Last Revised: January 13, 2000

Chapter 1

Introduction

Modern cryptography is a remarkable field. It manages to solve problems that one might guess to be unsolvable, to define goals that one might assume to be undefinable, and to prove theorems that one might think to be unprovable. Cryptography deals with very human concerns—issues of privacy, authenticity, and trust—but it does so in a way that is concrete and mathematical. This course is your invitation to this fascinating subject.

The word "cryptography" comes from the Latin *crypt*, meaning secret, and *graphia*, meaning writing. So "cryptography" is literally "secret writing": the study of how to obscure what you write so as to render it unintelligible to those who should not read it. Nowadays cryptography entails a lot more than finding good ways for keeping your communications secret. That problem remains one of cryptography's central problems, but many more problems have been added to the mix.

Despite the scope of cryptography having broadened, much of the flavor of this field is unchanged since the very early days: this is still a game of clever designs, sneaky attacks, and mathematical slight of hand. But one thing that has changed is that the "art" of cryptography has now been supplemented with a legitimate "science." In this class we shall focus on that science.

Be forewarned: cryptography is a slippery subject. Surprisingly often that which seems meaningful turns out to be meaningless, that which seems true turns out to false, and that what seems impossible turns out to be doable. So have fun—but retain a healthy skepticism, and always watch your step.

1.1 Some Sample Cryptographic Problems

Let us begin by looking at a few of the problems that cryptographers have considered. 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.

1.1.1 Message Privacy

Imitating the ideal channel Lets introduce the first two members of our cast of characters: our sender, S, and our receiver, R. The sender and receiver want to communicate with each other, say over a network.

(Sometimes people call these characters Alice, A, and Bob, B. Alice and Bob figure in many works on cryptography. But the authors can never remember what is the role for Alice and what is the role for Bob, and we're going to want the letter A for someone else, anyway.)

What is the *ideal* channel over which the sender and receiver could conceivably communicate? Imagine they are provided with a dedicated, untappable, impenetrable lead pipe into which the sender can whisper a message and the receiver will hear it. Nobody else can look inside the pipe or change what's there. This lead pipe provides the perfect medium, available only to the sender and receiver, as though they were alone in the world. See Figure 1.1.

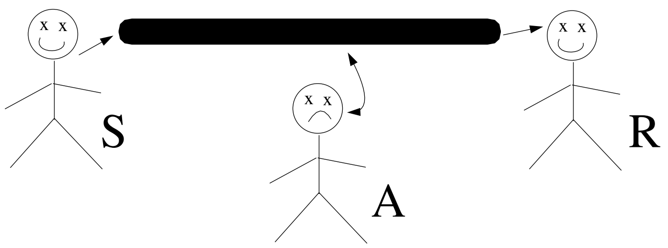


Figure 1.1: Several cryptographic goals aim to imitate some aspect of an ideal channel connecting a sender S to a receiver R.

Exercise 1.1.1 *Why is the adversary frowning in Figure 1.1?*

Unfortunately, in real life, ideal channels don't connect the pairs of parties that might like to communicate. Usually all we have is a public network like the Internet.

Several cryptographic goals concern themselves with imitating, in some respect, an ideal channel between the sender and receiver. In these problems the parities are communicating over is an insecure channel but they want to imagine that they have a perfect lead pipe between them. Cryptography is used to create the *illusion* that their channel is secure. The parties should be assured of the kinds of properties that they would expect of a secure channel.

Protocols and adversaries What mechanisms are available to help us to imitate the lead-pipe world? All we are allowed to do is to supply the sender and receiver with a ***protocol***. A protocol is just like a program, except that it is a distributed program. It tells the sender and receiver what to do.

How do we start building protocols? We first try to isolate the the *threats* and the *goals*. Once we have a good idea about these, we can try to find protocol solutions.

At this point we should introduce the third member of our cast. This is our ***adversary***, denoted A. An adversary is the source of the possible threats. Cryptographic protocols attempt to surmount the influence of the adversary.

In cryptography we must focus on the adversary. What can she do, and what can't she do? It is important straight away to give the adversary full membership and respect in the cast of characters. She is in many ways the central character.

If you think about it, there are actually several different things that the adversary might be trying to do in attacking our protocol-approximation of a lead pipe. The first thing we are concerned with is that the adversary might want to understand the content of the messages sent from the sender to the receiver. This is an attack on the parties' privacy.

Encryption In order to protect the privacy of transmissions we use a tool called ***encryption***. The sender *encrypts* his message M and sends it. The message M is called ***plaintext***. What the sender creates by encrypting M is called a ***ciphertext***, C. The receiver, on receipt of ciphertext C, *decrypts* it. If all goes well, the receiver should now have recovered the same plaintext, M, that the sender sent out.

You might hope for encryption to emulate all the properties of a lead pipe that have to do with privacy, but this is actually too much to hope for. First, we don't normally expect for encryption to hide the *existence* of a message. This is a potentially important piece of information to the adversary, but it is too often infeasible to try to hide this. Nor is encryption normally intended to hide the *length* of a message. The length of the plaintext is another potentially interesting piece of information to the adversary, but we won't usually try to hide this, either. If we know a maximal length for message we could use padding to hide the length of messages, but this would typically entail a large loss of efficiency.

Keys It is not hard to convince yourself that in order to communicate securely, there must be something that the receiver knows, or can do, that the adversary can not do. There has to be some "asymmetry" between the situation the the receiver finds himself in, and the adversary finds herself in. 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* trust model. Encrypting in the symmetric trust model is called ***symmetric encryption***, ***shared-key encryption***, or ***private-key encryption***. It's all the same thing.

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 Figure 1.2.) The sender and receiver must somehow use the key K to overcome the presence of the adversary.

Notice how randomness enters the picture. The key is random. Randomness is a central and unavoidable element of cryptography. Everything is about probabilities. Get used to it!

The key is kept securely on the parties' machines. It is an assumption that we must make that the adversary cannot penetrate these machines and recover the key.

A picture for symmetric encryption can be found in Figure 1.3. The sender sends the receiver a plaintext M by first computing $C \leftarrow \mathcal{E}_K(M)$. The *encryption function* $\mathcal{E}$ may be randomized, or it might keep some state around. The receiver recovers the plaintext M from the ciphertext C by *decrypting* the ciphertext, setting $M \leftarrow \mathcal{D}_K(C)$.

We warn you that a picture like Figure 1.3 is actually a little misleading about what the adversary can and can not do. In particular, it might seem to suggest that the adversary is just a passive eavesdropper, quietly listening to the communications between the sender and receiver. In fact, this might not be the case at all. We will consider adversaries that are much more powerful than that.

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. We'll assume the binary alphabet. 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, $Y =$ `cryptography` is a string of length 12. The string of length zero is called the ***emptystring*** 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, $1011 \,\|\, 0 = 10110$. The i-th character of a string X, where $1 \le i \le |X|$, is denoted $X[i]$, so that $X = X[1] \,\|\, X[2] \,\|\, \cdots \,\|\, X[|X|]$. If a is a character and $i \ge 0$ is a number then a^i is the string consisting of the character a repeated i times. It is understood that $a^0 = \varepsilon$ for any character a. So, for example, $0^3 = 000$ and 1^n is how you'd write the number n in unary notation. We can encode almost anything into a string. 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.

Encryption with a one-time-pad Now let's give an example of a protocol that encrypts. Here is how the sender and receiver encrypt in any spy novel. Let $K = K[1]\cdots K[k]$ denote the shared key, which is a random sequence of k bits. Think of k as some big number, like a million. Let $M = M[1]\cdots M[n]$ denote the plaintext message that the sender wants to send, also divided up into bits. Assume that $n \le k$ (that is, the key is at least as long as the plaintext).

What the sender does is to compute $C'[i] = K[i] \oplus M[i]$ for each $i = 1, \ldots, n$. The symbol $\oplus$ denotes the exclusive-or (XOR) operation: $0 \oplus 0 = 1 \oplus 1 = 0$, while $0 \oplus 1 = 1 \oplus 0 = 1$. The string C' is the main part of the ciphertext which the sender sends out. The receiver receives $C' = C'[1]\cdots C'[n]$ and can recover M via $M[i] = C'[i] \oplus K[i]$ for all $1 \le i \le n$.

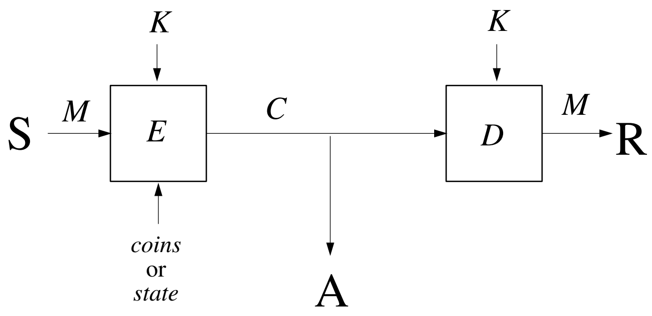


Figure 1.3: Symmetric encryption. The sender and the receiver share a secret key, K. The adversary lacks this key. The message M is the plaintext; the message C is the ciphertext.

This is possible for the receiver because he too knows the key K.

When the sender wants to encrypt another message she has to use new key bits. That is, she keeps track of where she is in the key, via a counter, and goes on from there. Key bits are never re-used. That's why this is called a *one-time pad*: each key bit is used at most once. You cannot encrypt more data than you have key bits. An indication of where the sender is in the key should be included in the ciphertext. See Figure 1.4.

Exercise 1.1.2 *Verify that the receiver recovers the plaintext that was sent.*

Exercise 1.1.3 *Figure out the damage that would be done if a key bit was used twice.*

Exercise 1.1.4 *Suppose that the key, instead of being a random bit, is a random number between 0 and 99. Describe how you would encrypt a number that itself was between 0 and 99. (Hint: don't use XOR, use something else.)*

Exercise 1.1.5 *Suppose that you want to encrypt a 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 secure protocol to you? Explain.*

What we have just done is specify a protocol. This is the sequence of instructions for the parties to execute. In the case of symmetric encryption, a protocol needs to specify three things: how to encrypt, how to decrypt, and how to generate the shared key. Formally, an encryption protocol Π is a three-tuple of algorithms, $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. Notice that the adversary is

Algorithm $\mathcal{K}$	Algorithm $\mathcal{E}_K(M)$	Algorithm $\mathcal{D}_K(C)$

for $i \leftarrow 1$ to k do
    $K[i] \stackrel{R}{\leftarrow} \{0,1\}$
return K
	static $ctr \leftarrow 0$
	$n \leftarrow \|M\|$
	if $ctr + n > k$
	    then return error
	for $i \leftarrow 1$ to n do
	    $C'[i] \leftarrow K[ctr + i] \oplus M[i]$
	$ctr \leftarrow ctr + n$
	return $\langle ctr, C' \rangle$
		$\langle ctr, C' \rangle \leftarrow C$
		$n \leftarrow \|C'\|$
		if $ctr + n > \|K\|$
		    then return error
		for $i = 1$ to n do
		    $M[i] = K[ctr + i] \oplus C[i]$
		return M

Figure 1.4: Encryption with a one-time pad. The first algorithm generates the key K, the second encrypts plaintext M, and the last decrypts ciphertext C.

not involved in the protocol. We never try to tell the adversary what to do. That is up to her.

We'll be discussing the security of one-time-pad encryption later in this chapter.

Public-key encryption The shared key K between the sender and the receiver is not the only way to create the information asymmetry that we need between the receiver and the adversary. In ***asymmetric encryption***, also called ***public-key encryption***, the receiver R possesses a *pair* of keys—a *public key*, pk_R, and a *secret key*, sk_R. The receiver's public key is made publicly known and bound to his identity. For example, the receiver's public key might be published in a phone book. When the sender wants to send a secret message M to the receiver, she looks up the receiver's public key in the phone book and computes $y \leftarrow \mathcal{E}_{pk_R}(M)$. When the receiver receives a ciphertext C he computes $M \leftarrow \mathcal{D}_{sk_R}(C)$.

The ***trust model*** specifies who, initially, has what keys. We have just described two different trust models for achieving the same basic aim: the symmetric (or shared-key) trust model and the asymmetric (or public-key) trust model.

The idea of public-key cryptography, and the fact that we can actually realize this goal, is a remarkable idea. Think about it! 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. Diffie was Hellman's graduate student at Stanford. The idea was published in 1976.

For a picture of encryption in the public-key setting, see Figure 1.5.

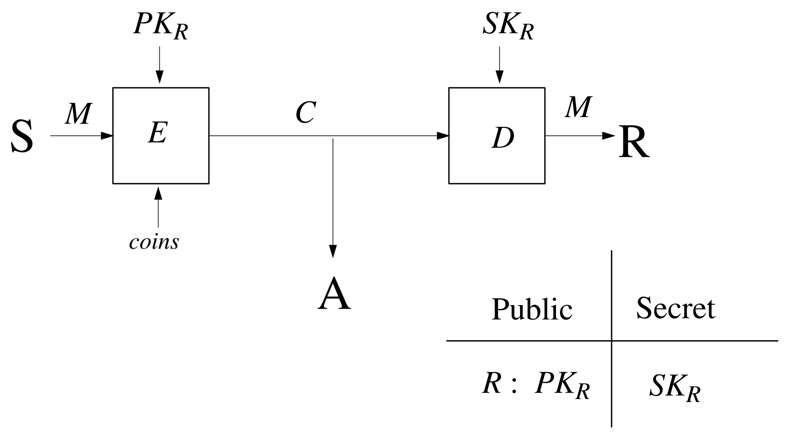


Figure 1.5: Asymmetric encryption. The receiver R has a public key, pk_R, which the sender knows belongs to R. The receiver also has a corresponding secret key, sk_R.

Exercise 1.1.6 *Think of two or three more ways to "create asymmetry" between the receiver and the adversary. Show how you would encrypt in each setting.*

1.1.2 Message Authenticity

Now we're going to try to imitate another aspect of the lead pipe between the sender and the receiver: it is the fact that only the sender can speak into her end of the pipe, and what will emerge at the receiver's end is exactly what the sender said. That is, 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 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.

Once again, information asymmetry is needed. The symmetric and asymmetric trust models described above are equally applicable here.

The usual tool for solving the message-authentication problem in the symmetric setting is a *message authentication code* (MAC). Here the sender and receiver share a secret key, K, and when the sender wants to send a message M to the receiver she attaches to it a few bits, σ, which is called the *tag* for the message. The tag is computed using M and K. The MAC computation might be probabilistic or use state, just as with encryption. Or it may well be deterministic. Bob, on receipt of M and σ, uses the key K to check if the tag is OK. If so, 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 1.6. The picture is again slightly misleading in terms of what the adversary A might or might not be able to do, but it gets the point across.

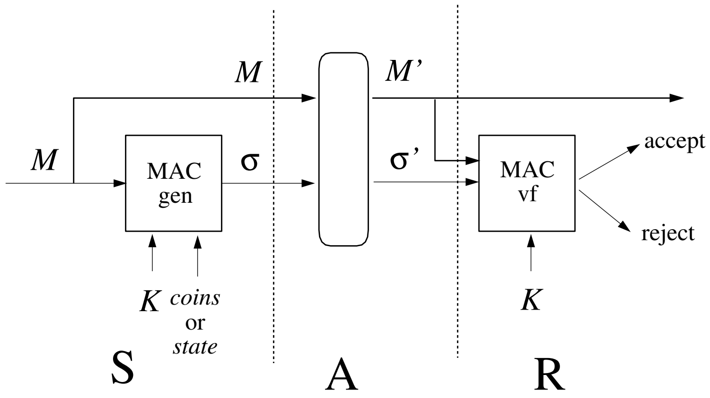
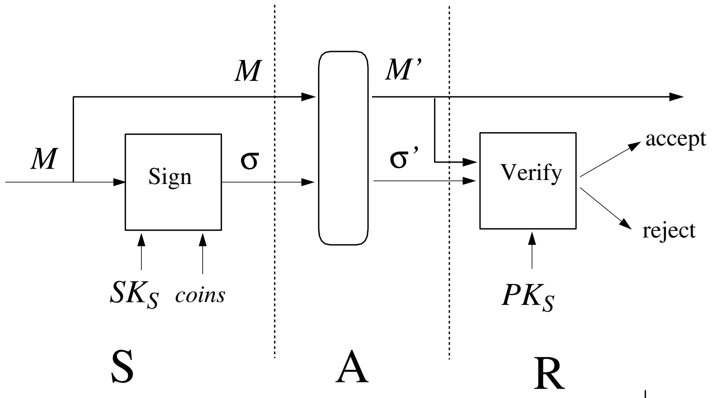


Figure 1.6: A message authentication code. The tag σ accompanies the message M. The receiver R uses it to decide if the message is really did originate with the sender S with whom he shares the key K.

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. Everyone (even the adversary) knows the key pk_S and that it belongs to party S. 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. The receiver, on receipt of M and σ, checks if it is OK using the public key of the sender, pk_S. If it is fine, the receiver regards M as authentic; otherwise, he regards M as an attempted forgery. A picture is given in Figure 1.7.



Public	Secret
$S:\ PK_S$	SK_S

Figure 1.7: A digital signature scheme. The signature σ accompanies the message M. The receiver R uses it to decide if the message is really did originate with the sender S with has public key pk_S.

One difference between a MAC and a digital signature concerns what is called **non-refutability**. 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 a court of law. But with a digitally-signed message the *only* party who should be able to produce a a message that verifies under public key pk_S is the party S herself. Thus if the signature scheme is good party S can not 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 can not refute that. Probably the sender S will have to claim that the key sk_S was stolen from her. Perhaps this, if true, might still be construed as the sender's fault.

To summarize, there are two common aims concerned with sending a message so as to 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 follows:

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

1.1.3 One-Way Functions

Describe this problem, in the context of storing passwords on a server.

1.1.4 Pseudorandom Number Generation

Computers are quite deterministic. But for lots of applications, "random numbers" are useful. These applications involve simulation, efficient algorithms, and cryptography itself. We've already seen an example protocol that required random bits.

How can a completely deterministic machine generate "random" numbers? Well, it can't. But a machine can do the next best thing: it can *stretch* a little bit of randomness into a lot of "pseudorandomness."

Suppose we wire to our computer a Geiger counter that generates a "random" bit every second. We run our computer for a little while and now it has 200 "random" bits. We won't worry about the "philosophical" question as to whether these bits 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.

A *pseudorandom generator* (PRG) stretches a short (eg., 200-bit) "truly random" string into a much longer (a million bits, say) string which "looks" random. This is another core problem of cryptography.

1.1.5 Authenticated Key Exchange

Describe this problem, in the context of telnet. Point out that there is no reasonable solution known to this problem when the server has crypt(pw).

1.1.6 Telephone Coin Flipping

Alice and Bob want to decide which of them has to show up to AI class tomorrow. They take turns going to class but, you see, there are an odd number of lectures. Neither wants to go, but one of them better do it.

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, I'll go to AI class and take notes for us. If you get it wrong, you'll have to go."

Bob's 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

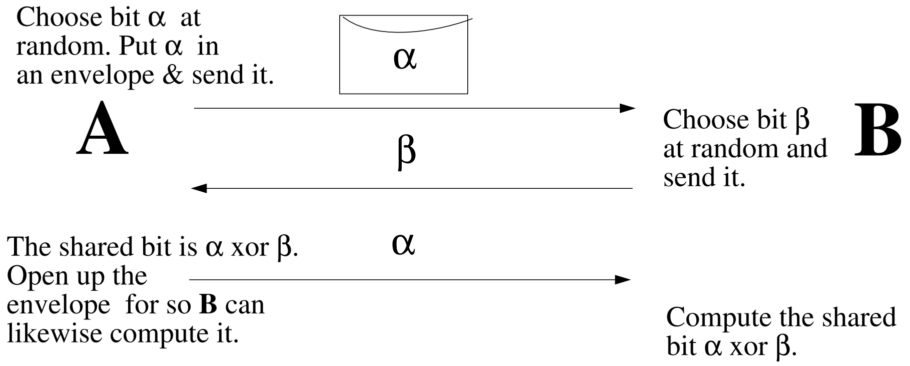


Figure 1.8: Envelope solution to the telephone-coin-flipping 5problem.

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 envelop) 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.

Exercise 1.1.7 *What should happen if Alice refuses to send her second message? Is this potentially damaging?*

1.1.7 Secure Function Evaluation

The dating problem Alice might—or might not—want to go out on a date with Bob. Bob might—or might not—what to go out on a date with Alice. The age-old problem (which keeps many of us apart) is a desire not to have A's interest in B known to B unless B is likewise interested; and, similarly, a desire not to have B's interest in A known to A unless A is likewise interested. That is, Bob would rather not even ask Alice if she's going to say no—and Alice would rather not even ask Bob if he's going to say no. But if no one ever asks, how is anyone to ever know?! Life is full of predicaments.

Solving the dating problem amounts to finding a way to compute the AND of privately held bits in such a manner that *only* the correctly computed conjunct is known to the two parties.

We won't give a solution to the dating problem—we don't know a simple solution in the communications model. See if you can find one!

Exercise 1.1.8 *Find a solution to the dating problem in the physical world. State (informally) the assumptions you are making.*

The millionaire's problem *Describe the problem.*

Exercise 1.1.9 *Find a solution to the millionaire's problem in the physical world. State (informally) the assumptions you are making.*

The general problem *Describe this problem.*

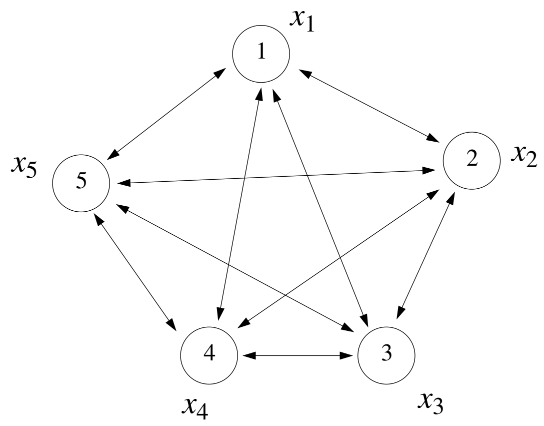


Figure 1.9: Secure function evaluation. In this picture there are five players, numbered 1, 2, 3, 4, and 5, and each player i has some private input x_i. The players want to compute $f(x_1, x_2, x_3, x_4, x_5)$ in a way that reveals only the correctly-computed output value.

More protocol problems Cryptographic protocols go beyond the samples we have given. They are used for electronic payments, signing contracts, holding elections, giving zero-knowledge proofs, and playing games like poker over a network. But to get there, we must first do the basics. So we will spend a lot of time on encryption, authentication, pseudorandom bit generators, and so on, before we get to sophisticated protocols.

1.2 What Cryptography is About

Protocols and adversaries Let us now move away from the particular examples we have given and ask what, in general, is cryptography about? Briefly, cryptography is about constructing and analyzing ***protocols*** which overcome the influence of ***adversaries***. One way to know that you've left the world of cryptography is that there is no protocol or no adversary anywhere in sight. In the last section 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 $i \stackrel{R}{\leftarrow} \{0, 1, \ldots, n-1\}$. This notation means that i is a random value from the indicated set, all values being equally likely.

- A party might run a protocol for a while, and then pause, waiting to hear from another party. Then the party resumes, running the protocol for another phase. This can go on and on. Whether or not this happens depends on the protocol problem that we are trying to solve.

- 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.

- Sometimes we will allow for parties to access an *oracle*. An oracle is a "magic box" to which the parties can ask questions. The protocol will specify, in any given case, how this box is supposed to answer the party's questions. This is just an "intermediate step" in designing protocols. In a "finished" protocol there aren't any oracles.

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.

Exercise 1.2.1 *Where is the adversary hiding in the pseudorandom-number-generation problem?*

Exercise 1.2.2 *Where is the adversary hiding in the dating problem?*

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.

Specify a RAM model. (Should we use fixed-width registers or arbitrary precision?)

Figure 1.10: The RAM model, with an oracle. An adversaries is a program written in this model of computation. Details of the model are not important, but one has to fix *some* model of computation.

When we formalize adversaries they will be "random access machines (RAMs) with access to an oracle." See Figure 1.10 for a for a description of this model of computation.

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 not to have been so good. In industry, 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 or 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.

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're not cryptography.

(Actually, most cryptographers have quite friendly feelings towards our adversaries. We like them. Without an adversary, at least a hypothetical one, cryptography would have nothing left to do. We'd have to seek employment as mathematicians, where jobs are scarce and salaries are low. No, better to have plenty of adversaries, and to stay on good terms with them.)

Another rule that most cryptographers insist on is to make the protocols *public*. That which must be secret should be embodied in **keys**. The keys specify 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 can not 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 which 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 known.

Exercise 1.2.3 *Argue that what we have just said—that protocols should be* *public, but their keys should be secret—is actually meaningless.*

1.3 Approaches to the Study of Cryptography

Phases in cryptography's development The history of cryptography can be divided into three rough stages. In the first, early stage, it was viewed as an *art*. Julius Caesar used cryptograms. His and other early schemes often took the form of substitution ciphers. If $\mathcal{A} = \{A, B, \ldots, Z\}$ is the alphabet (Caesar of course used the Roman one!), the simplest substitution cipher is simply a permutation $f\colon\ \mathcal{A} \to \mathcal{A}$, associating with each "plaintext" letter x its "ciphertext" letter $f(x)$. (Permutation means it is one-to-one and onto, that is, bijective.) The mapping f is known to receiver and sender, but, at least a priori, not to an adversary. To send a word, view it as a sequence of letters, $w = w[1] \ldots w[n]$. The sender computes $c[i] = f(w_i)$ for $i = 1, \ldots, n$ and transmits $c = c[1] \ldots c[n]$. The receiver, knowing f, also knows f^{-1}, and can decode. The adversary, not knowing the association f, but seeing only c, may be baffled at first. But once enough words have been transmitted, the code is soon broken, because we can make guesses based on repetitions of letters and knowledge of frequencies of letters in words in the English language. 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. Mathematical tools are used to design protocols and computers are used implement them. It is during this most recent stage that cryptography becomes much more a science.

We can characterize much of the work that has been going on in cryptography in a couple of different dimensions. The first distinction is between *cryptanalysis-driven* design and *proof-driven* design. The second distinction is between *information-theoretic* cryptography and *complexity-theoretic* cryptography. We would like to take up these two dimensions.

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.

The third step is called *cryptanalysis*. In the classical approach to design, cryptanalysis was an essential component of constructing any new design.

Sometimes one finds protocol problem in the form of subtle mathematical relationships which 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 reserve 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.

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 an good attack, but it is not easy to keep secret such a thing. See Figure 1.11.

Doing cryptanalysis well takes great 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 great mathematical prowess seem to be the requisite skills, not any specific piece of knowledge. Maybe you have heard of Don Coppersmith or Adi Shamir. These are two of the masters of this field.

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

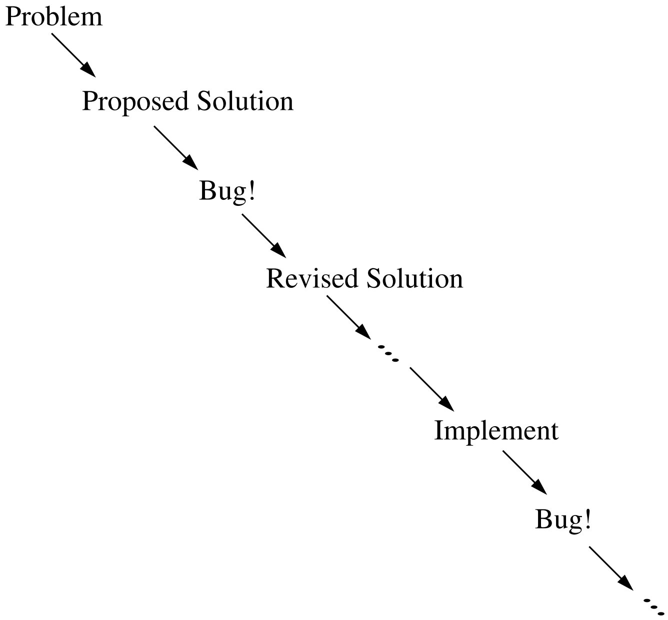


Figure 1.11: The classical-cryptography approach.

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 and our particular protocol for doing this, which was to use a one-time-pad. Security, we have said, means defeating an adversary, so we have to specify what is the adversary wants to do.

One might think that the adversary's goal is something like this: given a ciphertext C, and not knowing key K, try to figure out the plaintext M. Here is one attempt to make precise that the adversary can't do this if we encrypt using a one-time pad: "It is impossible for the adversary, given C, to write down M." Is this statement true? No. The adversary might well guess M, by outputting a random sequence of n bits, where $n = |C|$. She would be right with probability 2^{-n}. Not bad if, say $n = 1$. Does that make the scheme bad? Of course not. But it tells us that security is a probabilistic thing.

Another issue is a priori knowledge. Even before M is transmitted the adversary might know something about it. For example, the adversary might have reason to believe that M is either 0^n or 1^n. Why? Maybe because the adversary knows what the sender and receiver are are talking about. If we're trying to make a general definition, we can't assume that the adversary *doesn't* know what the parties are talking about. If the adversary knows that the message is either 0^n or 1^n then the adversary can get the message right with probability $1/2$. How is this factored in?

All this tells us that we need a proper definition of security, some formal way of saying what it means for the scheme to be secure. We present the idea of Shannon.

Let $\mathcal{M}$: $\{0,1\}^n \to [0,1]$ be a probability distribution on the set of n-bit messages. That is, assume Alice chooses M with probability $\mathcal{M}(M)$. This distribution is known to everyone, including the adversary. Thus, before C is transmitted, all the adversary knows is that any particular message M has probability $\mathcal{M}(M)$ of being transmitted.

We want to capture the constraint that the adversary's information about the message does not increase after seeing the ciphertext. We have fixed some encryption scheme $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ in mind. For any string C let $P_C(M)$ denote the a posteriori probability of M given ciphertext C, namely

$$P_{\mathcal{M}}(C, M) \;=\; \mathbf{P}\left[\, \text{Message was } M \mid \text{Ciphertext was } C \,\right] .$$

Here the probability is over the choice of key K and the choice of M from $\mathcal{M}$. Note it is a conditional probability, namely the probability that M was the message given that a particular ciphertext C has been seen.

Definition 1.3.1 Encryption scheme $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is ***Shannon secure*** if for every distribution $\mathcal{M}$ it is the case that for every ciphertext C which occurs with nonzero probability, and message M, we have $P_{\mathcal{M}}(C, M) = \mathcal{M}(M)$.

The way to interpret it is that after having seen C, let the adversary take her best guess as to what M was. The probability that she is right is not more than the probability that she would have been right had the sender simply chosen a message, transmitted nothing at all, and asked the adversary to guess this message.

As long as you don't end up with *more* information about the message after seeing C than you had before, then the encryption is secure.

We claim one-time-pad encryption has the above property, and propose to prove it. You might want to brush up on your probability before you tackle this: Bayes' rule, conditioning, and so on. We will use such tools many times again.

Proposition 1.3.2 One-time-pad encryption is Shannon secure.
Proof: We have to show that Definition 1.3.1 is met. Bayes' rule tells us that

$$\mathbf{P}[M \mid C] \;=\; \mathbf{P}[C \mid M] \cdot \frac{\mathbf{P}[M]}{\mathbf{P}[C]} \,. \tag{1.1}$$

Let's consider the terms on the right hand side one by one. If M is fixed and known, what's the probability that we see C? Since $C = K \oplus M$, it only happens if $K = C \oplus M$. The probability that K is this particular string is exactly 2^{-n}. Thus

$$\mathbf{P}[C \mid M] \;=\; 2^{-n} \,. \tag{1.2}$$

By definition $\mathbf{P}[M] = \mathcal{M}(M)$ is the a priori probability of M. Now for the last term:

$$\begin{aligned} \mathbf{P}[C] &= \sum_m \mathbf{P}[m] \cdot \mathbf{P}[C \mid m] \\ &= \sum_m \mathcal{M}(m) \cdot 2^{-n} \\ &= 2^{-n} \cdot \sum_m \mathcal{M}(m) \\ &= 2^{-n} \cdot 1 \,. \end{aligned}$$

The sum here was over all possible messages m. We used the fact that $\mathbf{P}[C \mid m] = 2^{-n}$ as in Equation (1.2), and that the sum over all m of the probability of m is of course 1 since $\mathcal{M}$ is a probability distribution. Finally, plugging all this into Equation (1.1) we get

$$\mathbf{P}[M \mid C] \;=\; 2^{-n} \cdot \frac{\mathcal{M}(M)}{2^{-n}} \;=\; \mathcal{M}(M)$$

as desired. ▌

A limitation on Shannon-secure encryption Recall that the key in the one-time-pad scheme had to be at least as long as the number of bits we want to encrypt. It turns out that this is necessary to achieve Shannon security. That is, if an encryption scheme is to meet Definition 1.3.1, the number of key bits must be at least a the total number of plaintext bits we're going to encrypt.

Exercise 1.3.3 *Prove that it is impossible to obtain Shannon-security if you want to encrypt a 10-bit message using a 9-bit key. (Hint: Use a counting argument.)*

This fact has some fundamental implications. 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 security. We must seek a different paradigm and a different notion of security.

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 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.

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.

$$\begin{array}{c} \boxed{\text{Atomic Primitives}} \\ \downarrow \\ \boxed{\text{Protocols}} \end{array}$$

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 *block ciphers* 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: $\{0, \ldots, p-2\} \to Z_p^*$ defined by $f(x) = g^x \bmod p$. This is called *discrete exponentiation*, and its inverse is called ***discrete logarithm***: $\log_g(y)$ 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.

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 mis-used.

Indeed, lots of protocols have been broken 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 loose 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, there 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 Figure 1.12.

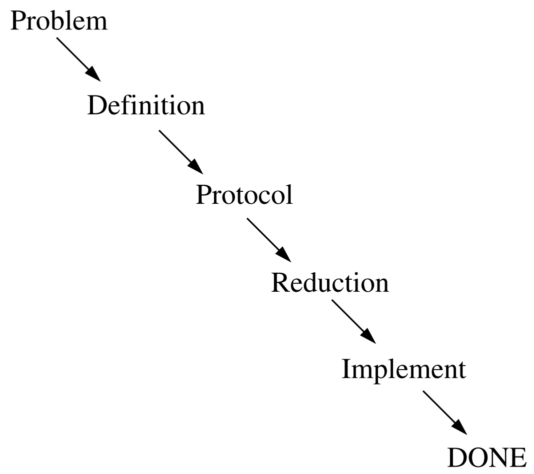


Figure 1.12: The provable-security paradigm.

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 one-wayness 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

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 1.13: The analogy between reductionist-cryptography and NP-Completeness.

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. The analogy is further spelled out in Figure 1.13, 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.

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.

I 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.

Problems

Problem 1.1 *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 fair 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 1.2 *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.

(What does it mean that Eve learns nothing about M? That for all strings κ, the probability that Alice says κ is independent of M: for all messages M_0, M_1 we have that

$$\Pr[\text{ Alice says } \kappa |\ M = M_0] = \Pr[\text{ Alice says } \kappa |\ M = M_1]\ .$$

The probability is over the the random shuffle of the cards.)

Problem 1.3 *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.)

History and references

To be written. Eliminate the paper-style references and talk about the history, making references to the final list of references. For now, just a reference list, on the next page.

Bibliography

[Be] MIHIR BELLARE. Practice-oriented provable security. Available via `http://www-cse.ucsd.edu/users/mihir/crypto-papers.html`.

[DH] WHITFIELD DIFFIE AND MARTIN HELLMAN. New directions in cryptography. *IEEE Trans. Info. Theory*, Vol. IT-22, No. 6, November 1976, pp. 644–654.

[Go97] ODED GOLDREICH, "The Foundations of Cryptography—An Essay." Full version of invited paper appearing at CRYPTO '98. `http://theory.lcs.mit.edu/~oded/foc.html`

[GM] SHAFI GOLDWASSER AND SILVIO MICALI. Probabilistic encryption. *J. of Computer and System Sciences*, Vol. 28, April 1984, pp. 270–299.

[Ka] DAVID KAHN. The Codebreakers; The Comprehensive History of Secret Communication from Ancient Times to the Internet. Scribner, Revised edition, December 1996.

[Od] ANDREW ODLYZKO. The rise and fall of knapsack cryptosystems. Available via `http://www.research.att.com/~amo/doc/cnt.html`.

[Ri90] RONALD RIVEST, "Cryptography." From the *Handbook of Theoretical Computer Science*, edited by J. van Leeuwen, Elsevier Science Publishers B.V., 1990.