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

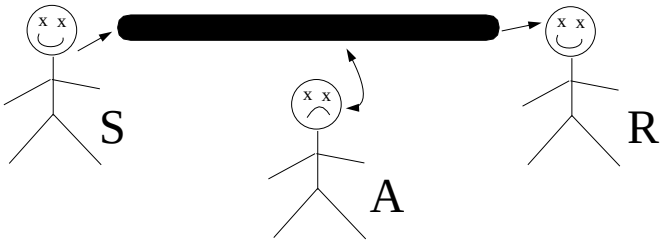


Figure 1.1: Several cryptographic goals aim to imitate some aspect of an ideal channel connecting a sender S to a receiver R.

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 $Y =$ `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 $\langle something \rangle$ 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

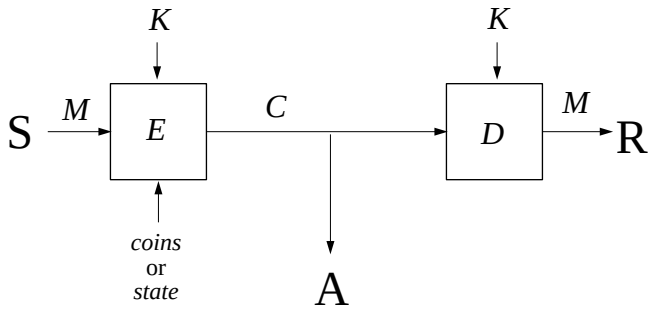


Figure 1.3: Symmetric encryption. The sender and the receiver share a secret key, K. The adversary lacks this key. The message M is the plaintext; the message C is the ciphertext.

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

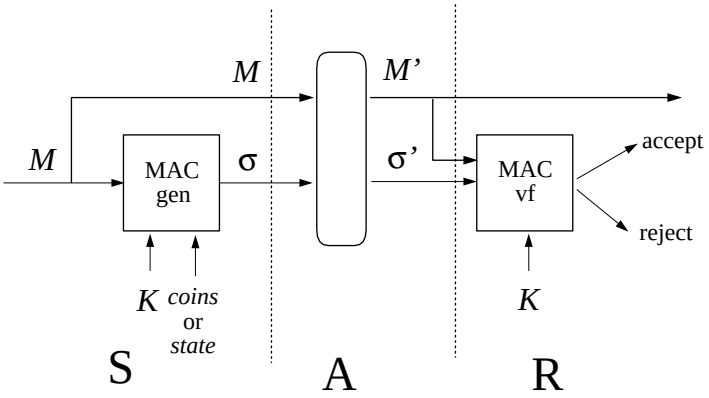


Figure 1.4: A message authentication code. The tag σ accompanies the message M. The receiver R uses it to decide if the message really did originate with the sender S with whom he shares the key K.

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

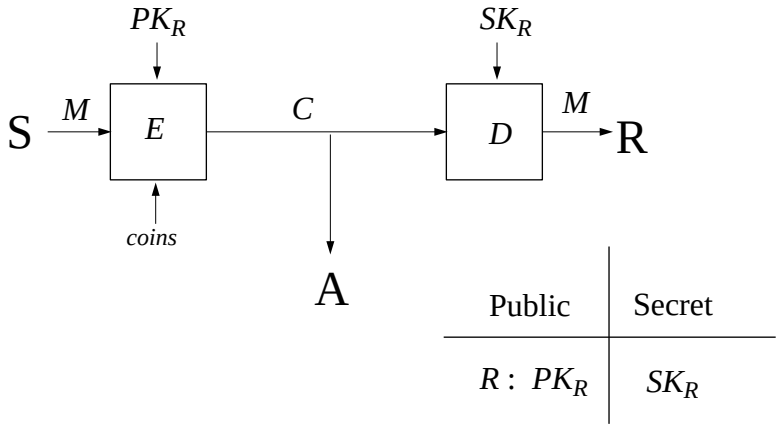


Figure 1.5: Asymmetric encryption. The receiver R has a public key, pk_R, which the sender knows belongs to R. The receiver also has a corresponding secret key, sk_R.

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 [DH].

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 Sign. 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 $\Pi = (\mathcal{K}, \mathrm{Sign}, \mathcal{V})$ 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.

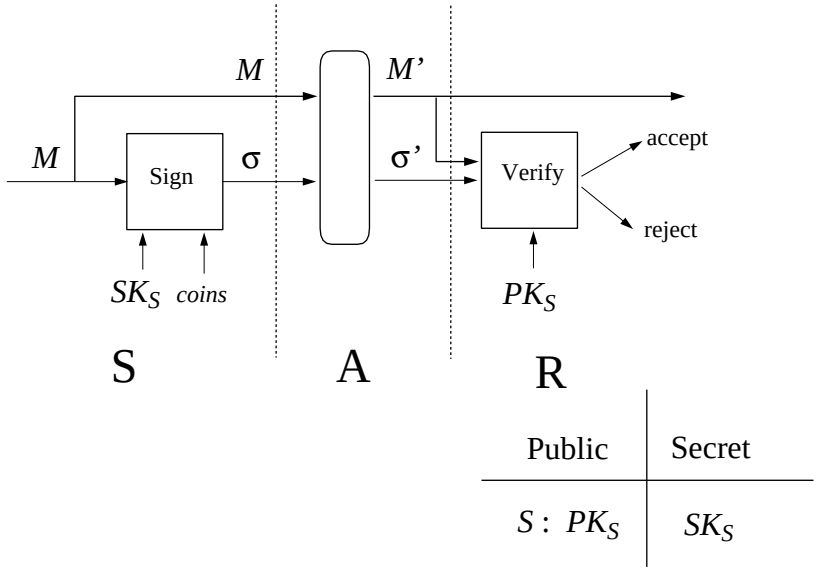


Figure 1.6: A digital signature scheme. The signature σ accompanies the message M. The receiver R uses it to decide if the message really did originate with the sender S with has public key pk_S.

	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 "pseudoran-

dom."

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."

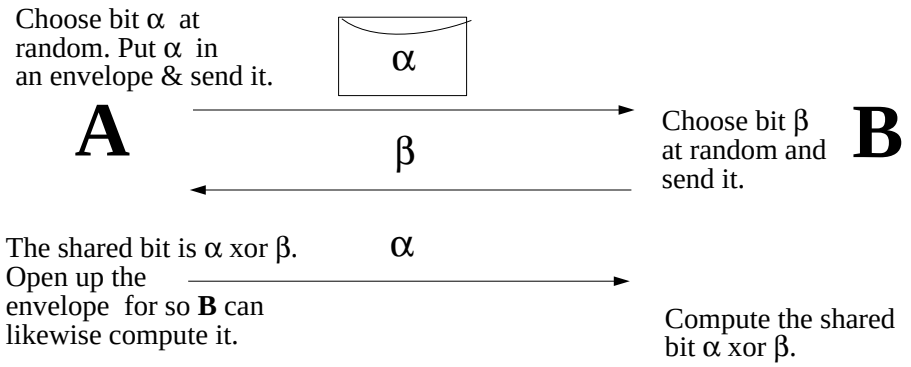


Figure 1.8: Envelope solution to the telephone-coin-flipping 5problem.

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 $i \xleftarrow{\$} \{0, 1, \ldots, n-1\}$. 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 $\pi\colon\ \Sigma \to \Sigma$ (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.

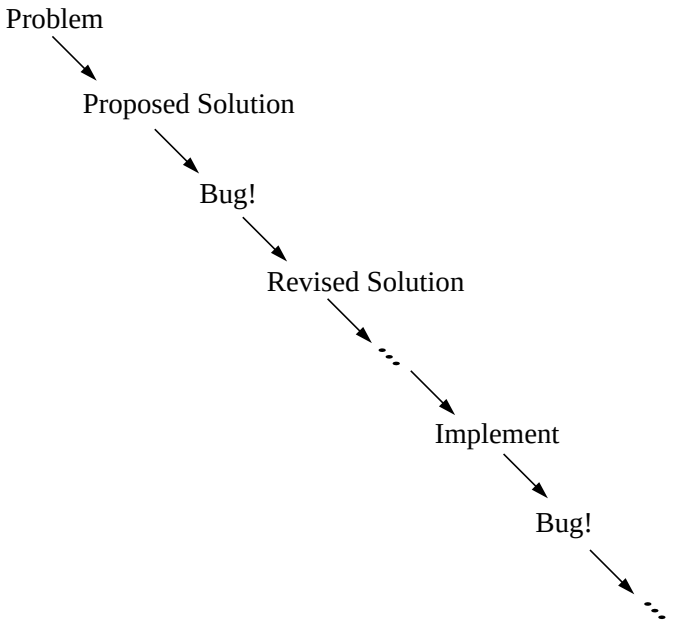


Figure 1.9: The classical-cryptography approach.

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.

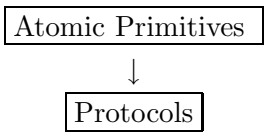
Notice another important thing. Nobody said anything about *how* the adversary operates. What algorithm, or technique, does she use? We do not know anything about that. The statement holds nonetheless. So it is a very strong statement.

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.



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

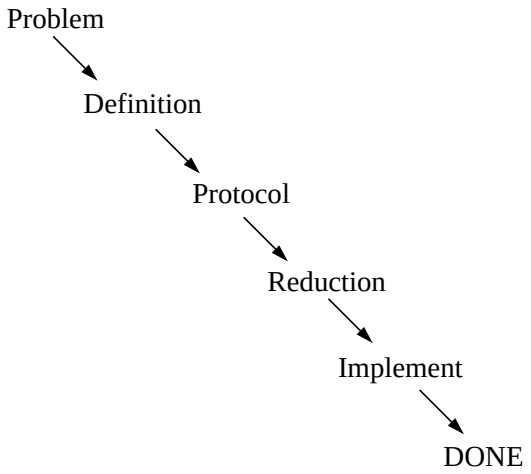


Figure 1.10: The provable-security paradigm.

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 $Q_{\rm H}$ with probability 1/2, and the next state will be $Q_{\rm T}$ 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: $T(n) = \max_{x,\,|x|=n}[\#\text{Steps}_M(x)]$. 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, $T(n) = \max_{x,\,|x|=n} \mathbf{E}[\#\text{Steps}_M(x)]$, 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.