{N,e}(M_2)$
    $x \leftarrow x_1 x_2 \bmod N$
    Return $(M, x)$
```

Given M the forger wants to compute a valid signature x for M. It creates M_1, M_2 as shown, and obtains their signatures x_1, x_2. It then sets $x = x_1 x_2 \bmod N$. Now the verification algorithm will check whether $x^e \bmod N = M$. But note that

$$x^e \equiv (x_1 x_2)^e \equiv x_1^e x_2^e \equiv M_1 M_2 \equiv M \pmod{N} .$$

Here we used the multiplicativity of the RSA function and the fact that x_i is a valid signature of M_i for $i = 1, 2$. This means that x is a valid signature of M. Since M_1 is chosen to not be 1 or M, the same is true of M_2, and thus M was not an oracle query of F. So F succeeds with probability one.

These attacks indicate that there is more to signatures than one-wayness of the underlying function.

11.3.3 The hash-then-invert paradigm

Real-world RSA based signature schemes need to surmount the above attacks, and also attend to other impracticalities of the trapdoor setting. In particular, messages are not usually group elements; they are possibly long files, meaning bit strings of arbitrary lengths. Both issues are typically dealt with by pre-processing the given message M via a hash function to yield a point y in the range of $\mathsf{RSA}_{N,e}$, and then applying $\mathsf{RSA}_{N,e}^{-1}$ to y to obtain the signature. The hash function is public, meaning its description is known, and anyone can compute it.

To make this more precise, let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k and let Keys be the set of all modulli N that have positive probability to be output by $\mathcal{K}_{\rm rsa}$. Let Hash be a family of functions whose key-space is Keys and such that Hash_N: $\{0,1\}^* \to \mathbf{Z}_N^*$ for every $N \in \mathsf{Keys}$. Let $\mathcal{DS} = (\mathcal{K}_{\rm rsa}, \mathrm{Sig}, \mathrm{VF})$ be the digital signature scheme whose signing and verifying algorithms are as follows:

```
Algorithm $\mathrm{Sig}_{N,p,q,d}(M)$      | Algorithm $\mathrm{VF}_{N,e}(M,x)$
    $y \leftarrow \mathsf{Hash}_N(M)$          |     $y \leftarrow \mathsf{Hash}_N(M)$
    $x \leftarrow y^d \bmod N$              |     $y' \leftarrow x^e \bmod N$
    Return $x$                       |     If $y = y'$ then return 1 else return 0
```

Let us see why this might help resolve the weaknesses of trapdoor signatures, and what requirements security imposes on the hash function.

Let us return to the attacks presented on the trapdoor signature scheme above. Begin with the first forger we presented, who simply output $(1,1)$. Is this an attack on our new scheme? To tell, we see what happens when the above verification algorithm is invoked on input $1,1$. We see that it returns 1 only if $\mathsf{Hash}_N(1) \equiv 1^e \pmod N$. Thus, to prevent this attack it suffices to ensure that $\mathsf{Hash}_N(1) \neq 1$. The second forger we had previously set M to $x^e \bmod N$ for some random $x \in Z_N^*$. What is the success probability of this strategy under the hash-then-invert scheme? The forger wins if $x^e \bmod N = \mathsf{Hash}(M)$ (rather than merely $x^e \bmod N = M$ as before). The hope is that with a "good" hash function, it is very unlikely that $x^e \bmod N = \mathsf{Hash}_N(M)$. Consider now the third attack we presented above, which relied on the multiplicativity of the RSA function. For this attack to work under the hash-then-invert scheme, it would have to be true that

$$\mathsf{Hash}_N(M_1) \cdot \mathsf{Hash}_N(M_2) \equiv \mathsf{Hash}_N(M) \pmod N\ . \tag{11.1}$$

Again, with a "good" hash function, we would hope that this is unlikely to be true.

The hash function is thus supposed to "destroy" the algebraic structure that makes attacks like the above possible. How we might find one that does this is something we have not addressed.

While the hash function might prevent some attacks that worked on the trapdoor scheme, its use leads to a new line of attack, based on collisions in the hash function. If an adversary can find two distinct messages M_1, M_2 that hash to the same value, meaning $\mathsf{Hash}_N(M_1) = \mathsf{Hash}_N(M_2)$, then it can easily forge signatures, as follows:

```
Forger $F^{\mathrm{Sig}_{N,p,q,d}(\cdot)}(N,e)$
    $x_1 \leftarrow \mathrm{Sig}_{N,p,q,d}(M_1)$
    Return $(M_2, x_1)$
```

This works because M_1, M_2 have the same signature. Namely because x_1 is a valid signature of M_1, and because M_1, M_2 have the same hash value, we have

$$x_1^e \equiv \mathsf{Hash}_N(M_1) \equiv \mathsf{Hash}_N(M_2) \pmod N ,$$

and this means the verification procedure will accept x_1 as a signature of M_2. Thus, a necessary requirement on the hash function Hash is that it be CR2-KK, meaning given N it should be computationally infeasible to find distinct values M, M' such that $\mathsf{Hash}_N(M) = \mathsf{Hash}_N(M')$.

Below we will go on to more concrete instantiations of the hash-then-invert paradigm. But before we do that, it is important to try to assess what we have done so far. Above, we have pin-pointed some features of the hash function that are necessary for the security of the signature scheme. Collision-resistance is one. The other requirement is not so well formulated, but roughly we want to destroy algebraic structure in such a way that Equation (11.1), for example, should fail with high probability. Classical design focuses on these attacks and associated features of the hash function, and aims to implement suitable hash functions. But if you have been understanding the approaches and viewpoints we have been endeavoring to develop in this class and notes, you should have a more critical perspective. The key point to note is that what we need is not really to pin-point necessary features of the hash function to prevent certain attacks, but rather to pin-point *sufficient* features of the hash function, namely features sufficient to prevent *all* attacks, even ones that have not yet been conceived. And we have not done this. Of course, pinning down necessary features of the hash function is useful to gather intuition about what sufficient features might be, but it is only that, and we must be careful to not be seduced into thinking that it is enough, that we have identified all the concerns. Practice proves this complacence wrong again and again.

How can we hope to do better? Return to the basic philosophy of provable security. We want assurance that the signature scheme is secure under the assumption that its underlying primitives are secure. Thus we must try to tie the security of the signature scheme to the security of RSA as a one-way function, and some security condition on the hash function. With this in mind, let us proceed to examine some suggested solutions.

11.3.4 The PKCS #1 scheme

RSA corporation has been one of the main sources of software and standards for RSA based cryptography. RSA Labs (now a part of Security Dynamics Corporation) has created a set of standards called PKCS (Public Key Cryptography Standards). PKCS #1 is about signature (and encryption) schemes based on the RSA function. This standard is in wide use, and accordingly it will be illustrative to see what they do.

The standard uses the hash-then-invert paradigm, instantiating Hash via a particular hash function $\mathsf{PKCS\text{-}Hash}$ which we now describe. Recall we have already discussed collision-resistant hash functions. Let us fix a function $h\colon\{0,1\}^* \to \{0,1\}^l$ where $l \geq 128$ and which is "collision-resistant" in the sense that nobody knows how to find any pair of distinct points M, M' such that $h(M) = h(M')$. Currently the role tends to be played by SHA-1, so that $l = 160$. Prior to that it was MD5, which has $l = 128$. The RSA PKCS #1 standard defines

$$\mathsf{PKCS\text{-}Hash}_N(M) \quad = \quad \mathsf{00\,01\,FF\,FF} \cdots \mathsf{FF\,FF\,00} \,\|\, h(M) \,.$$

Here $\|$ denotes concatenation, and enough FF-bytes are inserted that the length of $\mathsf{PKCS\text{-}Hash}_N(M)$ is equal to k bits. Note the the first four bits of the hash output are zero, meaning as an integer it is certainly at most N, and thus most likely in $\mathbf{Z}_N^*$, since most numbers between 1 and N are in $\mathbf{Z}_N^*$. Also note that finding collisions in $\mathsf{PKCS\text{-}Hash}$ is no easier than finding collisions in h, so if the latter is collision-resistant then so is the former.

Recall that the signature scheme is exactly that of the hash-then-invert paradigm. For concreteness, let us rewrite the signing and verifying algorithms:

```
Algorithm $\mathrm{Sig}_{N,p,q,d}(M)$
    $y \leftarrow \mathsf{PKCS\text{-}Hash}_N(M)$
    $x \leftarrow y^d \bmod N$
    Return $x$

Algorithm $\mathrm{VF}_{N,e}(M,x)$
    $y \leftarrow \mathsf{PKCS\text{-}Hash}_N(M)$
    $y' \leftarrow x^e \bmod N$
    If $y = y'$ then return 1 else return 0
```

Now what about the security of this signature scheme? Our first concern is the kinds of algebraic attacks we saw on trapdoor signatures. As discussed in Section 11.3.3, we would like that relations like Equation (11.1) fail. This we appear to get; it is hard to imagine how $\mathsf{PKCS\text{-}Hash}_N(M_1) \cdot \mathsf{PKCS\text{-}Hash}_N(M_2) \bmod N$ could have the specific structure required to make it look like the PKCS-hash of some message. This isn't a proof that the attack is impossible, of course, but at least it is not evident.

This is the point where our approach departs from the classical attack-based design one. Under the latter, the above scheme is acceptable because known attacks fail. But looking deeper there is cause for concern. The approach we want to take is to see how the desired security of the signature scheme relates to the assumed or understood security of the underlying primitive, in this case the RSA function.

We are assuming RSA is one-way, meaning it is computationally infeasible to compute $\mathsf{RSA}_{N,e}^{-1}(y)$ for a randomly chosen point $y \in Z_N^*$. On the other hand, the points to which $\mathsf{RSA}_{N,e}^{-1}$ is applied in the signature scheme are those in the set $S_N = \{\, \mathsf{PKCS\text{-}Hash}_N(M) \;:\; M \in \{0,1\}^* \,\}$. The size of S_N is at most 2^l since h outputs l bits and the other bits of $\mathsf{PKCS\text{-}Hash}_N(\cdot)$ are fixed. With SHA-1 this means $|S_N| \leq 2^{160}$. This may seem like quite a big set, but within the RSA domain Z_N^* it is tiny. For example when $k = 1024$, which is a recommended value of the security parameter these days, we have

$$\frac{|S_N|}{|Z_N^*|} \;\leq\; \frac{2^{160}}{2^{1023}} \;=\; \frac{1}{2^{863}}\,.$$

This is the probability with which a point chosen randomly from Z_N^* lands in S_N. For all practical purposes, it is zero. So RSA could very well be one-way and still be easy to invert on S_N, since the chance of a random point landing in S_N is so tiny. So the security of the PKCS scheme cannot be guaranteed solely under the standard one-wayness assumption on RSA. Note this is true no matter how "good" is the underlying hash function h (in this case SHA-1) which forms the basis for $\mathsf{PKCS\text{-}Hash}$. The problem is the design of $\mathsf{PKCS\text{-}Hash}$ itself, in particular the padding.

The security of the PKCS signature scheme would require the assumption that RSA is hard to invert on the set S_N, a miniscule fraction of its full range. (And even this would be only a necessary, but not sufficient condition for the security of the signature scheme.)

Let us try to clarify and emphasize the view taken here. We are not saying that we know how to attack the PKCS scheme. But we are saying that an absence of known attacks should not be deemed a good reason to be satisfied with the scheme. We can identify "design flaws," such as the way the scheme uses RSA, which is not in accordance with our understanding of the security of RSA as a one-way function. And this is cause for concern.

11.3.5 The FDH scheme

From the above we see that if the hash-then-invert paradigm is to yield a signature scheme whose security can be based on the one-wayness of the RSA function, it must be that the points y on which $\mathsf{RSA}_{N,e}^{-1}$ is applied in the scheme are random ones. In other words, the output of the hash function must always "look random". Yet, even this only highlights a necessary condition, not (as far as we know) a sufficient one.

We now ask ourselves the following question. Suppose we had a "perfect" hash function Hash. In that case, at least, is the hash-then-invert signature scheme secure? To address this we must first decide what is a "perfect" hash function. The answer is quite natural: one that is random, namely returns a random answer to any query except for being consistent with respect to past queries. (We will explain more how this "random oracle" works later, but for the moment let us continue.) So our question becomes: in a model where Hash is perfect, can we *prove* that the signature scheme is secure if RSA is one-way?

This is a basic question indeed. If the hash-then-invert paradigm is in any way viable, we really must be able to prove security in the case the hash function is perfect. Were it not possible to prove security in this model it would be extremely inadvisable to adopt the hash-then-invert paradigm; if it doesn't work for a perfect hash function, how can we expect it to work in any real world setting?

Accordingly, we now focus on this "thought experiment" involving the use of the signature scheme with a perfect hash function. It is a thought experiment because no specific hash function is perfect. Our "hash function" is no longer fixed, it is just a box that flips coins. Yet, this thought experiment has something important to say about the security of our signing paradigm. It is not only a key step in our understanding but will lead us to better concrete schemes as we will see later.

Now let us say more about perfect hash functions. We assume that Hash returns a random member of Z_N^* every time it is invoked, except that if twice invoked on the same message, it returns the same thing both times. In other words, it is an instance of a random function with domain $\{0,1\}^*$ and range Z_N^*. We have seen such objects before, when we studied pseudorandomness: remember that we defined pseudorandom functions by considering experiments involving random functions. So the concept is not new. We call Hash a random oracle, and denote it by H in this context. It is accessible to all parties, signer, verifiers and adversary, but as an oracle. This means it is only accessible across a specified interface. To compute $H(M)$ a party must make an oracle call. This means it outputs M together with some indication that it wants $H(M)$ back, and an appropriate value is returned. Specifically it can output a pair (hash, M), the first component being merely a formal symbol used to indicate that this is a hash-oracle query. Having output this, the calling algorithm waits for the answer. Once the value $H(M)$ is returned, it continues its execution.

The best way to think about H is as a dynamic process which maintains a table of input-output pairs. Every time a query (hash, M) is made, the process first checks if its table contains a pair of the form (M, y) for some y, and if so, returns y. Else it picks a random y in Z_N^*, puts (M, y) into the table, and returns y as the answer to the oracle query.

We consider the above hash-then-invert signature scheme in the model where the hash function Hash is a random oracle H. This is called the Full Domain Hash (FDH) scheme. More precisely, let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k. The *FDH-RSA signature scheme associated to* $\mathcal{K}_{\rm rsa}$ is the digital signature scheme $\mathcal{DS} = (\mathcal{K}_{\rm rsa}, \mathrm{Sig}, \mathrm{VF})$ whose signing and verifying algorithms are as follows:

```
Algorithm $\mathrm{Sig}_{N,p,q,d}^{H(\cdot)}(M)$  | Algorithm $\mathrm{VF}_{N,e}^{H(\cdot)}(M, x)$
    $y \leftarrow H(M)$                          |     $y \leftarrow H(M)$
    $x \leftarrow y^d \bmod N$                   |     $y' \leftarrow x^e \bmod N$
    Return $x$                                   |     If $y = y'$ then return 1 else return 0
```

The only change with respect to the way we wrote the algorithms for the generic hash-then-invert scheme of Section 11.3.3 is notational: we write H as a superscript to indicate that it is an oracle accessible only via the specified oracle interface. The instruction $y \leftarrow H(M)$ is implemented by making the query (hash, M) and letting y denote the answer returned, as discussed above.

We now ask ourselves whether the above signature scheme is secure under the assumption that RSA is one-way. To consider this question we first need to extend our definitions to encompass the new model. The key difference is that the success probability of an adversary is taken over the random choice of H in addition to the random choices previously considered. The forger F as before has access to a signing oracle, but now also has access to H. Furthermore, Sig and VF now have access to H. Let us first write the experiment that measures the success of forger F and then discuss it more.

```
Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$
    $((N,e),(N,p,q,d)) \xleftarrow{\$} \mathcal{K}_{\text{rsa}}$
    $H \xleftarrow{\$} \mathsf{Func}(\{0,1\}^*, \mathbf{Z}_N^*)$
    $(M,x) \xleftarrow{\$} F^{H(\cdot), \text{Sig}^{H(\cdot)}_{N,p,q,d}(\cdot)}(N,e)$
    If the following are true return 1 else return 0:
    – $\text{VF}^{H}_{pk}(M,\sigma) = 1$
    – $M$ was not a query of $A$ to its oracle
```

Note that the forger is given oracle access to H in addition to the usual access to the sign oracle that models a chosen-message attack. After querying its oracles some number of times the forger outputs a message M and candidate signature x for it. We say that F is successful if the verification process would accept M, x, but F never asked the signing oracle to sign M. (F is certainly allowed to make hash query M, and indeed it is hard to imagine how it might hope to succeed in forgery otherwise, but it is not allowed to make sign query M.) The *uf-cma-advantage* of A is defined as

$$\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(A) \quad = \quad \Pr\left[\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(A) = 1\right] .$$

We will want to consider adversaries with time-complexity at most t, making at most q_{sig} sign oracle queries and at most q_{hash} hash oracle queries, and with total query message length μ. Resources refer again to those of the entire experiment. We first define the *execution time* as the time taken by the entire experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. This means it includes the time to compute answers to oracle queries, to generate the keys, and even to verify the forgery. Then the time-complexity t is supposed to upper bound the execution time plus the size of the code of F. In counting hash queries we again look at the entire experiment and ask that the total number of queries to H here be at most q_{hash}. Included in the count are the direct hash queries of F, the indirect hash queries made by the signing oracle, and even the hash query made by the verification algorithm in the last step. This latter means that q_{hash} is always at least the number of hash queries required for a verification, which for FDH-RSA is one. In fact for FDH-RSA we will have $q_{\text{hash}} \geq q_{\text{sig}} + 1$, something to be kept in mind when interpreting later results. Finally μ is the sum of the lengths of all messages in sign queries plus the length of the final output message M.

However, there is one point that needs to be clarified here, namely that if time-complexity refers to that of the entire experiment, how do we measure the time to pick H at random? It is an infinite object and thus cannot be actually chosen in finite time. The answer is that although we write H as being chosen at random upfront in the experiment, this is not how it is implemented. Instead, imagine H as being chosen dynamically. Think of the process implementing the table we described, so that random choices are made only at the time the H oracle is called, and the cost is that of maintaining and updating a table that holds the values of H on inputs queried so far. Namely when a query M is made to H, we charge the cost of looking up the table, checking whether $H(M)$ was already defined and returning it if so, else picking a random point from $\mathbf{Z}_N^*$, putting it in the table with index M, and returning it as well.

In this setting we claim that the FDH-RSA scheme is secure. The following theorem upper bounds its uf-cma-advantage solely in terms of the ow-kea advantage of the underlying RSA generator.

Theorem 11.3.1 *Let $\mathcal{K}_{\rm rsa}$ be an RSA generator with associated security parameter k, and let $\mathcal{DS}$ be the FDH-RSA scheme associated to $\mathcal{K}_{\rm rsa}$. Let F be an adversary making at most $q_{\rm hash}$ queries to its hash oracle and at most $q_{\rm sig}$ queries to its signing oracle where $q_{\rm hash} \geq 1 + q_{\rm sig}$. Then there exists an adversary I such that*

$$\mathbf{Adv}^{\rm uf\text{-}cma}_{\mathcal{DS}}(F) \;\leq\; q_{\rm hash} \cdot \mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(I) \; . \tag{11.2}$$

and I, F are of comparable resources. ▮

The theorem says that the only way to forge signatures in the FDH-RSA scheme is to try to invert the RSA function on random points. There is some loss in security: it might be that the chance of breaking the signature scheme is larger than that of inverting RSA in comparable time, by a factor of the number of hash queries made in the forging experiment. But we can make $\mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(t')$ small enough that even $q_{\rm hash} \cdot \mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(t')$ is small, by choosing a larger modulus size k.

One must remember the caveat: this is in a model where the hash function is random. Yet, even this tells us something, namely that the hash-then-invert paradigm itself is sound, at least for "perfect" hash functions. This puts us in a better position to explore concrete instantiations of the paradigm.

Let us now proceed to the proof of Theorem 11.3.1. Remember that inverter I takes as input (N, e), describing $\mathsf{RSA}_{N,e}$, and also a point $y \in Z_N^*$. Its job is to try to output $\mathsf{RSA}_{N,e}^{-1}(y) = y^d \bmod N$, where d is the decryption exponent corresponding to encryption exponent e. Of course, neither d nor the factorization of N are available to I. The success of I is measured under a random choice of $((N, e), (N, p, q, d))$ as given by $\mathcal{K}_{\rm rsa}$, and also a random choice of y from Z_N^*. In order to accomplish its task, I will run F as a subroutine, on input public key (N, e), hoping somehow to use F's ability to forge signatures to find $\mathsf{RSA}_{N,e}^{-1}(y)$. Before we discuss how I might hope to use the forger to determine the inverse of point y, we need to take a closer look at what it means to run F as a subroutine.

Recall that F has access to two oracles, and makes calls to them. At any point in its execution it might output (hash, M). It will then wait for a return value, which it interprets as $H(M)$. Once this is received, it continues its execution. Similarly it might output (sign, M) and then wait to receive a value it interprets as $\mathrm{Sig}^{H(\cdot)}_{N,p,q,d}(M)$. Having got this value, it continues. The important thing to understand is that F, as an algorithm, merely communicates with oracles via an interface. It does not control what these oracles return. You might think of an oracle query like a system call. Think of F as writing an oracle query M at some specific prescribed place in memory. Some process is expected to put in another prescribed place a value that F will take as the answer. F reads what is there, and goes on.

When I executes F, no oracles are actually present. F does not know that. It will at some point make an oracle query, assuming the oracles are present, say query (hash, M). It then waits for an answer. If I wants to run F to completion, it is up to I to provide some answer to F as the answer to this oracle query. F will take whatever it is given and go on executing. If I cannot provide an answer, F will not continue running; it will just sit there, waiting. We have seen this idea of "simulation" before in several proofs: I is creating a "virtual reality" under which F can believe itself to be in its usual environment.

The strategy of I will be to take advantage of its control over responses to oracle queries. It will choose them in strange ways, not quite the way they were chosen in Experiment $\mathbf{Exp}_{\mathcal{DS}}^{\text{uf-cma}}(F)$. Since F is just an algorithm, it processes whatever it receives, and eventually will halt with some output, a claimed forgery (M, x). By clever choices of replies to oracle queries, I will ensure that F is fooled into not knowing that it is not really in $\mathbf{Exp}_{\mathcal{DS}}^{\text{uf-cma}}(F)$, and furthermore x will be the desired inverse of y. Not always, though; I has to be lucky. But it will be lucky often enough.

We begin by consider the case of a very simple forger F. It makes no sign queries and exactly one hash query (hash, M). It then outputs a pair (M, x) as the claimed forgery, the message M being the same in the hash query and the forgery. (In this case we have $q_{\text{sig}} = 0$ and $q_{\text{hash}} = 2$, the last due to the hash query of F and the final verification query in the experiment.) Now if F is successful then x is a valid signature of M, meaning $x^e \equiv H(M) \bmod N$, or, equivalently, $x \equiv H(M)^d \bmod N$. Somehow, F has found the inverse of $H(M)$, the value returned to it as the response to oracle query M. Now remember that I's goal had been to compute $y^d \bmod N$ where y was its given input. A natural thought suggests itself: If F can invert $\mathsf{RSA}_{N,e}$ at $H(M)$, then I will "set" $H(M)$ to y, and thereby obtain the inverse of y under $\mathsf{RSA}_{N,e}$. I can set $H(M)$ in this way because it controls the answers to oracle queries. When F makes query (hash, M), the inverter I will simply return y as the response. If F then outputs a valid forgery (M, x), we have $x = y^d \bmod N$, and I can output x, its job done.

But why would F return a valid forgery when it got y as its response to hash query M? Maybe it will refuse this, saying it will not work on points supplied by an inverter I. But this will not happen. F is simply an algorithm and works on whatever it is given. What is important is solely the distribution of the response. In Experiment $\mathbf{Exp}_{\mathcal{DS}}^{\text{uf-cma}}(F)$ the response to (hash, M) is a random element of Z_N^*. But y has exactly the same distribution, because that is how it is chosen in the experiment defining the success of I in breaking RSA as a one-way function. So F cannot behave any differently in this virtual reality than it could in its real world; its probability of returning a valid forgery is still $\mathbf{Adv}_{\mathcal{DS}}^{\text{uf-cma}}(F)$. Thus for this simple F the success probability of the inverter in finding $y^d \bmod N$ is exactly the same as the success probability of F in forging signatures. Equation (11.2) claims less, so we certainly satisfy it.

However, most forgers will not be so obliging as to make no sign queries, and just one hash query consisting of the very message in their forgery. I must be able to handle any forger.

Inverter I will define a pair of subroutines, *H-Sim* (called the hash oracle simulator) and *Sign-Sim* (called the sign oracle simulator) to play the role of the hash and sign oracles respectively. Namely, whenever F makes a query (hash, M) the inverter I will return *H-Sim*(M) to F as the answer, and whenever F makes a query (sign, M) the inverter I will return *Sign-Sim*(M) to F as the answer. (The *Sign-Sim* routine will additionally invoke *H-Sim*.) As it executes, I will build up various tables (arrays) that "define" H. For $j = 1, \ldots, q_{\text{hash}}$, the j-th string on which H is called in the experiment (either directly due to a hash query by F, indirectly due to a sign query by F, or due to the final verification query) will be recorded as $Msg[j]$; the response returned by the hash oracle simulator to $Msg[j]$ is stored as $Y[j]$; and if $Msg[j]$ is a sign query then the response returned to F as the "signature" is $X[j]$. Now the question is how I defines all these values.

Suppose the j-th hash query in the experiment arises indirectly, as a result of a sign query $(\mathsf{sign}, Msg[j])$ by F. In Experiment $\mathbf{Exp}_{\mathcal{DS}}^{\text{uf-cma}}(F)$ the forger will be returned $H(Msg[j])^d \bmod N$. If I wants to keep F running it must return something plausible. What could I do? It could attempt to directly mimic the signing process, setting $Y[j]$ to a random value (remember $Y[j]$ plays the role of $H(Msg[j])$) and returning $(Y[j])^d \bmod N$. But it won't be able to compute the latter since it is not in possesion of the secret signing exponent d. The trick, instead, is that I first picks a value $X[j]$ at random in Z_N^* and sets $Y[j] = (X[j])^e \bmod N$. Now it can return $X[j]$ as the answer to the sign query, and this answer is accurate in the sense that the verification relation (which F might check) holds: we have $Y[j] \equiv (X[j])^e \bmod N$.

This leaves a couple of loose ends. One is that we assumed above that I has the liberty of defining $Y[j]$ at the point the sign query was made. But perhaps $Msg[j] = Msg[l]$ for some $l < j$ due to there having been a hash query involving this same message in the past. Then the hash value $Y[j]$ is already defined, as $Y[l]$, and cannot be changed. This can be addressed quite simply however: for any hash query $Msg[l]$, the hash simulator can follow the above strategy of setting the reply $Y[l] = (X[l])^e \bmod N$ at the time the hash query is made, meaning it prepares itself ahead of time for the possibility that $Msg[l]$ is later a sign query. Maybe it will not be, but nothing is lost.

Well, almost. Something is lost, actually. A reader who has managed to stay awake so far may notice that we have solved two problems: how to use F to find $y^d \bmod N$ where y is the input to I, and how to simulate answers to sign and hash queries of F, but that these processes are in conflict. The way we got $y^d \bmod N$ was by returning y as the answer to query (hash, M) where M is the message in the forgery. However, we do not know beforehand which message in a hash query will be the one in the forgery. So it is difficult to know how to answer a hash query $Msg[j]$; do we return y, or do we return $(X[j])^e \bmod N$ for some $X[j]$? If we do the first, we will not be able to answer a sign query with message $Msg[j]$; if we do the second, and if $Msg[j]$ equals the message in the forgery, we will not find the inverse of y. The answer is to take a guess as to which to do. There is some chance that this guess is right, and I succeeds in that case.

Specifically, notice that $Msg[q_{\text{hash}}] = M$ is the message in the forgery by definition since $Msg[q_{\text{hash}}]$ is the message in the final verification query. The message M might occur more than once in the list, but it occurs at least once. Now I will choose a random i in the range $1 \le i \le q_{\text{hash}}$ and respond by y to hash query $(\mathsf{hash}, Msg[i])$. To all other queries j it will respond by first picking $X[j]$ at random in Z_N^* and setting $H(Msg[j]) = (X[j])^e \bmod N$. The forged message M will equal $Msg[i]$ with probability at least $1/q_{\text{hash}}$ and this will imply Equation (11.2). Below we summarize these ideas as a proof of Theorem 11.3.1.

It is tempting from the above description to suggest that we always choose $i = q_{\text{hash}}$, since $Msg[q_{\text{hash}}] = M$ by definition. Why won't that work? Because M might also have been equal to $Msg[j]$ for some $j < q_{\text{hash}}$, and if we had set $i = q_{\text{hash}}$ then at the time we want to return y as the answer to M we find we have already defined $H(M)$ as something else and it is too late to change our minds.

Proof of Theorem 11.3.1: We first decribe I in terms of two subroutines: a hash oracle simulator $H\text{-}Sim(\cdot)$ and a sign oracle simulator $Sign\text{-}Sim(\cdot)$. It takes inputs N, e, y where $y \in \mathbf{Z}_N^*$ and maintains three tables, Msg, X and Y, each an array with index in the range from 1 to q_{hash}. It picks a random index i. All these are global variables which will be used also be the subroutines. The intended meaning of the array entries is the following, for $j = 1, \ldots, q_{\text{hash}}$–

$Msg[j]$	–	The j-th hash query in the experiment
$Y[j]$	–	The reply of the hash oracle simulator to the above, meaning the value playing the role of $H(Msg[j])$. For $j = i$ it is y.
$X[j]$	–	For $j \neq i$, the response to sign query $Msg[j]$, meaning it satisfies $(X[j])^e \equiv Y[j] \pmod N$. For $j = i$ it is undefined.

The code for the inverter is below.

Inverter $I(N, e, y)$

```
    Initialize arrays $Msg[1 \ldots q_{\text{hash}}]$, $X[1 \ldots q_{\text{hash}}]$, $Y[1 \ldots q_{\text{hash}}]$ to empty
    $j \leftarrow 0$ ; $i \xleftarrow{\$} \{1, \ldots, q_{\text{hash}}\}$
    Run $F$ on input $(N, e)$
    If $F$ makes oracle query $(\mathsf{hash}, M)$
        then $h \leftarrow \textit{H-Sim}(M)$ ; return $h$ to $F$ as the answer
    If $F$ makes oracle query $(\mathsf{sign}, M)$
        then $x \leftarrow \textit{Sign-Sim}(M)$ ; return $x$ to $F$ as the answer
    Until $F$ halts with output $(M, x)$
    $y' \leftarrow \textit{H-Sim}(M)$
    Return $x$
```

The inverter responds to oracle queries by using the appropriate subroutines. Once it has the claimed forgery, it makes the corresponding hash query and then returns the signature x.

We now describe the hash oracle simulator. It makes reference to the global variables instantiated in in the main code of I. It takes as argument a value v which is simply some message whose hash is requested either directly by F or by the sign simulator below when the latter is invoked by F.

We will make use of a subroutine Find that given an array A, a value v and index m, returns 0 if $v \notin \{A[1], \ldots, A[m]\}$, and else returns the smallest index l such that $v = A[l]$.

```
Subroutine $\textit{H-Sim}(v)$
    $l \leftarrow \text{Find}(Msg, v, j)$ ; $j \leftarrow j + 1$ ; $Msg[j] \leftarrow v$
    If $l = 0$ then
        If $j = i$ then $Y[j] \leftarrow y$
        Else $X[j] \xleftarrow{\$} Z_N^*$ ; $Y[j] \leftarrow (X[j])^e \bmod N$
        EndIf
        Return $Y[j]$
    Else
        If $j = i$ then abort
        Else $X[j] \leftarrow X[l]$ ; $Y[j] \leftarrow Y[l]$ ; Return $Y[j]$
        EndIf
    EndIf
```

The manner in which the hash queries are answered enables the following sign simulator.

```
Subroutine $\textit{Sign-Sim}(M)$
    $h \leftarrow \textit{H-Sim}(M)$
    If $j = i$ then abort
    Else return $X[j]$
    EndIf
```

Inverter I might abort execution due to the "abort" instruction in either subroutine. The first such situation is that the hash oracle simulator is unable to return y as the response to the i-th hash query because this query equals a previously replied to query. The second case is that F asks for the signature of the message which is the i-th hash query, and I cannot provide that since it is hoping the i-th message is the one in the forgery and has returned y as the hash oracle response.

Now we need to lower bound the ow-kea-advantage of I with respect to $\mathcal{K}_{\rm rsa}$. There are a few observations involved in verifying the bound claimed in Equation (11.2). First that the "view" of F at any time at which I has not aborted is the "same" as in Experiment $\mathbf{Exp}^{\rm uf\text{-}cma}_{\mathcal{DS}}(F)$. This means that the answers being returned to F by I are distributed exactly as they would be in the real experiment. Second, F gets no information about the value i that I chooses at random. Now remember that the last hash simulator query made by I is the message M in the forgery, so M is certainly in the array Msg at the end of the execution of I. Let $l = \mathrm{Find}(\mathit{Msg}, M, q_{\rm hash})$ be the first index at which M occurs, meaning $\mathit{Msg}[l] = M$ but no previous message is M. The random choice of i then means that there is a $1/q_{\rm hash}$ chance that $i = l$, which in turn means that $Y[i] = y$ and the hash oracle simulator won't abort. If x is a correct signature of M we will have $x^e \equiv Y[i] \pmod N$ because $Y[i]$ is $H(M)$ from the point of view of F. So I is successful whenever this happens. ▌

11.3.6 PSS0: A security improvement

The FDH-RSA signature scheme has the attractive security attribute of possessing a proof of security under the assumption that RSA is a one-way function, albeit in the random oracle model. However the quantitative security as given by Theorem 11.3.1 could be better. The theorem leaves open the possibility that one could forge signatures with a probability that is $q_{\rm hash}$ times the probability of being able to invert the RSA function at a random point, the two actions being measured with regard to adversaries with comparable execution time. Since $q_{\rm hash}$ could be quite large, say 2^{60}, there is an appreciable loss in security here. We now present a scheme in which the security relation is much tighter: the probability of signature forgery is not appreciably higher than that of being able to invert RSA in comparable time.

The scheme is called PSS0, for "probabilistic signature scheme, version 0", to emphasize a key aspect of it, namely that it is randomized: the signing algorithm picks a new random value each time it is invoked and uses that to compute signatures. The scheme $\mathcal{DS} = (\mathcal{K}_{\rm rsa}, \mathrm{Sig}, \mathrm{VF})$, like FDH-RSA, makes use of a public hash function H: $\{0,1\}^* \to Z_N^*$ which is modeled as a random oracle. Additonally it has a parameter s which is the length of the random value chosen by the signing algorithm. We write the signing and verifying algorithms as follows:

```
Algorithm $\mathrm{Sig}^{H(\cdot)}_{N,p,q,d}(M)$
    $r \xleftarrow{\$} \{0,1\}^s$
    $y \leftarrow H(r \,\|\, M)$
    $x \leftarrow y^d \bmod N$
    Return $(r, x)$

Algorithm $\mathrm{VF}^{H(\cdot)}_{N,e}(M, \sigma)$
    Parse $\sigma$ as $(r, x)$ where $|r| = s$
    $y \leftarrow H(r \,\|\, M)$
    If $x^e \bmod N = y$
        Then return 1 else return 0
```

Obvious "range checks" are for simplicity not written explicitly in the verification code; for example in a real implementation the latter should check that $1 \le x < N$ and $\gcd(x, N) = 1$.

This scheme may still be viewed as being in the "hash-then-invert" paradigm, except that the hash is randomized via a value chosen by the signing algorithm. If you twice sign the same message, you are likely to get different signatures. Notice that random value r must be included in the signature since otherwise it would not be possible to verify the signature. Thus unlike the previous schemes, the signature is not a member of Z_N^*; it is a pair one of whose components is an s-bit string and the other is a member of Z_N^*. The length of the signature is $s + k$ bits, somewhat longer than signatures for deterministic hash-then-invert signature schemes. It will usually suffice to set l to, say, 160, and given that k could be 1024, the length increase may be tolerable.

The success probability of a forger F attacking $\mathcal{DS}$ is measured in the random oracle model, via experiment $\mathbf{Exp}^{\rm uf\text{-}cma}_{\mathcal{DS}}(F)$. Namely the experiment is the same experiment as in the FDH-RSA case; only the scheme $\mathcal{DS}$ we plug in is now the one above. Accordingly we have the insecurity function associated to the scheme. Now we can summarize the security property of the PSS0 scheme.

Theorem 11.3.2 *Let $\mathcal{DS}$ be the PSS0 scheme with security parameters k and s. Let F be an adversary making $q_{\rm sig}$ signing queries and $q_{\rm hash} \geq 1 + q_{\rm sig}$ hash oracle queries. Then there exists an adversary I such that*

$$\mathbf{Adv}^{\rm uf\text{-}cma}_{\mathcal{DS}}(F) \quad \leq \quad \mathbf{Adv}^{\rm ow\text{-}kea}_{\mathcal{K}_{\rm rsa}}(I) + \frac{(q_{\rm hash} - 1) \cdot q_{\rm sig}}{2^s} \, . \blacksquare \tag{11.3}$$

Say $q_{\rm hash} = 2^{60}$ and $q_{\rm sig} = 2^{40}$. With $l = 160$ the additive term above is about 2^{-60}, which is very small. So for all practical purposes the additive term can be neglected and the security of the PSS0 signature scheme is tightly related to that of RSA.

We proceed to the proof of Theorem 11.3.2. The design of I follows the same framework used in the proof of Theorem 11.3.1. Namely I, on input N, e, y, will execute F on input N, e, and answer F's oracle queries so that F can complete its execution. From the forgery, I will somehow find $y^d \bmod N$. I will respond to hash oracle queries of F via a subroutine *H-Sim* called the hash oracle simulator, and will respond to sign queries of F via a subroutine *Sign-Sim* called the sign oracle simulator. A large part of the design is the design of these subroutines. To get some intuition it is helpful to step back to the proof of Theorem 11.3.1.

We see that in that proof, the multiplicative factor of $q_{\rm hash}$ in Equation (11.2) came from I's guessing at random a value $i \in \{1, \ldots, q_{\rm hash}\}$, and hoping that $i = \mathrm{Find}(\mathit{Msg}, M, q_{\rm hash})$ where M is the message in the forgery. That is, it must guess the time at which the message in the forgery is first queried of the hash oracle. The best we can say about the chance of getting this guess right is that it is at least $1/q_{\rm hash}$. However if we now want I's probability of success to be as in Equation (11.3), we cannot afford to guess the time at which the forgery message is queried of the hash oracle. Yet, we certainly don't know this time in advance. Somehow, I has to be able to take advantage of the forgery to return $y^d \bmod N$ nonetheless.

A simple idea that comes to mind is to return y as the answer to all hash queries. Then certainly a forgery on a queried message yields the desired value $y^d \bmod N$. Consider this strategy for FDH. In that case, two problems arise. First, these answers would then not be random and indpendent, as required for answers to hash queries. Second, if a message in a hash query is later a sign query, I would have no way to answer the sign query. (Remember that I computed its reply to hash query $\mathit{Msg}[j]$ for $j \neq i$ as $(X[j])^e \bmod N$ exactly in order to be able to later return $X[j]$ if $\mathit{Msg}[j]$ showed up as a sign query. But there is a conflict here: I can either do this, or return y, but not both. It has to choose, and in FDH case it chooses at random.)

The first problem is actually easily settled by a small algebraic trick, exploiting what is called the *self-reducibility* of RSA. When I wants to return y as an answer to a hash oracle query $\mathit{Msg}[j]$, it picks a random $X[j]$ in Z_N^* and returns $Y[j] = y \cdot (X[j])^e \bmod N$. The value $X[j]$ is chosen randomly and independently each time. Now the fact that $\mathsf{RSA}_{N,e}$ is a permutation means that all the different $Y[j]$ values are randomly and independently distributed. Furthermore, suppose $(M, (r, x))$ is a forgery for which hash oracle query $r \parallel M$ has been made and got the reponse $Y[l] = y \cdot (X[l])^e \bmod N$. Then we have $(x \cdot X[l]^{-1})^e \equiv y \pmod N$, and thus the inverse of y is $x \cdot X[l]^{-1} \bmod N$.

The second problem however, cannot be resolved for FDH. That is exactly why PSS0 pre-pends the random value r to the message before hashing. This effectively "separates" the two kinds of hash queries: the direct queries of F to the hash oracle, and the indirect queries to the hash oracle arising from the sign oracle. The direct hash oracle queries have the form $r \parallel M$ for some l-bit string r and some message M. The sign query is just a message M. To answer it, a value r is first chosen at random. But then the value $r \| M$ has low probability of having been a previous hash query. So at the time any new direct hash query is made, I can assume it will never be an indirect hash query, and thus reply via the above trick.

Here now is the full proof.

Proof of Theorem 11.3.2: We first decribe I in terms of two subroutines: a hash oracle simulator $H\text{-}Sim(\cdot)$ and a sign oracle simulator $Sign\text{-}Sim(\cdot)$. It takes input N, e, y where $y \in \mathbf{Z}_N^*$, and maintains four tables, R, V, X and Y, each an array with index in the range from 1 to q_{hash}. All these are global variables which will be used also be the subroutines. The intended meaning of the array entries is the following, for $j = 1, \ldots, q_{\text{hash}}$–

$V[j]$	–	The j-th hash query in the experiment, having the form $R[j] \| Msg[j]$
$R[j]$	–	The first l-bits of $V[j]$
$Y[j]$	–	The value playing the role of $H(V[j])$, chosen either by the hash simulator or the sign simulator
$X[j]$	–	If $V[j]$ is a direct hash oracle query of F this satisfies $Y[j] \cdot X[j]^{-e} \equiv y \pmod N$. If $V[j]$ is an indirect hash oracle query this satisfies $X[j]^e \equiv Y[j] \pmod N$, meaning it is a signature of $Msg[j]$.

Note that we don't actually need to store the array Msg; it is only referred to above in the explanation of terms.

We will make use of a subroutine Find that given an array A, a value v and index m, returns 0 if $v \notin \{A[1], \ldots, A[m]\}$, and else returns the smallest index l such that $v = A[l]$.

```
Inverter $I(N, e, y)$
    Initialize arrays $R[1 \ldots q_{\text{hash}}]$, $V[1 \ldots q_{\text{hash}}]$, $X[1 \ldots q_{\text{hash}}]$, $Y[1 \ldots q_{\text{hash}}]$, to empty
    $j \leftarrow 0$
    Run $F$ on input $N, e$
    If $F$ makes oracle query $(\mathsf{hash}, v)$
        then $h \leftarrow H\text{-}Sim(v)$ ; return $h$ to $F$ as the answer
    If $F$ makes oracle query $(\mathsf{sign}, M)$
        then $\sigma \leftarrow Sign\text{-}Sim(M)$ ; return $\sigma$ to $F$ as the answer
    Until $F$ halts with output $(M, (r, x))$
    $y \leftarrow H\text{-}Sim(r \| M)$ ; $l \leftarrow \text{Find}(V, r \| M, q_{\text{hash}})$
    $w \leftarrow x \cdot X[l]^{-1} \bmod N$ ; Return $w$
```

We now describe the hash oracle simulator. It makes reference to the global variables instantiated in in the main code of I. It takes as argument a value v which is assumed to be at least s bits long, meaning of the form $r \| M$ for some s bit strong r. (There is no need to consider hash queries not of this form since they are not relevant to the signature scheme.)

```
Subroutine $H\text{-}Sim(v)$
    Parse $v$ as $r \| M$ where $|r| = s$
    $l \leftarrow \text{Find}(V, v, j)$ ; $j \leftarrow j + 1$ ; $R[j] \leftarrow r$ ; $V[j] \leftarrow v$
    If $l = 0$ then
        $X[j] \stackrel{\$}{\leftarrow} Z_N^*$ ; $Y[j] \leftarrow y \cdot (X[j])^e \bmod N$ ; Return $Y[j]$
    Else
        $X[j] \leftarrow X[l]$ ; $Y[j] \leftarrow Y[l]$ ; Return $Y[j]$
    EndIf
```

Every string v queried of the hash oracle is put by this routine into a table V, so that $V[j]$ is the j-th hash oracle query in the execution of F. The following sign simulator does not invoke the hash simulator, but if necessary fills in the necessary tables itself.

```
Subroutine $Sign\text{-}Sim(M)$
    $r \stackrel{\$}{\leftarrow} \{0,1\}^s$
    $l \leftarrow \mathrm{Find}(R, r, j)$
    If $l \neq 0$ then abort
    Else
        $j \leftarrow j+1$ ; $R[j] \leftarrow r$ ; $V[j] \leftarrow r \,\|\, M$ ; $X[j] \stackrel{\$}{\leftarrow} Z_N^*$ ; $Y[j] \leftarrow (X[j])^e \bmod N$
        Return $X[j]$
    EndIf
```

Now we need to establish Equation (11.3).

First consider $\mathbf{Exp}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I)$ and let $\Pr_1[\cdot]$ denote the probability function in this experiment. Let BAD_1 be the event that I aborts due to the "abort" instruction in the sign-oracle simulator.

Now consider $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$, and let $\Pr_2[\cdot]$ denote the probability function in this experiment. Let BAD_2 be the event that the sign oracle picks a value r such that F had previously made a hash query $r \,\|\, M$ for some M.

Let SUCC be the event (in either experiment) that F succeeds in forgery. Now we have

$$\begin{aligned}
\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(F) &= \Pr_2[\text{SUCC}] \\
&= \Pr_2[\text{SUCC} \wedge \overline{\text{BAD}_2}] + \Pr_2[\text{SUCC} \wedge \text{BAD}_2] \\
&\leq \Pr_2[\text{SUCC} \wedge \overline{\text{BAD}_2}] + \Pr_2[\text{BAD}_2] \\
&= \Pr_1[\text{SUCC} \wedge \overline{\text{BAD}_1}] + \Pr_1[\text{BAD}_1] && (11.4) \\
&= \mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) + \Pr_1[\text{BAD}_1] && (11.5) \\
&\leq \mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) + \frac{(q_{\text{hash}}-1)q_{\text{sig}}}{2^s} \,. && (11.6)
\end{aligned}$$

This establishes Equation (11.3). Let us now provide some explanations for the above.

First, Equation (11.6) is justified as follows. The event in question happens if the random value r chosen in the sign oracle simulator is already present in the set $\{R[1], \ldots, R[j]\}$. This set has size at most $q_{\text{hash}} - 1$ at the time of a sign query, so the probability that r falls in it is at most $(q_{\text{hash}} - 1)/2^s$. The sign oracle simulator is invoked at most q_{sig} times, so the bound follows.

It is tempting to think that the "view" of F at any time at which I has not aborted is the "same" as the view of F in Experiment $\mathbf{Exp}^{\text{uf-cma}}_{\mathcal{DS}}(F)$. This is not true, because it can test whether or not BAD occured. That's why we consider bad events in both games, and note that

$$\mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) = \Pr_1[\text{SUCC} \wedge \overline{\text{BAD}_1}] = \Pr_2[\text{SUCC} \wedge \overline{\text{BAD}_2}] \,.$$

This is justified as follows. Remember that the last hash simulator query made by I is $r \,\|\, M$ where M is the message in the forgery, so $r \,\|\, M$ is certainly in the array V at the end of the execution of

I. So $l = \mathrm{Find}(V, r \parallel M, q_{\mathrm{hash}}) \neq 0$. We know that $r \parallel M$ was not put in V by the sign simulator, because F is not allowed to have made sign query M. This means the hash oracle simulator has been invoked on $r \parallel M$. This means that $Y[l] = y \cdot (X[l])^e \bmod N$ because that is the way the hash oracle simulator chooses its replies. The correctness of the forgery means that $x^e \equiv H(r \parallel M)$ (mod N), and the role of the H value here is played by $Y[l]$, so we get $x^e \equiv Y[l] \equiv y \cdot X[l]$ (mod N). Solving this gives $(x \cdot X[l]^{-1})^e \bmod N = y$, and thus the inverter is correct in returning $x \cdot X[l]^{-1} \bmod N$. ▮

It may be worth adding some words of caution about the above. It is tempting to think that

$$\mathbf{Adv}^{\text{ow-kea}}_{\mathcal{K}_{\text{rsa}}}(I) \;\geq\; \left[1 - \frac{(q_{\text{hash}} - 1) \cdot q_{\text{sig}}}{2^s}\right] \cdot \mathbf{Adv}^{\text{uf-cma}}_{\mathcal{DS}}(F)\;,$$

which would imply Equation (11.3) but is actually stronger. This however is not true, because the bad events and success events as defined above are not independent.

Chapter 12

Authenticated Key Exchange

Chapter 13

The Asymptotic Approach

Chapter 14

Interactive Proofs and Zero Knowledge

This chapter assumes that the reader has background in basic computational complexity theory. You should know about complexity classes like **P**, **NP**, **PSPACE**, **RP**, **BPP**. It also assumes background in basic cryptography, including computational number theory.

We fix the alphabet $\Sigma = \{0, 1\}$. A member of Σ^* is called a *string*, and the empty string is denoted ε. Objects upon which computation is performed, be they numbers, graphs or sets, are assumed to be appropriately encoded as strings. A *language* is a set of strings.

14.1 Introduction

Consider two parties, whom we will call the *prover* and the *verifier*, respectively. They have a common input, denoted x. They also might have individual, private inputs, called auxiliary inputs, with that of the prover denoted w, and that of the verifier denoted a. Each party also has access to a private source of random bits, called the party's coins. The parties exchange messages, and, at the end of the interaction, the verifier either accepts or rejects. Each party computes the next message it sends as a function of the common input, its auxiliary input, its coins, and the conversation so far.

The computational powers of the parties are important. The verifier must be "efficient." (As per complexity-theoretic conventions, this means it must be implementable by an algorithm running in time polynomial in the length of the common input.) Accordingly, it is required that the number of moves (meaning, message exchanges) be bounded by a polynomial in the length of the common input x. The computational power of the prover varies according to the setting and requirements, as we will see below.

We are interested in various goals for the interaction. Some are more important in complexity-theory, others in cryptography.

14.1.1 Proofs of language membership

The prover claims that x is a member of some fixed and understood underlying language L. The verifier is skeptical, but willing to be convinced. The purpose of the interaction is to convince the verifier.

In the formalization of the notion of an interactive proof of language membership, the verifier is the defining party. A language L is said to possess such a proof if there exists a verifier V satisfying two conditions. The first, called "completeness," asks that the verifier be open to being convinced of true claims, meaning that there exist a prover strategy P such that the interaction between P and V on common input $x \in L$ leads the verifier to accept. The second, called "soundness," asks that the verifier is able to protect itself against being convinced of false claims, meaning that for any prover strategy $\widehat{P}$, the interaction between P and V on common input $x \notin L$ leads the verifier to reject except with some "small" probability. (This is called the error-probability and is a parameter of the system.)

As an example, suppose L is the language SAT of satisfiable, boolean formulae. In that case the common input x is a boolean formula. The protocol consists of a single move, in which the prover supplies a string y that the verifier expects to be an assignment to the variables of the formula that makes the formula true. The verifier program simply evaluates x at y, accepting iff this value is one. If the formula is satisfiable, the prover can prove that this by sending the verifier a satisfying truth assignment y. If the formula x is unsatisfiable, there is no string y that can make the verifier accept.

The above is a very simple proof system in that the interaction consists of a single message from prover to verifier, and randomness is not used. This is called an **NP**-proof system. There are several reasons for which we are interested in proof systems that go beyond this, comprising many rounds of exchange and allowing randomness. On the complexity-theoretic side, they enable one to prove membership in languages outside **NP**. On the cryptographic side, they enable one to have properties like "zero-knowledge," to be discussed below.

Within the definitional template outlined above, there are various variants, depending on the computational power allowed to the prover in the two conditions. When no computational restrictions are put on the prover in the completeness and soundness conditions, one obtains what is actually called an interactive proof in the literature, a notion due to Goldwasser, Micali and Rackoff [20]. A remarkable fact is that **IP**, the class of languages possessing interactive proofs of membership, equals **PSPACE** [27, 34], showing that interaction and randomness extend language membership-proof capability well beyond **NP**.

Thinking of the prover's computation as one to be actually implemented in a cryptographic protocol, however, one must require that it be feasible. A stronger completeness requirement, that we call poly-completeness, is considered. It asks that there exist a prover P, running in time polynomial in the length of the common input, such that for every $x \in L$ there exists some auxiliary input w which, when given to P, enables the latter to make the verifier accept. The **NP**-proof system outlined above satisfies this condition because a satisfying assignment to x can play the role of the auxiliary input, and the prover simply transmits it.

The auxiliary input is important, since without it one would not expect a polynomial-time prover to be able to prove anything of interest. Why is it realistic? We imagine that the input x, for the sake of example a boolean formula, did not appear out of thin air, but was perhaps constructed by the prover in such a way that the latter knows a satisfying assignment, and makes claims about the satisfiability of the formula based on this knowledge.

The soundness condition can analogously be weakened to ask that it hold only with respect to polynomial-time provers $\widehat{P}$, having no auxiliary input. This poly-soundness condition is usually enough in cryptographic settings.

A proof system satisfying poly-completeness and poly-soundness is sometimes called a computationally sound proof system, or an argument [6], with a proof system satisfying completeness and soundness called a statistically sound proof system, or, as mentioned above, an interactive proof. The class of languages possessing proofs of membership satisfying poly-completeness and poly-soundness is a subset of **IP** and a superset of **NP**. We do not know any simple characterization of it. For cryptographic applications, it is typically enough that it contains **NP**.

Although poly-soundness suffices for applications, it would be a mistake to think that soundness is not of cryptographic interest. It often holds, meaning natural protocols have this property, and is technically easier to work with than poly-soundness. An example illustrating this is the fact that soundness is preserved under parallel composition, while poly-soundness is not [5].

The focus of these notes being cryptography, we will neglect the large and beautiful area of the computational complexity of interactive proof systems. But let us at least try to note some highlights. The first evidence as to the power of interactive proofs was provided by the fact that the language of non-isomorphic graphs, although not known to be in **NP**, possesses an interactive proof of membership [15]. Eventually, as noted above, it was found that **IP** = **PSPACE**. The related model of probabilistically checkable proofs has been applied to derive strong non-approximability results for **NP**-optimization problems, solving age-old open questions in algorithms.

14.1.2 Proofs of knowledge

A proof of knowledge for satisfiability enables a prover to convince the verifier that it "knows" a satisfying assignment to a formula x that is the common input to both parties. Here, whether or not the formula is satisfiable is not the issue, and indeed the parties start out knowing that it is. However, since SAT is (probably) not in **P**, the fact that x is satisfiable does not mean that someone can find a satisfying assignment to it. Someone might have one, someone else might not. The purpose of the interaction is for the prover to convince the verifier that it does have one.

Of course, one way to do this is for the prover to transmit its truth assignment to the verifier. But this often defeats other, security related requirements of the interaction. Thus, one seeks non-trivial ways to transfer evidence of "knowledge."

How might such a situation arise? We can imagine, as above, that formula x was created by a certain entity P in such a way that P knows a satisfying assignment. P makes x public, and will then prove its identity by proving that it "knows" a satisfying assignment to x. Impersonation is presumably difficult because others, although knowing x, would find it computationally infeasible to compute a satisfying assignemnt to it. A proof of membership in the language SAT will not do in such a situation, because the satisfiability status of x is not in doubt. Everyone, verifier included, knows that x is satisfiable. But not everyone knows a satisfying assignment for x, and it is of this knowledge that the verifier seeks to be convinced by the interaction.

A proof of knowledge is an intriguing object, and the first question one has to ask is what exactly this means and how it can be formalized. Indeed, it is not obvious what it means for a party to "know" something.

A proof of knowledge will apply to **NP**-relations. Meaning we consider a boolean-valued function ρ computable in time polynomial in the length of its first input, called an **NP**-relation because the language L_ρ, consisting of all x such that there exists y such that $\rho(x, y) = 1$, is in **NP**. On common input $x \in L_\rho$, the verifier is seeking to be convinced that the prover knows a string y such that $\rho(x, y) = 1$. In the example above, ρ would be the boolean formula evaluation relation which on inputs x, y returns 1 if y is a satisfying assignment to formula x, and 0 otherwise, and L_ρ would be SAT. But one can consider many other relations and corresponding languages.

As in a proof of language-membership, a proof of knowledge is defined by a verifier. There is a completeness condition, saying that there exists a prover strategy that makes the verifier accept on any input $x \in L_\rho$. The crux is a "knowledge" condition which asks that there exist a polynomial-time algorithm E called the extractor that, given oracle access to a prover strategy $\widehat{P}$, succeeds in outputting y satisfying $\rho(x, y) = 1$, with a probability related to the probability that $\widehat{P}$ would have convinced the verifier to accept. Thus, a party "knows" something if it is possible, given access to this party, to succeed in "extracting" and outputting the quantity in question. Think of the extractor as the subconcious mind of the prover.

What does it mean for the extractor to have oracle access to the prover? It can choose the latter's coins, and can interact with it, playing the role of verifier. The important element distinguishing the extractor from the verifier is that the former can execute the prover many times on the same random tape, thereby being able to get its responses to two different verifier messages on the same conversation prefix. Extraction exploits this ability.

There are various technical issues relating to the definition that we will have to consider later. One is about how the probability of success of the extractor on prover $\widehat{P}$ and input x relates to the probability that $\widehat{P}$ would convince the verifier to accept x. Typically, these differ by a quantity we call the knowledge-error probability, akin to the error-probability in the soundness condition of language membership. But other relations are possible as well, and the precise relation is important in applications. Another issue relates to the computational power allowed to the prover in the two conditions. The issues being quite analogous to those for proofs of language membership, they will not be discussed again here.

The idea of a proof of knowledge was suggested in [20], and formalizations were explored in [11, 12, 4].

14.1.3 Zero-knowledge and its cousins

If a prover sends a verifier a truth assignment to their common input formula x, it does more than convince the verifier that x is satisfiable: it gives that verifier useful information regarding the satisfiability of the formula. In particular, it enables the verifier to then prove to a third party that the same formula is satisfiable. The goal of a zero-knowledge protocol in this setting would be to not reveal to the verifier anything other than the fact that x is satisfiable, and in particular not enable the verifier, after having completed its interaction with the prover, to turn around and convince another polynomial-time party that x is satisfiable.

Although the above discussion was about proofs of language membership, the same applies to proofs of knowledge. One can have zero-knowledge proofs of language membership, or zero-knowledge proofs of knowledge.

The prover claims that some fact related to x is true, namely that x is a member of some fixed and understood underlying language L. The verifier is skeptical, but willing to be convinced. It asks the prover to supply evidence as to the truth of the claim, and eventually makes a decision, based on this evidence, regarding whether or not the claimed fact about x is true. We require that if $x \in L$ then it should be possible for the prover to convince the verifier, but if $x \notin L$, then, no matter what the prover does, the verifier will not be convinced. This is what we call a proof system.

The simplest type of proof system is an **NP** one. Here, the evidence supplied by the prover is a witness string y, and the verifier can check some relation between x and y in time polynomial in the length of x. For example if the underlying language is SAT, the input x would be a formula, and the witness could be a satisfying truth assignment. The verifier would evaluate x at the given truth assignment, and accept if and only if the answer is 1.

Interactive proofs extend **NP**-proof systems in two ways: they allow an exchange of messages, and they allow both parties to be randomized. We require that if $x \in L$ then it should be possible for the prover to convince the verifier, but if $x \notin L$, then, no matter what the prover does, the probability that the verifier is convinced is low. As we can see, "conviction" has become probabilistic attribute, because there is some, small probability that the verifier accepts when $x \notin L$.

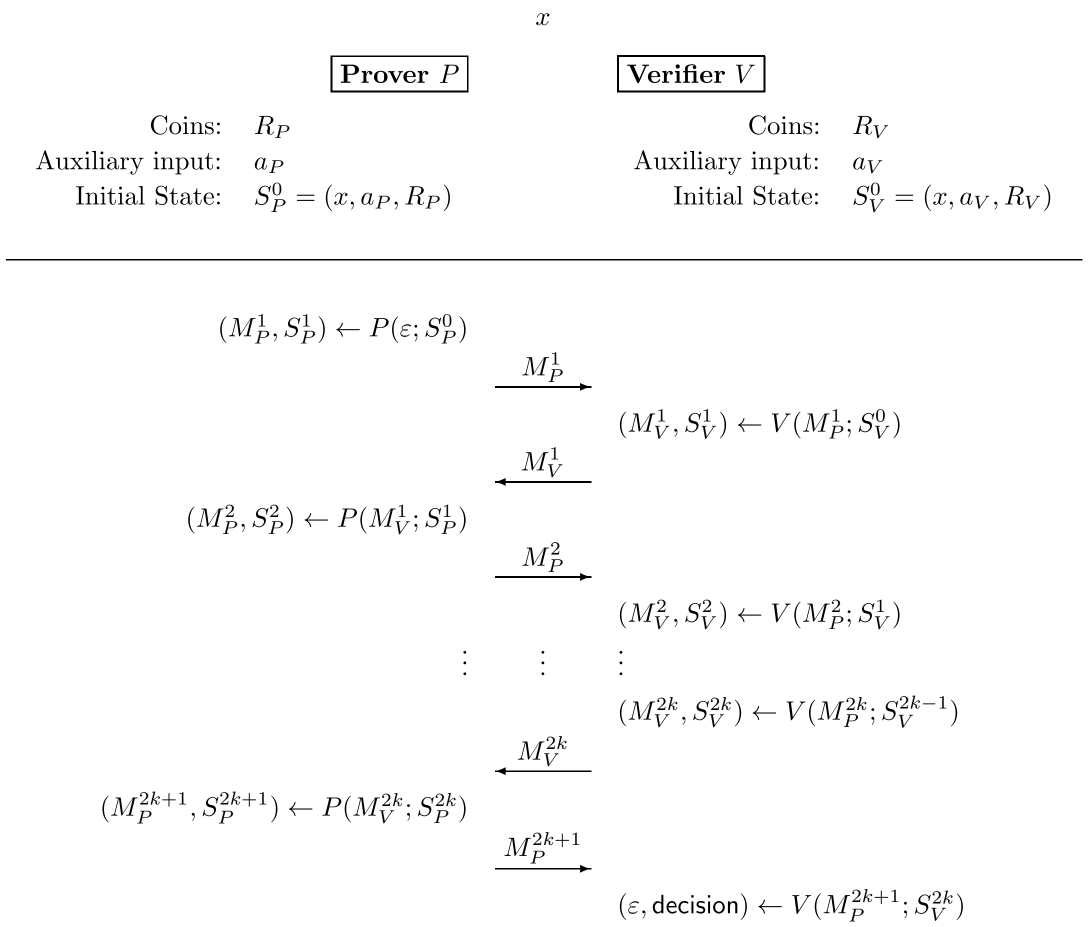
x
$\boxed{\textbf{Prover } P}$
$\boxed{\textbf{Verifier } V}$
Coins: R_P
Auxiliary input: a_P
Initial State: $S_P^0 = (x, a_P, R_P)$
Coins: R_V
Auxiliary input: a_V
Initial State: $S_V^0 = (x, a_V, R_V)$
$(M_P^1, S_P^1) \leftarrow P(\varepsilon; S_P^0)$
$\xrightarrow{M_P^1}$
$(M_V^1, S_V^1) \leftarrow V(M_P^1; S_V^0)$
$\xleftarrow{M_V^1}$
$(M_P^2, S_P^2) \leftarrow P(M_V^1; S_P^1)$
$\xrightarrow{M_P^2}$
$(M_V^2, S_V^2) \leftarrow V(M_P^2; S_V^1)$
$\vdots \quad \vdots \quad \vdots$
$(M_V^{2k}, S_V^{2k}) \leftarrow V(M_P^{2k}; S_V^{2k-1})$
$\xleftarrow{M_V^{2k}}$
$(M_P^{2k+1}, S_P^{2k+1}) \leftarrow P(M_V^{2k}; S_P^{2k})$
$\xrightarrow{M_P^{2k+1}}$
$(\varepsilon, \mathsf{decision}) \leftarrow V(M_P^{2k+1}; S_V^{2k})$

Figure 14.1: Prover-initiated interaction between a pair of interactive functions P, V on common input x, consisting of $m = 2k + 1$ moves. The verifier's decision is $\mathsf{decision} \in \{\mathsf{accept}, \mathsf{reject}\}$.

14.2 Interactive functions and the accepting probability

We will be considering a pair of interacting parties, called the prover and verifier, respectively. We will be interested in specific strategies, or algorithms, that they might use in computing the messages to send to each other. To this end, each party will be modeled by what we call an *interactive function*. Formally, an interactive function I takes the following inputs: An *incoming message*, and the *current state*. Denoting these by M_{in}, S, respectively, the output, denoted $I(M_{\text{in}}; S)$ is a pair (M_{out}, N) consisting of an *outgoing message* and the *next state*.

Now, imagine a pair P, V of interactive functions, representing the prover and verifier, respectively. They will have a common input, denoted x. They might have different auxiliary inputs. Let us denote that of P by a_P and that of V by a_V. They would also have different random tapes, with that of P denoted by R_P and that of V denoted by R_V. Their interaction begins with some initialization. Our convention is that the initial state provided to an interactive function is the common input, its auxiliary input, and its random tape. Thus P would have initial state $S_P^0 = (x, a_P, R_P)$ while V would have initial state $S_V^0 = (x, a_V, R_V)$. Once initialization has been completed and an initiating party has been selected, a sequence of messages, making up a conversation, is determined.

A current state is maintained by each party, and, when this party receives a message, it uses its interactive function to determine the next message to the other party, as well as a new, updated state to maintain, as a function of the message it received and its current state. The exchange continues for some pre-determined number of moves $m(\cdot)$, the latter being a polynomially-bounded, polynomial-time computable function of the length n of the common input. The last message is sent by the prover, and at the end of it, V must enter one of the special states accept or reject.

Fig. 14.1 illustrates the message exchange process for the case of a prover-initiated interaction. A verifier-initiated interaction proceeds analogously with the first message being sent by the verifier. The number of moves m is odd when the interaction is prover-initiated, and even when the interaction is verifier-initiated.

The random tape of a party is the only source of randomness for this party's interactive function, and the manner in which randomness enters the party's computation. The length of the random tape R_I for an interactive function I is a function $r_I(\cdot)$ of the length n of the common input.

Notice that once the interactive functions P, V have been chosen and their initial states (which include their random tapes) have been chosen, the sequence of messages between them, and the verifier's decision, are determined, meaning these are deterministic functions of the initial states. We let

$$\mathbf{Decision}^{P,a_P,R_P}_{V,a_P,R_V}(x)$$

denote this decision. This is the value, either accept or reject, given by the final state of V in the interaction with P in which the initial state of V is (x, a_V, R_V) and the initial state of P is (x, a_P, R_P). This leads to an important quantity, the *accepting probability*, defined as

$$\mathbf{Acc}^{P,a_P}_{V,a_V}(x) \;=\; \Pr\left[\mathbf{Decision}^{P,a_P,R_P}_{V,a_P,R_V}(x) = \mathsf{accept}\right] \;, \tag{14.1}$$

the probability being over the random choices of R_P and R_V. In more detail, the quantity of Equation (14.1) is the probability that the following experiment returns 1:

```
$n \leftarrow |x|$
$R_P \stackrel{\$}{\leftarrow} \{0,1\}^{r_P(n)}$ ; $R_V \stackrel{\$}{\leftarrow} \{0,1\}^{r_V(n)}$
decision $\leftarrow \mathbf{Decision}^{P,a_P,R_P}_{V,a_V,R_V}(x)$
If decision = accept then return 1 else return 0
```

The time complexity of an interactive function is measured as a function of the length n of the common input. In particular, interactive function I is said to be *polynomial time* if it is computable in time polynomial in the length of the common input. Henceforth, a *verifier* is a polynomial-time interactive function. When I is polynomial-time, it is assumed that r_I is polynomially-bounded and polynomial-time computable.

14.3 Proofs of language-membership

A language L has been fixed and agreed upon. We consider interactions in which the goal of the prover is to convince the verifier that their common input is a member of the language in question. In such a system, the central object is the verifier. This is a particular, polynomial-time interactive function V whose specification defines the system. Two classes of conditions of V relative to L are considered: completeness, or liveness conditions, and soundness, or security conditions. We discuss them briefly, then provide a formal definition with a few variants of each condition, and then provide a more in-depth discussion and comparision.

14.3.1 The basic definitions

A completeness condition asks that the verifier be open to being convinced of true facts. Namely, for inputs x in the language, there should exist an interactive function P whose interaction with V on common input x leads V to an accept decision. Different formalizations emerge depending on the time-complexity required of P.

A soundness condition asks that the verifier be capable of protecting itself against accepting false claims, meaning that when the common input x is not in the language, the probability of leading V to an accept decision should be "low." It is important that this condition hold regardless of the strategy used by the prover, so that it should hold for any interactive function $\widehat{P}$. Again, different formalizations emerge depending on the time-complexity of $\widehat{P}$.

Definition 14.3.1 Let V be a polynomial-time interactive function, which we call the verifier. Let L be a language, and let δ: $\mathbf{N} \to [0,1]$ be a function. We consider the following conditions on V relative to L and δ:

1. **True-completeness:** There exists an interactive function P such that
$$\forall x \in L \ \ \forall a_V \in \Sigma^* \quad : \quad \mathbf{Acc}_{V,a_V}^{P,\varepsilon}(x) = 1 \ .$$
2. **Poly-completeness:** There exists a polynomial-time interactive function P such that
$$\forall x \in L \ \ \exists a_P \in \Sigma^* \ \ \forall a_V \in \Sigma^* \quad : \quad \mathbf{Acc}_{V,a_V}^{P,a_P}(x) = 1 \ .$$
3. **True-soundness** with error-probability δ: For all interactive functions $\widehat{P}$
$$\forall x \notin L \ \ \forall a_V \in \Sigma^* \quad : \quad \mathbf{Acc}_{V,a_V}^{\widehat{P},\varepsilon}(x) \leq \delta(|x|) \ .$$
4. **Poly-soundness** with error-probability δ: For all polynomial-time interactive functions $\widehat{P}$
$$\exists N \in \mathbf{N} \ \ \forall x \notin L \ \ \forall a_V \in \Sigma^* \quad : \quad \left[\ |x| \geq N \ \Rightarrow \ \mathbf{Acc}_{V,a_V}^{\widehat{P},\varepsilon}(x) \leq \delta(|x|) \ \right] \ .$$

The prover P of a completeness condition is called the *honest prover*, while a prover being considered for a soundness condition is called a *cheating prover*.

Next we define classes of languages related to these conditions.

Definition 14.3.2 Let L be a language. We say that L has an interactive true-proof (of membership) if there exists a polynomial-time interactive function V such that the following conditions hold: true-completeness, and true-soundness with error-probability 1/2. We let **IP** denote the class of all languages possessing interactive true-proofs of membership.

Definition 14.3.3 Let L be a language. We say that L has an interactive poly-proof (of membership) if there exists a polynomial-time interactive function V such that the following conditions hold: poly-completeness, and poly-soundness with error-probability 1/2. We let **pIP** denote the class of all languages possessing interactive poly-proofs of membership.

Before we discuss the definitions further, let us look at a simple class of examples.

14.3.2 NP proof systems

The bulk of cryptographic applications pertain to languages in **NP**. Recall that a language L is in the class **NP** if there exist efficiently verifiable certificates for membership in L.

Definition 14.3.4 An **NP**-*relation* is a boolean-valued function $\rho(\cdot,\cdot)$, computable in time polynomial in the length of its first input. A language L is in the class **NP** if there exists an **NP**-relation ρ such that

$$L \;=\; \{\, x \in \Sigma^* \;:\; \exists y \in \Sigma^* \text{ such that } \rho(x,y) = 1 \,\}\;.$$

In this case, ρ is called an **NP**-*relation for* L. When ρ is understood, a string y such that $\rho(x,y)$ is called a certificate, or witness, to the membership of x in L.

An example is the language SAT consisting of all satisfiable boolean formulae. The associated **NP**-relation is $\rho(\varphi,y)=1$ iff y is a satisfying assignment to formula φ.

If $L \in$ **NP** then it has a very simple associated proof system. The protocol has just one move, from prover to verifier. The prover is expected to supply a certificate for the membership of the common input in the language, and the verifier checks it. Let us detail this, to make sure we see how to fit the models and definitions we have provided above.

Proposition 14.3.5 Suppose $L \in$ **NP**. Then there exists a verifier V defining a one-move protocol satisfying poly-completeness, and true-soundness with error-probability zero, with respect to L.

Note that this shows that **NP** $\subseteq$ **IP** and also **NP** $\subseteq$ **pIP**. Why? Because poly-completeness implies completeness, and soundness implies poly-soundness, as we will see below.

Proof of Proposition 14.3.5: We need to specify the interactive function V. We set $r_V(\cdot)=0$, meaning the verifier uses no coins. We let ρ be an **NP**-relation for L. Now, the description of V is the following:

```
Verifier $V(M;S)$
    Parse $S$ as $(x, a_V, \varepsilon)$
    If $\rho(x,M)=1$ then decision ← accept else decision ← reject
    Return $(\varepsilon, \mathsf{decision})$
```

The initial state of the verifier is the only state it can maintain, and this contains the common input x, an auxiliary input a_V, and the emptystring representing a random tape of length 0. The verifier treats the incoming message as a witness for the membership of x in L, and evaluates $\rho(x,M)$ to verify that this is indeed so. The outgoing message is ε, since the verifier takes its decision after receiving its first message. The verifier is polynomial-time because ρ is computable in time polynomial in the length of its first input.

Let us now show that poly-soundness holds. To do this we must specify an interactive function P meeting condition **2** of Definition 14.3.1. It is deterministic, meaning $r_P(\cdot)=0$. Its action is simply to transmit its auxiliary input to the verifier. Formally:

```
Prover $P(\varepsilon;S)$
    Parse $S$ as $(x, a_P, \varepsilon)$
    Return $(a_P, \varepsilon)$
```

Now we must check that this satisfies the conditions. Assume $x \in L$. Then, since ρ is an **NP**-relation for L, there exists a y such that $\rho(x,y)=1$. Setting $a_P = y$, we see that

$$\mathbf{Acc}_{V,a_V}^{P,a_P}(x) = 1$$

for all $a_V \in \Sigma^*$.

Finally, we show that true-soundness with error-probability zero holds. Let $\widehat{P}$ be an arbitrary interactive function, and assume $x \notin L$. In that case, there is no string y such that $\rho(x, y) = 1$, and thus no message M that could lead V to accept. ▌

14.3.3 Discussion of the definitions

Completeness in the absence of soundness is uninteresting. For any language L, there exists a verifier satisfying both types of completeness: the verifier V that always accepts. Similarly, soundness in the absence of completeness or poly-completeness is uninteresting. For any language L, there exists a verifier satisfying both types of soundness with error-probability zero: the verifier V that always rejects. It is only in combination that the two types of conditions become interesting.

Note that the auxiliary input of the prover in the true-completeness condition is set to the empty string ε, but in the poly-completeness condition the prover gets an auxiliary input that is allowed to depend on the common input. This is crucial because without this, the prover would be a polynomial-time interactive function just like the verifier, and the class of languages satisfying this weakened poly-completeness condition and the poly-soundness condition would be just **BPP**, and thus not interesting as a model of provability. The following problem asks you to verify this, and doing so is good to exercise your understanding of the definitions.

Problem 44 Let V be a verifier and L a language. Consider the condition:

- **Weak-poly-completeness:** There exists a polynomial-time interactive function P such that
$$\forall x \in L \;\; \forall a_V \in \Sigma^* \quad : \quad \mathbf{Acc}_{V,a_V}^{P,\varepsilon}(x) = 1 \; .$$

Let L be a language. We say that L has an interactive weak-poly-proof (of membership) if there exists a polynomial-time interactive function V such that the following conditions hold: weak-poly-completeness, and poly-soundness with error-probability 1/2. We let **wpIP** denote the class of all languages possessing interactive weak-poly-proofs of membership. Prove that **wpIP** = **BPP**.

Be careful with your arguments above. In particular, how exactly are you using the poly-soundness condition?

Another worthwhile exercise at this point is to prove the following relation, which says that poly-completeness is a stronger requirement than true-completeness.

Problem 45 Let V be a verifier satisfying poly-completeness relative to language L. Prove that V also satisfies true-completeness with respect to language L.

The reason this is true is of course the fact that the prover in the true-completeness condition is not computationally restricted, but the reason it is worth looking at more closely is to make sure that you take into account the auxiliary inputs of *both* parties and the roles they play.

We have denied the prover an auxiliary input in both the soundness conditions. True-soundness is not affected by whether or not the prover gets an auxiliary input, but poly-soundness might be, and we have chosen a simple formulation. Notice that poly-soundness is a weaker requirement than true-soundness in the sense that true-soundness implies poly-soundness.

The auxiliary input of the verifier will be used to model its history, meaning information from prior interactions. This is important for zero-knowledge, but in the interactive proof context you can usually ignore it, and imagine that $a_V = \varepsilon$.

Many natural protocols have an error-probability of $\delta(n) = 1/2$ in the soundness condition. One way to lower this is by independent repetitions of the protocol. As we will see later, the extent to which this is effective depends on the type of soundness condition (whether true or poly).

REMOVED DUE TO PDFLATEX PROBLEM

Figure 14.2: Known containment relations amongst some interactive proof related complexity classes. A line indicates that the class at the lower end of the line is contained in the class at the upper end of the line.

The requirement that the acceptance probability in the completeness or poly-completeness conditions be 1 can be relaxed. The class **IP** of Definition 14.3.2 would be unchanged had we required completeness with probability, say, 2/3. Whether the class **pIP** of Definition 14.3.3 would also remain unchanged had we required poly-completeness with probability 2/3 is not known. However, natural protocols seem to satisfy completeness or poly-completeness with probability 1, and this is simpler to work with.

The poly-soundness condition can do with some elucidation. It asks that if we fix a polynomial-time prover $\widehat{P}$, then the probability that $\widehat{P}$ can convince V to accept an $x \notin L$ is small, but *only if x is long enough.* For any $\widehat{P}$ there might exist a finite, but large set of inputs $x \notin L$ which the verifier does accept when talking to $\widehat{P}$. This

Fig. 14.2 illustrates relations between the complexity classes we have just defined and some standard ones, and it is worthwhile to pause and verify these relations.

14.3.4 Quadratic residuosity

To do so we must first recall some number theory. Let N be a positive integer. An element $S \in Z_N^*$ is a square, or quadratic residue, if it has a square root modulo N, namely there is a $s \in Z_N^*$ such that $s^2 \equiv S \bmod N$. If not, it is a non-square or non-quadratic-residue. Note a number may have lots of square roots modulo N.

$$\begin{array}{lcl}
\text{Prover} & & \text{V} \\
\text{Pick } r \stackrel{\$}{\leftarrow} Z_N^* & & \\
\text{Let } y \leftarrow r^2 \bmod N & & \\
 & \xrightarrow{\ y\ } & \\
 & & \text{Pick } c \stackrel{\$}{\leftarrow} \{0,1\} \\
 & \xleftarrow{\ c\ } & \\
z \leftarrow rs^c \bmod N & & \\
 & \xrightarrow{\ z\ } & \\
 & & \text{Accept iff } z^2 \equiv yx^c \bmod N
\end{array}$$

We must first recall some number theory. Let N be a positive integer. An element $S \in Z_N^*$ is a square, or quadratic residue, if it has a square root modulo N, namely there is a $s \in Z_N^*$ such that $s^2 \equiv S \bmod N$. If not, it is a non-square or non-quadratic-residue. The quadratic residuosity language is

$$\mathsf{QR} \ = \ \{\, (N, S) \ : \ S \text{ is a quadratic residue modulo } N \,\}\ .$$

Note a number may have lots of square roots modulo N. Recall that there exists a polynomial time algorithm to compute the gcd, meaning on inputs N, s it returns $\gcd(N, s)$. There also exists a polynomial time algorithm that on inputs N, S, s can check that $s^2 \equiv S \pmod{N}$. However, there is no known polynomial-time algorithm, even randomized, that on input N, S returns a square root of S modulo N.

Now imagine that the common input x to the prover and verifier is a pair (N, S). We consider various possible requirements of the protocol between the parties, and the motivations for these requirements.

The prover claims that S is a quadratic residue modulo N, meaning it claims that (N, S) is a member of the language QR.

14.4 NP proof-systems

Recall that there exists a polynomial time algorithm to compute the gcd, meaning on inputs N, s it returns $\gcd(N, s)$. There also exists a polynomial time algorithm that on inputs N, S, s can check that $s^2 \equiv S \pmod{N}$. However, there is no known polynomial-time algorithm, even randomized, that on input N, S returns a square root of S modulo N.

Example 14.4.1 We claim that QR is in **NP**. To justify this we must present an **NP**-relation ρ for QR. The relation in question is defined by

$$\rho((N,S),s) \;=\; \begin{cases} 1 & \text{if } S, s \in \mathbf{Z}_N^* \text{ and } s^2 \equiv S \pmod{N} \\ 0 & \text{otherwise.} \end{cases}$$

The facts recalled above tell us that ρ is computable in time polynomial in the length of the pair N, S. This involves two gcd computations and one squaring operation. So ρ is an **NP**-relation.

Now suppose $L \in \mathbf{NP}$ and let ρ be an **NP** relation for L. Imagine a party that has an input x and wants to know whether or not this input is in L. It probably cannot do this efficiently, since **NP** is probably different from **P**. Now imagine that there is another party that is willing to help out. We call this party the *prover*. It has, for some reason, the ability to determine whether or not x is in L. Our original party is willing to take the prover's help, but does not trust the prover. It asks that if the prover claims that $x \in L$, it should supply evidence to this effect, and the evidence should be efficiently verifiable, where "efficient" means in time polynomial in the length of x. Accordingly, we call our original party a *verifier*. Given that $L \in \mathbf{NP}$, the evidence can take the form of a witness y satisfying $\rho(x, y) = 1$, where ρ is an **NP**-relation for L. The verifier would compute ρ to check this evidence.

We thus visualize a game involving two parties, a prover and a verifier, having a common input x. The verifier is computationally restricted, specifically to run in time polynomial in the length n of the input x, but no computation restrictions are put on the prover. The prover claims that x is a member of the underlying language L. To prove its claim, the prover transmits to the verifier a string y. The latter evaluages $\rho(x, y)$, and accepts if and only if this value is 1. This is an **NP**-proof system.

Let us formalize this. We say that a language L has an **NP**-proof system if there exists an algorithm V, called the verifier, that is computable in time polynomial in its first input, and for which two conditions hold. The first, called completeness, says that there exists a function P such that, if the verifier is supplied the message $y = P(x)$, then it accepts. The second condition, called soundness

testing whether x is in L by running some decision procedure, but, being restricted to polynomial time, this will only lend it certainity if $L \in \mathbf{P}$. However, the verifier is willing to allow the prover to supply evidence, or "proof" or its claim. The prover is asked to supply a string y

Theorem 14.4.2 **IP = PSPACE**.

14.5 Exercises and Problems

Appendix A

The Birthday Problem

The setting is that we have q balls. View them as numbered, $1, \ldots, q$. We also have N bins, where $N \geq q$. We throw the balls at random into the bins, one by one, beginning with ball 1. At random means that each ball is equally likely to land in any of the N bins, and the probabilities for all the balls are independent. A collision is said to occur if some bin ends up containing at least two balls. We are interested in $C(N, q)$, the probability of a collision.

The birthday paradox is the case where $N = 365$. We are asking what is the chance that, in a group of q people, there are two people with the same birthday, assuming birthdays are randomly and independently distributed over the days of the year. It turns out that when q hits $\sqrt{365}$ the chance of a birthday collision is already quite high, around $1/2$.

This fact can seem surprising when first heard. The reason it is true is that the collision probability $C(N, q)$ grows roughly proportional to q^2/N. This is the fact to remember. The following gives a more exact rendering, providing both upper and lower bounds on this probability.

Theorem A.0.1 ***[Birthday bound]*** *Let $C(N, q)$ denote the probability of at least one collision when we throw $q \geq 1$ balls at random into $N \geq q$ buckets. Then*

$$C(N,q) \;\leq\; \frac{q(q-1)}{2N}$$

and

$$C(N,q) \;\geq\; 1 - e^{-q(q-1)/2N} \; .$$

Also if $1 \leq q \leq \sqrt{2N}$ then

$$C(N,q) \;\geq\; 0.3 \cdot \frac{q(q-1)}{N} \; . \blacksquare$$

In the proof we will find the following inequalities useful to make estimates.

Proposition A.0.2 The inequality

$$\left(1 - \frac{1}{e}\right) \cdot x \;\leq\; 1 - e^{-x} \;\leq\; x \; .$$

is true for any real number x with $0 \leq x \leq 1$. $\blacksquare$

Proof of Theorem A.0.1: Let C_i be the event that the i-th ball collides with one of the previous ones. Then $\Pr[C_i]$ is at most $(i-1)/N$, since when the i-th ball is thrown in, there are at most $i-1$ different occupied slots and the i-th ball is equally likely to land in any of them. Now

$$\begin{aligned} C(N,q) &= \Pr[C_1 \vee C_2 \vee \cdots \vee C_q] \\ &\le \Pr[C_1] + \Pr[C_2] + \cdots + \Pr[C_q] \\ &\le \frac{0}{N} + \frac{1}{N} + \cdots + \frac{q-1}{N} \\ &= \frac{q(q-1)}{2N} \,. \end{aligned}$$

This proves the upper bound. For the lower bound we let D_i be the event that there is no collision after having thrown in the i-th ball. If there is no collision after throwing in i balls then they must all be occupying different slots, so the probability of no collision upon throwing in the $(i+1)$-st ball is exactly $(N-i)/N$. That is,

$$\Pr[D_{i+1} \mid D_i] \;=\; \frac{N-i}{N} \;=\; 1 - \frac{i}{N} \,.$$

Also note $\Pr[D_1] = 1$. The probability of no collision at the end of the game can now be computed via

$$\begin{aligned} 1 - C(N,q) &= \Pr[D_q] \\ &= \Pr[D_q \mid D_{q-1}] \cdot \Pr[D_{q-1}] \\ &\vdots \quad \vdots \\ &= \prod_{i=1}^{q-1} \Pr[D_{i+1} \mid D_i] \\ &= \prod_{i=1}^{q-1} \left(1 - \frac{i}{N}\right) \,. \end{aligned}$$

Note that $i/N \le 1$. So we can use the inequality $1 - x \le e^{-x}$ for each term of the above expression. This means the above is not more than

$$\prod_{i=1}^{q-1} e^{-i/N} \;=\; e^{-1/N-2/N-\cdots-(q-1)/N} \;=\; e^{-q(q-1)/2N} \,.$$

Putting all this together we get

$$C(N,q) \;\ge\; 1 - e^{-q(q-1)/2N} \,,$$

which is the second inequality in Proposition A.0.1. To get the last one, we need to make some more estimates. We know $q(q-1)/2N \le 1$ because $q \le \sqrt{2N}$, so we can use the inequality $1 - e^{-x} \ge (1 - e^{-1})x$ to get

$$C(N,q) \;\ge\; \left(1 - \frac{1}{e}\right) \cdot \frac{q(q-1)}{2N} \,.$$

A computation of the constant here completes the proof. ▌

Appendix B

Information-Theoretic Security

Chapter to be absorbed elsewhere.

We discuss the information-theoretic notion of security called perfect security which we will show is possessed by the one-time-pad scheme.

We fix a particular symmetric encryption scheme $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. Two parties share a key K for this scheme and the adversary does not a priori know K. The adversary is assumed able to capture any ciphertext that flows on the channel between the two parties. Having captured a ciphertext, it attempts to glean information about the corresponding plaintext message.

Take for example the one-time-pad scheme, and assume a single k-bit message is encrypted and transmitted, where k is the length of the key. Due to the random choice of the key (pad), this certainly seems very "secure." We would like to say that the adversary, given the ciphertext, has "no idea" what the message was. But it is not clear how to say this, or if it is even really true. The adversary could always guess the message. Or, it could have a pretty good idea what the message was from some context surrounding the encryption. For example, it may know that the first few bytes of the message is a packet header containing the sender's (known) IP address.

So we can't really say the adversary has no idea what the message is given the ciphertext. Instead, we adopt a comparative measure of security. We are interested in how much more the adversary knows about the message given the ciphertext as opposed to what it knew before it saw the ciphertext. Perfect security holds if "the adversary's best guess as to the message after having seen the ciphertext is the same as before it saw the ciphertext." In other words, the ciphertext was no help in figuring out anything new about the message.

This is captured this as follows. We assume a single message will be encrypted, and are interested only in the security of this encryption. There is some plaintext space $\mathsf{Plaintexts} \subseteq \{0,1\}^*$ of messages that the encryptor is willing to encrypt. (For example, with the one-time pad scheme, if the key length is k bits then $\mathsf{Plaintexts} = \{0,1\}^k$.) Notice that this effectively makes the scheme stateless.

We model the a priori information (the information the adversary already possesses about the message) as a probability distribution on the set of possible messages. Formally, a *message distribution* on $\mathsf{Plaintexts}$ is a function D: $\mathsf{Plaintexts} \to [0,1]$ such that

$$\sum_{M \in \mathsf{Plaintexts}} D(M) = 1 \ ,$$

and also $D(M) > 0$ for all $M \in \mathsf{Plaintexts}$. For example, there might be four messages, $00, 01, 10, 11$, with

$$D(00) = 1/6, \quad D(01) = 1/3, \quad D(10) = 1/4, \quad \text{and} \quad D(11) = 1/4 \ .$$

We imagine that the sender chooses a message at random according to D, meaning that a specific message $M \in \mathsf{Plaintexts}$ has probability $D(M)$ of being chosen. In our example, the sender would choose 00 with probability 1/6, and so on.

The message distribution, and the fact that the sender chooses according to it, are known to the adversary. Before any ciphertext is transmitted, the adversary's state of knowledge about the message chosen by the sender is given by D. That is, it knows that the message was 00 with probability 1/6, and so on.

We say that the encryption scheme is perfectly secure if the possession of the ciphertext does not impart any *additional* information about the message than was known a priori via the fact that it was chosen according to D. The setup is like this. After the sender has chosen the message according to D, a key K is also chosen, according to the key generation algorithm, meaning $K \leftarrow \mathcal{K}$, and the message is encrypted to get a ciphertext, via $C \leftarrow \mathcal{E}_K(M)$. The adversary is given C. We ask the adversary: given that you know C is the ciphertext produced, for each possible value of the message, what is the probability that that particular value was actually the message chosen? If the adversary can do no better than say that the probability that M was chosen was $D(M)$, it means that the possession of the ciphertext is not adding any new information to what is already known. This is perfect security.

To state this more formally, we first let

$$S \;=\; \mathsf{Keys}(\mathcal{SE}) \times \mathsf{Plaintexts} \times \{0,1\}^r$$

denote the sample space underlying our experiment. Here r is the number of coins the encryption algorithm tosses. (This is zero if the encryption algorithm is deterministic, as is the case for the one-time pad.) We let introduce the following random variables:

$$\begin{array}{lll} \mathsf{K}\colon\ S \to \mathsf{Keys}(\mathcal{SE}) & \text{defined by} & (K,M,R) \mapsto K \\ \mathsf{M}\colon\ S \to \mathsf{Plaintexts} & \text{defined by} & (K,M,R) \mapsto M \\ \mathsf{C}\colon\ S \to \{0,1\}^* & \text{defined by} & (K,M,R) \mapsto \mathcal{E}_K(M;R) \end{array}$$

Thus K simply returns the value of the chosen key while M returns the value of the chosen message. The last random variable returns the encryption of the message using key K and coins R. The probability distribution underlying this sample space is denoted $\Pr_{D,\mathcal{SE}}[\cdot]$ and is given by a choice of K as per $\mathcal{K}$, a choice of M as per D, and a random choice of R, all these being made independently.

Definition B.0.1 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme with associated message space $\mathsf{Plaintexts}$. Let $D\colon \mathsf{Plaintexts} \to [0,1]$ be a message distribution on $\mathsf{Plaintexts}$. We say that $\mathcal{SE}$ is *perfectly secure with respect to* D if for every $M \in \mathsf{Plaintexts}$ and every possible ciphertext C it is the case that

$$\Pr_{D,\mathcal{SE}}[\mathsf{M} = M \mid \mathsf{C} = C] \;=\; D(M)\,. \tag{B.1}$$

We say that $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ is *perfectly secure* if it is perfectly secure with respect to *every* message distribution on $\mathsf{Plaintexts}$.

Here "$\mathsf{M} = M$" is the event that the message chosen by the sender was M, and "$\mathsf{C} = C$" is the event that the ciphertext computed by the sender and received by the adversary was C. The definition considers the conditional probability that the message was M given that the ciphertext was C. It says that this probability is exactly the a priori probability of the message M, namely $D(M)$.

In considering the one-time pad encryption scheme (cf. Scheme 5.2.1) we omit the counter as part of the ciphertext since only a single message is being encrypted. Thus, the ciphertext is a

	C	00	01	10	11
$D(M)$	M				
1/6	00	0.25	0.25	0.25	0.25
1/3	01	0.25	0.25	0.25	0.25
1/4	10	0.25	0.25	0.25	0.25
1/4	01	0.25	0.25	0.25	0.25

	C	00	01	10	11
$D(M)$	M				
1/6	00	1/6	1/6	1/6	1/6
1/3	01	1/3	1/3	1/3	1/3
1/4	10	1/4	1/4	1/4	1/4
1/4	01	1/4	1/4	1/4	1/4

Figure B.1: In the first table, the entry corresponding to row M and column C shows the value of $\Pr_{D,\mathcal{SE}}[\mathsf{C} = C \mid \mathsf{M} = M]$, for the one-time-pad scheme of Example B.0.2. Here the key and message length are both $k = 2$. In the second table, the entry corresponding to row M and column C shows the value of $\Pr_{D,\mathcal{SE}}[\mathsf{M} = M \mid \mathsf{C} = C]$, for the same scheme.

---

k-bit string where k is the length of the key and also of the message. Also note that in this scheme $r = 0$ since the encryption algorithm is not randomized.

Example B.0.2 Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the one-time-pad encryption scheme with the key length (and thus also message length and ciphertext length) set to $k = 2$ bits and the message space set to $\mathsf{Plaintexts} = \{0,1\}^k$. Let D be the message distribution on $\mathsf{Plaintexts}$ defined by $D(00) = 1/6$, $D(01) = 1/3$, $D(10) = 1/4$ and $D(11) = 1/4$. For each possible ciphertext $C \in \{0,1\}^k$, the first table of Fig. B.1 shows the value of $\Pr_{D,\mathcal{SE}}[\mathsf{C} = C \mid \mathsf{M} = M]$, the probability of obtaining this particular ciphertext if you encrypt M with the one-time pad scheme. As the table indicates, this probability is always 0.25. Why? Having fixed M, the possible ciphertexts are $M \oplus K$ as K ranges over $\{0,1\}^k$. So, regardless of the value of M, all different k bit strings are equally likely as ciphertexts. The corresponding general statement is stated and proved in Lemma B.0.3 below. The second table shows the value of $\Pr_{D,\mathcal{SE}}[\mathsf{M} = M \mid \mathsf{C} = C]$, the probability that the message was M given that an adversary sees ciphertext C. Notice that this always equals the a priori probability $D(M)$.

The following lemma captures the basic security property of the one-time-pad scheme: no matter what is the message, each possible k-bit ciphertext is produced with probability 2^{-k}, due to the random choice of the key. .

Lemma B.0.3 Let $k \geq 1$ be an integer and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the one-time-pad encryption scheme of Scheme 5.2.1 with the key length set to k bits and the message space set to $\mathsf{Plaintexts} = \{0,1\}^k$. Let D be a message distribution on $\mathsf{Plaintexts}$. Then

$$\Pr_{D,\mathcal{SE}}[\mathsf{C} = Y \mid \mathsf{M} = X] \;=\; 2^{-k} .$$

for any $X \in \mathsf{Plaintexts}$ and any $Y \in \{0,1\}^k$.

Proof of Lemma B.0.3: If X is fixed and known, what's the probability that we see Y? Since $Y = K \oplus X$ for the one-time-pad scheme, it only happens if $K = Y \oplus X$. The probability that K is this particular string is exactly 2^{-k} since K is a randomly chosen k-bit string. ▮

This enables us to show that the one-time-pad scheme meets the notion of perfect security we considered above.

Theorem B.0.4 *Let $k \geq 1$ be an integer and let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be the one-time-pad encryption scheme of Scheme 5.2.1 with the key length set to k bits and the message space set to* $\mathsf{Plaintexts} = \{0,1\}^k$*. Let D be a message distribution on* $\mathsf{Plaintexts}$*. Then $\mathcal{SE}$ is perfectly secure with respect to D.*

Proof of Theorem B.0.4: Let $M \in \mathsf{Plaintexts}$ be a message and let $C \in \{0,1\}^k$ be a possible ciphertext. We need to show that Equation (B.1) is true. We have

$$\begin{aligned}\Pr_{D,\mathcal{SE}}[\mathsf{M} = M \mid \mathsf{C} = C] &= \Pr_{D,\mathcal{SE}}[\mathsf{C} = C \mid \mathsf{M} = M] \cdot \frac{\Pr_{D,\mathcal{SE}}[\mathsf{M} = M]}{\Pr_{D,\mathcal{SE}}[\mathsf{C} = C]} \\ &= 2^{-k} \cdot \frac{\Pr_{D,\mathcal{SE}}[\mathsf{M} = M]}{\Pr_{D,\mathcal{SE}}[\mathsf{C} = C]} .\end{aligned}$$

The first equality was by Bayes' rule. The second equality was obtained by applying Lemma B.0.3 with $X = M$ and $Y = C$. By definition

$$\Pr_{D,\mathcal{SE}}[\mathsf{M} = M] \;=\; D(M)$$

is the a priori probability of M. Now for the last term:

$$\begin{aligned}\Pr_{D,\mathcal{SE}}[\mathsf{C} = C] &= \sum_X \Pr_{D,\mathcal{SE}}[\mathsf{M} = X] \cdot \Pr_{D,\mathcal{SE}}[\mathsf{C} = C \mid \mathsf{M} = X] \\ &= \sum_X D(X) \cdot 2^{-k} \\ &= 2^{-k} \cdot \sum_X D(X) \\ &= 2^{-k} \cdot 1 .\end{aligned}$$

The sum here was over all possible messages $X \in \mathsf{Plaintexts}$, and we used Lemma B.0.3. Plugging all this into the above we get

$$\Pr_{D,\mathcal{SE}}[\mathsf{M} = M \mid \mathsf{C} = C] \;=\; 2^{-k} \cdot \frac{D(M)}{2^{-k}} \;=\; D(M)$$

as desired. ▮

The one-time-pad scheme is not the only scheme possessing perfect security, but it seems to be the simplest and most natural one.

Bibliography

[1] Mihir Bellare. Practice-oriented provable security. Available via `http://www-cse.ucsd.edu/users/mihir/crypto-papers.html`.

[2] M. Bellare, J. Kilian and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences* , Vol. 61, No. 3, Dec 2000, pp. 362–399.

[3] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. *Proceedings of the 38th Symposium on Foundations of Computer Science*, IEEE, 1997.

[4] M. Bellare and O. Goldreich. On defining proofs of knowledge. *Advances in Cryptology – CRYPTO '92*, Lecture Notes in Computer Science Vol. 740, E. Brickell ed., Springer-Verlag, 1992.

[5] M. Bellare, R. Impagliazzo and M. Naor. Does parallel repetition lower the error in computationally sound protocols? *Proceedings of the 38th Symposium on Foundations of Computer Science*, IEEE, 1997.

[6] G. Brassard, D. Chaum, and C. Crépea. Minimum Disclosure Proofs of knowledge. *Journal of Computer and System Sciences*, Vol. 37, No. 2, 1988, pp. 156–189.

[7] Data Encryption Standard. FIPS PUB 46, Appendix A, Federal Information Processing Standards Publication, January 15, 1977, US Dept. of Commerce, National Bureau of Standards.

[8] J. Daemen and V. Rijmen. AES proposal: Rijndael. `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf`.

[9] J. Daemen and V. Rijmen. The design of Rijndael. Springer, 2002.

[10] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Info. Theory*, Vol. IT-22, No. 6, November 1976, pp. 644–654.

[11] U. Feige, A. Fiat, and A. Shamir. Zero-Knowledge Proofs of Identity. *Journal of Cryptology*, Vol. 1, 1988, pp. 77–94.

[12] U. Feige, and A. Shamir. Witness Indistinguishability and Witness Hiding Protocols. *Proceedings of the 22nd Annual Symposium on the Theory of Computing*, ACM, 1990.

[13] O. Goldreich. A uniform complexity treatment of encryption and zero-knowledge. *Journal of Cryptology,* Vol. 6, 1993, pp. 21-53.

[14] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, Vol. 25, No. 1, 1996, pp. 169–192.

[15] O. Goldreich, S. Micali, and A. Wigderson. Proofs that Yields Nothing but Their Validity, or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM*, Vol. 38, No. 1, July 1991, pp. 691–729.

[16] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology*, Vol. 7, No. 1, 1994, pp. 1–32.

[17] O. Goldreich, S. Goldwasser and S. Micali. How to construct random functions. *Journal of the ACM,* Vol. 33, No. 4, 1986, pp. 210–217.

[18] S. Goldwasser and S. Micali. Probabilistic encryption. *J. of Computer and System Sciences*, Vol. 28, April 1984, pp. 270–299.

[19] S. Landau. Standing the test of time: The Data Encryption Standard. *Notices of the AMS*, March 2000.

[20] S. Goldwasser, S. Micali and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. of Comp.*, Vol. 18, No. 1, pp. 186–208, February 1989.

[21] S. Goldwasser, S. Micali and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing,* Vol. 17, No. 2, pp. 281–308, April 1988.

[22] A. Joux and R. Lercier. Computing a discrete logarithm in GF(p), p a 120 digits prime, `http://www.medicis.polytechnique.fr/~lercier/english/dlog.html`.

[23] D. Kahn. The Codebreakers; The Comprehensive History of Secret Communication from Ancient Times to the Internet. Scribner, Revised edition, December 1996.

[24] L. Knudsen and J. E. Mathiassen. A Chosen-Plaintext Linear Attack on DES. *Fast Software Encryption '2000*, Lecture Notes in Computer Science Vol. , ed., Springer-Verlag, 192000.

[25] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput,* Vol. 17, No. 2, April 1988.

[26] M. Luby and C. Rackoff. A study of password security. *Advances in Cryptology – CRYPTO '87*, Lecture Notes in Computer Science Vol. 293, C. Pomerance ed., Springer-Verlag, 1987.

[27] C. Lund, L. Fortnow, H. Karloff and N. Nisan. Algebraic Methods for Interactive Proof Systems. *Journal of the ACM*, Vol. 39, No. 4, 1992, pp. 859–868.

[28] S. Micali, C. Rackoff and R. Sloan. The notion of security for probabilistic cryptosystems. *SIAM J. of Computing*, April 1988.

[29] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. *Proceedings of the 22nd Annual Symposium on the Theory of Computing*, ACM, 1990.

[30] A. ODLYZKO. The rise and fall of knapsack cryptosystems. Available via `http://www.research.att.com/~amo/doc/cnt.html`.

[31] C. RACKOFF AND D. SIMON. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. *Advances in Cryptology – CRYPTO '91*, Lecture Notes in Computer Science Vol. 576, J. Feigenbaum ed., Springer-Verlag, 1991.

[32] RONALD RIVEST, MATT ROBSHAW, RAY SIDNEY, AND YIQUIN YIN. The RC6 Block Cipher. Available via `http://theory.lcs.mit.edu/~rivest/publications.html`.

[33] R. RIVEST, A. SHAMIR, AND L. ADLEMAN. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM,* Vol. 21, No. 2, February 1978, pp. 120–126.

[34] A. SHAMIR. IP = PSPACE. *Journal of the ACM*, Vol. 39, No. 4, 1992, pp. 869–877.

[35] D. WEBER AND T. DENNY. The solution of Mccurley's discrete log challenge. *Advances in Cryptology – CRYPTO '98*, Lecture Notes in Computer Science Vol. 1462, H. Krawczyk ed., Springer-Verlag, 1998.