

1

INTRODUCTION

It was a dark and stormy night. Somewhere in the distance a dog howled. A shiny object caught Alice's eye. A diamond cufflink! Only one person in the household could afford diamond cufflinks! So it was the butler, after all! Alice had to warn Bob. But how could she get a message to him without alerting the butler? If she phoned Bob, the butler might listen on an extension. If she sent a carrier pigeon out the window with the message taped to its foot, how would Bob know it was Alice that was sending the message and not Trudy attempting to frame the butler because he spurned her advances?

That's what this book is about. Not much character development for Alice and Bob, we're afraid; nor do we really get to know the butler. But we do discuss how to communicate securely over an insecure medium.

What do we mean by "communicating securely"? Alice should be able to send a message to Bob that only Bob can understand, even though Alice can't avoid having others see what she sends. When Bob receives a message, he should be able to know for certain that it was Alice who sent the message, and that nobody tampered with the contents of the message in the time between when Alice launched the message and Bob received it.

What do we mean by an "insecure medium"? Well, in some dictionary or another, under the definition of "insecure medium" should be a picture of the Internet. The world is evolving towards interconnecting every computer, household appliance, automobile, child's toy, and embedded medical device, all into some wonderful global internetwork. How wonderful! You'd be able to control your nuclear power plant with simple commands sent across the network while you were vacationing in Fiji. Or sunny Havana. Or historic Pyongyang. Inside the network the world is scary. There are links that eavesdroppers can listen in on. Information needs to be forwarded through packet switches, and these switches can be reprogrammed to listen to or modify data in transit.

The situation might seem hopeless, but we may yet be saved by the magic of cryptography, which can take a message and transform it into a bunch of numbers known as ciphertext. The ciphertext is unintelligible gibberish except to someone who knows the secret to reversing the transformation. Cryptography allows us to disguise our data so that eavesdroppers gain no information from listening to the information as transmitted. Cryptography also allows us to create an unforgeable message and detect if it has been modified in transit. One method of accomplishing this is with a **digital signature**, a number associated with a message and its sender that can be verified as

authentic by others, but can only be generated by the sender. This should seem astonishing. How can there be a number you can verify but not generate? A person's handwritten signature can (more or less) only be generated by that person, though it can (more or less) be verified by others. But it would seem as if a number shouldn't be hard to generate, especially if it can be verified. Theoretically, you could generate someone's digital signature by trying lots of numbers and testing each one until one passed the verification test. But with the size of the numbers used, it would take too much compute time (for instance, several universe lifetimes) to generate the signature that way. So a digital signature has the same property as a handwritten signature (theoretically) has, in that it can only be generated by one person but can be verified by lots of people. But a digital signature does more than a handwritten signature. Since the digital signature depends on the contents of the message, if someone alters the message the signature will no longer be correct, and the tampering will be detected. This will all become clear if you read Chapter 2 *Introduction to Cryptography*.

Cryptography is a major theme in this book, not because cryptography is intrinsically interesting (which it is), but because many of the security features people want in a computer network can best be provided through cryptography.

1.1 OPINIONS, PRODUCTS

Opinions expressed are those of the authors alone (and possibly not even agreed upon by all the authors). Opinions do not necessarily represent the views of any of the authors' past, current, or future employers. Any mention of commercial products or reference to commercial organizations is for information only. It does not imply recommendation or endorsement by NIST or any of the current, future, or prior organizations employing any of the authors.

1.2 ROADMAP TO THE BOOK

We aim to make this book comprehensible to engineers, giving intuition about designs. But readability doesn't mean lack of technical depth. We try to go beyond the information one might find in specifications to give insight into the designs. Given that specifications are easily available on the web today, we do not give exact packet formats.

This book should be usable as a textbook at either the undergraduate or graduate level. Most of the chapters have homework problems at the end. And to make life easier for professors who want to use the book, we will provide slides and an answer manual (to the professors using the

book) for many of the chapters. Even if you are not taking a class, you might want to do the homework problems. This book should be understandable to anyone with technical curiosity, a sense of humor*, and a good night's sleep in the recent past. The chapters are:

- **Chapter 1 *Introduction*:** This gives an overview of the chapters and some network basics.
- **Chapter 2 *Introduction to Cryptography*:** This explains the cryptographic principles which will be covered in more detail in later chapters.
- **Chapter 3 *Secret Key Cryptography*:** This chapter explains the uses of secret key cryptographic algorithms and describes how cryptographers create these algorithm.
- **Chapter 4 *Modes of Operation*:** Since most secret key algorithms encrypt a fairly small (*e.g.*, 128 bits) block, this chapter explains various algorithms to efficiently and securely encrypt arbitrarily large amounts of data.
- **Chapter 5 *Cryptographic Hashes*:** This chapter explains what hashes are used for and intuition into the methods by which cryptographers create secure and efficient hashes.
- **Chapter 6 *First-Generation Public Key Algorithms*:** This chapter describes the designs of the current widely deployed public key algorithms. Unfortunately, these would be insecure if the world were able to create a sufficiently large quantum computer. So the world will migrate to different public key algorithms that we'll describe in Chapter 8 *Post-Quantum Cryptography*.
- **Chapter 7 *Quantum Computing*:** This chapter gives an intuitive understanding of how a quantum computer differs from a classical computer, as well as explaining the intuition behind the two major cryptography-relevant quantum algorithms (Grover's and Shor's).
- **Chapter 8 *Post-Quantum Cryptography*:** This describes the types of math problems that would remain difficult to solve even if there were quantum computers, how to turn them into public key algorithms, and various optimizations that can make them efficient.
- **Chapter 9 *Authentication of People*:** This chapter describes the challenges involved in authenticating humans, and various types of technology that are or could be deployed.
- **Chapter 10 *Trusted Intermediaries*:** This chapter describes technologies for distributing cryptographic keys. It also talks about trust model issues in today's deployed designs.
- **Chapter 11 *Communication Session Establishment*:** This chapter describes conceptual issues in doing mutual authentication handshakes and establishing secure sessions.
- **Chapter 12 *IPsec*:** This chapter goes into detail about the design of IPsec.
- **Chapter 13 *SSL/TLS and SSH*:** This chapter goes into detail about the design of SSL/TLS and SSH.

*Although a sense of humor is not strictly necessary for understanding the book, it is an important characteristic to have in general.

- **Chapter 14 Electronic Mail Security:** This chapter describes various issues and solutions involved in electronic mail.
- **Chapter 15 Electronic Money:** This chapter describes various goals of electronic money and various technologies to address them. It covers cryptocurrencies and anonymous cash.
- **Chapter 16 Cryptographic Tricks:** This describes various exotic technologies such as secure multiparty computation and homomorphic encryption, as well as widely used technologies such as secret sharing.
- **Chapter 17 Folklore:** This chapter gives a summary of some of the design lessons discussed in the rest of the book and also describes some common misconceptions.
- **Glossary:** We define many of the terms we use in the book.
- **Math:** This provides more in-depth coverage of the mathematics used in the rest of the book. The appendix is written solely by Mike Speciner. It's not essential for appreciating the rest of the book. It's a sample of the content in Mike's github repositories <https://github.com/ms0/>.

1.3 TERMINOLOGY

Computer science is filled with ill-defined terminology used by different authors in conflicting ways. Some people take terminology very seriously, and once they start to use a certain word in a certain way, are extremely offended if the rest of the world does not follow.

When I use a word, it means just what I choose it to mean—neither more nor less.

—Humpty Dumpty (in *Through the Looking Glass*)

Some terminology we feel fairly strongly about. We do *not* use the term *hacker* to describe the vandals that break into computer systems. These criminals call themselves hackers, and that is how they got the name. But they do not deserve the name. True hackers are master programmers, incorruptibly honest, unmotivated by money, and careful not to harm anyone. The criminals termed “hackers” are not brilliant and accomplished. It is really too bad that they not only steal money, people’s time, and worse, but they’ve also stolen a beautiful word that had been used to describe some remarkable and wonderful people. We instead use words like *intruder*, *bad guy*, and *impostor*.

We grappled with the terms *secret key* and *public key* cryptography. Often in the security literature the terms *symmetric* and *asymmetric* are used instead of *secret* and *public*. When we say *secret key*, we mean a key that is used both for encryption and decryption. When we say *public key*, we are referring to a key pair consisting of a public key (used for encryption or signature verifica-

tion) and a private key (used for decryption or signing). Using the terms *public key* and *private key* is occasionally regrettable because both the words *public* and *private* start with “p”.

We use the term *privacy* when referring to the desire to keep communication from being seen by anyone other than the intended recipients. Some people in the security community avoid the term *privacy* because they feel its meaning has been corrupted to mean *the right to know*, because in some countries there are laws known as *privacy laws* that state that citizens have the right to see records kept about them. *Privacy* also tends to be used when referring to keeping personal information about people from being collected and misused. The security community also avoids the use of the word *secrecy*, because *secret* has special meaning within the military context, and they feel it would be confusing to talk about the secrecy of a message that was not actually labeled *top secret* or *secret*. The term most commonly used in the security community for keeping communication from being seen is *confidentiality*. We find that strange because *confidential*, like *secret*, is a security label, and the security community should have scorned use of *confidential*, too. In the first edition, we chose not to use *confidentiality* because we felt it had too many syllables, and saw no reason not to use *privacy*. For the second edition we reconsidered this decision, and were about to change all use of *privacy* to *confidentiality* until one of us pointed out we’d have to change the book title to something like *Network Security: Confidential Communication in a Non-Confidential World*, at which point we decided to stick with *privacy*.

Speaker: *Isn’t it terrifying that on the Internet we have no privacy?*

Heckler₁: *You mean confidentiality. Get your terms straight.*

Heckler₂: *Why do security types insist on inventing their own language?*

Heckler₃: *It’s a denial-of-service attack.*

—Overheard at gathering of security types

We often refer to things involved in a conversation by name; for instance, *Alice* and *Bob*, whether the things are people or computers. This is a convenient way of making descriptions unambiguous with relatively few words, since the pronoun *she* can be used for Alice, and *he* can be used for Bob. It also avoids lengthy inter-author arguments about whether to use the politically incorrect *he*, a confusing *she*, an awkward *he/she* or *(s)he*, an ungrammatical *they*, an impersonal *it*, or an awkward rewriting to avoid the problem. We remain slightly worried that people will assume when we’ve named things with human names that we are always referring to people. Assume Alice, Bob, and the rest of the gang may be computers unless we specifically say something like *the user Alice*, in which case we’re talking about a human.

When we need a name for a bad guy, we usually choose *Trudy* (since it sounds like *intruder*) or *Eve* (since it sounds like *eavesdropper*) or *Mallory* (since it sounds like *malice*). Everyone would assume Alice, Eve, and Trudy are she, and Bob is he. For inclusivity, we wanted at least one of the evil characters to be male, and we chose Mallory as the name of a male evildoer. Mallory can be

used for either gender, and is gaining more popularity as a female name, but when we use Mallory we will assume Mallory is male and use the pronoun *he*.

With a name like yours, you might be any shape, almost.

—Humpty Dumpty to Alice (in *Through the Looking Glass*)

Occasionally, one of the four of us authors will want to make a personal comment. In that case we use *I* or *me* with a subscript. When it's a comment that we all agree with, or that we managed to slip past me₃ (the rest of us are wimpier), we use the term *we*.

1.4 NOTATION

We use the symbol \oplus (pronounced *ex-or*) for bitwise-exclusive-or. We use the symbol $|$ for concatenation. We denote encryption with curly brackets followed by the key with which something was encrypted, as in $\{message\}K$, which means *message* is encrypted with *K*. We denote a signature with square brackets followed by the key, as in $[message]_{Bob}$. Sometimes the key is a subscript, and sometimes not. There is no deep meaning to that. Honestly, it's that sometimes we are using a key that has subscripts, such as K_{Alice} , and the formatting tool we are using makes it very difficult to have a subscripted subscript.

1.5 CRYPTOGRAPHICALLY PROTECTED SESSIONS

When Alice and Bob use modern cryptography and protocols, such as IPsec (Chapter 12) or TLS (Chapter 13), they first exchange a few messages in which they establish session secrets. These session secrets allow them to encrypt and integrity-protect their conversation. Although their physical connectivity is a path across the Internet, once Alice and Bob create the protected session, data that they send to each other is as trustworthy as if they had a private physically protected link.

There are various terms for this type of protected session. We will usually refer to it as a **secure session**. It is considered good security practice to use several cryptographic keys in a secure session between Alice and Bob. For example, there might be different session keys for

- encryption of Alice to Bob traffic,
- encryption of Bob to Alice traffic,

-
- integrity protection of Alice to Bob traffic, and
 - integrity protection of Bob to Alice traffic.

Alice and Bob will each have a database describing their current secure sessions. The information in the database will include information such as how the session will be identified on incoming data, who is on the other end of the session, which cryptographic algorithms are to be used, and the sequence numbers for data to be sent or received on the session.

1.6 ACTIVE AND PASSIVE ATTACKS

A **passive attack** is one in which the intruder eavesdrops but does not modify the message stream in any way. An **active attack** is one in which the intruder may transmit messages, replay old messages, modify messages in transit, or delete or delay selected messages in transit. Passive attacks are less risky for the attacker, because it is tricky to detect or prove someone has eavesdropped. If the attacker is not on the path between Alice and Bob, the passive attack can be done by having an accomplice router make copies of the traffic and send them to the attacker, who can then analyze the data later, in private.

A typical active attack is one in which an intruder impersonates one end of the conversation, or acts as a **meddler-in-the-middle (MITM)**. (Note that the acronym MITM used to be expanded to be man-in-the-middle, but the industry is trying to move to more inclusive language. In this case, it is acknowledging that being annoying is not gender-specific.) A MITM attack is where an active attacker, say, Trudy, acts as a relay between two parties (Alice and Bob), and rather than simply forwarding messages between Alice and Bob, modifies, deletes, or inserts messages. If Trudy were faithfully forwarding messages, she could be acting as a passive eavesdropper, or she could be a correctly functioning router.

If Alice communicates with Bob using a secure session protocol with strong cryptographic protection, Trudy would gain no information by eavesdropping and would not be able to modify messages without being detected.

However, Trudy might be able to impersonate Bob's IP address to Alice, tricking Alice into establishing a secure session between Alice and Trudy. Then Trudy can simultaneously impersonate Alice to Bob, and establish a secure session between Trudy and Bob. (See Figure 1-1.)

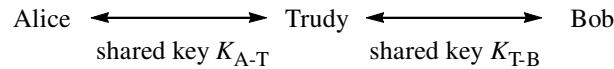


Figure 1-1. MITM Attack

Alice and Bob will think they are talking to each other, but in fact they are each talking to Trudy. Data sent by Alice to Bob will be decrypted by Trudy using the session secret for the Alice-Trudy secure session, and encrypted for Bob with the session secret for the Trudy-Bob secure session. It is difficult for Alice and Bob to know that they have a MITM. Alice could attempt to make sure she's really talking to Bob by asking questions such as "What did I order when we first met for dinner?", but Trudy can forward the questions and answers. We will explain in §11.6 *Detecting MITM* how Alice and Bob can detect a MITM. And as we will explain in later chapters, if Alice has credentials for Bob that Trudy cannot impersonate, Alice and Bob can prevent a MITM.

1.7 LEGAL ISSUES

The legal aspects of cryptography are fascinating, but the picture changes quickly, and we are certainly not experts in law. Although it pains us to say it, if you're going to build anything involving cryptography, talk to a lawyer. The combination of patents and export controls slowed down deployment of cryptographically secure networking, and caused strange technical choices.

1.7.1 Patents

One legal issue that affects the choice of security mechanisms is patents. Most cryptographic techniques were covered by patents and historically this has slowed their deployment. One of the important criteria for NIST's selection of algorithms (such as AES [§3.7 *Advanced Encryption Standard (AES)*], SHA-3 [§5.6.2 *Construction of SHA-3*], and post-quantum algorithms [Chapter 8 *Post-Quantum Cryptography*]) is whether they are royalty-free.

The widely deployed RSA algorithm (see §6.3) was developed at MIT, and under the terms of MIT's funding at the time, there were no license fees for U.S. government use. It was only patented in the U.S., and licensing was controlled by one company, which claimed that the Hellman-Merkle patent also covered RSA, and that patent is international. Interpretation of patent rights varies by country, so the legal issues were complex. At any rate, the last patent on RSA ran out on 20 September 2000. There were many parties on that day.

"I don't know what you mean by your way," said the Queen: "all the ways about here belong to me..."

—*Through the Looking Glass*

To avoid large licensing fees, many protocol standards used DSA (see §6.5) instead of RSA. Although in most respects DSA is technically inferior to RSA, when first announced it was advertised that DSA would be freely licensable so it would not be necessary to reach agreement with the RSA-licensing company. But the company claimed Hellman-Merkle covered all public key cryptography, and strengthened its position by acquiring rights to a patent by Schnorr that was closely related to DSA. Until the patents expired (and luckily the relevant patents have expired), the situation was murky.

1.7.2 Government Regulations

*Mary had a little key
(It's all she could export)
And all the email that she sent
Was opened at the Fort.*

—Ron Rivest

The U.S. government (as well as other governments) used to impose severe restrictions on export of encryption. This caused much bitterness in the computer industry and led to some fascinating technical designs so that domestic products, which were legally allowed to use strong encryption, could use strong encryption where possible, and yet interoperate with exportable products that were not allowed to use strong encryption. Although U.S. companies still need permission from the Department of Commerce to export products containing cryptography, since around the year 2000 it has been easy to get products approved.

Additionally, even today, some countries have usage controls, so even if it were legal to export a product, it might not be legally usable inside some other country. For instance, some countries have developed their own cryptographic algorithms, and they want all their citizens to use those. Most of the reason for these sorts of rules is so that a government can't be prevented from accessing data, for instance, for law enforcement purposes.

1.8 SOME NETWORK BASICS

Although I₂ get a bit frustrated with teaching network concepts as if TCP/IP is the only way, or the best way to design a network, this is what today's Internet is built with, so we need to understand some of the details. Here is a brief introduction.

1.8.1 Network Layers

A good way of thinking about networking concepts is with *layers*. The concept of a layer is that inside a node, there are interfaces to adjacent layers (the layer above or the layer below). Between nodes, there are protocols for talking to peer layers. The actual protocol inside a layer can in theory be replaced by a layer that gives similar functionality to the adjacent layers. Although layers are a good way to learn about networks, deployed networks do not cleanly follow a layering model. Layers often use data associated with layers other than peer layers or adjacent layers. Layers are often subdivided into more layers, and an implementation might merge layers. ISO (International Organization for Standardization) defined a model with seven layers. The bottom layers look like this:

- Layer 1, **physical layer**. Defines how to send a stream of bits to a neighbor node (neighbors reside on the same link).
- Layer 2, **data link layer**. Defines how to structure a string of bits (provided by layer 1) into packets between neighbor nodes. This requires using the stream of bits to signal information such as “this is the beginning of a packet”, “this is the end of a packet”, and an integrity check.
- Layer 3, **network layer**. This allows a source node to send a packet of information across many links. The source adds header information to a packet to let the network know where to deliver the packet. This is analogous to putting a postal message inside an envelope, and writing the destination on the envelope. A network will consist of many links. Nodes known as *routers* or *switches* forward between links. Such nodes are connected to two or more links. They have a table known as a *forwarding table* that tells them which link to forward on, to get closer to the destination. Usually, network addresses are assigned hierarchically, so that a bunch of addresses can be summarized in one forwarding entry. This is analogous to the post office only needing to look at the destination country, and then once inside that country, forwarding towards the state, and once inside the state, forwarding to the destination city, etc. The usual protocol deployed in the Internet today for layer 3 is IP (Internet Protocol), which basically consists of adding a header to a packet identifying the source and destination, a hop count (so the network can discard packets that are looping), and other information. There are two versions of IP. IPv4 has 32-bit addresses. IPv6 has 128-bit addresses. One extra piece of information in the IP header is the 16-bit “protocol type”, which indicates which layer 4 protocol is sending the data.
- Layer 4, **transport layer**. This is information that is put in by the source, and interpreted at the destination. The service provided by TCP (Transmission Control Protocol, RFC 793) to the layer above it consists of accepting a stream of bytes at the source, and delivering the stream of bytes to the layer above TCP at the destination, without loss or duplication. To accomplish this, TCP at the sender numbers bytes; TCP at the destination uses the sequence numbers to acknowledge receipt of data, reorder data that has arrived out of sequence, and

ask for retransmission of lost data. UDP (User Datagram Protocol, RFC 768) is another layer 4 protocol that does not worry about lost or reordered data. Many processes in the layer above TCP or UDP will be reachable at the same IP address, so both UDP and TCP headers include **ports** (one for source, and one for destination), which tell the destination which process should receive the data.

1.8.2 TCP and UDP Ports

There are two 16-bit fields in TCP and UDP—a source port and a destination port. Typically an application on a server will be reachable at a “well-known port”, meaning that the port is specified in the protocol. If a client wants to reach that application at a server, the protocol type field in the IP header will be either TCP (6) or UDP (17) and the destination port field in the layer 4 header (TCP or UDP in this case) will be the well-known port for that application. For example, HTTP is at port 80, and HTTPS is at port 443. The source port will usually be a dynamically assigned port (49152 through 65535).

1.8.3 DNS (Domain Name System)

Another aspect of Internet networking we will be discussing is DNS. It is basically a distributed directory that maps DNS names (*e.g.*, `example.com`) to IP addresses. DNS names are hierarchical. A simple way to think of DNS is that for each level in the DNS name (*e.g.*, root, `.org`, `.com`, `example.com`) there is a server that keeps a directory associated with names in that level. The root would have a directory that allows looking up servers for each of the top-level domains (TLDs) (*e.g.*, `.org`, `.com`, `.gov`, `.tv`). There are currently over a thousand TLDs, so the root would have information associated with each of those TLDs in its database. In general, to find a DNS name, a node starts at the root, finds the server that holds the directory for the next level down, and keeps going until it gets to the server that stores information about the actual name. There are several advantages to DNS being hierarchical.

- Someone that wishes to purchase a DNS name has a choice of organizations from which to purchase a name. If a name is purchased from the organization managing names in the TLD `.org`, the purchased name will be of the form `example.org`. If you purchase the name `example.org`, you can then name anything that would be below that name in the DNS hierarchy, such as `xyz.example.com` or `labs.xyz.example.com`.
- The DNS database will not become unmanageably large, because no organization needs to keep the entire DNS database. In fact, nobody knows how many names are in the DNS database.

- It is fine to have the same lower level name in multiple databases. For instance, there is no problem with there being DNS names `example.com` and `example.org`.

1.8.4 HTTP and URLs

When we access things on the web, we use a protocol known as HTTP (hypertext transfer protocol). HTTP allows specifying more than a DNS name; it allows specifying a particular web page at the service with a DNS name. The URL (uniform resource locator) is the address of the web page. The URL contains a DNS name of the service, followed by additional information that is interpreted solely by the server that receives the request. The additional information might be, for instance, the directory path at the destination server that finds the information to construct the page being requested.

Sometimes humans type URLs, but usually URLs are displayed as links in a webpage that can be clicked on. It is common for people to do an Internet search (*e.g.*, using Google or Bing) for something, and then click on choices. URLs can be very long and ugly, and people usually don't look at the URL they click on. Often the web page that displays a link does not display the actual URL. Mousing over the link will sometimes show the human a URL. Unfortunately, the web page can choose what to display on the page as the link, and what to display the mouse-over link as. These can be different from the actual URL that will be followed if the link is clicked on. For example, on a web page, a clickable link (which is usually displayed in a different color), might display as "click here for information", and if a suspicious user moused-over the link, it might display "`http://www.example.com/information`", but if the user clicks on the link, the malicious webpage could send them to any URL, *e.g.*, `http://www.rentahitman.com`.

The two main HTTP request types are `GET` and `POST`. `GET` is for reading a web page and `POST` is for sending information to a web server. The response contains information such as the content requested and status information (such as "OK" or "not found" or "unauthorized"). One status that might be included in a response is a redirect. This informs the browser that it should go to a different URL. The browser will then go to the new URL, as if the user had clicked on a link.

1.8.5 Web Cookies

If a client is browsing content that requires authentication and access control, or is accumulating information such as items in a virtual shopping basket to be purchased when the user is finished browsing the on-line catalog, the information for that session needs to be kept somewhere. But HTTP is stateless. Each request/response interaction is allowed to take place over a fresh TCP connection. The cookie mechanism enables the server to maintain context across many request/response interactions. A **cookie** is a piece of data sent to the client by the server in response

to an HTTP request. The cookie need not be interpreted by the client. Instead, the client keeps a list of DNS names and cookies it has received from a server with that DNS name. If the client made a request at example.com, and example.com sent a cookie, the client will remember (example.com: cookie) in its cookie list. When the client next makes an HTTP request to example.com, it searches its cookie database for any cookies received from example.com, and includes those cookies in its HTTP request.

The cookie might contain all the relevant information about a user, or the server might keep a database with this information. In that case, the cookie only needs to contain the user's identity (so the server can locate that user in its database), along with proof that the user has already authenticated to the server. For example, if Alice has authenticated to Bob, Bob could send Alice a cookie consisting of some function of the name "Alice" and a secret that only Bob knows. A cookie will be cryptographically protected by the server in various ways. It might be encrypted with a key that only the server knows. It might contain information that only allows the cookie to be used from a specific machine. And it is almost always protected when transmitted across the network because the client and server will be communicating over a secure session.

1.9 NAMES FOR HUMANS

It is sometimes important that an identifier for a human be unique, but it is not important for a human to have a *single* unique name. Humans have many unique identifiers. For example, an email address, a telephone number, or a username specific to a website. In theory, nobody but the human needs to know that the various identities refer to the same human, but, unfortunately, it has become easy for organizations to correlate various identities. Also, it is not uncommon for family members or close friends to share an account, so an email address or username at a website might actually be multiple humans, but we will ignore that issue.

Human names are problematic. Consider email addresses. Typically, companies let the first John Smith use the name `John@companyname` for his email address, and then perhaps the next one will be `Smith@companyname`, and the next one `JSmith@companyname`, and the next one has to start using middle initials. Then, to send to your colleague John Smith, you have to do the best you can to figure out which email address in the company directory is the one you want, based on various attributes (such as their location or their job title, if you are lucky enough to have this information included in the directory). There will be lots of confusion when one John Smith gets messages intended for a different John Smith. This is a problem for both the John Smith that is mistakenly sent the email, as well as the John Smith who was the intended recipient of the email. Usually, a person can quickly delete spam, but with a name like John Smith, irrelevant-looking email might actually be important email for a different John Smith in the company, so must be carefully

read, and forwarded just in case. And the unfortunate John Smith who received the email has the problem of figuring out which John Smith he should forward the email to.

One way of solving this problem is that once a company hired someone with a particular name, they just wouldn't hire another. I₂ (with the name Radia Perlman, which is probably unique in the entire world) think that's reasonable, but someone with a name like John Smith might start having problems finding a company that could hire him.

Now why did you name your baby John? Every Tom, Dick, and Harry is named John.

—Sam Goldwyn

1.10 AUTHENTICATION AND AUTHORIZATION

Authentication is when Alice proves to Bob that she is Alice. Authorization is having something decide what a requester is allowed to do at service Bob.

1.10.1 ACL (Access Control List)

Typically the way a server decides whether a user should have access to a resource is by first authenticating the user, and then consulting a database associated with the resource that indicates who is allowed to do what with that resource. For instance, the database associated with a file might say that Alice can read it, and George and Carol are allowed to read and write it. This database is often referred to as an **ACL (access control list)**.

1.10.2 Central Administration/Capabilities

Another model of authorization is that instead of listing, with each resource, the set of authorized users and their rights (*e.g., read, write, execute*), service Bob might have a database that listed, for each user, everything she was allowed to do. If everything were a single application, then the ACL model and the central administration model would be basically the same, since in both cases there would be a database that listed all the authorized users and what rights each had. But in a world in which there are many resources, not all under control of the same organization, it would be difficult to have a central database listing what each user was allowed to do. This model would have scaling problems if there were many resources each user was allowed to access, especially if resources

were created and deleted at a high rate. And if resources are under control of different organizations, there wouldn't be a single organization trusted to manage the authorization information.

Some people worry that ACLs don't scale well if there are many users allowed access to each resource. But the concept of groups helps the scaling issue.

1.10.3 Groups

Suppose there were a file that should be accessible to, say, any Dell employee. It would be tedious to type all the employee names into that file's ACL. And for a set of employees such as "all Dell employees", there would likely be many resources with the same set of authorized users. And it would take a lot of storage to have such huge ACLs on that many resources, and whenever anyone joined or left the company, all those ACLs would have to be modified.

The concept of a group was invented to make ACLs more scalable. It is possible to include a group name on an ACL, which means that any member of the group is allowed access to the resource. Then, group membership can be managed in one place, rather than needing to update ACLs at every resource when the membership changes.

Traditionally, a server that protected a resource with a group named on the ACL needed to know all the members of the group, but if there are many servers that store resources for that group, this would be inefficient and inconvenient. Also, that model would preclude more flexible group mechanisms; for example:

- cross-organizational groups, where no one server is allowed to know all the members;
- anonymous groups, where someone can prove membership in the group without having to divulge their identity.

Traditionally, groups were centrally administered, so it was easy to know all the groups to which a user belonged, and the user would not belong to many groups. But in many situations, it is useful for any user to be able to create a group (such as Alice's friends, or students who have already turned in their exams in my course), and have anyone be able to name such a group on an ACL.

Scaling up this simple concept of users, groups, and ACLs to a distributed environment has not been solved in practice. This section describes various ways that it might be done and challenges with truly general approaches.

1.10.4 Cross-Organizational and Nested Groups

It would be desirable for an ACL to allow any Boolean combination of groups and individuals. For instance, the ACL might be the union of six named individuals and two named groups. If someone is one of the named individuals, or in one of the groups, the ACL would grant them permission. It

would also be desirable to have the ACL be something like Group A and NOT group B, *e.g.*, U.S. citizen and not a felon.

Likewise, it would be desirable for group membership to be any Boolean combination of groups and individuals, *e.g.*, the members of **Alliance-executives** might be **CompanyA-execs**, **CompanyB-execs**, and **John Smith**. Each of the groups **Alliance-executives**, **CompanyA-execs**, and **CompanyB-execs** is likely to be managed by a different organization, and the membership is likely to be stored on different servers. How, then, can a server (Bob) that protects a resource that has the group **Alliance-executives** on the ACL know whether to allow Alice access? If she's not explicitly listed on the ACL, she might be a member of one of the groups on the ACL. But Bob does not necessarily know all the members of the group. Let's assume that the group name (**Alliance-executives**) can be looked up in a directory to find out information such as its network address and its public key. Or perhaps the group name would contain the DNS name of the server that manages that group, so the group name might be `example.com/Alliance-executives`.

- Bob could periodically find every group on any ACL on any resource it protects, and attempt to collect the complete membership. This means looking up all the members of all subgroups, and subgroups of subgroups. This has scaling problems (the group memberships might be very large), performance problems (there might be a lot of traffic with servers querying group membership servers for membership lists), and cache staleness problems. How often would this be done? Once a day is a lot of traffic, but a day is a lot of time to elapse for Alice's group membership to take effect, and for revocations to take effect.
- When Alice requests access, Bob could then ask the on-line group server associated with the group whether Alice is a member of the group. This could also be a performance nightmare with many queries to the group server, especially in the case of unauthorized users creating a denial of service attack by requesting access to services. At the least, once Alice is discovered to either belong or not belong, Bob could cache this information. But again, if the cache is held for a long time, it means that membership can take a long time to take effect, and revocation can also take a long time to take effect.
- When Alice requests access to a resource, Bob could reply, "You are not on the ACL, but here are a bunch of groups that are on the ACL, so if you could prove you are a member of one of those groups, you can access it." Alice could then contact the relevant group servers and attempt to get certification of group membership. Then, she can reply to Bob with some sort of proof that she is a member of one of the groups on the ACL. Again, this could have a cache staleness problem, and a performance problem if many users (including unauthorized troublemakers) contact servers asking for proof of group membership.

1.10.5 Roles

The term *role* is used in many different ways. Authorization based on roles is referred to as **RBAC (role-based access control)**. In most usage today, a **role** is the same as what we described as a group. In some cases, Alice will need to log in as the role rather than as the individual. For example, she might log in as `admin`. However, many users might need to log in as `admin`, and for auditing purposes, it is important to know which user was invoking the `admin` role. Therefore, for auditing purposes, a user might be simultaneously logged in as the role (`admin`) and as the individual (Alice). In many environments, a role is given a name such as “`admin`”, and being able to invoke the `admin` role on your own machine should not authorize you to invoke the `admin` role on controlling a nuclear power plant. So, both groups and roles should have names that specify the domain, *e.g.*, the DNS name of the service that manages the group or role membership.

Although a lot of systems implement roles as the same thing as groups, it is confusing to have two words that mean the same thing. We recommend that the difference between a group and a role is that a role needs to be consciously invoked by a user, often requiring additional authentication such as typing a different password. In contrast, we recommend that with a group, all members automatically have all rights of the group, without even needing to know which groups they are a member of. With roles, users may or may not be allowed to simultaneously act in multiple roles, and perhaps multiple users may or may not be allowed to simultaneously act in a particular role (like *President of the United States*). Again, we are discussing various ways things could work. Deployed systems make different choices. Some things people would like to see roles solve:

- When a user is acting in a particular role, the application presents a different user interface. For instance, when a user is acting as *manager*, the expense reporting utility might present commands for approving expense reports, whereas when the user is acting as *employee*, the application might present commands for reporting expenses.
- Having roles enables a user to be granted a subset of all the permissions they might have. If the user has to explicitly invoke the privileged role, and only keeps themselves authenticated as that role for as long as necessary, it is less likely that a typo will cause them to inadvertently do an undesirable privileged operation.
- Allowing a user to be able to run with a subset of her rights (not invoking her most privileged role except when necessary) gives some protection from malicious code. While running untrusted code, the user should be careful to run in an unprivileged role.
- Sometimes there are complex policies, such as that you are allowed to read either file A or file B but not both. Somehow, proponents of roles claim roles will solve this problem. This sort of policy is called a **Chinese wall**.

1.11 MALWARE: VIRUSES, WORMS, TROJAN HORSES

Lions and tigers and bears, oh my!

—Dorothy (in the movie *The Wizard of Oz*)

People like to categorize different types of malicious software and assign them cute biological terms (if one is inclined to think of worms as cute). We don't think it's terribly important to distinguish between these things, so in the book we'll refer to all kinds of malicious software generically as **malware**. However, here are some of the terms that seem to be infecting the literature.

- **Trojan horse**—instructions hidden inside an otherwise useful program that do bad things. Usually, the term *Trojan horse* is used when the malicious instructions are installed at the time the program is written (and the term *virus* is used if the instructions get added to the program later).
- **virus**—a set of instructions that, when executed, inserts copies of itself into other programs.
- **worm**—a program that replicates itself by installing copies of itself on other machines across a network.
- **trapdoor**—an undocumented entry point intentionally written into a program, often for debugging purposes, which can be exploited as a security flaw. Often people forget to take these out when the product ships. However, sometimes these undocumented “features” are intentionally put into software by an employee who thinks he might at some point become disgruntled.
- **bot**—a machine that has been infected with malicious code that can be activated to do some malicious task by some controller machine across the Internet. The controller is often referred to as a **bot herder**. The intention of the disease-infected rodents (apologies if we offend actual disease-infected rodents who might be reading this book) who turn a computer into a bot is that the owner of the machine should not be aware that their machine is infected. To the owner of the machine, it continues operating as usual. However, the controller can at some point rally all the bots under its control to do some sort of mischief like sending out spam or flooding some service with nuisance messages. The bots under its control are often referred to as a **bot army**. There are price lists on the dark web for renting a bot army, priced according to the size of the army and the amount of time you would like to rent them. (The **dark web** is a subset of the Internet that is not visible to search engines, and requires special access mechanisms. It is an ideal place for purchasing illegal goods.)
- **logic bomb**—malicious instructions that trigger on some event in the future, such as a particular time occurring. The delay might advantage criminals that create a logic bomb, because

they can first deploy the logic bomb in many places. Or, they can make the bad event occur long after they have left the company, so they won't be under suspicion.

- **ransomware**—malicious code that encrypts the user's data, and then the helpful criminal offers to help get the data back, for a small fee (such as several million dollars).

Most of the items above exploit bugs in operating systems or applications. There are also vulnerabilities that exploit bugs in humans. One example is **phishing**, the act of sending email to a huge unstructured list of email addresses, that will trick some small percentage of the recipients into infecting their own machines, or divulging information such as a credit card number. **Spear phishing** means sending custom messages to a more focused group of people.

1.11.1 Where Does Malware Come From?

Where do these nasties come from? Originally, it was hobbyists just experimenting with what they could do. Today, big money can be made with malware. A bot army can be rented to someone that wants to damage a competitor, or a political organization they disagree with. And malware can be used for demanding ransoms. It is also used by sophisticated spy organizations. A notable example is Stuxnet. This was malware aimed specifically at damaging a certain type of equipment (centrifuges) used in Iran, to slow Iran's ability to produce nuclear weapons.

How could an implementer get away with intentionally writing malware into a program? Wouldn't someone notice by looking at the program? A good example of how hard it can be to decipher what a program is doing, even given the source code, is the following nice short program, written by Ian Phillipps (see Figure 1-2).

It was a winner of the 1988 International Obfuscated C Code Contest. It is delightful as a Christmas card. It does nothing other than its intended purpose (I_1 have analyzed the thing carefully and I_2 have complete faith in me₁), but we doubt many people would take the time to understand this program before running it

But also, nobody looks. Often when you buy a program, you do not have access to the source code, and even if you did, you probably wouldn't bother reading it all, or reading it very carefully. Many programs that run have never been reviewed by anybody. A claimed advantage of the "open source" movement (where all software is made available in source code format) is that even if *you* don't review it carefully, there is a better chance that someone else will.

What does a virus look like? A virus can be installed in just about any program by doing the following:

- replace any instruction, say the instruction at location x , by a jump to some free place in memory, say location y ; then
- write the virus program starting at location y ; then

```

/* Have yourself an obfuscated Christmas! */
#include <stdio.h>
main(t,_,a)
char *a;
{
    return!0<t?t<3?main(-79,-13,a+main(-87,1-,main(-86,0,a+1)+a)) :
    1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
    main(2,_+1,"%s %d %d\n") : 9:16:t<0?t<-72?main(.,t,
    "@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,*{*,/w{%+,/w#q#n+,/#{1,+,/n{n+,/+#n+,/#\
    ;#q#n+,/+k#;*+,/'r :'d*'3,{w+K w'K:'+}e#';dq#'1 \
    q#+d'K#/!+k#;q#'r)eKK#)w'r)eKK{nl}'#/;#q#n'){})#w'){}){nl}'/+#n';d}rw' i;# \
    ){nl}!/n{n#'; r{#w'r nc{nl}'/#{1,'+K {rw' iK{;[{nl}]/w#q#n'wk nw' \
    iwk{KK{nl}!/#'{1##w# i;:{nl}'/*{(q#'ld;r'){nlwb!/*de}'c \
    ;:{nl}'-{}rw}'/+,}##'*#nc,',#nw]'/+kd'+e}+;#rdq#w! nr'/' ) }+}{rl#'{n' ')# \
    '+'}##(!!/")
:t<-50?___=a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s") :*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#1,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}

```

Figure 1-2. Christmas Card?

- place the instruction that was originally at location x at the end of the virus program, followed by a jump to $x+1$.

Besides doing whatever damage the virus program does, it might replicate itself by looking for any executable files in any directory and infecting them. Once an infected program is run, the virus is executed again, to do more damage and to replicate itself to more programs. Most viruses spread silently until some triggering event causes them to wake up and do their dastardly deeds. If they did their dastardly deeds all the time, they wouldn't spread as far.

1.11.2 Virus Checkers

How can a program check for viruses? There's rather a race between the brave and gallant people who analyze the viruses and write clever programs to detect and eliminate them, and the foul-smelling scum who devise new types of viruses that will escape detection by all the current virus checkers.

The oldest form of virus checker knows instruction sequences that have appeared in known viruses, but are believed not to occur in benign code. It checks all the files on disk and instructions in memory for those patterns of commands, and raises a warning if it finds a match hidden somewhere inside some file. Once you own such a virus checker, you need to periodically get updates of the patterns file that include the newest viruses.

To evade detection of their viruses, virus creators have devised what are known as a **polymorphic viruses**. When they copy themselves, they change the order of their instructions or change instructions to functionally similar instructions. A polymorphic virus may still be detectable, but it takes more work, and not just a new pattern file. Modern virus checkers don't just periodically scan the disk. They actually hook into the operating system and inspect files before they are written to disk.

Another type of virus checker takes a snapshot of disk storage by recording the information in the directories, such as file lengths. It might even take message digests of the files. It is designed to run, store the information, and then run again at a future time. It will warn you if there are suspicious changes. One virus, wary of changing the length of a file by adding itself to the program, compressed the program so that the infected program would wind up being the same length as the original. When the program was executed, the uncompressed portion containing the virus decompressed the rest of the program, so (other than the virus portion) the program could run normally.

Some viruses attack the virus checkers rather than just trying to elude them. If an attacker can penetrate a company that disseminates code, such as virus signatures or program patches, they can spread their malware to all the customers of that company.

1.12 SECURITY GATEWAY

A security gateway (see Figure 1-3) sits between your internal network and the rest of the network and provides various services. Such a box usually provides many functions, such as firewall (§1.12.1), web proxy (§1.12.2), and network address translation (§1.14). We will describe these features in the next few sections.

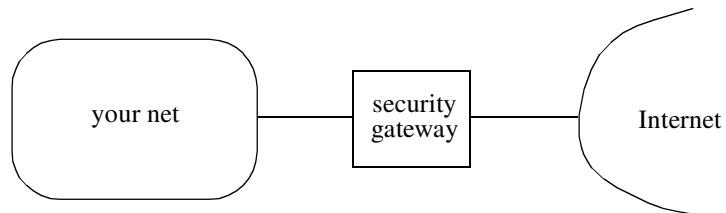


Figure 1-3. Security Gateway

1.12.1 Firewall

There was a time when people assumed that their internal network and all the users and systems on the internal network were trustworthy. The only challenge was connectivity to the Internet. Users

need to communicate with publicly available services. The company needs to make some of its own services available to customers located outside the corporate network. So the thought was to install a box between your network and the scary Internet that can keep things secure, somehow.

The belief that a firewall between your internal network and the scary Internet is all you need to protect you has long been discredited. Even if you had the most secure firewall, *i.e.*, a box that blocked any traffic between your network and the Internet, malware inside your network could do arbitrary damage. And even carefully configured firewalls often break legitimate usage.

A recent buzzword is **zero trust**. It basically means the opposite of the previous thinking. Although firewalls can add some amount of extra security, all applications must protect themselves. They must authenticate everything they are talking to and enforce access control rules. The buzzword **defense-in-depth** means having multiple stages of security, so if one fails, hopefully another will protect you. Although people pretty much agree that the old firewall-will-protect-you model is no longer the right way to think about security, a lot of the mechanisms supporting that model are still deployed.

Firewalls centrally manage access to services in ways that individual systems should, but often don't. Firewalls can enforce policies such as *systems outside the firewall can't access file services on any systems inside the firewall*. With such a restriction, even if the internal systems are more open than they should be, or have bugs, they can't be attacked directly from systems outside the firewall.

Note that a firewall need not be a physical box that the company buys. It could instead be software on each machine that does the same sorts of filtering that a firewall box would do. This concept is sometimes called a **distributed firewall**. Both “firewall” and “distributed firewall” are buzzwords, and, as with most buzzwords, the definitions evolve and are used by different vendors in different ways. Usually *distributed firewall* implies that all the components doing “firewall stuff” are centrally managed. Otherwise, it would just be multiple firewalls.

The simplest form of firewall selectively discards packets based on configurable criteria, such as addresses in the IP header, and does not keep state about ongoing connections. For example, it might be configured to only allow some systems on your network to communicate outside, or some addresses outside your network to communicate into your network. For each direction, the firewall might be configured with a set of legal source and destination addresses, and it drops any packets that don't conform. This is known as **address filtering**.

Packet filters usually look at more than the addresses. A typical security policy is that for certain types of traffic (*e.g.*, email, web surfing), the rewards outweigh the risks, so those types of traffic should be allowed through the firewall, whereas other types of traffic (say, remote terminal access), should not be allowed through.

To allow certain types of traffic between host A and B while disallowing others, a firewall can look at the protocol type in the IP header, at the ports in the layer 4 (TCP or UDP) header, and at anything at any fixed offset in the packet. For web traffic, either the source or destination port will likely be 80 (http) or 443 (https). For email, either the source or destination port will likely be 25.

Firewalls can be even fancier. Perhaps the policy is to allow connections initiated by machines inside the firewall, but disallow connections initiated by machines outside the firewall. Suppose machine A inside the firewall initiates a connection to machine B outside the firewall. During the conversation, the firewall will see packets from both directions (A to B as well as B to A), so it can't simply disallow packets from B to A. The way it manages to enforce only connections initiated by A, is to look at the TCP header. TCP has a flag (called ACK) that is set on all but the first packet, the one that establishes the connection. So if the firewall disallows packets from B without ACK set in the TCP header, then it will usually have the desired effect.

Another approach is a **stateful packet filter**, *i.e.*, a packet filter that remembers what has happened in the recent past and changes its filtering rules dynamically as a result. A stateful packet filter could, for instance, note that a connection was initiated from inside using IP address s , to IP address d , and then allow (for some period of time) connections from IP address d to IP address s .

1.12.2 Application-Level Gateway/Proxy

An application-level gateway, otherwise known as a **proxy**, acts as an intermediary between a client and the server providing a service. The gateway could have two network adaptors and act as a router, but more often it is placed between two packet-filtering firewalls, using three boxes (see Figure 1-4). The two firewalls are routers that refuse to forward anything unless it is to or from the gateway. Firewall F_2 refuses to forward anything from the global net unless the destination address is the gateway and refuses to forward anything to the global net unless the source is the gateway. Firewall F_1 refuses to forward anything from your network unless the destination address is the gateway, and refuses to forward anything to your network unless the source address is the gateway. To transfer a file from your network to the global network, you could have someone from inside transfer the file to the gateway machine, and then the file is accessible to be read by the outside world. Similarly, to read a file into your network, a user can arrange for it to first get copied to the gateway machine. To log into a machine in the global network, you could first log into the gateway machine, and from there you could access machines in the remote network. An application-level gateway is sometimes known as a **bastion host**. It must be implemented and configured to be very

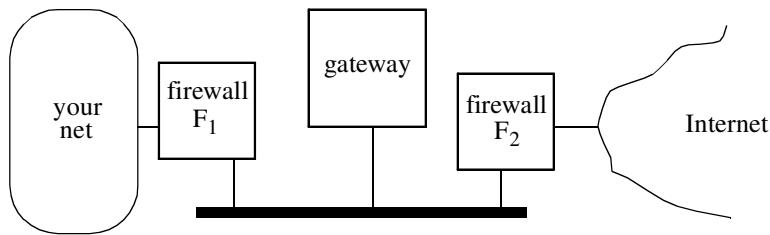


Figure 1-4. Application-Level Gateway

secure. The portion of the network between the two firewalls is known as the **DMZ** (demilitarized zone).

The gateway need not support every possible application. If the gateway does not support an application, either the firewalls on either side of the gateway should block that application (based on layer 4 port), or both firewalls would need to allow that application, depending on whether you want that application to work through the firewall. An example strategy is to allow only electronic mail to pass between your corporate network and the outside world. The intention is to specifically disallow other applications, such as file transfer and remote login. However, electronic mail can certainly be used to transfer files. Sometimes a firewall might specifically disallow very large electronic mail messages, on the theory that this will limit the ability to transfer files. But often, large electronic mail messages are perfectly legitimate, and any file can be broken down into small pieces.

Sometimes an application might specifically be aware of proxies running on an application gateway. For example, browsers can be configured with the address of a proxy, and then all requests will be directed to the proxy. The proxy might be configured with which connections are allowed and which are disallowed, and could even inspect the data passing through to check for malware.

1.12.3 Secure Tunnels

A *tunnel* (see Figure 1-5) is a point-to-point connection created between nodes A and B, where the connection is a path across a network. A and B can treat the tunnel as a direct link between each other. An *encrypted tunnel* is when A and B establish a secure session. An encrypted tunnel is sometimes called a **VPN** (virtual private network). We think that's a really bad term, and a more accurate term would be VPL (virtual private link). Whoever decided to call it a VPN imagined that the encrypted tunnel becomes an extra link in your private network, so a network consisting of a combination of private links and encrypted tunnels across the Internet becomes your private network. If we just use the acronym VPN for the encrypted tunnel (like the industry seems to) and don't think about what VPN expands to, I₂ guess we can live with the term.

Suppose the only reason you've hooked into the Internet is to connect disconnected pieces of your own network to each other. Instead of the configuration in Figure 1-5, you could have bought dedicated links between G₁, G₂, and G₃, and trusted those links as part of your corporate network because you owned them. But it's likely to be cheaper to have the Gs pass data across the Internet. How can you trust your corporate data crossing over the Internet? You do this by configuring G₁, G₂, and G₃ with security information about each other (such as cryptographic keys), and creating secure tunnels between them.

The mechanics of the tunnel between G₁ and G₂, from the IP (network layer 3) point of view, is that when A sends a packet to C, A will launch it with an IP header that has source=A, destination=C. When G₁ sends it across the tunnel, it puts it into another envelope, *i.e.*, it adds an

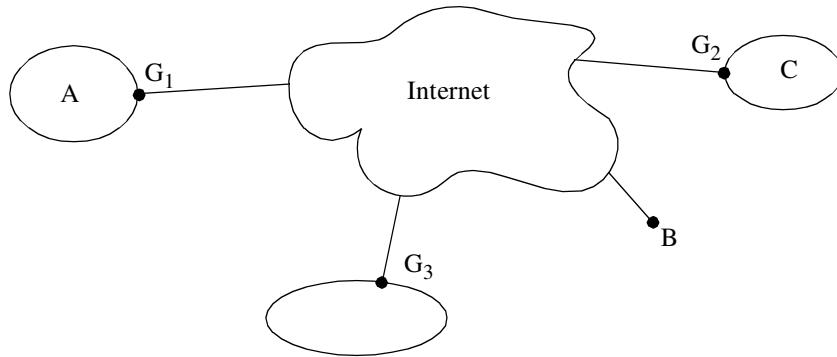


Figure 1-5. Connecting a Private Network over a Public Internet

additional IP header, treating the inner header as data. The outer IP header will contain source= G_1 and destination= G_2 . And all the contents (including the inner IP header) will be encrypted and integrity protected, so it is safe to traverse the Internet.

1.12.4 Why Firewalls Don't Work

Firewalls alone (without also doing end-to-end security) assume that all the bad guys are on the outside, and everyone inside can be completely trusted. This is, of course, an unwarranted assumption. Should employees have access (read and write) to the salary database, for instance?

Even if the company is so careful about hiring that no employee would ever do anything bad intentionally, firewalls can be defeated if an attacker can inject malicious code into a machine on the corporate network. This can be done by tricking someone into downloading something from the Internet or launching an executable from an email message. It is quite common for an attacker to break into one system inside your firewall, and then use that system as a platform for attacking other systems. Someone once described firewall-protected networks as “hard and crunchy on the outside; soft and chewy on the inside.”

Firewalls often make it difficult for legitimate users to get their work done. The firewall might be configured incorrectly or might not recognize a new legitimate application. And if the firewall allows one application through (say email or http), people figure out how to do what they need to do by disguising it as traffic that the firewall is configured to allow. The ironic term for disguising traffic in order to fool a firewall is to carry your traffic in a **firewall-friendly** protocol. Since firewalls commonly allow http traffic (since it's the protocol used for browsing the web), there are many proposals for doing things over http. The most extreme example is to carry IP over http, which would allow any traffic through! Firewall-friendly? The whole point is to defeat the best efforts of the firewall administrator to disallow what you are doing! It isn't somehow “easier” for the firewall

to carry http traffic than any other. The easiest thing for the firewall would be to allow everything or nothing through!

Just as breaking a large file into lots of pieces to be individually carried in separate emails is inefficient, having protocols run on top of *http* rather than simply on top of IP is also inefficient in terms of bandwidth and computation.

1.13 DENIAL-OF-SERVICE (DoS) ATTACKS

A **denial-of-service attack (DoS)** is one in which an attacker prevents good guys from accessing a service, but does not enable unauthorized access to any services. In the naive old days, security people dismissed the prospect of denial-of-service attacks as unlikely, since the attacker had nothing to gain. Of course, that turned out to be faulty reasoning. There are terrorists, disgruntled employees, and people who delight in causing mischief for no good reason.

In the earliest types of denial-of-service attacks, the attacker repeatedly sent messages to the victim machine. Most machines at the time were vulnerable to this sort of attack since they had resources that could easily be depleted. For instance, the storage area for keeping track of pending TCP connections tended to be very limited, on the order of, say, ten connections. The probability of ten legitimate users connecting during a single network round trip time was sufficiently small that ten was a reasonable number. But it was easy for the attacking machine to fill up this table on a server, even if the attacking machine was attached to the Internet with a low-speed link.

To avoid being caught at this mischief, it was common for the attacker to send these malicious packets from forged source addresses. This made it difficult to find (and prosecute) the attacker, and it made it difficult to recognize packets from the malicious machine and filter them at a firewall.

As a defense, people advocated having routers have the capability of doing sanity checks on the source address. These routers could be configured to drop packets with a source address that could not have legitimately come from the direction from which the packet was received. Routers might be configured with which source addresses to expect on each of their ports, or they might infer the expected direction from their forwarding tables. This concept was not deployed because it would cause problems. If sanity checks are based on configured information, topological changes in the Internet (such as links going down and alternative routes being used) could cause the routers to make incorrect assumptions. And Mobile IP (RFC 5944) allows a node to move around in the Internet and keep its IP address, which would confuse routers attempting sanity checks.

A deployed defense against a single malicious node attempting to swamp the resources of a server was to increase resources at the server so that a single attacker, at the speeds at which such attackers were typically connected to the Internet, could not fill the pending TCP connection table.

Another level of DoS escalation was to send a single packet that caused a lot of legitimate machines to send messages to the victim machine. An example of such a packet is a packet transmitted to the broadcast address, with the packet's source address forged to the address of the victim's machine, asking for all receivers to respond. All the machines that receive the broadcast will send a response to the victim's machine. Such a mechanism magnifies the effect the attacker can have from his single machine, since each packet he creates turns into n packets directed at the victim machine.

As a defense against machines sending packets from forged IP addresses, protocols such as TCP, IPsec, and TLS have been designed to avoid requiring a receiver, Bob, to keep state or do significant computation if requests are arriving from forged IP source addresses. Unless the requester can receive packets at the IP address they claim to be coming from, Bob will not need to keep state about the request. Only when the requester returns something that Bob sent to its claimed IP address will Bob pay attention to this request.

But then came the next level of escalation, which is known as a **distributed-denial-of-service attack (DDoS)**. In this form of attack, the attacker breaks into a lot of innocent machines, and installs software on them to have them all attack the victim machine. These innocent machines are called **zombies** or **drones** or **bots**. With enough bots attacking it, any machine can be made inaccessible, since even if the machine itself can process packets as fast as they can possibly arrive, the links or routers in front of that machine can be overwhelmed. The defenses in TCP, IPsec, and TLS will not help, since the bot machines are using their own IP addresses in the requests. Since requests are coming from hundreds or thousands of innocent machines, it is hard to distinguish these packets from packets coming from legitimate users.

1.14 NAT (NETWORK ADDRESS TRANSLATION)

NAT was designed, out of necessity, because IPv4 addresses were too small (four bytes) to give unique addresses to every node on the Internet. With NAT, a piece of the Internet (say a corporate network) can use IP addresses that are not globally unique, and, in fact, these addresses are reused in many other networks. This means that a node inside such a network cannot be contacted from outside that network. However, if the security gateway provides NAT functionality, it will have a pool of globally unique IP addresses that can be assigned as needed.

The NAT box will almost certainly not have a large enough pool of IP addresses to give a globally reachable IP address to every internal node communicating to outside nodes. So the NAT box also translates the TCP or UDP port as well. So a NAT implementation might have a mapping of \langle internal IP, internal port \rangle maps to \langle external IP, external port \rangle . When internal node Alice sends a packet to external destination Bob, the NAT box will replace the source IP and port to the external

IP and port in the NAT box's table. Likewise, when packets arrive from Bob for Alice's assigned external $\langle \text{IP}, \text{port} \rangle$, the NAT box replaces these fields in the destination fields in Bob's packet before forwarding the packet on the internal network.

There is somewhat of a security problem with this approach if the NAT box simply translates tuples of $\langle \text{IP address}, \text{port} \rangle$. Suppose Alice starts a connection to Bob, and the NAT box then creates an entry to translate Alice's internal address and port to, for instance, globally reachable tuple $\langle \text{IP}_{\text{Alice}}, \text{Port}_{\text{Alice}} \rangle$. If the NAT box will forward anything to Alice that is addressed to Alice's temporarily assigned global address and port, then any node on the Internet could send a packet to Alice by addressing it to $\langle \text{IP}_{\text{Alice}}, \text{Port}_{\text{Alice}} \rangle$. Sometimes this is the desired behavior. For example, assume there is a conferencing system. Alice, George, and Carol (all behind NATs) join the conference by contacting the central server, but it is not desirable for all of the conference communication to go through the central server. If the NAT box allows anyone that knows Alice's temporary global address to contact Alice, then the conference coordinator can tell all of the members the other members' global addresses, and they can then directly communicate with each other.

To create the behavior that only the node that Alice has initiated a connection to, to be able to reach Alice, then the NAT box will keep a mapping of 4-tuples to $\langle \text{external IP}, \text{port} \rangle$ pairs. For example, if Alice, at internal $\text{IP}=a$, internal $\text{port}=p$ initiates a connection to external Bob, at $\text{IP address}=B$, $\text{port}=P_B$, the NAT box might assign Alice, for this connection, the external address and port $\langle \text{IP}_A, \text{Port}_A \rangle$. The NAT table would include Bob's address in the mapping, and only allow packets from Bob's address and port to be forwarded to Alice. So the NAT entry would have a six-tuple $\langle B, P_B, \text{IP}_A, \text{Port}_A, a, p \rangle$, meaning that only packets from Bob (at $\langle B, P_B \rangle$) would be translated and forwarded to Alice.

Another fortuitous use of NATs is for all the devices inside a home to have the same IP address. This is useful because some ISPs (Internet Service Providers) charged for Internet connectivity per device, and the NAT box made it appear to the ISP as if there were only a single device in the house. The ISP's answer to this threat to their pricing model was to include in the 74-page EULA (end user license agreement) that everyone has to click on (but nobody reads), an agreement by the user not to use a NAT box. Now, if an actual human read the 74-page agreement, they would most likely think "What's a NAT box?"

1.14.1 Summary

These are the main concepts in the Internet. We will give more details about these as they come up in the book.

The Internet has evolved from the original design, and the design was never the only or best way to design a network, but the industry has made it work. An analogy is the English language. It might be overly complicated, with all the spelling and grammar exceptions, but it does the job. And

each year some mysterious panel of people decide which new words should officially be added to English, and how the grammar rules should change.

Similarly, for the Internet. If there is anything it can't do, at least so far, the world has figured out how to evolve the Internet to do what is needed.

2

INTRODUCTION TO CRYPTOGRAPHY

2.1 INTRODUCTION

The word *cryptography* comes from the Greek words κρυπτό (*hidden or secret*) and γράφη (*writing*). So, cryptography is the art of secret writing. More generally, people think of cryptography as the art of mangling information into apparent unintelligibility in a manner allowing a secret method of unmangling. The basic service provided by cryptography is the ability to send information between participants in a way that prevents others from reading it. In this book, we will concentrate on the kind of cryptography that is based on representing information as numbers and mathematically manipulating those numbers. This kind of cryptography can provide other services, such as

- integrity checking—reassuring the recipient of a message that the message has not been altered since it was generated by a legitimate source,
- authentication—verifying someone’s (or something’s) identity.

There are three basic kinds of cryptographic functions: hash functions, secret key functions, and public key functions. In this chapter, we’ll introduce what each kind is and what it’s useful for. In later chapters we will go into more detail. Public key cryptography (see §2.3 *Public Key Cryptography*) involves the use of two keys. Secret key cryptography (see §2.2 *Secret Key Cryptography*) involves the use of one key. Hash functions (see §2.4 *Hash Algorithms*) involve the use of zero keys! The hash algorithms are not secret, don’t involve keys, and yet they are vital to cryptographic systems.

But back to the traditional use of cryptography. A message in its original form is known as **plaintext** or **cleartext**. The mangled information is known as **ciphertext**. The process for producing ciphertext from plaintext is known as **encryption**. The reverse of encryption is called **decryption**.



While cryptographers invent clever secret codes, cryptanalysts attempt to break these codes. These two disciplines constantly try to keep ahead of each other.

2.1.1 The Fundamental Tenet of Cryptography

Ultimately, the security of cryptography depends on what we call the

Fundamental Tenet of Cryptography

*If lots of smart people have failed to solve a problem,
then it probably won't be solved (soon).*

2.1.2 Keys

Cryptographic systems tend to involve both an algorithm and secret information. The secret is known as the **key**. The reason for having a key in addition to an algorithm is that it is difficult to keep devising new algorithms that will allow reversible scrambling of information, and it is difficult to quickly explain a newly devised algorithm to the person with whom you'd like to start communicating securely. With a good cryptographic scheme, it is perfectly okay to have everyone, including the bad guys (and the cryptanalysts), know the algorithm, because knowledge of the algorithm without the key does not help unmangle the information.

The concept of a key is analogous to the combination of a combination lock. Although the concept of a combination lock is well known (you dial in the secret numbers in the correct sequence and the lock opens), you can't open a combination lock easily without knowing the combination.

2.1.3 Computational Difficulty

It is important for cryptographic algorithms to be reasonably efficient for the good guys to compute. The good guys are the ones with knowledge of the keys.* Cryptographic algorithms are not impossible to break without the key. A bad guy can simply try all possible keys until one works (this assumes the bad guy will be able to recognize plausible plaintext). The security of a cryptographic scheme depends on how much work it is for the bad guy to break it. If the best possible scheme will take ten million years to break using all the computers in the world, then it can be considered reasonably secure.

*We're using the terms *good guys* for the cryptographers and *bad guys* for the cryptanalysts. This is a convenient shorthand and not a moral judgment—in any given situation, which side you consider *good* or *bad* depends on your point of view.

Going back to the combination lock example, a typical combination might consist of three numbers, each a number between 1 and 40. Let's say it takes ten seconds to dial in a combination. That's reasonably convenient for the good guy. How much work is it for the bad guy? There are 40^3 possible combinations, which is 64000. At ten seconds per try, it would take a week to try all combinations, though on average it would only take half that long (even though the right number is always the last one you try!).

Often a scheme can be made more secure by making the key longer. In the combination lock analogy, making the key longer would consist of requiring four numbers to be dialed in. This would make a little more work for the good guy. It might now take thirteen seconds to dial in the combination. But the bad guy has forty times as many combinations to try, at thirteen seconds each, so it would take a year to try all combinations. And if it took that long, he might want to stop to eat or sleep.

With cryptography, computers can be used to exhaustively try keys. Computers are a lot faster than people, and they don't get tired, so millions of keys can be tried per second in software on a single computer, and dramatically more with specialized hardware. Also, lots of keys can be tried in parallel if you have multiple computers, so time can be saved by spending money on more computers.

The cryptographic strength of a cryptographic algorithm is often referred to as the **work factor** required to break it, defined as the number of operations required on classical computers. People are usually vague about what an “operation” is. An ideal encryption algorithm with n -bit keys should require a work factor of 2^n to break it. It can't get better than that, because a brute-force attack (where an attacker tries all possible keys) would be 2^n . Another way of expressing work factor is **security strength**, usually expressed as the size of key in an ideal secret key encryption algorithm (one in which there is no better attack than brute force).

Sometimes a cryptographic algorithm has a variable-length key. It can be made more secure by increasing the length of the key. Increasing the length of the key by one bit makes the good guy's job just a little bit harder but makes the bad guy's job up to twice as hard (because the number of possible keys doubles). Some cryptographic algorithms have a fixed-length key, but a similar algorithm with a longer key can be devised if necessary.

Assume a perfect algorithm, meaning that the work for the good guys is proportional to the length of the key (say n bits), and the work for the bad guys is exponential in the length of the key (*e.g.*, work factor 2^n). Let's say you were using 128-bit keys. If computers got twice as fast, the good guys can use a 256-bit key and have the same performance as before, but the bad guys now have 2^{128} times as many keys to try, and the fact that their computer is also twice as fast as before doesn't help them much. So faster computers mean that the computation gap between good guys and bad guys can be made much bigger, while maintaining the same performance for the good guys.

Keep in mind that breaking the cryptographic scheme is often only one way of getting what you want. For instance, a bolt cutter works no matter how many digits are in the combination. For another example, see xkcd.com/538.

You can get further with a kind word and a gun than you can with a kind word alone.

—Willy Sutton, bank robber

2.1.4 To Publish or Not to Publish

It might seem as though keeping a cryptographic algorithm secret will enhance its security. Cryptanalysts would not only need to break the algorithm, but first figure out what the algorithm is.

These days the consensus is that publishing the algorithm and having it get as much scrutiny as possible makes it more secure. Bad guys will probably find out what the algorithm is eventually (through reverse engineering whatever implementation is distributed), so it's better to tell a lot of nonmalicious people about the algorithm so that in case there are weaknesses, a good guy will discover them rather than a bad guy. A good guy who discovers a weakness will want to gain fame by letting the world know they have found a weakness (after first warning the developers of the system so they can fix the problem). Making your cryptographic algorithm publicly known provides free consulting from the academic community as cryptanalysts look for weaknesses so they can publish papers about them. A bad guy who discovers a weakness will exploit it for doing bad-guy things like embezzling money or stealing trade secrets.

2.1.5 Earliest Encryption

We use the terms *secret code* and *cipher* interchangeably to mean any method of encrypting data. The earliest documented cipher is attributed to Julius Caesar. The way the **Caesar cipher** would work if the message were in English is as follows. Substitute for each letter of the message the letter that is 3 letters later in the alphabet (and wrap around to A from Z). Thus an A would become a D, and so forth. For instance, DOZEN would become GRCHQ. Once you figure out what's going on, it's very easy to read messages encrypted this way (unless, of course, the original message was in Latin).

A slight enhancement to the Caesar cipher was distributed as a premium with Ovaltine in the 1940s as *Captain Midnight secret decoder rings*. (There were times when this might have been a violation of export controls for distributing cryptographic hardware!) The variant is to pick a secret number n between 1 and 25, instead of always using 3. Substitute for each letter of the message the letter that is n later (and wrap around to A from Z, of course). Thus, if the secret number was 1, an

A would become a B, and so forth. For instance, HAL would become IBM. If the secret number was 25, then IBM would become HAL. Regardless of the value of n , since there are only 26 possible values of n to try, it is still very easy to break this cipher if you know it's being used and you can recognize a message once it's decrypted.

The next type of cryptographic system developed is known as a **monoalphabetic cipher**, which consists of an arbitrary mapping of one letter to another letter. There are $26!$ possible pairings of letters, which is approximately 4×10^{26} . [Remember, $n!$, which reads “ n factorial”, means $n \times (n-1) \times (n-2) \times \dots \times 1$.] This might seem secure, because to try all possibilities, if it took a microsecond to try each one, would take about ten trillion years. However, by statistical analysis of language (knowing that certain letters and letter combinations are more common than others), it turns out to be fairly easy to break. For instance, many daily newspapers have a daily cryptogram, which is a monoalphabetic cipher, and can be broken by people who enjoy that sort of thing during their subway ride to work. An example is

Cf lqr'xs xsnyctm n eqxxqgsy iqul qf wdcp eqqh, erl lqrx qgt iqul!

Computers have made much more complex cryptographic schemes both necessary and possible. Necessary because computers can try keys at a rate that would exhaust an army of clerks, and possible because computers can execute the complex algorithms quickly and without errors.

2.1.6 One-Time Pad (OTP)

There is an easy-to-understand, fast-to-execute, and perfectly secure encryption scheme called a **one-time pad**. While it is rarely practical to implement, other more practical but less secure schemes are often compared to it. A one-time pad consists of a long stream of random bits known to the communicating parties but not to anyone else. We'll use the symbol \oplus for the operation XOR (bitwise exclusive or). Encryption consists of \oplus ing the plaintext with the random bits of a one-time pad, and decryption consists of \oplus ing the ciphertext with those same random bits.

An eavesdropper who only sees the ciphertext sees something that is indistinguishable from random numbers. One can think of the one-time pad as being the secret key, and while an eavesdropper could go through all possible keys to learn the message, they would also see all other possible messages of the same length and have no way to tell which one was actually sent. Because no amount of computation would allow an eavesdropper to do better than guessing, the scheme is called **information-theoretically secure**.

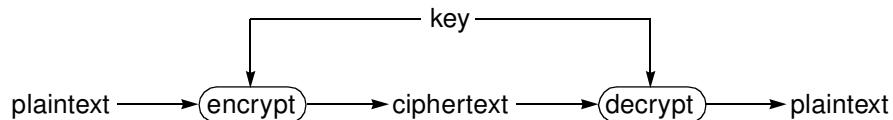
One-time pads are generally impractical because they require the two parties to agree in advance on a random bitstream that is as long as all the messages they will ever want to send to one another. One-time pads are insecure if the same random numbers are used to encrypt two different messages. That's because an eavesdropper who knows the communicating parties are doing that can learn the \oplus of the two messages. This might not seem like a big deal, but in practice it can leak

a lot of information. For example, assume two images encrypted by \oplus ing with the same one-time pad. If the two encrypted images are \oplus 'd, you'll get the \oplus of the two images. Even a human can often see what both images are if the result is displayed. The more times a one-time pad is reused, the easier it becomes to determine what the one-time pad is. This is why cryptographers call this method a *one-time pad*, to remind people not to use it more than once (see Homework Problem 11).

Because it tends to be impractical for Alice and Bob to agree upon a truly random one-time pad with as many bits as the total amount of information they might want to send to each other, some cryptographic schemes use a fixed-length secret number as a **seed** to generate a pseudorandom bitstream. The pseudorandom stream generated from the seed is \oplus 'd with plaintext as a means of encrypting the plaintext (and the same stream is \oplus 'd by the receiver to decrypt). This sort of scheme, usually referred to as a **stream cipher**, looks like a one-time pad scheme. However, if the stream is generated from a seed, it doesn't have information-theoretic security because an attacker could go through all possible seeds to find the one that results in an intelligible message. Some stream ciphers we discuss in the book include RC4 (§3.8), and CTR mode (§4.2.3).

2.2 SECRET KEY CRYPTOGRAPHY

Secret key cryptography involves the use of a single key. Given a message (the *plaintext*) and the key, encryption produces unintelligible data (called an IRS Publication—no! no! that was just a finger slip, we meant to say *ciphertext*), which is about the same length as the plaintext was. Decryption is the reverse of encryption, and uses the same key as encryption.



Secret key cryptography is sometimes referred to as **conventional cryptography** or **symmetric cryptography**. The Captain Midnight code and the monoalphabetic cipher are both examples of secret key algorithms, though both are easy to break. In this chapter we describe the functionality of cryptographic algorithms, but not the details of particular algorithms. In Chapter 3 we describe the details of some popular secret key cryptographic algorithms. The next few sections describe the types of things one might do with secret key cryptography.

2.2.1 Transmitting Over an Insecure Channel

It is often impossible to prevent eavesdropping when transmitting information. For instance, a telephone conversation can be tapped, a letter can be intercepted, and a message transmitted on the Internet can be observed by routers along the path.

If you and I agree on a shared secret (a key), then by using secret key cryptography we can send messages to one another on a medium that can be tapped, without worrying about eavesdroppers. All we need to do is have the sender encrypt the messages and the receiver decrypt them using the shared secret. An eavesdropper will only see unintelligible data. This is the classic use of secret key cryptography.

2.2.2 Secure Storage on Insecure Media

Since it is difficult to assume any stored data can be kept from prying eyes, it is best to store it encrypted. This must be done carefully, because forgetting the key makes the data irretrievable.

2.2.3 Authentication

In spy movies, when two agents who don't know each other must rendezvous, they are each given a password or pass phrase that they can use to recognize one another. For example, Alice's secret phrase might be "The moon is bright tonight." Bob's response might be "Not as bright as the sun." If Alice were not talking to the real Bob, she will have divulged the secret phrase to the imposter. Even if she is talking to Bob, she may have also divulged the secret phrase to an eavesdropper.

The term **strong authentication** means that someone can prove knowledge of a secret without revealing it. Strong authentication is possible with cryptography. Strong authentication is particularly useful when two computers are trying to communicate over an insecure network (since few people can execute cryptographic algorithms in their heads). Suppose Alice wants to make sure she is talking to Bob, and they share a key K_{AB} . Alice picks a random number r_A , encrypts it with K_{AB} , and sends that to Bob. The quantity $\{r_A\}K_{AB}$ is known as a **challenge**. Bob decrypts the challenge and sends r_A to Alice. This is known as Bob's **response** to the challenge $\{r_A\}K_{AB}$. Alice knows that she is speaking to someone who knows K_{AB} because the response matches r_A . See Figure 2-1.

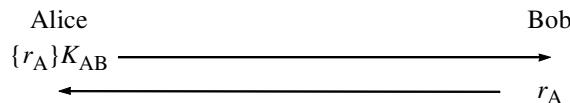


Figure 2-1. Challenge–Response Authentication with Shared Secret

If someone, say, Fred, were impersonating Alice, he could get Bob to decrypt a value for him (though Fred wouldn't be able to tell if the person he was talking to was *really* Bob because Fred doesn't know K_{AB}), but Bob's response to Fred's challenge would not be useful later in impersonating Bob to the real Alice because the real Alice would pick a different challenge.

Note that in this particular protocol, there is the opportunity for Fred to obtain some $\langle \text{chosen ciphertext}, \text{plaintext} \rangle$ pairs, since he can claim to be Alice and ask Bob to decrypt a challenge for him. For this reason, it is essential that challenges be chosen from a large enough space, say 2^{64} values, so that there is no significant chance of using the same challenge twice. We discuss all sorts of authentication tricks and pitfalls in Chapter 11 *Communication Session Establishment*.

2.2.4 Integrity Check

Another use of secret key cryptography is to generate a fixed-length cryptographic checksum associated with a message.

What is a **checksum**? An ordinary (noncryptographic) checksum protects against accidental corruption of a message. The original derivation of the term *checksum* comes from the operation of breaking a message into fixed-length blocks (for instance, 32-bit words) and adding them up. The sum is sent along with the message. The receiver similarly breaks up the message, repeats the addition, and *checks the sum*. If the message had been garbled en route, the sum will hopefully not match the sum sent and the message will be rejected. Unfortunately, if there were two or more errors in the transmission that canceled each other, the error would not be detected. It turns out this is not terribly unlikely, given that if flaky hardware flips a bit somewhere, it is likely to flip another somewhere else. To protect against such “regular” flaws in hardware, more complex checksums called cyclic redundancy checks (CRCs) were devised. But these still only protect against faulty hardware and not an intelligent attacker. Since CRC algorithms are published, an attacker who wanted to change a message could do so, compute the CRC on the new message, and send that along.

To provide protection against malicious changes to a message, a *secret* integrity check algorithm is required, *i.e.*, an attacker that does not know the algorithm should not be able to compute the correct integrity check for a message. As with encryption algorithms, it's better to have a common (known) algorithm and a secret key. This is what a cryptographic checksum does. Given a key and a message, the algorithm produces a fixed-length **message authentication code (MAC)** that can be sent with the message. MACs used to be called **MICs (message integrity codes)** in some older standards, but the term *MAC* seems to have become more popular.

If anyone were to modify the message without knowing the key, they would have to guess a MAC, and the chance of picking a correct MAC depends on the length of the MAC. A typical MAC is at least 48 bits long, so the chance of getting away with a forged message is only one in 280 trillion.

Such message authentication codes have been in use to protect the integrity of large interbank electronic funds transfers for quite some time. The messages are not kept secret from an eavesdropper, but the integrity check assures that only someone with knowledge of the key could have created or modified the message.

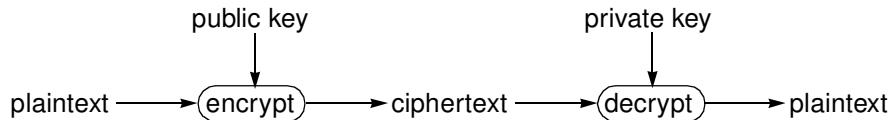
2.3 PUBLIC KEY CRYPTOGRAPHY

Public key cryptography is sometimes also referred to as **asymmetric cryptography**. The first publication to present the concept of public key cryptography was in 1975 [DIFF76b], though it is now known that scientists from GCHQ, the British equivalent to the U.S. NSA, had discovered this technology a few years earlier. Clifford Cocks is now known to have invented the RSA technology in 1973, and Malcom J. Williamson invented the Diffie-Hellman technology in 1974.

Unlike secret key cryptography, each individual has two keys: a private key that should only be known to the key owner, and a public key that can safely be told to the entire world. The keys are related to each other, in that Alice's public key could be used so that anyone with knowledge of her public key can encrypt something for Alice, and only Alice, knowing her private key, will be able to decrypt it. There are other uses as we will describe.

Note that we call the private key a *private key* and not a *secret key*. This convention is an attempt to make it clear in any context whether public key cryptography or secret key cryptography is being used. Some people use the term *secret key* for the private key in public key cryptography or use the term *private key* for the secret key in secret key technology. We hope to convince people to use the term *secret key* only as the single secret number used in secret key cryptography. The term *private key* should refer to the key in public key cryptography that must not be made public.

Unfortunately, both words *public* and *private* begin with *p*. We will sometimes want a single letter to refer to one of the keys. The letter *p* won't do. We will use the letter *e* to refer to the public key, since the public key is used when encrypting a message. We'll use the letter *d* to refer to the private key, because the private key is used to decrypt a message.



There is an additional thing one can do with public key technology, which is to generate a digital signature on a message. A **digital signature** is a number associated with a message, like a checksum or the MAC described in §2.2.4 *Integrity Check*. However, unlike a checksum, which can be generated by anyone, a digital signature can only be generated by someone knowing the private



key. A public key signature differs from a secret key MAC because verification of a MAC requires knowledge of the same secret as was used to create it. Therefore, anyone who can verify a MAC can also generate one, and so be able to substitute a different message and corresponding MAC.

In contrast, verification of the signature only requires knowledge of the public key. So Alice can sign a message by generating a signature that only she can generate (using her private key). Other people can verify that it is Alice's signature (because they know her public key) but cannot forge her signature. This is called a *signature* because it shares with handwritten signatures the property that it is possible to recognize a signature as authentic without being able to forge it.

Public key cryptography can do anything secret key cryptography can do, but to give the same security level, the known public key cryptographic algorithms are orders of magnitude slower than the best known secret key cryptographic algorithms. So public key algorithms are usually used in combination with secret key algorithms. Public key cryptography is very useful because network security based on public key technology tends to be more easily configurable. Public key cryptography might be used in the beginning of communication for authentication and to establish a temporary shared secret key, then the secret key is used to encrypt the remainder of the conversation using secret key technology.

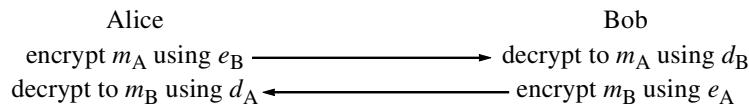
For instance, suppose Alice wants to talk to Bob. A typical technique is that Alice uses Bob's public key to encrypt a secret key, then uses that secret key to encrypt whatever else she wants to send to Bob. Since the secret key is much smaller than the message, using the slow public key cryptography to encrypt the secret key is not that much of a performance hit. Notice that given this protocol, Bob does not know that it was Alice who sent the message. This could be fixed by having Alice digitally sign the encrypted secret key using her private key.

Now we'll describe the types of things one might do with public key cryptography.

2.3.1 Transmitting Over an Insecure Channel

Suppose Alice's $\langle \text{public key}, \text{private key} \rangle$ pair is $\langle e_A, d_A \rangle$. Suppose Bob's key pair is $\langle e_B, d_B \rangle$. Assume Alice knows Bob's public key, and Bob knows Alice's public key, and they each know their own private key. Actually, accurately learning other people's public keys is one of the biggest

challenges in using public key cryptography and will be discussed in detail in §10.4 *PKI*. But for now, don't worry about it.



2.3.2 Secure Storage on Insecure Media

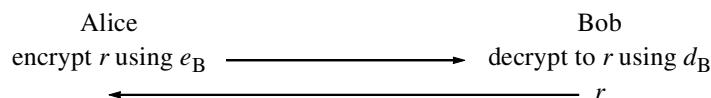
For performance reasons, you would encrypt the data with a secret key, but then you can encrypt the secret key with the public key of whoever is authorized to read the data, and include that with the encrypted data. An advantage of public key cryptography is that Alice can encrypt something for Bob without knowing his decryption key. If there are multiple authorized readers, Alice can encrypt the secret key with each of the authorized readers, and include those encrypted quantities as well, with the encrypted data.

2.3.3 Authentication

With secret key cryptography, if Alice and Bob want to communicate, they have to share a secret. If Bob wants to be able to prove his identity to lots of entities, then with secret key technology he will need to remember lots of secret keys, one for each entity to which he would like to prove his identity. Possibly he could use the same shared secret with Alice as with Carol, but that has the disadvantage that then Carol and Alice could impersonate Bob to each other.

Public key technology is much more convenient. Bob only needs to remember a single secret, his own private key. It is true that if Bob wants to be able to verify the identity of thousands of entities, then he will need to know (or be able to obtain when necessary) thousands of public keys. In §10.4 *PKI* we discuss how this might be done.

Here's an example of how Alice can use public key cryptography for verifying Bob's identity (assuming Alice knows Bob's public key). Alice chooses a random number r , encrypts it using Bob's public key e_B , and sends the result to Bob. Bob proves he knows d_B by decrypting the message and sending r back to Alice.



2.3.4 Digital Signatures

Forged in USA

engraved on a screwdriver claiming to be of brand *Craftsman*™

It is often useful to prove that a message was generated by a particular individual. This is easy with public key technology. Bob's signature for a message m can only be generated by someone with knowledge of Bob's private key. And the signature depends on the contents of m . If m is modified in any way, the signature no longer matches. So digital signatures provide two important functions. They prove who generated the information (in this case, Bob), and they prove that the information has not been modified (by anyone other than Bob) since the message and matching signature were generated.

Digital signatures offer an important advantage over secret key-based cryptographic MACs—**non-repudiation**. Suppose Bob sells widgets and Alice routinely buys them. Alice and Bob might agree that, rather than placing orders through the mail with signed purchase orders, Alice will send electronic mail messages to order widgets. To protect against someone forging orders and causing Bob to manufacture more widgets than Alice actually needs, Alice will include an integrity check on her messages. This could be either a secret key-based MAC or a public key-based signature. But suppose sometime after Alice places a big order, she changes her mind (the bottom fell out of the widget market). Since there's a big penalty for canceling an order, she doesn't confess that she's canceling, but instead denies that she ever placed the order. Bob sues. If Alice authenticated the message by computing a MAC based on a key she shares with Bob, then Bob knows Alice did place the order. That is because nobody other than Bob and Alice knows that key, so if Bob knows he didn't create the message, he knows it must have been Alice. But he can't prove it to anyone! Since he knows the same secret key that Alice used to sign the order, he could have forged the signature on the message himself, and he can't prove to the judge that he didn't! If it were a public key signature, he could show the signed message to the judge and the judge could verify that it was signed with Alice's key. Alice could still claim, of course, that someone had stolen and misused her key (it might even be true!), but the contract between Alice and Bob could reasonably hold her responsible for damages caused by her inadequately protecting her key. Unlike secret key cryptography, where the keys are shared, you can always tell who's responsible for a signature generated with a private key.

2.4 HASH ALGORITHMS

Hash algorithms are also known as **message digest algorithms**. A cryptographic hash function is a mathematical transformation that inputs a message of arbitrary length (transformed into a string of bits) and outputs a fixed-length (short) number. We'll call the hash of a message m , $\text{hash}(m)$. It has the following properties:

- For any message m , it is relatively easy to compute $\text{hash}(m)$. This just means that in order to be practical it can't take a lot of processing time to compute the hash.
- Given a hash value h , it is computationally infeasible to find an m that hashes to h .
- Even though many different values of m will hash to the same value (because m is of arbitrary length, and $\text{hash}(m)$ is fixed length), it is computationally infeasible to find two messages that hash to the same thing.

We'll give examples of secure hash functions in Chapter 5 *Cryptographic Hashes*.

2.4.1 Password Hashing

When a user types a password, the system has to be able to determine whether the user got it right. If the system stores the passwords unencrypted, then anyone with access to the system storage or backup tapes can steal the passwords. Luckily, it is not necessary for the system to know a password in order to verify its correctness. Instead of storing the password, the system can store a hash of the password. When a password is supplied, the system computes the password's hash and compares it with the stored value. If they match, the password is deemed correct. If the hashed password file is obtained by an attacker, it is not immediately useful because the passwords can't be derived from the hashes.

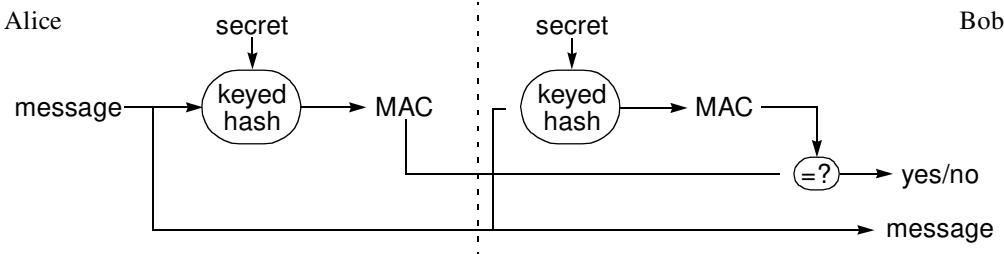
Historically, some systems made the password file publicly readable, an expression of confidence in the security of the hash. Even if there are no cryptographic flaws in the hash, it is possible to guess passwords and hash them to see if they match. If a user is careless and chooses a password that is guessable (say, a word that would appear in a 50000-word dictionary of potential passwords), an exhaustive search would "crack" the password even if the encryption were sound. For this reason, many systems hide the hashed password list (and those that don't, should).

2.4.2 Message Integrity

Cryptographic hash functions can be used to generate a MAC to protect the integrity of messages transmitted over insecure media in much the same way as secret key cryptography.

If we merely sent the message and used the hash of the message as a MAC, this would not be secure, since the hash function is well-known. The bad guy can modify the message, compute a new hash for the new message, and transmit that.

However, if Alice and Bob have agreed on a secret, Alice can use a hash to generate a MAC for a message to Bob by taking the message, concatenating the secret, and computing the hash of *message|secret*. This is called a **keyed hash**. Alice then sends the hash and the message (without the secret) to Bob. Bob concatenates the secret to the received message and computes the hash of the result. If that matches the received hash, Bob can have confidence the message was sent by someone knowing the secret. (Note: There are some cryptographic subtleties to making this actually secure. See §5.4.10 *Computing a MAC with a Hash*.)



2.4.3 Message Fingerprint

If you want to know whether some large data structure (*e.g.*, a program) has been modified from one day to the next, you could keep a copy of the data on some tamper-proof backup store and periodically compare it to the active version. With a hash function, you can save storage. You simply save the hash of the data on the tamper-proof backup store (which, because the hash is small, could be a piece of paper in a filing cabinet). If the hash value hasn't changed, you can be confident none of the data has.

A note to would-be users—if it hasn't already occurred to you, it has occurred to the bad guys—the program that computes the hash must also be independently protected for this to be secure. Otherwise, the bad guys can change the file but also change the hashing program to report the hash as though the file were unchanged!

2.4.4 Efficient Digital Signatures

Public key algorithms are much slower than hash algorithms. Therefore, to sign a message, which might be arbitrarily large, Alice does not usually sign the actual message. Instead, she computes a hash of the message and uses her private key to sign the hash of the message.

2.5 BREAKING AN ENCRYPTION SCHEME

What do we mean when we speak of a bad guy Fred *breaking* an encryption scheme? Examples are Fred being able to decrypt something without knowing the key, or figuring out what the key is. Various attacks are classified as **ciphertext only**, **known plaintext**, **chosen plaintext**, **chosen ciphertext** and **side-channel attacks**.

Note that cryptographers are well aware of all these attacks, so most modern systems are designed to be resilient against these attacks.

2.5.1 Ciphertext Only

In a ciphertext-only attack, Fred has seen (and presumably stored) some ciphertext that he can analyze at leisure. Typically, it is not difficult for a bad guy to obtain ciphertext. (If a bad guy can't access the encrypted data, then there would have been no need to encrypt the data in the first place!)

How can Fred figure out the plaintext if all he can see is the ciphertext? One possible strategy is to search through all the keys. Fred tries the decrypt operation with each key in turn. It is essential for this attack that Fred be able to recognize when he has succeeded. For instance, if the message was English text, then it is highly unlikely that a decryption operation with an incorrect key could produce something that looked like intelligible text. Because it is important for Fred to be able to differentiate plaintext from gibberish, this attack might be better called a **recognizable-plaintext** attack.

It is also essential that Fred have enough ciphertext. For instance, using the example of a monoalphabetic cipher, if the only ciphertext available to Fred were XYZ, then there is not enough information. There are many possible letter substitutions that would lead to a legal three-letter English word. There is no way for Fred to know whether the plaintext corresponding to XYZ is THE or CAT or HAT. As a matter of fact, in the following sentence, any of the words could be the plaintext for XYZ:

The hot cat was sad but you may now sit and use her big red pen.

[Don't worry—we've found a lovely sanatorium for the coauthor who wrote that.

—*the other coauthors*]

Often it isn't necessary to search through a lot of keys. For instance, the authentication scheme Kerberos (see §10.3 *Kerberos*) assigns to user Alice a secret key derived from Alice's password according to a straightforward, published algorithm. The secret key Kerberos uses today is typically 256 bits. If Alice chooses her password unwisely (*e.g.*, a word in the dictionary), then Fred does not need to search through all 2^{256} possible keys—instead he only needs to try the derived keys of the 50000 or so passwords in a dictionary of potential passwords.

Another form of information leakage from ciphertext alone is known as **traffic analysis**, in which information is inferred based on seeing transmitted ciphertext. For instance, it might be useful to know that a lot of traffic is suddenly being transmitted between two companies, because it might signal a potential merger. To avoid leaking information based on which parties are communicating, traffic can be sent through an intermediary, which could send lots of dummy traffic to all its customers at all times, and just replace dummy traffic with real traffic when there is real traffic to send. See §14.18 *Message Flow Confidentiality*.

The length of a message might leak information. For instance, when a teenager gets a letter from a college they applied to, they know before opening the envelope (based on whether it is thick or thin), whether the letter is a rejection or an acceptance. To avoid leaking information based on how long a message is, messages can be padded with extra bits so they are all the same size.

A cryptographic algorithm has to be secure against a ciphertext-only attack because of the accessibility of the ciphertext to cryptanalysts. But, in many cases, cryptanalysts can obtain additional information, so it is important to design cryptographic systems to withstand the next two attacks as well.

2.5.2 Known Plaintext

Sometimes life is easier for the attacker. Suppose Fred has somehow obtained some $\langle \text{plaintext}, \text{ciphertext} \rangle$ pairs. How might he have obtained these? One possibility is that secret data might not remain secret forever. For instance, the data might consist of specifying the next city to be attacked. Once the attack occurs, the plaintext to the previous day's ciphertext is now known. Another example is where all messages start with the same plaintext, for instance, the date.

With a monoalphabetic cipher, a small amount of known plaintext would be a bonanza. From it, the attacker would learn the mappings of a substantial fraction of the most common letters (every letter that was used in the plaintext Fred obtained). Some cryptographic schemes might be good enough to be secure against ciphertext-only attacks but not good enough against known plaintext attacks. In these cases, it becomes important to design the systems that use such a cryptographic algorithm to minimize the possibility that a bad guy will ever be able to obtain $\langle \text{plaintext}, \text{ciphertext} \rangle$ pairs.

2.5.3 Chosen Plaintext

Sometimes, life may be easier still for the attacker. In a chosen plaintext attack, Fred can choose any plaintext he wants and get the system to tell him what the corresponding ciphertext is. How could such a thing be possible?

Suppose a telegraph company offered a service in which they encrypt and transmit messages for you. Suppose Fred had eavesdropped on Alice's encrypted message. Now he'd like to break the telegraph company's encryption scheme so that he can decrypt Alice's message.

He can obtain the corresponding ciphertext to any message he chooses by paying the telegraph company to send the message for him, encrypted. For instance, if Fred knew they were using a monoalphabetic cipher, he might send the message

The quick brown fox jumps over the lazy dog.

knowing that he would thereby get all the letters of the alphabet encrypted and then be able to decrypt with certainty any encrypted message. Even if the telegraph company is using a strong cipher, if the same message sent twice encrypts to the same ciphertext, then if the attacker can guess the plaintext, he can verify his guess by sending that message and seeing whether the ciphertext matches. As we will see in Chapter 4 *Modes of Operation*, modern cryptographic systems encrypt in a way that encrypting the same plaintext twice will result in two different ciphertexts.

2.5.4 Chosen Ciphertext

If an attacker (Trudy) makes up a message or modifies a real message from Alice to Bob, and sends it to Bob, claiming to be Alice, Trudy might learn something by observing how Bob reacts. Even if Bob detects that the message is not formatted as a message from Alice should be, Trudy might learn something based on the particular error message Bob sends when rejecting the message, or even based on how long it takes Bob to respond with an error.

One example of a chosen ciphertext attack was a vulnerability in a widely deployed system (SSL), found by Bleichenbacher [BLEI98]. The attack works as follows: First, Trudy eavesdrops and records a session between Alice and Bob. In particular, Trudy will find the part of the session where Alice sent a secret key encrypted with Bob's public key. That secret key was then used to cryptographically protect the rest of the session. Trudy can then pretend to try to initiate a connection to Bob sending modified versions of the encrypted secret key that Alice sent. These connections will fail, because Trudy doesn't know the secret key Alice used, but this attack takes advantage of the fact that in this implementation, Bob was super helpful in letting Trudy know what was specifically wrong with each modified message. With this attack, Trudy would eventually (*e.g.*, after on the order of million connection attempts) discover the session key for the recorded Alice-Bob session, and, therefore, Trudy would be able to decrypt that entire Alice-Bob conversation. These sorts of attacks can generally be protected against by simple precautions, such as always protecting data with both encryption and integrity protection, and by giving the least information possible about why communication attempts fail.

There are more subtle chosen ciphertext attacks and defenses, which we discuss in (§8.1.3 *Defense Against Dishonest Ciphertext*). These generally force the attacker to prove they have

generated their ciphertext in a way that does not deviate from the specification, for example, by including in the ciphertext an encryption of the random seed used to create the ciphertext.

2.5.5 Side-Channel Attacks

A **side-channel attack**, introduced by Paul Kocher [KOCH96], isn't an attack on an algorithm itself, but rather on an implementation of an algorithm in a particular environment. By observing side effects of an implementation while it is executing, an attacker might discover information such as some or all the bits of the plaintext or the key being used. Usually, what's observed is how long it takes something to execute, though some side channels are based on careful measurement of power consumption or even the sound the computer makes. The most powerful attacks are possible when Trudy, the attacker, can run a program on a system very close to the implementation she would like to observe. For example, she might run a process on the same computer. Trudy can measure how long her own operations take while the victim process is running on another thread. By carefully measuring instruction times as it accesses memory, Trudy can get information about which cache lines the victim process is using. Another example where Trudy might powerfully gain information is if she installs malicious software or hardware on a smart card reader. Again, she is close to the implementation running on the smart card and can observe how much power it uses, and other information. Even an attacker with only network access may be able to determine things by carefully measuring how long it takes a server to respond to messages.

One way of defending against side-channel attacks is for an implementation to make sure that it always behaves the same way, no matter what its inputs are. For instance, it could perform extra computation so that timing always matches that of the worst-case input. Another type of mitigation is to randomize the inputs using a random function of the inputs and then convert the result to what it would be if the implementation computed based on the actual input.

2.6 RANDOM NUMBERS

The generation of random numbers is too important to be left to chance.

—Robert Coveyou, Oak Ridge National Laboratory

In this section, we will discuss some of the considerations that go into the generation and use of random numbers. For more reading on the subject of random number generators, see NIST SP800-90 parts A, B, and C.

You can have perfect cryptographic algorithms, and perfectly designed protocols, but if you can't select good random numbers, your system can be remarkably insecure. Random numbers may be required in choosing cryptographic keys, challenges, or other inputs to a cryptographic system.

Although I₃'d love to rigorously define "random", I₂ won't let me₃ because a really rigorous definition is beyond the scope and spirit of the book. For instance, in [KNUT69], fifteen pages are devoted to coming up with a definition of a random sequence. For those readers who are not intimidated by notions such as (m,k) -distributed, serial correlation coefficients, and Riemann-integrability (don't try to find those terms in our glossary), the discussion of randomness in [KNUT69] is actually very interesting (though, honestly, not to me₂).

In the context of cryptography, we measure the quality of random numbers by how much work it would be for an attacker to guess the chosen value. If that's more than the work required to break the cryptographic system in some other way, then the quantity is random enough. Ideally, it would require testing about 2^n strings to find a specific n -bit random string (actually, 2^{n-1} tests on average, assuming average luck on the part of the attacker). Random number generators used in cryptographic systems usually involve three steps:

- Gathering entropy
- Computing a random seed
- Compute a pseudorandom number stream from the seed

Each of these steps is perilous in its own way, in terms of bugs that could make a system insecure in ways that are hard to detect.

2.6.1 Gathering Entropy

The concept of entropy comes from physics, but in this context entropy is a measure of the difficulty of guessing the value of something. Something with 2^n equally likely values has n bits of entropy. If the values are not equally likely, there is less entropy. A perfectly random stream of bits would have one bit of entropy per bit of data, but even data that is easier than that to guess has value. Timing user keystrokes using a high-resolution clock may be fairly predictable but still give a few bits of entropy per keystroke. Measuring the seek time of disks is similarly predictable but not in the low order bits. If the device has a microphone or camera, a large number of bits can be read containing a reasonably high amount of entropy.

One of the problems with mechanical sources is that their quality can degrade over time. User keystroke timing may contain a lot of entropy until the user is replaced with an automated agent that feeds the keystrokes from a script. Disk seek times may become highly predictable when the disk hardware is replaced with a solid-state drive. A conservative design will combine lots of entropy sources in such a way that the system will be secure if any of them is good.

Most modern CPUs have a built-in random number generator from which random bits can be harvested, but it is dangerous to rely on that single source. That's because subverting the design of the random number source on a CPU chip would be a high-value target for intelligence agencies or well-funded criminal organizations. It would be very difficult to detect that the output of the chip's random number function was not based on truly random input, but instead, guessable by someone who knows the seed and algorithm that they secretly embedded in the chip design.

Some systems go to extreme lengths to build hardware that gathers provably good sources of entropy like counting events of radioactive decay with a Geiger counter or measuring the polarization of photons. There is no reason to believe that these sources of randomness are any better than more conventional ones, and like a random number source built into a CPU, it would be difficult to know whether the high-priced device is really counting Geigers :-) or generating numbers that are guessable by the agency that secretly embedded an algorithm.

2.6.2 Generating Random Seeds

The second step is to turn a lot of sources of unguessable quantities into a random seed. The best way to do that is to perform a cryptographic hash of the data. Some people recommend \oplus ing the different sources together, which is just as good if the different sources are uncorrelated. But if two sources turned out to be identical, \oplus ing them together would entirely remove their benefit (see Homework Problem 6), while hashing their concatenation would keep almost all the entropy they contain so long as the hash function output is large enough to hold it.

2.6.3 Calculating a Pseudorandom Stream from the Seed

A **pseudorandom number generator (PRNG)** takes the random seed and produces a long stream of cryptographically random data. Since the PRNG generates a stream deterministically from a seed, the entropy in the stream will not be greater than the size of the seed. Also, if someone were to capture the state of the PRNG, they would be able to calculate all the subsequent output. Sometimes parts of the stream will not be secret, such as when the stream is used for choosing a random initialization vector (IV) [see §4.2.2 *CBC (Cipher Block Chaining)*]. It is essential, then, that seeing part of the pseudorandom stream not allow an attacker to compute the rest of the stream. It is also desirable for an implementation to throw away state so that previous output of the stream will not be able to be calculated based on the current state. For example, if the implementation always kept the initial seed, someone who stole that seed would be able to calculate the entire stream, past and future. If an implementation were to crash, the system might dump the entire state.

There are many secure ways to construct a pseudorandom number generator using secret key encryption and hash functions, but it is best to use one of the standardized ones from [NIST15a].

There are two reasons for that—first, it is required for some security compliance regimes, but just as important is that those algorithms come with sample data that allow you to test whether your implementation is correct. Bugs in random number generators will produce output that almost certainly looks random to any tester but may be trivially attackable by someone who discovers the bug and sees some of the output.

Sometimes, Alice and Bob need to agree on a lot of different keys or other secret values. It is often convenient to derive all of them from a seed so that Alice need only send the seed to Bob and the other secret information can be derived from that seed. The deterministic function for deriving information from a seed is either known as a **pseudorandom function (PRF)** or a **key-derivation function (KDF)**. This function will generally take two inputs: the seed, and an additional input called a **data variable** which identifies which of the possible keys or secret values are being generated from the seed.

2.6.4 Periodic Reseeding

It is good security practice to periodically add randomness to the PRNG. This involves gathering entropy as the program runs, and when there is enough, mixing it in with the state. The reason to do this is so that if an attacker learns the seed you are using, they will only be able to predict the random stream for a limited amount of time.

It is better to wait until a reasonable amount of entropy has been built up, and mix it all in at once, rather than mixing in new entropy bit-by-bit as it is gathered. Adding 128 bits of entropy once is worth a lot more than adding eight bits of entropy a thousand times (see Homework Problem 9).

2.6.5 Types of Random Numbers

Applications that use random numbers have different requirements. For most applications, for instance one that generates test cases for debugging a computer program, all that might be required is that the numbers are spread around with no obvious pattern. For such applications it might be perfectly reasonable to use something like the digits of π . However, for cryptographic applications such as choosing a cryptographic key, it is essential that the numbers be unguessable. Consider the following pseudorandom number generator. It starts with a truly random seed, say by timing a human's keystrokes. Then it computes a hash of the seed, then at each step it computes a hash of the output of the previous step. Assuming a good hash function, the output will pass any sort of statistical tests for randomness, but an intruder that captures one of the intermediate quantities will be able to compute the rest.

In general, the functions provided in programming languages for random numbers are not designed to be unguessable in the cryptographic sense. They are designed merely to pass statistical tests. Calling one of these to generate cryptographic keys is an invitation to disaster.

2.6.6 Noteworthy Mistakes

Random number implementations have made some amusing mistakes. Typical mistakes are:

- Seeding the generator with a seed that is from too small a space. Suppose each time a cryptographic key needed to be chosen, the application obtained sixteen bits of true randomness from special purpose hardware and used those sixteen bits to seed a pseudorandom number generator. The problem is that there would be only 65536 possible keys it would ever choose, and that is a very small space for an adversary (equipped with a computer) to search. Jeff Schiller found an implementation in a product that computed RSA key pairs randomly, but from an 8-bit seed! Jeff attempted to report the bug to one of the lead developers of the product (let's call that person Bob, though that was not his name). Bob did not believe Jeff. So Jeff wrote a program to compute all 256 possible key pairs the product could generate, found Bob's key, and sent Bob an email signed with Bob's private key.
- Using a hash of the current time when an application needs a random value. The problem is that some clocks do not have fine granularity, so an intruder that knew approximately when the program was run would not need to search a very large space to find the exact clock value for the seed. For instance, if a clock has granularity of 1/60 second, and the intruder knew the program chose the user's key somewhere within a particular hour, there would only be $60 \times 60 \times 60 = 216000$ possible values. A widely deployed product used a microsecond granularity concatenated with some other values that were not very secret. Ian Goldberg and David Wagner discovered this bug and demonstrated breaking keys in 25 seconds on a slow machine [GOLD96].
- Divulging the seed value. An implementation, again discovered by Jeff Schiller (who warned the company so that the implementation has been fixed), used the time of day to choose a per-message encryption key. The time of day in this case may have had sufficient granularity, but the problem was that the application included the time of day in the unencrypted header of the message!

One event too important to leave out involves an intelligence agency that may have tricked the world into deploying a pseudorandom number generator with a back door that allowed them to predict the output (assuming they could see parts of the output). It was called Dual_EC_DRBG, and in spite of warnings from industry experts and much lower performance than competing algorithms, they managed to convince NIST to standardize it, and RSA Data Security to make it the default in

their widely used software. It was only after evidence of a back door was revealed in the Edward Snowden papers that the algorithm was discredited and removed from service.

2.7 NUMBERS

Cryptographic algorithms manipulate messages and keys, both of which are defined in terms of bit strings. The algorithms are usually defined in terms of arithmetic operations on numbers, where the numbers come from some interpretation of groups of bits. Mathematics has been studied for thousands of years, and mathematical properties are well understood, making analysis of algorithms based on mathematical principles more reliable.

Mathematics deals with many kinds of numbers: integers, rational numbers, real numbers, complex numbers, etc. There are infinitely many of each of these kinds of numbers, which makes them not representable in any finite number of bits. Cryptographic algorithms manipulate numbers with thousands of bits, and require exact, rather than approximate, representations of the quantities they manipulate.

Let's call the things in the set that a mathematical system operates on *elements*. Elements might be integers, real numbers, rational numbers, complex numbers, polynomials, matrices, or other objects. Various cryptographic algorithms require a mathematical system to have certain properties. Familiar properties of numbers in ordinary math are that there are two operations, + and \times (called *plus* and *times*, or *addition* and *multiplication*), and:

- **commutativity:** For any x and y , $x+y = y+x$ and $x\times y = y\times x$
- **associativity:** For any x , y , and z , $(x+y)+z = x+(y+z)$. Also, $(x\times y)\times z = x\times(y\times z)$
- **distributivity:** For any x , y , and z , $x\times(y+z) = (x\times y)+(x\times z)$
- **additive identity:** There is a number 0 such that for any x , $0+x=x$ and $x+0=x$
- **multiplicative identity:** There is a number 1 such that for any x , $1\times x=x$ and $x\times 1=x$
- **additive inverse:** For any x , there is a number $-x$ such that $x+(-x)=0$
- **multiplicative inverse:** For any x other than 0, there is a number x^{-1} such that $x\times x^{-1}=1$

A system that satisfies all these properties is called a **field**. A system that has these properties except that some elements might not have multiplicative inverses, and multiplication might not be commutative, is called a **ring**. A system that only has one operation, and for which every element has an inverse, is called a **group**.

Some cryptographic systems require a system with some or all the above properties and also require being able to exactly represent each element in a fixed number of bits. Will integers work? Integers have an additive identity, namely 0. And they have a multiplicative identity, namely 1.

However, most integers do not have multiplicative inverses. (For example, $\frac{1}{2}$ is the multiplicative inverse of 2 in real numbers, but $\frac{1}{2}$ is not an integer.) Also, integers can get arbitrarily large, and we want to be able to represent each element exactly in a fixed number of bits.

2.7.1 Finite Fields

Luckily, there is a mathematical structure that will do everything we want. It is called a **finite field**. A finite field of size n has n different elements and has all the properties we need. One form of finite field is the integers modulo p , where p is a prime. Arithmetic mod p uses integers between 0 and $p-1$. Addition and multiplication are done just like with regular arithmetic, but if the answer is not between 0 and $p-1$, the answer is divided by the modulus p , and the answer is the remainder. For example, with mod 7 arithmetic, the elements are $\{0, 1, 2, 3, 4, 5, 6\}$. If you add 2 and 4, you get 6, which is one of the elements. However, if you add 5 and 6, the answer is 11, so you need to divide by the modulus, 7, and take the remainder, which is 4. We will talk more about modular arithmetic in Chapter 6 *First-Generation Public Key Algorithms*, when we talk about the RSA algorithm, which is only secure if the modulus is *not* prime.

Évariste Galois founded the theory of finite fields. For each prime p , there is exactly one finite field with p elements, and it will be equivalent to the integers mod p . There is also exactly one finite field for each higher power of p , for instance, p^k . The field with p^k elements is usually denoted by $\text{GF}(p^k)$, where GF stands for *Galois field*, in honor of Galois. Arithmetic mod prime p would be denoted $\text{GF}(p)$, although it is sometimes denoted as \mathbf{Z}_p .

Note that if we use modular arithmetic, say mod 11, it will require four bits to represent each element, but there will be five values of the bits that will not be elements in the field $\{11, 12, 13, 14, 15\}$. This causes some problems, and it wastes space since we are using four bits to specify only 11 values. For that reason, cryptographic algorithms often use $\text{GF}(2^n)$ arithmetic, because each element corresponds to a unique n -bit value and vice versa.

Arithmetic in $\text{GF}(2^n)$ is very efficient. Addition is \oplus , which computers love to do. $\text{GF}(2^n)$ multiplication is also efficient (for computers). Multiplication in $\text{GF}(2^n)$ can be thought of as modular arithmetic of polynomials with coefficients mod 2. For instance, the bit string 110001 would represent the polynomial x^5+x^4+1 . Given that we need to represent $\text{GF}(2^n)$ elements in n bits, it is necessary after a multiplication of two n -bit polynomials to reduce the answer (which might be $2n-1$ bits) modulo a degree- n polynomial. The polynomial modulus needs to be **irreducible**, which in this case means only divisible by itself and 1.

There are also finite rings and finite groups, and these are also commonly used in cryptography. For instance, RSA, which uses modular arithmetic with a non-prime modulus, does not use a finite field, but it does use a finite ring.

2.7.2 Exponentiation

We talked about $+$ and \times as two operations on the elements in our set. But cryptographic algorithms often require exponentiation. Exponentiation is not another operation on two elements in the set like $+$ and \times are. Instead, exponentiation takes as input an element a and an integer x and multiplies a x times together. This is written a^x .

That's fine if x is small, but what if the exponent x is a number with thousands of digits? It would take several universe lifetimes, even for a supercomputer, to do that many multiplications. The trick that makes exponentiation by a very large number practical is *repeated squaring*. That means starting with a , squaring it (meaning multiplying it by itself) to get a^2 , and squaring a^2 to get a^4 , squaring a^4 to get a^8 , and so on. So, as long as the exponent x is a power of two, e.g., 2^k , we can compute a^x with only k multiplies (rather than $2^k - 1$ multiplies). If the exponent x is not a power of two, we can still use the trick of repeated squaring. If x is a power of two, it will look, in binary, like 1 followed by a bunch of zeroes. If x is not a power of two, the binary representation of x will consist of some 0s and some 1s. To raise a number a to the exponent x , do the following. Use a pointer that points to a bit in x and initially points to the leftmost (most significant) bit of x . Have a second value that we will refer to as the intermediate value, which is initialized to 1. For each bit in x , do the following:

1. If the bit in x pointed to by the pointer is 1, multiply the intermediate value by a .
2. If the pointer is at the rightmost bit of x , you are done; otherwise, move the pointer to the right one bit position.
3. Square the intermediate value.

Using this algorithm, the number of multiplies to raise something to the power x depends on the number of 1s in the binary representation of x . In the worst case, it will take twice the number of bits in x (if all the bits in x are 1). In the best case, it will take the number of bits in x (if x is a power of two).

We will go deeper into these concepts as they are required for specific algorithms.

2.7.3 Avoiding a Side-Channel Attack

A straightforward implementation of the algorithm in §2.7.2 *Exponentiation* is an opportunity for a side-channel attack, because the implementation would behave differently on each bit of the exponent, depending on whether the bit was a 0 or a 1. An implementation can avoid providing a side-channel by behaving the same way on each bit. For instance, it could always implement step 2 (multiplying the intermediate value by a), saving both the result of step 1 and of step 2, and then discard the result of step 1 if the bit was a 1 or discard the result of step 2 if the bit was a 0.

Even this remediation might still allow a side-channel attack because anything that contains conditional branches might cause observable behavior differences, such as which memory locations are read and written. So, an implementation might execute the exponentiation algorithm by computing both the result of squaring alone (let's call that the IF0 value) and of squaring and multiplying by a (let's call that the IF1 value). Then the implementation can take the relevant bit in the exponent (which will be 0 or 1), and \oplus the bit times the IF1 value with the complement of the bit times the IF0 value.

Even this might not be enough. To truly eliminate side channels, you need to know the details of the particular platform.

2.7.4 Types of Elements used in Cryptography

Sometimes cryptographic systems use mod p arithmetic, where p is a prime. RSA, as we will see, uses mod n arithmetic, where n is definitely not a prime. Other cryptographic systems, especially some of the post-quantum algorithms, use polynomials or matrices, but with coefficients that use either modular arithmetic or $GF(2^n)$. When modular arithmetic is used in post-quantum cryptography, the modulus is either prime or a power of two. Note that numbers modulo a power of two do not form a finite field (because powers of two other than 2^1 are not prime).

2.7.5 Euclidean Algorithm

The Euclidean algorithm is an efficient method of finding the greatest common divisor (gcd) of two numbers a and b . The gcd of a and b is written as $\text{gcd}(a,b)$. It is the largest number that evenly divides both a and b .

The Euclidean algorithm can also be used to find the multiplicative inverse of $b \bmod a$, and this is what the Euclidean algorithm is mostly used for in cryptography. Sometimes, using the Euclidean algorithm for finding multiplicative inverses is called the extended Euclidean algorithm.

The insight into efficiently finding gcds is that if a number d is a divisor of both a and b , it is also a divisor of $a-b$. It is also a divisor of a minus any multiple of b . Therefore, if you divide a by b , the remainder is also divisible by d . The Euclidean algorithm starts with a and b , and at each step, calculates smaller numbers that are divisible by d , until the last step when the remainder is 0.

At each step, there will be two numbers A and B , initialized to a and b . Divide A by B to get the remainder R , which will be smaller than B . At the next step, set A equal to B , and B equal to R , and keep iterating until R is 0. The final value of B will be the gcd of a and b . For example, consider finding the gcd of 420 and 308.

$$420 \div 308 = 1 \text{ remainder } 112$$

$$308 \div 112 = 2 \text{ remainder } 84$$

$$112 \div 84 = 1 \text{ remainder } 28$$

$$84 \div 28 = 3 \text{ remainder } 0$$

Therefore, 28 is the gcd of 420 and 308.

When we want to find the multiplicative inverse of $b \bmod a$, the goal is to find integers u and v such that $u \times a + v \times b = 1$. Then v will be the multiplicative inverse of $b \bmod a$, because $v \times b$ will be 1 more than a multiple of a . The integer b will only have an inverse mod a if $\gcd(a, b) = 1$. So let's use two numbers a and b such that $\gcd(a, b) = 1$. We'll use 109 and 25. First, let's apply the Euclidean algorithm to find $\gcd(109, 25)$.

$$109 \div 25 = 4 \text{ remainder } 9$$

$$25 \div 9 = 2 \text{ remainder } 7$$

$$9 \div 7 = 1 \text{ remainder } 2$$

$$7 \div 2 = 3 \text{ remainder } 1$$

$$2 \div 1 = 2 \text{ remainder } 0$$

So, $\gcd(109, 25) = 1$. We want to represent each of the remainders as some multiple of a (109) plus some multiple of b (25). We know that $a = 1 \times a + 0 \times b$ and that $b = 0 \times a + 1 \times b$. After the first step, we know that $a \div b = 4 \text{ remainder } 9$, or rewriting, we get

$$9 = 1 \times a - 4 \times b.$$

The second line tells us that $25 \div 9 = 2 \text{ remainder } 7$. That means $7 = 25 - 2 \times 9$. Well, $25 = b$. And 9 is $1 \times a - 4 \times b$. Substituting for 9 , we get $7 = 25 - 2 \times (1 \times a - 4 \times b)$. Since $25 = b$, this simplifies to

$$7 = -2 \times a + 9 \times b.$$

The third line tells us that $9 \div 7 = 1 \text{ remainder } 2$. That means that $2 = 9 - 1 \times 7$. Substituting for 9 ($1 \times a - 4 \times b$) and 7 ($-2 \times a + 9 \times b$), we get

$$2 = 3 \times a - 13 \times b.$$

The next line tells us that $7 \div 2 = 3 \text{ remainder } 1$. That means that $1 = 7 - 3 \times 2$. Substituting for 7 ($-2 \times a + 9 \times b$) and 2 ($3 \times a - 13 \times b$), we get $1 = (-2 \times a + 9 \times b) - 3 \times (3 \times a - 13 \times b)$, which simplifies to

$$1 = -11 \times a + 48 \times b.$$

This tells us that 48 times b is one more than a multiple of a . In other words, 48 is b 's inverse mod a . Going back to the values we chose for a and b (109 and 25), we have calculated that 25's inverse mod 109 is 48. And indeed, $25 \times 48 = 1200$ and $1200 \div 109 = 11 \text{ remainder } 1$.

2.7.6 Chinese Remainder Theorem

The Chinese Remainder Theorem will be useful for making certain RSA operations more efficient. We'll give a special case definition, since that's all that we need. As we will see, a typical RSA modulus n is the product of two primes p and q . The Chinese Remainder Theorem states that if you know what a number, say x , equals mod n , then you can easily calculate what x is mod p and what x is mod q . And likewise, if you know what x is mod p , and what x is mod q , then you can easily calculate what x is mod n .

It's pretty easy to see how knowing what x is mod n enables you to calculate what x is mod p , and what x is mod q . To find out what x is mod p , just divide x mod n by p and the remainder will be what x is mod p . Likewise for calculating what x mod n is mod q .

The other direction is a bit trickier. Suppose you know that a number equals a mod p and b mod q , and you want to know what it is mod pq .

The first thing you need to do is calculate p 's multiplicative inverse mod q and q 's multiplicative inverse mod p . You can do that with the extended Euclidean algorithm.

Now you know p^{-1} mod q . And you know q^{-1} mod p . Note

$$p \times (p^{-1} \text{ mod } q) = 1 \text{ mod } q \text{ and it is equal to } 0 \text{ mod } p.$$

$$q \times (q^{-1} \text{ mod } p) = 1 \text{ mod } p \text{ and } 0 \text{ mod } q.$$

You are looking for a number that is a mod p and b mod q . Multiply $q \times (q^{-1} \text{ mod } p)$ by a , but don't reduce mod p . You'll get something that is a mod p and also 0 mod q . (When you started, you knew that a was equal to a mod p , but a is probably not equal to 0 mod q .)

Likewise, multiply $p \times (p^{-1} \text{ mod } q)$ by b , but don't reduce mod q . You now have a number that is equal to 0 mod p and b mod q .

Add these two quantities together. You'll get $a \times (q \times (q^{-1} \text{ mod } p)) + b \times (p \times (p^{-1} \text{ mod } q))$. Now you can reduce mod n , and the result will be the number, mod n , that is also equal to a mod p and b mod q .

Why is the Chinese Remainder Theorem useful? As we will see in §6.3.4.5 *Optimizing RSA Private Key Operations*, instead of doing calculations mod n , the numbers can be converted into their representations mod p and mod q , and the operations can be performed mod p and mod q , and knowing the answer mod p , and knowing the answer mod q , the answer can be converted into the answer mod n . Since p and q are roughly half the size of n , this will be more efficient than doing the operations mod n .

Note the Chinese Remainder Theorem holds even if the smaller moduli are not themselves prime, as long as they are relatively prime (no factors in common). Since people think of p and q as primes, we'll use j and k and restate the Chinese Remainder Theorem as "If j and k are relatively prime and you know a mod j and a mod k , you can compute a mod jk , and vice versa."

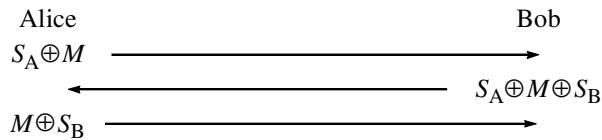
2.8 HOMEWORK

1. What is the dedication to this book?
2. Random J. Protocol-Designer has been told to design a scheme to prevent messages from being modified by an intruder. Random J. decides to append to each message a hash of that message. Why doesn't this solve the problem?

3. Suppose Alice, Bob, and Carol want to use secret key technology to authenticate each other. If they all used the same secret key K , then Bob could impersonate Carol to Alice (actually any of the three can impersonate the other to the third). Suppose instead that each had their own secret key, so Alice uses K_A , Bob uses K_B , and Carol uses K_C . This means that each of Alice, Bob, and Carol, to prove his or her identity, responds to a challenge with a function of his or her secret key and the challenge. Is this more secure than having them all use the same secret key K ? (Hint: What does Alice need to know in order to verify Carol's answer to Alice's challenge?)
4. As described in §2.4.4 *Efficient Digital Signatures*, it is common, for performance reasons, to sign a hash of a message rather than the message itself. Why is it so important that it be difficult to find two messages with the same hash?
5. Assume a cryptographic algorithm in which the performance for the good guys (the ones that know the key) grows linearly with the length of the key, and for which the only way to break it is a brute-force attack of trying all possible keys. Suppose the performance at a certain key-size is adequate for the good guys (*e.g.*, encryption and decryption can be done as fast as the bits can be transmitted over the wire). Then suppose advances in computer technology make computers twice as fast. Given that both the good guys and the bad guys get faster computers, does this advance in computer speed work to the advantage of the good guys, the bad guys, or does it not make any difference?
6. Suppose you had a source of really good randomness, and you copied the random output into two places and \oplus 'd these. How random would the result be? What if you concatenated the two quantities and hashed the result? Would that be random?
7. Suppose the seed for a PRNG (see §2.6.3 *Calculating a Pseudorandom Stream from the Seed*) is n bits long. How many bits of output of the PRNG would you need to see in order to verify a guess of a seed that it is using (with high probability)?
8. Suppose you could see some, but not all, of the output of a PRNG. Assuming you know the algorithm the PRNG is using, would you be able to verify a guess of a seed based solely on seeing every tenth bit? Would this require trying more potential seeds than if you were able to see all the bits of the PRNG output (assuming you were able to see enough output bits)?
9. Suppose you could see 400 bits of output of a PRNG every second, and you knew that it started with eight bits of randomness and mixed in eight bits of new randomness every second. How difficult would it be, after sixteen seconds, to calculate what the state of the PRNG is, compared to a PRNG that mixed in 128 bits of randomness every sixteen seconds?
10. Suppose a process generates about eight bits of randomness every second, and suppose a PRNG used the 8-bit random output every second, and suppose you could see the output of the PRNG. Why is it better to wait until you have, say, 128 bits of randomness, before using the randomness to reseed your PRNG? In other words, if the random seed were a function of

sixteen 8-bit random chunks, how many possible seeds would there be? What if you waited until there were 128 bits of randomness?

11. Suppose Alice wants to send a secret message M to Bob, but has not agreed upon a secret with Bob. Furthermore, assume that Alice can be assured that when she sends something to Bob, it will arrive at Bob without anyone having tampered with the message. So the only threat is someone reading the transmissions between Alice and Bob. Alice chooses a random secret S_A , and sends $S_A \oplus M$ to Bob. Nobody (including Bob) can read this message. Bob now creates his own secret S_B , \oplus s what he received from Alice with S_B , and transmits $S_A \oplus M \oplus S_B$ to Alice. Alice now removes her secret by \oplus ing with S_A and returns $M \oplus S_B$ to Bob. Bob can now \oplus with his secret (S_B) in order to read the message. Is this secure?



12. Use the Euclidean algorithm to compute $\gcd(1953, 210)$.
13. Use the Euclidean algorithm to compute the multiplicative inverse of 9 mod 31.
14. A number is 11 mod 36 and also 7 mod 49. What is the number mod 1764? How about if a number is 11 mod 36 and also 11 mod 49—what would that number be mod 1764?

3

SECRET KEY CRYPTOGRAPHY

3.1 INTRODUCTION

Secret key encryption schemes require that both the party that does the encryption and the party that does the decryption share a secret key. We will discuss two types of secret key encryption schemes:

- **block cipher.** This takes as input a secret key and a plaintext block of fixed size (older ciphers used 64-bit blocks, modern ciphers use 128-bit blocks). It produces a ciphertext block the same size as the plaintext block. To encrypt messages larger than the blocksize, the block cipher is used iteratively with algorithms called *modes of operation* that are the subject of the next chapter. A block cipher also has a decryption operation that does the reverse computation.
- **stream cipher.** This uses the key as a seed for a pseudorandom number generator, produces a stream of pseudorandom bits, and \oplus s (bitwise exclusive ors) that stream with the data. Since \oplus is its own inverse, the same computation performs both encryption and decryption.

This chapter describes the block ciphers DES, 3DES, and AES. The world has mostly converted to AES. However, it is useful to see the structure of DES and 3DES because they give insight into the design of a block cipher. This chapter also describes the stream cipher RC4. Many weaknesses have been found in RC4, so although it has nice performance properties, it is no longer widely used. We describe RC4 because it is an example of a stream cipher that isn't constructed from a block cipher.

3.2 GENERIC BLOCK CIPHER ISSUES

3.2.1 Blocksize, Keysize

It's fairly obvious that if the keysize is too small (for instance, four bits), the cryptographic scheme would not be secure because it would be too easy to search through all possible keys. There's a

similar issue with the size of the block of plaintext to be encrypted. If the blocksize is too small (say one octet, as in a monoalphabetic cipher), then if you ever had enough paired $\langle \text{plaintext}, \text{ciphertext} \rangle$ blocks, you could construct a table to be used for decryption. It might be possible to obtain such pairs because messages might only remain secret for a short time, for example, if a message says where the army will attack the next day.

Having a blocksize too large has performance penalties, so you wouldn't want a blocksize much larger than you need. The cryptographer's guideline is that with a blocksize of n bits, you shouldn't encrypt more than $2^{n/2}$ blocks with the same key, and, in fact, you should change your key significantly more frequently than once every $2^{n/2}$ blocks. In the 1970s, when DES was being designed, 64 bits seemed a reasonable size, since no one expected to be encrypting anywhere close to 32 gigabytes of data. Modern standards call for 128-bit blocks, which is reasonably safe as long as you don't plan on encrypting petabytes of data without changing the key.

3.2.2 Completely General Mapping

For ease of explanation (and readability, because we are going to need to write out 64-bit values), let's assume for now a scheme that uses 64-bit blocks. The most general way of encrypting a 64-bit block is to take each of the 2^{64} input values and map it to a unique one of the 2^{64} output values. (It is necessary that the mapping be **one-to-one**, *i.e.*, only one input value maps to any given output value, since otherwise decryption would not be possible.)

Suppose Alice and Bob want to decide upon a mapping that they can use for encrypting their conversations. How would they specify it? To specify a monoalphabetic cipher with English letters requires mapping each of 26 values to 26 possible values, which can be specified with 26×5 bits. For instance,

```
a→q  b→d  c→w  d→x  e→a  f→f  g→z  h→b  ...
```

But now assume a block cipher that maps 64-bit blocks to 64-bit blocks. How would you specify a general mapping? Well, let's start:

```
00000000000000000000000000000000→8ad1482703f217ce
00000000000000000000000000000001→b33dc8710928d701
00000000000000000000000000000002→29e856b28013fa4c
```

Hmm, we probably don't want to write this all out. There are 2^{64} possible input values, and for each one we have to specify a 64-bit output value. Constructing such a table would take 2^{70} bits. In theory, Alice and Bob could share a 2^{70} -bit quantity specifying the full mapping, using it like a

secret key. But it is doubtful that they could remember a key that large, or even be able to say it to each other within a lifetime, or store it on anything. So this is not particularly practical.

3.2.3 Looking Random

Block ciphers are designed to take a reasonable-size key (*i.e.*, more like 128 bits rather than the 2^{70} bits as described in the previous section) and generate a one-to-one mapping that looks completely random to someone who does not know the key. *Looking random* means that it should look (to someone who doesn't know the key) as if the mapping from an input value to an output value were generated by using a random number generator. A block cipher that looks random in this way is called a **pseudorandom permutation (PRP)**.

If the mapping were truly random, any single-bit change to the input would result in a totally independently chosen random number output. The two different output numbers should have no correlation, meaning that about half the bits should be the same and about half the bits should be different. For instance, it can't be the case that the third bit of output always changes if the twelfth bit of input changes, or even that the probability of the third bit of output changing if the twelfth bit of input changes is anything significantly different from $\frac{1}{2}$ (averaged over all possible inputs). So cryptographic algorithms are designed to *spread bits around*, in the sense that a single input bit should have influence on all the bits of the output, and be able to change any one of them with a probability of about $\frac{1}{2}$ (depending on the values of the other bits of input).

A (not very practical) way of creating a truly random mapping uses an imaginary box whimsically called an **oracle**. Note that a similar concept exists with hashes, and there it is called the *random oracle model*. The box will answer questions of the form “What is P encrypted with key K ?” and “What is C decrypted with key K ?” The oracle will keep a table of entries $\langle \text{key } K, \text{plaintext } P, \text{ciphertext } C \rangle$, to remember all the answers it has ever given. To answer “What is plaintext P encrypted with K ?", the box will check whether it has an entry for $\langle K, P, C \rangle$. If so, it will return the answer “ C ”. If no such entry exists, the box will generate a random value R , and if R does not already exist as the answer for that K and some other P , the box will make an entry $\langle K, P, R \rangle$, and reply “ R ”. If R already exists as ciphertext in some entry, then the box chooses a different random R .

Likewise for decryption. If the box is asked “What is ciphertext C decrypted with key K ?”, the box will answer “ P ” if there is an entry $\langle K, P, C \rangle$. Otherwise, it will generate a random value R and check whether the triple $\langle K, R, x \rangle$ already exists for any ciphertext x . If so, it generates a different R and checks again. If not, it enters $\langle K, R, C \rangle$ and returns “ R ”.

This obviously impractical approach of asking an oracle to encrypt and decrypt each block is used to define the **ideal cipher model**. Ideal cipher security is stronger than PRP security, because if someone knows the key that generates the PRP, they can see that it is not random. Unlike PRP security, but like random oracle security, ideal cipher security is so strong that it is provably impos-

sible for any efficiently computable function to have it [CANE98]. Nonetheless, the ideal cipher model gives block cipher designers something to aspire to. Both the ideal cipher model and the PRP model are used in security proofs, where the proof assumes the underlying block cipher conforms to the PRP (or ideal cipher) model, and then proves properties of the system.

3.3 CONSTRUCTING A PRACTICAL BLOCK CIPHER

A generic block cipher takes as input an n -bit plaintext block and a key, and outputs an n -bit ciphertext block. The usual construction uses an inexpensive but not-very-secure block cipher multiple times. This is analogous to shuffling cards multiple times. Each time is called a **round**. Let's call the not-very-secure block cipher a *round transformation*. In designing a block cipher, cryptographers try to match the strength of the keysize with the number of rounds. Once there are enough rounds so that the most efficient method of breaking the cipher is by doing a brute-force search on all the potential keys, doing extra rounds is a waste of computation. If there aren't enough rounds, then a larger keysize does not add security.

3.3.1 Per-Round Keys

The round takes a key as a parameter, so that the round transformation will be different depending on the key. Sometimes using the same key for all rounds can lead to attacks that can't be fixed by simply increasing the number of rounds, so typically each round uses a different key. If the block cipher does r rounds, and each round uses an x -bit key, this would result in a very large key ($x \times r$ bits). So, instead of requiring the block cipher to use an $x \times r$ -bit key, per-round keys are derived from the actual block cipher key, say k bits. DES simply uses different 48-bit subsets of the 56-bit key for each round, while AES uses a slightly more complicated approach to get per-round keys that are less obviously related to one another. Creating per-round keys from the main key is referred to as the **key expansion** step or **key schedule**. In cases where many plaintext blocks will be encrypted with the same key, it can be a performance optimization to only perform the key expansion step once and cache the key schedule.

3.3.2 S-boxes and Bit Shuffles

The round transformation (in both DES and AES) contains a component that transforms a set of input bits into a set of output bits. This function is known in the literature as an **S-box** (S stands for

substitution). An S-box specifies, for each of the 2^k possible values of the k -bit input, the value of the output. To make the round transformation practical to implement, the S-box only maps a small number of bits, instead of mapping full blocks. To use the small S-box, the round breaks the input into pieces and uses an S-box on each piece. An S-box in DES maps 6-bit inputs to 4-bit outputs. The S-box in AES maps 8-bit inputs into 8-bit outputs. DES uses eight different S-boxes, *i.e.*, each S-box does a different mapping. In AES, all the S-boxes are identical.

Note that since the S-boxes only act on a subset of the bits, if we only did a single round, then a bit of input can only affect a small number of bits of the output (*e.g.*, eight bits for AES), since each input bit goes into only one of the S-boxes. (To be precise, in AES each bit goes into only one S-box. In DES, some of the bits are input into two S-boxes.)

In each round, after the S-boxes, the bits affected by a particular input bit get spread around so that after several rounds, each single input bit will affect all the output bits. DES calls this operation a **P-box** (the P is for *permutation*). The word “permutation” is sometimes used to describe functions of n input bits that rearrange the positions of those n input bits. “Permutation” is also sometimes used to describe any one-to-one mapping from the 2^n possible input values of n bits to the 2^n possible output values. The P-box in DES is a permutation in the first sense; to avoid confusion, we will call the DES P-box a *bit shuffle*. A **bit shuffle** takes as input n bits and moves the bits around, so that the third bit might become the eleventh bit. The output will have the same number of 0s and 1s as the input.

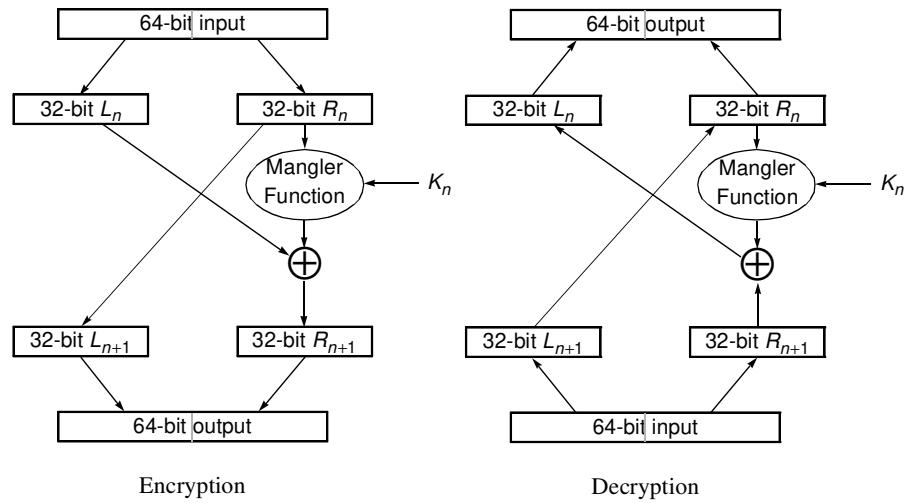
3.3.3 Feistel Ciphers

Of course, it is important that an encryption algorithm can be reversed so that decryption is possible. One method (used by AES) of having a cipher that is reversible is to make all the components reversible. With DES, the S-boxes are clearly not reversible since they map 6-bit inputs to 4-bit outputs. So instead, DES is designed to be reversible using a clever technique known as a **Feistel cipher**.

A Feistel cipher [FEIS73] builds reversible transformations out of one-way transformations by only working on half the bits of the input value at a time. Let’s assume 64-bit blocks. Figure 3-1 shows both how encryption and decryption work. In a Feistel cipher there is some irreversible component that scrambles the input. We’ll call that component the *mangler function*.

In encryption for round n , the 64-bit input to round n is divided into two 32-bit halves called L_n and R_n . Round n generates as output 32-bit quantities L_{n+1} and R_{n+1} . The concatenation of L_{n+1} and R_{n+1} is the 64-bit output of round n and, if there’s another round, the input to round $n+1$.

L_{n+1} is simply R_n . To compute R_{n+1} , do the following. R_n and K_n are input to the mangler function, which takes as input 32 bits of data plus some bits of key to produce a 32-bit output. The 32-bit output of the mangler is \oplus ’d with L_n to obtain R_{n+1} .

**Figure 3-1.** Feistel Cipher

Given the above, suppose you want to run a Feistel cipher (e.g., DES) backward, *i.e.*, to decrypt something. Suppose you know the round key K_n , and the two 32-bit outputs L_{n+1} and R_{n+1} . How do you get L_n and R_n ? Well, R_n is just L_{n+1} . Now you know R_n , L_{n+1} , R_{n+1} , and K_n . You also know that R_{n+1} equals $L_n \oplus \text{mangler}(R_n, K_n)$. You can compute $\text{mangler}(R_n, K_n)$ since you know R_n and K_n . Now \oplus that with R_{n+1} . The result will be L_n . Note that the mangler is never run backwards. The Feistel cipher is elegantly designed to be reversible without constraining the mangler function to be reversible. Theoretically, the mangler could map all values to zero, and it would still be possible to run the algorithm backwards, but having the mangler function map all functions to zero would make the algorithm pretty insecure (see Homework Problem 8).

If you examine Figure 3-1 carefully, you will see that decryption is identical to encryption with the 32-bit halves swapped. In other words, feeding $R_{n+1}|L_{n+1}$ into round n produces $R_n|L_n$ as output. If the algorithm specifies that at the end of a series of Feistel rounds the two halves of the output are swapped (and DES does this), there is a cute trick one can do in an implementation. A single implementation can perform either encryption or decryption depending only on the order in which the per-round keys are supplied. So whether you are encrypting or decrypting depends only on the key schedule supplied, where decryption reverses the order of the per-round keys.

3.4 CHOOSING CONSTANTS

Cryptographers are (justifiably) a suspicious bunch. It is theoretically possible to design a cryptographic function that is breakable if you know the secret of how it was designed. Such a function (that is breakable by the designer) is said to have a **back door**. For example, the S-boxes in DES seem chosen arbitrarily, so it is understandable for someone to be suspicious about why that particular design was chosen.

It is common for cryptographic algorithms to have components that need somewhat arbitrary parameters. To avoid suspicion, designers today choose constants in such a way that they are outside the control of the designer, and yet still have the necessary properties for the security of the design. This technique for designing algorithms is sometimes known as the **nothing up my sleeve** technique. Example choices of constants might be some number of bits of π or $\sqrt{2}$.

3.5 DATA ENCRYPTION STANDARD (DES)

DES was published in 1977 by NIST (the National Institute of Standards and Technology, at that time called NBS—the National Bureau of Standards) for use in commercial and unclassified U.S. Government applications. It was designed by a team from IBM based on their own *Lucifer cipher*, with consultation from NSA. DES uses a 56-bit key, and maps a 64-bit input block into a 64-bit output block. The key actually looks like a 64-bit quantity, but one bit in each of the eight octets is used for odd parity on each octet. Therefore, only seven of the bits in each octet are actually meaningful as a key.

DES is efficient to implement in hardware but relatively slow if implemented in software. Although making software implementations difficult was not a documented goal of DES, people have asserted that DES was specifically designed with this in mind, perhaps because this would limit its use to organizations that could afford hardware-based solutions, or perhaps because it made it easier to control access to the technology. At any rate, advances in CPUs have made the cost of implementing DES in software acceptable.

Advances in semiconductor technology, and the ability to use technology such as GPUs for inexpensive massive parallelism, make the keyspace issue more critical. Perhaps a 64-bit key might have extended the useful lifetime of DES by a few years, but even that would not be secure today. NIST today recommends that all crypto be at least as hard to break as a block cipher with a keyspace of 112 bits, and most designs aim for at least 128 bits.

Why 56 bits?

Use of a 56-bit key is one of the most controversial aspects of DES. Even before DES was adopted, people outside of the intelligence community complained that 56 bits provided inadequate security [DENN82, DIFF76a, DIFF77, HELL79]. So why were only 56 of the 64 bits of a DES key used in the algorithm? The disadvantage of using eight bits of the key for parity checking is that it makes DES considerably less secure (256 times less secure against exhaustive search).

OK, so what is the advantage of using eight bits of the key for parity? Well, let's say you receive a key electronically, and you want to sanity-check it to see if it could actually be a key. If you check the parity of the quantity, and it winds up not having the correct parity, then you'll know something went wrong.

The problem with this reasoning is that there is a one in 256 chance (given the parity scheme) that even if you were to get 64 bits of garbage, the result will happen to have the correct parity and therefore look like a key. That is way too large a possibility of error for it to afford any useful protection to any application.

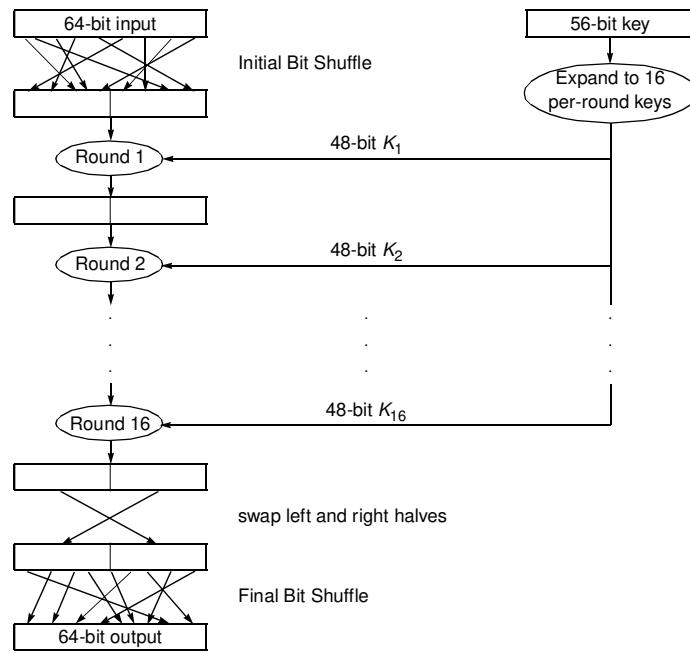
The key, at 56 bits, was always pretty much universally acknowledged to be too small to be secure. People (not us, surely!) have suggested that our government consciously decided to weaken the security of DES just enough so that NSA would be able to break it. We would like to think there is an alternative explanation, but we have never heard a plausible one proposed.

3.5.1 DES Overview

DES is quite understandable, and has some very elegant tricks. Let's start with the basic structure of DES (Figure 3-2).

The 64-bit input is subjected to an initial bit shuffle (which has no security value) to obtain a 64-bit result. The 56-bit key is expanded into sixteen 48-bit per-round keys by taking a different 48-bit subset of the 56 bits for each of the keys. Each round takes as input the 64-bit output of the previous round and the 48-bit per-round key, and produces a 64-bit output. After the sixteenth round, the 64-bit output has its halves swapped and is then subjected to another bit shuffle that happens to be the inverse of the initial bit shuffle. Swapping the two halves of the 64-bit output after the final round adds no cryptographic strength to the algorithm, but it does have an interesting side benefit. As noted in §3.3.3 *Feistel Ciphers*, swapping the two halves allows the encrypt and decrypt operations to be identical except for the key schedule.

That is the overview of how encryption works. Decryption works by essentially running DES backwards. To decrypt a block, you'd first run it through the initial bit shuffle to undo the final bit

**Figure 3-2.** Basic Structure of DES

shuffle. (The initial and final bit shuffles are inverses of each other.) You'd do the same key expansion, though you'd use the keys in the opposite order (first using K_{16} , the key you produced last). Then you run sixteen rounds just like for encryption. (We explained why this works in §3.3.3 *Feistel Ciphers*.) After sixteen rounds of decryption, the output has its halves swapped and is then subjected to the final bit shuffle (to undo the initial bit shuffle).

3.5.2 The Mangler Function

The mangler function takes as input the 32-bit R_n (the right half of the 64-bit input into round n) that we'll simply call R , and the 48-bit K_n that we'll call K , and produces a 32-bit output that, when \oplus 'd with L_n , produces R_{n+1} (the next R).

The mangler function first expands R from a 32-bit value to a 48-bit value. It does this by breaking R into eight 4-bit chunks and then expanding each of those chunks to 6 bits by taking the

adjacent bits and concatenating them to the chunk. The leftmost and rightmost bits of R are considered adjacent. (See Figure 3-3.)

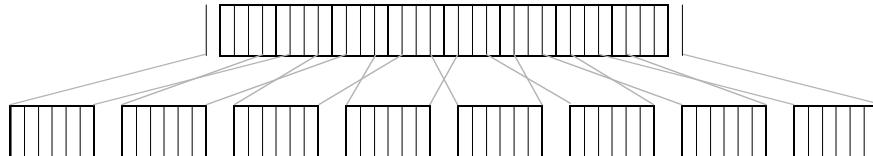


Figure 3-3. Expansion of R to 48 bits

The 48-bit round key K is broken into eight 6-bit chunks. Chunk i of the expanded R is \oplus 'd with chunk i of K to yield a 6-bit output. That 6-bit output is fed into an **S-box**, a substitution that produces a 4-bit output for each possible 6-bit input. Since there are 64 possible input values (6 bits) and only sixteen possible output values (4 bits), the S-box clearly maps several input values to the same output value. In DES, there are eight S-boxes, each of which does a different mapping of 6-bit inputs to 4-bit outputs.

3.5.3 Undesirable Symmetries

The combination of the fact that DES key expansion only involves picking subsets of the initial key bits, the fact that the only use of per-round keys is to \oplus them with plaintext or intermediate values, and the use of a Feistel structure, results in unfortunate symmetries that result in weaknesses in DES. Nobody really cares at this point, because the small keysize makes DES obsolete, but these weaknesses do illustrate the kinds of weaknesses that can be avoided with careful design. The DES weaknesses covered in this section are:

- There are four keys that are *weak*, meaning that these keys are their own inverses, *i.e.*, encrypting twice with the same key results in the plaintext. Two obvious weak keys are all 0s and all 1s, because round keys are just different subsets of the key bits. So, for example, with a key of zero, all round keys (in either decrypt or encrypt) will be zero, and since encrypt and decrypt are the same operation, except for the order of the round keys, encrypt will be the same operation as decrypt.
- There are six pairs of keys that are *semi-weak*, meaning that the keys in the pair are inverses of each other.
- For all keys K , if plaintext P encrypted with key K becomes ciphertext C , then $\sim P$ encrypted with $\sim K$ becomes $\sim C$. (Note: $\sim x$ is the one's complement of x , *i.e.*, x with all its bits inverted—0s changed to 1s and vice versa.) What is the implication of this? Suppose there is

a system that encrypts using a key K , and it allows the attacker (say Eve) two chosen plaintexts. Now assume Eve chooses P and $\sim P$ to be encrypted, so Eve will know the encryption of P is C_1 , and the encryption of $\sim P$ is C_2 . Now Eve wants to do a brute-force search for key K . If DES (with a 56-bit key) did not have this weakness, it would take 2^{56} encryptions (worst case), or 2^{55} encryptions (average case) in the brute-force search to find a key K that mapped P to C_1 . However, with this weakness, the brute-force search for Eve only takes half as many encryptions. The reason for this is that Eve need only try half the keys, say, the keys that have the top bit 0. For each such key K , Eve encrypts P with K . If the result is C_1 , then she knows that K was the key. If the result is $\sim C_2$, then she knows that the key is $\sim K$. Because she only needs to do encryptions with half the keys, it would take 2^{55} encryptions (worst case), or 2^{54} encryptions (average case).

3.5.4 What's So Special About DES?

DES is actually quite simple. One gets the impression that anyone could design a secret key encryption algorithm. Just take the input and key, mix them together somehow, do this over and over until you think it's enough, and you have an algorithm. In fact, however, these things are very mysterious. For example, the DES S-boxes seem totally arbitrary. However, they must have been carefully designed for strength. Biham and Shamir [BIHA91] have shown that with an incredibly trivial change to DES consisting of swapping S-box 3 with S-box 7, DES is about an order of magnitude less secure in the face of a specific (admittedly not very likely) attack. [COPP94] is a nice paper that describes how DES was secretly designed to avoid differential cryptanalysis. There is, however, another cryptanalysis technique, linear cryptanalysis, that first appears in public literature in [MATS92]. The DES S-boxes do not appear to be designed to be especially strong against linear cryptanalysis, suggesting that the DES designers didn't know about it. Linear cryptanalysis was applied to attack DES (still not very practically) in 1993 and 1994 [MATS93, MATS94].

3.6 3DES (MULTIPLE ENCRYPTION DES)

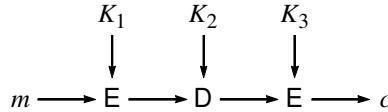
Remember that a cryptographic scheme has two functions, known as *encrypt* and *decrypt*. The two functions are inverses of each other, but in fact each one takes an arbitrary block of data and garbles it in a way that is reversed by the other function. So it would be just as secure to perform the *decrypt* operation on the plaintext as a method of encrypting it, and then perform *encrypt* on the result as a method of getting back to the plaintext again. It might be confusing to say something like *encrypt with decryption*, so we'll just refer to the two functions as E and D.

The generally accepted method of making DES more secure through multiple encryptions is known as **EDE**, **3DES**, **TDEA (Triple Data Encryption Algorithm)**, or **TDES (Triple Data Encryption Standard)**.

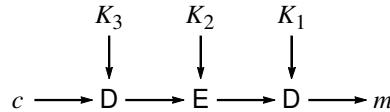
Actually, any encryption scheme might be made more secure through multiple encryptions. EDE could as easily be done with, say, AES. But AES has already standardized variants with different keysizes, and it is much more efficient to create versions of AES with larger keysizes by modifying the algorithm than by doing multiple encryptions with a small-keysize version of AES.

DES uses EDE as follows:

1. Three keys are used: K_1 , K_2 , and K_3 .
2. Each block of plaintext is subjected to E with K_1 , then D with K_2 , and then E with K_3 . The result is simply a new secret key scheme—a 64-bit block is mapped to another 64-bit block.



Decryption simply reverses the operation.



Now we'll discuss why 3DES is defined this way. There are various choices that could have been made:

1. Three encryptions were chosen. It could have been two or 714. Is three the right number?
2. Why are the functions EDE rather than, say, EEE or EDD?

3.6.1 How Many Encryptions?

Let's assume that the more times the block is encrypted, the more secure it is. So encrypting 714 times would be some amount more secure than encrypting three times. The problem is that it is expensive to do an encryption. We don't want to do any more encryptions than are necessary for the scheme to be really secure. There are problems with using only two encryptions (see §3.6.1.2 *Encrypting Twice with Two Keys*), and three was chosen because it is the smallest integer bigger than two.

3.6.1.1 Encrypting Twice with the Same Key

Suppose we didn't want to bother with two keys. Would it make things more secure if we encrypted twice in a row with the same key?

$$\text{plaintext} \xrightarrow{K} \xrightarrow{K} \text{ciphertext}$$

This turns out not to be much more secure than single encryption with K , since exhaustive search of the (56-bit) keyspace still requires searching only 2^{56} keys. Each step of testing a key is twice as much work, since the attacker needs to do double encryption, but a factor of two for the attacker is not considered much added security, especially since the good guys have their work doubled as well.

(How about E followed by D using the same key? That's double the work for the good guy and no work for the bad guy, so that's generally not considered good cryptographic form.)

3.6.1.2 Encrypting Twice with Two Keys

If encrypting twice using two different keys were as secure as a DES-like scheme with a keysize of 112 bits, then encrypting twice would have sufficed. However, it isn't, as we will show.

We'll use two DES keys, K_1 and K_2 , and encrypt each block twice, first using key K_1 and then using key K_2 .

$$\text{plaintext} \xrightarrow{K_1} \xrightarrow{K_2} \text{ciphertext}$$

Is this as cryptographically strong as using a double-length secret key (112 bits)? The reason you might think it would be as strong as a 112-bit key is that the straightforward brute-force attack would have to guess both K_1 and K_2 in order to determine if a particular plaintext block encrypted to a particular ciphertext block.

However, there is a less straightforward attack that breaks double-encryption DES in roughly twice the time of a brute-force breaking of single-encryption DES. The attack is not particularly practical as it requires a somewhat unreasonably large amount of memory. However, there are variants of this attack that require only a little bit more computation and much less memory. For this reason, double encryption is not generally done.

One attack (called a **Meet-in-the-Middle attack**) involves the following steps:

1. Assume you have a few $\langle \text{plaintext}, \text{ciphertext} \rangle$ pairs $\langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \langle m_3, c_3 \rangle$ where c_i was derived from doubly encrypting m_i with K_1 and K_2 . You want to find K_1 and K_2 .
2. First make Table A with 2^{56} entries, where each entry consists of a DES key K and the result r_K of applying that key to encrypt m_1 . Sort the table in numerical order by r_K .
3. Now make Table B with 2^{56} entries, where each entry consists of a DES key K and the result s_K of applying that key to decrypt c_1 . Sort the table in numerical order by s_K .

-
4. Search through the sorted lists to find matching entries, $\langle K_A, t \rangle$ from table A and $\langle K_B, t \rangle$ from Table B. Each match provides K_A as a candidate K_1 and K_B as a candidate K_2 because K_A encrypts m_1 to t and K_B encrypts t to c_1 . So $K_A|K_B$ is a candidate double-length key that maps m_1 to c_1 .
 5. If there are multiple matching entries (pairs of keys $\langle K_A, K_B \rangle$) that map m_1 to c_1 (which there almost certainly will be), test whether each candidate pair of keys maps m_2 to c_2 . If you've tested all the candidate $\langle K_A, K_B \rangle$ pairs and multiple of them map m_1 to c_1 and also map m_2 to c_2 , then try a third $\langle \text{plaintext}, \text{ciphertext} \rangle$ pair $\langle m_3, c_3 \rangle$. The correct key pair $\langle K_1, K_2 \rangle$ will always work, of course, and an incorrect pair of keys will almost certainly fail to work on any other $\langle m_i, c_i \rangle$ pair.

How many matches should you expect to find after searching the two tables? Well, there are 2^{64} possible blocks and only 2^{56} table entries in each table (because there are only 2^{56} keys). Therefore, each 64-bit block has only a one in 256 chance of appearing in each of the tables. Of the 2^{56} blocks that appear in Table A, only 1/256 of them also appear in Table B. That means that there should be about 2^{48} entries that appear in both tables. One of those corresponds to the correct $\langle K_1, K_2 \rangle$ pair and the others are imposters. We'll test them against $\langle m_2, c_2 \rangle$. If $\langle K_A, K_B \rangle$ is an imposter, the probability that $D(c_2, K_B)$ will equal $E(m_2, K_A)$ is about $1/2^{64}$. There are about 2^{48} imposters, so the probability of one of them satisfying $D(c_2, K_B) = E(m_2, K_A)$ is about $2^{48}/2^{64}$, or roughly one in 2^{16} . Each test against an additional $\langle m_i, c_i \rangle$ reduces the probability by a factor of 2^{64} , so the probability that there will still be false matches after trying three $\langle m, c \rangle$ pairs is about $1/2^{80}$.

3.6.1.3 Triple Encryption with Only Two Keys

3DES does triple encryption. A once-popular variation of 3DES was to only use two keys and set $K_3 = K_1$. Here are two reasons for using only two keys with 3DES.

1. The attack on the 2DES variant described in §3.6.1.2 *Encrypting Twice with Two Keys* could be applied to 3DES with three keys and reduce the work of breaking 3DES to 2^{112} encryptions, and it is generally considered bad form to have a cryptographic system where there are more efficient attacks than brute force.
2. A property of an ideal cipher is that it is difficult to find a key that will map a given plaintext block to a given ciphertext block. With either double encryption or with 3DES with three keys, it is relatively easy to find a key that maps a given plaintext to a given ciphertext. Following the pattern of attack from section §3.6.1.2 *Encrypting Twice with Two Keys*, if you encrypted the plaintext with 2^{32} randomly chosen $\langle K_1, K_2 \rangle$ pairs and decrypted the ciphertext with 2^{32} randomly chosen values of K_3 , there would likely be at least one common value. That would give the attacker the needed three keys to encrypt the plaintext to the ciphertext. Most uses of a block encryption algorithm could not be exploited just because an attacker could find some key that mapped from a given plaintext to a given ciphertext. In any system

where the keys are bigger than the blocksize, there will be lots of keys with that property, but usually only the one correct key is useful. An example where such a threat would be important is the use of encryption in the Unix password hash (see §5.5 *Creating a Hash Using a Block Cipher*). In such a scheme, a password is used as a cryptographic key to encrypt a constant, and the result is stored. Using EDE with three keys, it is straightforward to find a triple of keys that maps a given plaintext to a given ciphertext, and the result would be accepted by the system as a valid password. It wouldn't have to be the actual user's password; any value that maps the plaintext (the constant) to the stored value would be accepted by the system.

There is no known practical way of finding such a triple with $K_1=K_3$.

Subsequent to these arguments, it was discovered that two-key 3DES did not provide 112 bits of security [MERK81]. So 3DES with two keys was generally abandoned in favor of 3DES with three keys despite all the theoretical problems with doing so. Due to the risk of using any block cipher with a 64-bit blocksize, AES has replaced DES and 3DES in almost all uses except, surprisingly, banking.

3.6.2 Why EDE Rather Than EEE?

Why is K_2 used in decrypt mode? Admittedly, it is no more trouble to run DES in either mode, and either way gives a mapping. DES would be just as good always done backwards (*i.e.*, swap *encrypt* and *decrypt*). One reason for the choice of EDE was to allow hardware to be built that could be used for both EDE encryption and ordinary DES encryption by setting $K_1=K_2=K_3$. The hardware does three times the work, but it gets the job done (see Homework Problem 7).

3.7 ADVANCED ENCRYPTION STANDARD (AES)

3.7.1 Origins of AES

In the 1990s, the world needed a new secret key standard. DES's key was too small. Triple DES (3DES) was too slow, and the 64-bit blocksize in DES and 3DES was also too small.

The National Institute of Standards and Technology (NIST) decided it wanted to facilitate creation of a new standard, but it had at least as difficult a public relations problem as a technical problem. After years of some branches of the U.S. government trying everything they could to hinder deployment of secure cryptography, there was likely to be strong skepticism if a branch of the government stepped forward and said *We're from the government, and we're here to help you develop and deploy strong crypto.*

NIST really did want to help create an excellent new security standard. The new standard should be efficient, flexible, secure, and unencumbered (free to implement). But how could it help create one? Staying out of the picture wouldn't help, since no similar organizations were volunteering. Proposing an NSA-designed cipher, designed in secret, wouldn't work since everyone would speculate that there were back doors. So on January 2, 1997, NIST announced a contest to select a new encryption standard. Proposals would be accepted from anyone, anywhere in the world. The candidate ciphers had to meet a bunch of requirements, including having a documented design rationale (and not just *Here's a bunch of transforms we do on the data*). Then there were several years in which conferences were held for presentation of papers analyzing the candidates. In addition to NIST, there was a group of highly motivated cryptographers (including the authors of competing entries) looking for flaws in the competing proposals, and comparing the candidate algorithms for characteristics such as performance.

After lots of analysis, NIST chose *Rijndael*, named after the two Belgian cryptographers who developed and submitted it—Joan Daemen and Vincent Rijmen [DAEM99]. On 26 November 2001, AES, a standardization of Rijndael, became a Federal Information Processing Standard [NIST01].

Rijndael has twenty-five variants because it provides for five different blocksizes and five different keysizes. These two parameters can be chosen independently from 128, 160, 192, 224, and 256 bits. AES has only three variants because it mandates a blocksize of 128 bits with a keysize of 128, 192, or 256 bits, but AES is otherwise identical to the submitted Rijndael proposal. We will only describe the AES variants, and not refer to Rijndael further.

3.7.2 Broad Overview

A common structure of an encryption algorithm is to process the input in rounds, where each round has two sorts of transformations—linear and nonlinear. A **linear transformation** is one where each of the outputs can be independently computed as a weighted sum of the inputs. (Or more formally, a transformation F is linear if it satisfies $F(a+b) = F(a)+F(b)$, with a and b members of some additive group. In most of the cases we will discuss, $+$ is the \oplus operation.) If the linear transformation is based on individual bits, then each output bit is the \oplus of some subset of the input bits. If the linear transformation is computed on octets (as in AES), then each output octet is a weighted sum of the input octets (with the computation done in a finite field so the output is the appropriate size). In DES, the linear transformation is just a bit shuffle, which is very easy to compute in hardware (just move wires around, and no gates are required). In AES, the linear function is more complex. In DES, the bit shuffle results in the same number of 1s in the output as in the input. In AES, because the linear function is not a simple bit shuffle, the input and output might have different numbers of 1s.

Typically, the nonlinear transformation (called an *S-box*) is done on a small subset of the bits. The input is partitioned into small groups, and each group is transformed with an S-box that translates each input value into an output value, ideally in as nonlinear a manner as possible. Why is the S-box implemented on a small set of bits? In DES, the eight S-boxes are defined as tables of input→output, each entry mapping a 6-bit input to a 4-bit output (so the table for each S-box in DES has 64 entries, each with a 4-bit output value). In AES, there is just one S-box, with 8-bit input and output. The AES S-box was originally envisioned to be implemented as a 256-octet lookup table, although rather than a randomly chosen mapping, there is a formula for computing the S-box output. The formula was provided to show there was no nefarious choice of mapping. Most likely, AES would be secure with most mappings, although the designers of the AES S-box made sure that their defined mapping met certain criteria (such as not mapping an input to itself or its complement, or having two inputs that map to each other), as well as having good quantitative properties for resistance to differential and linear cryptanalysis. Note that if a table-lookup S-box had 128 bits of input, the table would have to have 2^{128} entries. Even with a computed S-box, although there would be no huge lookup table, if the input were a large number of bits, it would take a lot of analysis to ensure that the chosen mapping didn't have any weak transformations (such as having an input *A* mapping to *B* and input *B* mapping to *A*).

Why not just make all the transformations linear? If you do a sequence of linear transformations, you will wind up with another linear transformation. If an encryption algorithm were a linear transformation, it would not be secure, because given a small number of $\langle \text{plaintext}, \text{ciphertext} \rangle$ pairs, you could solve linear equations to calculate the key.

Why not make all the transformations nonlinear? This actually could be secure, but it is not usually done. With small S-boxes, an input bit can only affect the other bits with which it is grouped, so it is necessary to have another step, after the S-boxes, to move the bits around. While it's possible that this other step (called the mixing layer) could also be nonlinear, designers usually prefer linear operations to provide mixing between the S-boxes, because it's easier to prove certain properties for a linear function than a nonlinear function. For a deeper analysis see [DAEM01].

There are various terms in the literature for the nonlinear transformation and the linear transformation. Sometimes the nonlinear layer is referred to as the **confusion layer**, and the linear layer is referred to as the **diffusion layer**. *Confusion* and *diffusion* are terms coined by Shannon [SHAN45] that refer to design goals of ciphers—confusion has to do with making sure the mathematical relationship between any two bits is complicated, while diffusion has to do with making sure each bit affects all the other bits. The implication of using this terminology is that the goal of the nonlinear layer is to provide confusion, while the goal of the linear layer is to provide diffusion. While these concepts don't necessarily correspond to the real design goals for the layers of a round in all cipher designs, or even those that can be neatly divided into linear and nonlinear layers, they are close enough to the design goals in most ciphers that the terminology is popular.

Another common terminology is a **substitution-permutation network (SPN)**. We think this terminology is incredibly confusing, because in AES (which is considered to be an SPN) both the

nonlinear (substitution) and the linear (permutation) transformations are one-to-one, and therefore according to math terminology, both are permutations of the input values. So we will not use the term *SPN*, and instead use the terms *nonlinear layer* and *linear layer*.

An encryption algorithm needs to be reversible (so that decryption is possible). DES manages to be reversible because it uses the Feistel construction, which allows the inverse of a DES round to be computed even though the mangler function is not invertible. That gave the DES designers a lot of flexibility in their choice of S-boxes but at the cost of only being able to scramble half the bits of state in each round. AES takes a different approach, in that each operation is reversible, which implies that all the operations are constrained to be one-to-one.

In AES, there actually are more input bits than output bits, because the input consists of the bits in a block plus the bits in a key, and the output consists of just a block. So, how does the key affect the computation? The key is expanded into round keys, and the round key is \oplus 'd into the state between rounds.

3.7.3 AES Overview

AES is a 128-bit block cipher with a choice of keysizes—128, 192, or 256 bits—with the resulting variants imaginatively called AES-128, AES-192, and AES-256.

AES is similar to DES in that there is a key expansion algorithm that takes the key and expands it into a bunch of round keys, and the algorithm executes a series of rounds that progressively mangle a plaintext block into a ciphertext block. (See Figure 3-4.) With DES, each round takes a 64-bit input and a 48-bit round key and produces a 64-bit output that (along with the next round key) is the input to the next round. With AES, each round takes a 128-bit input and a 128-bit round key and produces a 128-bit output that is fed into the next round. Unlike DES, AES is not a Feistel cipher. Instead, each of its steps is itself reversible. That means that AES can scramble all the bits on each round, and therefore only needs about half as many rounds to achieve the same level of security as it would need if it were a Feistel cipher. In accordance with the principle that you should only have as many rounds as are needed to make exhaustive search of the key space cheaper than any other form of cryptanalysis, AES has more rounds when the keys are bigger. AES-128 has ten rounds, AES-192 has twelve rounds, and AES-256 has fourteen rounds. If AES had a 64-bit version, it would have eight rounds, which would be comparable to the sixteen rounds in DES. (Again, because AES is not a Feistel cipher, each AES round is twice as effective as a DES round.)

Another difference between DES and AES is that DES operates on bits, whereas AES operates on octets. This makes AES software implementations more efficient.

Another difference is that the S-boxes and P-boxes in DES seem somewhat arbitrary and need to be specified with tables, whereas with AES the design gives a relatively simple mathematical formula for all the transformations, and the rationale is openly stated.

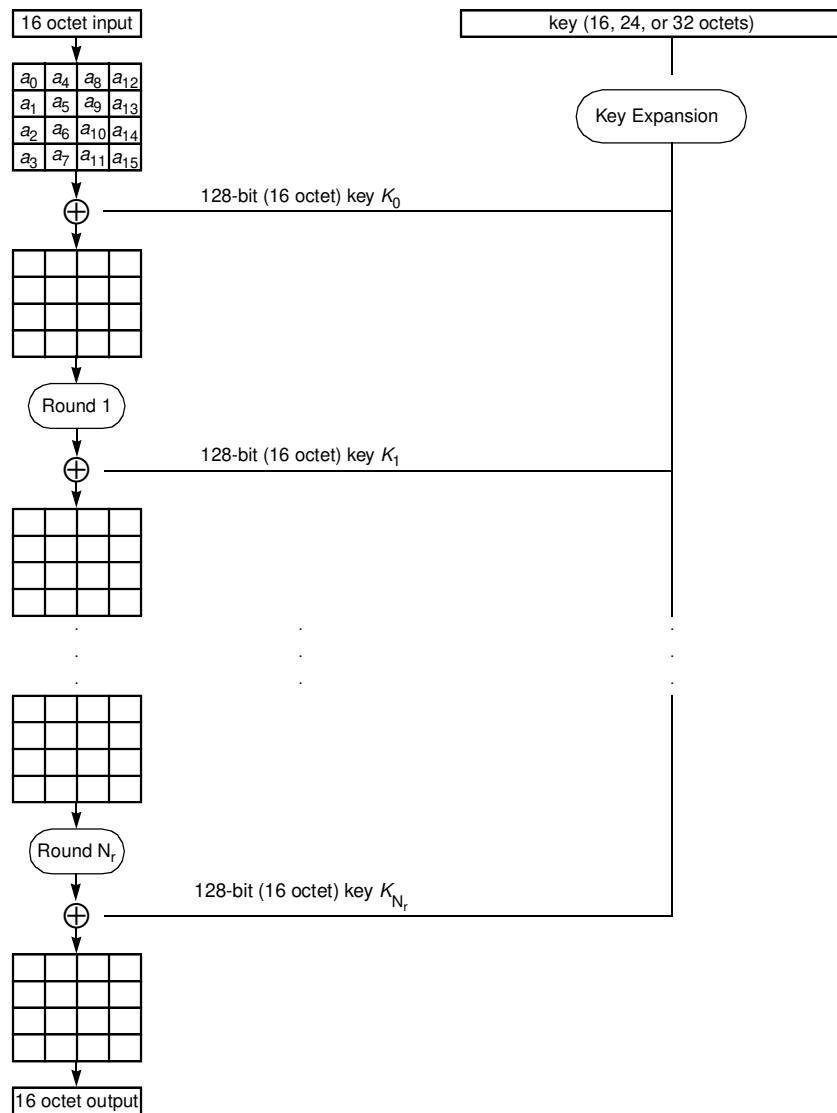


Figure 3-4. Basic Structure of AES

In AES, the key (128, 192, or 256 bits) is expanded into a series of 128-bit round keys, where there is one more round key than there are rounds. AES encryption or decryption consists of \oplus ing a round key into the intermediate value at the beginning of the process, at the end of the process, and between each pair of rounds.

Each round operates on a 128-bit block, which can be thought of as a 4×4 array of octets. We will refer to the block at the beginning of the round as the **input state** and at the end of the round as the **output state**.

The nonlinear operation in a round (AES's S-box) is known as **SubBytes**. It performs a one-to-one mapping of input octet value to output octet value for each of the sixteen octets in the state. Although that would allow for sixteen different S-boxes (one for each octet of input), all the AES S-boxes are identical.

There are two operations in the linear layer of AES:

- **ShiftRows.** Each of the state's four rows in the 4×4 array of octets is rotated left by a row-specific number of octets, with the octets falling off the left rotated into the right position. The top (first) row is rotated zero octets, the second row by one octet, the third row by two octets, and the fourth row by three octets.
- **MixColumns.** This acts on each column of four octets in the 4×4 array of octets independently. Each column is multiplied by the same, fixed, 4×4 matrix (using $\text{GF}(2^8)$ arithmetic). Because that matrix has an inverse, this step (as are all the steps) is reversible. When combined with ShiftRows, every octet in the 4×4 array affects every other octet within two rounds.

3.7.4 Key Expansion

Key expansion with DES is very simple. In DES, there is a need for sixteen round keys, each 48 bits long, and they are all generated by taking different subsets of the 56-bit key in different orders. Each key bit ends up being used in thirteen or fourteen of the sixteen round keys.

AES key expansion works differently. It has 128-bit round keys, while the full key can be 128, 192, or 256 bits. AES starts with the full key and generates additional keysize blocks (**key-blocks**) of key material. Each new keyblock is a complicated (but one-to-one) function of the previous keyblock, combining a number of simpler functions: applying the S-box to each octet in a 4-octet group, rotating the octets in a 4-octet group, \oplus ing a per-keyblock constant into one octet, and \oplus ing 4-octet groups together. Once sufficiently expanded, the round keys get taken from the keying material 128 bits at a time. You get one round key from each 128-bit keyblock, two round keys from each 256-bit keyblock, or three round keys from each pair of 192-bit keyblocks.

3.7.5 Inverse Rounds

Since each operation is invertible, decryption can be done by performing the inverse of each operation in the opposite order from that for encryption and using the round keys in reverse order.

Just as DES does an additional swap step at the end to enable a single implementation to do either encryption or decryption just by supplying a different expanded key, AES has some special features designed to make a combined implementation of encryption and decryption easier. In particular, in order to make the cipher and its inverse more similar in structure, the MixColumns step is omitted from the last round.

3.7.6 Software Implementations of AES

Unlike DES, AES was specifically designed to be efficient to implement in software. Since $\text{GF}(2^8)$ arithmetic (Chapter 2.7.1 *Finite Fields*) is not generally fast in software, and octet-oriented operations tend to be time consuming, AES was cleverly designed so that by building some large tables dependent on the key, each round can be implemented with sixteen 4-octet memory fetches, some rotates, and some \oplus s.

This is almost never done, however. All the major CPU vendors have implemented AES instructions that combine several steps and run substantially faster than any software implementation could. Furthermore, on any CPU where multiple processes are running, researchers discovered that any implementation that does table lookups where the addresses are dependent on secret data is highly vulnerable to side channel attacks [KOCH96, BERN05].

3.8 RC4

A long random (or pseudorandom) string used to encrypt a message with a simple \oplus operation is known as a **one-time pad**. A **stream cipher** generates a one-time pad and applies it to a stream of plaintext with \oplus .

RC4 is a stream cipher designed by Ron Rivest that for a time was the most widely used encryption algorithm in the world. RC4 was initially a trade secret but was “outed” in 1994 by an anonymous posting on the Cypherpunks mailing list. It was later incorporated into a number of standard protocols, including SSL and WEP. It is noteworthy because of its extreme simplicity and high performance.

A number of subtle flaws have been discovered. One issue is the problem of **related keys**. You must use a different key for each message or stream you want to encrypt. A design where you do this by concatenating a secret and a nonce will not be secure because it will produce two streams with similarities that enable cryptanalysis. Instead, you should compute the key as a hash of the secret and the nonce.

There have also been flaws discovered where the initial octets of RC4 output are not equally likely to have any of the 256 possible values. For example, the second octet has a probability of 1/128 rather than the expected 1/256 of being zero. This can be a problem if the same message is encrypted many times with different keys. This flaw was practical to exploit in WEP because the first few octets of plaintext sent by each node is a constant. That problem (that the initial output of RC4 is not uniformly distributed) can be avoided by skipping the first 4K (or so) octets of the stream before using the pad. Another problem is that RC4 makes key-dependent memory accesses, making it potentially vulnerable to side channel attacks. Because of these security issues, and the fact that most modern CPUs implement AES in specialized hardware that make it even faster than RC4, RC4 has fallen into disuse. It has been deprecated in most standards.

The RC4 algorithm is an extremely simple (and fast) generator of pseudorandom streams of octets. So simple that we can show you the code! The key can be from one to 256 octets long. Even with a minimal key (a single null octet), the generated pseudorandom stream passes all the usual randomness tests (and so makes a good enough pseudorandom number generator for a variety of non-cryptographic purposes—I₃'ve used it for that purpose on numerous occasions). RC4 keeps 258 octets of state information: 256 octets are a pseudorandom permutation of 0,1,...,255 (called **s** in the code below) and two indices into that array called **i** and **j**. The index **i** increments mod 256. The index **j** is pseudorandom and on each iteration is incremented by the octet indexed by **i**. Each iteration of RC4 swaps the octets pointed to by **i** and **j**, which maintains **s** as a permutation of 0,1,...,255 that changes after each iteration. To initialize RC4 from a key, **rc4init** iterates through 256 steps swapping octets and incrementing **j** by both a pseudorandom octet from the permutation and an octet from the key:

```
static uint8_t s[256], i, j;

void rc4init(uint8_t *key, int keylen) {
    i = j = 0;
    do s[i] = i; while (++i);
    do { j = (j + s[i] + key[i%keylen]);
        uint8_t t = s[i]; s[i] = s[j]; s[j] = t;
    } while (++i);
    j = 0;
}
```

Once the state is initialized, for each octet output from RC4, **i** and **j** are both updated, the two octets they point to are swapped, and then the sum of those two octets is used as an index into the permutation to decide which octet to return.

```
uint8_t rc4step() {
    j += s[++i];
    uint8_t t = s[i]; s[i] = s[j]; s[j] = t;
    return (s[t=s[i]+s[j]]);
}
```

3.9 HOMEWORK

1. Write a program that implements an oracle that encrypts and decrypts, using 8-bit blocks and 8-bit keys (as described in §3.2.3 *Looking Random*).
2. If both the keysize and the blocksize were 128 bits, for how many keys, on average, would the encryption of a block with given value x be a block with given value y ?
3. If the keysize is 256 bits and the blocksize is 128 bits, for how many keys, on average, would the encryption of a block with given value x be a block with given value y ?
4. Keeping in mind that a useful encryption algorithm requires the ability to decrypt, which of the following are possible?
 - a) An encryption algorithm for which there are two tuples, $\langle \text{key}_1, \text{block}_1 \rangle$ and $\langle \text{key}_2, \text{block}_2 \rangle$, that map to the same ciphertext block.
 - b) An encryption algorithm for which there are two tuples, $\langle \text{key}_1, \text{block}_1 \rangle$ and $\langle \text{key}_1, \text{block}_2 \rangle$, that map to the same ciphertext block.
 - c) An encryption algorithm that maps plaintext blocks of size k bits to ciphertext blocks of size j bits. Is this possible if $k < j$? How about if $k > j$?
 - d) An encryption algorithm that takes as input a triple $\langle \text{key}, \text{plaintext block of size } k, \text{random number } R \rangle$, and outputs $\langle R, \text{ciphertext block of size } k \rangle$ (where the ciphertext depends on all three inputs).
5. Is it possible for an encryption algorithm, when encrypting with key K , to map plaintext P to itself?
6. Show that if each round transformation in an encryption algorithm is linear, that the encryption algorithm itself will be linear.
7. Suppose one had a piece of hardware that did 3DES implemented using EDE. How could you use that hardware to implement DES?
8. Suppose in a Feistel cipher, the mangle function mapped every 32-bit value to zero, regardless of the value of its input. What function would encryption compute if there was only one round? What function would encryption compute if there were eight rounds?
9. Show that DES encryption and decryption are identical except for the order of the 48-bit keys. Hint: Running a round backwards is the same as running it forwards but with the halves swapped, and DES has a swap after round 16 when run forwards (see §3.5.1 *DES Overview*).

4

MODES OF OPERATION

4.1 INTRODUCTION

We've covered how to encrypt a 128-bit block with AES, or a 64-bit block with DES, and these primitives have the nice property that if an attacker changes any part of the ciphertext, the result of a decryption will be effectively a random number. Sadly, most useful messages are longer than 128 bits. **Modes of operation** are techniques for encrypting arbitrary-sized messages using the block encryption algorithms as primitives to be applied iteratively. There are additional desirable properties that such algorithms can have. If we send the same message multiple times, it would be desirable to have the encrypted message be different each time so that an eavesdropper can't tell that we're sending the same message repeatedly. If an attacker modifies an encrypted message, we would like to be able to detect that it is no longer a valid encrypted message by computing some sort of message authentication code (MAC). If an attacker can cause the system to encrypt a message of his choosing (a **chosen-plaintext attack**), this should not give the attacker the ability to verify guesses of any other data that was sent. This chapter will tie up these various loose ends.

We will discuss three kinds of modes:

- those that encrypt a message of any size, which we describe in §4.2
- those that compute a MAC of message of any size, which we describe in §4.3
- those that do both, which we describe in §4.4

There are a surprising number of modes of operation defined of each type. Some of them are used for historical reasons (when better algorithms are defined, it takes a very long time for the world to convert to them), while others exist because they solve the challenges of some particular scenario better than the more commonly used ones. We will present the most commonly used ones along with the scenarios they were designed for.

The purpose of this chapter is to give intuition into the algorithms, with the rationale for the design and potential pitfalls. For exact formats, refer to the standards.

Many of these modes use the bitwise-exclusive-or (XOR) operation that we will denote with the symbol \oplus . We debated whether, when referring to the bitwise-exclusive-or operation in text, we

should use XOR or \oplus and decided to use \oplus because “XOR” looked too similar visually to some of the mode names such as XEX and XTS. We will continue using the symbol \oplus throughout the book.

4.2 ENCRYPTING A LARGE MESSAGE

Let's assume we're using a block cipher with 128-bit blocks. How do you encrypt a message larger than 128 bits? There are several schemes defined in the NIST standard documents (the NIST SP 800-38 series). These schemes are applicable to any secret key block cipher that encrypts fixed-length blocks.

4.2.1 ECB (Electronic Code Book)

This mode consists of doing the obvious thing, and it is usually the worst method. You break the message into 128-bit blocks (padding the last one out to a full 128 bits), and encrypt each block with the secret key (see Figure 4-1). The other side receives the encrypted blocks and decrypts each block in turn to recover the original message (see Figure 4-2).

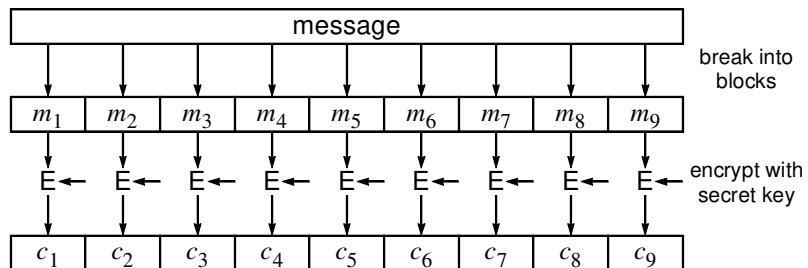


Figure 4-1. Electronic Code Book Encryption

There are a number of problems with this approach. First, if a message contains two identical blocks, the corresponding two blocks of ciphertext will be identical. This will give an eavesdropper some information. Whether it is useful or not depends on the context. We'll give an example where ECB would have a problem. Suppose that the eavesdropper knows that the plaintext is an alphabetically sorted list of employees and salaries, tabularly arranged (see Figure 4-3).

To create an example that fits nicely on a page, assume a cipher like DES or 3DES with 8-octet blocks. Further suppose that, as luck would have it, each line is exactly 64 octets long, and the blocks happen to be divided in the salary field between the thousands' and the ten-thousands'

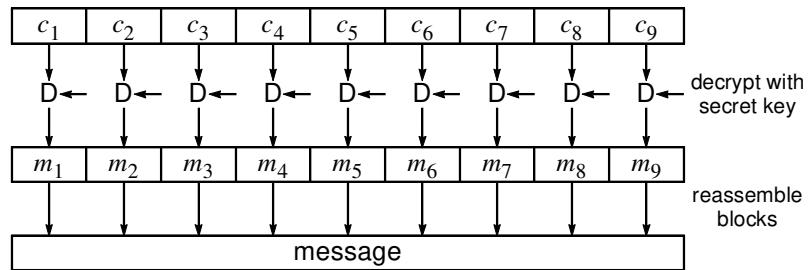


Figure 4-2. Electronic Code Book Decryption

digits. Since identical plaintext blocks produce identical ciphertext blocks, if you could see the ciphertext, you could tell which employees have identical salaries. You could also tell which employees have salaries in the same \$10,000 ranges. Furthermore, if you were one of the employees, you could change your salary in the database to match that of any other employee by copying the ciphertext blocks from that employee to the corresponding blocks of your own entry.

Figure 4-3. Payroll Data

So, ECB has two serious flaws. Patterns in the ciphertext, such as repeated blocks, leak information, and nothing prevents someone from rearranging, deleting, modifying, or duplicating blocks.

4.2.2 CBC (Cipher Block Chaining)

CBC mode avoids some of the problems in ECB. Using CBC, even if the same block repeats in the plaintext, it will not cause repeats in the ciphertext. CBC mode is still commonly used even though modes discussed later in this chapter are technically superior.

4.2.2.1 Randomized ECB

First we'll give an example of how to avoid some of the issues with ECB with an example mode (that we'll call *randomized ECB*) that is inefficient but helps for understanding CBC. Note that we are only introducing this mode (randomized ECB) as a way of explaining CBC mode in the next section. Randomized ECB is not a mode that is used or standardized.

Assuming a 128-bit blocksize, generate a 128-bit random number r_i for each plaintext block m_i to be encrypted. \oplus the plaintext block with the random number, encrypt the result, and transmit both the unencrypted random number r_i and the ciphertext block c_i (see Figure 4-4). To decrypt this, you'd decrypt each ciphertext block c_i , and then \oplus the result with the random number r_i .

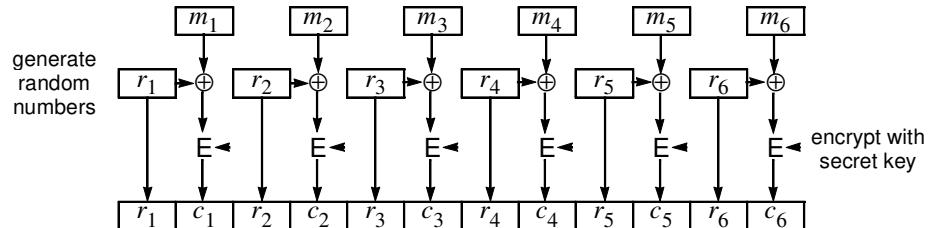


Figure 4-4. Randomized Electronic Code Book Encryption

The main problem with this scheme is efficiency. It causes twice as much information to be transmitted, since a random number has to be transmitted along with each block of ciphertext. Another problem with it is that an attacker can still rearrange the blocks and have a predictable effect on the resulting plaintext. For instance, if $r_2|c_2$ were removed entirely, it would result in m_2 being absent in the decrypted plaintext. If $r_2|c_2$ were swapped with $r_7|c_7$, then m_2 and m_7 would be swapped in the result. Worse yet, an attacker knowing the value of any block m_n can change it in a predictable way by making the corresponding change in r_n . (See Homework Problem 4.)

4.2.2.2 CBC

Now we can explain CBC. CBC generates its own “random numbers” for all but the first block. It uses c_i as r_{i+1} . In other words, it takes the previous block of ciphertext and uses that as the “random number” that will be \oplus 'd into the next plaintext block. To avoid leaking the information that two messages encrypted with the same key have the same first plaintext blocks, CBC selects one

random number that gets \oplus 'd into the first block of plaintext and transmits it along with the data. This initial random number is known as an **IV (initialization vector)**. (See Figure 4-5.)

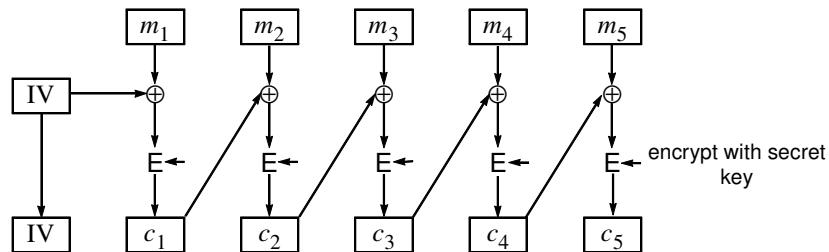


Figure 4-5. Cipher Block Chaining Encryption

Decryption is simple because \oplus is its own inverse (see Figure 4-6).

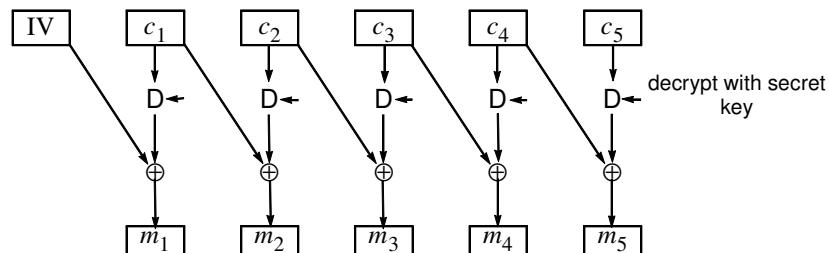


Figure 4-6. Cipher Block Chaining Decryption

CBC encryption requires essentially the same amount of computation as ECB encryption, since the cost of the \oplus is trivial compared to the cost of an encryption. However, CBC has an important performance disadvantage: Since encrypting each block of plaintext requires knowing the ciphertext value for the previous block, CBC encryption cannot take advantage of the parallel processing capabilities of most modern CPUs to encrypt multiple blocks at once. The cost of generating and transmitting the IV can also be considered a performance disadvantage.

Some older implementations of CBC mode omitted the IV (or equivalently, used the value 0 as the IV). There are cases where this would not adversely affect security: *e.g.*, where each message is encrypted with a different key or where the first block of each of the messages encrypted with the same key contains a sequence number. However, there are enough cases where omitting the IV would cause problems that the NIST specification disallows it. For example, suppose the encrypted file of employees and salaries is transmitted weekly. If there were no IV, then an eavesdropper could tell where the ciphertext first differed from the previous week and perhaps determine the first person whose salary had changed.

Another example is where a general sends information each day saying *continue holding your position*. Without the IV, the ciphertext will be the same every day until the general decides to send something else, like *start bombing*. Then the ciphertext would suddenly change, alerting the enemy.

A randomly chosen IV guarantees that even if the same message is sent repeatedly, the ciphertext will be completely different each time. There remain some other threats, however, that can and have caused security problems. We'll explain those in the next few sections.

4.2.2.3 CBC Threat—Modifying Ciphertext Blocks

Using CBC (rather than ECB) does not eliminate the problem of someone creating desired plaintext by modifying the ciphertext. It just changes the nature of the threat. Attackers can no longer see repeated values and simply copy or move ciphertext blocks in order to, say, swap their salary with the salary of the VP of marketing. But there are still possible attacks. What would happen if they changed a block of the ciphertext, say, the value of ciphertext block c_n ? Changing c_n has a predictable effect on m_{n+1} because c_n gets \oplus 'd with the decrypted c_{n+1} to yield m_{n+1} . For instance, changing bit 3 of c_n changes bit 3 of m_{n+1} . Modifying c_n also garbles block m_n to some unpredictable value.

For example, let's say our attacker Ann knows that the plaintext corresponding to a certain octet range in the ciphertext is her personnel record. And to make the diagram simpler, let's assume 64-bit blocks and that the salary is expressed as ASCII characters:

Tacker, Ann A.	System Security Officer	44,122.10
m1 m2 m3 m4 m5 m6 m7 m8		

Let's say Ann wants to increase her salary by 10K. In this case she knows that the final octet of m_7 is the ten-thousands digit of her salary. The bottom three bits of the ASCII for 4 is 100. To give herself a raise of 10K, she merely has to flip the last bit of c_6 . Since c_6 gets \oplus 'd into the decrypted c_7 (c_7 has not been modified, so the decrypted c_7 will be the same), the result will be the same as before, *i.e.*, the old m_7 , but with the last bit flipped, which changes the ASCII 4 into a 5.

Unfortunately for Ann, as a side-effect of this change, a value she will not be able to predict will appear in her job-title field (since she cannot predict what the modified c_6 will decrypt to), which will affect m_6 :

Tacker, Ann A.	System Security Offi!z°E(%9™	54,122.10
m1 m2 m3 m4 m5 m6 m7 m8		

A human who reads this report and issues checks is likely to suspect something is wrong. However, if it's just a program, it would be perfectly happy with it. And a bank would most likely take the check even if some unorthodox information appeared in what to them is a comment field.

In the above example, Ann made a change she could control in one block at the expense of getting a value she could neither control nor predict in the preceding block.

An encryption mode that protects both the confidentiality and the authenticity of a message with no greater cost than just encrypting the data was the holy grail of cryptographic protocol design for many years, with many proposals subsequently discredited because security flaws were found. In 2007, GCM (see §4.4.2) was formally standardized as a means to accomplish confidentiality and integrity [NIST07]. It is somewhat more expensive than encryption alone because of the arithmetic operations it does with the data, but in software these are much less expensive than encryption. Most modern CPUs implement encryption in hardware, making this a much less important concern.

4.2.3 CTR (Counter Mode)

Counter mode uses the key and the IV to generate a pseudorandom bit sequence that is then \oplus 'd with the data. Such a sequence is known as a one-time pad. CTR creates the first block of the one-time pad by encrypting an IV. For block n of the one-time pad, CTR encrypts $IV+n-1$. (See Figure 4-7.)

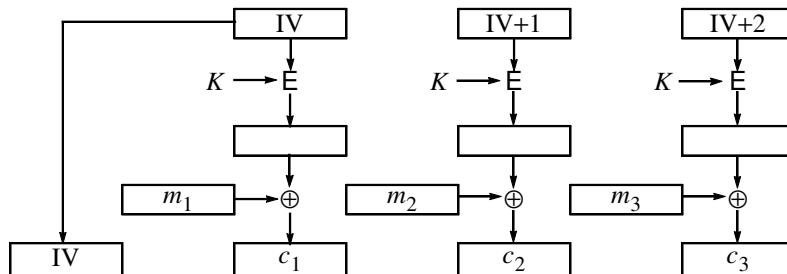


Figure 4-7. Counter Mode (CTR)

An advantage of counter mode is that if multiple CPUs (or multiple hardware implementations of the block cipher) are being used, different parts of the one-time pad can be computed in parallel. With CBC, blocks must be encrypted in sequence. With CTR, the one-time pad can be pre-computed, and encryption is simply an \oplus .

CTR is insecure if different data is encrypted with the same key and IV (because then a one-time pad would be used multiple times). Even if a different key and IV are used each time, an attacker can also change the plaintext in predictable ways (if there is no additional MAC), merely by \oplus ing the ciphertext with the bits the attacker wishes to change.

4.2.3.1 Choosing IVs for CTR Mode

As noted in the previous section, it is crucial that different plaintexts are never encrypted using the same encrypted counters. Choosing unique IVs is somewhat challenging in CTR mode because an IV value must be different for each block rather than just for each message. So if a hundred-block message is encrypted with an IV of X , it must be assured that no future messages choose an IV in the range X through $X+99$. The CTR standard does not specify how to assure uniqueness. It gives several examples, one being that the initial IV is chosen randomly when the key is chosen, and the IV is then incremented for each encrypted block. The other modes that use CTR (CCM [§4.4.1] and GCM [§4.4.2]) are a bit more explicit. For example, GCM specifies that the 128-bit IV is partitioned into the most significant 96 bits, which must be unique for each message encrypted with the same key (which we'll call the per-message portion), and the remaining 32 bits count blocks within a message. That design limits the length of a single GCM-encrypted message to 2^{32} blocks, *i.e.*, about 69 gigabytes, assuming 128-bit blocks. That is much longer than the maximum message length in most protocols. The CCM standard is similar, but it allows an implementation to choose how many bits to use for the per-message portion and how many to use for the block counter.

How do you assure that the per-message portion is indeed never used twice? One approach is to select it at random. This is nice in some ways. For example, it is stateless, so if one of the communicating parties reboots and forgets what its last IV was, it is no more likely to repeat an IV than it would otherwise be. The downside is, at least with a large number of messages (*i.e.*, a few trillion), there's a non-negligible chance of repeating a 96-bit per-message value. The NIST standard specifies that if you choose the per-message portion of the IV at random, you need to change keys before you encrypt more than 2^{32} messages.

Another approach is to start the per-message portion at a random value (when the key is changed) and increment it for each new message. This guarantees that the IV will not repeat until the counters wrap around. However, this approach introduces the burden of keeping track of where you are in the sequence, and it's likely that implementations will have flaws where IVs get accidentally reused because of losing counter state.

In some applications, encryption is done in parallel, using the same key, by multiple processors that cannot stay tightly synchronized with each other. A trick to avoid reusing per-message values in this scenario is to partition the per-message field into two fields—one that each of the parallel processors assigns in a way that avoids conflicting with values it has used with that key (*e.g.*, starting at a random value and incrementing for each message) and the other a unique constant assigned to each of the parallel processors.

4.2.4 XEX (XOR Encrypt XOR)

XEX mode was designed for disk encryption, where the data is encrypted at a low level and the disk layout is fixed. As a result, the XEX design had the constraint that the ciphertext could not be bigger than the plaintext. All the previously described modes except ECB use a randomly generated IV for each independently encrypted message, which causes the ciphertext to be longer than the plaintext. XEX does use an IV, but it is implicit and predictable—the disk sector number. XEX was invented by Rogaway [ROGA04]. It's not a standard, but we explain it here because it makes it easier to understand XTS (which is a standard and described in the next section). (See Figure 4-8.)

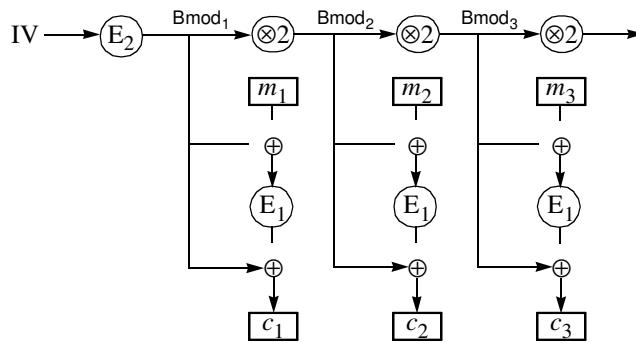


Figure 4-8. XEX Mode

XEX mode acts on 128-bit blocks and encrypts each block independently from all the others. This mode is well-suited for applications that read or write individual portions of a data structure. Achieving length preservation requires sacrificing integrity protection. Unlike CTR mode, XEX and its XTS variant (see §4.2.5) are designed so that even attackers that know the corresponding plaintext of an encrypted block cannot change it to a value of their choosing. (See Homework Problem 6.) Any change to the ciphertext will result in an effectively random value in the corresponding plaintext blocks when the data is decrypted. On the other hand, unlike with CBC mode, only the blocks where ciphertext was modified are affected.

The basic idea of XEX mode is to have a block-specific value that is \oplus 'd into each data block before and after the block is encrypted. We'll refer to the block-specific value as the **block modifier (Bmod)**. The Bmod is a function of the block's address and a secret key. The desired properties of a function that computes the Bmod is that it be inexpensive to compute (in particular, more efficient than an encryption operation) and that the Bmod value of each block looks like a random number to anyone who does not know the secret key.

XEX and XTS modes can be thought of as having two secret keys—one that encrypts the IV and one that encrypts each block of data. In Figures 4-8, 4-9, and 4-10, E_1 indicates encryption with the first key, while E_2 indicates encryption with the second. The original published version of XEX uses the same value for both keys, while the standardized version of XTS uses double-length keys

(256 bits for AES-128, 384 bits for AES-192, or 512 bits for AES-256), which specify independent values for the two keys.

XEX mode assumes there is an implicit unique IV for each message. If this is used for disk encryption, the location on disk where the message starts (*i.e.*, its disk sector number) is a reasonable implicit IV. (Note that in the literature, this implicit IV is referred to as a *tweak*.)

The block modifier B_{mod} is computed as a function of the secret key, the IV, and the block number within the message. For those readers who are not completely comfortable with finite field arithmetic, the important point is that although $B_{\text{mod},1}$ requires an encryption operation to compute, any other $B_{\text{mod},j}$ can be computed in software more efficiently than an encryption operation, with an operation that is a function of $B_{\text{mod},1}$ and j .

For those readers comfortable with finite field arithmetic, in the Rogaway paper, $B_{\text{mod},1}$ (the B_{mod} of block 1, the first block of the message) consists of the IV encrypted with the key. $B_{\text{mod},j+1}$ (the B_{mod} of block $j+1$ of the message) is $B_{\text{mod},j} \otimes 2$, where $\otimes 2$ means multiplication by 000...010 in the finite field $\text{GF}(2^{128})$, *i.e.*, multiplication by x mod the primitive polynomial $x^{128} + x^7 + x^2 + x + 1$. It is important that the modulus is primitive so that x has maximal order, meaning that $B_{\text{mod},j}$ won't repeat before taking on $2^{128}-1$ different values.

There is an important security weakness in this approach. The IV is supposed to be unique for each encrypted message, but by choosing it based on the disk sector number, it will be repeated whenever a new message overwrites an existing one on the disk. This allows someone who can observe the ciphertext on the disk on multiple days to figure out which blocks were modified. If attackers could also update the disk, they can revert any blocks to previous values. This weakness is accepted in order to achieve the goal of not expanding the plaintext during encryption.

4.2.5 XTS (XEX with Ciphertext Stealing)

This mode is officially called XTS-AES, but it could be used with any encryption algorithm, so we'll just call it XTS. XEX requires the size of a message to be encrypted to be a multiple of the cryptographic blocksize. XTS is a clever variant of XEX that encrypts multiblock messages that need not be a multiple of the cryptographic blocksize while still keeping the ciphertext the same size as the plaintext. XTS is standardized in [IEEE07 and NIST10]. XTS accomplishes this goal (of being length-preserving despite a message not being a multiple of the cryptographic blocksize) through a very clever but somewhat complicated trick known as **ciphertext stealing**. Note that only the final block of the message (let's call that m_n) might be undersized, so as we'll see, all blocks other than the last two can be encrypted normally. (See Figure 4-9 and Figure 4-10.)

What can you do with the final block, m_n , if, for example, it is 93 bits instead of the blocksize of 128 bits? You could pad m_n to be 128 bits, but then the ciphertext would be longer than the plaintext, and furthermore, how would you know which of the bits of the decrypted block m_n were padding?

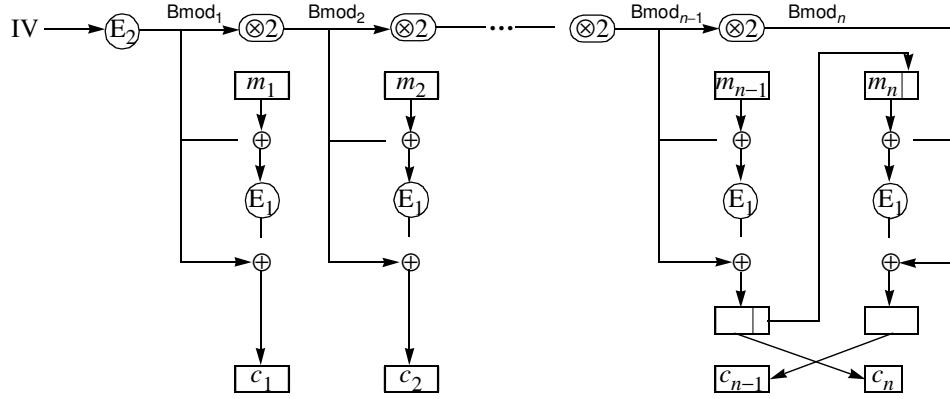


Figure 4-9. XTS Mode Encryption

Ciphertext stealing pads the final block m_n with as many of the bits of the previous block of ciphertext (c_{n-1}) as necessary to make block m_n be full sized (in our example, $128 - 93 = 35$ bits need to be stolen from c_{n-1}). The padded block m_n is then encrypted.

But the ciphertext is now longer than the plaintext, since the last block of ciphertext is now a full-sized block. To solve that problem, we swap ciphertext blocks c_n and c_{n-1} and truncate the final ciphertext block (which was the encryption of block m_{n-1}) to be the size of the original block m_n . In our example, where the original block m_n was 93 bits long, the final ciphertext block will be 93 bits. Now the ciphertext is the same size as the message, but how do we decrypt either of the last two blocks? To obtain block m_n , decrypt c_{n-1} (using implicit IVs and Bmods associated with block m_n). The result will be plaintext m_n with appended padding, but you need to know how much of the result is message and how much is padding. The size of plaintext block m_n will be the size of the

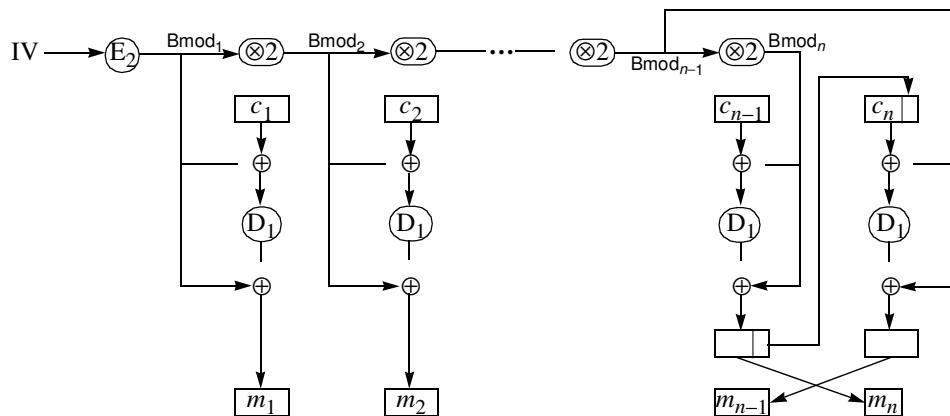


Figure 4-10. XTS Mode Decryption

final ciphertext block (in our example, 93 bits). The padding (the remaining $128 - 93 = 35$ bits) is stolen ciphertext from c_{n-1} .

Suppose you only needed to read plaintext block $n-1$. You would need to do two decryptions. First, do what's in the previous paragraph to obtain the padding (the stolen ciphertext) used to pad block m_n . Then, take the final ciphertext block c_n , append the stolen ciphertext to it, and decrypt the now-complete ciphertext block as if it were the encryption of m_{n-1} .

4.3 GENERATING MACS

A secret key system can be used to generate a cryptographic integrity check known as a MAC (Message Authentication Code). While the preceding modes, when properly used, offer good protection against an eavesdropper deciphering a message, none offers good protection against an attacker modifying the message without being detected. It is, therefore, good practice whenever encrypting data to include a MAC with the data. For some applications encryption is unnecessary, but MACs are used to protect the data from modification in transit. Some MACs in common use are based on secret key encryption functions, some on hashes, and some on public key algorithms. MACs using public key algorithms are known as digital signatures. In this chapter, we'll focus on MACs based on secret key encryption functions. MACs using a secret key cannot be computed or verified without knowledge of the secret key. Note that one of the most commonly used MAC schemes is known as HMAC. It has the same properties as the schemes in this chapter, but because it is based on hashes rather than on secret key encryption functions, it is discussed in Chapter 5 *Cryptographic Hashes* (§5.8 *The Internal Encryption Algorithms*).

4.3.1 CBC-MAC

CBC-MAC computes a MAC of a message using key K . It encrypts the message in CBC mode, using key K , and uses the last block (called the **residue**, see Figure 4-11) as the MAC for the message. If an attacker modifies any portion of a message, the residue will no longer be the correct value (except with probability 1 in 2^{128} , assuming 128-bit blocks).

4.3.1.1 CBC Forgery Attack

There are some weaknesses in using CBC residues with the same key to protect many messages of varying lengths. Having seen CBC residues on some messages, an attacker might be able to predict the CBC residues computed over other messages. For example, suppose someone sent a message

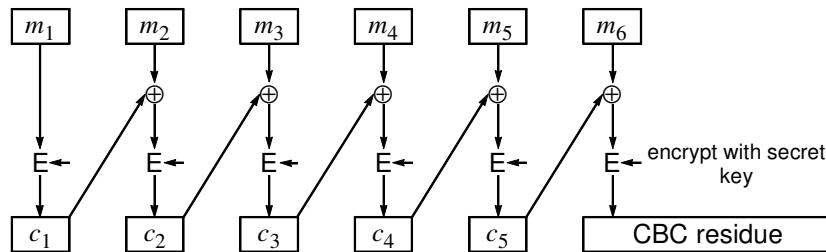


Figure 4-11. Cipher Block Chaining Residue

$m_1|m_2|m_3$ with CBC residue c_3 and later sent a longer message $m_1|m_2|m_3|m_4|m_5|m_6$ with CBC residue c_6 . An attacker could then forge a message $(c_3 \oplus m_4)|m_5|m_6$ and know its CBC residue would also be c_6 . See Homework Problem 12.

4.3.2 CMAC

To avoid the above problem with CBC-MAC, a slight variant called CMAC was defined and standardized in [NIST93]. It is very similar to CBC-MAC except that it \oplus s a secret value (derived from the key K) into the final block of plaintext before encrypting (see Homework Problem 9).

There are actually two secret values (X_1 and X_2) derived from the key K . The reason for deriving two secrets (X_1 and X_2) is to be able to pad a message to an integral number of blocks without adding an additional block when the message does not require padding. Padding a message to be an integral number of blocks is always an annoying detail. Since it's necessary to know what portion of the plaintext is padding, it can be tricky to distinguish a message that is padded from one whose plaintext happens to look like the padding format. The usual solution is to require every message to contain padding, so plaintext that is already an integral number of blocks will wind up being padded with an extra block of padding.

CMAC deals with the padding issue in a different way. If the message for which the MAC is being computed is not a multiple of the blocksize, it is padded with a 1 bit followed by as many 0 bits as are required to fill out the block. But, you wonder, how can you tell the difference between a padded message and an unpadded message, given that any block (other than all zeros) will have a final 1 somewhere? CMAC's solution is to \oplus the value X_1 into the final block if that final block was not padded and to \oplus a different value (X_2) into the final block if that final block was padded. The MAC computed by CMAC is based on the plaintext of the first $n-1$ blocks and the (possibly) padded n th block \oplus 'd with either X_1 or X_2 (depending on whether the last block was padded). Then the actual message (*i.e.*, with the padding bits removed), along with the MAC, is transmitted (or stored).

The special values X_1 and X_2 are derived from the key by a means that does not make a lot of intuitive sense unless you understand finite field arithmetic. K is used to encrypt a constant block containing all zeros. The result is \otimes' d by 2 to get X_1 . X_1 is then \otimes' d by 2 to get X_2 .

4.3.3 GMAC

While MACs based on CBC residues and hash algorithms have been around for years, GMAC works in a fundamentally different way and is typically faster to compute, although the performance gap between GMAC and CBC-based MACs using AES has narrowed significantly since the introduction of hardware-supported CPU instructions for AES on most platforms. Note that the G in GMAC stands for Galois, because these modes use Galois field arithmetic (see §2.7.1 *Finite Fields*). The older MAC functions require a cryptographic pass over the data—hash algorithms and encryption algorithms are of similar complexity and performance. GMAC is more like a checksumming algorithm, doing inexpensive calculations based on the contents of the message. GMAC only needs to do the expensive encrypt operation on two single-block values regardless of the length of the message. The world is slowly migrating to this new kind of MAC algorithm. Its only downside is that, unlike earlier MAC algorithms, it requires that a unique IV be used for each message until the key is changed. If the same IV and key are ever used with two different messages, an attacker who sees the results can then compute a MAC with that key for any message. We won't explain how reusing an IV in GMAC allows this, but if you take our word for it, contrast that consequence with the consequence of reusing an IV in CBC mode (see Homework Problem 7).

4.3.3.1 GHASH

GHASH (Figure 4-12) is a building block for computing a MAC. GHASH is used by both GMAC and GCM. It is much less expensive to compute than CBC residue, because the expensive encryption operation in CBC residue is replaced with an inexpensive finite field multiplication (\otimes). The inputs to GHASH are a message and a value H . The flow of GHASH is similar to CBC residue, in that the computed output of block m_i is \oplus' d with m_{i+1} , and then (where CBC residue would encrypt with key K) GHASH does $\otimes H$.

GHASH itself is not usable as a MAC, because anyone who knows the message and H can compute the GHASH value of that message. Furthermore, even if someone does not know H but sees a message and the corresponding GHASH value, they can derive H . However, GHASH will be used as a component in GMAC (§4.3.3), and it will be secure because GMAC will hide H and the GHASH value.

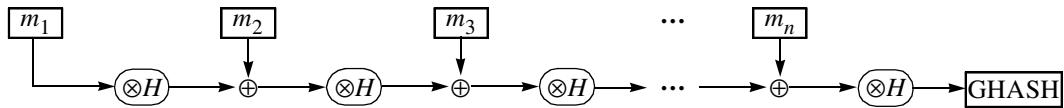


Figure 4-12. GHASH

4.3.3.2 Transforming GHASH into GMAC

GMAC (Figure 4-13) transforms GHASH into a cryptographically secure but still inexpensive MAC. It has been proven that if AES is secure as an encryption algorithm, GMAC will be secure as a MAC. GMAC takes as input a message M , a 96-bit IV that should never be used on two different messages, and a key K . It will output a 128-bit MAC.

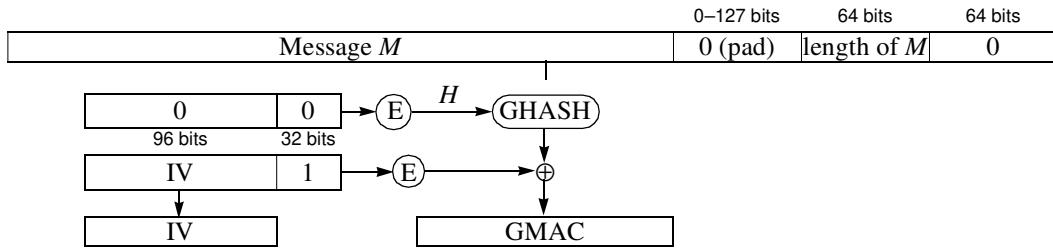


Figure 4-13. GMAC

GMAC starts by padding the message to be an integral number of 128-bit blocks. Then, GMAC gives the padded message and a value H to GHASH (§4.3.3.1). The value H is computed by GMAC as the AES encryption of the value 0 by the key K .

Next, GMAC takes the 128-bit value consisting of the 96-bit IV concatenated with the value 1 expressed in 32 bits, *i.e.*, 31 bits of 0 followed by a 1, encrypts that 128-bit value with K , and \oplus s that with GHASH. The result is GMAC. Note that neither H nor the GHASH value is exposed. Since any given IV will only be used once, an eavesdropper will get no information about GHASH nor know whether two messages have the same GHASH. H could have been specified to be a second key as input to the algorithm, but to avoid that, H is defined to be the encryption of the constant 0, and the constant 1 is appended to the IV to prevent the result from being 0.

Other than using two blocks requiring AES encryption, one to compute H and one to encrypt the 128-bit value containing the IV, all the other computations in GMAC (\otimes and \oplus) are inexpensive. Since H will be the same for any message with the same K , an implementation could save H , and on subsequent messages, only do a single AES-encrypt operation (to encrypt the 128-bit value containing the IV). The security of the scheme depends on the same IV never being used with two

different messages processed with the same key. Techniques for accomplishing that are described in §4.2.3.1 *Choosing IVs for CTR Mode*.

4.4 ENSURING PRIVACY AND INTEGRITY TOGETHER

Commonly, you will want to ensure both the privacy and the integrity of a transmitted message. You can do that by applying any of the MAC algorithms and any of the encryption algorithms to the message (and in either order). But that assumes that the two keys are independent of one another, and that you are willing to do twice the work.

There are two standardized combined modes (CCM and GCM) that manage to use a single key for encryption and integrity protection but ensure that doing so does not weaken either. CCM requires twice the cryptographic work of encryption or integrity protection alone, while GCM is typically less expensive because it uses GMAC as the MAC, and GMAC is typically less expensive to compute than an encryption of a message.

4.4.1 CCM (Counter with CBC-MAC)

CCM mode, standardized in NIST SP 800-38C, combines CTR mode encryption and CBC-MAC (CBC residue) integrity protection. CCM allows part of the message (the **associated data**) to be integrity protected but not encrypted. The application specifies which part of the message is associated data and which part of the message is to be both encrypted and integrity protected.

CCM mode uses CBC-MAC and not CMAC. CCM is still designed to resist the CBC forgery attack (see §4.3.1.1), but it does so in a different way than CMAC. The CBC standard specifies that the MAC is computed and appended to the plaintext before the combined quantity is encrypted. The fact that the MAC is encrypted prevents the CBC forgery attack. Although CCM uses the same key for integrity and encryption, encryption and integrity protection using CCM requires twice the cryptographic work as doing encryption or integrity protection alone. The MAC is computed over not just the message but also over an additional prefix block. That prefix block contains the nonce (from which the first counter value is derived) and plaintext length for CTR mode encryption. The fact that the plaintext length needs to be known before the CBC-MAC operation starts is a performance drawback for CCM.

4.4.2 GCM (Galois/Counter Mode)

GCM (Figure 4-14) combines counter mode encryption (see §4.2.3) with GMAC (see §4.3.3) in a way that allows both modes to use the same cryptographic key.

In CTR mode, an IV is incremented for each block. The IV for the CTR encryption has the same format as the 128-bit value in GMAC (see Figure 4-13) except that the low 32 bits (which in GMAC are always 0…01) start at 2 (0…010) and increment for each block to be encrypted.

As with CCM mode, GCM mode has the ability to have part of the message, called **associated data**, be integrity protected but not encrypted. The plaintext to be encrypted is CTR encrypted to produce the ciphertext. The MAC computation in GCM covers the associated data, the ciphertext, and an extra block containing the 64-bit length of the encrypted message and the 64-bit length of the associated data, to prevent the kind of attacks mentioned in §4.3.1.1 *CBC Forgery Attack*.

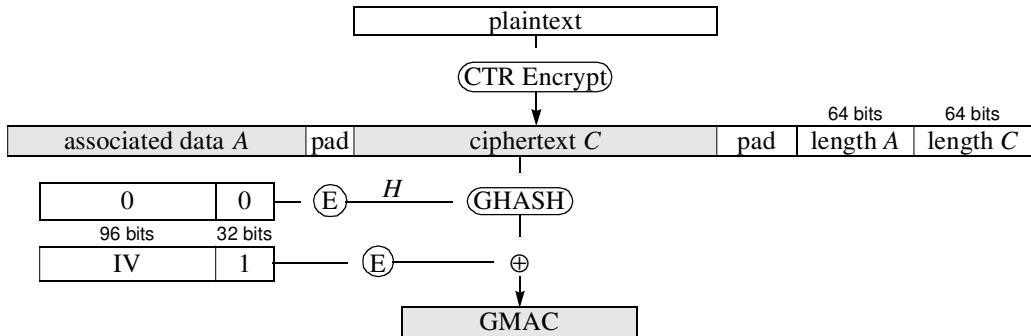


Figure 4-14. Galois/Counter Mode

Note in Figure 4-14, the boxes that are shaded (associated data A , ciphertext C , and GMAC) all need to be transmitted. Other items, such as the IV and the lengths of A and C , have to be known by the application or communicated somehow.

GCM also has another interesting property. As in all versions of counter mode, subtly bad things happen to confidentiality if an IV is ever reused with the same key, but in GCM, horrible things also happen to integrity if an IV is ever reused with the same key. Upon seeing the MACs associated with two messages encrypted using the same key and IV, an attacker can compute the integrity protection secret H and can then forge arbitrary messages. So it is very important that an implementation never reuse an IV.

GMAC takes a 96-bit IV that the caller must ensure is never reused. The IV is concatenated with a 32-bit constant 1 and then encrypted. GMAC made this choice in order to be compatible with GCM. Both use the same format of IV, where the top 96 bits must never be reused with the same key, concatenated with a counter starting at 2 and incremented for each succeeding block. Since it would be a problem for two blocks to use the same 128-bit value, GCM limits the amount of plaintext in a single message to less than 2^{32} blocks.

As you can see, GCM requires considerably more care to use correctly than the other modes, but the payoff is doubling the speed of the implementation (more if parallelism is exploited) compared to previous mechanisms for encrypting with integrity protection, so this approach is becoming much more common.

4.5 PERFORMANCE ISSUES

Performing encryption and decryption is often computation-intensive, and in uses where data is going out over a communications line or onto a disk, there is a real-time requirement associated with it that can be challenging. One way to improve performance is to perform the block encryption operations in parallel, either with multiple CPUs or with multiple hardware encryption engines operating in parallel. While this can always be done if there are multiple streams of data being processed in parallel, the ability to exploit parallelism when encrypting or decrypting a single stream is influenced by the design of the cryptographic mode.

For example, with CBC mode the encryption of a block cannot begin until the encryption of the previous block is completed (because the previous block of ciphertext is \oplus 'd into the plaintext before it is encrypted). With ECB or the various counter modes, once a message is known, all the blocks can be encrypted in parallel if resources permit. Interestingly, this is also possible with CBC decryption even though it is not possible with CBC encryption (see Homework Problem 5). With counter mode, the encryption (or decryption) can begin even before any of the message is known. That's because the values being encrypted are counters, and these are not dependent on the message. When the message becomes known, a simple \oplus with the keystream is all that is necessary, which minimizes latency.

4.6 HOMEWORK

1. In our example mode *Randomized ECB* (see Figure 4-4), suppose that a bit of one of the ciphertext blocks, say c_i , is flipped (intentionally or unintentionally). What will that do to the decrypted message? Suppose a bit of an r_i is flipped? What would that do to the decrypted message? Suppose the ciphertext were being transmitted as a stream of octets, and one octet was lost (and the receiver was not notified that an octet was lost)? What would that do to the decrypted message?

2. In CBC mode, suppose one bit of ciphertext is flipped. What will that do to the decrypted message? Suppose the ciphertext were being transmitted as a stream of octets, and one octet was lost (without the receiver being notified that an octet was lost)? What would that do to the decrypted message? Suppose an entire block was lost. What would that do to the decrypted message?
3. Consider the following alternative method of encrypting a message. To encrypt a message, use the algorithm for CBC decryption with an implicit IV of zero. To decrypt a message, use the algorithm for CBC encryption with an implicit IV of zero. Would this work? (“Working” means that decrypt reverses the encrypt operation.) What are the security implications of this, if any, as contrasted with the “normal” CBC? Hints: How many ciphertext blocks change if you change one block of message in normal CBC compared to our proposed CBC variant? If there are repeated blocks of plaintext, are there detectable patterns in the ciphertext?
4. Suppose you know that the employee database is encrypted using randomized ECB (see §4.2.2.1), and suppose you know that your salary is in plaintext block m_7 , that you have access to the ciphertext, and you know the syntax of the plaintext. How can you modify the ciphertext to increase your salary?
5. Why is it possible to decrypt, in parallel, blocks of ciphertext that were encrypted using CBC mode, but it is not possible to encrypt, in parallel, blocks of plaintext with CBC mode?
6. Suppose a message is encrypted with CTR mode, and you know that plaintext block m_7 contains your salary. Suppose you have access to the ciphertext. How can you modify the ciphertext to increase your salary?
7. In CBC mode, suppose you reused the same IV for two messages. What are the security implications?
8. Suppose you wanted to decrypt only block n of a file that was encrypted with CBC mode (and you knew the key with which the file was encrypted). Which blocks of the encrypted file would you need to read? What cryptographic operations would you need to do?
9. How does \oplus ing K_1 into the final block in CMAC prevent the attack described in section §4.3.1.1 *CBC Forgery Attack*?
10. For each scenario, compare the number of AES operations required in XTS *versus* XEX:
 - a) Encrypting an n -block plaintext that is an integral number of blocks.
 - b) Encrypting an n -block plaintext that is not an integral number of blocks.
 - c) Decrypting an n -block ciphertext.
 - d) Decrypting only block n . (Two questions: if the ciphertext is an integral number of blocks, or if it isn’t.)
 - e) Decrypting only block $n-1$. (Two questions: if the ciphertext is an integral number of blocks, or if it isn’t.)

- f) Decrypting only block $n-2$? (Two questions: if the ciphertext is an integral number of blocks, or if it isn't.)
11. How can you do an XTS-like trick with CBC mode encryption (and decryption) so as to be length-preserving?
 12. What message could you forge (see section §4.3.1.1 *CBC Forgery Attack*) if you saw two messages with CBC-MACs—one that consisted of $m_1|m_2$ with CBC-MAC x_1 , and another that consisted of $m_1|m_2|m_3|m_4$ with CBC-MAC x_2 ?
 13. With CBC-MAC, it is possible to create a message that has all but one block chosen by the user and has any chosen CBC-MAC value, provided that one block somewhere in the message is constrained by the computation rather than chosen by the user. Show how you can construct a message that has CBC-MAC equal to 0, where you want particular values for $m_1, m_2, m_3, m_4, m_6, m_7, m_8$, and you don't care what value is in m_5 . You can do this with any key, so let's say you are constrained to use $\text{key}=K$, and let's say you are using AES-128. What does m_5 need to be so that the message $m_1|m_2|m_3|m_4|m_5|m_6|m_7|m_8$ will have CBC-MAC equal to 0?

5

CRYPTOGRAPHIC HASHES

5.1 INTRODUCTION

A hash function inputs an arbitrary-sized bitstring and outputs a fixed-size bitstring, ideally so that all output values are equally likely. A **cryptographic hash** (also known as a **message digest**) has some extra security properties:

- **preimage resistance:** It should be computationally infeasible to find a message that has a given pre-specified hash.
- **collision resistance:** It should be computationally infeasible to find two messages that have the same hash.
- **second preimage resistance:** It should be computationally infeasible to find a second message that has the same hash as a given message.

The term *message digest* was originally more popular, but *hash* is more commonly used today. As evidence that the world has largely abandoned use of the term *message digest*, the NIST hash functions [NIST15b][NIST15c] have names beginning with *SHA*, which stands for **secure hash algorithm**. Earlier hash algorithms had names beginning with *MD* (*e.g.*, *MD5*), which stands for message digest. We will use the terms *cryptographic hash*, *hash*, and *message digest* interchangeably.

In §3.2.3 *Looking Random* we talked about the ideal cipher model for a block cipher, using an imaginary box known as an oracle. An almost identical model of an (imaginary) oracle applies to hash functions. The hash **oracle** has a table of $\langle \text{input-value}, \text{output-value} \rangle$ pairs, and if asked what the output is for an input, if that input value is in its table, it responds with the output value as specified. If the input value is not in its table, the oracle generates a random bitstring for the output value, adds that $\langle \text{input-value}, \text{output-value} \rangle$ pair to its table, and gives that output as the answer. Often hash functions are used in ways that are only secure if they “look random” in ways that are not completely captured by the properties of collision resistance, preimage resistance, and second preimage resistance.

In proofs, cryptographers would like to say things like “This protocol is secure so long as the hash function it uses is secure.” But it’s hard to define exactly what it means for a hash function to

be secure, so they instead say it would be secure if the hash function were a random oracle. This is often called a proof in the random oracle model.

Ideally, the difficulty of finding a preimage of a given n -bit hash should be no better than brute-force search, which would mean testing approximately 2^n messages. As we'll see, finding a collision (two messages with the same n -bit hash) is easier. Even if the hash were a random oracle, finding a collision will only require testing $2^{n/2}$ messages. Given these properties, a secure hash function with n bits should be derivable from a hash function with more than n bits merely by taking any particular subset of n bits from the larger hash.

There certainly will be many messages that yield the same hash, because a message can be of arbitrary length and the hash will be some fixed length, say 256 bits. For instance, for 1024-bit messages and a 256-bit hash there are, on average, 2^{768} messages that map to any one particular hash. So, certainly, by trying lots of messages, one would eventually find two that mapped to the same hash. The problem is that "lots" is so many that it is essentially impossible. Assuming a good 256-bit hash function, it would take trying approximately 2^{256} possible messages before one would find a message that mapped to a particular hash, or approximately 2^{128} messages before finding two that had the same hash (see §5.2 *The Birthday Problem*).

What can an attacker do if he can find a message with a specific hash? If Alice has signed a message saying "Alice agrees to pay Bob \$10", she is signing the hash of that message. If Bob could find another message that has the same hash, then Alice's signature works as a signature on the new message. For instance, Bob might try different messages until he finds one with the hash that Alice signed, where the new message says "Alice agrees to pay Bob \$9493840298.21".

What can an attacker do if he can find two messages with the same hash? Suppose Alice wants to fire Fred, and asks her diabolical secretary, Bob, who happens to be Fred's friend, to compose a letter explaining that Fred should be fired, and why. After Bob writes the letter, Alice will read it, compute a hash, and cryptographically sign the hash using her private key. Bob would like to instead write a letter saying that Fred is wonderful and his salary ought to be doubled. However, Bob cannot forge Alice's signature on a new hash. If he can find two messages with the same hash, though, one that Alice will agree to sign because it captures what she'd like to say and one that says what Bob would like to say, then Bob can substitute his own message after Alice generates the signed hash, and it will look like Alice signed Bob's message.

To make the example easy, suppose the hash function outputs only 64 bits and is a good hash function in the sense that its output looks random. Then the only way to find two messages with the same hash would be by trying enough messages so that, by the birthday problem, two would have the same hash.

If Bob started by writing a letter that Alice would approve of, found the hash of that, and then attempted to find a different message with that hash, he'd have to try 2^{64} different messages. However, suppose he had a way of generating lots of messages of each type (type 1—those that Alice would be willing to sign; type 2—those that Bob would like to send). Then by the birthday problem he'd only have to try about 2^{32} messages of each type before he found two whose hashes matched.

How can Bob possibly generate that many letters, especially since they'd all have to make sense to a human? Well, suppose there are two choices of wording in each of 32 places in the letter. Then there are 2^{32} possible messages he can generate. For example:

Type 1 message

I am writing {this memo | } to {demand | request | inform you} that {Fred | Mr. Fred Jones} {must | } be {fired | terminated} {at once | immediately}. As the {July 11 | 11 July} {memorandum | memo} {from | issued by} {personnel | human resources} states, to meet {our | the corporate} {quarterly | third quarter} budget {targets | goals}, {we must eliminate all discretionary spending | all discretionary spending must be eliminated}.

{Despite | Ignoring} that {memo | memorandum | order}, Fred {ordered | purchased} {Post-its | nonessential supplies} in a flagrant disregard for the company's {budgetary crisis | current financial difficulties}.

Type 2 message

I am writing {this letter | this memo | this memorandum | } to {officially | } commend Fred Jones {for his | } courage and independent thinking | independent thinking and courage}. {He | Fred} {clearly | } understands {the need | how} to get {the | his} job {done | accomplished} {at all costs | by whatever means necessary}, and {knows | can see} when to ignore bureaucratic {nonsense | impediments}. I {am hereby recommending | hereby recommend} {him | Fred} for {promotion | immediate advancement} and {further | } recommend a {hefty | large} {salary | compensation} increase.

There are enough computer-generatable variants of the two letters that Bob can compute hashes on the various variants until he finds a match. With a standard laptop processor, it's pretty easy to generate and test 2^{32} messages in a few minutes. This shows that a 64-bit hash (our example) is too small. As a rule, if you want your hash function to be as secure as your block cipher with an n -bit key, the hash should have $2n$ bits.

Ideally, the hash function should be easy to compute. One wonders what the “minimal” secure hash function might be. It is safer for a function to be overkill, in the sense of shuffling beyond what is necessary, but then it is harder to compute than necessary. The designers would rather waste computation than discover later that the function was not secure. Just as with block ciphers, the hash algorithms tend to be computed in rounds. Designers typically choose the number of rounds by finding the smallest number of rounds necessary to protect against known attacks, and then add a few extra rounds just to be safe, in case attack techniques improve.

Later in this chapter we'll describe the hash functions that have been standardized by NIST.

5.2 THE BIRTHDAY PROBLEM

If there are 23 or more people in a room, the odds are better than 50% that two of them will have the same birthday. Analyzing this parlor trick can give us some insight into cryptography. We'll assume that a birthday is basically an unpredictable function taking a human to one of 365 values (yeah yeah, 366 for you nerds).

Let's do this in a slightly more general way. Let's assume n inputs (which would be humans in the birthday example) and k possible outputs, and an unpredictable mapping from input to output. With n inputs, there are $n(n-1)/2$ pairs of inputs. For each pair there's a probability of $1/k$ of both inputs producing the same output value, so you'll need about $k/2$ pairs in order for the probability to be about $1/2$ that you'll find a matching pair. That means that if n is greater than \sqrt{k} , there's a good chance of finding a matching pair.

5.3 A BRIEF HISTORY OF HASH FUNCTIONS

Surprisingly, the drive for hash algorithms started with public key cryptography. RSA was invented, which made it possible to digitally sign messages. But computing a signature on a long message with RSA was sufficiently slow that RSA would not have been practical by itself. A cryptographically secure hash function with high performance would make RSA much more useful. Instead of computing a signature over a long message, the RSA signature would be computed on the message's hash. So MD and MD2 (RFC 1319) were created (around 1989). MD was proprietary and never published. It was used in some of RSADSI's secure mail products.

Around 1990, Ralph Merkle of Xerox developed a hash algorithm called SNEFRU [MERK90] that was several times faster than MD2. This prodded Ron Rivest into developing MD4 (RFC 1320), a hash algorithm that took advantage of the fact that newer processors could do 32-bit operations and was therefore able to be even faster than SNEFRU. Then SNEFRU was broken by Biham and Shamir [BIHA92] (the cryptographic community considered it broken because they were able to find two messages with the same SNEFRU hash). Independently, den Boer and Bosselaers [DENB92] found weaknesses in a version of MD4 with two rounds instead of three. This did not officially break MD4, but it made Ron Rivest sufficiently nervous that he decided to strengthen it and create (in 1991) MD5 (RFC 1321), which is a little slower than MD4.

What does a “weakness” or “being broken” mean for hash algorithms? In an ideal n -bit hash algorithm, finding a collision should require $2^{n/2}$ operations. A weakness is an attack that could theoretically find collisions in less than $2^{n/2}$ operations. Being broken means publishing an actual collision.

MD4 and MD5 were subsequently broken (in the sense that collisions were found), though they could be used securely in some cases if you're careful. (However, if you were careful and used one of these functions in a way that was secure, you'd have to be prepared to explain to your customers why you are using a function that they read was "broken".)

After MD4 and MD5 were published, but before they were broken, NIST proposed the NSA-designed SHA (1993). SHA is very similar in design to MD5, but even more strengthened. It is 160 bits instead of 128, and a little slower. After a never-published flaw in the SHA proposal was discovered, NIST revised it at the eleventh hour in an effort to make it more secure, and called the revised version SHA-1 (in 1995, and the previous version was retroactively named SHA-0). They were apparently right to be concerned. After years of cryptanalysis, weaknesses have been found in both SHA-0 and SHA-1, but as of 2022, the best known SHA-1 attack for finding collisions costs something like 2^{63} operations, while SHA-0 collisions can be found for only about 2^{39} operations.

Wanting a longer hash length, NIST adopted and standardized SHA-2 in 2004 (which, like SHA-1, was designed by NSA). Then SHA-1 was broken (a collision-finding algorithm with an expected runtime of significantly less than the expected 2^{80} hash operations was found). Since SHA-2 uses an algorithm similar to SHA-1, it was feared that SHA-2 might someday be broken (although it hasn't been as of 2021). So NIST standardized the SHA-3 family, which uses a rather different structure in hopes that any vulnerabilities in SHA-1 would not apply. Another attraction of SHA-3 is that, like AES, SHA-3 was the result of a competition rather than being designed by NSA. While SHA-1 produces a 160-bit hash, the SHA-2 and SHA-3 families produce 224-, 256-, 384-, and 512-bit hashes. The various hash sizes in SHA-2 and SHA-3 are intended to match the security of 3DES, AES-128, AES-192, and AES-256, respectively.

Note that the standard terminology for the SHA-2 family is to only refer to the hash size, as in SHA-224, SHA-256, SHA-384, SHA-512, and SHA-512/n where n is the truncated size of the hash and can be any value from 1 through 511 except for 384 (for that you're supposed to use SHA-384). We think the terminology SHA2-224, SHA2-256,... is clearer, especially since the SHA-3 family can produce many of those same hash lengths and are referred to as SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

SHA-2 could have specified a single algorithm, say SHA2-512, and then specified each of the smaller hashes as just truncating the 512-bit hash to the relevant size. But since SHA2-512 was designed to have security equal to AES-256, it would be slower than necessary if it just needed the security of AES-128. All the SHA-2 family are very similar. SHA2-384 is essentially the same as SHA2-512 but truncated to 384 bits. And SHA2-224 is essentially the same as SHA2-256 but truncated to 224 bits. And if you are in the mood for a hash value other than these sizes (but less than 512), you can truncate SHA2-512 to any size you like. One difference between the sizes, though, is that a value we call the IV (see Figure 5-4) is unique to each intended hash length in order to avoid having the first n bits of a larger hash be the same value as the computed n-bit hash. SHA2-256 and SHA2-512 are very similar algorithms, but the word sizes in SHA2-512 are double the sizes in SHA2-256, there are more rounds in SHA2-512, and they use different constants. SHA2-512 actu-

ally runs faster on 64-bit CPUs than SHA2-256, so many implementations that only want a 256-bit hash use SHA2-512 truncated to 256 bits.

The SHA-3 family has parameters (r and c ; see §5.6.2 *Construction of SHA-3*) that can trade off security and performance, and the standard specifies values of these parameters for each of the standardized hash sizes. Other than the values of these parameters, all the SHA-3 algorithms are identical.

There have been many other hash functions that were designed along the way, including perfectly reasonable ones that were submitted to the SHA-3 hash competition that simply lost out to KECCAK due to issues such as performance or safety margin. There were also hash competitions in Europe and Japan, and we haven't described the winners of those. We can't mention all the hash functions that have been designed, or we'd have to change the name of this section from *A Brief History of Hash Functions* to *A Comprehensive History of Hash Functions*. However, we will mention the RIPEMD family of various-size hashes (128, 160, 256, and 320) because RIPEMD-160 [DOBB96] is used in some cryptocurrencies, and no weaknesses in RIPEMD-160 have been published. Note that *MD* stands for message digest.

5.4 NIFTY THINGS TO DO WITH A HASH

Before we look at the details of several popular hash algorithms, let's look at some interesting uses of hashes.

5.4.1 Digital Signatures

Since a hash is fixed length (e.g., 256 bits), and messages can be arbitrarily long, there will be many messages with the same 256-bit hash. However, hashes are designed so that it is very unlikely to ever find two messages with the same hash. Therefore, the hash can be used as a shorthand of a message. Since public key signatures are slow, and computing hashes are fast, digital signatures sign the hash of a message instead of the actual message.

In fact, hashing the message before signing is also more secure. Without hashing, many digital signature algorithms (including RSA) have the property that an attacker who sees a signed message can forge other $\langle \text{message}, \text{signature} \rangle$ pairs that will successfully verify. If, instead, the public key is used to sign a hash of the message, then although the attacker might still be able to construct valid signatures for certain constructed hash values, these signatures would only be useful if the attacker could find a message that hashed to the constructed hash value.

5.4.2 Password Database

Let's assume a human at a client machine wishes to authenticate to a server. In the simplest form of password authentication, the server keeps a list of $\langle \text{username}, \text{password} \rangle$ pairs. The human types their username and password, and the client machine sends these to the server (perhaps over an encrypted connection to the server). The server verifies that the $\langle \text{username}, \text{password} \rangle$ pair is in the list. This design has the problem that if someone stole the database at the server, they would know all the usernames and passwords.

It is more secure to have the server store a database of $\langle \text{username}, \text{hash}(\text{password}) \rangle$ pairs. There are various ways the hashed password can be used for authentication. The client might send the $\langle \text{username}, \text{password} \rangle$ to the server, which would then hash the received password and verify that the result matches the $\langle \text{username}, \text{hash}(\text{password}) \rangle$ in the database. Or the client might hash the password and use that as a shared secret with the server, for instance, by having the server send a challenge and having the client machine send back $\text{hash}(\text{challenge}|\text{hash}(\text{password}))$. We discuss storage of passwords more fully in §9.8 *Off-Line Password Guessing*.

5.4.3 Secure Shorthand of Larger Piece of Data

A typical storage device stores blocks of data, typically 8K octets, though the blocks might be variable length. In some cases, such as backing up the contents of all the laptops at a company, a lot of the data will be the same on all the laptops (*e.g.*, the software of the operating system). To save storage, a storage system might perform what is known as **deduplication**, where the storage system only keeps a single copy of a block and has all references to that block point to the one copy of the stored block.

A typical method of performing deduplication is for the storage system to keep a table of hashes of all the blocks that are stored. When the storage system receives a block to be stored, it hashes the block, checks whether that hash is in the deduplication table, and, if so, it does not need to store the block again, although it might need to store a new pointer to that block.

To save network bandwidth as well as storage, the client might send the hash of a block to the storage system, and the storage system could respond “already have that” or “send me the block”.

5.4.4 Hash Chains

A hash chain takes a piece of data D and hashes it many times, *e.g.*, $\text{hash}^{500}(D)$, *i.e.*, $\text{hash}(\text{hash}(\cdots\text{hash}(D)\cdots))$. There are many applications of this concept.

- Hash chains can make brute-force password guessing slower. Suppose someone stole the database of hashed user passwords. If the server stores $\text{hash}^{500}(\text{pwd})$ rather than $\text{hash}(\text{pwd})$

for each user, then an attacker that stole the server database and is attempting through brute force to find potential passwords in the database would have 500 times as much work for each password. The performance impact of the client machine needing to compute $\text{hash}^{500}(\text{pwd})$ rather than $\text{hash}(\text{pwd})$ when the user is logging in is minimal.

- Hash chains are used in post-quantum hash-based signatures (see §8.2 *Hash-based Signatures*). A signer signs the value $n-i$ by revealing $\text{hash}^{n-i}(\text{secret})$.
- Hash chains have been used as a way of proving, a limited number of times, that you know a secret, without needing to use public keys. A good example of this is Lamport's hash (see §9.12 *Lamport's Hash*). A server Bob is configured with user Alice's password hashed a lot of times, *e.g.*, $\text{hash}^{1000}(\text{pwd})$, along with n , the number of times Bob believes the password is hashed (in this case, $n=1000$). When Alice attempts to authenticate, Bob sends n , and Alice replies with her password hashed $n-1$ times. Bob takes what he receives from Alice, hashes it, and if the result matches his database, he replaces $\text{hash}^n(\text{pwd})$ with the value he received from Alice ($\text{hash}^{n-1}(\text{pwd})$) and decrements n .

5.4.5 Blockchain

A blockchain is a sequence of blocks, each containing the hash of the previous block. This data structure prevents someone from easily tampering with a block in the middle of the chain, because they would have to recompute all the subsequent blocks. See §15.3 *Bitcoin*.

5.4.6 Puzzles

If a server Bob is being overwhelmed due to a denial of service attack, Bob can slow down requests from each client by refusing a normal connection request, saying, “Try again, but this time include a value whose hash has the bottom k bits = x .” He can choose how much to slow a client down by choosing the number of bits that a client needs to solve. Bob doesn't want to remember which puzzle he sent to which client, and he doesn't want to give them all the same puzzle, so he'd probably choose the puzzle to be some sort of secret he knows, hashed with the IP address of the client.

While the previous paragraph adjusts the computation necessary to solve a puzzle by adjusting how many bits of a hash have to match a particular value, a variant (used by cryptocurrencies; see §15.3.5 *Mining*) adjusts how much computation is necessary to find a block by specifying a maximum value of the hash of a block.

5.4.7 Bit Commitment

Suppose Alice and Bob are talking on the phone, and Alice flips a coin to determine which of them will get to keep the house after their divorce. Bob will call “heads” or “tails”. Alice will reveal the result of the coin flip.

The problem with that protocol is that if Bob tells Alice which he chooses (say, “heads”), then Alice can say “It was tails” (regardless of what the coin flip actually was). If instead, Alice reveals the result of the coin flip before Bob reveals which he chose and she says that the result was heads, then Bob can say “I chose heads.”

Bit commitment is a very clever solution to this dilemma. Alice reveals a quantity that commits her to the coin-flip value but does not reveal to Bob what the value is. Alice chooses a random number R and sends the hash of R to Bob. If the low order bit of R is 0, Alice is committing to heads, and if 1, to tails. Alice sends $\text{hash}(R)$ to Bob. Alice can’t change her mind, since she cannot find two numbers that have the same hash. Knowing $\text{hash}(R)$ does not help Bob choose. Bob makes the choice of heads or tails, and then Alice must reveal R (the jargon is that Alice *opens the commitment*).

5.4.8 Hash Trees

A single hash value for a large amount of data can be problematic, particularly if only a small piece of the data needs to be read or written at any time. Ralph Merkle invented a scheme in 1979 known as a hash tree or Merkle tree [MERK79] (see Figure 5-1*). The idea is that each block of data is hashed, then groups of hashes are concatenated and hashed, then groups of those hashes are hashed,

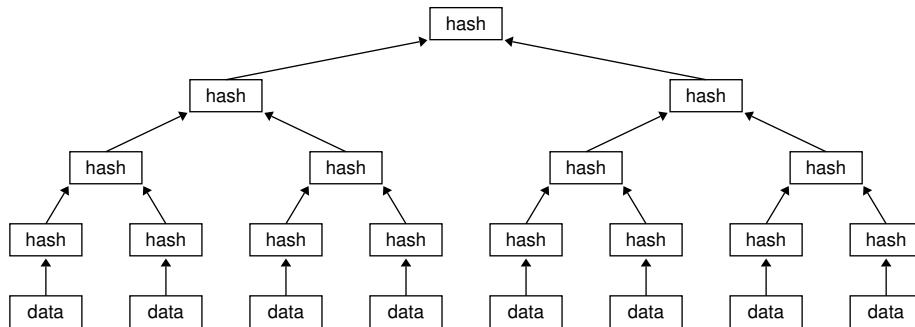


Figure 5-1. Hash Tree

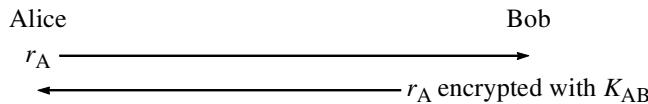
*I₃ think the root of the tree should be at the bottom and the leaves at the top, but I₂ consulted the Internet and determined that I₃’m wrong.

and so on until there is only a single master hash, which is the root of the hash tree. Given the master hash, you can verify any block of data if you have the data and all the hash values of siblings along the path from the data to the root, because this allows you to compute the intermediate hash values on the path and then compare your final value with the root hash. Typically, a binary tree is used so only one auxiliary hash value is needed per level in order to verify a data block. You still have to verify that you've obtained the actual value of the root hash.

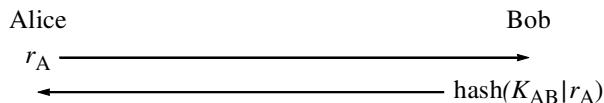
5.4.9 Authentication

Surprisingly, if there is a shared secret, a hash algorithm can be used in all the ways that secret key block ciphers are used.

In §2.2.3 *Authentication*, we discussed an example of how to use a block cipher for authentication if Alice and Bob share a secret key K_{AB} . To authenticate Bob, Alice sends a challenge r_A , Bob encrypts r_A with K_{AB} , and Alice decrypts what she receives and verifies it matches r_A .



If Bob and Alice share a secret K_{AB} , they can authenticate using a hash algorithm instead of a block cipher. Since hash algorithms aren't reversible, it can't work quite the same way. However, the hash function can accomplish the same thing. Alice still sends a challenge r_A . Bob then concatenates K_{AB} with r_A , computes a hash of that, and transmits the result. Alice can't "decrypt" the result. However, she can do the same computation and check that the result matches what Bob sent.



5.4.10 Computing a MAC with a Hash

In §4.3 *Generating MACs*, we described how to compute a MAC with a block cipher, using a secret K_{AB} that Alice and Bob share. Can we compute a MAC using a hash function instead of a block cipher? We will do roughly the same trick for the MAC as we did for authentication. We concatenate a shared secret K_{AB} with the message M and use $\text{hash}(K_{AB}|M)$ as the MAC.

This scheme should work, except for some idiosyncrasies of most of the popular hash algorithms (including MD4, MD5, SHA-1, SHA2-256, and SHA2-512), which would allow an attacker

to be able to compute a MAC of a longer message beginning with M , given message M and $\text{hash}(K_{AB}|M)$.

Assume hash is a function that acts as described in Figure 5-2. Assume the input to hash is $K_{AB}|M$. Hash will add some padding at the end to make sure the input is an integral number of blocks, but to make the example simpler, let's ignore the padding for now. Hash breaks the input into blocks. At each stage, hash uses a function we'll call *compress*, which takes two inputs: the next block of message and the output of the previous stage of the hash (called *intermediate hash* in Figure 5-2), and outputs a new intermediate value. The final intermediate hash is the hash of the message.

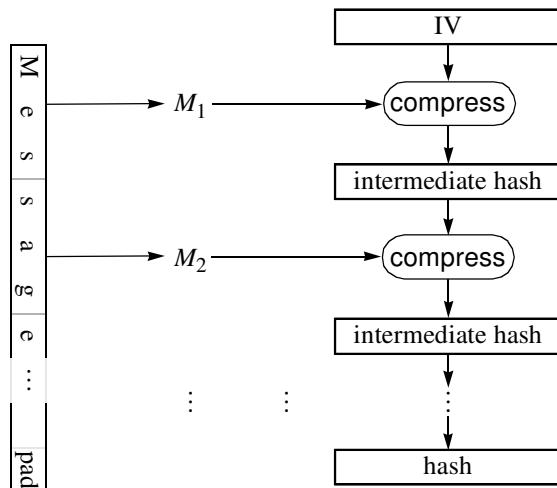


Figure 5-2. Hash structure vulnerable to append attack

There's an attack that we'll call an **append attack**. Let's assume Carol (who does not know K_{AB}) would like to send a different message to Bob, and have it look like it came from Alice. Assume Carol doesn't care what's in most of the message. She only cares that the end of the message says P.S. Give Carol a promotion and triple her salary. Alice has transmitted to Bob some message M , and $\text{hash}(K_{AB}|M)$. Carol can see both of those quantities. She concatenates whatever she likes to the end of M and initializes the intermediate hash computation with $\text{hash}(K_{AB}|M)$. In other words, in Figure 5-2, she sets IV to the value that Alice sent to Bob as the MAC of M . Carol does not need to know the shared secret K_{AB} in order to continue the hash computation from the end of M .

How can we avoid this flaw? One technique that would work is to use only some of the bits of the hash as the MAC. For instance, use the low-order 64 bits of a 256-bit hash. The attacker doesn't have enough information to continue the hash computation (other than correctly guessing the missing 192 bits of the hash). Using only 64 bits of the hash as a MAC is not any less secure since there

is no way, without knowing the secret, that an attacker can calculate or verify the MAC. The best that can be done is to generate a random 64-bit MAC for the message you'd like to send and hope that you'll be really, really lucky (1 in 2^{64} lucky). Note that SHA2-384 and (to a lesser extent because it only truncates by 32 bits) SHA-224 effectively do this because they compute a larger hash (512 and 256, respectively) and truncate.

The solution the industry has settled on is HMAC (§5.4.11). HMAC concatenates the secret to the front of the message, hashes the combination, concatenates the secret to the front of the hash, and hashes the combination again. The actual construction is a little more complicated than this and is described in §5.4.11. HMAC has lower performance than truncating the hash because it does a second hash. But the second hash is only computed over the secret and a hash, so it does not add much cost to large messages. In the worst case, if the message concatenated with the key fit into a single (m -bit) block, HMAC would be four times as expensive as using a truncated hash. However, if many small messages are to be HMAC'd with the same key, it is possible to reuse the part of the computation that hashes the key, so that HMAC would only be twice as slow. With a large enough message, HMAC's performance is only negligibly worse.

We call any hash combining the secret key and the data a **keyed hash**.

5.4.11 HMAC

HMAC resulted from an effort to find a MAC algorithm that could be proven to be secure if the underlying hash's compression function was secure. HMAC was proven to have these two properties (given the underlying hash was secure):

- collision resistance (infeasible to find two inputs that yield the same output)
- an attacker who doesn't know the key K cannot compute $\text{HMAC}(K, x)$ for data x , even if they can see the value of $\text{HMAC}(K, y)$, for arbitrarily many values of y not equal to x

In essence, HMAC prepends the key to the data, hashes it, and then prepends the key to the result and hashes that. This nested hash with secret inputs to both iterations prevents the extension attacks that would be possible if you simply hashed the key and message once. In detail, HMAC takes a variable-length key and a variable-sized message and produces a fixed-size output that is the same size as the output of the underlying hash. (See Figure 5-3.)

The number of bits in the output of HMAC is equal to the number of bits in the underlying hash function. HMAC also pads differently depending on the blocksize (in bits) of the underlying hash. Let's call the blocksize B . $B=512$ bits (64 octets) in SHA-1, SHA2-224, and SHA2-256. $B=1024$ bits (128 octets) in SHA2-384 and SHA2-512. $B=1152$ bits (144 octets) in SHA3-224. $B=1088$ bits (136 octets) in SHA3-256. $B=832$ bits (104 octets) in SHA3-384. $B=576$ bits (72 octets) in SHA3-512.

If the key is smaller than B bits, HMAC pads it out to B bits by appending 0s. If the key is larger than B bits, HMAC first hashes the key, and if the hash is smaller than B bits, HMAC pads the result out to B bits. It then \oplus s (bitwise exclusive ors) the padded key with a constant bit string repeating 00110110, concatenates it with the message to be protected, and computes a hash. It \oplus s the padded key with another constant bit string repeating 01011100, concatenates that with the result of the first hash, and computes a second hash on the result.

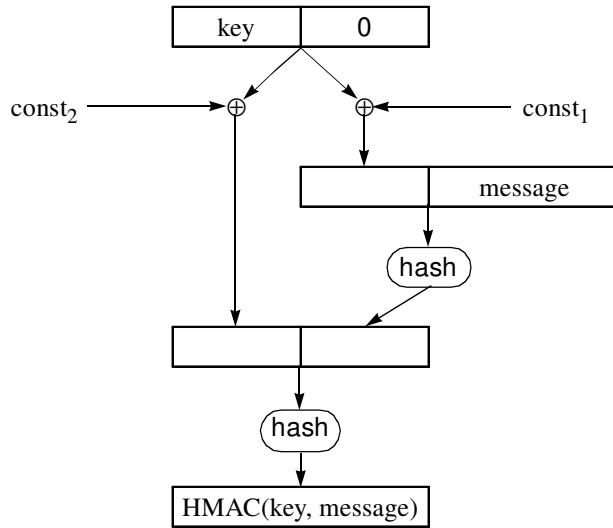


Figure 5-3. HMAC

Note that with HMAC, it is possible to find pairs of keys that result in the same HMAC values. For example, if a key K is shorter than B bits, say, 200 bits, then $K|0, K|00, \dots$ will all result in the same HMAC of any message. Likewise, if K is longer than B bits, HMAC first hashes the key, and if the hash is smaller than B bits, HMAC pads the result out to B bits. Therefore, K and the hash of K would be equivalent keys. This issue is not a problem for most applications, especially if the application requires a fixed-size key.

You can use HMAC with SHA-3. However, since SHA-3 is not vulnerable to the append attack, there's no need for the added complication of hashing twice and using the key twice. NIST has specified a simple prepend MAC based on SHA-3 (with different padding for domain separation) called KMAC [NIST16]. KMAC has the additional advantage that, in the unlikely event that the equivalent key property of HMAC is a problem for your application, you don't have to think about it, because KMAC does not have the equivalent key problem (unless of course you can find SHA-3 collisions).

5.4.12 Encryption with a Secret and a Hash Algorithm

“Encryption with a hash algorithm is easy!” you say. “But let me see you do decryption!” Hash algorithms are not reversible, so the trick is to design a scheme in which both encryption and decryption run the hash algorithm in the forward direction. The scheme we’ll describe is reminiscent of CTR mode (§4.2.3) used by block ciphers.

CTR mode generates a pseudorandom bit stream that can be used as a one-time pad. Encryption (and decryption) consist of \oplus ing the message with the pseudorandom bit stream. CTR mode uses a block cipher to generate the bit stream. We can just as easily use a hash algorithm to generate a pseudorandom bit stream.

Alice and Bob need a shared secret, K_{AB} . Alice wants to send Bob a message. She computes $\text{hash}(K_{AB}|1)$. That gives the first block of the bit stream, b_1 . Then she computes $\text{hash}(K_{AB}|2)$ and uses that as b_2 . In general, b_i is $\text{hash}(K_{AB}|i)$. To encrypt multiple messages with the same key, a unique IV is needed. In that case the first block would be $\text{hash}(K_{AB}|\text{IV}|1)$.

Note that there are not only many ways of doing this securely, but there are also ways of doing it insecurely. For example, if the secret, the IV, and the counter are variable length, concatenation of these quantities could lead to different triples resulting in the same value to be hashed.

Alice can generate the bit stream in advance, before the message is known. Then when Alice wishes to send the message, she \oplus s it with as much of the generated bit stream as necessary.

$$\begin{array}{ll} b_1 = \text{hash}(K_{AB}|1) & c_1 = p_1 \oplus b_1 \\ b_2 = \text{hash}(K_{AB}|2) & c_2 = p_2 \oplus b_2 \\ \vdots & \vdots \\ b_i = \text{hash}(K_{AB}|i) & c_i = p_i \oplus b_i \\ \vdots & \vdots \end{array}$$

It is not secure to use the same bit stream twice, so Alice must use a unique key or a unique IV for each message she sends to Bob. She must transmit the IV to Bob. Alice can generate the bit stream in advance of encrypting the message, but Bob cannot generate the bit stream until he sees the IV. As with CTR mode, this mode of encryption gives no integrity protection, so it should be used with a separate MAC scheme.

5.5 CREATING A HASH USING A BLOCK CIPHER

Secret key block ciphers do a great job of scrambling their inputs, but they don't work as hash functions because they are reversible. If you think of a block cipher as a function that takes two inputs (plaintext and key) and produces one output (ciphertext), a design goal of a good block cipher is that it is impossible to figure out what key is being used even if you see arbitrary amounts of plaintext and ciphertext. That suggests that if you supply the quantity to be hashed as the key and encrypt a constant, the output could be used as a hash of the input.

The original UNIX password hash did just that. It used a block cipher, a slightly modified version of DES, to compute a hash of a password. It never had to reverse the hash to obtain a password. Instead, when the user typed a password, UNIX used the same algorithm to hash the entered quantity and compared the result with the stored password hash.

The hashing algorithm first converted the password into a secret key. This key was then used to encrypt a block of 0s. The method of turning a text string into a secret key was simply to pack the 7-bit ASCII associated with each of the first eight characters of the password into a 56-bit quantity into which DES parity was inserted. (UNIX passwords were allowed to be longer than eight characters, but only the first eight characters were checked, so the passwords `PASSWORDqv` and `PASSWORDxyz` would both work if the user's password was `PASSWORD`.)

A 12-bit random number, known as **salt**, was stored with the hashed password. For an explanation of why salt is useful, see §9.8 *Off-Line Password Guessing*. A modified DES was used instead of standard DES to prevent hardware accelerators designed for DES from being used to reverse the password hash. The salt was used to modify the DES algorithm.

To summarize, each time a password was set, a 12-bit salt was generated. The salt was not chosen by, or visible to, the user. The first eight characters of the password was converted into a secret key. The salt was used to define a modified DES algorithm. The modified DES algorithm used the converted password as a key to encrypt a block of 0s. The result was stored along with the salt as the user's hashed password.

Block ciphers are used internally in many hash algorithms, as we'll see in §5.6.1.

5.6 CONSTRUCTION OF HASH FUNCTIONS

As with encryption systems, hash functions (which can hash arbitrary-length messages) are constructed from simpler functions that take fixed-length inputs. The hash function applies the simpler function iteratively over fixed-size chunks of the (padded to an integral number of chunks) message. In this section, we just give an intuitive understanding of these algorithms. If you need to

implement them, you will need to refer to the standards. Or, if you like to read code, you can find (courtesy of me₃) Python implementations in <https://github.com/ms0/crypto>. That code is intended to be readable rather than efficient, and is also not designed to be secure against side-channel attacks.

5.6.1 Construction of MD4, MD5, SHA-1 and SHA-2

All these hash functions have the same structure, known as the Merkle–Damgård construction (see Figure 5-4), but they have different padding, hash size, block size, and compression functions. A **compression function** takes n bits of input and produces m bits of output, where $m < n$.

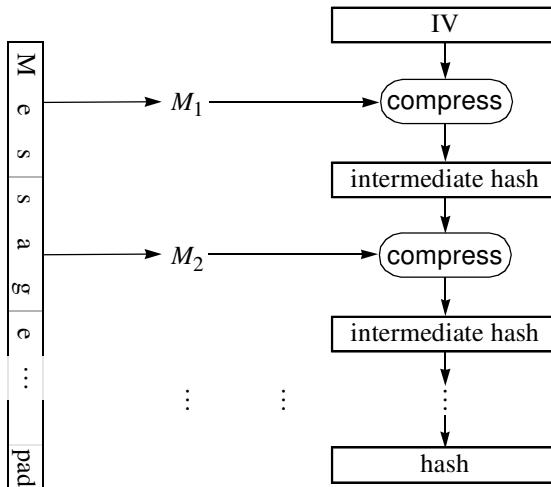


Figure 5-4. Merkle–Damgård Structure

A hash algorithm using this construction keeps an m -bit state, where m is the size of the hash to be produced. In Figure 5-4, we refer to the state as the intermediate hash. The intermediate hash is initialized to an m -bit constant defined by the algorithm. We've called the initial state IV, since it acts very similarly to an IV in an encryption algorithm. The output of the final stage is the m -bit hash.

The (padded) message is broken into k -bit chunks. At each stage, the compression function takes as input the previous m -bit intermediate hash and k bits of the message and outputs the next m -bit intermediate hash.

The hash functions (MD4, MD5, SHA-1, and the SHA-2 family) build their compression algorithm out of an encryption algorithm with a k -bit key and an m -bit blocksize. The k -bit chunk of the message is used to encrypt the m -bit intermediate state. For more detail on the structure of their encryption algorithms, see §5.8.1 and §5.8.2.

You might think that the encryption algorithm could be the compression algorithm, but there is a security problem with doing so. Assume we have an encryption algorithm with an m -bit block-size. We would like a brute-force search for a preimage of a particular m -bit hash value v to require work approximately 2^m . However, if the compression function in Figure 5-4 were just an encryption function, an attack known as the meet-in-the-middle attack (described in the next paragraph) allows a brute-force search for a preimage to v to only require work proportional to about $2^{m/2}$.

The **meet-in-the-middle attack** works as follows. To review, the hash size is m , and the block size of the message fed in at each stage of the hash is k . The preimage we will find with this attack is a message that is two blocks ($2k$ bits) long. It doesn't matter how big the block size k is. The work required for this attack depends solely on m (the hash size). The beginning state is constrained to be the constant value specified as the IV, and the final output is constrained to equal v (because we want to find a preimage of v).

Choose $2^{m/2}$ random k -bit quantities and encrypt the IV with each one (so you will have $2^{m/2}$ encryptions of IV). Then start with v , choose another $2^{m/2}$ random k -bit quantities, and decrypt v with each. Now, by the birthday problem, you will probably have a match, *i.e.*, an intermediate hash that equals the encryption of IV with key₁ and that also equals the decryption of v with key₂. The concatenation of key₁ and key₂ would therefore be a preimage of v .

The Davies-Meyer construction (Figure 5-5) defends against the meet-in-the-middle attack.

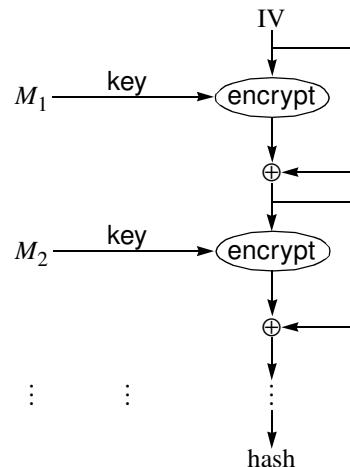


Figure 5-5. Davies-Meyer Hash Using Secret Key Cryptography

In the Davies-Meyer construction, the compression function is not simply an encryption function. Instead, at each stage, the intermediate state is not only encrypted with the next block of the message, but also \oplus 'd into the output of the encryption function.

This prevents the meet-in-the-middle attack. For each potential intermediate state resulting from one of the encryptions of IV, the result needs to be compared, not with decryptions of v , but with decryptions of $v \oplus d$ with the candidate intermediate state. In other words, the attacker needs to try decrypting $2^{m/2}$ different values with $2^{m/2}$ different keys, making the attacker's work proportional to 2^m . Note that all the hash functions we discuss in this section use a variant of Davies-Meyer where, instead of \oplus , $+$ is used on each 32- or 64-bit word (throwing away the carry). Either operation has the same effect.

5.6.2 Construction of SHA-3

SHA-3 is based on KECCAK, the winner of NIST's SHA-3 Cryptographic Hash Algorithm Competition. The specification can be found in [NIST15c]. The designers of the chosen algorithm were Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche.

The hash algorithms we've discussed before take as input an arbitrary-sized input and generate a fixed-length output. In contrast, KECCAK is much more flexible, and allows both an arbitrary-sized input and an arbitrary-sized output, as well as having parameters that allow trading off performance against security.

NIST standard FIPS 202 defines KECCAK parameter settings for SHA3-224, SHA3-256, SHA3-384, and SHA3-512 (where the number after the hyphen refers to the number of bits in the hash). It also defines parameter settings for two additional functions called SHAKE128 and SHAKE256. Both of the SHAKE functions can generate an arbitrary number of hash bits. The difference is the security strength. SHAKE128 is equivalent in security to AES128 and is faster than SHAKE256, which is equivalent in security strength to AES256. The SHAKE functions are known as **extendable output functions (XOFs)**. If instead of a known message, the input were a secret seed, the SHAKE functions could be used to compute a stream of pseudorandom bits, *e.g.*, for a stream cipher.

The central part of the algorithm is called the SPONGE construction, so named because an arbitrary number of input bits are absorbed and an arbitrary number of output bits are squeezed out. Referring to Figure 5-6, the intermediate state is 1600 bits, regardless of the desired size of the hash or the desired security level. The 1600 bits are partitioned into two parts, one of size r bits (*rate*) and one of size c bits (*capacity*). If c is larger, then r will be smaller (because their sum is 1600), and it will require more stages to compute the hash. Let's call the r -sized portion of the state the r -bits and the c -sized portion of the state the c -bits. Note that in the standard terminology, the c -bits are referred to as the *inner part*, and the r -bits are referred to as the *outer part*, but we think our terminology is clearer. In SHA-3, the number of c -bits is twice the size of the hash, and r is always larger than the size of the hash. In SHAKE128, c is 256, and in SHAKE256, c is 512.

At each stage, the next r bits of the message are \oplus 'd into the r -bits of the state. Then the entire 1600 bits of state (both the r -bits and the c -bits) are fed into a function f , which outputs 1600

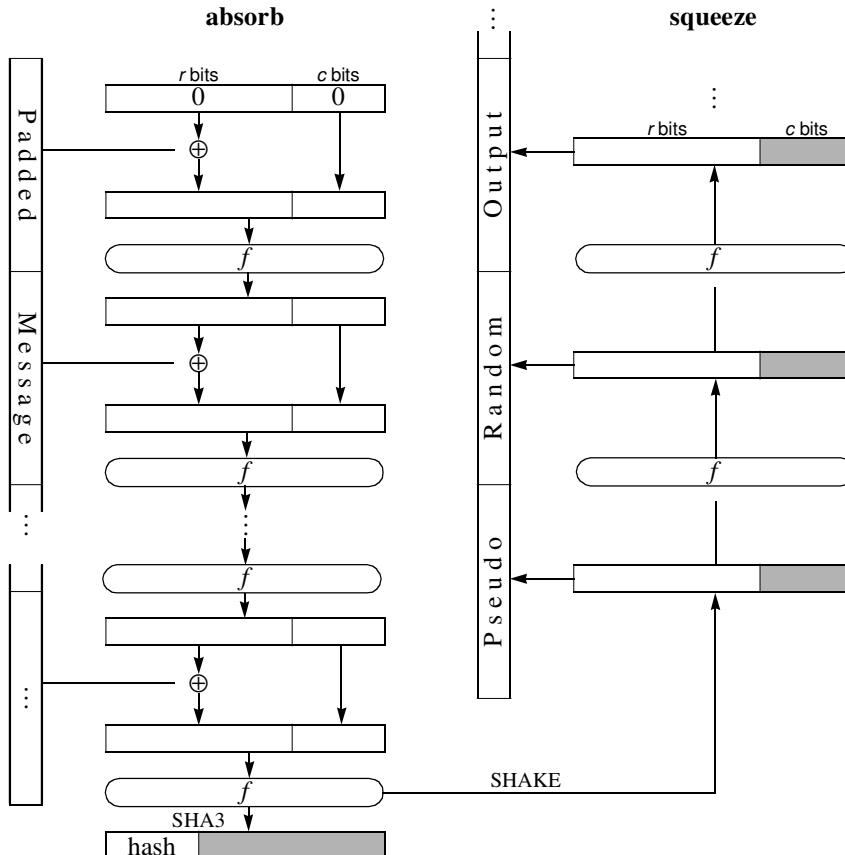


Figure 5-6. SPONGE Construction

bits as the state for the next stage. Note that both the input and the output to f are 1600 bits, so f is not a compression function. It is actually a permutation, a reversible mapping of each possible value of the input to a unique value of the output. f is defined as a mathematical function that is fast to compute in both hardware and software.

SHA-3 continues until all the bits of the message are absorbed. Then the top bits of the state are output as the SHA-3 hash. The portion of the SPONGE construction that is \oplus ing bits of the message is known as the **absorb phase**.

For SHAKE, the algorithm outputs bits after the message is completely absorbed. This phase is known as the **squeeze phase**. At each stage in the squeeze phase, the 1600 bits of state are input into f , and the r -bits of the state are copied into the output.

Notice that the larger the value of c , the more stages will be required to hash the message, because only r bits of the message will be absorbed at each stage. Also, during the squeeze phase, if c is larger, the number of bits copied into the output at each stage (r) will be smaller.

If there were no c -bits, it would be easy to find a preimage (see Homework Problem 12).

5.7 PADDING

Hash algorithms break a message into fixed-sized blocks and at each stage input the next block of the message. The message is always padded first, even if it is already a multiple of the block size.

Suppose you simply padded a message with 0s to make it be a multiple of the block size. In that case, how would you distinguish a message that was an integral number of blocks, but just happened to end in a 0, from one that was one bit short of an integral number of blocks with one bit of padding added? And if it ended in, say, five 0s, how would you know how many of those 0s were part of the message and how many were padding? Note that HMAC padding of the key does exactly that (pad with 0s), and, therefore, it is possible to find different keys that will result in the same HMAC (see §5.4.11).

5.7.1 MD4, MD5, SHA-1, and SHA2-256 Message Padding

The message to be fed into the hash computation must be a multiple of 512 bits long. The original message is padded by adding a 1 bit followed by enough 0 bits to leave the message 64 bits less than a multiple of 512 bits long. Then a 64-bit big-endian number representing the bit-length of the unpadded message is appended to the message. (See Figure 5-7.)

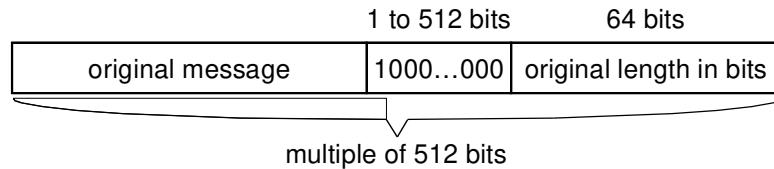


Figure 5-7. Padding for MD4, MD5, SHA-1, and SHA2-256

The case that would need the most padding is if the final block of the message were 64 bits less than a block. Then there wouldn't be room for a 1 followed by a 64-bit length, so the final message block would need to be padded with a 1 followed by 63 0s, and then a new final block would need to have 448 bits of 0 followed by the 64-bit length.

Note that since the length field is only 64 bits, the message must be less than 2^{64} bits long. We don't consider that a problem. Such a message would take over a decade to transmit at a hundred gigabits per second. As our grandmothers would surely have said, had the occasion arisen,

If you can't say something in less than 2^{64} bits, you shouldn't say it at all.

Note that SHA2-512 has a blocksize of 1024, and the padding has 128 bits for the original length. So if you *really* wanted to say something in more than 2^{64} bits (but less than 2^{128} bits), you could use SHA2-512. And if you are nervous about being limited to 2^{128} bits, you will be pleased to know that SHA-3 has no limit on message length at all.

Why is the length field useful in the padding? Given that these hashes use the Merkle–Damgård Structure (see Figure 5-4), without the length field it would be slightly easier to find a **second preimage**, which is a message M_2 , given a message M_1 , such that M_2 's hash is the same as M_1 's hash. Let's assume M_1 is, say, 256 blocks long. That means hashing would produce 256 intermediate hashes. So, if the attacker computes all 256 intermediate hashes for M_1 , say $M_{1:1}, M_{1:2}, \dots, M_{1:256}$, and if the attacker tries random messages and finds one, say M_3 , that matches any of these intermediate hashes, say $M_{1:207}$, then $M_2 = M_3|M_{1:208}| \dots |M_{1:256}$ would be a second preimage of the hash of M_1 . This means it would be 256 times easier to find a second preimage than a first preimage, which translates to eight bits less security strength.

Unfortunately, even with the length field, there is another somewhat more complicated attack to find a second preimage. Since finding collisions is so much easier than finding a preimage ($2^{n/2}$ compared with 2^n), we can think of searching for collisions as free. Ignore padding for now. Choose a bunch of single-block and double-block messages at random, search for one of each size whose SHA-2 hashes (without padding) match, and call the common SHA-2 value H_1 . Now use H_1 as an IV, search for a collision of one-block messages and three-block messages, and call the common SHA-2 value H_2 . Now use H_2 as an IV, search for a collision of single-block messages and five-block messages, and call the common SHA-2 value H_3 . With these three colliding pairs, you can construct a message with anywhere from three to ten blocks with a hash of H_3 . If you keep doing this up to H_8 , you will be able to construct a message of any length from 8 to 263 blocks long with hash H_8 (by concatenating the correct-length messages from the collisions). Now, as in the attack without the length field, you can search for a block that maps from H_8 to any of the intermediate hashes of the message for which you'd like to find a second preimage (other than the last eight). If you can find one, you can construct a message that is the appropriate length with the correct hash.

So the length field in the padding doesn't provide significant protection against any known threat. It does, however, allow for a nicer security proof. In particular, with length padding it's possible to prove that the hash function is collision resistant as long as the compression function is collision resistant.

In contrast, without the length padding, it's possible to create a plausible-looking hash function with an adversarially chosen IV that makes it easy to find collisions in the hash function

without finding collisions in the compression function. To do this, use a Davies-Meyer compression function (Figure 5-5) and set the IV to the decryption of 0, using some message block M as a key. Then any message of the form $M|X$ will have the same hash as $M|M|X$ or $M|M|M|X$ or

5.7.2 SHA-3 Padding Rule

SHA-3 and SHAKE do padding differently than earlier hashes.

- There is no length field. The length field is unnecessary in SHA-3 because an attacker attempting to find a second preimage would have to find a collision on the c -bits, which are set in the SHA-3 standard to be twice as long as the full hash.
- The pad field ends with 1 rather than 0. This prevents someone from constructing two strings where the SHA3-256 value of the first string equals the SHA3-512 value of the second string (this applies to any two SHA3 lengths or SHAKE strengths).
- The SHA-3 specification inserts an extra field between the message and the padding (that we will lump in with the padding) that they call the **domain separator**. This is intended to prevent a SHAKE output from having the same value as a SHA-3 hash. The domain separator for SHA-3 is 01. The domain separator for SHAKE is 1111. They could have accomplished the same thing by using a different initial state, but instead, SHA-3 (and SHAKE) always start with the initial state equal to zero. One might wonder why they didn't use a single bit to distinguish SHA-3 and SHAKE, but they are leaving room in the domain separator for distinguishing potential additional uses of KECCAK.

The padding after the domain separator consists of $100\cdots001$, where the number of 0s is between 0 and $r-1$, inclusive. (Remember the state length is 1600 bits, consisting of some number of r -bits, and the remainder being c -bits. Given that c can be between 256 and 1024, r will be between 576 and 1344). (See Figure 5-8 and Figure 5-9.)

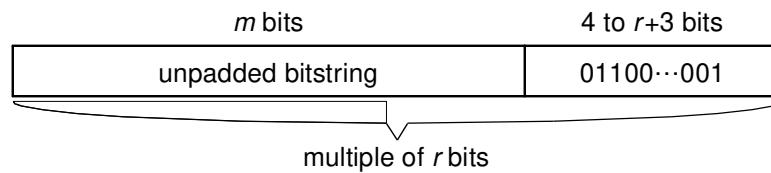


Figure 5-8. Padding for SHA-3

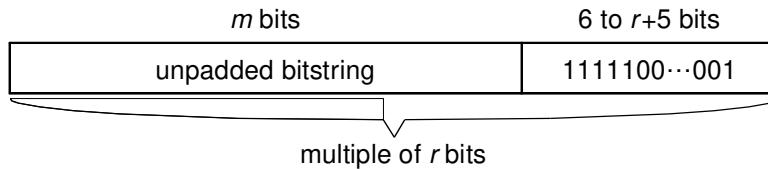


Figure 5-9. Padding for SHAKE

5.8 THE INTERNAL ENCRYPTION ALGORITHMS

Recall that SHA-1 and SHA-2 use the Davies-Meyer construction (Figure 5-5), which incorporates an encryption function. Although the encryption functions used by SHA1 and SHA2 differ, they are both Feistel-like (see §3.3.3 *Feistel Ciphers*) in that each round is reversible. MD4 and MD5 are similar, but with 128 bits instead of 160 and having fewer rounds. But our description will only be for SHA-1 and SHA-2.

5.8.1 SHA-1 Internal Encryption Algorithm

The exact details are not important (unless you need to implement it, of course), but the structure is rather interesting. SHA-1's hash is 160 bits, so it has a 160-bit intermediate state. It treats the state as five 32-bit words (labeled v_0, v_1, v_2, v_3, v_4 in the diagram). We'll call the five words in the output of a round w_0, w_1, w_2, w_3, w_4 . (They become the v_i s for the next round.) (See Figure 5-10.)

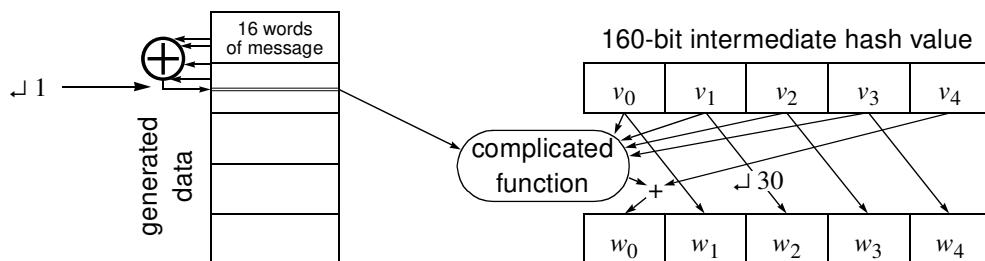


Figure 5-10. Inner Loop of SHA-1—80 Iterations per Block

The internal encryption algorithm inputs a 512-bit chunk of the message that is used to generate eighty 32-bit per-round keys (because there are eighty rounds, and each round uses a 32-bit key). It treats the 5-word value (v_0, v_1, v_2, v_3, v_4) as the quantity to be encrypted.

The first sixteen round keys are simply the sixteen 32-bit words of the 512-bit message chunk. After that, the 32-bit key for round i is generated as the \oplus of four previous 32-bit keys ($i-16$, $i-14$, $i-8$, and $i-3$), and then rotated left one bit.

Each of the eighty rounds uses a complicated nonreversible function, which takes as input the next 32-bit key and v_0 , v_1 , v_2 , and v_3 . Note how the Feistel-like structure enables each round to be reversible even though the complicated function is not. Notice that starting with the output of a round $(w_0, w_1, w_2, w_3, w_4)$, the input values v_0 , v_1 , v_2 , and v_3 are trivially derived from w_1 , w_2 , w_3 , w_4 by simply being copied into a different position, though when v_1 is copied into the output position w_2 , it is rotated right by 2 bits (or equivalently, rotated left by 30 bits).

So how can we recover v_4 from w_0 , w_1 , w_2 , w_3 , w_4 ? We know all the inputs to the complicated function (the 32-bit round key, v_0 , v_1 , v_2 , and v_3) so we can calculate the complicated function in the forward direction, and if we subtract the result from w_0 , we will get v_4 .

If you are curious about the insides of the complicated function, it takes as input the 32-bit quantities v_0 , v_1 , v_2 , v_3 , the round key, and a per-round constant. The per-round constant is the same for the first twenty rounds, then a different constant is used in the next twenty rounds, and so on, so there are four different constants. And to show nothing nefarious was done by choosing the constants, the four constants are the largest integer smaller than 2^{30} times the square root of 2, 3, 5, and 10, respectively.

The complicated function adds v_0 (but rotated left 5 bits), 32 bits of key, the 32-bit constant, and then a round-specific function of v_1 , v_2 , and v_3 that is the same for each set of twenty rounds, but then changes to a different function for the next twenty rounds. The round-specific function is either a bitwise \oplus of v_1 , v_2 , and v_3 , or a bitwise majority, or a bitwise selection. Majority means that if two or three input bits are 1, then the output bit is 1, else the output bit is 0. Selection means that if the bit in v_1 is 0, then the output is the corresponding bit in v_2 ; otherwise, the output is the corresponding bit in v_3 .

5.8.2 SHA-2 Internal Encryption Algorithm

SHA-2 is actually a family of algorithms producing different hash sizes. SHA2-512 differs from SHA2-256 in the number of rounds in the internal encryption algorithm (sixty-four for SHA2-256, eighty for SHA2-512), and the sizes of the blocks are doubled in SHA2-512. The intermediate state in both consists of eight words $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$ with the state at the output of the round being eight words $(w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7)$. SHA2-256 uses 32-bit words; SHA2-512 uses 64-bit words. SHA2-256 processes 512-bit chunks at each stage; SHA2-512 processes 1024-bit chunks. (See Figure 5-11.)

The key schedule takes the 16-word message block and expands it with 48 more words (for SHA2-256) or 64 more words (for SHA2-512).

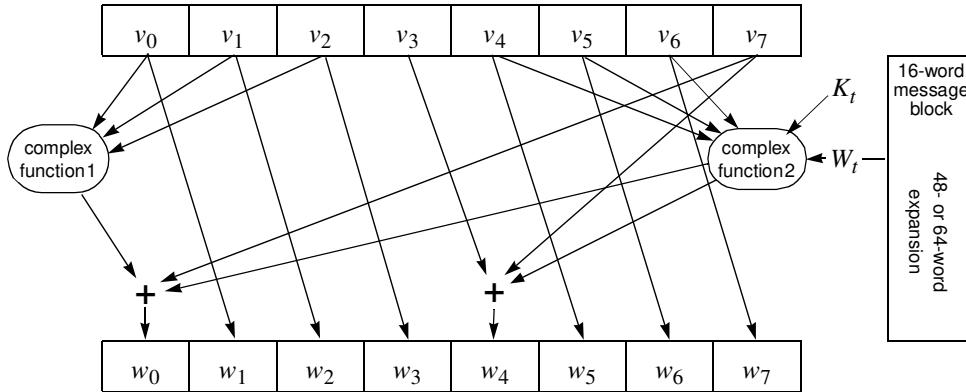


Figure 5-11. Inner Loop of SHA-2—64 or 80 Iterations per Block

Now, referring to Figure 5-11, just as with SHA-1, the Feistel-like structure allows reversing a round, *i.e.*, computing $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$ if you know $w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7$ and the per-round key. Note that all the v_i s other than v_3 and v_7 are easily derived from the w_i s, because $v_0, v_1, v_2, v_4, v_5, v_6$ are simply copied (shifted by a word). The tricky part is deriving v_3 and v_7 .

There are two nonreversible complex functions. Complex function 1 takes as input v_0, v_1, v_2 . We know all those inputs, so we can compute complex function 1.

Complex function 2 takes as input v_4, v_5, v_6 , which we also know, and the per-round key, and a per-round constant. So we can also compute complex function 2.

Now we can compute v_7 by subtracting the outputs of the two complex functions from w_0 . Now that we know v_7 , we can compute v_3 by subtracting v_7 and the output of complex function 2 from w_4 .

If you're still awake, you'll notice that we haven't described what's inside the two complex functions. As with the SHA-1 functions, they are easy to compute (constructed out of \oplus , majority, rotation, +, and selection). Each complex function takes three of the v_i s as input, and complex function 2 adds in the per-round constant K_t and the round key W_t .

5.9 SHA-3 f Function (Also Known as KECCAK-f)

In §5.6.2 *Construction of SHA-3*, we described most of the operation of the SHA-3 (and SHAKE) families. We did not say what happens inside the function f (see Figure 5-6). The f function is a one-to-one mapping of 1600-bit inputs to 1600-bit outputs. In theory, this could be done with a single table lookup in a table with 2^{1600} entries, but that would be an impractically large table.

As discussed in §3.7.2, a common component of both secret key encryption algorithms and hashes consists of alternating operations (usually called layers), where in each round, the input bits are partitioned into small groups, and each group is operated on with a nonlinear operation (often referred to as an S-box). Then a linear layer spreads the bits around and mixes them with one another so that the output bits of each S-box affect the inputs of multiple S-boxes in the next round. After enough rounds, every bit will influence all the bits of output.

The design philosophy of the f permutation in SHA-3 is broadly similar to that of AES. In fact, one of the designers of AES, Joan Daemen, was also on the KECCAK design team. The function f is a one-to-one mapping of 1600-bit inputs to 1600-bit outputs, using twenty-four rounds each comprising one substitution (nonlinear) operation and four linear operations. All these operations are reversible, so it's easy to show f is one-to-one, which makes analysis of the algorithm easier.

The function f is called each time the next r -bit block of message is processed (in the absorption phase) or each time r bits are output (during the squeezing phase). The 1600 bits can be visualized as a three-dimensional array of $5 \times 5 \times 64$ bits. The SHA-3 specification [NIST15c] has lots of pretty pictures, intended to help people visualize what the bit transformations are. f is made up of twenty-four nearly identical rounds. Each round performs five bit-scrambling operations, known as θ (theta), ρ (rho), π (pi), χ (chi), and ι (iota), which are done in that order.

The three dimensions in the $5 \times 5 \times 64$ array of bits are known as rows, columns, and lanes. The rows and columns are five bits long, and the lanes are 64 bits long. Why are the bits organized this way? The reason is that 64-bit processors naturally organize data into 64-bit words, so the obvious way to implement this data structure is as a 5×5 array of words (where each lane is represented by a 64-bit word). In this arrangement, all the component functions of the round can be efficiently implemented using common CPU instructions—in particular, bitwise \oplus (exclusive or), bitwise \wedge (and), and bit rotation operations.

Let's look at the five component functions (θ , ρ , π , χ , and ι) in order.

θ is probably the least intuitive of the five operations. A parity bit is computed for each column by \oplus ing together the five bits in the column, and these parity bits are \oplus 'd into all the bits in two adjacent columns. The point of this operation is to increase the number of bits that each bit can affect by the end of the round. θ is a linear transformation since each output bit is the \oplus of eleven specific input bits. The fact that these eleven bits are all in different lanes and all in different rows helps supplement the mixing provided by the other functions.

In contrast, the ρ and π operations (also linear) really are what we would call bit shuffles (since each output bit is determined by a single input bit). They do not change any bits; they just rearrange them. The ρ operation rotates each of the 25 lanes, each by a different number of bits. The Greek letter ρ was presumably chosen because it sounds like the first syllable of “rotate”. The π operation takes each of the 25 bits in each 5×5 slice, and moves it to a new specified location within the slice. Together, ρ and π thoroughly mix the bits around in all three dimensions, though the 64 bits in a lane remain together in a lane (albeit rotated).

χ is the only nonlinear operation. It acts independently on each of the 320 5-bit rows, so it is an S-box with 5-bit inputs and outputs. Each bit in the row is affected by the two bits that follow according to the formula $b_1 = b_1 \oplus (\sim b_2 \wedge b_3)$. In processing bits at the end of the row, the row is considered to wrap around. Since this S-box performs exactly the same operations on each of the 64 5×5 sheets in the 1600-bit state, it can be implemented efficiently by performing bitwise \wedge and \oplus operations on the 25 lanes of the state. This implementation strategy is called **bit-slicing** the S-box layer. This stands in contrast to the design strategy for the AES S-box. The AES S-box was initially intended to be implemented by table lookup, but it was soon found that due to the behavior of CPU caches, a table lookup took a variable amount of time in a way that can allow side-channel attackers to get secret information by carefully observing how long it took the encryptor and decryptor to perform cryptographic operations.

The τ function uses a 7-bit per-round constant. It \oplus s those bits into seven particular bits in the 1600-bit structure. The value of this per-round constant is the only difference between the operations performed in the twenty-four rounds of the f permutation. The τ function is linear.

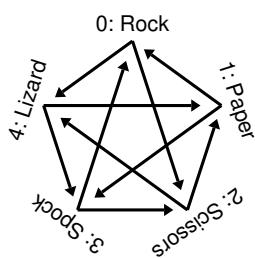
For the exact bit transformations of the five functions, we refer the reader to [NIST15c]. The characteristics of these functions are:

- With θ , each bit affects ten nearby bits.
- With π and ρ , bits are only rearranged and do not affect other bits.
- With χ , each bit affects only two nearby bits and does so with probability only $\frac{1}{2}$.
- τ touches at most seven bits in the whole structure in each round, and usually only about $\frac{1}{2}$ of those bits are flipped in a round. Which bits are flipped depends only on the round number.

If all the input bits were 0s, τ is the only function that can create any 1s, and it can create at most seven of them in any given round. It may seem surprising that the mixing is so thorough in a mere twenty-four rounds, but if each bit can affect 32 others in each round, the compounding adds up. In fact, nine rounds are enough to avoid all known attacks costing less than 2^{512} operations.

5.10 HOMEWORK

1. Draw the Merkle tree associated with a file containing eight data blocks (b_1, b_2, \dots, b_8). Which items do you need to know in order to verify that b_3 has not been modified? If you need to modify b_3 , which items in the Merkle tree need to be modified?
2. Suppose Alice and Bob want to play *Rock Paper Scissors Lizard Spock** (Figure 5-12).



- Scissors cuts Paper
- Paper covers Rock
- Rock crushes Lizard
- Lizard poisons Spock
- Spock smashes Scissors
- Scissors decapitates Lizard
- Lizard eats Paper
- Paper disproves Spock
- Spock vaporizes Rock
- Rock crushes Scissors

Figure 5-12. Rock Paper Scissors Lizard Spock

Suppose they are talking on walkie talkies, where they have to take turns talking. If Alice tells Bob her choice, Bob can simply make a choice that beats Alice's choice. What protocol can they use so that neither of them can cheat?

3. Why do SHA-1 and SHA-2 require padding for messages that are already a multiple of 512 bits?
4. Open-ended project: Implement one or more of the hash algorithms and test how “random” the output appears. For example, test the percentage of 1 bits in the output or test how many bits of output change with minor changes in the input. Also, design various simplifications of the hash functions (such as reducing the number of rounds) and see how these change things.
5. What are the minimal and maximal amounts of padding that would be required in each of the hash functions?
6. In section §5.7.1, we stated that without a length field and if the attacker can choose the IV, “any message of the form $M|X$ will have the same hash as $M|M|X$ or $M|M|M|X$ ”. Assume a hash that follows the Davies-Meyer construction (Figure 5-5) and that the attacker can choose the IV. Explain why the attacker can find an IV and a block M such that repeating M will result in collisions. For example, what does the IV need to be? What will the output of the encrypt function be? What will be the input of the second encrypt function be?

**Rock Paper Scissors Spock Lizard* was invented by Sam Kass and Karen Bryla. After it was popularized by the TV show *Big Bang Theory*, which called it *Rock Paper Scissors Lizard Spock*, most everyone uses that name.

7. Assume a secure 128-bit hash function and a 128-bit value d . Suppose you want to find a message that has a hash equal to d . Given that there are many more 2000-bit messages that map to a particular 128-bit hash than 1000-bit messages, would you theoretically have to test fewer 2000-bit messages to find one that has a hash of d than if you were to test 1000-bit messages?
8. For you statistics fans, calculate the mean and standard deviation of the number of 1 bits in a randomly chosen 256-bit number.
9. For purposes of this exercise, we will define **random** as having all elements equally likely to be chosen. So a function that selects a 100-bit number will be random if every 100-bit number is equally likely to be chosen. Using this definition, if we look at the function $x+y \bmod 2^{100}$, then the output will be random if at least one of x and y is random. For instance, y can always be 51, and yet the output will be random if x is random. For the following functions, find sufficient conditions for x , y , and z under which the output will be random:
 - $\sim x$
 - $x \oplus y$
 - $x \vee y$
 - $x \wedge y$
 - The choice function: $\text{Ch}(x,y,z) = (x \wedge y) \vee (\sim x \wedge z)$
 - The majority function: $\text{Maj}(x,y,z) = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$
 - The parity function: $\text{Parity}(x,y,z) = x \oplus y \oplus z$
 - $y \oplus (x \vee \sim z)$
10. Prove that the function $(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ and the function $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ are equivalent. (Sorry—this isn't too relevant to cryptography, but we'd stumbled on two different versions of the majority function in different documentation, and we had to think about it for a bit to realize they were the same. We figured you should have the same fun.)
11. In §5.4.10 *Computing a MAC with a Hash*, we describe the append attack and listed the algorithms MD4, MD5, SHA-1, SHA2-256, and SHA2-512 as being vulnerable to this attack. Why aren't SHA2-224 and SHA2-384 vulnerable to this attack?
12. Show how to compute a preimage of a hash in a modified form of SHA-3, where there are no c -bits. In other words, assume $r=1600$ and $c=0$, but otherwise, the algorithm is the same.
13. Show how in SHA-3 the work factor for finding a preimage is dependent on the size of c .
14. Suppose Alice and Bob share a key K_{AB} , and Alice uses, as a MAC for message M to Bob, $\text{SHA-3}(M|K_{AB})$. Would this be vulnerable to the append attack discussed in §5.4.10?
15. In §5.8.1 *SHA-1 Internal Encryption Algorithm*, we mention three bitwise operations, \oplus , majority, and selection. Write the truth table for these three functions assuming the three

inputs are each one bit long. Now what would the output be if the inputs a , b , and c are 32 bits long, in these cases:

- Each of a , b , c is all 0s
- Each of a , b , c is all 1s
- a , b , and c are random numbers (meaning that for each of them, about half the bits are 0)
What is the probability of 0 *versus* 1 in a specified bit of the output under each of these functions?

6

FIRST-GENERATION PUBLIC KEY ALGORITHMS

6.1 INTRODUCTION

This chapter describes the most common public key algorithms that have been deployed as of 2022 and that will probably remain the most common public key algorithms in use for several years. As we will show in Chapter 7 *Quantum Computing*, if someone could build a quantum computer of sufficient size, the algorithms in this chapter would no longer be secure. The world will soon be converting to different public key algorithms (see Chapter 8 *Post-Quantum Cryptography*). But the current algorithms, the focus of this chapter, are widely deployed and fascinating to understand.

Public key algorithms are a motley crew. All the hash algorithms do the same thing—they take a message and perform an irreversible transformation on it. All the secret key algorithms do the same thing—they take a block and encrypt it in a reversible way, and there are modes (Chapter 4 *Modes of Operation*) to convert the block ciphers into message ciphers. But public key algorithms look very different from each other not only in how they perform their functions but in what functions they perform. The post-quantum algorithms are also public key algorithms, but we will defer discussion of them until after we've discussed quantum computers in Chapter 7 *Quantum Computing*. In this chapter, we'll describe:

- RSA, which does encryption and digital signatures
- ElGamal and DSA and ECDSA (elliptic curve DSA), which do digital signatures
- Diffie-Hellman and ECDH (elliptic curve Diffie-Hellman), which establish a shared secret and can also do encryption

The thing that all public key algorithms have in common is the concept that a principal has a pair of related quantities, one private and one public.

As we discussed in §2.7 *Numbers*, cryptographic algorithms require a type of mathematics in which numbers can be represented exactly and with a guaranteed upper limit on the number of bits required to represent them. Most of them use modular arithmetic. We introduced modular arithmetic in §2.7.1 *Finite Fields*, but we'll go into a bit more detail here.

6.2 MODULAR ARITHMETIC

*There was a young fellow named Ben
Who could only count modulo ten.
He said, “When I go
Past my last little toe
I shall have to start over again.”*

—Anonymous

Modular arithmetic uses the non-negative integers less than some positive integer n , performs ordinary arithmetic operations such as addition and multiplication, and then replaces the result with its remainder when divided by n . The result is said to be **modulo n** or **mod n** . When we write “ $x \bmod n$ ”, we mean the remainder of x when divided by n . Sometimes we’ll leave out “ $\bmod n$ ” when it’s clear from context.

6.2.1 Modular Addition

Let’s look at mod 10 addition. $3 + 5 = 8$, just like in regular arithmetic. The answer is already between 0 and 9. In regular arithmetic, $7 + 6 = 13$, but the mod 10 answer is 3. Basically, one can perform mod 10 arithmetic by using the last digit of the answer. For example,

$$5 + 5 = 0 \quad 3 + 9 = 2 \quad 2 + 2 = 4 \quad 9 + 9 = 8$$

Let’s look at the mod 10 addition table (Figure 6-1).

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Figure 6-1. Addition Modulo 10

Addition of a constant mod 10 can be used as a scheme for encrypting digits, in that it maps each decimal digit to a different decimal digit in a way that is reversible; the constant is our secret key.

It's not a *good* cipher, of course, but it is a cipher. (It's actually a Caesar cipher.) Decryption would be done by subtracting the secret key modulo 10, which is an easy operation—just do ordinary subtraction, and if the result is less than 0, add 10.

Just like in regular arithmetic, subtracting x can be done by adding $-x$, also known as the **additive inverse** of x . The additive inverse of x is the number you'd add to x to get 0. For example, 4's additive inverse will be 6, because in mod 10 arithmetic $4 + 6 = 0$. If the secret key were 4, then to encrypt we'd add 4 (mod 10), and to decrypt we'd add 6 (mod 10).

6.2.2 Modular Multiplication

Now let's look at the mod 10 multiplication table (Figure 6-2).

\times	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

Figure 6-2. Multiplication Modulo 10

Multiplication by 1, 3, 7, or 9 works as a cipher (again, not a secure cipher), because it performs a one-to-one substitution of the digits. But multiplication by any of the other numbers will not work as a cipher. For instance, if you tried to encrypt by multiplying by 5, half the numbers would encrypt to 0, and the other half would encrypt to 5. You've lost information. You can't decrypt the ciphertext 5, since the plaintext could be any of $\{1, 3, 5, 7, 9\}$. So multiplication mod 10 can be used for encryption provided that you choose the multiplier wisely. But how do you decrypt? Well, just like with addition, where we undid the addition by adding the additive inverse, we'll undo the multiplication by multiplying by the multiplicative inverse. In ordinary (real or rational numbers) arithmetic, x 's multiplicative inverse is $1/x$. If x is an integer, then its multiplicative inverse is a fraction. In modular arithmetic, though, the only numbers that exist are integers. The **multiplicative inverse** of x (written x^{-1}) is the number by which you'd multiply x to get 1. Only the numbers $\{1, 3, 7, 9\}$ have multiplicative inverses mod 10. For example, 7 is the multiplicative inverse of 3. So encryption could be performed by multiplying by 3, and decryption could be performed by multiplying by 7. 9 is its own inverse. And 1 is its own inverse. Multiplication mod n is not a secure

cipher, but it works in the sense that we can scramble the digits by multiplying by x and get back to the original digits by multiplying by x^{-1} (assuming the “key” is a number with a multiplicative inverse mod n).

It is by no means obvious how you find a multiplicative inverse in mod n arithmetic, especially if n is very large. For instance, if n was a 100-digit number, you would not be able to do a brute-force search for an inverse. But it turns out there is an algorithm (see §2.7.5 *Euclidean Algorithm*) that will efficiently find multiplicative inverses mod n . Given x and n , it finds the number y such that $xy \bmod n = 1$ (if there is one).

What’s special about the numbers $\{1, 3, 7, 9\}$? Why is it they’re the only ones, mod 10, with multiplicative inverses? The answer is that those numbers are all relatively prime to 10. **Relatively prime** means they do not share any common factors other than 1. For instance, the largest integer that divides both 9 and 10 is 1. The largest integer that divides both 7 and 10 is 1. In contrast, 6 is not one of $\{1, 3, 7, 9\}$, and it does not have a multiplicative inverse mod 10. It’s also not relatively prime to 10 because 2 divides both 10 and 6. In general, when we’re working mod n , all the numbers relatively prime to n will have multiplicative inverses, and none of the other numbers will. Mod n multiplication by any number x relatively prime to n will work as a cipher because we can multiply by x to encrypt, and then multiply by x^{-1} to decrypt. Again let us hasten to reassure you that we’re not claiming it’s a *good* cipher in the sense of being secure. What we mean by its being a cipher is that we can modify the information through one algorithm (multiplication by $x \bmod n$) and then reverse the process (by multiplying by $x^{-1} \bmod n$).

How many numbers less than n are relatively prime to n ? Why would anyone care? Well, it turns out to be so useful that it’s been given its own notation— $\phi(n)$. ϕ is called the **totient function**, supposedly from *total* and *quotient*. How big is $\phi(n)$? If n is prime, then all the integers $\{1, 2, \dots, n-1\}$ are relatively prime to n , so $\phi(n) = n-1$. If n is a product of two distinct primes, say p and q , then there are $(p-1)(q-1)$ numbers relatively prime to n , so $\phi(n) = (p-1)(q-1)$. Why is that? Well, there are $n = pq$ total numbers in $\{0, 1, 2, \dots, n-1\}$, and we want to exclude those numbers that aren’t relatively prime to n . Those are the numbers that are either multiples of p or of q . There are p multiples of q less than pq and q multiples of p less than pq . So there are $p+q-1$ numbers less than pq that aren’t relatively prime to pq . (We can’t count 0 twice!). Thus $\phi(pq) = pq - (p+q-1) = (p-1)(q-1)$.

6.2.3 Modular Exponentiation

Modular exponentiation is again just like ordinary exponentiation. Once you get the answer, you divide by n and get the remainder. For instance, $4^6 = 6 \bmod 10$ because $4^6 = 4096$ in ordinary arithmetic, and $4096 = 6 \bmod 10$. Let’s look at the exponentiation table mod 10. We are purposely putting in extra columns because in exponentiation, $x^y \bmod n$ is not the same as $x^{y+n} \bmod n$. For instance, $3^1 = 3 \bmod 10$, but $3^{11} = 7 \bmod 10$ (it’s 177147 in ordinary arithmetic).

Let's look at mod 10 exponentiation (Figure 6-3).

x^y	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	2	4	8	6	2	4	8	6	2	4	8	6
3	1	3	9	7	1	3	9	7	1	3	9	7	1
4	1	4	6	4	6	4	6	4	6	4	6	4	6
5	1	5	5	5	5	5	5	5	5	5	5	5	5
6	1	6	6	6	6	6	6	6	6	6	6	6	6
7	1	7	9	3	1	7	9	3	1	7	9	3	1
8	1	8	4	2	6	8	4	2	6	8	4	2	6
9	1	9	1	9	1	9	1	9	1	9	1	9	1

Figure 6-3. Exponentiation Modulo 10

Note that exponentiation by 3 would act as an encryption of the digits in that it rearranges all the digits. Exponentiation by 2 would not, because both 2^2 and 8^2 are 4 mod 10.

How would you decrypt if exponentiation was used to encrypt? Is there an exponentiative inverse like there is a multiplicative inverse? Just like with multiplication, the answer is *sometimes*. (Note we invented the term *exponentiative inverse*, because it's useful, it's obvious what it means, and there really is no other word. Mathematicians might wince if you use the term, though.)

6.2.4 Fermat's Theorem and Euler's Theorem

Fermat's theorem states that if p is prime and $0 < a < p$, $a^{p-1} \equiv 1 \pmod{p}$. Euler generalized Fermat's theorem to non-prime moduli. **Euler's theorem** states that for any a relatively prime to n , $a^{\phi(n)} \equiv 1 \pmod{n}$.

The implication is that if we can find two numbers e and d that are multiplicative inverses mod $\phi(n)$, they will be exponentiative inverses mod n , meaning that if we exponentiate x mod n by e , and then exponentiate the result mod n by d , we'll wind up with x mod n .

The Euclidean algorithm (§2.7.5) allows us to calculate multiplicative inverses mod n . There is no efficient algorithm for finding exponentiative inverses mod n directly. However, if we know how to factor n , we will be able to calculate $\phi(n)$, and we can then find multiplicative inverses mod $\phi(n)$. Therefore, if we know how to factor n , we will be able to calculate exponentiative inverses mod n , but if we do not know how to factor n , we will not be able to calculate $\phi(n)$, and we will not be able to efficiently find an exponentiative inverse mod n .

Armed with this knowledge, let's look at RSA.

6.3 RSA

RSA is named after its inventors, Rivest, Shamir, and Adleman. It is a public key cryptographic algorithm that does encryption, decryption, signature generation, and signature verification. The key length is variable. Anyone using RSA can choose a long key for enhanced security or a short key for efficiency. The most commonly used key length for RSA is 2048 bits.

The block size in RSA (the chunk of data to be encrypted) is also variable. The plaintext block must be smaller than the key length. The ciphertext block will be the length of the key. RSA is much slower to compute than popular secret key algorithms like AES. As a result, RSA is not used for encrypting long messages. Instead, RSA is used to encrypt a secret key, and then secret key cryptography is used to actually encrypt the message. Likewise, instead of signing a large message, RSA is used to sign a hash of the message.

6.3.1 RSA Algorithm

First, you need to generate a public key and a corresponding private key. Choose two large primes p and q (probably around 1024 bits each). Multiply them together and call the result n . The factors p and q will remain secret. The modulus n will be part of your public key, so people will know n . However, you won't tell anybody the factors p and q , and it's practically impossible to factor numbers as large as a 2048-bit n .

You've now chosen the modulus n . To generate the rest of your public key, choose a number e that is relatively prime to $\phi(n)$. Since you know p and q , you can easily compute $\phi(n)$ —it's $(p-1)(q-1)$. Your public key is the pair $\langle e, n \rangle$.

To generate your private key, use the extended Euclidean algorithm to find the number d that is the multiplicative inverse of e mod $\phi(n)$. Your private key is $\langle d, n \rangle$. Someone that sees your public key $\langle e, n \rangle$, will not be able to calculate your private key unless they can factor n .

Note that there is an optimization. When Alice is computing her two exponents e and d , she can compute e 's multiplicative inverse, d , mod $\lambda(n)$ rather than $\phi(n)$. The quantity $\lambda(n)$ is the smallest integer x such that for every integer a with $1 < a < n$, $a^x \bmod n = 1$. With an RSA number n being the product of two odd primes p and q , $\lambda(n)$ will be at most $(p-1)(q-1)/2$ and might be smaller if $p-1$ and $q-1$ have other common factors. The d she'd compute using $\phi(n)$ may be different from the d she'd compute using $\lambda(n)$, but either d will be an exponentiative inverse mod n of her chosen exponent e . So that means that for a given e , there will be at least two exponentiative inverses mod n .

To encrypt a quantity m , someone using your public key should compute ciphertext $c = m^e \bmod n$. (Again, the quantity m being encrypted will usually not be an entire message, but will be a single block that contains the AES key that will be used to encrypt a long message.) Only you,

using your private key, will be able to decrypt c , by computing $m = c^d \bmod n$. Also, only you can create a signature s for a quantity m , using your private key, by computing $s = m^d \bmod n$. Anyone can verify your signature by checking that $m = s^e \bmod n$. (Again, you would be signing a hash of a message, and not signing a long message.) That's all there is to RSA. Now, there are some questions we should ask:

- Why does it work? Will decrypting an encrypted message get the original message back?
- Why is it secure? Given e and n , why can't someone easily compute d ?
- Are the operations encryption, decryption, signing, and verifying signatures all sufficiently efficient to be practical?
- How do we find big primes?

6.3.2 Why Does RSA Work?

RSA does arithmetic mod n , where $n=pq$. We know that $\phi(n)=(p-1)(q-1)$. We've chosen d and e such that $de=1 \bmod \phi(n)$. Therefore, by Euler's Theorem, for any x , $x^{de}=x \bmod n$. An RSA encryption consists of taking x and raising it to e . If we take the result and raise it to the d (*i.e.*, perform RSA decryption), we'll get $(x^e)^d$, which equals x^{ed} , which is the same as x . So we see that decryption reverses encryption.

In the case of signature generation, x is first raised to the d power to get the signature, and then the signature is raised to the e power for verification; the result, $x^{de} \bmod n$, will equal x .

6.3.3 Why Is RSA Secure?

We don't know for sure that RSA is secure. We can only depend on the Fundamental Tenet of Cryptography—lots of smart people have been trying to figure out how to break RSA, and they haven't come up with anything yet (§2.1.1 *The Fundamental Tenet of Cryptography*).

The real premise behind RSA's security is the assumption that factoring a big number is hard. The best known factoring methods are really slow. To factor a 2048-bit number with the fastest known technique (the general number field sieve) would take more than 10^{20} MIPS-years. We suspect that a better technique is to wait a few hundred years and *then* use the fastest known technique.

If you can factor quickly, you can break RSA. Suppose you are given Alice's public key $\langle e, n \rangle$. If you could find e 's exponential inverse mod n , then you'd have figured out Alice's private key $\langle d, n \rangle$. How can you find e 's exponential inverse? Alice did it by knowing the factors of n , allowing her to compute $\phi(n)$. She found the number that was e 's multiplicative inverse mod $\phi(n)$. She didn't have to factor n —she started with primes p and q and multiplied them together to get n . You can do what Alice did if you can factor n to get p and q .

We do not know that factoring n is the only way of breaking RSA. We know that breaking RSA (for example, having an efficient means of finding d , given e and n) is no more difficult than factoring [CORM91], but there might be some other means of breaking RSA.

Note that it's possible to misuse RSA. For instance, let's say Bob is holding an auction where bids must be an integral number of dollars and the item being auctioned is worth about \$1000. Alice will bid \$742. Trudy would like to find out what Alice's bid is, so that Trudy can bid just a tiny bit more. Let's say Alice encrypts her bid (742) with Bob's public key. What can Trudy learn from eavesdropping on Alice's encrypted message?

Trudy can't decrypt something encrypted with Bob's public key, but she can encrypt using his public key. She can try all thousand possible bids that Alice might be making, encrypt each with Bob's public key, and find the one that matches the ciphertext that Alice sent.

The defense that prevents Trudy from guessing and verifying plaintext is that Alice should concatenate her bid with a large random number, say 128 bits long. Then instead of a thousand possible messages for Trudy to check, there are 1000×2^{128} , and checking that many messages is computationally infeasible.

6.3.4 How Efficient Are the RSA Operations?

The operations that need to be routinely performed with RSA are encryption, decryption, generating a signature, and verifying a signature. These need to be very efficient because they will be used a lot. Finding an RSA key (which means picking appropriate n , d , and e) also needs to be reasonably efficient, but it isn't as performance critical as the other operations since it is done less frequently. As it turns out, finding an RSA key is substantially more computationally intensive than using one.

6.3.4.1 Exponentiating with Big Numbers

Encryption, decryption, signing, and verifying signatures all involve taking a large number m , raising it to a large power x , and finding the remainder mod a large number n . For the sizes the numbers have to be for RSA to be secure, these operations would be prohibitively expensive if done in the most straightforward way (multiplying m by itself x times, and then reducing mod n). The following will illustrate some tricks for doing the calculation faster.

Suppose you want to compute $123^{54} \bmod 678$. The straightforward thing to do (assuming your computer has a multiple-precision arithmetic package) is to multiply 123 by itself 54 times, getting a really big product (about 100 digits), and then to divide by 678 to get the remainder. A computer could do this with ease, but for RSA to be secure, the numbers must be on the order of 600 digits. Raising a 600-digit number to a 600-digit power by this method would exhaust the

capacity of all existing computers for more than the expected life of the universe and thus would not be cost-effective.

Luckily, you can do better than that. If you do the modular reduction after each multiplication, it keeps the number from getting really ridiculous. To illustrate:

$$\begin{aligned}123^2 &= 123 \times 123 = 15129 = 213 \bmod 678 \\123^3 &= 123 \times 213 = 26199 = 435 \bmod 678 \\123^4 &= 123 \times 435 = 53505 = 621 \bmod 678\end{aligned}$$

This reduces the problem to 54 small multiplications and 54 small divisions, but it would still be unacceptable for exponents of the size used with RSA.

However, there is a much more efficient method. To raise a number m to an exponent that is a power of 2, say 32, you could start with m and multiply by m 31 times, which is reasonable if you have nothing better to do with your time. A much better scheme is to first square m , then square the result, and so on. Then you'll be done after five squarings (five multiplications and five divisions):

$$\begin{aligned}123^2 &= 123 \times 123 = 15129 = 213 \bmod 678 \\123^4 &= 213 \times 213 = 45369 = 621 \bmod 678 \\123^8 &= 621 \times 621 = 385641 = 537 \bmod 678 \\123^{16} &= 537 \times 537 = 288369 = 219 \bmod 678 \\123^{32} &= 219 \times 219 = 47961 = 501 \bmod 678\end{aligned}$$

What if you're not lucky enough to be raising something to a power of 2? First note that if you know what 123^x is, then it's easy to compute 123^{2x} —you get that by squaring 123^x . It's also easy to compute 123^{2x+1} —you get that by multiplying 123^{2x} by 123. Now you use this observation to compute 123^{54} .

Well, 54 is 110110_2 (represented in binary). You'll compute 123 raised to a sequence of powers— $1_2, 11_2, 110_2, 1101_2, 11011_2, 110110_2$. Each successive power concatenates one more bit of the desired exponent. And each successive power is either twice the preceding power or one more than twice the preceding power:

$$\begin{aligned}123^2 &= 123 \times 123 = 15129 = 213 \bmod 678 \\123^3 &= 123^2 \times 123 = 213 \times 123 = 26199 = 435 \bmod 678 \\123^6 &= (123^3)^2 = 435^2 = 189225 = 63 \bmod 678 \\123^{12} &= (123^6)^2 = 63^2 = 3969 = 579 \bmod 678 \\123^{13} &= 123^{12} \times 123 = 579 \times 123 = 71217 = 27 \bmod 678 \\123^{26} &= (123^{13})^2 = 27^2 = 729 = 51 \bmod 678 \\123^{27} &= 123^{26} \times 123 = 51 \times 123 = 6273 = 171 \bmod 678 \\123^{54} &= (123^{27})^2 = 171^2 = 29241 = 87 \bmod 678\end{aligned}$$

The idea is that squaring is the same as doubling the exponent, which, in turn, is the same as shifting the exponent left by one bit. And multiplying by the base is the same as adding one to the exponent.

In general, to perform exponentiation of a base to an exponent, you start with your value set to 1. As you read the exponent in binary bit by bit from high-order bit to low-order bit, you square your value, and if the bit is a 1, you then multiply by the base. You perform modular reduction after each operation to keep the intermediate results small.

By this method you've reduced the computation of 123^{54} to eight multiplications and eight divisions. More importantly, the number of multiplications and divisions rises linearly with the length of the exponent in bits rather than with the value of the exponent itself.

RSA operations using this technique are sufficiently efficient to be practical.

Note that as we explained in §2.7.3 *Avoiding a Side-Channel Attack*, this implementation would allow a side-channel attack, because the implementation would behave differently on each bit of the exponent, depending on whether the bit was a 0 or a 1. An implementation should avoid providing a side channel by behaving the same way on each bit, as explained in §2.7.3.

6.3.4.2 Generating RSA Keys

Most uses of public key cryptography do not require frequent generation of RSA keys. If generation of an RSA key is only done, for instance, when an employee is hired, then it need not be as efficient as the operations that use the keys. However, it still has to be reasonably efficient.

6.3.4.2.1. Finding Big Primes p and q

There is an infinite supply of primes. However, they thin out as numbers get bigger and bigger. The probability of a randomly chosen number n being prime is approximately $1/\ln n$. The natural logarithm function, \ln , rises linearly with the size of the number represented in digits or bits. For a ten-digit number, there is about one chance in 23 of it being prime. For a 300-digit number (a size of prime that would be useful for RSA), there is about one chance in 690.

So, we'll choose a random odd number and test if it is prime. On the average, we'll only have to try 690 of them before we find one that is a prime. So, how do we test if a number n is prime?

One naive method is to divide n by all numbers $\leq \sqrt{n}$ and see if there is always a nonzero remainder. The problem is that it would take several universe lifetimes (with numbers the size used in RSA) to verify that a candidate is prime. We said finding p and q didn't need to be as easy as generating or verifying a signature, but forever is too long.

For a long time, there was no sufficiently fast way for absolutely determining that a number of this size is prime. Mathematicians have now invented such an algorithm [AGRA04], but almost no one uses it. That's because there is a much faster test for determining that a number is *almost* certainly prime, and the more time we spend testing a number the more assured we can be that the number is prime.

Recall **Fermat's Theorem**: If p is prime and $0 < a < p$, $a^{p-1} \equiv 1 \pmod{p}$. Does $a^{n-1} \equiv 1 \pmod{n}$ hold even when n is not prime? The answer is—usually not! A primality test, then, for a number n is to pick a number $a < n$, compute $a^{n-1} \pmod{n}$, and see if the answer is 1. If it is not 1, n is certainly not prime. If it is 1, n may or may not be prime. If n is a randomly generated number of about 300 digits, the probability that n isn't prime but $a^{n-1} \pmod{n} = 1$, is less than 1 in 10^{40} [POME81, CORM91]. Most people would decide they could live with that risk of falsely assuming n was prime when it wasn't. The cost of such a mistake would be either that (1) RSA might fail—they could not decrypt a message addressed to them, or (2) someone might be able to compute their private exponent with less effort than anticipated. There aren't many applications where a risk of failure of 1 in 10^{40} is a problem.

But if the risk of 1 in 10^{40} is unacceptable, the primality test can be made more reliable by using multiple values of a . If for any given n , each value of a had a probability of 1 in 10^{40} of falsely reporting primality, a few tests would assure even the most paranoid person. Unfortunately, there exist numbers n that are not prime but which satisfy $a^{n-1} \equiv 1 \pmod{n}$ for all values of a . They are called **Carmichael numbers**. Carmichael numbers are sufficiently rare that the chance of selecting one at random is nothing to lose sleep over. Nevertheless, mathematicians have come up with an enhancement to the above primality test that will detect non-primes (even Carmichael numbers) with high probability and negligible additional computation, so we may as well use it.

The method of choice for testing whether a number is prime is due to Miller and Rabin [RABI80]. We can always express $n-1$ as a power of two times an odd number, say $2^b c$. We can then compute $a^{n-1} \pmod{n}$ by computing $a^c \pmod{n}$ and then squaring the result b times. If the result is not 1, then n is not prime and we're done. If the result is 1, we can go back and look at those last few intermediate squarings. (If we're really clever, we'll be checking the intermediate results as we compute them.) If $a^c \pmod{n}$ is not 1, then one of the squarings took a number that was not 1 and squared it to produce 1. That number is a mod n square root of 1. It turns out that if n is prime, then the only mod n square roots of 1 are ± 1 . Further, if n is not a power of a prime, then 1 has multiple square roots, and all are equally likely to be found by this test. A square root other than ± 1 is known as a *nontrivial square root* of 1. So if you can find a nontrivial square root of n , you know n is not prime. For more on why, see §6.3.4.3 *Why a Non-Prime Has Multiple Square Roots of One*.

So if the Miller-Rabin test finds a square root of 1 that is not ± 1 , then n is not prime. Furthermore, if n is not prime (even if it is a Carmichael number), at least $\frac{3}{4}$ of all possible values of a will fail the Miller-Rabin primality test. By trying many values for a , we can make the probability of falsely identifying n as prime inconceivably small. In actual implementations, how many values of a to try is a trade-off between performance and paranoia.

To summarize, an efficient method of finding primes is:

1. Pick an odd random number n with the desired number of bits.
2. Test n 's divisibility by small primes and go back to step 1 if you find a factor. (Obviously, this step isn't necessary, but it's worth it since it has a high enough probability of catching some non-primes and is much faster than the next step.)
3. Repeat the following until n is proven not prime (in which case go back to step 1) or as many times as you feel necessary to show that n is probably prime:

Pick an a at random and compute $a^c \bmod n$ (where c is the odd number for which $n-1 = 2^b c$). If $a^c \equiv \pm 1 \pmod{n}$, pick a different a and repeat. Otherwise, keep squaring the result mod n until you get a value that equals $\pm 1 \pmod{n}$, or until the exponent is $n-1$. If the value is $+1$, the value you just squared is a square root of 1 other than ± 1 , so the number is definitely not prime. If the exponent is $n-1$, the number is definitely not prime because $a^{n-1} \bmod n \neq 1$. Otherwise, n has passed the primality test for this a , and if you want, you can try another a .

6.3.4.3 Why a Non-Prime Has Multiple Square Roots of One

By the Chinese remainder theorem (§2.7.6), there are four ways that a number y can be a square root of 1 mod n when $n=pq$:

1. y is 1 mod p and 1 mod q . In this case, y will be 1 mod n .
2. y is -1 mod p and -1 mod q . In this case, y will be -1 mod n .
3. y is 1 mod p and -1 mod q . In this case, y will be a nontrivial square root of 1 mod n .
4. y is -1 mod p and 1 mod q . In this case, y will be a nontrivial square root of 1 mod n .

Note that if you find a nontrivial square root of 1 mod n , you can factor n (Homework Problem 9).

6.3.4.3.1. Finding d and e

How do we find d and e given p and q ? As we said earlier, for e we can choose any number that is relatively prime to $(p-1)(q-1)$, and then all we need to do is find the number d such that $ed \equiv 1 \pmod{\phi(n)}$. This we can do with the extended Euclidean algorithm.

There are two strategies one can use to ensure that e and $(p-1)(q-1)$ are relatively prime.

1. After p and q are selected, choose e at random. Test to see if e is relatively prime to $(p-1)(q-1)$. If not, select another e .
2. Don't pick p and q first. Instead, first choose e , then select p and q carefully so that $(p-1)$ and $(q-1)$ are guaranteed to be relatively prime to e . The next section will explain why you'd want to do this.

6.3.4.4 Having a Small Constant e

A rather astonishing discovery is that RSA is no less secure (as far as anyone knows) if e is always chosen to be the same number. And if e is chosen to be small, then the operations of encryption and signature verification become much more efficient. Given that the procedure for finding a $\langle d, e \rangle$ pair is to pick one and then derive the other, it is straightforward to make e be a small constant. This makes public key operations faster while leaving private key operations unchanged. You might wonder whether it would be possible to select small values for d to make private key operations fast at the expense of public key operations. The answer is that you can't. If d were a constant, the scheme would not be secure because d is the secret. If d were small, an attacker could search small values to find d .

Two popular values of e are 3 and 65537.

Why 3? The number 2 doesn't work because it is not relatively prime to $(p-1)(q-1)$ (which must be even because p and q are both odd). 3 can work, and with 3, public key operations require only two multiplications. Using 3 as the public exponent maximizes performance.

As far as anyone knows, using 3 as a public exponent does not weaken the security of RSA if some practical constraints on its use are followed. Most dramatically, if a message m to be encrypted is small—in particular, smaller than $\sqrt[3]{n}$ —then raising m to the power 3 and reducing mod n will simply produce the value m^3 . Anyone seeing such an encrypted message could decrypt it simply by taking a cube root. This problem can be avoided by padding each message with a random number before encryption so that m^3 is always large enough to be guaranteed to need to be reduced mod n .

A second problem with using 3 as an exponent is that if the same message is sent encrypted to three or more recipients, each of whom has a public exponent of 3, the message can be derived from the three encrypted values and the three public keys $\langle 3, n_1 \rangle, \langle 3, n_2 \rangle, \langle 3, n_3 \rangle$.

Suppose a bad guy sees $m^3 \bmod n_1$, $m^3 \bmod n_2$, and $m^3 \bmod n_3$ and knows $\langle 3, n_1 \rangle, \langle 3, n_2 \rangle, \langle 3, n_3 \rangle$. Then by the Chinese Remainder computation, the bad guy can compute $m^3 \bmod n_1 n_2 n_3$. Since m is smaller than each of the n_i 's (because RSA can only encrypt messages smaller than the modulus), m^3 will be smaller than $n_1 n_2 n_3$, so $m^3 \bmod n_1 n_2 n_3$ will just be m^3 . Therefore, the bad guy can compute the ordinary cube root of m^3 (which again is easy if you are a computer) to get m .

Now this isn't anything to get terribly upset about. In practical uses of RSA, the message to be encrypted is usually a key for a secret key encryption algorithm and in any case is much smaller than n . As a result, the message must be padded before it is encrypted. If the padding is randomly chosen (and it should be for a number of reasons), and if it is re-chosen for each recipient, then there is no threat from an exponent of 3 no matter how many recipients there are. The padding doesn't really have to be random—for example, the recipient's ID would work fine.

Finally, an exponent of 3 works only if 3 is relatively prime to $\phi(n)$ (in order for it to have an inverse d). How do we choose p and q so that 3 will be relatively prime to $\phi(n)=(p-1)(q-1)$? Clearly, $(p-1)$ and $(q-1)$ must each be relatively prime to 3. To ensure that $p-1$ is relatively prime

to 3, we want p to be $2 \bmod 3$. That will ensure $p-1$ is $1 \bmod 3$. Similarly we want q to be $2 \bmod 3$. We can make sure that the only primes we select are congruent to $2 \bmod 3$ by choosing a random number, multiplying by 3 and adding 2, and using that as the number we will test for primality. Indeed, we want to make sure the number we test is odd (since if it's even it is unlikely to be prime), so we should start with an odd number, multiply by 3, and add 2. This is equivalent to starting with any random number, multiplying by 6, and then adding 5.

An even more popular value of e is 65537. Why 65537? The appeal of 65537 (as opposed to others of the same approximate size) is that $65537 = 2^{16}+1$, and it is prime. Because its binary representation contains only two 1s, it takes only 17 multiplications to exponentiate. While this is much slower than the two multiplications required with an exponent of 3, it is much faster than the 3072 (on average) required with a randomly chosen 2048-bit value (the typical size of an RSA modulus in practical use today). Also, using the number 65537 as a public exponent largely avoids the problems with the exponent 3.

The first problem with 3 occurs if $m^3 < n$. Unless n is much longer than the 2048 bits in typical use today, there aren't too many values of m for which $m^{65537} < n$, so being able to take a normal 65537th root is not a threat.

The second problem with 3 occurs when the same message is sent to at least 3 recipients. In theory, with 65537 there is a threat if the same message is sent encrypted to at least 65537 recipients. A cynic would argue that under such circumstances, the message couldn't be very secret.

The third problem with 3 is that we have to choose n so that $\phi(n)$ is relatively prime to 3. For 65537, the easiest thing to do is just reject any p or q that is equal to $1 \bmod 65537$. The probability of rejection is very small (2^{-16}) so this doesn't make finding n significantly harder.

6.3.4.5 Optimizing RSA Private Key Operations

There is a way to speed up RSA exponentiations in generating signatures and decrypting (the operations using the private key) by taking advantage of knowledge of p and q . Feel free to skip this section—it isn't a prerequisite for anything else in the book, and it requires more than the usual level of concentration.

In RSA, d and n are 2048-bit numbers, or about 617 digits. p and q are 1024 bits, or about 308 digits. RSA private key operations involve taking some c (usually a 2048-bit number) and computing $c^d \bmod n$. It's easy to say "raise a 2048-bit number to a 2048-bit exponent mod a 2048-bit number", but it's certainly processor intensive, even if you happen to be a silicon-based computer. A way to speed up RSA operations is to do all the computation mod p and mod q , then use the Chinese Remainder Theorem to compute what the answer is mod pq .

So suppose you want to compute $m = c^d \bmod n$. Instead of computing $c^d \bmod n$, you can take $c_p = c \bmod p$ and $c_q = c \bmod q$ and compute $m_p = c_p^d \bmod p$ and $m_q = c_q^d \bmod q$, then use the Chinese Remainder Theorem to convert back to what m would equal mod n , which would give you $c^d \bmod n$. Also, it is not necessary to raise to the d th power mod p , given that d is going to be bigger

than p (by a factor of about q). Since (by Euler's Theorem) any $a^{p-1} \equiv 1 \pmod{p}$, we can take d 's value mod $p-1$ and use that as the exponent instead. In other words, if $d = k(p-1)+r$, then $c^d \pmod{p} = c^r \pmod{p}$.

So, let us compute $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$. Then, instead of doing the expected RSA operation of $m = c^d \pmod{n}$ (which involves 2048-bit numbers), we'll compute both $m_p = c_p^{dp} \pmod{p}$ and $m_q = c_q^{dq} \pmod{q}$ and then compute m from the Chinese Remainder Theorem. To save ourselves work, since we'll be using d_p and d_q a lot (every time we do an RSA private key computation), we'll compute them once and remember them. Similarly, to use the Chinese Remainder Theorem at the end, we need to know $p^{-1} \pmod{q}$ and $q^{-1} \pmod{p}$, so we'll precompute and remember them as well.

All told, instead of one 2048-bit exponentiation, this modified calculation does two 1024-bit exponentiations, followed by two 1024-bit multiplications and a 2048-bit addition. This might not seem like a net gain, but because the exponents are half as long, using this variant makes RSA about twice as fast.

Note that to do these optimizations for RSA operations, we need to know p and q . Someone who is only supposed to know the public key will not know p and q (or else they can easily compute d). Therefore, these optimizations are only useful for the private key operations (decryption and generating signatures). However, that's okay because we can choose e to be a convenient value (like 3 or 65537) so that raising a 2048-bit number to e will be easy enough without the Chinese Remainder optimizations.

6.3.5 Arcane RSA Threats

Any number $x < n$ is a signature of $x^e \pmod{n}$. So it's trivial to forge someone's signature if you don't care what you're signing. The following sections explain the vulnerabilities that PKCS #1 [see §6.3.6 *Public-Key Cryptography Standard (PKCS)*] is attempting to avoid.

The goal is to constrain what is being signed so that a random number has negligible probability of being a valid message. For example, often what is being signed is a hash value, which is sufficiently smaller than an RSA modulus that there is plenty of room for padding the hash before digitally signing it. If the pad in a valid signature is required to include hundreds of bits of a specific constant, it is extremely unlikely that a random number will look like a valid padded hash. An attacker knowing only Alice's public key would have to find a value x that when raised to $e \pmod{n}$ would have valid padding.

Note: RSA deals with large numbers, and there is, unfortunately, more than one way to represent such numbers. In what follows, we have chosen to order the octets left to right from most significant to least significant. This is called **big-endian** format.

6.3.5.1 Smooth Numbers

A **smooth number** is defined as one that is the product of reasonably small primes. There's no absolute definition of a smooth number, since there's no real definition of *reasonably small*. The more compute power the attacker has at her disposal, and the more signatures she has access to, the larger the “reasonably small” primes need to be, to prevent an attacker from taking advantage of this threat.

The threat we are about to describe is known as the **smooth number threat**. It is really only of theoretical interest because of the immense amount of computation, the immense numbers of signed messages that need to be gathered, and the luck involved. However, it costs very little to have an encoding that avoids this threat. The smooth number threat was discovered by Desmedt and Odlyzko [DESM86].

The first observation is that if you have signed m_1 and m_2 , and a bad guy Carol can see your signature on m_1 and m_2 , she can compute your signature on $m_1 \times m_2$, on m_1/m_2 , on m_1^j , and on $m_1^j \times m_2^k$. For instance, if Carol sees $m_1^d \bmod n$ (which is your signature on m_1), then she can compute your signature on m_1^2 by computing $(m_1^d \bmod n)^2 \bmod n$ (see Homework Problem 5).

If Carol collects a lot of your signed messages, she will be able to compute your signature on any message that can be computed from her collection by multiplication and division. If the messages you sign are mostly smooth, there will be a lot of other smooth messages on which she will be able to forge your signature.

Suppose she collects your signatures on two messages whose ratio is a prime. Then she can compute your signature on that prime. If she's lucky enough to get many such message pairs, she can compute your signature on lots of primes, and then she can forge your signature on any message that is the product of any subset of those primes, each raised to any power. With enough pairs, she will be able to forge your signature on any message that is a smooth number.

Actually, Carol does not have to be nearly that lucky. With as few as k signatures on messages that are products of different subsets of k distinct primes, she will be able to isolate the signatures on the individual primes through a carefully chosen set of multiplications and divisions.

The typical thing being signed with RSA is a padded hash. If it is padded with zeroes, it is much more likely to be smooth than is a random mod n number. A random mod n quantity is extremely unlikely to be smooth (low enough probability so that if you are signing random mod n numbers, we can assume Carol would have to have a lot of resources and a lot of luck to find even one smooth number you've signed, and she might need millions of them in order to mount the attack).

Padding on the left with zeroes keeps the padded message digest small and therefore likely to be smooth. Padding on the right with zeroes is merely multiplying the message digest by some power of two, and so isn't any better.

Another tempting padding scheme is to pad on the right with random data. That way, since you are signing fairly random mod n numbers, it is very unlikely that any of the messages you sign

will be smooth, so Carol won't have enough signed smooth messages to mount the threat. However, this leaves us open to the next obscure threat.

6.3.5.2 The Cube Root Problem

Let's say you pad on the right with random data. You chose that scheme so that there is a negligible probability that anything you sign will be smooth. However, if the public exponent is 3, this enables Carol to forge your signature on virtually any message she chooses!

Let's say Carol wants your signature on some message. The hash of that message is h . Carol pads h on the right with zeroes. She then computes its ordinary cube root and rounds up to an integer r . Now she has forged your signature, because $r^e = r^3 = (h \text{ padded on the right with a seemingly random number})$.

6.3.6 Public-Key Cryptography Standard (PKCS)

It is useful to have some standard for the encoding of information that will be signed or encrypted through RSA, so that different implementations can interwork, and so that the various pitfalls with RSA can be avoided. Rather than expecting every user of RSA to be sophisticated enough to know about all the attacks and develop appropriate safety measures through careful encoding, the standard known as **PKCS** recommends encodings. PKCS is actually a set of standards, called PKCS #1 through PKCS #15. There are also two companion documents, *An overview of the PKCS standards*, and *A layman's guide to a subset of ASN.1, BER, and DER*. (**ASN.1** = *Abstract Syntax Notation 1*, **BER** = *Basic Encoding Rules*, and **DER** = *Distinguished Encoding Rules*—aren't you glad you asked?).

There is a newer version of PKCS #1 documented in RFC 8017, but most implementations use the format described here.

The PKCS standards define the encodings for things such as an RSA public key, an RSA private key, an RSA signature, a short RSA-encrypted message (typically a secret key), a short RSA-signed message (typically a hash), and password-based encryption.

The threats that PKCS has been designed to deal with are:

- encrypting guessable messages
- signing smooth numbers
- multiple recipients of a message when $e = 3$
- encrypting messages that are less than a third the length of n when $e = 3$
- signing messages where the information is in the high-order part and $e = 3$

6.3.6.1 Encryption

PKCS #1 defines standards for formatting a message to be encrypted with RSA. RSA is not generally used to encrypt ordinary data. The most common quantity that would be encrypted with RSA is a secret key, and for performance reasons, the actual data would be encrypted with the secret key.

The format normally used is

0	2	at least eight random nonzero octets	0	data
---	---	---	---	------

The data to be encrypted, usually a secret key, is much smaller than the modulus. If it's an AES key, it's 128, 192, or 256 bits. Sometimes it is a seed to be used as input to a PRF (§2.6.3 *Calculating a Pseudorandom Stream from the Seed*).

The top octet is 0, which is a good choice because this guarantees that the message m being encrypted is smaller than the modulus n . (If m were larger than n , decryption would produce $m \bmod n$ instead of m .) Note that PKCS specifies that the high-order octet (not bit!) of the modulus n must be non-zero, and given that the PKCS padding requires the top octet to be zero, this guarantees that the thing being encrypted will be smaller than the modulus.

The next octet is 2, which is the format type. The value 2 is used for a block to be encrypted. The value 1 is used for a value to be signed (see next section).

Each octet of padding is chosen independently to be a random nonzero value. The reason 0 cannot be used as a padding octet is that 0 is used to delimit the padding from the data.

Let's review the RSA threats and see how this encoding addresses them.

- encrypting guessable messages: Since there are at least eight octets of randomly chosen padding, knowing what might appear in the data does not help the attacker who would like to guess the data, encrypt it, and compare it with the ciphertext. The attacker would have to guess the padding as well, and this is infeasible.
- sending the same encrypted message to more than three recipients (assuming 3 is chosen for e): As long as the padding is chosen independently for each recipient, the quantities being encrypted will not be the same.
- encrypting messages that are less than a third the length of n when $e = 3$: Because the second octet is nonzero, the message will be guaranteed to be more than a third the length of n .

6.3.6.2 The Million-Message Attack

There was an attack on SSL (the precursor to TLS) that could have been interpreted as a flaw in an implementation of SSL, but the world has come to see it as a flaw in the PKCS #1 encryption format. There is a PKCS #1 version 2 format that fixes the “flaw”. The attack, published by Daniel Bleichenbacher [BLEI98], is known as the **million-message attack**. It is possible because SSL made some incorrect assumptions about the services PKCS #1 padding provides. This attack allows

an attacker (Trudy) to recover the key used for an Alice-Bob connection but requires Trudy to send a million modified versions of the third message in the Alice-Bob communication.

In the SSL protocol, the client (Alice) sends the server (Bob) a randomly chosen secret key (S), padded according to PKCS #1 and encrypted using RSA. SSL decrypts the value, and, if the padding is correct, sends a response encrypted with S . If after the decryption the padding is not correct, Bob sends an error message. The problem is that this allows the attacker (Trudy) to use Bob as an oracle. Trudy can send Bob a message, and Bob will tell Trudy whether the message (when decrypted) has proper PKCS #1 padding. Some SSL servers were particularly helpful and would say whether the padding was wrong because the first two octets were something other than 0 and 2, or whether it was wrong because the length of the encrypted quantity was something other than what was expected.

Trudy can then carefully craft SSL connection requests with modified versions of the original Alice-to-Bob encrypted message and note what error message she got from Bob. If Bob gave a different error message when the decrypted value began with the octets 0 and 2 (and on average one message in 2^{16} would have that form), Trudy could eventually figure out the encrypted key after seeing the responses to about a million carefully crafted messages.

Attacks of this sort can be avoided if the padding for encryption includes enough redundancy that the probability of a randomly chosen value decrypting into something that looks like it is properly padded is negligible. (*One in a million* is not considered negligible to cryptographers. To them “negligible” should be less than one in 2^{100} or so.) A particularly complex scheme for making the probability negligible is specified in PKCS #1 version 2, also known as OAEP [BELL94], and standardized in IEEE P1363 and RFC 8017.

Because this is an obscure attack easily avoided by other means (like not being so helpful in error messages when people send you invalid messages), the world has not scrambled to migrate to OAEP. But OAEP might be mandated in newly defined protocols where backward compatibility is not an issue.

6.3.6.3 Signing

PKCS #1 also defines standards for formatting a quantity to be signed with RSA. Usually the data being signed is a hash. As with encryption, padding is required. The format normally used is

0	1	at least eight octets of ff_{16}	0	ASN.1-encoded hash type and hash
---	---	---	---	----------------------------------

As with encryption, the top octet of 0 ensures that the quantity to be signed will be less than n . The next octet is the PKCS type; in this case, a quantity to be signed. The padding ensures that the quantity to be signed is very large and therefore unlikely to be a smooth number.

Inclusion of the hash type instead of merely the hash standardizes how to tell the other party which hash function you used.

6.4 DIFFIE-HELLMAN

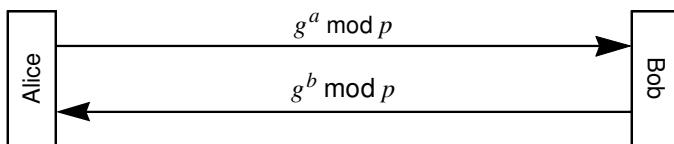
Diffie-Hellman allows two individuals (Alice and Bob) to agree on a shared key even though they can only exchange messages in public. For instance, they might be sending messages across a network, on a telephone that might be tapped, or shouting at each other across a crowded room. Alice shouts “My number is 92847692718497.” Bob shouts “My number is 28379487198225.” Everyone in the room hears what Alice and Bob said, but after that exchange, Alice and Bob know a secret that nobody else in the room can feasibly calculate.

The security of Diffie-Hellman depends on the difficulty of solving the **discrete log problem**, which can be informally stated as, “If you know g , p , and $g^x \bmod p$, what is x ?” In other words, what exponent did you have to raise g to, mod p , to get g^x ?

Diffie-Hellman works with many types of groups (*e.g.*, integers with various moduli, polynomials, elliptic curves), but let’s assume for now that we are using integers with a prime modulus. Some values of p and g are more secure than others. Typically, cryptographers define a few groups to choose from, and you would choose one of those.

So, there are integers p and g , where p is a large prime (say 2048 bits) and g is a number less than p with some restrictions that aren’t too important for a basic understanding of the algorithm. The values p and g are known beforehand and can be publicly known, or Alice and Bob can negotiate with each other across the crowded room. For instance, Alice could shout “I’d like to use any of these published cryptographer-approved Diffie-Hellman groups”, and Bob can reply “Of your proposed groups, I choose the third one in your list”.

Once Alice and Bob agree on a p and g , each chooses a large, say, 512-bit number at random and keeps it secret. Let’s call Alice’s private Diffie-Hellman key a and Bob’s private Diffie-Hellman key b . Each raises g to their private Diffie-Hellman number, mod p , resulting in their public Diffie-Hellman value. So Alice computes $g^a \bmod p$ and Bob computes $g^b \bmod p$, and each informs the other (Protocol 6-4). Finally, each raises the other side’s public value to their own private value.



Protocol 6-4. Diffie-Hellman Exchange

Alice raises $g^b \bmod p$ to her private number a . Bob raises $g^a \bmod p$ to his private number b . So Alice computes $(g^b \bmod p)^a = g^{ba} \bmod p$. Bob computes $(g^a \bmod p)^b = g^{ab} \bmod p$. And, of course, $g^{ba} \bmod p = g^{ab} \bmod p$.

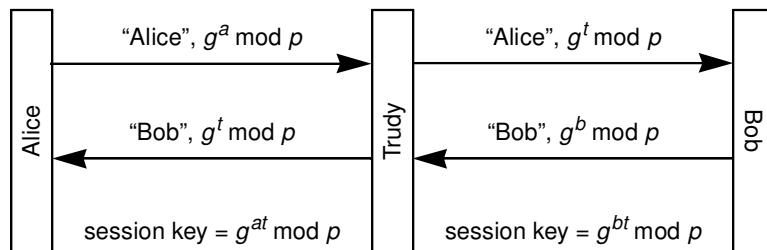
Without knowing either Alice’s private number a or Bob’s private number b , nobody else can calculate $g^{ab} \bmod p$ in a reasonable amount of time even though they can see $g^a \bmod p$ and $g^b \bmod p$.

p . We assume they can't compute discrete logarithms, because of the Fundamental Tenet of Cryptography.

6.4.1 MITM (Meddler-in-the-Middle) Attack

As explained in §1.6 *Active and Passive Attacks*, a MITM attack is where someone (say Trudy) manages to get in the middle of a conversation between Alice and Bob, relaying messages between them, but being able to decrypt or modify messages. Diffie-Hellman alone does not prevent MITM attacks, if we assume that Alice and Bob do not have credentials (such as public keys) for each other.

A classic case of a MITM attack (Protocol 6-5) is where Alice and Bob do a Diffie-Hellman exchange and use the resulting Diffie-Hellman key as their session key. Alice will send $g^a \bmod p$. Bob will send $g^b \bmod p$. But Alice doesn't realize that she is really talking to Trudy. She sends $g^a \bmod p$ to Trudy. Trudy responds with Trudy's own Diffie-Hellman value, say $g^t \bmod p$. So the Alice-Trudy connection will use the session key $g^{at} \bmod p$. Trudy doesn't know Alice's private Diffie-Hellman value a , so Trudy sends her own Diffie-Hellman value $g^t \bmod p$ to Bob. The Trudy-Bob connection will use the session key $g^{bt} \bmod p$. Trudy will be able to see the entire Alice-Bob conversation because every message from Alice to Bob (encrypted with $g^{at} \bmod p$) will be decrypted by Trudy and re-encrypted with $g^{bt} \bmod p$ and then sent to Bob.



Protocol 6-5. Diffie-Hellman with Trudy in the Middle

How can Alice and Bob detect that there is a MITM? Each of them think they have established a secure session to the other. They could try sending each other passphrases that they had agreed-upon for authentication—one password that Bob is to say to Alice, perhaps “The fish are green”, and one that Alice is to say to Bob, for instance, “The moon sets at midnight”. If Alice receives the expected password, can she assume she is talking to Bob? No. Trudy is decrypting each message from Alice and encrypting it when forwarding it to Bob. So the passphrases will get forwarded between Alice and Bob, and they will not detect MITM Trudy.

Perhaps Alice and Bob could transmit their session key over the encrypted channel they have established. That won't work either. Alice sends the message "I think we are using $g^{at} \bmod p$ ". However, Trudy decrypts the message and edits it to be "I think we are using $g^{bt} \bmod p$ " before encrypting and forwarding it on to Bob.

Using Diffie-Hellman alone, there's really nothing Alice and Bob can do to detect the MITM through whom they are communicating. As a result, this form of Diffie-Hellman is only secure against passive attacks where the intruder just watches the encrypted messages.

6.4.2 Defenses Against MITM Attack

One technique by which Diffie-Hellman can be secure against active attacks is for each person to have a somewhat permanent Diffie-Hellman value instead of inventing one for each exchange. The public Diffie-Hellman values would then all be certified by some means that is assumed reliable (for instance, through a PKI; see §10.4 *PKI*). This enables Alice and Bob to start communicating securely without doing any handshake first! Alice's certified public key would be $g^a \bmod p$. Bob's certified public key would be $g^b \bmod p$. Alice and Bob would know that to communicate with each other, they would use the session key $g^{ab} \bmod p$.

Always using the same session secret between Alice and Bob can create vulnerabilities, so we'll describe a more common technique. If Alice and Bob know some sort of secret with which they can authenticate each other, either a shared secret key or knowledge of the other side's public key (e.g., RSA key) and their own private key, then they can prove they generated their Diffie-Hellman value. We call such an exchange an **authenticated Diffie-Hellman exchange**. Authentication can be done simultaneously with sending the Diffie-Hellman values or after the Diffie-Hellman exchange. Examples are:

- Encrypt the Diffie-Hellman exchange with the pre-shared secret.
- Encrypt the Diffie-Hellman value with the other side's public key (Homework Problem 2).
- Sign the Diffie-Hellman value with your private key.
- Following the Diffie-Hellman exchange, transmit a hash of the agreed-upon shared Diffie-Hellman value, your name, and the pre-shared secret.
- Following the Diffie-Hellman exchange, transmit a hash of the pre-shared secret and the Diffie-Hellman value you transmitted.

Yet another defense against MITM is to use channel bindings. (See §11.6 *Detecting MITM*.)

6.4.3 Safe Primes and the Small-Subgroup Attack

While Diffie-Hellman will “work” (in the sense of Alice and Bob agreeing on a common value to use as a key) for any values of p and g , there will be security vulnerabilities unless certain other criteria are met. In particular, there are known attacks on Diffie-Hellman if $(p-1)/2$ is smooth (see §6.3.5.1 *Smooth Numbers*).

Traditionally, Diffie-Hellman is done with primes where $(p-1)/2$ is also prime. A prime p that satisfies this additional constraint is called a **safe prime**, while $(p-1)/2$ is called a **Sophie Germain prime**. The number of elements in a group is known as the **order** of the group. If the group is multiplication mod p , and p is prime, then the order of the group is $p-1$ (because 0 is not in the multiplicative group mod p).

As we will see, DSA (§6.5.1 *The DSA Algorithm*) uses Diffie-Hellman-style key pairs, but has an optimization that speeds up signing and verification, and it makes signatures smaller. Instead of having p be a safe prime, $(p-1)/2$ only needs at least one reasonably large factor q , where the length of q is twice the desired security strength. So for 128-bit security, q would be 256 bits. This optimization makes DSA faster because some of the computation can be done mod q rather than mod p . Since typically p will be 3072 bits, and q will be 256 bits, the exponents will be a twelfth the size when operating mod q .

The number g is known as a **generator**. If you take an element g of a group and multiply it by itself over and over, you will get elements g^1, g^2, \dots etc. Once you get to 1 (the identity element), you will have generated some subset of the group. This is known as a **cyclic subgroup**. This subgroup will be **closed** under multiplication, meaning that multiplying any two elements of the subgroup together will result in one of the elements of the subgroup. The number of elements in the subgroup is known as the **order** of the subgroup. The order of a subgroup always divides the order of the group.

Recall, if p is a prime, then the order of the group is $\phi(p)$, which equals $p-1$. If p is a safe prime, then the only factors of $p-1$ are 1, 2, the prime $(p-1)/2$, and $p-1$. So the only subgroups are:

- The subgroup of order 1, generated by $g=1$ (since all powers of 1 = 1).
- The subgroup of order 2, consisting of the elements ± 1 , and generated by $g = -1$.
- The subgroup of order $p-1$, consisting of the entire group. Almost half of the group elements (those that are not squares, other than -1) will each generate the entire group.
- The subgroup of (prime) order $(p-1)/2$. Again, almost half the group elements (those that are squares, other than 1) will each generate this subgroup. For instance, assume you have a generator g that generates all $p-1$ elements of the group. Then g^2 will only generate every other element in the list generated by g .

It is computationally expensive to choose p and g . Theoretically, one only needs to choose them once and keep using the same p and g . It could even be a constant in the standard, and everyone could use the same p and g . The advantage of having everyone use the same p and g or choosing

from a small set of standardized values is that if you get the values from a trusted source, you don't have to test whether p is appropriate, for instance, whether both p and $(p-1)/2$ are prime.

It turns out to be possible, though incredibly space- and computation-intensive, to calculate a large table based on a single p , which would allow you to compute discrete logarithms *for that p*. For example, a 1024-bit prime is too short, since it would take workfactor of about 80 bits to break that prime, and then workfactor 50 bits to break Diffie-Hellman using that prime. The recommended size of a Diffie-Hellman prime is at least 2048. To break a prime of that size is workfactor 112 bits, and subsequently breaking a Diffie-Hellman exchange using that prime would be about 70 bits. For a Diffie-Hellman prime of 3072 bits, breaking the prime has a workfactor of 128 bits, and subsequently breaking Diffie-Hellman with that prime has a workfactor of 80 bits. That is why Diffie-Hellman uses 3072-bit primes in order to get 128-bit security.

A similar scheme would allow you to break RSA in the sense of being able to calculate someone's private key based on their modulus. There is not that much incentive for an attacker to do that, however. Since everyone is using a different RSA modulus, that would only allow the attacker to break one person's key. If the p for Diffie-Hellman were broken, and many people were using the same p , then every key exchange based on Diffie-Hellman with that modulus could be broken. Simply changing p occasionally will eliminate this threat.

Now we'll discuss the **small group attack**. Things would certainly not be secure if one were doing Diffie-Hellman in a group of really small order (such as if you were using the element -1 as the generator g), because g^{ab} would have few possible values. So it's important for the subgroup in which Diffie-Hellman is executed to be quite large.

First, let's assume $p-1$ is smooth, meaning that its factors are all small numbers, say f_1, f_2, \dots, f_k . An eavesdropper, Trudy, that sees $g^a \bmod p$ will be able to calculate Alice's private number a , as follows.

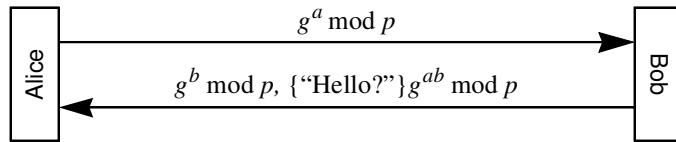
Take any factor f of $p-1$. There will be f elements generated by $g^{(p-1)/f}$. Since f is small, Trudy can enumerate all of those elements. Trudy can see what Alice's a is, mod f , by taking $g^a \bmod p$ and raising it to the power $(p-1)/f$. By matching the result with the enumerated members of the subgroup of size f , Trudy now knows that a is mod f .

Trudy does this with all the factors of $p-1$. Now Trudy knows, for each factor f , what Trudy's a is mod that f . By the Chinese remainder theorem (§2.7.6), Trudy can now compute what a is mod the product of all the f s, which is Alice's number a .

So having $p-1$ be smooth would not be secure at all. Any eavesdropper would be able to calculate Alice's and Bob's private values.

Now let's instead choose a p such that $p-1$ has one large factor q . It doesn't matter what the other factors of $p-1$ are, so let's assume they are all small. There will be a subgroup with q elements. An algorithm using this technique (*e.g.*, DSA) will be designed so that some of the operations will still be done mod p , so the "prime breaking" we discussed in the beginning of the section is not a problem. The generator g that will be chosen for this kind of optimized Diffie-Hellman will generate the order q subgroup.

Consider Protocol 6-6:



Protocol 6-6. Diffie-Hellman Exchange

If Alice were malicious, and if she can interact with Bob lots of times, and if Bob keeps using the same private number b , then Alice will be able to calculate b by using a technique similar to what Trudy used when $p-1$ was smooth, as follows.

Malicious Alice creates her public number A , not by raising the chosen generator g to a power, but instead by choosing an A that generates a small subgroup of size f . With this A , Alice can calculate what Bob's value b is mod f , because he encrypts something recognizable using key A^b —Alice goes through the f possible values of A^b and discovers which of those Bob used to encrypt his message. Since, for efficiency, these algorithms allow Alice and Bob to use reasonably small exponents (e.g., 256 bits), Alice only needs to get what Bob's value is mod a set of f s such that the product of those f s is at least 256 bits.

So, Bob has to check whether Alice is sending him a malicious A . He does this by calculating $A^q \bmod p$. If the result is 1, then A is assumed honestly chosen. Otherwise, Bob recognizes that Alice is not playing by the rules.

6.4.4 ElGamal Signatures

Using the same sort of keys as Diffie-Hellman, ElGamal came up with a signature scheme [ELGA85] to sign $\text{hash}(m)$. It is much harder to understand than the RSA signature scheme. It is useful to understand ElGamal signatures, though, because it will make it easier to understand DSA and ECDSA (§6.5). (ECDSA is the DSA algorithm using elliptic curves.)

ElGamal signatures require using a well-known g and p . The signer (Alice) needs a long-term public/private Diffie-Hellman key pair (the public key being $A = g^a \bmod p$ and the private key being a as described for Diffie-Hellman). She then signs message m by doing the following:

- Alice computes a hash of the message: $M = \text{hash}(m)$.
- She randomly chooses a per-message secret t relatively prime to $p-1$ and computes $t^{-1} \bmod (p-1)$. Since t is chosen to be relatively prime to $p-1$, this calculation is possible.
- She computes $T = g^t \bmod p$. The value T will be part of her signature on $\text{hash}(m)$.
- She computes $S = (M - aT) \times t^{-1} \bmod (p-1)$.
- Her signature on m is $\langle T, S \rangle$.

- Bob verifies the signature by doing the following:
 - He computes the hash of the message: $M = \text{hash}(m)$.
 - He verifies that $g^M = A^T \times T^S \pmod{p}$.
 - To prevent certain forgery attacks, he must also verify that T and S are each greater than 0 and less than $p-1$.

Why does verification work?

- $A^T \times T^S \pmod{p} = (g^a)^T \times (g^t)^S \pmod{p} = g^{aT+tS} \pmod{p}$.
- Since $g^{p-1} \pmod{p} = 1$, $g^{aT+tS} \pmod{p} = g^{aT+tS \pmod{(p-1)}} \pmod{p}$.
- From how S was computed, we get $tS = (t \times (M - aT) \times t^{-1}) \pmod{p-1} = M - aT \pmod{p-1}$.
- Substituting for tS , we get $A^T \times T^S = g^{aT+M-aT \pmod{(p-1)}} \pmod{p} = g^M \pmod{p-1} \pmod{p}$.

It is not very intuitive why this is secure. The important things to try to convince yourself of are:

- If the signature is done correctly, the verification will succeed.
- If the message is modified after being signed, the inputs to the signature function will have changed, and with overwhelming probability the signature will not match the modified message.

The things you have to believe based on the Fundamental Tenet of Cryptography are:

- Someone who doesn't know a will not be able to find values for T and S that will satisfy the above equation.
- Seeing any number of signatures will not help an attacker compute Alice's private key, a .

ElGamal signatures in this form are rarely used today because there are variations on this theme (*e.g.*, DSA) that have better performance that are used instead. We mention ElGamal signatures because they are historically important and easier to understand than the more advanced techniques that follow.

6.5 DIGITAL SIGNATURE ALGORITHM (DSA)

NIST, the (U.S.) National Institute of Standards and Technology, published an algorithm for digital signatures in 1991 similar to ElGamal but with better performance and shorter signatures. The algorithm is known as **DSA**, for **Digital Signature Algorithm**.

The evolution of the design of cryptographic algorithms is guided by the work of cryptanalysts. As mentioned in §6.4.3 *Safe Primes and the Small-Subgroup Attack*, Diffie-Hellman and ElGamal are traditionally done with safe primes because if $(p-1)/2$ is smooth, there are some spe-

cialized attacks possible. Safe primes are as far as you can get from having $(p-1)/2$ be smooth. Using safe primes turns out to be overkill, since those specialized attacks only work if *all* factors of $(p-1)/2$ are smaller than twice the workfactor to break the scheme. That means that if p is chosen large enough to have a 128-bit workfactor to break (currently believed to be about 3072 bits), it's only important that $(p-1)/2$ have at least one prime factor q that is at least 256 bits in size. DSA takes advantage of that to construct a variation of ElGamal signatures that is just as secure, while being much faster to compute and verify.

Instead of all calculations being done mod p (where p is a large prime), some are done mod q (where q is a smaller prime that divides $p-1$ but is at least twice as big as the desired security strength). Given the best known attacks, to get security comparable to a 128-bit AES key, ElGamal signatures would need a 3072-bit prime p , while DSA would need a 3072-bit prime p and a 256-bit prime q . Because DSA can do a lot of its computation using a 256-bit q instead of ElGamal's 3072-bit p , signing with DSA is six times faster, and the signatures are twelve times smaller.

As with RSA and Diffie-Hellman, DSA is defined for a number of different-sized parameters, trading off security against performance.

6.5.1 The DSA Algorithm

As with ElGamal, the DSA algorithm requires choosing long-lived parameters g , p , and q that will be public. The process of choosing these parameters is computationally intensive, but generating new key pairs, signing, and verification are relatively fast. The procedure for finding p and q involves first finding a 256-bit prime q and then searching for 3072-bit primes of the form $qn+1$. The generator g is chosen to ensure that the subgroup generated by g has order q . To generate a signature, we can assume that the parameters p , q , and g are already known.

As with Diffie-Hellman and ElGamal, our signer Alice will have a public/private key pair where a is her private key, and $A=g^a \text{ mod } p$ is her public key. The signing procedure follows the signing procedure for ElGamal except that some of the operations are done mod q , and many of the exponents are q -sized rather than p -sized:

- Alice computes a hash of the message: $M = \text{hash}(m)$.
- She randomly chooses a per-message secret t (where $0 < t < q$), and she computes $t^{-1} \text{ mod } q$.
- She computes $T=((g^t \text{ mod } p) \text{ mod } q)$. The value T will be part of her signature on $\text{hash}(m)$.
- She computes $S=(M+aT) \times t^{-1} \text{ mod } q$.
- Her signature on m is $\langle T, S \rangle$.

Bob verifies the signature with the following calculation (which is somewhat different from the ElGamal verification):

- He computes the hash of the message m : $M = \text{hash}(m)$.

- He calculates $x=M \times S^{-1} \pmod{q}$.
- He calculates $y=T \times S^{-1} \pmod{q}$.
- He verifies that $T=(g^x \times A^y \pmod{p}) \pmod{q}$,

Why does this work? (Homework Problem 8).

6.5.2 Why Is This Secure?

What does it mean to be secure? It means several things.

- Signing something does not divulge Alice's private key a .
- Nobody should be able to generate a signature for a given message without knowing a .
- Nobody should be able to generate a message that matches a given signature.
- Nobody should be able to modify a signed message in a way that keeps the same signature valid.

Why does DSA have all these properties? It is believed secure because of the Fundamental Tenet of Cryptography. DSA also has the blessing of the NSA, arguably the best cryptographers in the world. Unfortunately, it's a mixed blessing, since some cynics believe NSA would never propose an algorithm it couldn't break.

6.5.3 Per-Message Secret Number

Both DSA and ElGamal require that the signer generate a unique secret number t for each message. If the same secret number were used for two different messages, it would expose the signer's private key. Likewise, if a secret number t were predictable or guessable, the signer's private key would be exposed.

How is Alice's private key exposed if the secret number for a message is known? In DSA, the signature is $S=(M+aT) \times t^{-1} \pmod{q}$. Remember that t is the secret number, T is $g^t \pmod{p}$, and a is the signer's private key. So if t is known (or can be guessed), then we can compute

$$a = (S - M) T^{-1} \pmod{q}$$

Knowing Alice's private key a , we can forge DSA signatures on anything.

How is the private key exposed when two messages share the same secret number? In DSA, if m and m' are signed using the same secret number t , then we can compute

$$a = (S - S')^{-1} (\text{hash}(m) - \text{hash}(m')) \pmod{q}$$

Similar arguments exist for ElGamal signatures. See Homework Problem 7.

An application that NIST was particularly designing for was doing signatures on the type of low-cost, low-performance smart card available in the 1990s. For example, a smart card could be

embedded on a badge that would authenticate to a door. It would be unpleasant if a person had to wait many seconds for the door to open.

DSA requires taking a multiplicative inverse mod q , which is quite expensive. Both the signer and the verifier will need to do this multiplicative inverse operation. But DSA can be implemented so that the signer can pre-compute a per-message secret t and the multiplicative inverse of t mod q . There are several ways of generating a unique secret number for each message.

- Use truly random numbers. The problem with this is that it requires special hardware. It's difficult enough to make hardware predictable, but it's even harder to make it predictably unpredictable.
- Use a cryptographic pseudorandom number generator. The problem with this is that it requires nonvolatile storage in order to store its state.
- Compute t to be a cryptographic hash of a combination of the message and the signer's private key. The problem with this is that t (and therefore t^{-1}) can't be computed until the message is known.

6.6 HOW SECURE ARE RSA AND DIFFIE-HELLMAN?

A brute-force attack (trying all possible keys) requires an exponential (in the length of the key) work factor. The security of RSA is based on the difficulty of factoring. The security of Diffie-Hellman is based on the difficulty of solving the discrete log problem. The best-known algorithms for solving them on a classical computer are subexponential (less than exponential) but superpolynomial (more than any fixed-degree polynomial). Because the difficulty is subexponential, the required size of the keys in these public key algorithms is much larger (say, 3072 bits) than a corresponding secret key (say 128 bits).

NIST has constructed tables estimating the keysizes needed in various algorithms to get various levels of security (Figure 6-7). These may change if there are breakthroughs in cryptanalysis but should not be affected by speedups in hardware (though the needed level of security will be). The table starts at 80 bits, which was considered adequate at one time, but going forward, a cryptographic strength of at least 128 bits should be used.

Note, as we will explain in Chapter 7 *Quantum Computing*, a sufficiently large quantum computer would be able to factor and compute discrete logs efficiently enough that none of the algorithms in this chapter would be secure, though secret key algorithms and hashes would survive.

Security Strength	Secret KeySize	Hash Size	RSA Modulus	DH Exponent	DH Modulus	DSA q	DSA p	ECC KeySize
80	80	160	1024	160	1024	160	1024	160
112	112	224	2048	224	2048	224	2048	224
128	128	256	3072	256	3072	256	3072	256
192	192	384	7680	384	7680	384	7680	384
256	256	512	15360	512	15360	512	15360	512

Figure 6-7. Security Strength of Various Algorithms

6.7 ELLIPTIC CURVE CRYPTOGRAPHY (ECC)

As we said in §6.6 *How Secure Are RSA and Diffie-Hellman?*, there are known subexponential (but superpolynomial) algorithms for breaking RSA and Diffie-Hellman based on modular arithmetic on a classical computer. Elliptic curve cryptography (ECC) is important because no one has (yet?) found subexponential algorithms (on a classical computer) for breaking it. Therefore, it is believed to be secure against attacks with classical computers with much smaller keysizes, which is important for performance. ECC is a popular replacement for public key cryptographic schemes like RSA, Diffie-Hellman, ElGamal, DSA, etc. For some of these cryptographic schemes, you can replace modular multiplication by elliptic curve multiplication directly, resulting in algorithms referred to as ECC Diffie-Hellman (or ECDH) and ECC DSA (or ECDSA). Of course, none of these would be secure if there were a sufficiently large quantum computer running Shor's algorithm (§7.3 *Shor's Algorithm*).

In general, an **elliptic curve** is a set of points on the coordinate plane satisfying an equation of the form $y^2 + axy + by = x^3 + cx^2 + dx + e$, though only a few special forms of elliptic curves are used in cryptography. Note that ellipses are not elliptic curves. While some of the original mathematical exploration of elliptic curves was related indirectly to ellipses, there is no relationship that is useful in cryptography. Originally, elliptic curves were conceived where the two coordinates x and y were real or complex numbers, but cryptography uses elliptic curves where x and y are members of a finite field whose size is either a large prime p or a number of the form 2^n . With finite field coordinates, there is no way to make a sensible graph that looks like curves, although they are still called “curves”. The currently most popular curves (because they give the best performance for a given level of security) have formulas of the form $y^2 = x^3 + ax + b$ and use a finite field with mod p arithmetic where p is a large prime very close to (but less than) a power of 2. Elements on the curve are points with an x and y coordinate. Because the formula defining the curve has y^2 on the left side of

the equation, if the point $\langle x, y \rangle$ is on the curve, then $\langle x, -y \rangle$ is also on the curve. These will be the only points on the curve with that particular x coordinate, so specifying the x coordinate and the sign of the y coordinate is enough to determine the point. When encoding the value of an elliptic curve point, it is common to represent it as an x coordinate that is an integer between 0 and p and a single bit that distinguishes between the two possible y values.

In order to use elliptic curves for, say, Diffie-Hellman, there needs to be some mathematical operation on two points on the curve that will always produce a third point on the curve. Let's call that operation "multiplication", although in ECC it will not look like any multiplication you are used to.

Be warned when reading other papers about elliptic curve cryptography, most authors refer to the operation applied to points on the curve as addition rather than multiplication. Where we talk about raising g to some power, they will speak of scalar multiplication of g by an integer. This is just a matter of notation used and does not change any of the results. We believe the use of multiplication for the operation on points makes things clearer because it makes the parallels between the mod p and the elliptic curve versions of Diffie-Hellman and DSA more obvious. And while they call the repeated operation of a point with itself *scalar multiplication*, they still call the process of reversing that operation (which is needed to break the cryptosystem) *computing a discrete logarithm* rather than *scalar division*. When describing elliptic curves with the operation being addition, the identity element of the group is referred to as **0**. If the operation is referred to as multiplication, the identity element is referred to as **1**.

The operation (that we call multiplication) has to be associative, so that you can use the repeated squaring trick to raise a number to a large power in time linear with the length of the exponent. In other words, to "exponentiate" a point by 128, you should be able to multiply the point by itself (you've now raised it to the power 2), then multiply the result by itself (you've now raised it to the power 4), multiply the result by itself (to have raised it to the power 8), and so on. Since this multiplication operation is associative, it will be true that $(g^x)^y = g^{xy} = (g^y)^x$. And it is also important that doing discrete logs is hard (knowing g and g^x , it is difficult to compute x). We'll give the formulas for the multiplication operation shortly.

Before giving the formulas, there is one more important detail. To make the set of points satisfying the elliptic curve formula a group, it is necessary to define one additional point, called the **point at infinity**, and come up with some unique representation for it in memory when doing this arithmetic on a computer. This is the identity element of the group, so we'll call it **1**. Multiplying any point by **1** is easy. For any point P , $P \times \mathbf{1} = \mathbf{1} \times P = P$. To multiply any two points $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$ where $x_1 \neq x_2$, the result is $\langle x_3, y_3 \rangle$ where:

- $x_3 = ((y_2 - y_1)/(x_2 - x_1))^2 - x_1 - x_2$
- $y_3 = ((y_2 - y_1)/(x_2 - x_1)) \times (x_1 - x_3) - y_1$

These formulas don't work when $x_1=x_2$. If $x_1=x_2$, then either $y_1=y_2$ or $y_1=-y_2$. If $y_1=-y_2$, then the two points are inverses, and $\langle x_1, y_1 \rangle \times \langle x_1, -y_1 \rangle = \mathbf{1}$. Multiplying a point by itself has a special formula: $\langle x_1, y_1 \rangle \times \langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle$ where:

- $x_2 = ((3x_1^2 + a)/2y_1)^2 - 2x_1$ where a comes from the formula $y^2 = x^3 + ax + b$
- $y_2 = ((3x_1^2 + a)/2y_1) \times (x_1 - x_2) - y_1$

With point multiplication so defined, the set of points on an elliptic curve form a group, complete with an identity, inverses, and a multiplication operation that is commutative and associative. While more complex to understand, it is faster, at least for private key operations, since until someone comes up with a subexponential algorithm for breaking ECC, the keys can be smaller. For public key operations, such as signature verification, RSA is likely to be faster, even with larger keys, because it can use a small public exponent.

Note that since the difficulty of Shor's algorithm (§7.3) depends on the size of the key (whether doing ECC or modular arithmetic), ECC, with its smaller keys, will likely be broken before RSA.

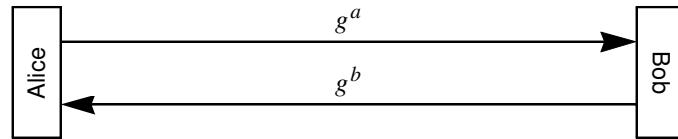
When doing cryptography with elliptic curve groups, we don't always use all the elements of the group. We pick some element g of the group and use only elements that can be computed as g^i for some integer i . This will be a subgroup of the elliptic curve group (sometimes the entire group). With the right choice of g , a , b , and p , the result will be a subgroup whose size is a prime about the same size as p .

6.7.1 Elliptic Curve Diffie-Hellman (ECDH)

Unlike mod p Diffie-Hellman and ElGamal, no one has discovered any separate operation to "break" an elliptic curve with an expensive operation that makes future breaking of individual public keys or exchanges using that curve easy. As a result, almost all uses of elliptic curve cryptography use one of the pre-calculated curves published by NIST [NIST13—Appendix D]. They have published a series of curves with different sizes (for different security strengths) and based on different finite fields.

The math associated with ECDH (Protocol 6-8) looks just like the math associated with Diffie-Hellman. Alice and Bob each choose a random number of twice the needed security strength (e.g., 256 bits for security comparable to AES-128). Alice raises the generator of the elliptic curve group to the power a , Bob raises it to the power b , and they exchange values. They then each com-

pute g^{ab} by raising the value they receive to the power of their private exponent. But an eavesdropper who sees g^a and g^b will not be able to compute g^{ab} without an extraordinary effort.



Protocol 6-8. Elliptic Curve Diffie-Hellman Exchange

6.7.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

The formulas for ECDSA are remarkably similar to those for DSA. It's just that they use a different form of arithmetic. Given an elliptic curve group with some prime number n of elements and a generator g , all the elements can be expressed as g^i , where $0 \leq i < n$. The exponents are integers and since $g^n = 1$, you can do arithmetic with them mod n .

The points on the elliptic curve will have x and y coordinates and be expressed as an ordered pair $\langle x, y \rangle$. Alice will choose a public/private key pair where her private key will be a randomly chosen integer a between 2 and $n-1$. In order to sign a message m , she must perform the following steps:

- Alice computes a hash of the message: $M = \text{hash}(m)$
- Alice has a long-term public key $A = g^a$. She chooses a per-message secret t .
- She computes $T =$ the x -coordinate of g^t . The value T will be part of her signature on m .
- She computes $S = (M + aT) \times t^{-1} \pmod{n}$
- Her signature on m is $\langle T, S \rangle$. Note that these are two integers—no curve points involved.

The calculations Bob does to verify the signature also follow the same formulas as for DSA:

- He calculates $x = M \times S^{-1} \pmod{n}$.
- He calculates $y = T \times S^{-1} \pmod{n}$.
- The signature is valid if $T =$ the x -coordinate of g^xA^y

As with DSA, Bob needs to do some additional checks to ensure that Alice's value is properly generated (see §6.4.3 *Safe Primes and the Small-Subgroup Attack*).

6.8 HOMEWORK

1. In mod n arithmetic, why does x have a multiplicative inverse if and only if x is relatively prime to n ?
2. In section §6.4.2 *Defenses Against MITM Attack*, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?
3. In RSA, given that the primes p and q are approximately the same size, approximately how big are p and q compared to n ? How big is $\phi(n)$ compared to n ?
4. What is the probability that a randomly chosen number would not be relatively prime to some particular RSA modulus n ? What threat would finding such a number pose?
5. Suppose there is no enforced padding structure such as in PKCS #1. Suppose Fred sees your RSA signature on m_1 and on m_2 (*i.e.*, he sees $m_1^d \bmod n$ and $m_2^d \bmod n$). How does he compute the signature on each of $m_1^j \bmod n$ (for positive integer j), $m_1^{-1} \bmod n$, $m_1 \times m_2 \bmod n$, and in general $m_1^j \times m_2^k \bmod n$ (for arbitrary integers j and k)?
6. Suppose we have the encoding that enables Carol to mount the cube root attack (see §6.3.5.2 *The Cube Root Problem*). If Carol sends a message to Bob, supposedly signed by you, will there be anything suspicious and noticeable about the signed message, so that with very little additional computation, Bob can detect the forgery? Is there anything Carol can do to make her messages less suspicious?
7. In ElGamal, how does knowing the secret number used for a signature reveal the signer's private key? How do two signatures using the same secret number reveal the signer's private key? Hint: $p-1$ is twice a prime. Even though not all numbers have inverses mod $p-1$, division can still be performed if one is willing to accept two possible answers. (We're neglecting the case where the divisor is $(p-1)/2$, since it's extremely unlikely.)
8. Show that verification works in DSA.
9. Suppose (see §6.3.4.3 *Why a Non-Prime Has Multiple Square Roots of One*) you know a non-trivial square root, y , of n . Show how this will let you factor n . Hint: $(y^2 - 1) \bmod n = 0$.

7

QUANTUM COMPUTING

7.1 WHAT IS A QUANTUM COMPUTER?

Quantum mechanics predicts that it should be possible to build a computer that can do certain calculations much faster than would be possible on a conventional (classical) computer. Aspects of quantum mechanics may seem nonintuitive, but all evidence supports it. In this chapter, we describe how a quantum computer differs from a classical computer and give intuitive descriptions of the quantum algorithms most relevant to cryptography.

There are entire books about quantum mechanics. Our goal isn't to pack years of physics and math into a few pages but to give some insight into the concepts, terminology, and notation, as well as the algorithms that run on quantum computers. And for readers who want to study the topic more deeply, hopefully this introduction will make it easier to understand the deeper literature.

When I₂ challenged me₄ to write something “really short” about what a quantum computer is, I₄ wrote “a quantum computer is a magic box that factors numbers quickly.” We want to go into more detail than that, but keep the chapter reasonably intuitive and short.

7.1.1 A Preview of the Conclusions

First, we'll hint at the conclusions, and then we'll explain why they are true.

A common misconception is that a quantum computer is faster than a classical computer, and that because Moore's law is slowing down, eventually all computers will be replaced by quantum computers. This is definitely not true. There is only a narrow set of problems for which a quantum computer would be faster. The property of a quantum computer that makes it excel at some tasks is that it can compute, with only storage size n , as if it were operating on 2^n values in parallel. However, as we'll see later in this chapter, it also has serious limitations, such as if you read the state you will see only one value and the others disappear. The magic of a typical quantum program is for the

quantum computation to raise the likelihood that when you finally read a result, it will be a useful value.

Another common misconception is that a quantum computer can solve NP-hard problems (such as the traveling salesman problem) in polynomial time. This is almost certainly not true. Although nobody has proven that it is impossible, no known quantum algorithm for solving such problems is that powerful. Indeed, a quantum algorithm that powerful would be nearly as surprising, given our current state of knowledge, as a classical algorithm that was that powerful.

Although a quantum computer can, in principle, do any calculation that a classical computer can do, for most calculations quantum computers would be no faster than conventional computers. Also, in practice, quantum computers are likely to be much more expensive to build and operate. For example, most designs would need to operate at temperatures very close to absolute zero. So it is not likely that quantum computers will ever serve more than a small niche market doing extremely CPU-intensive calculations on tiny amounts of data. However, there are two important quantum algorithms that are relevant to cryptography.

- Grover's algorithm, which makes brute-force search faster. This might seem worrisome for things like encryption or hashes, where a brute-force search can find a key or a pre-image. But the limit of Grover's algorithm is that it only reduces the search time to the square root of the size of the search space. Squaring the size of the space being searched (for instance, using an encryption key twice as long) is more than adequate to protect against Grover's algorithm.
- Shor's algorithm, which can efficiently factor numbers and calculate discrete logs. Shor's algorithm, run on a sufficiently large quantum computer, would break all our widely used public key algorithms (*e.g.*, RSA, Diffie-Hellman, elliptic curve cryptography, ElGamal). At the time of this writing, no quantum computer large enough to break the currently deployed public keys has ever been publicly demonstrated, and some experts remain skeptical that one ever will be built. There are a number of difficult engineering challenges that remain, and it may never be economically viable to overcome them. But because such a computer might be possible, it is important to convert to quantum-safe public key algorithms well before a quantum computer (of sufficient size) might exist. The cryptographic community is actively developing and standardizing such algorithms, which we describe in Chapter 8 *Post-Quantum Cryptography*.

7.1.2 First, What Is a Classical Computer?

Classical computers compute on information stored in bits. Each bit stores either a 0 or a 1. A classical computer with n bits can be in one of 2^n states. For instance, with three bits, the possible states are 000, 001, 010, 011, 100, 101, 110, 111. A classical computer operates on bits using **gates** (such as AND and NOT) that take some number of bits as input and then output some number of

bits. A classical gate can be described with a table that indicates, given the value of the input bit(s), what the value of the output bit(s) are. For instance, the classical AND gate's table is:

Input	Output
00	0
01	0
10	0
11	1

In quantum computation, there is an intuitively similar notion of a gate. However, whereas a classical gate reads inputs and writes the corresponding output value to a different location, a quantum gate operates on a collection of qubits and changes the state of those qubits. We'll describe more about that later.

7.1.3 Qubits and Superposition

Instead of bits, a quantum computer uses **qubits**, where a qubit's state can be a mixture of a 0 and a 1. This is known as **superposition**. The standard notation for describing superposition is known as **ket** notation. A ket is a symbol or value written between a vertical bar and a right angle bracket representing a state. The state of one or more qubits is usually denoted by $|\psi\rangle$. The standard notation for a single qubit's state is $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. The coefficients (α and β) determine how likely the measured value will be 0 *versus* 1. Measuring a qubit (reading it) destroys the superposition information, and the qubit takes on the value read—either completely 0 (written as $|0\rangle + 0|1\rangle$, or simply $|0\rangle$) or completely 1 (written as $0|0\rangle + |1\rangle$, or simply $|1\rangle$).

The standard notation for the state of two qubits is $|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$. The coefficients ($\alpha, \beta, \gamma, \delta$) determine how likely the measured value will be 00, 01, 10, or 11.

Once measured, a qubit is no different from a classical bit—it's either 0 or 1. How does a qubit become a mixture of 0 and 1? A quantum computer can compute with gates that create superposition, as we will show in §7.1.5.

The coefficients (α and β) of the qubit state are called **probability amplitudes**, and their squared absolute values are probabilities. Given that the total probability must be 1, $|\alpha|^2 + |\beta|^2 = 1$. The probability that a qubit in state $\alpha|0\rangle + \beta|1\rangle$ would be read as 0 is $|\alpha|^2$, and the probability that it would be read as 1 is $|\beta|^2$. For example, if $\alpha = \frac{1}{\sqrt{2}}$ and $\beta = -\frac{1}{\sqrt{2}}$, then the probability of the state $\alpha|0\rangle + \beta|1\rangle$ being read as 0 is $\frac{1}{2}$, and the probability of it being read as 1 is also $\frac{1}{2}$. In most literature the coefficients are referred to as amplitudes.*

* For a while, we'll use the term *coefficient* rather than *amplitude* because α in the expression $\alpha|0\rangle$ describes both the magnitude and the phase (see Figure 7-2), and the words *amplitude* and *magnitude* are often used as synonyms in nontechnical English.

The coefficients are actually complex numbers. But, for ease of drawing a figure, and because until we explain Shor's algorithm we only need to use real coefficients, assume for now that the coefficients are real numbers. Given that the probabilities $|\alpha|^2 + |\beta|^2$ need to add up to 1, if you were to graph all the potential (real) values of α and β , you'd wind up with a circle of radius 1. Note that if you change the sign of α or β , the probability of reading 0 or 1 is not changed. For instance, in the left half of Figure 7-1, we show $\alpha|0\rangle + \beta|1\rangle$. And in the right half of Figure 7-1, we show $\alpha|0\rangle - \beta|1\rangle$. In both cases, the probability that the qubit would be measured as 0 is $|\alpha|^2$, and the probability that the qubit would be measured as 1 is $|\beta|^2$.



Figure 7-1. Real Coefficients

If two coefficients are different, but they have the same absolute value, the two coefficients are said to have different **phases**. If the coefficient is a real number, the phase is just a choice of plus or minus. For complex numbers, there is a continuum of possible phases. (If we plot a complex coefficient $x+yi$ as a vector in two dimensions, its phase is the angle the vector makes with the real axis, and its absolute value is the vector's length. See Figure 7-2 and note that a coefficient can have absolute value no more than 1 and so can only exist within the radius-1 outer circle; any coefficient on the radius- m inner circle has absolute value m .)

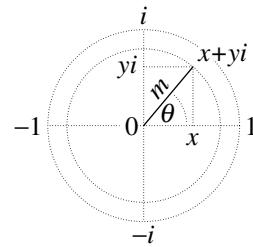


Figure 7-2. A Complex Number $x+yi$ with Absolute Value (aka Magnitude) m and Phase θ

Why should we care about the phase of a coefficient if it doesn't change the probability of that value being read? The answer is that in useful quantum computations the state is repeatedly altered by transformations called **quantum gates** before the state is measured, and the probability of reading a certain value after the gates are applied generally depends on what the phases of the

coefficients were before the gates were applied. We'll talk more about quantum gates later in the chapter.

7.1.3.1 Example of a Qubit

A photon behaves as a qubit, where the polarization of the photon can be thought of as its quantum state. It can be polarized to be up/down (which we'll interpret as 1), right/left (which we'll interpret as 0), or anything in between. If a polarizing filter is exactly aligned with the photon, the photon will definitely pass through the filter. If the filter is 90° off from the photon's polarization, the photon will definitely not pass through. If the filter is 45° off from the polarization of the photon, the photon will pass through with probability $\frac{1}{2}$. More generally, if the photon is polarized at angle ϕ relative to the filter, then the probability that the photon passes through the filter is $\cos^2\phi$.

What's happening is that the polarizing filter is measuring the photon. The values 0 and 1 are relative to the alignment of the filter; the photon's state is $\sin\phi|0\rangle + \cos\phi|1\rangle$. If the photon passes through, it means it has been measured as 0 relative to the polarizer. And since measurement destroys the 1 component of the photon, the photon will now be exactly aligned with the filter.

This behavior can easily be demonstrated in real life with three polarizing filters. If you put two filters aligned 90° from each other on top of each other, no light passes through (none of the photons get through). However, if you insert the third filter between those two, and align it 45° from the others, light will pass through the combination of the three filters ($\frac{1}{2}$ will pass through the first filter, then of those, $\frac{1}{2}$ will pass through the second filter, and then of those, $\frac{1}{2}$ will pass through the third filter). (See Homework Problem 1.)

Note that what you define as 0 and 1 is arbitrary, as long as what is defined as 0 is orthogonal to what is defined as 1.

7.1.3.2 Multi-Qubit States and Entanglement

Another strange concept in quantum computers is **entanglement**—where a collection of qubits has a state that can be described collectively but cannot be described by specifying the states of the individual qubits. For instance, three qubits can be in a superposition of all eight possible classical states: 000, 001, 010, 011, 100, 101, 110, 111. To express the state $|\psi\rangle$ of the set of three qubits, the notation would be

$$|\psi\rangle = \alpha|000\rangle + \beta|001\rangle + \gamma|010\rangle + \delta|011\rangle + \epsilon|100\rangle + \zeta|101\rangle + \eta|110\rangle + \theta|111\rangle.$$

The probability is 1 that the set will be in one of those eight states. So,

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 + |\epsilon|^2 + |\zeta|^2 + |\eta|^2 + |\theta|^2 = 1,$$

and the probability that the set of three qubits would be measured as 011 is $|\delta|^2$.

It requires 2^n complex numbers (coefficients) to express the state of n entangled qubits. However, if the qubits are not entangled, *i.e.*, are independent of each other, their state can be expressed more compactly, with only $2n$ coefficients. The state of the first qubit can be expressed as

$\alpha_1|0\rangle + \beta_1|1\rangle$. The state of the second qubit can be expressed as $\alpha_2|0\rangle + \beta_2|1\rangle$. The state of the third qubit can be expressed as $\alpha_3|0\rangle + \beta_3|1\rangle$, and so on. However, to specify the state of ten entangled qubits, it requires 1024 coefficients (one coefficient for each of the 2^{10} possible values of ten bits). In contrast, if the ten qubits are unentangled, the state of each of the qubits can be expressed with two coefficients, requiring only 20 coefficients.

While the compact notation using $2n$ coefficients cannot be used to describe an entangled state, the less compact notation using 2^n coefficients can be used to describe unentangled states. In an unentangled state, you can derive the coefficients of all 2^n classical states from the coefficients of $|0\rangle$ and $|1\rangle$ in each of the n unentangled qubit states. For instance, with three unentangled qubits in states $\alpha_1|0\rangle + \beta_1|1\rangle$, $\alpha_2|0\rangle + \beta_2|1\rangle$, $\alpha_3|0\rangle + \beta_3|1\rangle$, respectively, the coefficient of state $|000\rangle$ in the collective state of three qubits would be $\alpha_1\alpha_2\alpha_3$. Likewise, the coefficient of $|001\rangle$ in the collective state would be $\alpha_1\alpha_2\beta_3$, the coefficient of $|010\rangle$ would be $\alpha_1\beta_2\alpha_3$, and so on. (See Homework Problem 2.)

7.1.4 States and Gates as Vectors and Matrices

The quantum state of a collection of qubits can be represented as a column vector of coefficients, one for each bit combination. By convention, we'll order the coefficients according to the corresponding bit combinations, so, for one qubit the order is just 0, 1; for two qubits it's 00, 01, 10, 11. You can think of this as a shorthand to avoid writing out the bit combinations every time. For example, we can write $\alpha|0\rangle + \beta|1\rangle$ as

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

A quantum gate can be represented as a matrix of multipliers. For a single-qubit gate, the first column gives the coefficients of final states $|0\rangle$ and $|1\rangle$ when the initial state is $|0\rangle$, while the second column gives the final state coefficients when the initial state is $|1\rangle$. Again, this is a shorthand to avoid writing out the bit combinations. When we describe gates, we'll show their matrix representations as well. For example, we can write the NOT gate as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

An example of a 2-qubit gate is known as **CNOT (controlled not)** (Figure 7-3), which flips the second qubit between 0 and 1 if the first qubit is 1. It can be defined by how it acts on $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. In the matrix representation, the first column gives the coefficients of final states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ when the initial state is $|00\rangle$. The second column gives the coefficients of final states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ when the initial state is $|01\rangle$, and so forth.

initial state	final state	
$ 00\rangle$	$ 00\rangle$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
$ 01\rangle$	$ 01\rangle$	
$ 10\rangle$	$ 11\rangle$	
$ 11\rangle$	$ 10\rangle$	

Figure 7-3. CNOT Gate

The beauty of this representation is that the application of a gate to a state is accomplished by multiplying the gate matrix times the state vector. For instance, to apply the NOT gate to a qubit in state $\alpha|0\rangle+\beta|1\rangle$, you'd multiply the NOT matrix by the column vector representing the qubit state.

The result of applying a sequence of gates is given by the product of the gate matrices in right-to-left order.

7.1.5 Becoming Superposed and Entangled

Entanglement and superposition are what can make a quantum computer more powerful than a classical computer. How do qubits become superposed and entangled? You'd likely start a computation on a quantum computer by initializing the qubits to a known state, which would be done by measuring all the qubits. Now none of them are entangled, and they are all firmly 0s or 1s.

There are various operations (gates) one can apply to qubits that will cause superposition and/or entanglement. For instance, superposition can be created on a single qubit using a quantum gate known as a **Hadamard gate**, which takes a qubit that is solidly zero or solidly one and sets it to be an equal mixture of 0 and 1. The Hadamard gate operation is shown in Figure 7-4:

initial state	final state	
$ 0\rangle$	$\frac{1}{\sqrt{2}} 0\rangle + \frac{1}{\sqrt{2}} 1\rangle$	$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$
$ 1\rangle$	$\frac{1}{\sqrt{2}} 0\rangle - \frac{1}{\sqrt{2}} 1\rangle$	
truth table representation		matrix representation

Figure 7-4. Hadamard Gate

Gates that operate on a set of qubits can entangle the set. For instance, if you have unentangled qubits x , y , and z and operate on x and y , then x and y may become entangled. If you then operate on y and z , then all three qubits (x , y , and z) may now be entangled. A gate might also unentangle previously entangled qubits.

The truth table representation shows what the Hadamard gate does to a qubit initialized to 0 or 1. If the qubit started out with a superposed state, for instance, $\alpha|0\rangle+\beta|1\rangle$, then to calculate the output of the Hadamard on $\alpha|0\rangle+\beta|1\rangle$, you add the outputs of $|0\rangle$ and $|1\rangle$ in proportion to the coeffi-

cients of the input state. Since the coefficient of $|0\rangle$ in the input is α , and the output of $|0\rangle$ is $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, you multiply the output of $|0\rangle$ by α to obtain $\frac{\alpha}{\sqrt{2}}|0\rangle + \frac{\alpha}{\sqrt{2}}|1\rangle$. Likewise, you multiply the output of $|1\rangle$ (which is $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$) by β .

Adding the two results, the Hadamard gate's output on a qubit with state $\alpha|0\rangle + \beta|1\rangle$ is $\frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle$.

This can also be calculated by multiplying the matrix representation of the Hadamard gate by the column vector representing the qubit state $\alpha|0\rangle + \beta|1\rangle$:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}}\alpha + \frac{1}{\sqrt{2}}\beta \\ \frac{1}{\sqrt{2}}\alpha - \frac{1}{\sqrt{2}}\beta \end{bmatrix} = \begin{bmatrix} \frac{\alpha+\beta}{\sqrt{2}} \\ \frac{\alpha-\beta}{\sqrt{2}} \end{bmatrix}$$

The reason we can multiply the output of a state by the coefficient of that superposed state in the input is due to one of the properties of quantum gates known as *linearity*, described in the next section.

Now, for something really fascinating, what happens when you apply a Hadamard gate twice? Hadamard is its own inverse! (See Homework Problem 7.)

7.1.6 Linearity

Quantum gates satisfy the property of **linearity**. The intuition for linearity is that the output of a quantum gate is a weighted sum of the outputs of all the superposed classical input states, weighted by the coefficients of the superposed classical input states. So, if you know how the gate will operate on the possible classical input states, you can multiply the output of each classical input state by the coefficient of that classical input state, and the sum will be the quantum state of the result.

Linearity is nice, because it means that we can describe n -qubit quantum gates by tables that only describe the outputs produced when the input is one of the 2^n classical states. The table is sufficient to determine what the gate will do to any input state, because all the possible quantum states can be expressed as a superposition (linear combination) of the 2^n classical states. For example, for the CNOT gate (Figure 7-3), if the 2-qubit input state is $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$, linearity implies that the output state is $\alpha|00\rangle + \beta|01\rangle + \delta|10\rangle + \gamma|11\rangle$.

7.1.6.1 No Cloning Theorem

An interesting consequence of linearity is that it is impossible to clone a qubit (or set of entangled qubits). If you had a qubit in state $\alpha|0\rangle + \beta|1\rangle$, you might think you could \oplus it into a qubit in state $|0\rangle$ and wind up with two qubits, each in state $\alpha|0\rangle + \beta|1\rangle$. But instead, you'd wind up with two entangled qubits in state $\alpha|00\rangle + \beta|11\rangle$. (See Homework Problem 8.)

An important implication of the no cloning theorem is that you can't read a quantum state more than once. If you could clone the state, you could make lots of copies, and by reading them, read multiple of the superposed values. If you could read enough copies, you could then derive the probabilities of the various superposed states. But you can't. Reading the quantum state destroys the superposition, and because of no cloning, you can only read it once.

7.1.7 Operating on Entangled Qubits

Quantum gates seldom operate on more than one or two qubits, because the engineering challenge of building a quantum gate increases wildly with the number of qubits it is acting upon. One could imagine a quantum gate that operates on n qubits, but in practice it would almost certainly be built out of gates that operate on one or two qubits at a time. The running time of a quantum algorithm employing a virtual n -qubit gate is adjusted to account for the actual gate configuration. So, given that we will only be using 1- or 2-qubit gates, what happens if you operate on a subset of entangled qubits?

For instance, suppose there were three entangled qubits with state $\alpha|000\rangle + \beta|011\rangle + \gamma|100\rangle$. Now suppose we use a NOT gate on the first qubit. The result would be $\alpha|100\rangle + \beta|111\rangle + \gamma|000\rangle$. See Homework Problem 5 for a more complicated example.

Measurement typically happens one qubit at a time. Suppose you measure one qubit of an entangled set of qubits and get the value 1. A side effect of the operation of measuring that qubit is that the coefficients of all the states that don't match the measured value go to 0, and the coefficients of all the states that do match the measured value are increased linearly to make the sum of their squared absolute values equal to 1. Note that this linear scaling does not change the phases of the remaining coefficients. (See Homework Problem 6.)

7.1.8 Unitarity

Linearity (§7.1.6) constrains the possible things a quantum gate can do, such as preventing cloning qubits. Another constraint on quantum gates is that they must be **unitary**. Unitarity is the property of a linear gate that says if the input state of the gate is normalized (*i.e.*, the sum of the squared absolute values of the coefficients is 1), then the output state is normalized. These constraints are not arbitrarily imposed by us or by some committee. Instead, they result from the known laws of physics.

Unitarity seems like a sensible property. It seems rather absurd to imagine that the probability of all measurement outcomes could add to something other than 1 after you do a physically possible operation. Nonetheless, there are some operations that are forbidden by this constraint that seem

like they should be legal operations. For example, suppose we tried to make a linear gate out of the operation that sets a qubit to 0. We will call this the “zeroize” gate, and its truth table will be

Input	Output
0	0
1	0

Such a gate would violate unitarity, since when we apply the rule for linearity to the above truth table, we find the gate would take the normalized state $\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle$ to the unnormalized state $\frac{3}{5}|0\rangle + \frac{4}{5}|0\rangle = \frac{7}{5}|0\rangle$. (Note that $\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle$ is normalized because $3^2+4^2=5^2$.)

Classical operations such as the zeroize gate that can't be straightforwardly turned into a unitary gate are called **irreversible** operations. What they have in common is that they take at least two different inputs to the same output. In contrast, **reversible** classical operations, those that take every possible input to a different output, can be straightforwardly turned into valid unitary quantum gates. In the next two sections, we will show how to implement even irreversible classical operations with quantum components, first by measurement and next by converting the irreversible classical operation into a reversible classical operation.

7.1.9 Doing Irreversible Operations by Measurement

One way to perform classical operations that are irreversible (from quantum building blocks) is to use measurement. For instance, the zeroize operation can be applied to a qubit as follows:

1. Measure the qubit
2. If you measure a 1, apply the NOT gate: $|0\rangle \rightarrow |1\rangle, |1\rangle \rightarrow |0\rangle$. If you measure a 0, do nothing.

However, this way of implementing classical operations has a major drawback. When you measure a qubit in a superposition of multiple classical states, you only get information about one of the classical states in that superposition. Moreover, if the qubit you measured was entangled with some other qubits, it won't be entangled with them after you make the measurement. Since superposition and entanglement are necessary for quantum computing to be more powerful than classical computing, it would be unfortunate if measurement was the only way to do certain classical computations you wanted to use as part of a quantum algorithm.

7.1.10 Making Irreversible Classical Operations Reversible

When we want to incorporate an irreversible classical computation into a quantum algorithm, it is typical to add some extra qubits to the operation in order to make it reversible (and therefore unitary).

Suppose you wanted to implement an irreversible function that, in the classical world, used i input bits and o output bits. One way that this can be made into a reversible function is by modifying the function so that it takes $i+o$ input bits, computes the irreversible function on the first i bits, and \oplus s the result into the last o bits. If the o bits are initialized to 0, the computation is the same, and the new function is clearly reversible since if you execute it twice you get the original input.

In a quantum version of this, assume the i input qubits are in a superposition and the o output qubits are 0s. After the function is executed, the $i+o$ entangled qubits will be in a superposition of the same number of states, with the same coefficients. Each superposed value of the $i+o$ entangled qubits consists of one of the original superposed values of the i input qubits, with the o qubits entangled with that state corresponding to the output of the function on that input.

There are known algorithms for converting any computable classical function into a similarly efficient quantum circuit that acts in the way we just described.

7.1.11 Universal Gate Sets

The terms *circuit* and *gate* used in quantum computing were chosen to sound familiar to people used to conventional computers. In principle, all classical operations could be built out of NAND gates, and a circuit would carry out a more complex calculation (such as adding together two 32-bit quantities) using a large number of physical gates wired together.

Similarly, in a quantum computer, all circuits can be approximated with a relatively small number of gate types. An example of a **universal quantum gate set** (a collection of gates from which any desired circuit can be constructed) is a Hadamard gate, a CNOT gate, and a $\pi/8$ gate. These gates can be defined as follows:

Hadamard gate

initial state	final state	
$ 0\rangle$	$\frac{1}{\sqrt{2}} 0\rangle + \frac{1}{\sqrt{2}} 1\rangle$	$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$
$ 1\rangle$	$\frac{1}{\sqrt{2}} 0\rangle - \frac{1}{\sqrt{2}} 1\rangle$	

CNOT gate

initial state	final state	
$ 00\rangle$	$ 00\rangle$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
$ 01\rangle$	$ 01\rangle$	
$ 10\rangle$	$ 11\rangle$	
$ 11\rangle$	$ 10\rangle$	

$$\begin{array}{lll}
 & \text{\bf \pi/8 gate} & \\
 \text{initial state} & \text{final state} & \\
 |0\rangle & |0\rangle & \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix} \\
 |1\rangle & \frac{1+i}{\sqrt{2}} |1\rangle &
 \end{array}$$

The kinds of gates implemented in an actual quantum computer would depend on engineering tradeoffs between the cost of having more gates and the performance penalty of requiring more steps to implement a circuit.

The above description is a little misleading because there is a continuum of possible quantum gates, and almost none of them can be constructed from finite sequences of these universal gates. But any possible quantum gate can be approximated as closely as desired with a finite sequence of universal gates.

State and Gate Geometry

In Figure 7-1, we showed the state of a qubit as a point on a unit circle. More generally, since amplitudes can be complex numbers, the state of a qubit is actually a point on a 4-dimensional unit sphere, with the real and imaginary parts of each complex amplitude considered as separate coordinates. Similarly, the state of a collection of n qubits is a point on a 2^{n+1} -dimensional unit sphere. It's a *unit* sphere because states are normalized, *i.e.*, the sum of the squared absolute values of all the amplitudes must be 1.

A quantum gate must be unitary, which means that it is linear and maps states to states. The vector from the origin to any state has length 1, so the gate maps that vector to another vector of length 1. Because it is linear, the gate must map the vector *between* any two states to another of the same length. In other words, the gate preserves the distance between states. So the sphere of states remains rigid under gate application, which means that it can only rotate or flip. And if several states lie in a plane, they will still lie in a (likely different) plane after gate application.

7.2 GROVER'S ALGORITHM

Grover's algorithm [GROV96] is a quantum algorithm that can do brute-force search in the square root of the time it would take on a classical computer. For instance, assume we want to know which n -bit secret key k encrypts plaintext m to ciphertext c . On a classical computer, we could try all possible $N=2^n$ keys, and on average, it would take $\frac{N}{2}$ guesses to find the right key (and worst case, N). On a quantum computer running Grover's algorithm, the number of iterations to get n qubits into a state where it is very probable that the state will be read as the key k is proportional to $\sqrt{N}=2^{n/2}$.

Grover's algorithm begins by initializing n qubits such that if the set were read to produce an n -bit value, each of the $N=2^n$ possible values would be equally likely. It then iteratively boosts the probability of reading the value k (the answer we are searching for), little by little, until after roughly $\sqrt{N}=2^{n/2}$ iterations the likelihood of reading k will be close to 1.

To superpose all $N=2^n$ possible n -bit values onto n qubits, we begin by zeroing each qubit (measuring it, and if it's 1, inverting it) and then applying a Hadamard gate to each qubit. The left-most part of Figure 7-5 shows the amplitudes of each of the $N=2^n$ superposed values at this point, which will be $\frac{1}{\sqrt{N}}=2^{-n/2}$ for each.

Now we will loop over the following two operations (the rest of Figure 7-5):

1. Multiply the amplitude of $|k\rangle$ by -1
2. Reflect the amplitudes of all $N=2^n$ states across the mean of all the amplitudes

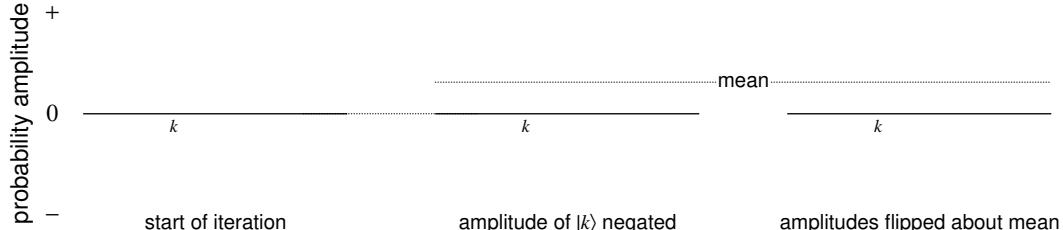


Figure 7-5. First Iteration of Grover's Algorithm, $n=4$, $N=16$, $k=5$

Note that negating the amplitude of one state ($|k\rangle$) is a unitary operation (the probabilities still add up to 1 afterwards) since the squared absolute value of the amplitude of $|k\rangle$ will not change when the amplitude is multiplied by -1 . It may be somewhat mysterious at this point how to create a quantum circuit that will recognize and negate the amplitude of $|k\rangle$, but we'll explain how to do that in §7.2.2.

Now we will reflect each of the amplitudes across the mean of all the amplitudes. The mean will be very close to the common amplitude of every state except $|k\rangle$, because there are $N-1$ of those and only one instance of $|k\rangle$. The mean will be slightly below all of them, because the amplitude of $|k\rangle$ (which is now negative) will bring down the average. You might be skeptical that this operation is unitary, but it is. In the next section, we will give an alternate description of this operation as a sequence of simple unitary operations.

The rightmost part of Figure 7-5 shows the amplitudes after the first iteration. Notice that we've now made all the amplitudes (other than $|k\rangle$'s) smaller by a tiny amount (because the mean was so close to the amplitude of every other state). But we've made the amplitude of $|k\rangle$ a lot bigger, because it was further from the mean. Now, the amplitude of $|k\rangle$ is bigger by approximately a factor of three. Because the probability is the square of the amplitude, the chances of measuring k goes up by approximately a factor of nine after the first iteration. Despite that, it will still be very improbable, if we read the n qubits, that we'll get k (assuming n is large). To make the figure legible, we've used $n=4$ qubits. However, to make Grover's useful, a value of $n=128$ would be more realistic.

The next time we do the two operations

1. Multiply the amplitude of $|k\rangle$ by -1
2. Reflect the amplitudes of all $N=2^n$ states across the mean of all the amplitudes

all the amplitudes (other than $|k\rangle$'s) again get a little bit smaller, and $|k\rangle$'s amplitude increases (but the increase is a little less than before since the mean is a little smaller). After iterating the optimal number of times, the amplitudes of each of the states (other than $|k\rangle$) are nearly 0, and the amplitude of $|k\rangle$ is nearly 1.

Interestingly, if we continue iterating beyond the optimal number of times, the mean becomes negative. Then, when we reflect across the mean, the amplitude of $|k\rangle$ decreases, and the amplitude of the other states increases. If we overshoot the optimal number of iterations, then for a while it will become less and less likely that when we measure the qubits we'll read k . Figure 7-6 shows how the probability of reading k changes as the algorithm proceeds. Note that the probability of reading k increases to a maximum then decreases as more iterations are performed, and then cycles. So it's important to know when to stop iterating in order to have the best chance of reading the correct result. You only get one chance to read a result.

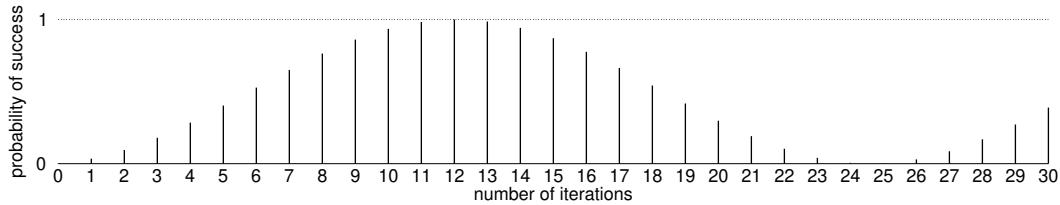


Figure 7-6. Overview of Grover's Algorithm, $n=8$

7.2.1 Geometric Description

In general, a quantum state of n qubits, superposing $N=2^n$ classical states, is represented by a point on a $2N$ dimensional unit sphere. The reason it's $2N$ (rather than N) is that the amplitudes, in the general case, are complex numbers. However, for Grover's, we can focus on just one unit circle on that $2N$ dimensional sphere. That circle lies in a plane we'll call Plane K (because the vertical coordinate is the amplitude of $|k\rangle$). The amplitude of $|\text{anything-but-}k\rangle$ (an equal superposition of all classical states other than $|k\rangle$) is the horizontal coordinate.

The initial state in Plane K (after initializing each qubit to 0 and then applying Hadamard to each) is shown in Figure 7-7. The amplitude of $|k\rangle$ is tiny ($\frac{1}{\sqrt{N}}$). So the state is just slightly above and counterclockwise from the horizontal axis. Let's call that angle a . If we were using Grover's to search for a 128-bit key, angle a would be way too small to show in a diagram, so we've chosen $n=6$ ($N=2^6=64$) for the figures. Figure 7-8 shows a full run of Grover's.

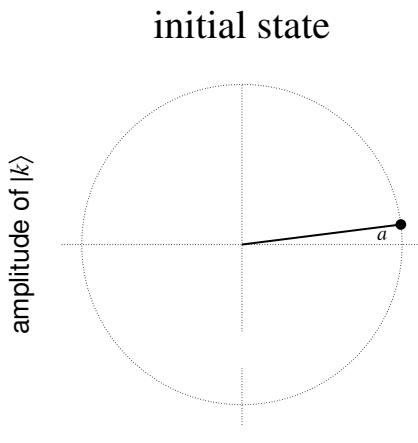


Figure 7-7. Grover's Plane K Initial State

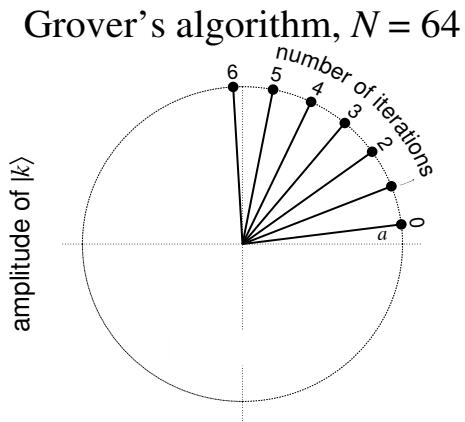


Figure 7-8. Grover's Plane K State Progression

So, the state in Plane K will start at angle a . Each iteration will rotate the state counterclockwise by an angle $2a$. After $\frac{\pi}{4a}$ iterations, the state in Plane K will have traveled roughly $\frac{\pi}{2}$ around the circle, arriving near the top of the circle, so the probability of reading k will be nearly 1.

Since the arc of the circle is nearly vertical at the initial state and levels off to near horizontal near the top of the circle, the amplitude of $|k\rangle$ (the vertical coordinate) increases more at the beginning than it does when the state nears the top of the circle. If there are too many iterations (going past the top of the circle), the amplitude of $|k\rangle$ starts decreasing. If you kept doing iterations, you'd cycle around the circle, with the probability of success decreasing to near 0 before increasing again once the amplitude of $|k\rangle$ goes negative.

That is the high-level summary. Now we will explain how we can manage to do these operations.

7.2.2 How to Negate the Amplitude of $|k\rangle$

We assume that there is some efficiently computable function f that takes an n -bit classical bitstring s and returns 1 if s has the correct value and returns 0 otherwise. For example, if we were searching for an s for which $\text{hash}(s)=h$, then $f(s)$ would return 1 if $\text{hash}(s)=h$ and return 0 otherwise. If we want to know what n -bit secret key encrypts plaintext p to ciphertext c , then $f(s)$ would return 1 if p encrypted with key s is c and return 0 otherwise.

For simplicity, we will assume that we know that s is an n -bit integer and that there is exactly one value, k , such that $f(k)=1$. That is, $f(s)=1$ if $s=k$ and $f(s)=0$ otherwise.

In order to construct a quantum version of f , say f_Q , we will use an extra qubit, which we'll refer to as the **ancilla** qubit, and a quantum circuit that will operate on $n+1$ qubits (the n qubits of s plus the one ancilla). Using the notation $s;b$ to mean the value s together with the ancilla qubit b , f_Q flips the ancilla between 0 and 1 when $s=k$, and leaves the ancilla alone for all other values of s .

After performing f_Q , we will have $n+1$ entangled qubits. If we read the qubits after only a few iterations of Grover's, we'd almost certainly read a value s different from k in the first n qubits and 0 in the ancilla. But if we were lucky enough to read k , we'd get $k;1$.

We now describe how to negate the amplitude of $|k;1\rangle$ and leave the other amplitudes unchanged. We will need a quantum gate called a Z gate, which acts on a single qubit and takes $|0\rangle$ to $|0\rangle$ and $|1\rangle$ to $-|1\rangle$.

To negate the amplitude of $|k;1\rangle$, apply a Z gate to the ancilla. Since the ancilla is entangled with the other qubits, this multiplies the amplitude of state $|k;1\rangle$ by -1 and does not change any of the amplitudes for $|s;0\rangle$. (All states other than $s=k$ will have the ancilla equal to 0.)

Now apply f_Q again. This leaves the ancilla unchanged for all values except k but flips the ancilla from 1 back to 0 for $|k\rangle$. Now the ancilla is solidly 0 for all states and is therefore no longer entangled with the other qubits. (See Homework Problem 11.)

The circuit described above is not the most efficient way to negate the amplitude of $|k\rangle$, since it requires us to compute f_Q twice. There is a cleverer way to negate the amplitude of $|k\rangle$. It turns out that if we set the ancilla qubit to Hadamard($|1\rangle$) = $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$, then we only need to apply f_Q once. It's not obvious, but it works (see Homework Problem 12).

So now we know how to negate the amplitude of $|k\rangle$, or to put it another way, we know how to reflect the amplitude of a selected state (in this case $|k\rangle$) across the horizontal axis.

7.2.3 How to Reflect All the Amplitudes Across the Mean

This is accomplished in a way very similar to negating the amplitude of $|k\rangle$ but is slightly more complicated to explain. What we really want to do is reflect the state across the line showing the initial state in Figure 7-7. We know from the previous section (§7.2.2) how to reflect an amplitude across the horizontal axis in a plane. But the line in Figure 7-7 that we want to reflect across is not horizontal.

The trick we will use is to visit another plane, which we'll call Plane Z (for *zero*), in which the horizontal axis is the amplitude of $|0\rangle$, and the vertical axis is the amplitude of $|\text{anything-but-}0\rangle$. The line we wanted to reflect across in Plane K will be the horizontal axis in Plane Z.

Figure 7-9 shows the initial state (in Plane Z) at the beginning of Grover's, when all the qubits are initialized to zero—the amplitude of $|0\rangle$ is 1, and the amplitude of all other states is 0.

To travel between Plane Z and Plane K, we apply a Hadamard to each of the n qubits. As the state travels from Plane Z to Plane K, it rotates counterclockwise by angle a . If we apply Hadamards while in Plane K, it will return to Plane Z, rotated clockwise by angle a .

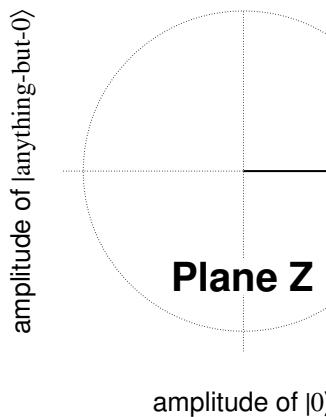


Figure 7-9. Grover's Plane Z Initial State

The line we want to reflect across in Plane K is now horizontal in Plane Z, and we know how to reflect states across the horizontal axis.

We mark which states we wish to reflect across the horizontal axis by flipping the ancilla between 0 and 1 for all states other than $|0\rangle$. We use a quantum function, say g_Q , which flips the ancilla between 0 and 1 when the value of the n qubits is not equal to 0, and leaves the ancilla alone for $|0\rangle$.

Now we apply a Z gate to the ancilla, and we've reflected all values (other than $|0\rangle$) across the horizontal axis. Then we need to apply g_Q again to reset the ancilla to 0 for all states.

Now we return to Plane K by applying Hadamards to the n qubits. The combination of the three operations

- apply Hadamards to get from Plane K to Plane Z
- reflect all states other than $|0\rangle$ across the x axis
- apply Hadamards to get back to Plane K

accomplishes the goal of reflecting all states across the mean of their amplitudes in plane K.

This is depicted graphically in Figure 7-10.

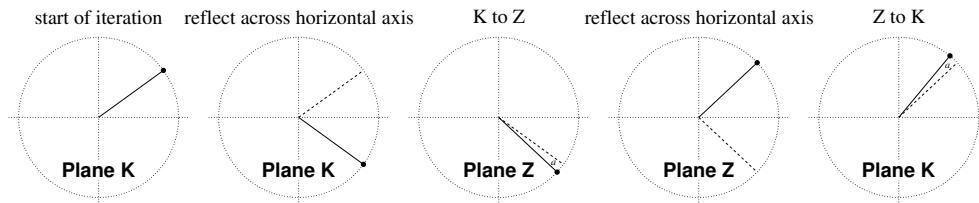


Figure 7-10. One Iteration of Grover's

7.2.4 Parallelizing Grover's Algorithm

Grover's algorithm speeds up key search quadratically. Does that mean that we need to double our keysizes for secret key algorithms? Not necessarily. Apart from the very real possibility that quantum gates might be more difficult to build than their classical counterparts, there is a more fundamental difference between Grover's algorithm and classical key-search algorithms that makes Grover's less practical—Grover's algorithm doesn't parallelize well.

To provide a concrete example, DES with its 56-bit key is considered to be completely insecure. However, if all you had was a single CPU, trying one key after another, it would take decades to get through all the keys, even if it was a pretty good CPU. Nonetheless, DES keys are routinely broken in a matter of hours. This is done by using multiple CPUs (or GPUs, or specialized hardware) to try many keys at once. For example, if you can try a hundred thousand keys at the same time, you can break a DES key in a few hours. There is no similar way to speed up Grover's algorithm.

You could, of course, divide the key space into segments. For example, if you tried to search for a 128-bit key using a million quantum processors running in parallel, each with a few hundred qubits, you could get the first processor to run Grover's algorithm under the assumption that the first twenty bits of the key were $0\cdots 00$, you could get the next one to assume the first twenty bits were $0\cdots 01$, and so on. However, doing this would only reduce the number of iterations of Grover's algorithm by a factor of about a thousand (from about 2^{64} to about 2^{54} .) You made your computer a million times bigger, but it only made the search a thousand times faster. One might hope that there would be a more efficient way to parallelize Grover's algorithm. However, Zalka [ZALK99] showed that there isn't one.

Therefore, doubling the size of secret keys and hashes is more than sufficient to defend against the threat of quantum computers running Grover's algorithm.

7.3 SHOR'S ALGORITHM

Shor's algorithm [SHOR95], which can efficiently factor integers and find discrete logarithms, is the most important application of quantum computing for cryptography. Shor's algorithm, combined with a general-purpose quantum computer of a few thousand logical qubits and a quantum program that applies billions of logical gates (see §7.6 *Quantum Error Correction*), would render insecure all current widely deployed public key cryptography. Shor's algorithm finds the period of a periodic function. We'll explain in §7.3.1 how RSA relates to a periodic function, and we'll explain in §7.3.2 why knowing the period of that function helps us factor the RSA modulus.

Then starting in §7.3.3, we describe how Shor's algorithm can factor a b -bit RSA modulus n . Typically, RSA moduli are 2048 bits long, but to make the numbers easy, we'll use $b=2000$ in examples. The version of Shor's algorithm that finds discrete logs is a little more complicated but conceptually similar to the one for factoring.

7.3.1 Why Exponentiation mod n Is a Periodic Function

An RSA modulus n is (usually) the product of two odd primes p and q , i.e., $n=pq$. As we described in §6.2 *Modular Arithmetic*, $\phi(n)$ (the totient function) is the number of positive integers less than n that are relatively prime to n . For $n=pq$, $\phi(n)=(p-1)(q-1)$. Euler's theorem states that for any a relatively prime to n , $a^{\phi(n)}=1 \pmod{n}$. So for any value x , $a^x \pmod{n} = a^{x+\phi(n)} = a^{x+2\phi(n)} = a^{x+3\phi(n)}$, and in general, for any integer k , $a^x \pmod{n}$ equals $a^{x+k\phi(n)}$. In other words, $a^x \pmod{n}$ is a periodic function of x that repeats every $\phi(n)$.

In fact, although $a^x \pmod{n}$ indeed repeats every $\phi(n)$, the actual period will be a divisor of $\lambda(n)$, the least common multiple of $p-1$ and $q-1$. So $\lambda(n)$ is smaller than $\phi(n)$ by a factor of $\gcd(p-1, q-1)$. For instance, since p and q will be odd numbers, both $p-1$ and $q-1$ will be even (having a common factor of 2), so $\lambda(n)$ can be at most half of $\phi(n)$. Also, if a is, for instance, a square mod n , the period of $a^x \pmod{n}$ will be half that of its square root, so the period might again be halved. Shor's algorithm will compute the period of $a^x \pmod{n}$. We'll call the period d .

7.3.2 How Finding the Period of $a^x \pmod{n}$ Lets You Factor n

First we'll show how, knowing d (the period of $a^x \pmod{n}$) for a suitable choice of a , you can find some number y that is a nontrivial square root of $1 \pmod{n}$, i.e., a number, other than $\pm 1 \pmod{n}$, such that $y^2=1 \pmod{n}$. Then we'll explain why knowing y lets you factor n .

Because d is the period of $a^x \pmod{n}$, we know that $a^d \pmod{n}=1$. If $a^d \pmod{n}=1$, then if d is even, $a^{d/2} \pmod{n}$ is a square root of 1. By the Chinese remainder theorem (§2.7.6), there are four ways that a number y can be a square root of $1 \pmod{n}$ when $n=pq$:

1. y is $1 \pmod{p}$ and $1 \pmod{q}$. In this case, y will be $1 \pmod{n}$.
2. y is $-1 \pmod{p}$ and $-1 \pmod{q}$. In this case, y will be $-1 \pmod{n}$.
3. y is $1 \pmod{p}$ and $-1 \pmod{q}$. In this case, y will be a nontrivial square root of $1 \pmod{n}$.
4. y is $-1 \pmod{p}$ and $1 \pmod{q}$. In this case, y will be a nontrivial square root of $1 \pmod{n}$.

In the first case ($y = 1 \pmod{n}$), if the exponent is even, you can divide the exponent by 2 again to get $a^{d/4} \pmod{n}$ and hope to get a nontrivial square root of 1. But if the exponent is odd, or in the second case ($y = -1 \pmod{n}$), you can't go further, and you need to choose a different a and start over.

Note that there's no reason to start with $a^{d/2} \bmod n$. Instead, you can first find the largest odd divisor of d by dividing d by 2 until you get an odd number, say w . Then you calculate $a^w \bmod n$. If it's 1, choose a different a . Otherwise, keep squaring the result until you get 1 or -1. If you get -1, choose a different a . If the first time you get 1 is after calculating, say, $a^{8w} \bmod n$, then $a^{4w} \bmod n$ is your nontrivial square root of $1 \bmod n$. (It's no coincidence that this procedure is reminiscent of the Miller-Rabin primality test [RABI80] described in §6.3.4.2.1 *Finding Big Primes p and q* .)

So now you've found a nontrivial square root of n . Let's call that y . By definition of being a square root of 1, $y^2=1$, or equivalently, $y^2-1=0$. Factoring y^2-1 , the equation becomes $(y+1)(y-1) \bmod n=0$.

If the product of two numbers $\bmod n$ is 0, there are two cases:

- One of the numbers ($y+1$ or $y-1$) is $0 \bmod n$.
- One of the numbers is a multiple of p , and the other is a multiple of q .

But since y is a nontrivial square root of 1, we know it is the second case. Therefore, taking $\gcd(y+1, n)$ or $\gcd(y-1, n)$, you'll get p or q . And, of course, once you know one of the factors of n , you can divide by that to get the other.

7.3.3 Overview of Shor's Algorithm

For factoring a b -bit RSA modulus n , we'll need $3b$ qubits. The first $2b$ qubits, which we'll call the **exponent qubits**, will be initialized to hold an equal superposition of all 2^{2b} classical values of t . Note that we're using a double-length exponent t so that the periodic function $a^t \bmod n$ repeats many times. The remaining b qubits, which we'll call the **result qubits**, will be initialized to 0. We will $\oplus a^t \bmod n$ into the b result qubits. We will then have $3b$ entangled qubits.

Assuming a 2000-bit modulus, that would be 6000 qubits—4000 exponent qubits to hold t , plus 2000 result qubits to hold $a^t \bmod n$. We'll use N to denote the number of classical values of the exponent t . So $N = 2^{2b}$, (or 2^{4000} for a 2000-bit n).

Initialize all $3b$ qubits to 0 (measure each and invert if not 0). Then set the exponent qubits to an equal superposition of all N possible classical values of t by applying a Hadamard to each one.

Next, we randomly choose an x relatively prime to n , and use a quantum circuit that calculates $x^t \bmod n$ and \oplus s the result into the result qubits. Now we have $3b$ entangled qubits, where the $2b$ exponent qubits hold an equal superposition of all N classical values of t , and the b result qubits hold $x^t \bmod n$ for each possible value of t .

In other words, if we were to measure the $3b$ qubits at this point, we'd get some random number j in the $2b$ exponent qubits, and $x^j \bmod n$ in the b result qubits. That would not be very useful. Using a classical computer, we could have picked a random number j , computed $x^j \bmod n$, and gotten the same information. So we're not going to do that.

Instead of measuring all $3b$ qubits at this point, we are only going to measure the b result qubits. We'll get some random uninteresting number u . Because the $2b$ exponent qubits are entangled with the b result qubits, the result of reading the result qubits and getting the value u is that the amplitudes of all the classical states of t for which $x^t \bmod n$ is not equal to u go to zero. The resulting graph of the amplitudes of all the superposed values of t will be a periodic function with spikes every d . For reasons that will become clear later, we call this graph (Figure 7-11) the *time graph*. If we could now somehow read the value of d , we'd be able to factor n . Although the spikes will be d apart, the location of the first spike will be at some random value τ , which is the first value of t for which $x^t \bmod n = u$. So reading the qubits at this point will not let us find d .

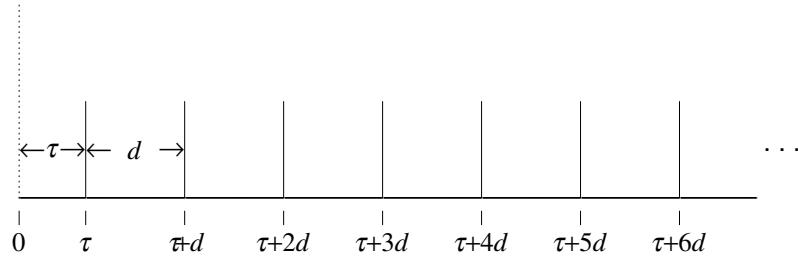


Figure 7-11. Time Graph

If we could get the locations of two of the spikes, we could take the difference, which would be a multiple of d , and that would be useful. But, once we measure, all the other amplitude spikes disappear, and due to the no-cloning theorem (§7.1.6.1), we can't copy the state before measuring.

Shor, inspired by discrete Fourier transforms, observed that there is a way to convert the quantum state from what we have in the time graph into a new quantum state whose graph (Figure 7-12) we'll call the *frequency graph* (for reasons that will become clear later). The nice thing about the frequency graph is that it also has spikes at regular intervals, ($f=N/d$ apart), but the first spike

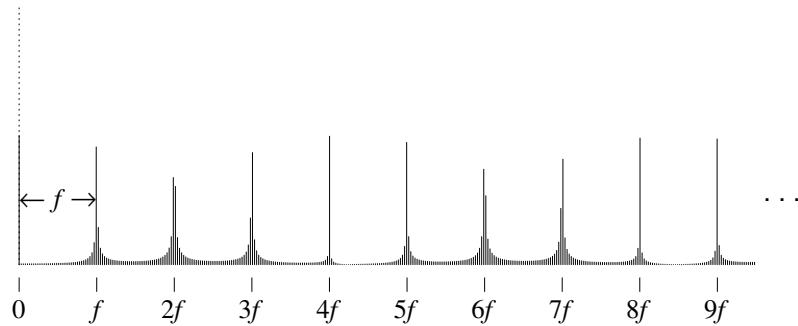


Figure 7-12. Frequency Graph

occurs at the value 0, so each spike is at an integer multiple of f . That means that if we read the state at this point, we'll almost certainly get a number that is a multiple of f .

There are a few subtleties. Unfortunately, N/d will almost certainly not be an integer, since N will almost certainly not be a multiple of d . As a result, the spikes in the frequency graph are actually spread over a few values, as you can see from Figure 7-12. But that's okay. If we read the qubits, with high probability we'll read a value v sufficiently close to a multiple of $f=N/d$ that we will be able to compute d with a simple classical algorithm using the continued fraction of N/v .

Another subtlety is that, in the time graph, all the amplitudes are positive real numbers. In the frequency graph, the amplitudes are complex numbers. In Figure 7-12 we're only showing the absolute values of the spikes in the frequency graph, because that's all that matters. The only thing we do once the quantum state is as represented by the frequency graph is read the qubits.

The next section will give a *classical* algorithm, hopefully optimized for comprehensibility, for converting the time graph into a frequency graph. This is not how a quantum computer would do it, or even how a classical computer would do it, but rather how it's simplest to understand. The description involves doing things a quantum computer can't do (like reading the amplitudes) and a thoroughly impractical number of operations (on the order of 2^{8000} to factor a 2000-bit number). On a quantum computer, Shor's algorithm for factoring a b -bit number would run in time proportional to b^3 . In fact, the most expensive part of Shor's algorithm is the modular exponentiation, not the conversion of the time graph to the frequency graph.

7.3.4 Converting to the Frequency Graph—Introduction

We'll call the values plotted along the horizontal axis in the time graph t and the values plotted along the horizontal axis in the frequency graph s . Both t and s are integers between 0 and $N-1$. When we start, the quantum state of the $2b$ qubits of t is represented by the time graph. Later, the quantum state of the same $2b$ qubits will be represented by the frequency graph.

Throughout this section, we will use vectors to represent complex numbers. A complex number $x+yi$ can also be described as a vector (x,y) . The vector (x,y) starts somewhere on a plane and goes x to the right and y up.

The amplitude at each s in the frequency graph will be the sum of a bunch of vectors, as we'll describe shortly, normalized so that the s probabilities add to 1. The phase of the sum (the direction in which the vector points) does not affect the probability of reading the value. However, Shor's algorithm demonstrates how an amplitude can be a complex number. If the normalized sum vector of s is the vector (a,b) , that means the amplitude of $|s\rangle$ is the complex number $a+bi$.

To add a set of vectors, you add the x values to get the x value of the sum and add the y values to get the y value of the sum *e.g.*, $(x_1,y_1)+(x_2,y_2)+(x_3,y_3)=(x_1+x_2+x_3, y_1+y_2+y_3)$. In our case, all the vectors being added into the sum for a given s will have the same magnitude but may be pointing in different directions. If n vectors point in the same direction, their sum will be a vector n times

as long (see Figure 7-13a). If two vectors point in opposite directions, say (x,y) and $(-x,-y)$, they will cancel each other out, resulting in a zero-length vector. Likewise, if n equal-sized vectors are equally spaced around a circle, they will cancel each other out (see Figure 7-13c). If the n vectors are equally spaced within an arc of a circle, then the direction of the sum vector will be in the middle of the arc (see Figure 7-13b), and the length of the sum will depend on how large the arc is. If the arc is smaller, the sum is larger.

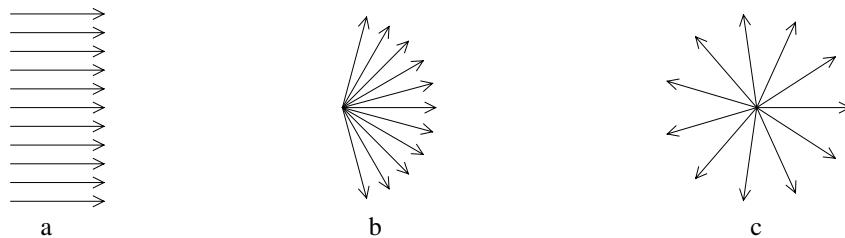


Figure 7-13. Adding Vectors

7.3.5 The Mechanics of Converting to the Frequency Graph

Imagine the time graph as a giant springy string with spikes corresponding to the spikes in the time graph. To calculate the amplitude at s in the frequency graph:

- Initialize the sum vector S to $(0,0)$.
- Stretch the time graph string so that it wraps around the circle exactly s times.
- For each spike in the string, add to S the vector pointing in the same direction as the spike, and of length equal to the size of the spike (in our case, all the spikes are the same length).
- Normalize S by dividing by \sqrt{N} so that the sum of the squared absolute values of the S 's is 1.

For $s=0$, the string is shrunk to a point, so the spikes all point in the same direction. See Figure 7-14. Note the vector inside the circle in Figure 7-14 is the normalized sum of the spikes.)

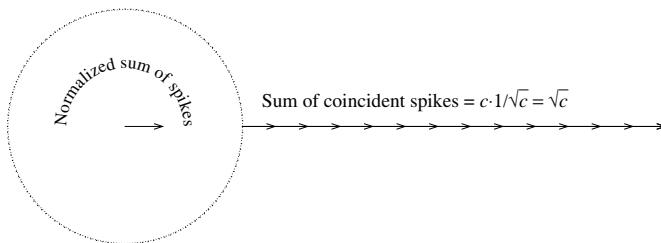


Figure 7-14. Frequency Graph Calculation for Stretch Factor $s = 0$

For $s=1$, the string wraps around the circle exactly once. Since the spikes are spaced symmetrically around the circle, as vectors they will essentially cancel each other out, and so the resulting amplitude will be very nearly 0. (See Figure 7-15.)

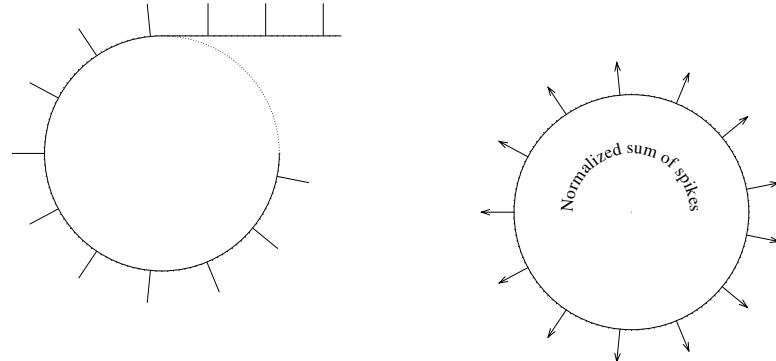


Figure 7-15. Frequency Graph Calculation for Stretch Factor $s = 1$

Similarly, for $s=2$ (see Figure 7-16), the normalized sum of spikes is basically zero, so all we see inside the circle in Figure 7-16 is a point.

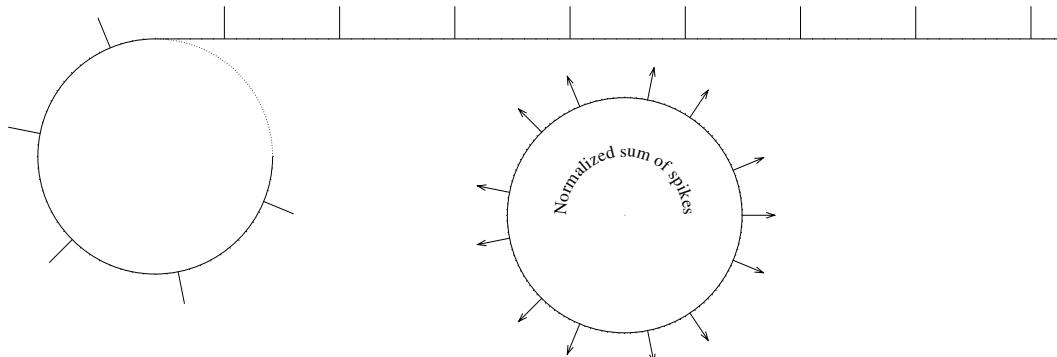


Figure 7-16. Frequency Graph Calculation for Stretch Factor $s = 2$

But for $s \approx f$ (or s being close to an integer multiple of f), the spike hits the circle in approximately the same place on each wrap. Since f is not usually an integer, s (when s is close to a multiple of f) is either a little smaller than a multiple of f or a little larger than a multiple of f . If smaller, each successive spike will undershoot the previous spike a little, while if larger, each successive spike will overshoot the previous spike a little. In either case, the spikes will fall on an arc of the circle. The length of the sum vector will be somewhat smaller than if they were pointing in exactly the same direction. (See Figure 7-17.) The case where the height of the spike will be smallest is when a multiple of f is a half integer so that the spikes span half the circle, reducing the magnitude of the sum by a factor of roughly $\frac{2}{\pi}$ (see Figure 7-18.). Even for values of s for which the spikes go completely around the circle, they might not exactly cancel out, so the sum could be very small but

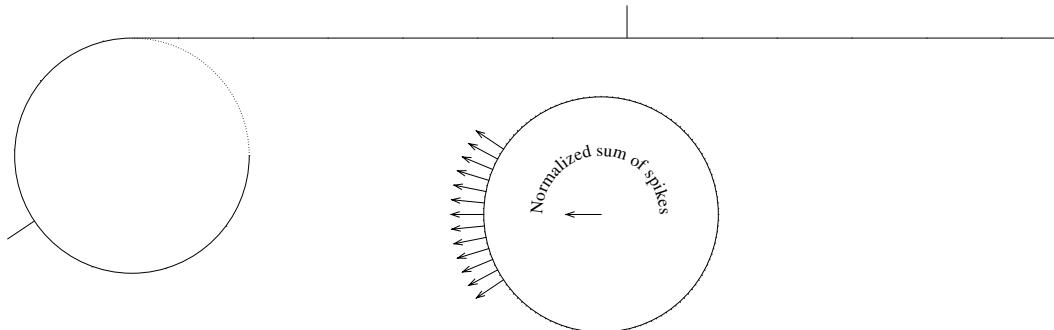


Figure 7-17. Frequency Graph Calculation for a Stretch Factor s near f

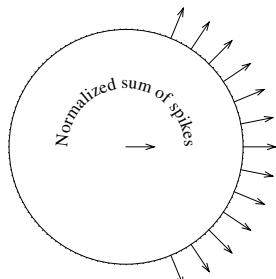


Figure 7-18. Frequency Graph Calculation for a Stretch Factor s nearly $\frac{1}{2}$ away from a multiple of f nonzero. Visit <https://rawcdn.githack.com/ms0/docs/main/dft.html> for an animation of this process.

When we've computed the amplitude for each s , we have the frequency graph. As we've seen, it will have spikes near each integer multiple of f . The differences between the time graph (Figure 7-11) and the frequency graph (Figure 7-12) are

- The time graph starts with a spike at a random place that depends on the value read from the result qubits. The frequency graph starts with a spike at 0.
- The spikes in the time graph are spaced d apart. The spikes in the frequency graph are spaced $f = \frac{N}{d}$ apart.
- Each spike in the time graph is a single value of t . Each spike in the frequency graph comprises a few values of s near an integer multiple of f , with the values of s closest to such a multiple providing the largest amplitudes.
- The time graph amplitude at each non-spike t is 0. The frequency graph amplitudes away from each spike are nearly 0.
- The spikes in the time graph all have identical amplitudes (both magnitude and phase). The spikes in the frequency graph vary in magnitude and phase.

7.3.6 Calculating the Period

With high probability, when we read the qubits in the state represented by the frequency graph, we will get a value v that is close to an integer multiple of f , where $f = \frac{N}{d}$. What we want is the denominator d .

There is a small probability that the v that we read will be 0, which is not useful, but given that there are so many other spikes with similar sizes, we almost certainly won't be that unlucky. There's also a small probability that we'll read a value that isn't near a spike, because, although small, those still have nonzero amplitudes. If we manage to read a non-useful value, we just run the algorithm again (starting with randomly choosing a number a to exponentiate).

Once we read a value v that is close to an integer multiple of f , we need to calculate d . Remember that $f = \frac{N}{d}$. We expand $\frac{N}{v}$ as a continued fraction and use that to get good rational approximations. The numerator of one of those rational approximations is usually going to be d , although it might occasionally be a divisor of d .

The choice to use a $2b$ -bit exponent to factor a b -bit modulus is made because that makes it very likely that the measured v is close enough to an integer multiple of f for the continued fraction technique to efficiently find d .

7.3.7 Quantum Fourier Transform

The algorithm we've described for transforming the time graph into the frequency graph is actually performing a discrete Fourier transform. Although I_{3,4} cannot imagine anyone that isn't completely comfortable with discrete Fourier transforms, I_{1,2} spent a lot of energy on the previous section tricking potentially math-phobic readers into understanding discrete Fourier transforms before surprising them by telling them, in this section, that they now understand them.

A discrete Fourier transform computes a frequency graph from a time graph using a formula that might look intimidating at first glance.

$$\tilde{\alpha}(s) = \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} \alpha(t) \cdot e^{-2\pi i \frac{st}{N}}$$

But let's break the formula down. It is describing the computation we described in §7.3.4. The amplitude of each s (written as $\tilde{\alpha}(s)$ to the left of the equal sign) is the sum of N values, each of which depends on $\alpha(t)$ (the amplitude of t in the time graph) from $t=0$ to $t=N-1$.

To compute $\tilde{\alpha}(s)$, we are going to travel around a unit circle and, for each t , calculate a complex number to add to our sum. For each of the N values of t , the arc that we travel over is s/N of the circle. So for $s=1$, we travel around the circle exactly once, while for $s=3$, we travel around the circle exactly three times. Also, recall that the circumference of a unit circle is 2π , and the units of distance along the circumference that add up to 2π (on a unit circle) are known as *radians*. That means

that for a given value of s , we want to travel s/N of the circle for each increment of t , i.e., $2\pi s/N$ radians for each increment of t . So after t steps we've traveled $2\pi s t/N$ radians.

We've explained most of the exponent of e in the formula. The minus sign is the convention for the (forward) Fourier transform. Without the minus sign, it would be equivalent to wrapping around the circle counterclockwise. Each $\tilde{\alpha}(s)$ would have the same magnitude with or without the minus sign, but its phase would be different, since it would be reflected across the horizontal axis. For our purposes, only the magnitude matters. (It turns out that the inverse Fourier transform, which recovers the time graph from the frequency graph, just leaves out the minus sign.)

But what is i , the square root of -1 , doing in the exponent? This is an application of Euler's formula, which states that for any real number θ , $e^{i\theta} = \cos\theta + i \sin\theta$. What Euler's formula says is that there are three ways of expressing a point on a unit circle:

- as coordinates: $(x, y) = (\cos\theta, \sin\theta)$
- as $e^{i\theta}$
- as the complex number $\cos\theta + i \sin\theta$

Each value we add into the sum is the complex number representing the angle at which t hits the unit circle times the amplitude at t in the time graph. While in this case the amplitudes in the time graph are real, in general, when you multiply complex numbers, you multiply their magnitudes and add their angles.

The only remaining part of the formula is dividing by \sqrt{N} to make the probabilities add up to 1. So the resulting transform is unitary!

7.4 QUANTUM KEY DISTRIBUTION (QKD)

In 1984, Bennett and Brassard [BENN84] came up with a way for two parties to establish a secret that would be impossible for an eavesdropper to see, by leveraging an existing shared key together with a medium (such as a fiber optic link) for sending quantum information (such as a stream of photons). The goal is similar to using authenticated Diffie-Hellman, leveraging an existing secret, to create a new secret for perfect forward secrecy.

The theory behind quantum key distribution is that if an eavesdropper Eve attempted to read the secret bits Alice is sending to Bob, Eve would introduce enough errors in the information stream that Alice and Bob would detect the interference based on seeing more than the expected number of errors (and they would fail to agree on a new secret key).

The idea is fairly simple. Alice will carefully send a single photon at regular time intervals, randomly polarizing that photon to be up ($|$), 45° ($/$), -45° (\backslash), or sideways ($-$).

Bob will read the photon after passing it through a polarizing filter in one of two positions he randomly chooses at each time interval—vertical or 45° diagonal. If Bob chooses to read one of the photons with a vertical filter, then

- If Alice sent $|$, Bob will see the photon.
- If Alice sent $-$, Bob will definitely not see the photon.
- If Alice sent \backslash or $/$, then Bob has probability $\frac{1}{2}$ of seeing the photon.

Similarly, if Bob chooses to use a 45° diagonal filter, then

- If Alice sent $|$ or $-$, Bob has probability $\frac{1}{2}$ of seeing the photon.
- If Alice sent $/$, Bob will see the photon.
- If Alice sent \backslash , Bob will not see the photon.

Next, Bob tells Alice which sequence of filter positions he used, say $||/-||//|/\|$. This message (where Bob reports the filter positions he chose) must be authenticated using the pre-shared secret we are assuming Alice and Bob have been configured with before this exchange. Otherwise, our eavesdropper, Eve, can simply do this protocol acting as a meddler-in-the-middle, establishing a separate secret between Bob-Eve and Eve-Alice.

When Bob reads with a filter $\pm 45^\circ$ off from the polarization of the photon, it's totally random whether he sees a photon or not, so those bits are useless. Therefore, Alice checks the sequence of filter positions that Bob reports and tells Bob to ignore bits for which his filter was $\pm 45^\circ$ off of the polarization of the photon Alice transmitted.

If Alice sent $|$ or $-$ and Bob chose $/$ for that time slice, Alice tells Bob to ignore that bit. Similarly, if Alice sent $/$ or \backslash and Bob chose $|$ for the filter, Alice tells Bob to ignore that bit.

For the remaining bits, Bob (or Alice) sends a checksum computed using an error-correcting code. Since sending single photons will be a somewhat noisy channel under the best of circumstances (no eavesdropper), they will expect a certain number of errors, and the error-correcting code should be able to correct more than the expected number of benign errors but not as many errors as would be introduced by an eavesdropper.

Why would eavesdropper Eve introduce errors? She might try to read the photons, and allow a photon that makes it through her filter to continue on to Bob. However, she has to guess what configuration to set her filter. If the photon passes through her filter, and the filter is not exactly aligned with the photon, she'll have twisted the photon, so that even if Bob's filter is aligned with Alice's choice, Bob may not be able to read the now-twisted photon. Or if Bob should have read a 0, the now-twisted photon might pass through Bob's filter.

7.4.1 Why It's Sometimes Called Quantum Encryption

Why do people worry about making high performance QKD (as described above)? If QKD were only used to send, say, 256 bits, to be used as a secret key to encrypt data over a traditional communication channel, high performance of the quantum channel is unnecessary.

A system has **information-theoretic security** if even an attacker with unlimited compute power cannot get any information from seeing ciphertext (*i.e.*, there are no attacks based on going through all possible keys and recognizing plaintext when the correct key is tried). \oplus with a one-time pad has that property. If the goal is to use the quantum channel to establish a one-time pad (and achieve information-theoretic security), the name *quantum encryption* makes sense. The one-time pad needs to be as long as the amount of data to be sent. High performance is required in this case, because the quantum communication channel (over which the one-time pad will be established) will need to have several times the desired bandwidth of the channel over which the encrypted data will be sent, because of all the bits that will need to be discarded or used for error correction.

7.4.2 Is Quantum Key Distribution Important?

Quantum key distribution has been highly advertised as an important breakthrough with guaranteed security, but in our opinion, it's not as useful or as secure as claimed.

QKD depends on having established a pre-shared secret. If you believe that cryptography works, any mechanism described elsewhere in this book for Alice and Bob to securely communicate will work. And these traditional mechanisms can work over a traditional multi-hop network, whereas QKD requires (at a minimum) a very expensive special direct link. At the time of our writing, deployed QKD systems have additional problems. Typically, end-to-end QKD only works over a limited distance (no more than a few hundred kilometers). Middleboxes, called **trusted repeaters**, that are capable of decrypting traffic are required for longer-distance communication. QKD systems may also be vulnerable to side-channel attacks that learn the shared secret, not by measuring the transmitted photons, but by observing the behavior of the transmitting and receiving devices.

7.5 HOW HARD ARE QUANTUM COMPUTERS TO BUILD?

We haven't talked much about how a quantum computer might be implemented. We have implicitly promised that all these superposed and entangled manipulations of quantum information are consis-

tent with physics as we know it, but this raises the question: If all this is consistent with physics, why hasn't anyone built a quantum computer big enough to break 2048-bit RSA? What's so hard about it? What are the challenges involved, and how might they be overcome?

The first challenge is coming up with physical systems that can reliably store and manipulate qubits. Then the qubits must be isolated from their environment, because any interaction between a qubit and its environment affects the state of the qubit. But, in order to apply gates, it is necessary to interact with the qubits in a carefully controlled way.

One way of isolating qubits from their environment is to choose qubits that interact very weakly with their environment, *e.g.*, polarized photons in empty space or in optical fiber. This makes the qubits very hard to manipulate, especially when we need to apply two-qubit gates that require the qubits to interact with each other. Often, reducing unwanted interactions between qubits and the environment requires extreme cooling. At normal temperatures, there are too many particles bouncing around to allow qubits to stay unmeasured for very long. Qubits encoded in very small physical systems like atoms and ions can sometimes deal with higher temperatures provided they are separated from other atoms by a significantly large region of high-quality vacuum.

Unfortunately, even without a noisy environment, most qubits will decay (or *decohere*) on their own. If the $|1\rangle$ state is slightly more energetic than the $|0\rangle$ state, for example, it will have a certain nonzero probability of decaying to the $|0\rangle$ state in any given interval of time, and that probability will be directly proportional to the energy difference between the two states. The downside, however, of having a small energy gap between possible states of the qubit is that it tends to make applying gates to the qubit very slow.

Despite all these challenges, there are a number of ways to create qubits that do the right thing about 99% of the time. Serious proposals have been based on photons traveling through an obstacle course of half-silvered mirrors and polarizers (optical quantum computers), manipulating the states of valence electrons in trapped ions using lasers (ion-trap quantum computers), nanoscale circuits involving semiconductors (single-electron transistors, quantum dots) or superconductors (superconducting quantum computing), and pushing around excitations of planar solids in braid-like trajectories (topological quantum computers).

At the time of this writing, the most promising candidate is superconducting quantum computing. Here the qubits are represented by tiny oscillating currents in superconducting circuits. The circuits need to be cooled to around .01 kelvin. The qubits can interact using resonators and they can be manipulated by hitting them with lasers or simply by running a pulse of current near them.

So, qubits that can be manipulated to do the right thing 99% of the time have been produced (with some difficulty), but this on its own is not good enough. A cryptographically relevant quantum computation like Shor's algorithm involves billions of gates. Doing something a billion times that has a 1% chance of completely ruining your computation is pretty much guaranteed to ruin your computation. Solving this problem has led to the development of advanced schemes for achieving fault tolerance using error-correcting codes (see §7.6 *Quantum Error Correction*), where a group of flaky physical qubits act as a single well-behaved qubit known as a **logical qubit**.

The challenges involved in building a quantum computer appear daunting but not insurmountable. In fact, it seems increasingly likely that a cryptographically relevant quantum computer will be built in the next few decades, and if built, such a computer would render all widely used public key cryptography insecure. We live in interesting times.

7.6 QUANTUM ERROR CORRECTION

When building conventional computers, we can build circuitry whose error rate is almost arbitrarily low, but sometimes it is more cost effective to build something with a higher error rate and use error-correcting codes to compensate for the small number of expected errors. In traditional communications and classical memory, typically less than 10% overhead is needed for error-correction bits. Some classical technologies, such as DRAM, need to be refreshed by being read and rewritten periodically, because otherwise the state decays.

Quantum error correction is much harder, for various reasons:

- The intrinsic error rate is much higher (at least given current qubit and quantum gate technology). While a conventional computer might have one error bit in a million, all known quantum gates have error rates no better than one in a thousand.
- A quantum state cannot be read and rewritten to refresh the state before it decays beyond repair (as is done with classical DRAM).
- A conventional computer bit can only have one kind of error (a bit is flipped). In contrast, a qubit can drift into an infinite number of states, and its entanglement can change.

It is somewhat surprising that quantum error correction is possible at all. Quantum error correction is an active area of research. With all known technology, qubits have a very high error rate. Therefore, the only hope for doing complex computations is to have a set of intrinsically flaky physical qubits act as a group to behave like a single more-stable logical qubit. A quantum error correction scheme is characterized by a **threshold**, which is the maximum error rate for physical qubits that will result in logical qubits being more reliable than the physical qubits they are based on. When physical qubits and gates have error rates below the threshold, it is possible to adjust the error correction method to make the error rate for logical qubits arbitrarily low without requiring exponentially more physical qubits than logical qubits. As of 2022, the best quantum error correction schemes have a threshold corresponding to something like a 1% error rate.

How much error correction is needed (and the size of the physical qubit group creating a logical qubit) depends on the fidelity of the gates acting on the physical qubits and how many gates the logical qubits need to experience while keeping the resulting quantum states sufficiently accurate. While some of these quantities are subject to change due to better technology or improvements in

optimizing quantum algorithms, academics have been giving estimates for some time. These can perhaps give some idea of the scale of the numbers involved and how much or little they've changed over the years. For example, a highly cited 2012 paper [FOWL12] said a practical quantum computer for factoring would require at least a hundred million physical qubits, while a more recent analysis from 2019 [GIDN21] said factoring a 2048-bit RSA number would only require about twenty million physical qubits. These numbers may be compared to the several thousand logical qubits required to run Shor's algorithm against a 2048-bit RSA number.

Acting on one or two logical qubits with a logical gate to perform the equivalent of acting on physical qubits with a physical gate requires more physical gates. Therefore, the physical qubits must have high enough fidelity to withstand the extra gates required to have the logical qubit group be operated upon by at least one logical gate. If acting on a logical qubit with a logical gate has sufficient fidelity, then the error correction algorithm can be applied recursively to produce a higher level of logical qubits with higher fidelity still. This in theory results in logical qubits with arbitrarily high fidelity. There is a trade-off between the fidelity that can be achieved by the physical qubits and the number of them needed. The frontiers of quantum computer design involve increasing both the number of qubits and their fidelity.

For a simplified example of a quantum error correction algorithm that will correct for a single qubit's bit flip (flipping between $|0\rangle$ and $|1\rangle$) but not correct for other types of errors such as changes in phase (where the coefficient of $|0\rangle$ or $|1\rangle$ is multiplied by a complex number of absolute value 1), use three physical qubits (which we'll call a , b , and c) to represent one logical qubit. The correct state of the three qubits is that they are supposed to be equal, so the two superposed states are $|000\rangle$ (indicating the logical qubit is $|0\rangle$) and $|111\rangle$ (indicating the logical qubit is $|1\rangle$). So the quantum state of the entangled group of three qubits might be $\alpha_1|000\rangle+\beta_1|111\rangle$.

Some quantum gates can act on logical qubits in a way that doesn't cause single qubit errors to propagate from one physical qubit to the other physical qubits in the same logical qubit. For example, a NOT gate can be performed by applying NOT to each of the three physical qubits. If there were no errors, a , b , and c should all be identical after the gate, so the only two superposed states should still be $|000\rangle$ and $|111\rangle$, possibly with different coefficients, say $\alpha_2|000\rangle+\beta_2|111\rangle$. In fact, any quantum gate that is equivalent to a classical gate can be performed in this way. In other words, our simplified example is good enough to do error-corrected *classical* computation.

We assume the probability of more than one bit flip is very low. Suppose qubit b suffers a bit flip, so now the state might be $\alpha_2|010\rangle+\beta_2|101\rangle$. If we read any of the qubits, we'll get 0 or 1 and destroy the superposition. So instead, there is a clever way of detecting which of the qubits disagrees, by measuring both $a \oplus b$ and $b \oplus c$ (in a nondestructive way*). If there was no error, both measurements ($a \oplus b$ and $b \oplus c$) will be 0; if bit a flipped, we will measure 1 and 0; if bit b flipped, we will measure 1 and 1; and if bit c flipped, we will measure 0 and 1. Thus, we can apply a NOT

*Note that to measure $a \oplus b$, you use an extra qubit, let's call it d , initialized to 0. First, apply a CNOT gate to a and d , then apply a CNOT gate to b and d .

gate to flip the affected physical qubit, and the state of the logical qubit will be repaired to be $\alpha_2|000\rangle + \beta_2|111\rangle$.

Unfortunately, there are ways a single qubit can be damaged other than a simple bit flip, which is why the above triplet scheme does not work in practice. For example, an error could turn $|0\rangle$ into $\frac{3}{5}|0\rangle - \frac{4}{5}|1\rangle$ and turn $|1\rangle$ into $\frac{4}{5}|0\rangle + \frac{3}{5}|1\rangle$. Or it could turn $\alpha|0\rangle + \beta|1\rangle$ into $\alpha|0\rangle - \beta|1\rangle$. The error correction schemes that work correctly are somewhat more complicated to explain and use more qubits to constitute a logical qubit. Those error correction schemes limit the variety of 1- and 2-qubit gates that can be fault tolerant. However, there are error correction schemes that allow a universal gate set. That is to say, they cannot produce any 2-qubit gate you might write a truth table for, but they can, with logarithmic overhead, produce something that is close enough to the desired gate. So even if many of these gates are required, the overall computation will succeed with high probability.

7.7 HOMEWORK

1. Suppose you use four polarizing filters, with each one 30° off from the previous one. What percentage of photons would pass through the four filters? (See §7.1.3.1.)
2. Suppose there are three unentangled qubits with qubit 1 having state $\alpha_1|0\rangle + \beta_1|1\rangle$, qubit 2 having state $\alpha_2|0\rangle + \beta_2|1\rangle$, and qubit 3 having state $\alpha_3|0\rangle + \beta_3|1\rangle$. Write out the amplitudes of each of the eight possible classical values for the three qubits, expressed in terms of $\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3, \beta_3$.
3. Suppose we make a 2-qubit gate that (on classical inputs) \oplus s the value of qubit 1 into qubit 2, while leaving qubit 1 alone. Write out the truth table for this 2-qubit gate. Have we seen it before? What is the output of this gate when qubit 1 and qubit 2 are initialized to Hadamard($|0\rangle$) = $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and Hadamard($|1\rangle$) = $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$, respectively? Are the two qubits entangled in the output state? Did the state of qubit 1 change? What about qubit 2?
4. Consider a Hadamard gate operating on a qubit with real amplitudes α and β , so the state of the qubit is a point on a unit circle. The Hadamard gate reflects the state across a line through the origin. At what angle is this line? (Hint: Where does the Hadamard move the state $|0\rangle$? Where does it move the state $|1\rangle$?)
5. Suppose we have three entangled qubits in state $\alpha|001\rangle + \beta|010\rangle + \gamma|100\rangle$. What would the state of the qubits be after operating on the first qubit with a Hadamard gate? (Hint: Compute the result for each of the three superposed states and add the results in proportion to their coefficients. For instance, operating on the first qubit of $\alpha|001\rangle$ with a Hadamard results in $\frac{\alpha}{\sqrt{2}}|001\rangle + \frac{\alpha}{\sqrt{2}}|101\rangle$.)

6. Suppose we have the same entangled set of three qubits in state $\alpha|001\rangle + \beta|010\rangle + \gamma|100\rangle$. What is the state of the set of three qubits after we measure the first qubit and get 1? What would the state of the set of three qubits be if we measured the first qubit and got 0?
7. For each of the following states of a qubit, show the result of applying Hadamard once, then show the result of applying Hadamard twice:
 - a. $|1\rangle$
 - b. $|1\rangle$
 - c. $\alpha|0\rangle + \beta|1\rangle$
8. What is the state of two unentangled qubits, each in state $\alpha|0\rangle + \beta|1\rangle$ expressed as coefficients of $|00\rangle, |01\rangle, |10\rangle$, and $|11\rangle$?
9. Show that any linear quantum gate on n qubits whose truth table is a one-to-one mapping of classical input states to classical output states is unitary.
10. If you choose two random 128-bit blocks x and y , is it always true that there is exactly one 128-bit AES key k that maps plaintext x to ciphertext y ? How about with 256-bit AES?
11. In §7.2.2, what would happen if instead of performing f_Q a second time, you simply read the ancilla, and if it's 0, then leave it as 0, and if it's 1, then perform NOT on the ancilla?
12. Show that the optimization in §7.2.2 (initializing the ancilla to Hadamard($|1\rangle$) and performing f_Q once) indeed negates the amplitude of $|k\rangle$.

8

POST-QUANTUM CRYPTOGRAPHY

As we described in Chapter 7 *Quantum Computing*, a sufficiently large quantum computer implementing Shor’s algorithm would break our currently deployed public key algorithms. However, long before that can happen, the world will (hopefully) have converted to replacement algorithms. The replacement algorithms will be based on math problems that (hopefully) not even a combination of classical and quantum computers would be able to solve in a reasonable amount of time.

These new algorithms are known by several equivalent names: quantum-resistant, quantum-safe, or post-quantum cryptography (PQC). The world seems to have settled on the term *post-quantum*, so that is what we will use, even though we have noticed that the term *post-quantum* sometimes confuses people into thinking these new algorithms run on quantum computers.

It is important to start migrating away from the current public key algorithms well before a sufficiently large quantum computer might exist. One reason is that conversion will be slow. Another reason is that there might be data that should be kept secret for many years, and if that data were encrypted with today’s public key algorithms, it could be stored now and decrypted later, when quantum computers do exist. Given that it is not possible to know when (or even if) quantum computers might realistically threaten our current public key algorithms, people should convert to new public key algorithms very soon. However, before conversion to new algorithms can happen, the world needs to standardize on some replacement algorithms.

The National Institute of Standards and Technology (NIST) has played an important role in standardization of cryptography (*e.g.*, AES and SHA) and is playing an important role in the standardization of post-quantum algorithms. In late 2017 (the deadline for submissions), NIST received about 80 proposed schemes. Rather than picking a single “winner” scheme, as NIST did for AES, NIST will standardize several. There would be no way to pick a single “best” scheme, because the post-quantum algorithms have such different properties, such as dramatic differences in key size, signature size, and computation required.

The intention of this chapter is to give some intuition into how these algorithms work and to be comprehensible to people who have not recently been taking advanced math courses. For those who want mathematical rigor, complete specifications, and security proofs, there are (and will be)

many excellent resources. For example, NIST’s post-quantum site [NISTPQC] gives pointers to detailed information about all the submissions.

In subsequent sections, we’ll discuss four of the best-known families of schemes: hash-based (§8.2), lattice-based (§8.3), code-based (§8.4), and multivariate cryptography (§8.5).

In addition to the four major types of post-quantum algorithms we discuss at length in this chapter, there is a fifth major type—isogeny-based cryptography. An isogeny is a particular kind of mapping between elliptic curves. Isogeny-based cryptography uses elliptic curves, but unlike traditional elliptic curve cryptography, does not depend on the difficult of the (elliptic curve) discrete log problem. (Recall that the elliptic curve discrete log problem would be easily solvable with a quantum computer using Shor’s algorithm.) Isogeny-based cryptography, in contrast, relies on the hardness of various problems such as constructing an isogeny between two specific elliptic curves that are known to have an isogeny between them. Isogeny-based cryptography is relatively well studied. A number of promising schemes have been proposed, including the encryption scheme Supersingular Isogeny Key Exchange (SIKE), which is, at the time of this writing, an alternate candidate in the NIST PQC standardization process. Several other encryption and signature schemes have also been proposed using isogenies. Proposed schemes typically have small public keys, ciphertexts, and signatures compared to other post-quantum schemes, but also tend to be slower, although much progress has been made in creating more efficient implementations of isogeny-based schemes. We omit more extensive discussion of isogeny-based cryptography, not because we think it is less important than the other major areas of post-quantum cryptography, but because we think providing enough mathematical background to do these schemes justice will seem tedious to most of our readers, and readers who are interested will have many places to read the details.

8.1 SIGNATURE AND/OR ENCRYPTION SCHEMES

With RSA, any key pair can be used for either signatures or encryption. It’s not considered good security practice to use the same key pair for both purposes, but mathematically, it can be done. In contrast, most post-quantum algorithms are only good for one or the other (signatures or encryption). That means that in the post-quantum world, entities that do both signatures and encryption will likely need not just two different key pairs, but also two different algorithms.

Any public key scheme requires an algorithm for key pair generation. A signature scheme additionally requires algorithms for signature generation and signature verification. An encryption scheme requires algorithms for encryption and decryption. Since public key schemes are slower than secret key schemes, a public key signature scheme will usually only sign a hash of the

message, and a public key encryption scheme will usually only use the public key encryption to establish a secret S and then encrypt the message with S .

Sometimes cryptographers refer to a third type of scheme, **key encapsulation mechanism (KEM)**. KEM schemes can be converted to encryption schemes and vice versa. The distinction is that with an encryption scheme, one side chooses S and encrypts S using the other party's public key. In contrast, in a KEM scheme, a shared secret is derived from information that one or both sides contribute to the exchange. We will not use the term *KEM* further, and simply refer to both types of schemes as *encryption*.

8.1.1 NIST Criteria for Security Levels

NIST defined five security strength categories as a coarse-grained measure of how hard it would be to break the algorithm. The post-quantum algorithm submissions are specified with parameter sets for some or all the categories. *e.g.*, to meet category 1, use these values of the parameters. The categories are defined as:

1. At least as hard as key search against a 128-bit block cipher (*e.g.*, AES128)
2. At least as hard as collision search against a 256-bit hash function (*e.g.*, SHA256)
3. At least as hard as key search against a 192-bit block cipher (*e.g.*, AES192)
4. At least as hard as collision search against a 384-bit hash function (*e.g.*, SHA384)
5. At least as hard as key search against a 256-bit block cipher (*e.g.*, AES256)

In terms of security against classical attack, categories 1 and 2 are essentially the same (128-bits of security), as are categories 3 and 4 (192 bits of classical security). However, when we consider quantum attacks (*e.g.*, Grover's algorithm), category 2 might be considered more secure than category 1, and category 4 might be considered more secure than category 3.

8.1.2 Authentication

With all public key schemes, Alice and Bob need to reliably know each other's public keys in order to do secure signatures, encryption, or authentication. An example method is having them present certificates to each other, signed by an entity the other trusts. The public key of the certifying entity might be embedded in the software. At any rate, this is not an issue specific to post-quantum public key algorithms, and we will not discuss it further.

8.1.3 Defense Against Dishonest Ciphertext

Public key encryption schemes have to be carefully designed to defend against a type of attack known as a **chosen-ciphertext attack**. An attacker, say Trudy, might gain information by sending Alice ciphertexts that were not generated according to the rules of the algorithm. This attack usually requires Trudy sending many (*e.g.*, millions of) ciphertexts to Alice and depends on Trudy being able to figure out from Alice’s behavior whether the decryption succeeded. In some public key encryption algorithms, there is a possibility that even honestly generated ciphertexts might fail to decrypt, and Trudy may be able to gain information even if she does generate her ciphertexts according to the rules of the algorithm, provided she sends enough ciphertexts so that it is likely that Alice will fail to decrypt some of them. The probability that an honestly generated ciphertext will fail to decrypt is called the **decryption failure rate**.

The information that Trudy might gain with this attack might allow her to learn Alice’s private key, or, if Trudy sees an encrypted message that honest Bob had sent to Alice, Trudy might learn Bob’s plaintext by sending lots of slight variants of Bob’s ciphertext and seeing whether Alice can decrypt them.

This attack is not specific to post-quantum encryption algorithms. For instance, an attack of this form, known as the **million-message attack**, was successfully demonstrated against a particular implementation of RSA [BLEI98].

Any public key encryption algorithm should contain explicit countermeasures to this sort of attack. One simple countermeasure to prevent chosen ciphertext attacks is having Alice change her public key for every single communication, but this can be impractical or hard to enforce. In cases where algorithm designers want it to be safe to reuse a key pair, there are provably secure systematic constructions for providing security against chosen ciphertext attacks. The general principle behind these constructions is for Alice to refuse to decrypt a ciphertext unless she can verify that it was honestly generated. Typically, the way Bob proves to Alice that he generated his ciphertext honestly is as follows:

- Bob will typically need to generate random numbers in the process of creating a ciphertext. Instead of generating these numbers truly randomly, he will pseudorandomly derive them from a seed in a way that is dictated by the algorithm specification.
- Rather than directly encrypting a plaintext or a shared secret directly, Bob will instead encrypt the seed from which other quantities are derived.
- After Alice decrypts the ciphertext to obtain Bob’s random seed, she verifies that the ciphertext sent by Bob is indeed derived from the seed. If so, she proceeds to pseudorandomly derive a shared secret (also in a way dictated by the algorithm specification) from the seed.
- If the recreated ciphertext does not match what Alice received, Alice does not want to give an attacker any information about whether or why the decryption failed. This behavior of Alice is known as **implicit rejection**. This means that rather than sending an error message, Alice

pretends everything is fine and uses a random number for the shared secret, so whoever is speaking with Alice will perceive this as Alice speaking gibberish (*i.e.*, Alice communicating using an encryption key the other side does not know). In contrast **explicit rejection** means that Alice sends an error message or terminates the connection. Using implicit rejection instead of explicit rejection is believed to make it easier to avoid implementation errors that might unintentionally leak information to an attacker.

All the encryption schemes that made it to the third round of NIST’s PQC Standardization process use this or a similar construction. It should be noted that these sorts of constructions only effectively protect against chosen ciphertext attacks if the decryption failure rate for honestly generated ciphertexts is very low, *e.g.*, one failed decryption per 2^{128} ciphertexts. Therefore, rigorous analysis of the decryption failure rates of these schemes is necessary for analyzing their security, and not just their reliability.

For simplicity, when presenting the various families of post-quantum schemes, we will present simplified versions that do not include these protections against chosen ciphertext attacks.

8.2 HASH-BASED SIGNATURES

This family of algorithms is based on the assumption that it is computationally infeasible to find a pre-image of a hash h , *i.e.*, a value v , such that $\text{hash}(v)=h$. The proposed hash-based algorithms are only for public key signatures (not encryption).

Cryptographic hashes have been well studied, and are believed to be quantum-resistant. They are a conservative choice since, if properly implemented, they are less likely to be broken than other signature schemes—hash-based signature schemes only rely on the security of a hash function, while other signature schemes rely not only on the security of a hash function, but also on the hardness of one or more additional computational problems. There are two types of hash-based signatures schemes:

- **Stateful**, meaning that a signer must be extremely careful to keep track of how many items were previously signed, or else the scheme is insecure.
- **Stateless**, meaning that a signer need not keep track of how many things it has signed in the past.

The stateful schemes are vastly more efficient than the stateless ones, but stateful implementations have to be incredibly careful to keep accurate state. It might be easy for an implementation to lose track of how many things have been signed if a process crashes and restarts or if there are multiple instances of a service using the same public key.

We'll start our description with schemes that are impractical but easy to understand, and then explain some optimizations to make these more efficient.

Hash-based signatures tend to be described with lots of parameters. This enables variants that can trade off things like security, signature size, and computation, but all this flexibility can lead to descriptions that are hard to read. Since we're primarily intending to give intuition rather than exact specification, we'll choose numbers for parameters to make our descriptions more readable.

8.2.1 Simplest Scheme – Signing a Single Bit

The first concept is a one-time signature of a single bit. An example of signing one bit of information is where Alice will announce which of two candidates won an election—candidate 0 or candidate 1. Alice's private key will consist of two randomly chosen 256-bit numbers p_0 and p_1 , and her public key will consist of the 256-bit hashes of each of those— h_0 and h_1 . In other words, $\text{hash}(p_0)=h_0$, and $\text{hash}(p_1)=h_1$. To sign the value 0, Alice reveals the pre-image of h_0 , namely p_0 . To sign the value 1, Alice reveals the pre-image of h_1 , namely p_1 .

With this scheme, Alice can only sign one bit, since knowledge of both p_0 and p_1 allows someone to sign either a 0 or a 1 bit. If Alice wants to certify the results of another election, she needs to create a new public key. Assuming we are using 256-bit ps and hashes, the size of a public key would be 512 bits (consisting of h_0 and h_1), a private key would also be 512 bits (consisting of p_0 and p_1), and a signature would be 256 bits (either p_0 or p_1 , depending on whether she was signing 0 or 1).

8.2.2 Signing an Arbitrary-sized Message

Next, let's build on the single-bit scheme to enable Alice to sign a single arbitrary-sized message. As with other public key algorithms that we are familiar with (*e.g.*, RSA), Alice doesn't directly sign the message, but rather, signs a hash of the message. For example, if she uses the hash algorithm SHA-256, she'll need to be able to sign 256 bits.

To be able to sign 256 bits, Alice chooses two 256-bit secrets for each of the 256 bits. So, for bit i , she'll choose the pair of secrets $\langle p_0^i, p_1^i \rangle$. If bit i is 0, she'll reveal p_0^i . If bit i is 1, she'll reveal p_1^i . This will result in 512 secrets that we'll call $\langle p_0^0, p_1^0 \rangle, \langle p_0^1, p_1^1 \rangle, \dots, \langle p_0^{255}, p_1^{255} \rangle$. She hashes each of these 512 secrets to obtain $\langle h_0^0, h_1^0 \rangle, \langle h_0^1, h_1^1 \rangle, \dots, \langle h_0^{255}, h_1^{255} \rangle$. That's 512 hashes, and the hash of all of them, $H = \text{hash}(h_0^0 | h_1^0 | h_0^1 | h_1^1 | \dots | h_0^{255} | h_1^{255})$, will be her 256-bit public key.

To sign a 256-bit quantity, say 001011101…1, Alice does the following for each of the 256 bits: If the i th bit is 0, Alice reveals p_0^i and h_1^i . If the i th bit is 1, Alice reveals h_0^i and p_1^i . Bob (the verifier) hashes each of the 256 p^i 's Alice reveals, in order to now know all the 512 hashes $(h_0^0, h_1^0,$

$h_0^1, h_1^1, \dots, h_0^{255}, h_1^{255}$). Then Bob computes $\text{hash}(h_0^0 | h_1^0 | h_0^1 | h_1^1 | \dots | h_0^{255} | h_1^{255})$ and verifies that the result is indeed Alice's public key H .

Alice can only sign one message digest, because if she signs two different message digests, where bit i is 1 in one digest and 0 in the other, both p_0^0 and p_1^0 are revealed, and then someone can sign either a 0 or 1 for bit i . Note that each time Alice signs something, she reveals half of the pre-images. If she were to sign two messages with this scheme, then probably half of the bits in the hash of the second message to be signed will be equal to those bits in the first message she signed, so there would still be a quarter of the pre-images that were not yet revealed. If Eve wants to forge Alice's signature, and Eve knows all 512 pre-images, she can sign anything. But if there are, say n unrevealed pre-images, then Eve has to keep testing messages until she can find one whose hash has bits for which all the pre-images were revealed.

For this scheme, the size of a public key is 256 bits, the size of a private key is 128K bits (256 $\langle p_0^i, p_1^i \rangle$ pairs, which is 256×512 bits), and the size of a signature is also 128K bits. Verifying a signature requires hashing the message and doing an additional 256 hashes (of either the pre-image p_0^i or p_1^i , depending on whether that bit in the message hash is a 0 or 1, and then hashing the 512 hashes).

8.2.3 Signing Lots of Messages

A scheme that only allows Alice to sign a single message would not be very useful. So we'll modify the scheme, using hash trees (also known as Merkle trees; see §5.4.8) to allow Alice to sign a lot of messages. In Figure 8-1, she can sign four different messages (each with a 256-bit digest), and her public key is still just 256 bits (the value of the tree root).

Starting at the top of the tree, Alice's public key is H_{Alice} , the root node. H_{Alice} is equal to $\text{hash}(H_0 | H_1)$. H_0 is $\text{hash}(H_{00} | H_{01})$. Each of the four nodes at the second level equals the hash of its 512 children. This tree will allow Alice to sign four different messages using the subtrees H_{00} , H_{01} , H_{10} , or H_{11} .

We'll use the terminology of a *treelet* as the bottom level of the tree (with 512 children). In order to make a tree with n levels have 2^n treelets, we won't count the root or the leaves as a level. Therefore, the tree in Figure 8-1 has two levels and four treelets.

Alice can use the treelet under H_{01} to sign a message as follows. For each of the i bits in the message digest of the message to be signed, if the i th bit is 0, then Alice reveals p_0^i and h_1^i . If the i th bit is 1, then she reveals h_0^i and p_1^i . Bob (the verifier) hashes each p to obtain the corresponding h , so Bob now knows all 512 children of H_{01} and therefore can compute H_{01} . Bob also has to be told H_{01} 's sibling (H_{00}) so he can compute H_0 (which is $\text{hash}(H_{00} | H_{01})$), and has to be told H_0 's sibling (H_1) so he can compute $\text{hash}(H_0 | H_1)$ and verify that the result is H_{Alice} .

To summarize, a public key in this scheme is a 256-bit hash, the root of the hash tree. A signature (of a 256-bit digest) consists of 512 256-bit quantities—for each bit, a pair $\langle p_0^i, h_1^i \rangle$ if the

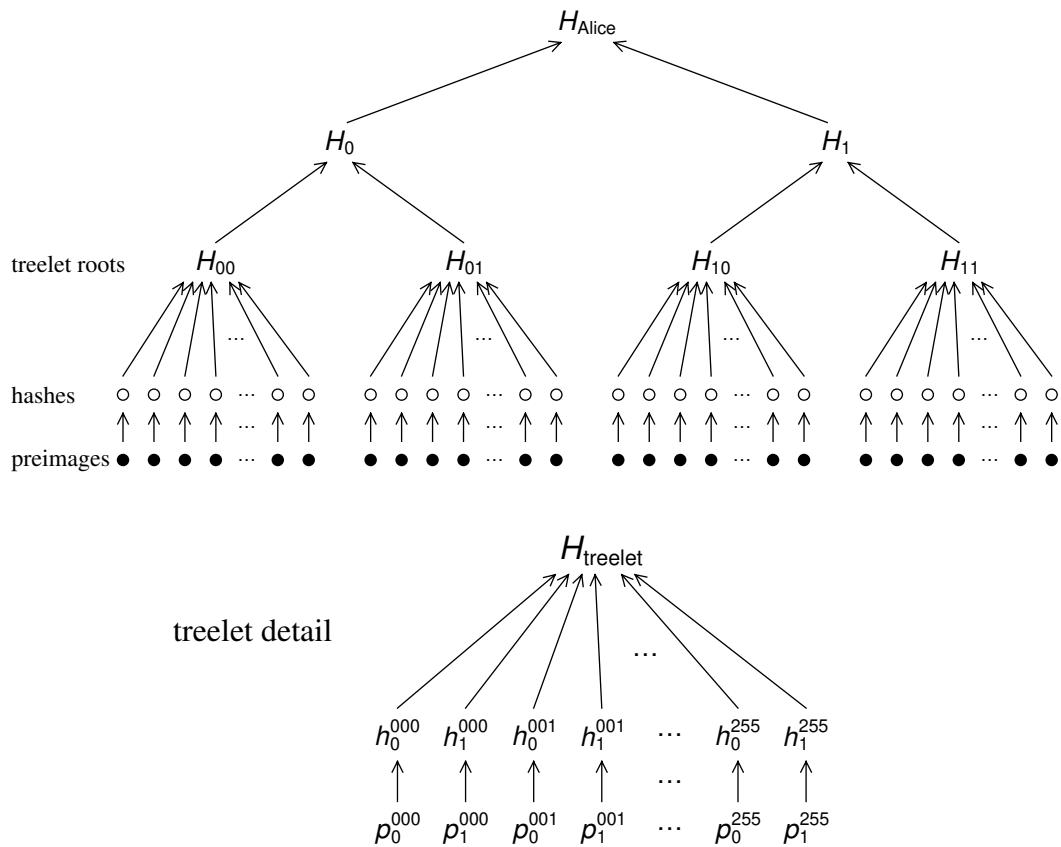


Figure 8-1. Hash Tree for Signing

bit is 0, or $\langle h_0^i, p_1^i \rangle$ if the bit is 1. The signature also needs the sibling hashes in the hash tree, consisting of a 256-bit quantity for each level in the tree. Even for deep trees, signatures will only be a little larger than 128K bits (16K octets).

Suppose Alice wants to be able to sign a really large number of messages, say a million. Generating and storing the 20-level hash tree will be expensive (more than 130 billion bits—a million times 512 times 256). Another issue is that Alice might not know how many messages she will need to sign. She could overestimate, at the expense of creating and storing a bigger tree than she needed. But what if the number of messages exceeds the size of the tree she initially computed?

Suppose she built a tree big enough for 1024 signatures with root R_1 . Her public key would then be R_1 . What can she do if she needs to sign more messages than that?

After signing 1023 messages, she can create a new hash tree with 1024 treelets and root R_2 and use the last treelet in her first hash tree to sign the root of the new hash tree (effectively signing

a message saying “ R_2 is also my public key”). This enables her to sign an additional 1023 messages using R_2 , plus leave room for signing the root of a third hash tree for the next 1023 messages. Signatures for the first 1023 messages would consist of the 512 values (p_0^i and h_1^i , or h_0^i and p_1^i) for each of the 256 bits of the hash, plus the sibling hashes of the ancestor nodes in the first tree. This works out to 133632 bits, or 16704 octets.

For the next 1023 messages, the signature would have to be double-sized, because not only would it need to consist of the signature of the message being signed using the second tree, but also the signature for the root of the second tree (that was signed using the last treelet in the first tree). This comes out to 33408 octets.

Then for the next 1023 messages, the signature would be triple-sized, and so forth.

An alternative strategy is that Alice could use the first tree only to sign roots of 1024 trees, with each tree having 1024 treelets. Then she’d be able to sign a million messages, each of whose signatures would be 33408 octets. If that’s not enough, a three-level structure would allow a billion signatures, each of which would be 50112 octets.

8.2.4 Deterministic Tree Generation

Generating a tree deterministically from a secret seed S enables Alice to only need a small secret (e.g., 256 bits), which will be used to generate the hash trees as needed, instead of having to pre-generate and store all the hash trees. This saves Alice storage for her private key but costs computation because she needs to regenerate the entire tree to sign anything.

To generate a tree with ten levels, Alice uses S to compute each of the leaf pre-image values as a function of S , the treelet number, and an indicator of whether the pre-image is for 0 or 1. For example, p_1^7 of treelet 42 might be $\text{hash}(S|7|1|42)$.

Alice will need to generate the entire tree in order to sign a message. But after signing (and carefully remembering how many messages she has signed), she can forget the tree she generated and just remember S and the number of messages she has signed. There is a trade-off between how much data she stores and how much computation she does. If she remembers all the peer hashes from the hash tree but recomputes the treelet she needs for each signature, she will need only a moderate amount of stored data and computation.

Suppose she is using a two-level tree of trees (where the treelets in the first tree are used to sign second-level trees). For each signature, she will need to generate (or store) the upper tree and the one lower tree she is currently working on. The secret for generating each tree has to be unique, of course, so the single secret S would have to be hashed with the tree number to create the secret for generating that tree.

As with the single-tree case, a lot of computation can be saved by keeping track of all the sibling hashes already computed and computing sibling hashes for new trees as needed.

This concept of deterministic tree generation will be particularly important for the stateless schemes (see §8.2.7.1 *Stateless Schemes*).

8.2.5 Short Hashes

Using shorter hashes, *e.g.*, 128 bits instead of 256 bits, would obviously improve performance. Treelets could have half as many leaf nodes, and each leaf value could be half the size. But can a scheme that uses 128-bit hashes have a security level of 128 bits? Recall that finding a hash collision for an n -bit hash requires work $2^{n/2}$, whereas finding a pre-image of a specific n -bit hash value requires work 2^n .

First, let's make it safe for Alice to sign 128-bit hashes of messages rather than 256-bit hashes, where by “safe” we mean maintaining NIST’s security category 1. We need to ensure that an attacker has to find a pre-image of a specific 128-bit hash rather than find any 128-bit hash collision.

We can force Ivan (the attacker) to find a pre-image of a specific hash instead of finding any collision by doing the following. To sign a message, Alice chooses a random number R (similar to an IV), and her signature consists of R , the message, and the signature on $\text{hash}(R|\text{message})$. This solves the collision issue, since Ivan cannot, in advance, find any two messages with the same hash—he has to wait for Alice to sign a message and choose an R , and then Ivan has to find a pre-image of $\text{hash}(R|\text{message})$.

Since Ivan cannot predict which random R Alice will choose, he cannot create an evil message with the same hash as the combination of R and a message Alice will sign. Instead, Ivan will have to find an evil message and an R such that $\text{hash}(R|\text{message})$ is one of the hashes Alice has already signed. If Alice only signed a single message, Ivan would have to match the one hash she signed (*i.e.*, find a pre-image of that hash), and this would be secure.

There is an additional attack we should be aware of that makes finding a pre-image a little easier. This is known as the **multi-target attack**. Suppose Ivan doesn’t need to find a pre-image of one specific hash, but instead has a list of a million possible hashes for which finding a pre-image will work for his purposes. For example, Alice might have signed a million different messages, and Ivan, with a list of a million hashes that Alice has signed, just needs to find a message that is a pre-image of any of those million hashes. Each time Ivan does a hash of a trial message, he can compare the result against the entire list. Since a million is approximately 2^{20} , this reduces the number of messages he needs to try from 2^{128} to 2^{108} .

To prevent the multi-target attack, Alice should also include the treelet number in the quantity that she hashes. Note that to enable Bob to verify Alice’s signature, the treelet number must already be part of what is specified in Alice’s signature. So, Alice’s i th signature will include message M_i , random number R_i , treelet number used (i in this case), hash tree sibling hashes, and the treelet secrets revealed based on $\text{hash}(R_i|M_i|i)$.

Now let's modify the design so that Alice can also use 128-bit hashes inside her hash tree and still avoid the multi-target attack. The hashes will need to be customized with a unique value (similar to the purpose of salt in the Unix password hash—see §5.5 *Creating a Hash Using a Block Cipher*). And the list of hashes that would be useful for Ivan to match might not only be hashes that Alice has signed, but hashes that other authorized entities have signed. So, for Alice's hash tree, an additional input into each hash should be the name **Alice**, and the location in Alice's tree.

These optimizations reduce the size of a signature by a factor of four, since each constant released is only half as many bits and there are only half as many bits to sign. So the size of a signature can be about 4K octets for signing up to a thousand messages, 8K for signing up to a million, and 12K for signing up to a billion.

8.2.6 Hash Chains

A hash chain is computed by taking a value p and hashing it multiple times. Our notation is that $\text{hash}^7(p)$ means computing $\text{hash}(\text{hash}(\text{hash}(\text{hash}(\text{hash}(\text{hash}(\text{hash}(p)))))))$. Someone who knows p can calculate $\text{hash}^i(p)$ for any value i , but someone that sees $\text{hash}^i(p)$ cannot calculate anything in the chain before i . For example, if someone is told $\text{hash}^3(p)$, they'd be able to compute $\text{hash}^4(p)$, $\text{hash}^5(p)$, $\text{hash}^6(p)$, $\text{hash}^7(p)$, but not p , $\text{hash}(p)$, or $\text{hash}^2(p)$. Note that, as in the previous section, in order to avoid the multi-target attack, the hashes have to be “customized” by including some constant specific to the hash chain in each hash calculation. For example, in addition to the customization in the previous section (*e.g.*, Alice's name and the treelet number), the third hash of p should include 3. So $\text{hash}^3(p)$ would be $\text{hash}(\text{Alice}|\text{treelet number}|3|\text{hash}^2(p))$. However, we'll leave all that customization out of our notation to keep things simple.

We can use hash chains to make signatures smaller, at the cost of more computation. Previously, Alice signed one bit by releasing two numbers—either p_0 and h_1 for 0, or h_0 and p_1 for 1.

Suppose Alice wants to sign four bits at once. The 4-bit chunk will have one of the values {0000, 0001, ..., 1111}. Let Alice choose a single secret, say p , and hash p fifteen times. If the 4-bit chunk equals 7, she reveals $\text{hash}^7(p)$. If the 4-bit chunk equals 6, she reveals $\text{hash}^6(p)$, and so forth, until, if the chunk equals 0, she reveals p . This isn't secure yet, though, since if she signs 0, once she has revealed p , a forger could sign any other value for the chunk. If she signed 13, a forger would be able to sign 14 or 15.

To fix that security problem, Alice uses two hash chains, based on secrets that we'll call p_{up} and p_{down} . To sign the 4-bit value i , Alice divulges $\text{hash}^i(p_{\text{up}})$, and $\text{hash}^{15-i}(p_{\text{down}})$. Now a forger cannot sign extra values. If Alice signed i , the forger would need to find pre-images of $\text{hash}^i(p_{\text{up}})$ to sign a value smaller than i and would need to find pre-images of $\text{hash}^{15-i}(p_{\text{down}})$ to sign values larger than i .

Using this technique with 4-bit chunks, a treelet for signing 128 bits only requires 64 values: a pair of values— $\text{hash}^{15}(p_{\text{up}})$ and $\text{hash}^{15}(p_{\text{down}})$ —for each of the 32 4-bit chunks. In contrast,

signing each of the 128 bits individually requires 256 values. This optimization of using 4-bit chunks reduces the signature size from 8512 octets to 2368 octets in the case where up to a million signatures can be generated per public key.

A variant of this scheme, credited to Winternitz, makes the signature even smaller. Again, assuming 4-bit chunks, Winternitz still uses $\text{hash}^{15}(p_{\text{up}})$ for each chunk but requires only a single p_{down} chain, which signs the sum of all the chunks. Since there are 32 4-bit chunks in a 128-bit value, each with a value between 0 and 15, the sum of the chunks will be a number between 0 and $15 \times 32 = 480$. A treelet would only need 33 values: $\text{hash}^{15}(p_{\text{up}})$ for each chunk and a single value $\text{hash}^{479}(p_{\text{down}})$ for the sum.

To save the work of having to compute up to 480 hashes, the sum can be represented in binary with three 4-bit chunks, requiring three values of $\text{hash}^{15}(p_{\text{down}})$, one for each of those chunks. A treelet for signing a 128-bit value would then consist of 35 values: $\text{hash}^{15}(p_{\text{up}})$ for each of the 32 chunks in the 128-bit value and three values of $\text{hash}^{15}(p_{\text{down}})$ for the sum. This optimization reduces the signature size from 2368 octets to 1440 octets in the case where up to a million signatures can be generated per public key.

How many bits are in each chunk is one of the parameters of a hash signature algorithm that trades off signature size against computation. Using 8-bit chunks instead of 4-bit chunks results in signatures that are half as big (because there are half as many chunks) but increases the computation cost by a factor of eight (since 256 hashes are required to complete a chain rather than 16). Most standardized schemes tend to use chunk sizes between one and eight bits.

8.2.7 Standardized Schemes

RFCs 8391 (XMSS) and 8554 (LMS) each specify a stateful hash-based scheme. They differ in their details such as how they pad the data being hashed, and they each offer a wide range of parameter values trading off number of signatures with a single key, compute time, and signature size. RFC 8554 signature sizes range from 1616 octets (to sign up to 32K messages) to 3652 (to sign a trillion). RFC 8391 signature sizes range from 2500 octets (to sign up to 1K messages) to 27688 octets to sign up to 2^{60} messages.

8.2.7.1 Stateless Schemes

The schemes we've outlined so far require Alice to remember how many messages she's ever signed, so that she doesn't ever reuse the same treelet and wind up divulging both p_0^i and p_1^i for the same bit. For some applications, such as root certificates and code signing, this might be an acceptable requirement, since the application inherently requires strong version control, backups, and record keeping. However, for other applications, where signatures are created more dynamically

(e.g., TLS authentication, where the parameters for each new connection must be signed), using the stateful protocol is very likely to lead to security problems.

In the schemes we've looked at so far, the way we assured that we never used the same treelet to sign two different hashes was to keep track of how many things we have signed and never use a treelet more than once. But what if Alice is stateless?

One example of a stateless approach is as follows. Suppose Alice has a tree with 128 levels, so there are 2^{128} treelets, one for each possible hash value (assuming 128-bit hashes). To sign a hash with value X , she'd use the X th treelet. Alice would not need to keep track of how many hashes she's signed, because a treelet could only be used to sign a single value. Signatures would be fairly small (just 127 sibling hashes and the hashes or pre-images in the treelet), but Alice would need to generate the entire tree beforehand, which is obviously infeasible.

But if Alice used a multi-level tree, using the technique that we described in §8.2.3 *Signing Lots of Messages*, Alice need not generate the entire tree in advance and can instead deterministically generate trees as needed from a secret seed S . Alice's top-level tree might have 2^8 treelets (eight levels). To sign a hash that has the first eight bits equal to X , Alice uses the X th treelet in the top-level tree to sign the root of a tree T_X . The tree T_X is used for signing hashes starting with X .

Tree T_X will also have eight levels, and the Y th treelet in tree T_X will be used for signing the root of a tree to be used for signing message digests that start with $X|Y$. And so forth, through sixteen trees, to get up to the size of a 128-bit hash.

As we described before, in order to force Ivan to find a pre-image of a specific hash rather than find a collision, Alice will need to choose a random R when signing message m , and her signature will include R , m , and her signature on $\text{hash}(R|m)$.

Although Alice will never use the same treelet to sign two different hash values, there is still the problem of the multi-target attack. If Alice has signed, say a million $\langle R, m \rangle$ pairs, Ivan could try $\langle R', m' \rangle$ pairs, and see if $\text{hash}(R'|m')$ matches any of the $\text{hash}(R|m)$ values in any of Alice's signatures.

The defense is to increase the size of the hash according to how many signatures Alice is expected to sign. If she is likely to sign a million times ($\sim 2^{20}$), then the hash size should be increased by twenty bits, and the number of levels in the multi-level tree should be increased by twenty.

One optimization is to note that if Alice only uses the treelet $\text{hash}(R|m)$ for signing the value $\text{hash}(R|m)$, there is no need for the treelet to have more than a single leaf—treelet i need only contain a single hash value, say H_i , and Alice can sign the value i by revealing the pre-image of H_i . This scheme is attributed to John Kelsey, who called it Pyramid.

An alternative way to build a stateless hash-based signature scheme is that instead of having Alice use treelet Z to sign the hash value Z , Alice could choose a treelet, say treelet T , at random, compute $Z = \text{hash}(R|m|T)$, and sign the value Z using treelet T . Then the number of treelets does not need to depend on the length of the hash, but instead on the number of things that Alice will sign. For instance, if Alice will never sign more than a million messages (2^{20}), and the desired

probability that Alice accidentally choose the same treelet twice should be less than, say, 2^{-64} , then the number of treelets required should be at least 2^{104} . How do we get the exponent 104? By the birthday problem (§5.2), the probability of Alice accidentally choosing the same treelet if there are 2^{40} treelets with Alice signing 2^{20} messages is about $\frac{1}{2}$. To make the probability less than 1 in 2^{64} that she'll accidentally choose the same treelet, we need to add 64 to 40.

One of the proposals submitted to the NIST competition is a stateless hash signature scheme known as SPHINCS+. It defines some additional optimizations specific to the case of a stateless hash and allows up to 2^{64} signatures to be generated that are only 7856 octets long. Proposed schemes often have parameters that can be modified to trade off security, signature sizes, and computation, and often the authors of the schemes make modifications to the schemes to improve performance or security. So these numbers are just approximate, and we are providing them just to give the reader a feel for the performance.

8.3 LATTICE-BASED CRYPTOGRAPHY

What is a lattice? A **lattice** is a set of points in n -dimensional space. A point in an n -dimensional space is represented by an n -tuple of numbers $\langle x_1, x_2, \dots, x_n \rangle$. A point can also be thought of as a vector. For example, $\langle x_1, y_1, z_1 \rangle$ is the vector that gets you from the origin $\langle 0, 0, 0 \rangle$ to the point $\langle x_1, y_1, z_1 \rangle$. To add vectors, you add the i th coordinate of each to get the i th coordinate of the sum, *e.g.*, $\langle x_1, y_1, z_1 \rangle + \langle x_2, y_2, z_2 \rangle = \langle x_1+x_2, y_1+y_2, z_1+z_2 \rangle$.

A lattice is closed under addition and subtraction, which means that if two points $v_1 = \langle x_1, y_1, z_1 \rangle$ and $v_2 = \langle x_2, y_2, z_2 \rangle$ are points in a lattice, then so are $v_1 + v_2$ and $v_1 - v_2$. It follows then, that all the following linear combinations of v_1 and v_2 are also in the lattice:

- $v_1 - v_1$ which equals $\langle 0, 0, 0 \rangle$
- $-v_1$ which equals $\langle -x_1, -y_1, -z_1 \rangle$
- $2v_1 - 3v_2$ which equals $\langle 2x_1 - 3x_2, 2y_1 - 3y_2, 2z_1 - 3z_2 \rangle$

A **basis** of a lattice is a set of lattice vectors b_1, b_2, \dots, b_n such that any point in the lattice can be written in exactly one way as an integer linear combination of the n basis vectors. For example, if c_1, c_2, \dots, c_n are integers, then $c_1 b_1 + c_2 b_2 + \dots + c_n b_n$ is a point in the lattice. A lattice is completely specified by giving a basis for it.

A basis of an n -dimensional lattice will consist of n vectors, each with n components. A common way to specify a basis is as an $n \times n$ matrix, where the i th row in the matrix represents the i th basis vector.

There are many different bases that specify the same lattice. For example, the 2-dimensional lattice in Figure 8-2 can be specified with the two short basis vectors in the left diagram or the two

long vectors in the right diagram. Either basis generates the same lattice. A basis consisting of short vectors is known as a “good basis” while a basis consisting of long vectors is known as a “bad basis”.

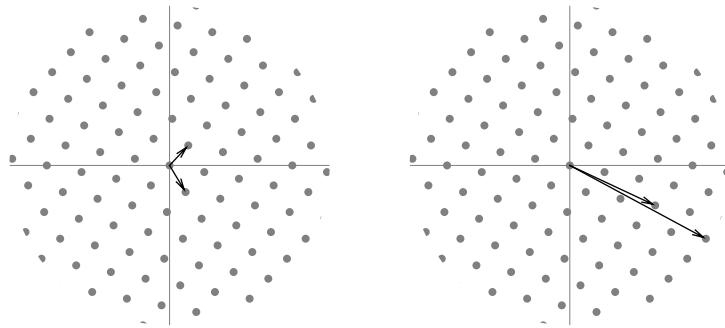


Figure 8-2. Lattice Bases

8.3.1 A Lattice Problem

A lattice problem that is considered difficult: Given a bad basis for a lattice in an n -dimensional space and some non-lattice point in that space, find a nearby lattice point. Given a good basis, however, it's easy (see §8.3.4.2). This leads to an intuition for a lattice-based scheme: have Bob's private key be a good basis for a lattice and his public key be a bad basis that generates the same lattice. If Alice knows Bob's public key (a bad basis for his lattice), she can establish a secret \mathbf{m} with Bob, as follows (see Figure 8-3):

- Alice uses Bob's public key (the bad basis) to compute some random point \mathbf{P} in Bob's lattice.
- She chooses a random small n -dimensional vector \mathbf{m} .
- She computes $\mathbf{X} = \mathbf{P} + \mathbf{m}$. Point \mathbf{X} will not be in Bob's lattice.
- Bob uses his private key (the good basis) to find the nearest lattice point to \mathbf{X} , which is \mathbf{P} .
- Bob computes $\mathbf{X} - \mathbf{P}$ to get \mathbf{m} .

$$\mathbf{P} \nearrow \mathbf{X}$$

Figure 8-3. Message Encoded as Offset from a Lattice Point

Lattice-based encryption schemes that work this way include the Goldreich, Goldwasser, Halevi cryptosystem [GOLD97] and the NTRU cryptosystem that we will discuss shortly.

8.3.2 Optimization: Matrices with Structure

One optimization cryptographers use to make schemes more practical is to use matrices with structure so that only a small amount of the matrix needs to be stored and transmitted, and the rest can be derived.

This optimization is such an important part of lattice schemes that we describe it before describing the schemes. To specify a general $n \times n$ matrix requires n^2 elements, each element requiring some number of bits to specify its value. Unfortunately, in order to make the lattice problems hard enough to be cryptographically strong, we need a lot of dimensions. For example, with $n=1000$, element size twelve bits, this would require twelve million bits. Using structured matrices can vastly improve the efficiency of lattice-based cryptosystems. While the most popular ways of using structured matrices are not known to significantly weaken security, some cryptosystems using structured matrices have been broken. Paranoid lattice cryptosystems that forgo the use of structured matrices entirely have been proposed. Nonetheless, given the lack of known attacks and the vast efficiency improvements, it seems likely that the most widely used lattice cryptosystems in the future will use structured lattices.

One example of a structured matrix is known as a **circulant matrix**, where only the top row needs to be specified, and each subsequent row is created by rotating the entries in the previous row one column to the right, with the element that falls off the end rotating into the leftmost position.

An example of a circulant matrix is:

$$\begin{bmatrix} 17 & 33 & 5 & 0 & -12 \\ -12 & 17 & 33 & 5 & 0 \\ 0 & -12 & 17 & 33 & 5 \\ 5 & 0 & -12 & 17 & 33 \\ 33 & 5 & 0 & -12 & 17 \end{bmatrix}$$

The way to think of this mathematically is that each row represents a polynomial of degree $n-1$, where n is a prime, with the elements being the coefficients of the n terms of the polynomial, constant term being on the left. If the row has elements $r_0, r_1, r_2, \dots, r_{n-1}$, then the polynomial represented by that row is $r_0 + r_1x + r_2x^2 + \dots + r_{n-1}x^{n-1}$. And to be more tangible, the top row of the above circulant matrix would represent the polynomial $17 + 33x + 5x^2 - 12x^4$.

Each row (other than the top one) is calculated by multiplying the polynomial represented by the previous row by x , and then reducing modulo $x^n - 1$. This results in just what we want. All the values will shift one position to the right, and the element shifted off the end will simply be shifted into the leftmost position. This technique is also known as *lattices from polynomial rings*, or *ideal lattices*.

Suppose you have two circulant matrices, \mathbf{M}_1 and \mathbf{M}_2 , where the polynomial representing the top row of \mathbf{M}_1 is \mathbf{p}_1 , and the polynomial representing the top row of \mathbf{M}_2 is \mathbf{p}_2 . It is easy to show that

$\mathbf{M}_1\mathbf{M}_2$ is a circulant matrix, and the top row of $\mathbf{M}_1\mathbf{M}_2$ is $\mathbf{p}_1\mathbf{p}_2 \bmod x^n - 1$. That means that not only do you save storage when using a circulant matrix (it only takes $1/n$ of the storage to specify the top row), but it saves computation to do addition or multiplication of circulant matrices, because all you need to do is multiply or add the polynomials representing the top rows of the matrices.

For example, all the following are circulant:

- The identity matrix
- The sum of two circulant matrices
- The product of two circulant matrices

While in general, matrix multiplication is not commutative, multiplication of circulant matrices is commutative (Homework Problem 8). While not all circulant matrices have inverses, for those that do, the inverses are also circulant matrices (Homework Problem 9).

Because circulant matrices can be specified by just specifying the top row, this greatly reduces key sizes. For instance, with 1000 dimensions and 12-bit elements, it only takes 12000 bits to specify a circulant matrix, rather than 12 000 000 bits to specify a general 1000×1000 matrix of 12-bit elements.

In a typical lattice-based cryptography scheme using circulant matrices, one or more circulant matrices with small elements are chosen by Bob to be his private key. Then he calculates a public key with large elements from the private key. The math for obfuscating the private key (to transform it into a public key) is chosen such that the public key will also consist of one or more circulant matrices, but with large elements.

8.3.3 NTRU-Encryption Family of Lattice Encryption Schemes

Now we will describe an actual scheme, NTRU (named for *N*th degree *T*RUuncated polynomial *r*ing) [HOFF98]. The original scheme has been improved for efficiency, and there are various proposals based on the original scheme. We will call the scheme in our description NTRU, even though *NTRU* describes a family of similar schemes, and if there is a standardized scheme named NTRU, it is likely to differ from the description here in some details. The variant we propose here is chosen to be simple to understand. It is similar to, but not identical to, the scheme called NTRU that is a NIST submission. It will not at first be obvious how NTRU relates to lattices, but we'll explain that after we explain our NTRU variant.

NTRU is described using degree $n-1$ polynomials. Arithmetic on the coefficients of the polynomials is done mod q , where n is a prime and q is a power of 2. A typical value of n is 509 and a typical value of q is 2048 (which would result in elements requiring eleven bits). After multiplication of polynomials, the result is reduced mod $x^n - 1$, so that the result will be a degree $n-1$ polynomial with coefficients interpreted as 11-bit signed integers ranging from -1024 through 1023. For

readability, we will avoid repeating everywhere that polynomial multiplication is mod $x^n - 1$ and arithmetic on the coefficients is done mod q .

8.3.3.1 Bob Computes a (Public, Private) Key Pair

There are some parameters that are well known, *i.e.*, specified as part of the algorithm. These parameters are the degree, n , of the polynomial modulus; the coefficients' modulus, q ; and a smaller modulus, p , which is 3 in most variants. To make the description easier to read, we'll cut down on the number of variables and use $p=3$ in our description. n is a prime number, and q is a power of 2. In the several variants proposed to NIST, which trade off performance versus security, $\langle n, q \rangle$ is $\langle 509, 2048 \rangle$, $\langle 677, 2048 \rangle$, $\langle 701, 8192 \rangle$, or $\langle 821, 4096 \rangle$. Although we will set p to 3, we will use n and q rather than choosing specific numbers for n and q in our description.

To create his key pair, Bob chooses two polynomials, f and g , each of degree $n-1$, with small integer coefficients. For polynomial g , the coefficients will be $-3, 0$, or 3 . Likewise for f , except the constant coefficient of f will be $-2, 1$, or 4 . As a result of this choice, g will be $0 \bmod 3$ (all the coefficients of the polynomial g will be $0 \bmod 3$), and f will be $1 \bmod 3$ (all the coefficients of f will be $0 \bmod 3$, except the constant term, which will be $1 \bmod 3$). The fact that $f=1 \bmod 3$ and $g=0 \bmod 3$ will be relevant later.

The polynomial f needs to have the extra property that it has an inverse, a polynomial f^{-1} such that $f \times f^{-1} = 1 \pmod{x^n - 1 \bmod q}$, as always in NTRU, though for readability we'll elide these moduli henceforth). If the coefficients Bob chose for f result in a polynomial that does not have an inverse, he chooses a different f . It is not difficult for Bob to calculate f^{-1} (see Homework Problem 10).

To turn f and g into a public key, Bob multiplies g by f^{-1} to get a new polynomial, which we'll call H . So, $H=f^{-1}g$.

Bob's public key is H , which will be an $n-1$ degree polynomial with large coefficients (in the range of $-q/2$ through $q/2-1$). Bob's private key is f .

8.3.3.2 Alice encrypts m with Bob's public key

Alice knows H . She chooses two $(n-1)$ -degree polynomials, r and m , both with small coefficients. The polynomial m is the encoding of the message she is encrypting for Bob, and its coefficients are all chosen to be $1, 0$, or -1 .

Alice sends $rH+m$ to Bob. Only Bob, with knowledge of his private key f , will be able to obtain m .

8.3.3.3 How Bob Decrypts to Find \mathbf{m}

Bob multiplies the polynomial he receives ($\mathbf{r}\mathbf{H} + \mathbf{m}$) by his private key f , getting $f \times (\mathbf{r}\mathbf{H} + \mathbf{m})$. Since $\mathbf{H} = f^{-1}\mathbf{g}$, the result is $f \times (f\mathbf{g}^{-1}\mathbf{g} + \mathbf{m}) = f \times f\mathbf{g}^{-1}\mathbf{g} + f\mathbf{m} = f \times f^{-1}\mathbf{rg} + f\mathbf{m} = \mathbf{rg} + f\mathbf{m}$. He then reduces the coefficients in $\mathbf{rg} + f\mathbf{m}$ mod 3. Since $\mathbf{g} = 0 \text{ mod } 3$ (all the terms in polynomial \mathbf{g} are 0 mod 3), and since $f = 1 \text{ mod } 3$, the result of reducing the coefficients in $\mathbf{rg} + f\mathbf{m}$ mod 3 is \mathbf{m} .

It might be surprising that reduction mod 3 works. It wouldn't work if the coefficients had been first reduced mod q , because if z is divisible by 3, $z-q$ won't be. So why can Bob reduce $\mathbf{rg} + f\mathbf{m}$ mod 3?

The size of the coefficients in $\mathbf{r}, \mathbf{g}, f$, and \mathbf{m} are chosen to be small enough that the coefficients in $\mathbf{rg} + f\mathbf{m}$ would not need to be reduced mod q , if $\mathbf{rg} + f\mathbf{m}$ were calculated directly (without having intermediate multiplication by f^{-1}). Although the journey by which Alice and Bob each did part of the computation to get $\mathbf{rg} + f\mathbf{m}$ did involve mod q arithmetic, the result ($\mathbf{rg} + f\mathbf{m}$) was the same as if Bob had directly calculated $\mathbf{rg} + f\mathbf{m}$. Since direct calculation of $\mathbf{rg} + f\mathbf{m}$ will not involve mod q , Bob can take what he calculated and reduce mod 3.

8.3.3.4 How Does this Relate to Lattices?

When we introduced lattices, we described the basis of an n -dimensional lattice as n n -element vectors, or alternatively, the n rows of an $n \times n$ matrix. The definition of lattices might seem to say that the elements (the coordinates of the points in the lattice) are real numbers, which would be awkward for implementations, because we'd like to express an element in a small fixed number of bits. If the basis vectors for a lattice all have integer elements, the lattice points would all have integer elements; but this would still be awkward because the integers would be of unbounded size.

Modular arithmetic would be ideal, but if we simply defined an n -dimensional lattice with mod q arithmetic, any n basis vectors that are linearly independent mod q would result in a very uninteresting lattice—every point with integer coordinates would be a lattice point.

Bob's lattice is actually $2n$ -dimensional, with infinitely many lattice points, including lattice points with elements that are arbitrarily large (larger than q). However, NTRU uses a trick so that elements in the basis vectors, and in the computed lattice points, are small enough that they can be represented as mod q integers.

A basis for Bob's $2n$ -dimensional lattice consists of $2n$ basis vectors, each $2n$ -dimensional. The basis can be represented by the $2n \times 2n$ matrix constructed by gluing four $n \times n$ matrices together:

- An identity matrix on the top left
- \mathbf{H} on the top right (recall that \mathbf{H} is Bob's public key)
- $\mathbf{0}$ on the bottom left
- q times an identity matrix on the bottom right

$$\begin{bmatrix} \mathbf{I} & \mathbf{H} \\ \mathbf{0} & q\mathbf{I} \end{bmatrix}$$

Recall that given a basis for Bob's lattice, replacing any basis vector with that basis vector plus any linear combination of other basis vectors will result in another basis for the same lattice. Likewise, adding or subtracting a basis vector to any lattice point yields another lattice point in Bob's lattice.

In NTRU, the elements of \mathbf{H} (the elements in the top right of this matrix) can be kept within the desired range by adding or subtracting the relevant rows of the bottom half. For example, to reduce the rightmost element in the top row mod q , subtract the bottom row as many times as necessary. Therefore, with the trick of using the bottom half of the basis vectors to reduce elements with size outside the range, the NTRU implementation computes a public key with elements in the proper range and finds lattice points with elements in the desired range ($-q/2$ through $q/2-1$) by adding or subtracting the relevant bottom rows.

In our description of NTRU, we said that Alice chooses two n -dimensional vectors, \mathbf{r} and \mathbf{m} . What is actually happening in the $2n$ -dimensional lattice? Since Bob's lattice is really $2n$ -dimensional, Alice would be choosing two $2n$ -dimensional vectors, which we'll call \mathbf{r}' and \mathbf{m}' :

- \mathbf{r}' is the n elements of \mathbf{r} followed by n zeroes, which we'll write as $\mathbf{r}' = \mathbf{r}|0^n$.
- \mathbf{m}' is the n elements of $-\mathbf{r}$ followed by the n elements of \mathbf{m} , which we'll write as $\mathbf{m}' = -\mathbf{r}|\mathbf{m}$.

Note that \mathbf{m}' consists of small elements since both \mathbf{r} and \mathbf{m} consist of small elements.

The vector \mathbf{r}' is used to compute a lattice point in Bob's lattice. Multiplying \mathbf{r}' by Bob's $2n$ -dimensional public key matrix results in a $2n$ -dimensional vector consisting of \mathbf{r} concatenated with $\mathbf{r}\mathbf{H}$, which we'll write as $\mathbf{r}|\mathbf{r}\mathbf{H}$, which is a lattice point in Bob's $2n$ -dimensional lattice. Alice can use the bottom half of Bob's public matrix to reduce any of the elements in the last half of the vector mod q , which will compute another lattice point in Bob's lattice, where the elements will be in the desired range.

The vector \mathbf{m}' is the offset from a point in Bob's lattice. Even if Alice has reduced the elements of $\mathbf{r}\mathbf{H}$ mod q , adding \mathbf{m}' might require her to use the bottom half of Bob's public matrix to reduce some more but will still result in another lattice point, where \mathbf{m}' will be the offset from that lattice point.

The result of Alice adding $\mathbf{m}' = -\mathbf{r}|\mathbf{m}$ to the lattice point $\mathbf{r}|\mathbf{r}\mathbf{H}$ will be the $2n$ -dimensional vector that is n zeroes followed by $\mathbf{r}\mathbf{H} + \mathbf{m}$ (mod q), or equivalently, $0^n|\mathbf{r}\mathbf{H} + \mathbf{m}$ (mod q). This is a point in $2n$ -dimensional space that is close to a point in Bob's lattice (it's close because \mathbf{m}' , as we said before, consists of all small elements). There's no reason for Alice to send n zeroes, so she just sends $\mathbf{r}\mathbf{H} + \mathbf{m}$.

What does the good basis for Bob's $2n$ -dimensional lattice look like? His private key consists of \mathbf{f} and \mathbf{g} . The top half (n rows) of the matrix representing his good basis consists of two $n \times n$ circulant matrices—a circulant matrix on the left with \mathbf{f} as the top row and a circulant matrix with \mathbf{g} as the top row on the right. Because both \mathbf{f} and \mathbf{g} were chosen to have small elements, this will be n vectors in a good basis. To see that the n rows of this $n \times 2n$ matrix indeed represent the same lattice points as Bob's public key, note that multiplying the top half of Bob's public matrix $(\mathbf{I}|\mathbf{H})$ by the

circulant matrix with f as the top row, and using the bottom half of his (bad) basis as necessary to reduce elements by mod q , results in the top n rows of the good basis.

This is only half of Bob's private basis (because there are only n rows). Bob can get another n vectors that are nearly as short to fill out his basis. These extra rows are not required for decryption. The extra rows are used in signatures based on the NTRU lattice, but the process used to generate them is somewhat tedious, so we will not discuss it.

8.3.4 Lattice-Based Signatures

Just as with encryption, the first lattice-based signature scheme we describe will consist of Bob's private key being a good basis for a lattice and his public key being a bad basis for the same lattice. First, we'll present a simple, intuitive signature scheme that was shown to be insecure. Then we'll show how it was fixed.

8.3.4.1 Basic Idea

We assume there is a special hash function whose output contains the correct number of bits to be interpreted as a random point \mathbf{P} in an n -dimensional space, and we assume that Bob's lattice is n -dimensional.

To sign a message M , Bob uses that hash function to find the corresponding $\mathbf{P} = \text{hash}(M)$ in n -dimensional space. Bob signs M by showing a lattice point \mathbf{L} in his lattice near to \mathbf{P} . So Bob's signature consists of $\langle M, \mathbf{L} \rangle$. This proves Bob knows the private key because only someone that knows the good basis can calculate a nearby lattice point.

To verify Bob's signature on M , Alice uses the hash function to find \mathbf{P} from M , verifies that $\mathbf{P} - \mathbf{L}$ is small, and using Bob's public key (the bad basis), can verify that \mathbf{L} is in Bob's lattice.

8.3.4.2 Insecure Scheme

How does Bob use his short basis to find a nearby lattice point? One simple method, proposed in the Goldreich, Goldwasser, Halevi signature scheme [GOLD97], works as follows:

Recall, given a basis $\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n$ for a lattice, that $c_1\mathbf{b}_1 + c_2\mathbf{b}_2 + \dots + c_n\mathbf{b}_n$ is a point in the lattice as long as c_1, \dots, c_n are integers. Suppose $\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n$ is Bob's good basis. Bob can find a lattice point near \mathbf{P} by

1. Solving for real numbers x_1, \dots, x_n such that $x_1\mathbf{b}_1 + \dots + x_n\mathbf{b}_n = \mathbf{P}$. This step involves solving linear equations of the real numbers to limited precision, which is easy.
2. Rounding each of x_1, \dots, x_n to the nearest integer to get integers c_1, \dots, c_n .
3. Multiplying the basis vectors by c_1, \dots, c_n to get the lattice point $\mathbf{L} = c_1\mathbf{b}_1 + c_2\mathbf{b}_2 + \dots + c_n\mathbf{b}_n$.

Rounding x_i to get c_i moves \mathbf{L} away from \mathbf{P} by at most half the length of a basis vector. Since Bob's good basis consists of short vectors, the total distance from \mathbf{L} to \mathbf{P} isn't very big. If Trudy (who doesn't know a good basis for Bob's lattice) had tried the same process using Bob's bad public basis, she would likely have gotten an \mathbf{L} that was much further away. So Alice just needs to check that \mathbf{P} is as close to \mathbf{L} as would be expected if Bob's good basis was used.

This proposed lattice-based signature scheme was shown to be insecure by Gentry and Szydlo in 2002 [GENT02], and the attack was improved by Nguyen and Regev in 2006 [NGUY06] so that breaking a private key required the attacker seeing only 400 of Bob's signatures.

Why is this insecure? Unfortunately, the public key is not the only information an attacker might have. Typically, the attacker also has access to some appreciable number of previously signed messages, and so $\langle \mathbf{P}, \mathbf{L} \rangle$ pairs. If the vectors consisting of $\mathbf{P} - \mathbf{L}$ for each of Bob's signatures are graphed, they are all confined to a parallelepiped whose edges are parallel to the good basis vectors in Bob's private key. The attacker can do statistics on $\mathbf{P} - \mathbf{L}$ for a moderate number of signatures to learn the shape of this parallelepiped (Figure 8-4)..

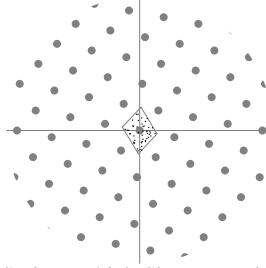


Figure 8-4. How Seeing Multiple Signatures Divulges the Private Key

8.3.4.3 Fixing the Scheme

In 2008, Gentry, Peikert, and Vaikuntanathan [GENT08] published a method that avoids the flaws in the first scheme. The basic idea is that Bob should not always give the closest lattice point to $\text{hash}(M)$, but instead he should give a somewhat nearby lattice point. Bob has to be careful to always choose the same \mathbf{L} for a given \mathbf{P} ; otherwise, if Bob signed \mathbf{P} with \mathbf{L}_1 , and later signed \mathbf{P} with \mathbf{L}_2 , the two lattice points \mathbf{L}_1 and \mathbf{L}_2 would be sufficiently close to each other that an attacker that saw both signatures could compute $\mathbf{L}_1 - \mathbf{L}_2$, which would be a small basis vector for Bob's lattice.

This provably secure hash-then-sign signature is the basis for the NIST candidate Falcon. There are other efficient provably secure lattice-based signature constructions, such as the one proposed in 2012 by Lyubashevsky [LYUB12] that is the basis for the NIST candidate Dilithium. By *provably secure*, as usual, we mean the scheme is secure assuming certain assumed-hard problems are actually hard. In both these cases, however, the proofs are strong enough to rule out the possibility that seeing signed messages in addition to the public key could significantly help the attacker.

8.3.5 Learning with Errors (LWE)

Another encryption scheme in the lattice family is known as **learning with errors (LWE)**. As with NTRU, there are many variants.

LWE schemes always use modular arithmetic. As with NTRU, they can be translated to a lattice problem, and the result looks very similar to the $2n$ -dimensional lattice in §8.3.3.4.

LWE is intuitively similar to Diffie-Hellman (§6.4), in that in both algorithms Alice and Bob each choose a secret, generate a public message derived from that secret, transmit their public message, and use the received public message and their own secret to generate a shared secret. Recall that in traditional Diffie-Hellman protocol with modular exponentiation, the non-secret parameters are a generator g and a prime p . Alice chooses a secret A and Bob chooses a secret B . The public message Alice sends is $g^A \bmod p$. Bob sends $g^B \bmod p$, and they agree on the secret $g^{AB} \bmod p$.

In contrast, in LWE, the non-secret parameters include a modulus size q and an $n \times n$ matrix A with elements between 0 and $q-1$. The value of n is a parameter of the scheme and is typically in the range 500–1000. In our example, we use $q=2^{15}$ and $n=1000$.

It is safer to agree upon a new randomly generated matrix for each exchange, and designs ensure that it is inexpensive to do so. The matrix A will not be secret, but Alice and Bob need to agree upon the same matrix. Rather than enumerate all n^2 elements of A , the matrix can be specified by having Alice transmit a 256-bit seed to Bob and having a deterministic publicly agreed-upon algorithm for generating A using the seed. So Alice, Bob, and any eavesdropper will be able to compute A . When we say below that Alice transmits A to Bob, we mean that she chooses a seed that generates A and sends the seed to Bob.

In LWE, Alice and Bob each generate a secret that is a pair of n -length vectors with small elements. The specific limit on “small” from which the elements are chosen (*e.g.*, $-2, -1, 0, 1$, or 2) is specified by the scheme. Schemes also specify a distribution of elements. For instance, if a scheme specifies a uniform distribution, there should be an equal probability of choosing each value for each element. A binomial distribution would give higher probability to 0, lower probability to ± 1 , even lower to ± 2 .

Using terminology in most of the papers, we’ll call Alice’s secret vectors r and e_A . We’ll call Bob’s secret vectors s and e_B . The vectors e_A and e_B are **error vectors**. Their elements are mod q integers. Alice’s vector r will multiply as a row vector (*i.e.*, a $1 \times n$ matrix), while Bob’s vector s will multiply as a column vector (*i.e.*, an $n \times 1$ matrix).

The protocol steps are:

1. Alice chooses n -element vectors r and e_A , both with small coefficients, and a matrix A .
2. Alice computes, and transmits to Bob, A and the n -dimensional vector $rA + e_A$.
3. Bob chooses n -element vectors s and e_B , also both with small coefficients, and transmits to Alice the n -dimensional vector $As + e_B$.

-
4. Alice multiplies \mathbf{r} by what she received from Bob (the n -dimensional vector $\mathbf{As} + \mathbf{e}_B$) to get the scalar $\mathbf{rAs} + \mathbf{re}_B$.
 5. Bob multiplies the n -dimensional vector he received from Alice ($\mathbf{rA} + \mathbf{e}_A$) by s to get the scalar $\mathbf{rAs} + \mathbf{e}_As$.
 6. Since \mathbf{r} , s , and the error vectors are all small, both Alice and Bob will calculate scalars that are close to $\mathbf{rAs} \pmod{q}$.

Let's give the value rAs the name Z (which is what Alice and Bob would have both computed if it weren't for adding in the error vectors). If q is 2^{15} , Z will be 15 bits long. Alice and Bob each have computed a value that is approximately equal to Z .

How far apart might "approximately equal" be? Call Alice's value Z_A and Bob's value Z_B . $Z_A - Z$ (Alice's difference from Z) is \mathbf{re}_B . In the worst case (all the coefficients in \mathbf{r} and \mathbf{e} are, say, 2, and with $n=1000$), the product can have magnitude at most 4000. Likewise, the maximum magnitude for \mathbf{e}_As (Bob's difference from Z) is also 4000, so the maximum magnitude for $Z_B - Z_A$ is 8000. But since elements are supposed to be distributed among $\{-2, -1, 0, 1, 2\}$, the actual difference will usually be much smaller than that.

To send a single bit to Alice, Bob adds a small number (e.g., $-2, -1, 0, 1$, or 2) to Z_B to transmit 0 , and he adds a large number to Z_B ($q/2 +$ one of $-2, -1, 0, 1$, or 2) to transmit 1 . When Alice receives this number from Bob, she can tell whether the result is closer (\pmod{q}) to Z_A (in which case Bob is sending 0) or closer to Z_A plus half the maximum value of an element (in which case Bob is sending 1).

8.3.5.1 LWE Optimizations

8.3.5.1.1 Reusing $\langle r, e_A \rangle$ and $\langle s, e_B \rangle$ Values

Using the strategy in the previous section, Alice and Bob each need to send an n -element vector in order to agree on a single secret bit. If $n=1000$, with elements that are 15-bits long, each vector is 15 000 bits.

In typical uses, Alice and Bob really need to agree on a 256-bit secret (rather than a single bit). Rather than doing the strategy in the previous section 256 times, once for each bit, where Alice and Bob would each have to transmit 256 different n -element vectors, the optimization in this section allows Alice and Bob to agree upon a 256-bit secret and each only transmit sixteen n -element vectors!

Alice chooses sixteen pairs $\langle r_0, e_{A0} \rangle, \langle r_1, e_{A1} \rangle, \dots, \langle r_{15}, e_{A15} \rangle$. Bob chooses sixteen pairs $\langle s_0, e_{B0} \rangle, \langle s_1, e_{B1} \rangle, \dots, \langle s_{15}, e_{B15} \rangle$. Each $\langle i, j \rangle$ combination yields a different approximate Z , resulting in 256 different values for Z , i.e., $Z_{ij} = \mathbf{r}_i \mathbf{As}_j$ for $0 \leq i \leq 15$ and $0 \leq j \leq 15$.

At this point there are 256 Z values, which means 256 Z_A values known to Alice and 256 Z_B values known to Bob. For each of the 256 Z_B values, Bob adds either approximately 0 (for a 0) or approximately $q/2$ (for a 1). Alice and Bob each transmit sixteen n -element vectors, and Bob additionally transmits 256 scalars. With $n=1000$, and elements being fifteen bits each, Alice will transmit $16 \times 15 \times 1000$ bits for her sixteen vectors (240000 bits), and Bob will also transmit 240000 bits, plus 256 15-bit scalars (about 4000 bits).

8.3.5.1.2. Sending Multiple Bits for each Z

Assuming the scheme is carefully specified so that the difference between Z_A and Z_B is sufficiently small relative to the modulus, another optimization is possible. Instead of Bob partitioning each Z_B into two ranges—one range for 0 and the other for 1—the set of numbers can be partitioned into more ranges, say sixteen, in order to send four bits at a time (0000, 0001, ..., 1111). So, instead of Bob adding approximately $q/2$ to send a 1, he adds approximately $iq/16$ to send the 4-bit value i . A specific scheme will carefully choose the parameters so that it is possible to partition the values into the specified number of pieces. (See Homework Problem 14.)

8.3.5.1.3. Ring LWE

As with NTRU, LWE can be further optimized by using structured matrices. Note that a circulant matrix as described in §8.3.2 is not the only type of structured matrix. A variant that is popular for ring LWE is one in which the polynomial representing row k is derived from the polynomial representing row $k-1$ by multiplying the polynomial in row $k-1$ by x and reducing mod x^n+1 (instead of x^n-1 as before). The result of reducing mod x^n+1 is that the element that gets shifted off the right is brought into the leftmost position, but with the opposite sign. In this form of structured matrix, n is usually a power of 2 (instead of n being a prime as before). As with circulant matrices, arithmetic can be done using only the top row of the structured matrix, treated as a polynomial. An example of this kind of matrix is

$$\begin{bmatrix} 17 & 33 & 5 & -12 \\ 12 & 17 & 33 & 5 \\ -5 & 12 & 17 & 33 \\ -33 & -5 & 12 & 17 \end{bmatrix}$$

Alice chooses a single pair of degree $n-1$ polynomials with small coefficients, \mathbf{r} and \mathbf{e}_A . Bob chooses a single pair of degree $n-1$ polynomials with small coefficients, \mathbf{s} and \mathbf{e}_B . What was the matrix \mathbf{A} for unstructured LWE is a degree $n-1$ polynomial \mathbf{A} for ring LWE. As before, to allow \mathbf{A} to be specified with fewer bits, the coefficients of \mathbf{A} can be generated from a 256-bit seed.

Alice computes rA , resulting in a degree $n-1$ polynomial, and adds e_A , resulting in a degree $n-1$ polynomial). Likewise, Bob computes the degree $n-1$ polynomial $As+e_B$. Note that since polynomial multiplication is commutative, it doesn't matter whether Alice computes $rA+e_A$ or $Ar+e_A$. Likewise, Bob could compute either $sA+e_B$ or $As+e_B$. Polynomials are reduced to a degree $n-1$ polynomial (reduced mod x^n+1 or x^n-1 , depending on the type of structured matrix used), and arithmetic on the coefficients is done mod q .

Alice takes the $n-1$ degree polynomial she received from Bob and multiplies that by r . Bob takes the polynomial he received from Alice and multiplies that by s (again reducing the polynomial to a degree $n-1$ polynomial and reducing the coefficients mod q .)

Now Alice and Bob each have computed $n-1$ degree polynomials with coefficients between 0 and $q-1$, that will both be approximately rAs .

Each of the n coefficients in that polynomial serves the purpose of the Z s in the previous section—a value that Alice and Bob will agree on approximately, but an eavesdropper will not be able to compute. If it weren't for the error polynomials (e_A and e_B), Alice and Bob would have computed the same polynomial (rAs), but because of the error polynomials, the n coefficients of rAs will be close to what each of Alice and Bob will have computed. Therefore, Bob can use these coefficients to send up to n secret bits to Alice, again, by adding approximately 0 to a coefficient to send a 0 and adding approximately $q/2$ to send a 1.

8.3.5.1.4. Structured LWE

Ring LWE drastically reduces the keysize and ciphertext size as compared to unstructured LWE, but there is a broader class of similar systems (including but not limited to ring LWE) called structured LWE. An example of a structured-LWE cryptosystem that is not a ring-LWE cryptosystem is the NIST submission CRYSTALS-Kyber*, which uses a structured variant of LWE called module LWE. In Kyber, the matrix A is structured as a $k \times k$ array of 256×256 submatrices, where k is 2, 3, or 4, depending on the security level. For instance, for NIST security level 1, k is 2 and so A is 512×512 . Each of the submatrices is structured so that arithmetic on the submatrices can be done with polynomials, as with ring LWE, and some cryptographers think Kyber is a little less risky because the matrix A is a little less structured than with ring LWE. For example, when k is 4, there are 4 times as many independent coefficients in A as there would be for a same-sized A using ring LWE.

*CRYSTALS is an acronym for CRYptographic SuiTe for ALgebraic LatticeS.

8.3.5.2 LWE-based NIST Submissions

LWE based on structured matrices is generally believed to be secure and is much more efficient than unstructured LWE. A NIST submission based on structured LWE is CRYSTALS-Kyber. A submission based on unstructured matrices is called Frodo.*

In Frodo-640 (the Frodo variant for NIST level 1 security, n is 640, elements are mod 2^{15} (so fifteen bits long), Alice and Bob each transmit eight vectors, and two bits are sent each time. So, to transmit the vectors, Alice and Bob each need to transmit $8 \times 15 \times 640$ bits, or approximately 10000 octets.

In Frodo-1344 [the variant to match AES-256 security (NIST level 5)], $n=1344$, elements are mod 2^{16} (so 16 bits long), Alice and Bob each transmit eight vectors, and four bits are sent at a time. So to transmit the vectors, Alice and Bob each need to transmit $8 \times 16 \times 1344$ bits, or approximately 22000 octets.

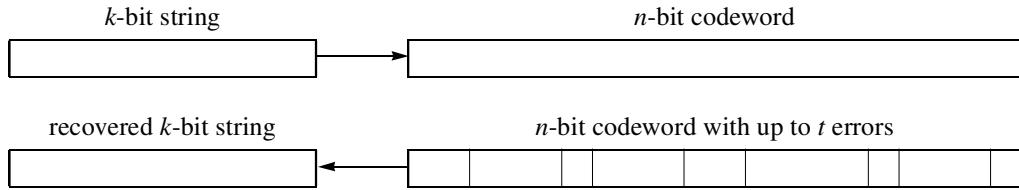
In Kyber-512 (the CRYSTALS-Kyber variant for NIST level 1 security), Alice and Bob each transmit a 2-dimensional vector of degree-256 polynomials whose coefficients are modulo 3329 (so 12 bits). Alice and Bob each need to transmit a little more than 12×512 bits, or approximately 800 octets. In Kyber-1024 (the CRYSTALS-Kyber variant for NIST level 5 security), Alice and Bob each transmit a 4-dimensional vector of degree-256 polynomials whose coefficients are modulo 3329. Alice and Bob each need to transmit a little more than 12×1024 bits, or approximately 1600 octets.

8.4 CODE-BASED SCHEMES

Code-based cryptographic schemes use error-correcting codes in their construction. Error-correcting codes were originally invented to solve the problem of communication media or storage systems that might corrupt a small portion of the bits. (See Figure 8-5.) An error-correcting code expands a k -bit string into an n -bit string known as a **codeword**. There is also an inverse function that can take an n -bit codeword with up to t errors (flipped bits), find which bits were flipped, and recover the original k -bit string. The greater the number of redundant bits added to a string, the more errors the error-correcting code can correct. If too many errors occur, the error-correcting code may report that the error is unrecoverable, or it may recover incorrect information.

When used as the basis of a cryptographic scheme, the error-correcting code is not used to correct accidentally introduced errors. Instead, errors are deliberately introduced, and these deliberately introduced errors are the secret being transmitted.

*The name is a *Lord of the Rings* reference, where Frodo lets go of the ring.

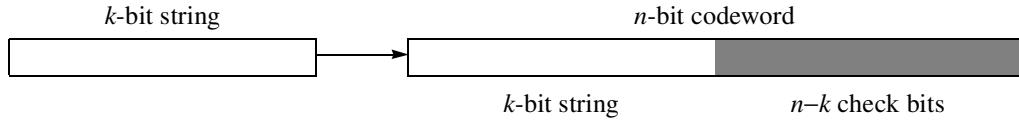
**Figure 8-5.** Error-Correcting Code

Code-based schemes are very similar to lattice-based schemes. The scheme we describe (based on the MDPC-McEliece scheme of Misoczki *et al* [MISO13], which subsequently was submitted to NIST as BIKE) is very similar to the lattice-based scheme NTRU. In NTRU, the secret is the offset from a lattice point. In the scheme we describe, the secret is the offset from a codeword.

8.4.1 Non-cryptographic Error-correcting Codes

First, to build intuition, we'll describe a non-cryptographic error-correcting code.

The codeword might be in a form that consists of the original k -bit string with $n-k$ check bits appended. That is known as **systematic form** (see Figure 8-6). If the algorithm chosen for converting a k -bit string into an n -bit codeword does not generate systematic form codewords, a codeword will look (to a human) like an n -bit quantity not obviously related to the original string.

**Figure 8-6.** Systematic Form

The simple non-cryptographic error-correcting scheme we describe in this section is inspired by moderate density parity-check (MDPC) schemes. Later, we'll show how to turn our non-cryptographic scheme back into the public key encryption scheme that inspired it.

The algorithms for creating codewords and for correcting errors (in this non-cryptographic scheme) are both public. The codewords generated are in systematic form. The steps in the algorithm, each of which will be described below, are:

- Invention step: Define an error-correcting code.
- Codeword creation step: Input a k -bit string M and output an n -bit codeword Y in systematic form. The codeword Y is $M|C$, where M is the k -bit string, and C is $n-k$ check bits.
- Misfortune step: Flip up to t bits of the codeword Y , producing what we'll refer to as the “mangled codeword” Y' . Think of Y' as equal to $Y \oplus E$, where E is an n -bit error vector

whose 1s flip the bits in \mathbf{Y} . Since the error vector might have 1s in both the \mathbf{M} and the \mathbf{C} portions of \mathbf{Y} , the mangled codeword \mathbf{Y}' will be $\mathbf{M}'|\mathbf{C}'$.

- Diagnosis step: Calculate the error vector \mathbf{E} .

Note that in the original use for which error-correcting codes were designed, the goal is to recover \mathbf{M} . However, for the cryptographic scheme we will be describing, the value of \mathbf{M} is unimportant. We can think of \mathbf{M} as a random seed for calculating a codeword in Alice's error-correcting code. The actual message Bob will want to send to Alice is the error vector that Bob will add to the codeword.

8.4.1.1 Invention Step

To create an error-correcting code, Alice creates a matrix \mathbf{G} , known as the **generator matrix**. \mathbf{G} will be a binary matrix (entries are 0 or 1), sparse (only about 1% of the entries being 1), and of size $k \times n$. We treat a binary string as a binary row vector and vice versa. Multiplying a k -bit string by \mathbf{G} will produce an n -bit codeword. Note that \mathbf{G} will have thousands of rows and columns.

Arithmetic is mod 2. This means that \oplus , $+$, and $-$ are all equivalent. So, for instance, if you added a matrix \mathbf{X} to itself, *i.e.*, $\mathbf{X}+\mathbf{X}$, you'd get $\mathbf{0}$. Likewise $\mathbf{X}-\mathbf{X}=\mathbf{0}$.

We want \mathbf{G} to produce systematic form. In order to produce systematic form codewords, the left side of \mathbf{G} will be the identity matrix \mathbf{I} (where the diagonal entries are 1 and the rest 0).

The rest of \mathbf{G} will be a randomly generated sparse $(n-k) \times k$ matrix that we'll call \mathbf{Q} (see Figure 8-7). We apologize for the figure. \mathbf{Q} should be sparse (only 1% of the entries should be 1), but that would require \mathbf{Q} in the figure to have hundreds of columns, and that would be unreadable, so please imagine a much larger \mathbf{Q} , one with hundreds of columns and with only about 1% of the entries being 1.

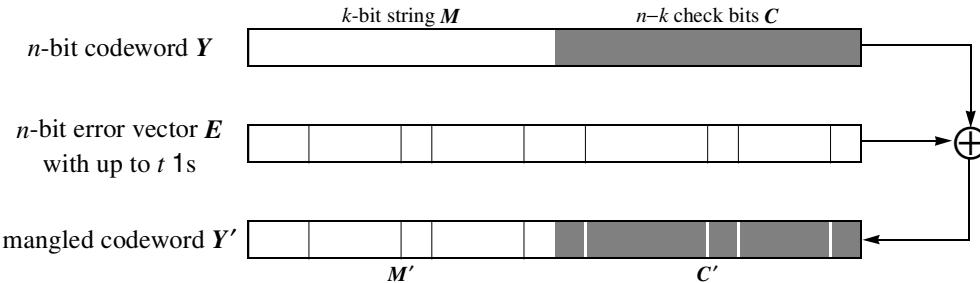
8.4.1.2 Codeword Creation Step

This is easy. We take the k -bit string \mathbf{M} and multiply it by matrix \mathbf{G} to get a codeword \mathbf{Y} . Because the left part of \mathbf{G} is the identity matrix, \mathbf{MG} will be in systematic form, so \mathbf{Y} will be $\mathbf{M}|\mathbf{C}$, where \mathbf{C} is the $n-k$ check bits.

8.4.1.3 Misfortune Step

Select an n -bit error vector \mathbf{E} with up to t 1s. Compute mangled codeword $\mathbf{Y}'=\mathbf{Y}\oplus\mathbf{E}$. Since \mathbf{E} can mangle bits in both the first k bits of \mathbf{Y} (which is the string \mathbf{M}) and the $n-k$ check bits (which is the string \mathbf{C}), $\mathbf{Y}'=\mathbf{M}'|\mathbf{C}'$. (See Figure 8-8.)

$$\begin{array}{c}
 k \times n \text{ matrix } \mathbf{G} \\
 \left[\begin{array}{ccccccccc|ccccccccc}
 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array} \right] \\
 k \times k \text{ Identity} \qquad \qquad \qquad k \times (n-k) \text{ matrix } \mathbf{Q}
 \end{array}$$

Figure 8-7. Generator Matrix for Systematic Form**Figure 8-8.** Generating Mangled Codeword

8.4.1.4 Diagnosis Step

Alice receives codeword \mathbf{Y}' with possibly some errors (if there were no errors, $\mathbf{Y}'=\mathbf{Y}$). She wants to calculate \mathbf{E} .

Alice, knowing she's receiving something in systematic form, starts by assuming that the first k bits of the codeword (\mathbf{M}') is the string \mathbf{M} . She uses \mathbf{G} to compute the codeword for \mathbf{M}' and gets check bits \mathbf{C}'' . Then she computes $\mathbf{C}' \oplus \mathbf{C}''$, which is known as a **syndrome**, and is of length $n-k$. If there were no errors in the codeword, then \mathbf{C} , \mathbf{C}' , and \mathbf{C}'' would all be equal, so the syndrome would be $\mathbf{0}$.

If the syndrome is not $\mathbf{0}$, then Alice will use the sparse matrix \mathbf{Q} to diagnose where the errors are. This is not a simple calculation, and, indeed, would be computationally infeasible if \mathbf{Q} were not sparse.

Consider what happens to the syndrome if a single bit (call it bit i) of the codeword is flipped in the mangling step. Before any bits are flipped, the syndrome will be $\mathbf{0}$.

- If the flipped bit is in the last $n-k$ bits of the codeword (the check bits), this will only flip a single bit in the syndrome. (See Homework Problem 17.)
- If the flipped bit is in the first k bits of the codeword, this will cause all the columns where there are 1s in the i th row of \mathbf{Q} to be flipped in the syndrome. (See Homework Problem 18.)

Alice will try to compute the error vector \mathbf{E} with the fewest nonzero bits that results in the discovered syndrome. The syndrome will be the sum of all the rows of \mathbf{Q} corresponding to the error bits in \mathbf{M} , plus the individual bits corresponding to error bits in \mathbf{C} . If \mathbf{Q} is sparse enough, we can usually derive \mathbf{E} from the syndrome by a process of progressive approximation, looking for rows of \mathbf{Q} with a lot of bits in common with the syndrome bits.

First, Alice attempts to find the 1s in the first k bits of \mathbf{E} . She chooses the row of \mathbf{Q} that, when \oplus 'd with the syndrome, makes the biggest reduction in the number of 1s in the syndrome. Some of the 1s in the chosen row of \mathbf{Q} might not have corresponding 1s in the syndrome. This will cause those bits of the syndrome to become 1s, but if this is indeed the best row of \mathbf{Q} , the total number of 1s in the syndrome will decrease. Then Alice tries another row of \mathbf{Q} . Each row of \mathbf{Q} represents a corresponding 1 in the first k bits of \mathbf{E} .

Once Alice has guessed, say, s bits in the first k bits of \mathbf{E} so that there are less than $t-s$ remaining 1s in the syndrome, she can assume that the remainder of the 1s are due to 1s in the final $n-k$ bits of \mathbf{E} .

It is possible for Alice's algorithm to incorrectly choose a bit to flip and will find that unflipping that bit later will be the choice that most reduces the number of 1s in the syndrome. It is possible that the process will loop and never terminate. And it is possible that she will get a wrong answer, *i.e.*, a different codeword happens to be within t bits of what she received from Bob, so she'll compute a different error vector than Bob had chosen.

The fact that the algorithm might not terminate, or might get the wrong answer, could seem somewhat disturbing. However, with the parameter values in the proposed code-based post-quantum schemes, the probability of an issue is estimated to be very small (*e.g.*, for NIST security level 1, less than 1 in 2^{128}).

Not all error correction schemes are probabilistic. The one we are describing is based on MDPC codes and is probabilistic. However, there are other types of error correction codes. The original McEliece cryptosystem, published in 1978, was based on Goppa codes, and the cryptosystem called classic McEliece also uses Goppa codes. Goppa codes are better behaved (always terminating and never having multiple potential codewords within t bits). They also correct more errors for a given message and codeword size than MDPC codes. However, since the math for

understanding Goppa codes is somewhat harder than the math for understanding MDPC codes, we will not be explaining Goppa codes in this book. Additionally, the important key-size reduction optimization, circulant matrices, works well for MDPC codes, but not nearly as well for Goppa codes*, so the MDPC scheme we describe has significant advantages besides being easier to explain.

8.4.2 The Parity-Check Matrix

The usual implementation of an MDPC scheme involves a second matrix, known as a **parity-check matrix**, which we'll call \mathbf{H} . Using the parity-check matrix \mathbf{H} is a more elegant way to get the syndrome than the procedure explained in §8.4.1.4 *Diagnosis Step*. With the parity-check matrix \mathbf{H} , Alice just needs to multiply the (possibly mangled) codeword \mathbf{Y}' by \mathbf{H} and she'll directly get the syndrome. She will still have to do the same work to figure out which bits were flipped.

The matrices \mathbf{G} and \mathbf{H} are related. Indeed, in the scheme we've been explaining so far, the only difference between \mathbf{G} and \mathbf{H} is where we glue the identity matrix onto our matrix \mathbf{Q} . In \mathbf{G} , it's on the left. In \mathbf{H} , it's on the bottom. Generator and parity-check matrices that are constructed by gluing an identity matrix onto another matrix this way are said to be in **systematic form**. (See Figure 8-9.)

In our cryptographic scheme that we will explain shortly, Alice will use a parity-check matrix that is not in systematic form to generate a syndrome. So, we will give a more general definition of what makes something a parity-check matrix and how it is related to the generator matrix: \mathbf{G} will always be a $k \times n$ matrix (so that when k -bit string \mathbf{M} is multiplied by \mathbf{G} we'll get an n -bit codeword \mathbf{Y}). \mathbf{H} will always be an $n \times (n-k)$ matrix with linearly independent columns. \mathbf{H} 's size is $n \times (n-k)$ so that when an n -bit, possibly mangled, codeword is multiplied by \mathbf{H} , we'll get an $(n-k)$ -bit syndrome. The crucial relation between \mathbf{G} and \mathbf{H} is that their product is a zero matrix, *i.e.*, $\mathbf{GH} = \mathbf{0}$. This guarantees that the syndrome for any unaltered codeword is $\mathbf{0}$. It also guarantees that for a mangled codeword \mathbf{Y}' , the syndrome $\mathbf{Y}'\mathbf{H}$ is the same as \mathbf{EH} , where \mathbf{E} is the error vector. (See Homework Problem 19.)

A scheme that produces a codeword by multiplying a message by a generator matrix is known as a **linear code**. All linear codes have parity-check matrices. (Indeed, any linear code typically has a large number of different parity-check matrices.) But, what makes an MDPC code special is that it has at least one sparse parity-check matrix. The fact that the parity-check matrix \mathbf{H} is sparse makes it computationally feasible to decode a mangled codeword \mathbf{Y}' .

*It may be possible to get some keysize reduction by making parts of the public key of a Goppa-code-based cryptosystem circulant, but not nearly as much as with moderate density parity-check codes. The first-round NIST submission BIG-QUAKE attempts to do this, resulting in a public keysize of about 2×10^5 bits instead of about 2×10^6 . For comparison, MDPC codes using circulant matrices typically achieve keysizes on the order of 10^4 bits for the same security level. Previous attempts to get smaller keysizes with Goppa codes have proved insecure.

$$\begin{array}{c}
 k \times n \text{ generator matrix } \mathbf{G} \\
 \begin{array}{ccc}
 k \times k \text{ Identity} & k \times (n-k) \text{ matrix } \mathbf{Q} & n \times (n-k) \text{ parity-check matrix } \mathbf{H}
 \end{array}
 \end{array}$$

$$\left[\begin{array}{cccccc|cccccc}
 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
 0 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 \vdots & \vdots & \ddots & & \vdots & \vdots & & \cdots & \cdots & & & \\
 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1
 \end{array} \right]$$

$$\begin{array}{c}
 (n-k) \times (n-k) \text{ Identity}
 \end{array}$$

$$\left[\begin{array}{cccccc|cccccc}
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \vdots & \vdots \\
 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
 \end{array} \right]$$

Figure 8-9. Generator Matrix and Parity-check Matrix for Systematic Form

How can a generator matrix \mathbf{G} have lots of different parity-check matrices? If \mathbf{H} is a parity-check matrix for \mathbf{G} , and if \mathbf{R} is any invertible square matrix with the same number of columns as \mathbf{H} , then \mathbf{HR} will also be a parity-check matrix for \mathbf{G} , because $\mathbf{YH} = \mathbf{0}$ iff $\mathbf{Y}(\mathbf{HR}) = \mathbf{0}$. (See Homework Problem 20.)

8.4.3 Cryptographic Public Key Code-based Scheme

So far, we've described how Alice can create an MDPC code, but the scheme we described so far would not work as a cryptographic scheme, because anyone that could create a codeword would also be able to find errors. To create a public key scheme, we need to modify our scheme so that Bob can produce codewords (using Alice's public key), but only Alice (using her private key) can find codewords that differ from mangled codewords by fewer than t bits.

Alice's public key will be a generator matrix \mathbf{G} . To establish a shared secret with Alice, Bob chooses an error vector \mathbf{E} with at most t 1s, computes a codeword, mangles that codeword by \oplus ing with \mathbf{E} , and the secret that Alice and Bob will share is a hash of \mathbf{E} , e.g., SHA-256.

To create her key pair, Alice will start by creating a sparse parity-check matrix \mathbf{H} , which will be her private key. Then she will produce a generator matrix \mathbf{G} that is compatible with \mathbf{H} (*i.e.*, $\mathbf{GH}=\mathbf{0}$), but where \mathbf{G} is not sparse. This \mathbf{G} will be her public key.

Alice's sparse parity-check matrix \mathbf{H} (her private key) will not be in systematic form. However, her public key, the generator matrix \mathbf{G} , will be in systematic form. Having \mathbf{G} be in systematic form allows Alice's public key to be smaller (since she doesn't need to send the $k \times k$ identity matrix), and it also allows a very cute optimization due to Niederreiter (described in the next section), which reduces the size of the ciphertext. We'll describe this optimization before explaining in detail how to compute \mathbf{G} from \mathbf{H} .

8.4.3.1 Niederreiter Optimization

Remember that Bob is going to choose an error vector \mathbf{E} with at most t 1s, and the secret Alice and Bob will share is $\text{hash}(\mathbf{E})$. Niederreiter observed in his paper [NIED86] that if Bob uses the first k bits of \mathbf{E} as the string \mathbf{M} , and if the generator matrix produces systematic form codewords, then when \mathbf{E} is \oplus 'd with the resulting codeword $(\mathbf{M} | \mathbf{C})$, the first k bits of the mangled codeword will be $\mathbf{0}$. Therefore, if this optimization is built into the scheme (everyone uses the first k bits of \mathbf{E} as the string \mathbf{M}), Alice can pretend that the $n-k$ bits of ciphertext she receives from Bob was actually an n -bit ciphertext with the first k bits all zero. Since typically k is about half n , this means that the size of what Bob needs to transmit to Alice is half as big. (See Figure 8-10.)

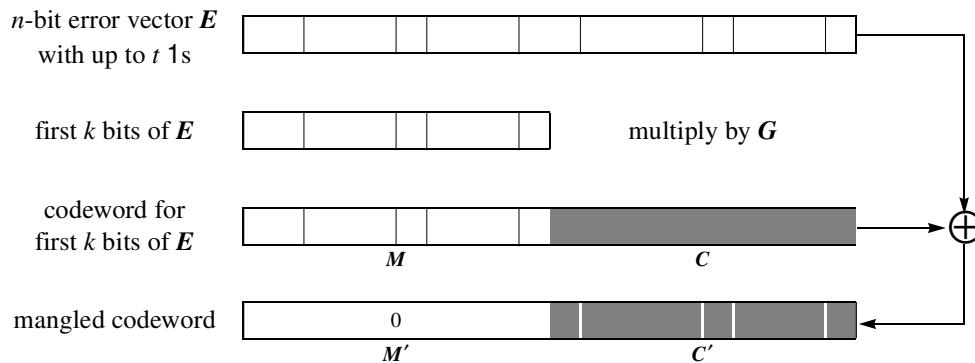


Figure 8-10. Niederreiter Optimization

We will build Niederreiter's optimization into the scheme we are describing.

8.4.3.2 Generating a Public Key Pair

Alice creates a randomly generated sparse $n \times (n-k)$ matrix \mathbf{H} , which will be her private key. Let's break \mathbf{H} into two parts (see Figure 8-11):

- Matrix \mathbf{A} of size $k \times (n-k)$
- Matrix \mathbf{B} of size $(n-k) \times (n-k)$

$$\begin{array}{c}
 \begin{array}{c} n \times (n-k) \text{ matrix } \mathbf{H} \\ \hline
 \end{array} \\
 \begin{array}{c} k \times (n-k) \text{ matrix } \mathbf{A} \\ \hline
 \end{array} \\
 \begin{array}{c} (n-k) \times (n-k) \text{ matrix } \mathbf{B} \\ \hline
 \end{array}
 \end{array}
 \left[\begin{array}{cccccc|cccccc}
 0 & 0 & 0 & 0 & 1 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \vdots & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 1 & \\
 \dots & & & & & & \dots & & & & & \\
 \hline
 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 1 & 0 & \\
 0 & 0 & 1 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & \vdots & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & & 0 & 1 & 0 & 0 & 0 & 0 \\
 \dots & & & & & & \dots & & & & & \\
 \hline
 0 & 0 & 0 & 0 & 0 & & 1 & 0 & 0 & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \vdots & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & & 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0
 \end{array} \right]$$

Figure 8-11. Sparse Parity-Check Matrix \mathbf{H}

Alice will now obfuscate \mathbf{H} , meaning that she will turn it into a parity-check matrix for the same code that will not be sparse (and will be in systematic form).

8.4.3.2.1. Calculating the Public Key

Alice calculates \mathbf{B}^{-1} . By definition of inverse, $\mathbf{B}\mathbf{B}^{-1} = \mathbf{I}$, the identity matrix. She multiplies her private key matrix \mathbf{H} by \mathbf{B}^{-1} and gets the result in Figure 8-12.

Not only does multiplying by \mathbf{B}^{-1} turn the top half into a dense matrix, and therefore not useful for decoding, but the advantage of turning the bottom half into an identity matrix is that it will be really easy to create the systematic form generator matrix \mathbf{G} by using the top half, \mathbf{AB}^{-1} , as what we called the matrix \mathbf{Q} in Figure 8-9.

Her public key, therefore, is just the matrix \mathbf{AB}^{-1} . The generator matrix \mathbf{G} is created by gluing a $k \times k$ identity matrix to the left (see Figure 8-13). Her private key is \mathbf{H} . The reader may check that \mathbf{G} and \mathbf{H} have the required relationship of a generator and a parity-check matrix, $\mathbf{GH} = \mathbf{0}$. (See Homework Problem 21.)

$$\begin{array}{c}
 n \times (n-k) \text{ matrix } \mathbf{H}\mathbf{B}^{-1} \\
 \hline
 k \times (n-k) \text{ matrix } \mathbf{A}\mathbf{B}^{-1} \\
 \hline
 (n-k) \times (n-k) \text{ Identity}
 \end{array}
 \left[\begin{array}{ccccccccc}
 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
 1 & 1 & 0 & 1 & 1 & \vdots & 1 & 0 & 1 & 1 \\
 0 & 0 & 0 & 1 & 0 & & 0 & 0 & 0 & 1 \\
 \dots & & & & & & \dots & & & \\
 \hline
 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & & 0 & 0 & 0 & 0 \\
 \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots & \vdots & \vdots & \vdots \\
 0 & 0 & 0 & 0 & 0 & & 1 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1
 \end{array} \right]$$

Figure 8-12. Obfuscated \mathbf{H}

$$\begin{array}{c}
 k \times n \text{ matrix } \mathbf{G} \\
 \hline
 k \times k \text{ Identity} \qquad \qquad \qquad k \times (n-k) \text{ matrix } \mathbf{A}\mathbf{B}^{-1} \text{ (Alice's public key)}
 \end{array}
 \left[\begin{array}{ccccccccc}
 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & & 0 & 0 & 0 & 0 & 0 \\
 \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots & \vdots & \vdots & \vdots & \vdots \\
 0 & 0 & 0 & 0 & 0 & & 1 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1
 \end{array} \right]$$

Figure 8-13. Alice's Generator Matrix

Here are the steps in the protocol, using public key \mathbf{G} and private key \mathbf{H} to establish the shared secret hash(E):

1. Alice sends Bob her public key \mathbf{G} . (Since the first k columns of \mathbf{G} are an identity matrix, Alice only needs to send the last $n-k$ columns.)
2. Bob chooses an n -bit error vector \mathbf{E} , with at most t 1s.
3. Bob multiplies the first k bits of \mathbf{E} by \mathbf{G} to get n -bit codeword \mathbf{Y} . Then he \oplus s \mathbf{E} with \mathbf{Y} to get \mathbf{Y}' . (Since this construction forces the first k bits of \mathbf{Y}' to be zero, Bob only needs to send the final $n-k$ bits of \mathbf{Y}' .)
4. Alice then takes \mathbf{Y}' (the quantity she received from Bob, with k bits of 0 inserted at the beginning) and multiplies it by \mathbf{H} to get the syndrome. (Since the first k bits of \mathbf{Y}' are zero, Alice can get the same syndrome by multiplying the last $n-k$ bits of \mathbf{Y}' , i.e., what she received from Bob, by \mathbf{B} .) (See Homework Problem 22.)
5. She then uses the sparse \mathbf{H} to calculate \mathbf{E} .

8.4.3.3 Using Circulant Matrices

The scheme we described can easily use circulant matrices. Alice creates her sparse parity-check matrix (see Figure 8-11) as two sparse circulant matrices \mathbf{A} and \mathbf{B} . Since \mathbf{B} is circulant, \mathbf{B}^{-1} will also be circulant, as will \mathbf{AB}^{-1} , her public key. So her public key is just the top row of \mathbf{AB}^{-1} .

The NIST submission that is closest to what we've described is BIKE. It uses both circulant matrices and the Niederreiter optimizations. Its k is half n . Therefore, the public key is of size k , and for NIST security levels 1, 3, and 5, k is 12323, 24659, and 40973 bits, respectively.

8.5 MULTIVARIATE CRYPTOGRAPHY

Multivariate cryptosystems are based on the difficulty of solving systems of nonlinear (usually quadratic) equations over many variables. Linear equations are easy to solve, but nonlinear equations are not.

In a linear (degree 1) equation, each term is a constant, or the product of a constant and a single variable. For example, $x+15y-2z+3q=15$.

In contrast, in a nonlinear equation, variables may be multiplied by themselves or each other. For example, a degree 3 equation might have terms such as $4x^3$, $19xyz$, $3xy^2$.

Multivariate public key signature schemes (e.g., Unbalanced Oil and Vinegar (UOV) and Rainbow [DING05]) use degree 2 (quadratic) equations with more than a hundred variables. UOV and Rainbow have very large public keys, but they have fast signing and verification and small

signatures, and so it seems like they could be useful for applications where the public key doesn't have to be downloaded every time a signature needs to be verified (*e.g.*, software or firmware updates).

What about multivariate encryption schemes? There are some proposed schemes, but of the ones that have not been broken, their performance characteristics do not seem promising compared to encryption schemes based on codes or lattices.

We will first describe the intuition behind multivariate signature schemes in general. Then we will explain one of the oldest, and arguably the simplest, multivariate signature scheme, UOV.

8.5.1 Solving Linear Equations

It is easy to solve a set of n (linearly independent) linear equations in n variables. For example, consider these two linear equations in two variables:

$$\begin{aligned} 3x + 2y &= 19 \\ 4x + y &= 22 \end{aligned}$$

Use the second equation to solve for y in terms of x , getting $y = 22 - 4x$. Then substitute $22 - 4x$ for y in the first equation to get $3x + 2(22 - 4x) = 19$. Simplify that to get $44 - 19 = 5x$, then divide by 5 on each side to get $x = 5$, and substitute 5 for x in the equation $y = 22 - 4x$ to get $y = 2$.

If the equations are linearly independent and we have as many equations as variables, we can solve for all the variables.

8.5.2 Quadratic Polynomials

A quadratic polynomial over two variables might look like $x^2 + 3xy + 2y^2 + 3y + 4$. A quadratic equation sets the polynomial to some value, so a quadratic equation might be $x^2 + 3xy + 2y^2 + 3y + 4 = 17$.

In a quadratic polynomial (or quadratic equation), each term has at most two variables multiplied together (for instance, x^2 is x multiplied by itself, and $3xy$ has x and y multiplied together). A quadratic polynomial over the four variables x, y, z, q might be $5xq + 7z^2 + 4q^2 + 5xz + 7y^2 + 3y + 2x$. It becomes a quadratic equation in four variables if we specify a value, *e.g.*, $5xq + 7z^2 + 4q^2 + 5xz + 7y^2 + 3y + 2x = 85$.

8.5.3 Polynomial Systems

A polynomial system is a set of polynomials over some number of variables. It is characterized by its degree (the maximum number of variables that are multiplied together in any term), the number

of variables, and the number of polynomials. So for instance, the following polynomial system has degree = 2 (it's quadratic), number of variables = 2, and number of polynomials = 3.

$$\begin{aligned} &x^2 + 3xy + 2y^2 + 3y + 4 \\ &xy + y^2 + x + 2y \\ &5x^2 + 3y^2 + 7xy + 2x \end{aligned}$$

As with other algorithms used in cryptography, in multivariate schemes, the arithmetic is typically done over a small finite field, *e.g.*, GF(64), the 6-bit finite field. See §M.7.2 *Finite Fields*.

Solving a set of polynomial equations means finding values of the variables that satisfy all the equations. In general, this is very difficult. The best-known techniques are not much better than using brute force to choose values of the variables. With typical parameters for a multivariate signature scheme (*e.g.*, more than 100 variables and 50 equations), this is computationally infeasible. This is wonderful for us, because it means it can be used to create a public key cryptosystem.

8.5.4 Multivariate Signature Systems

Alice's public key is the set of coefficients of the terms in a set of polynomials. Her private key in this sort of cryptography is some sort of trapdoor function that enables her to efficiently solve a set of equations based on the polynomials in her public key.

In a given scheme, the size and representation of the finite field, the number of variables, the number of polynomials, the maximum degree of the equations, and so on, are parameters. For readability, though, we'll choose specific numbers in our description.

In a general quadratic polynomial with n variables, there are approximately $n^2/2$ terms— $n(n-1)/2$ terms with two different variables, n terms with one squared variable, n linear terms, and one constant term, for a total of $n^2/2 + 3n/2 + 1$ terms. With 150 variables, that's 11476 terms per polynomial. For 6-bit coefficients, each polynomial contributes about 69 thousand bits to the public key. So with 50 polynomials, the public key is about 3.45 million bits.

8.5.4.1 Multivariate Public Key Signatures

Numbers of variables, size of coefficients, and size of hash are all parameters of a specific scheme. We will pick values that are easy to visualize, but they are just for explanation.

Suppose Alice's public key consists of 50 quadratic polynomials with 150 variables, using 6-bit arithmetic and a 300-bit hash function. To sign a message M , Alice hashes M to get a value H . Alice will partition the 300-bit H into fifty 6-bit chunks, say H_1, H_2, \dots, H_{50} . She'll create 50 equations by setting each polynomial to the corresponding chunk of H .

Alice's signature consists of values of the 150 variables that satisfy those 50 equations.

To verify a signature on M , the verifier computes the 300-bit hash of M , sets each of the polynomials in Alice's public key to the corresponding chunk of the hash of M , plugs the specified val-

ues of the variables into the equations, and verifies that each equation is satisfied. If there are 150 variables, each requiring six bits to specify, the signature consists of 900 bits.

This small signature size is a very nice feature of multivariate signature schemes, although they unfortunately do have very large public keys.

8.5.4.1.1. Creating a (Public, Private) Key Pair

To create a multivariate key pair, one needs to produce a system of polynomials with a trapdoor that allows someone with knowledge of the secret to solve equations based on these polynomials but is infeasible to solve without the secret.

The simplest apparently secure way of creating a trapdoor is a scheme known as **unbalanced oil and vinegar** [KIPN99]. To create a public key, the oil and vinegar approach starts with, say, 150 variables, and partitions the variables into two categories—*oil* and *vinegar*. There will be more vinegar variables than oil variables. Let’s use 50 oil variables and a 100 vinegar variables. Alice creates what we’ll call **constrained quadratic polynomials** for her private key, by randomly choosing coefficients for all possible terms of degree two or less, with the restriction that oil variables are never multiplied together—any term with an oil variable consists of that variable alone, or that variable times a vinegar variable. Vinegar variables, however, can be squared, multiplied by each other, or multiplied by an oil variable.*

Suppose we have oil variables o_1 and o_2 and vinegar variables v_1, v_2, \dots, v_5 . Then this is an example of a constrained quadratic: $3o_2v_5 + 5v_2v_3 - 7 + v_2 + 2o_1 + 4v_5^2$. Examples of terms that are not allowed: $o_1o_2, o_1^2, o_1v_1^2, v_1v_4^2$.

As we’ll see, the purpose of the restriction (that oil variables not be multiplied by each other, or squared), is that with constrained quadratic equations, if values are chosen for the vinegar variables, it will be easy to solve for the oil variables, because the equations at that point will be linear in the oil variables. But it wouldn’t be secure if the constrained quadratic polynomials were Alice’s public key, because then anyone would be able to forge Alice’s signature. Therefore, the polynomials in her private key have to be transformed so that the polynomials in the public key will be quadratic (not constrained quadratic), and it will not be computationally feasible (without the private key) to transform the public polynomials back into a form where it is possible to tell which are the oil variables.

In our example, we are using the parameter values of 50 oil variables, 50 polynomials, 100 vinegar values, and a hash function of 300 bits. To create her private key, Alice randomly selects 50 constrained quadratic polynomials with 50 oil and 100 vinegar variables. Knowledge of this set of polynomials will be part of her private key. Let’s call these variables u_1, u_2, \dots, u_{150} (the first 50 of

*Note that whoever came up with this terminology seems to think that vinegars mix with each other, oils mix with any vinegar, but oils do not mix with each other. We admire their cryptographic skills (this scheme is very clever), but we wouldn’t trust them in our kitchen.

which are oil and the others are vinegar). As we'll see, the public key will be a transformation of the constrained quadratics into a set of unconstrained quadratics, so that only Alice will be able to create a signature.

8.5.4.1.2. Solving Constrained Quadratics

The first step in creating a signature for message M will be solving for the variables in the constrained quadratics in her private key.

- Alice computes $\text{hash}(M)$ and partitions the result into 6-bit chunks (which, with a 300-bit hash, will be 50 chunks).
- She sets each of the constrained quadratic polynomials in her private key to one of the 6-bit chunks of the $\text{hash}(M)$.
- She randomly chooses values for the 100 vinegar variables and substitutes those values in the equations, resulting in 50 linear equations with 50 variables.
- She solves these linear equations, so that she has values for all the oil variables that satisfy the equations.
- She now has values for all the variables in her private key (the 100 vinegar variables for which she chose random values and the 50 oil variables she solved for).

However, she has to transform the polynomials and variables from her private key into the obfuscated polynomials and variables that will be her public key.

8.5.4.1.3. Obfuscating to Create a Public Key

Alice is going to convert the 150 u variables in the constrained quadratic polynomials of her private key into variables that we'll call x variables. The result will be unconstrained quadratic polynomials, also in 150 variables. This is done as follows:

- Alice randomly selects 150 linear equations, each one setting one of the u_i to a linear combination of the x variables, *e.g.*, $u_3=3x_1+17x_2+x_4+\dots+9x_{150}$. These equations will also be part of the private key.
- Alice substitutes the value (in terms of x variables) of each of the u variables in each of the 50 constrained quadratic polynomials in her private key. The result will be 50 (unconstrained) quadratic polynomials with variables x_1,\dots,x_{150} . Those 50 polynomials will be the public key. Note that the polynomials in the public key will be quadratic (*i.e.*, substituting a linear combination of x variables for any of the u variables will not result in any terms of degree larger than two).

As we will see, Alice will solve equations based on the u variables in her constrained quadratics of her private key but will need to demonstrate values of the x variables for her signature.

Alice can solve for each of the x variables in the 150 linear equations that express each u in terms of x variables. Now she has 150 linear equations that express each x variable in terms of u variables.

Once Alice has solved her constrained quadratics to find values of all the u variables, she uses these equations to calculate the values of the x variables.

Alice's signature consists of the values of the x variables that satisfy the quadratic equations in her public key.

8.6 HOMEWORK

Hash-based signatures

1. Consider the following variant of the original scheme (no hash tree). Instead of Alice's public key consisting of 512 quantities $h_0^0, h_1^0, h_0^1, h_1^1, \dots, h_0^{255}, h_1^{255}$, use only a single hash for each bit, *i.e.*, h^0, h^1, \dots, h^{255} . If bit i is 0, she reveals p^i , and if i is 1, she reveals h^i . Why is this not secure?
2. Now we'll modify the scheme in Homework Problem 1 by using an additional eight values to sign the sum of the number of 0 bits. We'll call these extra eight values $h^{256}, h^{257}, \dots, h^{263}$, computed as the hashes of p^{256}, p^{257}, \dots . If the total number of 0s in the signed message is 120 (decimal), that would be 01111000 binary. Therefore, Alice will divulge $p^{256}, h^{257}, h^{258}, h^{259}, h^{260}, p^{261}, p^{262}$, and p^{263} . Is this scheme secure?
3. In §8.2.3 *Signing Lots of Messages*, we compared having a single tree with at least as many treelets as messages to be signed, versus reserving one slot in the tree for signing a root of a new tree, versus having a single tree whose treelets are used solely to sign roots of other trees. Suppose we have a 3-level structure, where the primary tree has, say k levels (2^k treelets), with all the treelets of that primary tree used to sign roots of secondary k -level hash trees, all of whose treelets are used to sign roots of tertiary k -level hash trees. The tertiary k -level hash trees are used to sign messages. How many messages could be signed with this scheme? How big would signatures be?
4. Suppose Alice is using the scheme where each treelet has 512 children, a pair $\langle h_0^i, h_1^i \rangle$ for each of the 256 possible bits. Suppose Alice signs a message using that treelet. How many messages (approximately) would Ivan need to try in order to find a message that he can forge using that treelet (given that for each i , either p_0^i or p_1^i would have been divulged)? What if

Alice accidentally uses the treelet twice for two different messages? How many hashes would Ivan need to try? What about if Alice signed three different messages with that treelet?

Lattice-based cryptography

5. Multiply the polynomial $3 - 12x + 2x^2 + 5x^4$ by x and reduce mod $x^5 - 1$.
6. Show that the sum of two circulant matrices is a circulant matrix.
7. Show that the product of two circulant matrices is a circulant matrix whose polynomial is equal to the product of the corresponding polynomials of the two matrices, reduced mod $x^n - 1$.
8. Show that multiplication of circulant matrices is commutative.
9. Show that if a circulant matrix has an inverse, the inverse will be circulant. Hint₁: Matrix inverses are unique. Hint₂: Show that the circulant matrix formed from the first row of the inverse is an inverse.
10. For a polynomial f over a field like \mathbf{Z}_2 , we can calculate the inverse of f mod $x^n - 1$ by using the Euclidean algorithm (§2.7.5) to get polynomials a and b such that $a \times f + b \times (x^n - 1) = 1$, provided that f is relatively prime to $x^n - 1$. But \mathbf{Z}_q is not a field, so how do we get f^{-1} ? It's actually quite clever. First, we calculate the inverse mod 2, which we can do since \mathbf{Z}_2 is a field. Let a be that inverse; its coefficients are all zeroes and ones, but we will consider the coefficients mod q . We then iterate $\lceil \log_2 \log_2 q \rceil$ times: $a = a \times (2 - f \times a)$ mod $x^n - 1$ mod q . Why does this work? Hint: What's special about the coefficients of the error polynomial $1 - f \times a$ before the first iteration and then after each successive iteration?
11. In the LWE cryptosystem (see §8.3.5) would it be insecure if Alice and Bob did not add their error vectors (*i.e.*, used $\mathbf{0}$ as their error vector)? For simplicity, assume the matrix A is invertible.
12. Why would it be insecure for Alice to send multiple bits using the same Z ?
13. Suppose Alice and Bob want to agree on 256 secret bits, but the bandwidth from Alice to Bob is much greater than the bandwidth from Bob to Alice. Would the scheme work if Alice transmitted 256 vectors, and Bob would send only one vector? If Bob only sends four vectors, how many vectors would Alice need to send to Bob in order for them to agree on 256 secret bits?
14. If Bob sends four bits at a time (see §8.3.5.1.2), how many $\langle i, j \rangle$ pairs are required to send 256 secret bits? Assuming Alice and Bob send equal numbers of vectors, how many vectors does each need to send?

Code-based schemes

15. Multiply the bit vector 11001 by the 5×5 identity matrix. What is the result?
16. Suppose we modified the generator matrix from Figure 8-7 so that the identity matrix was on the right instead of the left. What would the codeword for string M look like?

17. Suppose the error vector has only a single 1, and it is in the last $n-k$ bits. (See §8.4.1.4.) Show that the syndrome will have only a single 1 bit.
18. Suppose the error vector has only a single 1, say bit j , and j is in the first k bits. Show that the syndrome will be the j th row of \mathbf{Q} . (See §8.4.1.4.)
19. Show that for a mangled codeword $\mathbf{Y}' = \mathbf{MG} + \mathbf{E}$, the syndrome $\mathbf{Y}'\mathbf{H}$ is the same as \mathbf{EH} , where \mathbf{E} is the error vector. Hint: $\mathbf{GH} = \mathbf{0}$.
20. Show that if \mathbf{H} is a parity-check matrix for \mathbf{G} , and \mathbf{R} is an invertible square matrix with the same number of columns as \mathbf{H} , $\mathbf{YH} = \mathbf{0}$ iff $\mathbf{Y}(\mathbf{HR}) = \mathbf{0}$ for any (possibly mangled) codeword \mathbf{Y} . What if \mathbf{R} isn't invertible?
21. As described in §8.4.3.2.1 *Calculating the Public Key*, Alice's public key is just the matrix \mathbf{AB}^{-1} . The generator matrix \mathbf{G} is created by gluing a $k \times k$ identity matrix to the left. Her private key is \mathbf{H} , where the top half of \mathbf{H} is the sparse matrix \mathbf{A} , and the bottom half is the sparse matrix \mathbf{B} . Verify that $\mathbf{GH} = \mathbf{0}$.
22. As described in §8.4.3.2.1, Alice computes the syndrome by taking the n -bit quantity \mathbf{Y}' (the quantity she received from Bob, with k bits of 0 inserted at the beginning) and multiplies it by \mathbf{H} . Verify that since the first k bits of \mathbf{Y}' are zero, Alice can get the same syndrome by multiplying the last $n-k$ bits of \mathbf{Y}' , i.e., what she received from Bob, by \mathbf{B} .
23. Suppose $n=2k$ (so in Figure 8-9, \mathbf{Q} will be the same size as \mathbf{I}). Show that $\mathbf{GH} = \mathbf{0}$. Now suppose k is not half of n (so \mathbf{Q} and \mathbf{I} will not be the same size). Will \mathbf{GH} in Figure 8-9 still be $\mathbf{0}$?
24. Assume that $n=2k$. What is \mathbf{GH} from Figure 8-13 (where $\mathbf{G} = \mathbf{I}|\mathbf{AB}^{-1}$, and \mathbf{H} is \mathbf{A} on top and \mathbf{B} on the bottom)? What is \mathbf{GHB}^{-1} from Figure 8-13?
25. Can Bob, knowing only \mathbf{AB}^{-1} and not the sparse \mathbf{H} , be able to tell whether an n -bit quantity is a codeword?
26. Suppose an attacker Eve can send arbitrary ciphertext messages to Alice and be informed by Alice when there is a decryption failure. Suppose also that Alice doesn't change her public key. How can an attacker that sees the ciphertext message \mathbf{Y}' that Bob sends to Alice eventually find the secret Bob was sending to Alice? Assume that \mathbf{Y}' is a mangled codeword with exactly t errors and that any codeword with t or fewer errors will decrypt properly, while any mangled codeword with more than t errors will produce a decryption failure. Minor hint: Each message Eve sends allows her to recover one bit of the error vector used to produce \mathbf{Y}' .
27. Bob purposely mangles the codeword in places where the error vector has 1s. What happens if the communication channel between Alice and Bob introduces more errors? If, for an n -bit message, the channel might introduce, say, a hundred bit errors, should Alice and Bob use an error-correcting code that can find up to $t+100$ errors?

Multivariate cryptography

28. Suppose variables a, b, c are oil variables, and w, x, y, z are vinegar variables. Which of the following terms would be allowable in a constrained quadratic: $5wz, 12a, 4az, 3a^2, 6aw^2, 2bc, 8w$?
29. Suppose the parameter set in an oil-vinegar scheme has more oil variables than equations. How could Alice solve her constrained quadratics? For example, suppose there were 80 oil variables, 80 vinegar variables, and 50 constrained quadratic equations in her private key.
30. Suppose there are fewer oil variables than equations. For example, suppose Alice's constrained quadratics had 20 oil variables and 100 vinegar variables, with 50 equations? Can Alice solve these constrained quadratics?
31. Once the constrained quadratics (in u variables) are translated into x variables, why will the result still be a quadratic?

9

AUTHENTICATION OF PEOPLE

Humans are incapable of storing high-quality cryptographic keys and they have unacceptable speed and accuracy when performing cryptographic operations. They are also large, expensive to maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed, but they are sufficiently pervasive that we must design our systems around their limitations.

—Radia Perlman

Authentication is the process of reliably verifying the identity of someone (or something). There are lots of examples of authentication in human interaction. People who know you can recognize you based on your appearance or voice. A guard might authenticate you by comparing you with the picture on your badge. A mail order company might accept as authentication the fact that you know the number and expiration date on your credit card.

We sometimes use the names Alice and Bob for machines. We clarify examples when necessary by using the term *the human Alice* or *the user Bob* when we are referring to a human. However, in this chapter, Alice is always a human, and she is not authenticating to a human, so we will name what she is authenticating to as *Steve* to make it clear Steve is not a human, but is a server somewhere. We do not intend to imply that everything named Steve is not a human.

There are special challenges when humans are involved. If it were simply two computers communicating across a network, they are perfectly capable of storing high-quality cryptographic keys and doing cryptographic authentication.

There are various scenarios in which humans might authenticate to something:

- A human authenticating to a local resource to unlock a door, a car, or a phone.
- A human authenticating to a remote resource across the network, using a personal device (e.g., a laptop, a phone) that will act on the human's behalf. The device might store credentials for the user, making remote authentication on the user's behalf more user-friendly. But a complication is that the user might want to use several personal devices and have a newly purchased device somehow obtain the credentials for the user.

- A user might want to use a public device, such as a hotel lobby computer, or someone else's computer. This device will not have stored credentials for the user, and the user will need to enter credentials in order to access remote services. Hopefully the device will forget the credentials after this one use. Note our use of *hopefully*. If the device is not trustworthy, it could remember the credentials, or send them off to some criminal organization.

The traditional categories of techniques for authenticating a human are: what you know, what you have, what you are, and where you are. Passwords fit into the *what-you-know* category. Physical keys or ATM cards fit into the *what-you-have* category. Biometric devices, such as voice recognition or fingerprint scanners, fit into the *what-you-are* category. Having a resource only reachable from a user located at a physically protected location or having resources authenticate a user (or decide what the user should have access to) based, for example, on the source IP address in received data packets count as *where you are*.

Each of these categories has problems. A secret (*what you know*) can be accidentally overheard or forgotten. A physical thing (*what you have*) can be stolen or lost or broken. A biometric tends to have false positives and false negatives (especially if some injury changes the biometric, such as having a bandage on your finger, or a cold that changes your voice), and may require special scanners. Therefore, it is more secure to use **multifactor authentication**, in which the user is authenticated using mechanisms from multiple categories. Some companies advertise their systems as being multifactor when they involve multiple passwords, but that is not what most people think should count as multifactor. A multifactor authentication system should involve different categories. For example, a credit card might include a picture or a signature, so in theory it can combine biometrics with physical access to the card (assuming the person at the store checkout has the time to stare at the tiny picture or do handwriting analysis). Another form of multifactor authentication is having a website send a PIN to the user Alice's phone, so that someone attempting to authenticate as Alice would not only need to know her password but would also have to have access to Alice's phone.

Although multifactor authentication is more secure, it tends to make things less convenient for the user and multiplies reliability issues if any of these systems fail, since with multifactor authentication, all the factors should work.

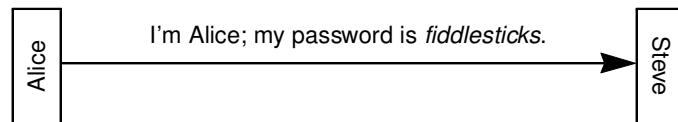
In many scenarios in this chapter, a user Alice will be authenticating across the network to a service S. She will be entering her credentials into some sort of device, such as a tablet, phone, or laptop. We will refer to that device as a *workstation* or *device*, but the term is intended to include all the devices Alice might use.

Also, when Alice's device is communicating with a service on the web, it will usually be using an encrypted connection such as TLS (see Chapter 13 *SSL/TLS and SSH*). But there are still potential security issues, as we will discuss in the examples.

9.1 PASSWORD-BASED AUTHENTICATION

It's not who you know. It's what you know.

If authentication of a human is based on something the human can remember, it is usually a password (or other string of characters that a human can remember and type).



Unfortunately, even computer-computer authentication is often based on passwords. Sometimes, cryptography is not used because the protocol started out as a human-computer protocol and was not redesigned when its use got expanded to computer-computer communication.

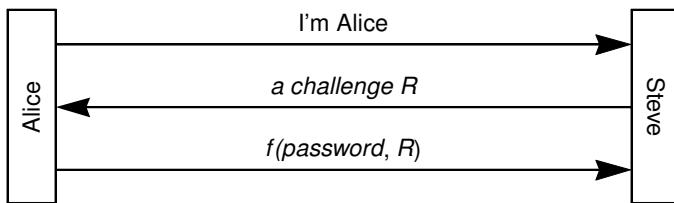
There are some cases in which it is surprising that the designers opted for a simple password-based scheme. For instance, the first generation of cellular phones transmitted the telephone number of the phone and a password (unencrypted) when making a call, and if the password corresponded to the telephone number, the phone company let the call go through and billed that telephone number. The problem was, anyone could eavesdrop on cellular phone transmissions and **clone** such a phone, meaning they could make a phone that used the overheard ⟨telephone number, password⟩ pair. Indeed, it *was* a problem—criminals did clone phones for stealing phone service and/or making untraceable calls. It would have been technologically easy to implement a simple cryptographic challenge-response protocol in the phone, and, indeed, all newer cell phones do this.

Note that without a secure connection between Alice and Steve, an eavesdropper will be able to see the password. Even with a secure connection, if Trudy impersonates Steve and tricks Alice into thinking she is talking to Steve, (perhaps because Trudy has obtained a DNS name that looks to human Alice like “Steve”), Alice will be divulging her password to Trudy, who in our examples, is always evil.

9.1.1 Challenge-Response Based on Password

Note that authenticating using a password can be more secure than simply transmitting the password. Server Steve can send a challenge (a random number), and Alice’s machine can compute a function of Alice’s password and Steve’s challenge (Protocol 9-1). This is basically how the CHAP protocol [RFC 1994] works.

As in the previous example, if Alice is using TLS, there is no security issue with eavesdroppers. However, if Alice can be tricked into talking to Trudy instead of server Steve, Trudy will not



Protocol 9-1. Challenge-Response Based on Password

directly learn Alice's password using this protocol, but she could do a dictionary attack. (See §9.8 *Off-Line Password Guessing*).

9.1.2 Verifying Passwords

Suppose Alice has a single network password, as might be done for accessing resources within the company in which she works. We are not discussing single sign-on here, where Alice types her password once. Instead, we are assuming Alice will need to separately type her password at each of the company resources, but it will be the same password for all of them.

How does a server know that the string Alice typed is her valid password? There are several possibilities:

1. Alice's authentication information is individually configured into every server Alice will use.
2. One location that we'll call an **authentication storage node** stores Alice's information, and servers retrieve that information when they want to authenticate Alice. This model is rarely used today.
3. One location that we'll call an **authentication facilitator node** stores Alice's information, and a server that wants to authenticate Alice sends the information received from Alice to the authentication facilitator node, which does the authentication and tells the server **yes** or **no**. An example of this is RADIUS [RFC 2865].

In cases 2 and 3, it's important for the server to authenticate the authentication storage or facilitator node, since if the server were fooled into thinking a bad guy's node was the authentication storage or facilitator node, the server could be tricked into believing invalid information, and therefore let bad guys impersonate valid users. Or the server might be tricked into forwarding the user's password (or information that would allow a dictionary attack §9.8) to the fraudulent authentication node.

Regardless of where authentication information is stored, it is undesirable to have the database consist of unencrypted passwords, because anyone who captured the database could impersonate all the users. Someone could capture the database by breaking into the node with the database or by stealing a backup. In the first case (authentication information individually configured into every

server), capturing a server's database (of unencrypted passwords) would enable someone to impersonate all the users of that server. Also, if a user had the same password on multiple servers, that user could then be impersonated at the other servers as well. In the second and third cases, many servers would use the one location, and capturing its database would enable someone to impersonate the users of all those servers. There's a trade-off, though. It might be difficult to physically protect every server, whereas if all the security information is in one location, it is only necessary to protect that one location.

Put all your eggs in one basket, and then watch that basket very carefully.

—Anonymous

An alternative to directly storing passwords is to store hashes of passwords. A password would be verified by hashing it and comparing it with the stored value, or using the hashed password as a shared secret in a challenge-response protocol. Then if anyone were to read the password database they could do off-line password-guessing attacks (§9.8), but would not be able to obtain passwords of users who chose passwords carefully. With hashed passwords an intruder who can read the hashed password database can do a password-guessing attack because the intruder will likely know the hash function (the function itself would not be secret).

Note that if the hashed password is transmitted to Bob instead of the actual password, or if it is used as a shared secret in a challenge-response protocol, then the hashed password is known as a **password equivalent**. That means that if Trudy steals Bob's password database, she can use Alice's hashed password to directly impersonate Alice, even if Trudy can't guess Alice's actual password. However, since Alice's client software only allows Trudy to input the actual password (not the hashed password), Trudy will have to modify the client software to bypass the step where it hashes the user input and uses the result for authentication.

9.2 ADDRESS-BASED AUTHENTICATION

It's not what you know. It's where you are.

Address-based authentication does not rely on sending passwords around the network but rather assumes that the identity of the source can be inferred based on the network address from which packets arrive. At one point it was a widely used method of authentication, and it is still sometimes used today. For instance, firewalls might allow access based solely on the source address in the IP header.

In some cases, it might be somewhat safe to rely on address-based authentication, for instance, if the network is completely isolated, or assuming you trust all the bridges (switches) in a bridged Ethernet, you might trust that the network protects you from receiving any packets that originated outside your VLAN.

9.2.1 Network Address Impersonation

Location, location, location

How hard is it for Trudy to impersonate Alice's network address? Generally, it is easy to transmit a packet claiming any address as the source address. Sometimes it is more difficult. A router could have features to make it difficult for someone to claim to be a different network address. If a router has a point-to-point link to an endnode, it could refuse to accept a packet if the source address is not coming from the expected direction.

It is often more difficult for Trudy to receive messages sent by Steve to Alice's network address than for Trudy to send packets using Alice's address as the source address. If Trudy attempts to send packets with Alice's source address and cannot receive packets from Steve to Alice, usually the TCP protocol will fail to work for Trudy. This is because TCP connections start with random sequence numbers. Trudy would not be able to send TCP acknowledgments to Steve's packets, because she would not know the sequence number chosen by Steve. So, Trudy will need to both be able to forge Alice's source address and be able to receive packets from Steve to Alice's address. If Trudy is on the path between Alice and Steve, it is easy, because packets from Steve to Alice will go through Trudy. If Trudy is not on the path, she might be able to get a colluding router on the path between Steve and Alice to hijack packets and forward them to Trudy. Or Trudy might be able to compromise the routing protocol to cause packets to go through Trudy.

Compromising the routing protocol is a common problem with BGP [RFC 4271]. BGP is very fragile and configuration intensive. For instance, in 2008, Pakistan was attempting to block access to YouTube from within Pakistan, so they had their BGP router advertise that it was the best path to YouTube's IP address. Since Internet routing prefers the most specific destination address regardless of distance, and since everyone else was advertising an address block containing YouTube's address, all traffic worldwide for YouTube was routed to Pakistan's BGP router, resulting in accidentally blocking access to YouTube worldwide.

Mobile IP [RFC 5944] is designed to allow a node X to keep its IP address and move anywhere on the Internet. There will be a server S somewhere on the Internet that will receive traffic addressed to a block of addresses (including X). X acquires a temporary address T that is specific to the location where X is currently residing, and X lets S know that X is currently reachable at address T. When Steve sends a packet to X, it will be received by S and then tunneled to X with an

outer header with destination=T. However, return traffic from X to Steve will have source=X and destination=Steve. If routers really tried to enforce that source address X would need to arrive from the expected direction, this would not work. Because enforcing the direction from which an address is received would break mobile IP, most routers do not enforce source address direction.

9.3 BIOMETRICS

Biometric devices authenticate you according to *what you are*. They measure your physical characteristics and match them against a profile. Some people imagine biometrics being used for remote authentication. Although biometrics are useful for local authentication (*e.g.*, unlocking a smartphone or unlocking a door), they are not useful as a “secret” with which to authenticate remotely, because biometrics cannot be assumed to be secret. If you want to know someone’s fingerprint, just offer them a drink of water and get their print off the glass they used. Remote authentication by asserting the value of a biometric can be done safely if the measuring device is trustworthy and has an authenticated connection to the remote server. Whether local or remote, the measuring device must be designed and policed somehow to prevent someone tricking it with a rubber finger with someone else’s fingerprint, or by presenting the user’s severed finger.

Examples of biometrics include fingerprint scans, iris scans, retinal scans, facial features, or hand geometry. DNA would be quite accurate, other than the case of identical twins, though the technology is not quite available. Then there are behavior-measuring strategies, such as typing rhythm, mouse movements, gait, signature, and voice. I₂ don’t quite believe behavior measuring will work. Will my₂ gait change if I₂ have a blister, or if I₂’ve sprained my ankle, or if due to hell freezing over I₂ am wearing high-heeled shoes?

9.4 CRYPTOGRAPHIC AUTHENTICATION PROTOCOLS

*It’s not what you know or where you are, it’s
%zPy#bRw Lq(epAoa&N5nPk9W7Q2EfjaP!yDB\$S*

Cryptographic authentication protocols can be much more secure than either password-based or address-based authentication. The basic idea is that Alice (Alice’s machine) proves Alice’s identity to server Steve by performing a cryptographic operation on a quantity Steve supplies. The cryptographic operation performed by Alice is based on Alice’s secret. We talked in Chapter 2

Introduction to Cryptography about how hashes, secret key cryptography, and public key cryptography could all be used for authentication. The remainder of this chapter discusses some of the more subtle aspects of making these protocols work.

9.5 WHO IS BEING AUTHENTICATED?

Suppose user Alice is at a workstation and wants to access her files at a file server. The purpose of the authentication exchange between the workstation and the file server is to prove Alice's identity. The file server generally does not care which workstation Alice is using.

There are other times when a machine is acting autonomously. For instance, directory service replicas might coordinate updates among themselves and need to prove their identity to each other.

Sometimes it might be important to authenticate both the user and the machine from which the user is communicating. For example, a bank teller might be authorized to do some transactions but only from the machine at the bank where that teller is employed. Or Alice might be allowed to download a movie but only from a device that is trusted (by the content provider) to enforce rules such as deleting the movie after some period, or never copying the movie.

A computer can do cryptographic operations on behalf of the human, but it has to be designed to somehow acquire a credential from the human to prove that the computer is operating on behalf of that human.

9.6 PASSWORDS AS CRYPTOGRAPHIC KEYS

Cryptographic keys, especially those for public key cryptography, are specially chosen very large numbers. A normal person would not be able to remember such a quantity. But a person can remember a password. It is possible to convert a text string memorizable by a human into a cryptographic key. For instance, to form an AES-128 key, a possible transformation of the user's password is to do a cryptographic hash of the password and take 128 bits of the result. It is much trickier (and computationally more expensive) to convert a password into something like an RSA private key, since an RSA key has to be a carefully constructed number. In theory, the password could be used as seed for a pseudorandom number generator that will deterministically compute an RSA key pair from that seed.

Schemes based on this idea (direct conversion of the user's password to a public key pair) are not used because they perform poorly and because knowledge of the public key alone gives enough

information for an off-line password-guessing attack. Although conversion to an AES key can be computationally inexpensive, it is also vulnerable to password guessing, because human-memorizable and typeable passwords do not have sufficient entropy. An approach to slow down password-guessing attacks is to purposely design the password-to-key conversion to be expensive, such as by hashing the password 10000 times.

Another alternative is to store the cryptographic key encrypted with a function of the user's password rather than trying to directly derive the key from the password. To retrieve the key requires the human to remember their password and have access to the encrypted key. It also means that anyone with access to the encrypted key can do an off-line password-guessing attack.

There are clever schemes for leveraging a password (a weak secret) to establish a high-quality key between the user's machine and the server (see §9.17 *Strong Password Protocols*).

9.7 ON-LINE PASSWORD GUESSING

Sorry, but your password must contain an uppercase letter, a number, a haiku, a gang sign, a hieroglyph, and the blood of a virgin.

—anonymous

Trudy can impersonate you if she can guess your password. And that might not be hard. On some systems, passwords are administratively set to a fixed attribute of a person, such as their birthday or their badge number. This makes passwords easy to remember and distribute, but it also makes them easy to guess. I₁ once worked on a system where students' passwords were administratively set to their first and last initials. Faculty accounts were considered more sensitive, so they were protected with *three* initials.

Many systems use common attributes of a person to authenticate them, such as their birthdate or social security number. When many systems use the same information about a person to authenticate them, this information is no longer secret.

A popular mechanism today, usually used as backup authentication to allow someone to reset their forgotten password, is to ask a human to choose answers to security questions, from an extremely small list, usually consisting of questions like "What is your favorite ice cream flavor?" The human might sometimes like vanilla and sometimes mint chip, so this is difficult for the actual user to remember but easy for someone to guess given the limited set of choices. This is an actual set of questions that a system was forcing me₂ to choose from:

- *Favorite sports team?* (What's a "sport")?
- *Name of your veterinarian?* (I₂ don't have a pet.)

- *Name of your second grade teacher?* (I₂ couldn't even remember my second grade teacher's name when I₂ was in second grade!)
- *Your middle name?* (Aha! I₂ do have a middle name! It's Joy. So I typed "Joy", thrilled that they gave me₂ a question I₂ could answer. The system replied *Not enough letters*.)

Even if passwords are chosen so they are not obvious, they may be guessable if the impostor gets enough guesses. In fact, given enough guesses, any password, no matter how carefully chosen, can be guessed (by enumerating all finite character sequences until you hit on the correct password). Whether this is feasible depends on how many guesses it takes and how rapidly passwords can be tested.

In some military uses of passwords, guessing is not a problem. You show up at the door. You utter a word. If it's the right word, they let you in; if it's the wrong word, they shoot you. Even if you know the password is the month in which the general was born (so there are 12 possibilities), guessing is not an attractive pursuit.

Most machines do not have an "execute user" function controllable from the remote end (as useful as that might be), so this mechanism for limiting password guessing is not available. Given that people's fingers slip or they forget that they changed their password, it's important that they get more than one chance anyway. There are ways to limit the number or rate of guesses. The first is to design the system so that guesses have to actually be typed by a human. Computers are much faster and more patient than people at making guesses, so the threat is much greater if the impostor can get a computer to do the guessing.

One seemingly attractive mechanism for limiting password guessing is to keep track of the number of consecutive incorrect password guesses for an account. When the number exceeds a threshold, say five, "lock" the account and refuse access, even with a correct password, until the system is administratively reset. This technique is used with PINs on ATM cards—three wrong guesses and the machine eats your card. You have to show up at the bank with ID to get it back. An important downside of this approach is that a vandal (someone who simply wants to annoy people) can guess five bad passwords and lock out a user.

Another approach to slow down a guesser is to lock the account for some time after a few bad password guesses.

Speaking on behalf of humans, I₂ think repeated guessing of the same wrong password should not count against me₂. An imposter won't guess the same wrong password repeatedly, but the actual user might, because she isn't sure whether she might have mistyped her password. See Homework Problem 7.

The expected time to successfully guess a password is the expected number an impostor has to guess before getting it right divided by the guess rate. So far, we've concentrated on limiting the rate of password guesses. Another promising approach is to ensure that the number of passwords an attacker would need to search is large enough to be secure against an off-line, unaudited search with

a lot of CPU power. Sometimes systems assign randomly generated character strings for a user’s password. These would not be vulnerable to password guessing, but they have another problem:

Users hate them...and forget them...and write them down. Sometimes the random password generator is clever enough to generate pronounceable strings, which makes it a little easier for the human to remember the password. Constraining the generated passwords to pronounceable strings of the same length limits the number of possible passwords by at least an order of magnitude, but since a 10-character pronounceable string is probably easier to remember than an 8-character completely random string and is about as secure, generating pronounceable strings is a good idea if the administrator wants to impose passwords.

Another approach is to let users choose their own passwords but to warn them to choose “good” ones and enforce that choice where possible. One strategy is to use a “pass-phrase” with intentional misspelling or punctuation and odd capitalization, like *GoneFi\$hing* or *MyPassworD-isTuff!*, or the first letter of each word of a phrase, like *Mhall;lfwwas* (Mary had a little lamb; Its fleece was white as snow). Most systems today require a password to contain upper- and lower-case letters, numbers, and special characters. However, when most users take the password they’d like to use (“password”) and change it to “Password1!” to meet the requirements, these rules do not ensure hard-to-guess passwords. There is great wisdom to be found in *xkcd* cartoons, *e.g.*, see xkcd.com/936.

The program that lets users set passwords typically checks for easy-to-guess passwords and disallows them. There are resources that have compiled dictionaries of user passwords. A large dictionary might contain 500000 potential passwords, and it is not difficult for a computer to check that many.

Many systems, in a misguided attempt to make things more secure, actually impose rules that make things less usable and less secure. Every system administrator thinks their system is the only one that a user needs to access, but, in fact, users need passwords at zillions of different systems. Users are told not to use the same password or similar passwords at multiple systems, to change the password at each system every few months (see §9.10), and to never reuse a password. Systems have long memories and will catch you if you try to reuse a password. Servers enforce rules that really make Alice think she is living in a Kafka novel. If she forgets her password and goes through an arduous password-resetting ritual to convince the system she is really Alice, the system will allow her to reset her password but won’t allow her to set it to the password that she momentarily had forgotten. That password would have been considered secure if she had remembered it, but now she’s not allowed to ever use it again, perhaps to punish her for having momentarily forgotten it. To increase the absurdity, every system has different rules for passwords (*e.g.*, must contain special characters, must only contain English letters, must be at least n characters long, must not be longer than m characters). Alice might remember the password she chose if the system would tell her what its password rules are if she asks, but it won’t tell Alice the rules until she needs to set a new password. There was even a system I_1 was using that wouldn’t ever state its password rules. When I_1 attempted to create a password, it said “Unacceptable password. Choose something else.” After

several attempts with what I₁ considered high-quality passwords, I₁ finally in frustration tried setting my_I password to “something else”, and it was happy with that. I₁ assume all users of that system wound up with the password “something else”.

9.8 OFF-LINE PASSWORD GUESSING

In the previous section, we discussed on-line password guessing, where guessing can be slowed down and audited. Passwords do not have to be very strong if the only threat is on-line password guessing.

But sometimes it is possible for an attacker, through either eavesdropping or stealing a database, to obtain a quantity such as a cryptographic hash of the password. In such a case, even though the attacker cannot reverse the hash of the password, the attacker can guess a password, perform the same hash, and compare it with the stolen quantity. The attacker can perform password guessing without anyone knowing it and at a speed limited only by procurable compute power. This attack is known as a **dictionary attack**, or an **off-line password-guessing attack**.

One way to slow down this type of attacker is to make it computationally expensive to compute each hash, for example, by having the stored hash consist of the password hashed 10 000 times. The extra computational burden will not be noticeable to a user when she logs in, but this will be 10 000 times as much work for the attacker to create the hash of all the passwords in his dictionary. Unfortunately, it also makes the work for the server 10 000 times as much when it needs to authenticate users, which might be a real burden if users log on frequently or someone is mounting an on-line password-guessing attack (see Homework Problem 10).

When disclosure of whole files full of hashed passwords is a concern, a useful technique is to apply *salt*. Rather than guessing passwords against a single user account, an attacker who has stolen a file full of hashed passwords might hash all the words in a dictionary and check to see whether any of the passwords match any of the stored hashed values. This can be prevented as follows: When a user chooses a password, the system chooses a number unique to that user (the **salt**). It need not be secret—just unique, so badge number or user ID would be fine. It then stores both the salt and a hash of the combination of the salt and the password. When the user supplies a password

user ID	salt value	password hash
Mary	2758	hash(2758 password _{Mary})
John	886d	hash(886d password _{John})
Ed	5182	hash(5182 password _{Ed})
Jane	1763	hash(1763 password _{Jane})

during authentication, the system computes the hash of the combination of the stored salt and the supplied password and checks the computed hash against the stored hash. The presence of the salt does not make it any harder to guess any single user's password, but it makes it impossible to perform a single cryptographic hash operation and see whether a password is valid for any of a group of users.

An attacker can use a time/space trade-off by pre-computing a huge table of hashes and the corresponding passwords. These tables are called **rainbow tables**. They make password guessing cheap when hashed passwords are discovered. Use of salt also prevents the creation of rainbow tables.

9.9 USING THE SAME PASSWORD IN MULTIPLE PLACES

One of the tough trade-offs is whether to recommend that users use the same password in multiple places or keep their passwords different for different systems. All things being equal, use of different passwords is more secure because if one password is compromised, it only gives away the user's rights on a single system. Things are rarely equal, however. When weighed against the likelihood that users will resort to writing passwords down if they need to remember more than one, the trade-off is less clear.

An issue that weighs in favor of different passwords is that of a cascaded break-in. An attacker that breaks in to one system may succeed in reading the password database. If users have different passwords on different systems, this information will be of no use except on that one system. But if users reuse passwords, a break-in to a system that was not well-protected because it contained no "important" information might, in fact, leak passwords that are useful on critical systems.

9.10 REQUIRING FREQUENT PASSWORD CHANGES

Security is a wet blanket.

—apologies to Charles Schulz

A technique that is intended to enhance security, but winds up making things so much less usable that it decreases security, is requiring frequent password changes. The idea behind frequent password changes is that if someone does learn your password, it will only be useful until it next changes. This protection may not be worth much if a lot of damage can be done in a short time.

The problem with requiring frequent password changes is that users are more likely to write passwords down and less likely to give much thought or creativity to choosing them. The result is observable and guessable passwords. It's also true that users tend to circumvent password change policy unless enforcement is clever. In a spy-versus-spy escalation, the following is a common scenario:

1. The system administrator decides that passwords must be changed every 90 days, and the system enforces this.
2. The user types the change password command, resetting the password to the same thing as before.
3. The system administrator discovers users are doing this and modifies the system so that it makes sure the password, when changed, is set to a different value.
4. The user then does a change password procedure that sets the password to something new, and then immediately sets it back to the old, familiar value.
5. The system administrator then has the system keep track of the previous n password values and does not allow the password to be set to any of them.
6. The user then does a change password procedure that goes through n different password values and finally, on the $n+1^{\text{st}}$ password change, returns the password to its old familiar value.
7. The system is modified to keep track of the last time the password was changed and does not allow another password change for some number of days.
8. The user, when forced to change passwords, constructs a new password by appending 1 to the end of the previous password. Next time the user replaces the 1 with a 2, and so on.
9. The system, in looking for guessable passwords, looks for passwords that look “too much like” one of the previous n passwords.
10. The users throw up their hands in disgust, accept impossible-to-remember passwords, and post them on their terminals.

9.11 TRICKING USERS INTO DIVULGING PASSWORDS

A threat as old as timesharing is to leave a program running on a public terminal that displays a login prompt. An unsuspecting user then enters a user name and password. The Trojan horse program logs the name and password to a file before the program terminates in some way designed to minimize suspicion.

Today, popup boxes that can be posted by any program or web page ask the user to type credentials. An example is a common email program. At unpredictable (to the user) times, such as a server somewhere going down, it displays a popup box asking the user to enter their password. The user did not do anything that would cause them to expect such a message. Since users want to receive their email, they will dutifully tell whatever displays such a popup box their password. They have become used to having these boxes pop up at any time, and there is no way for a user to distinguish an honest popup (such as the one from their email program) from a malicious one.

Another example is maliciously sent email (phishing) where the sender hopes to trick the user into divulging her credentials. The email might claim to be from the user's bank, asking the user to click on a link to verify a recent transaction or something. The link will take the user to a page that looks exactly like the login page of the bank, and they will type their username and password into a malicious site that now knows the user's credentials and can access the user's account at the real bank.

9.12 LAMPORT'S HASH

It's a poor sort of memory that only works backwards.

—The White Queen (in *Through the Looking Glass*)

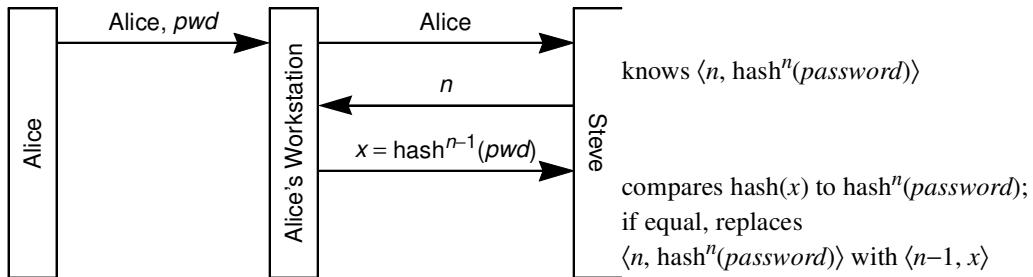
Leslie Lamport invented an interesting one-time password scheme [LAMP81]. Although not widely used, it is a very elegant cryptographic trick. The scheme allows server Steve to authenticate Alice in a way that neither eavesdropping on an authentication exchange nor reading Steve's database enables someone to impersonate Alice, and it does it without using public key cryptography. (See Protocol 9-2.) Alice (a human) remembers a password. Steve (the server that will authenticate Alice) has a database where it stores, for each user:

- username
- n , an integer that decrements each time Steve authenticates the user
- $\text{hash}^n(\text{password})$, i.e., $\text{hash}(\text{hash}(\dots(\text{hash}(\text{password}))\dots))$

First, how is the password database entry associated with Alice configured? Alice chooses a password, and a reasonably large number n (like 1000) is chosen. The user registration software computes $x_1 = \text{hash}(\text{password})$. Then it computes $x_2 = \text{hash}(x_1)$. It continues this process n times, resulting in $x_n = \text{hash}^n(\text{password})$, which it sends to Steve, along with n .

When Alice wishes to prove her identity to Steve, she types her name and password to her workstation. The workstation then sends Alice's name to Steve, which sends back n . Then the workstation computes $\text{hash}^{n-1}(\text{password})$ and sends the result to Steve. Steve takes the received

quantity, hashes it once, and compares it with its database. If it matches, Steve considers the response valid, replaces the stored quantity with the received quantity, and replaces n by $n-1$.



Protocol 9-2. Lamport's Hash

If n ever gets to 1, then Alice needs to set her password again with Steve. Alice can choose a new password, compute $\text{hash}^n(\text{new password})$, and transmit $\text{hash}^n(\text{new password})$ and n to Steve.

An enhancement is to use salt (see §9.8). Instead of Alice's machine computing $\text{hash}^n(\text{password})$ when she sets her password, it could use her username (presumably a unique value) as salt, and compute $\text{hash}^n(\text{password} \mid \text{Alice})$. If she would like to securely use the same password on multiple servers, she could include the name of the server in the computation, *i.e.*, compute $\text{hash}^n(\text{password} \mid \text{Alice} \mid \text{Steve})$. See Homework Problem 5.

Lamport's hash has interesting properties. It is similar to public key schemes in that the database at Steve is not security sensitive (for reading—other than dictionary attacks to recover the user's password). It has several disadvantages relative to public key schemes. One problem is that you can only log in a finite number of times before having to reinstall password information at the server.

Another problem is there is no mutual authentication, *i.e.*, Alice does not know she is definitely talking to Steve unless she is talking over TLS.

There's another security weakness that we'll call the **small- n attack**. Suppose an intruder, Trudy, were to impersonate Steve's network address and wait for Alice to attempt to log in. When Alice attempts to log into Steve (but she's really talking to Trudy), Trudy sends back a small value for n , say, 50. When Alice responds with $\text{hash}^{50}(\text{password})$, Trudy will have enough information to impersonate Alice for some time, assuming that the actual n at Steve is greater than 50. What can be done to protect against this? Alice's workstation could display n to the human Alice and give her a chance to object. If Alice remembers that n was approximately 850 the last time she connected to Steve, then when Trudy is impersonating Steve, the human Alice might be suspicious to see a much smaller value of n displayed.

Lamport's hash can also be used in environments where the workstation doesn't calculate the hash, for example, when:

- Alice is logging in from a workstation that does not have Lamport hash code, or
- Alice is logging in from a workstation that she doesn't trust enough to tell her password

We'll call this the **human and paper** environment and call the other environment the **workstation** environment. The way Lamport's hash works in the human and paper environment is that when the information $\langle n, \text{hash}^n(\text{password}) \rangle$ is installed at the server, all the values of $\text{hash}^i(\text{password})$ for $i < n$ are computed, encoded into a typeable string, printed on a paper, and given to Alice. When she logs in, she uses the string at the top of the page and then crosses that value out, using the next value the next time. This approach automatically protects against the small n attack. Of course, losing the piece of paper, especially if it falls into the wrong hands, is a problem.

It is interesting that the human and paper environment is not vulnerable to the small n attack, since the human just always uses the next value on the list and can't be tricked into sending an item further down on the list.

There is a deployed version of Lamport's hash, known as S/Key, implemented by Phil Karn. It was standardized in RFC 1938. It operates in both the workstation and human and paper environments. It makes no effort to address the small n attack, but it certainly is an improvement over cleartext passwords.

9.13 PASSWORD MANAGERS

Security administrators have a dream. Users are supposed to choose long unguessable passwords at every site. The passwords should not be similar to each other and they should be changed periodically. Each site has its own rules about how long passwords should be and what characters they must or must not contain. Unfortunately, this dream is a user's nightmare.

So various technologies have been developed to help users cope. A user could personally maintain a file containing her username and password for each site. This is, of course, dangerous if anyone were to steal this file. To make it less vulnerable, the user could encrypt the file with a very good password. Users can remember and type one really good password, provided it is only one. The user would want to surf the Internet from various devices, so she'd have to move the file between her devices. And if she created a username/password at a new service that needed to be added to the file, she'd have to make sure this entry was included in all the copies of the file. This is sufficiently cumbersome that users seldom do this.

Instead, there are various password manager services, usually implemented in a user's browser. We are not describing specific implementations, but rather conceptual issues.

If user Alice agrees, the browser she is using might remember her username/password at websites on which Alice has accounts. When Alice is asked to log into a service, the browser might

offer to autofill her username and password on the site. This is not only convenient, but has the security advantage that the password manager would not be tricked into sending Alice's username and password at website X to a phishing site with a name that looks similar to a human.

The browser's database of Alice's username/passwords might be local to that one machine, or the browser might back it up in the cloud. If Alice explicitly logs into that browser on a different device, the browser can download Alice's username/password list from the cloud. Sometimes Alice might not be aware that she has logged into the browser, because the same company that provides the browser might also provide email service, and when Alice logs into her email, she will also be logging into the browser.

The disadvantage of a password manager is that a single password, if guessed or divulged, reveals all the user's passwords to the attacker. Likewise, if the user's device is infected with malware, or if the user walks away from their device while it is unlocked, all the user's passwords will be compromised. Note that password managers often offer to choose a completely unguessable password for each site, but all these incredibly secure passwords are locked by a single password that the user has to be able to remember and type.

9.14 WEB COOKIES

When user Alice is browsing the web, she is using a protocol known as HTTP. HTTP is stateless, which means that it treats each HTTP request independently from any other HTTP request. The two main HTTP request types are **GET** and **POST**. The best way to think of them is that **GET** is for reading a web page and **POST** is for sending information to a web server. The response contains information such as the content requested and status information (such as *OK* or *not found* or *unauthorized*). One status that might be included in a response is a redirect. This informs the browser that it should go to a different URL. If redirected, the browser will then go to the new URL as if the user had clicked on a link.

Note **URL** stands for **Uniform Resource Locator** and is used by the HTTP protocol to find the resource (the URL contains a DNS name that can be looked up to find an IP address, and additional information interpreted by the resource, which helps it do things like find the specific web page being requested).

A website can include a string of octets in the HTTP response that is known as a **cookie**. The browser need not interpret the cookie in any way, but it must store it along with the DNS name of the site that gave it that cookie. When the browser sends an HTTP request to a DNS name (inside a URL), it sends any cookies it has for that DNS name along with the HTTP request.

This enables a user to have the illusion of a long-lived connection with service Steve, so Alice only needs to log in to Steve occasionally. Once she logs into Steve, Steve sends a cookie, and on

each subsequent HTTP request to Steve, the browser will include that cookie, indicating to Steve that this is Alice. Steve can have policies about how long Alice's browser should keep the cookie. If the cookie expires (Alice's browser deletes the cookie, or Steve includes a timestamp in the cookie and rejects cookies that are too old), then when Alice's browser makes a request to Steve, Steve will redirect her to a login page.

9.15 IDENTITY PROVIDERS (IDPS)

An alternative way of making authentication convenient for the user Alice is to have Alice link her account at a site known as an **identity provider** with her account at other websites. What this looks like to Alice is that when she attempts to log in to Steve, Steve might give her the choice of logging in with username/password or with one of several companies that offer identity provider service, such as Google or Facebook.

When Alice creates an account at Steve, she can choose to create a username and password, or she might choose to authenticate with IdentityProvider4. If she chooses IdentityProvider4, then Steve's webpage will send her to IdentityProvider4. If she has logged into IdentityProvider4 before, her browser will have stored a cookie from IdentityProvider4 and will send that cookie to IdentityProvider4. If her browser does not have a cookie from IdentityProvider4, she will be directed to a login page at IdentityProvider4. IdentityProvider4 will now know which IdentityProvider4 user Alice is and that she was sent there from Steve. IdentityProvider4 will ask Alice if she wants to log into Steve. If Alice says yes, then IdentityProvider4 will send her back to Steve, this time with information tacked onto the end of the URL directing her to Steve, which is a signed statement from IdentityProvider4 saying that this user is Alice at IdentityProvider4.

This is convenient for the user, since she need only remember how to log into IdentityProvider4. It does have privacy implications, in that IdentityProvider4 will know all the websites that Alice has visited using IdentityProvider4 as the identity provider. And it will be easy for websites to notice that the user that website X knows as *Alice at IdentityProvider4* is the same user that Y knows as *Alice at IdentityProvider4*. It also means that if anyone were to break into IdentityProvider4, many users would be compromised at many different websites.

Note that if Alice has accounts at several identity providers, she needs to remember which identity provider she used when creating an account at Steve. A different identity provider would know Alice by a different name, and Steve would not be able to recognize Alice's account.

The protocols enabling identity providers were designed to fit with mechanisms already in browsers (HTTP, cookies, redirects). There are various standards, including OpenID Connect and SAML. **OpenID Connect** is an authentication protocol that is layered on top of OAuth 2.0. **SAML**

has pretty much the same functionality as OpenID/OAuth but with different syntax (SAML uses XML) and defined by a different standards organization.

9.16 AUTHENTICATION TOKENS

An authentication token is usually a physical thing that a person carries around and uses in authenticating. NIST refers to these devices as *authenticators*. In the breakdown of security according to *what you know*, *what you have*, and *what you are*, authentication tokens fall in the *what-you-have* category. Although we will describe various broad categories, many products do not fit nicely into a single category. And again, we are not describing specific products. We are discussing the conceptual issues and trade-offs and potential features of such tokens. We will refer to the machine from which Alice is connecting to the Internet as her *device*, with example devices being a laptop, a tablet, or a phone. We will use the term *token* to refer to either a physical hardware widget used for authentication or a software implementation of it.

9.16.1 Disconnected Tokens

The older type of token (though some are still in use) is a little thing that Alice carries around that cannot directly communicate with Alice's device. Alice has to act as the interface between the token and her device.

Alice's token displays a string of perhaps eight numbers, and Alice will type the value displayed on her token into her device. Each token has an internal secret configured into the token before it is given to Alice. There is a server somewhere on the Internet that manages a particular set of tokens (*e.g.*, all the ones from a particular company). Let's call that the token server. The token server has a database consisting of a set of $\langle \text{userID}, \text{secret} \rangle$ pairs. The userID is an ID for the human Alice, and the secret is the secret configured into Alice's token. When Alice logs into a site, say, Steve, that wants Alice to use the token as one of the factors of authentication, she will input the value displayed on the token, and Steve will query the server about whether Alice's token value is correct. There are at least two designs of these tokens.

One type is **time-based**. The value displayed by the token is a hash of the time of day (usually in minutes) hashed with the token's secret. Sometimes the user types a PIN into the token, and the input into the hash includes the PIN as well as the current time in minutes and the token secret. If the user mistypes the PIN, the only feedback she will get is that authentication will not work (because the token is displaying the wrong value). But authentication might not have worked because she might have mistyped the 8-digit number displayed on the token, so Alice has to decide

whether she should retype the PIN into the token. An interesting challenge with time-based tokens is that the clocks can drift. Typically, a solution to this is that if the value from the token seems to be more than a minute too old or new, the token server asks the user to input both what the token is currently displaying and what the next token value is. The token server computes and stores the time adjustment for Alice's token so that it can again be in sync.

Another type is **challenge-response**. The token has a little keyboard, like a calculator. When using the token to authenticate, Steve (the service Alice is authenticating to) sends a number (perhaps four digits) to Alice's workstation. Alice types that value into her token, and the token computes a function of the challenge and the token's secret, displaying the result. Alice types the result into her workstation. This form of token is less convenient for Alice because she not only has to type the value from the token into her device, but she has to type the challenge displayed on her device into the token. It is also more expensive than the time-based one, because it needs to include a keyboard. It is less secure, too. If an eavesdropper Eve were to observe a few hundred challenge/responses from Alice, Eve might be lucky enough when attempting to impersonate Alice to get a challenge from Steve that Eve knows the answer to. And if Eve doesn't know the answer to a particular challenge, she can just try logging into Steve again to get a new challenge. If Eve gave a wrong response to the challenge, Alice's account might get locked after some number of attempts, but Steve does not typically count a non-response as a wrong response. The reason these tokens were deployed was to avoid patent issues, but since relevant patents have expired, they are rarely used anymore.

It is inconvenient for Alice to have to read a display from the token and type the result into her device. Users also frequently misplace the tokens. Although these types of tokens were originally physical little things, they are now commonly implemented as software on the user's device. If Alice is communicating to service Steve using the same device on which the software token is running, she can cut and paste the displayed value into the login page presented by service Steve.

Having a software implementation is less expensive to implement than a physical token. If Alice is only accessing Steve from one device, this is very convenient. Unfortunately, Alice is likely to want to access service Steve from multiple devices (*e.g.*, laptop, tablet, phone), and for some reason, providers of this software do not want to allow Alice to have token software running on more than one of her devices. If the token software is running on her laptop and she wants to browse using her phone, the laptop becomes an extremely cumbersome token that Alice will need in order to access Steve from her phone.

A security implication of these secret-based tokens is that there needs to be a server on the Internet (the token server) that knows the secrets for all the tokens. If this database is stolen, then all tokens are compromised.

9.16.2 Public Key Tokens

Public key tokens need to be connected to Alice's device, because public key signatures are too long for a human to type. The tokens connect to a device using technologies such as Bluetooth, NFC, or USB. Given that they are based on public keys, there is no need for an on-line server that knows the secrets for all the tokens.

The token might be multifactor, in that Alice might need to activate it by inputting a password, scanning her finger, or merely pressing a button on the token to give it permission to authenticate on her behalf. Sometimes the physical token can be implemented in software on Alice's device. This is convenient, but it has the security issue that malware on Alice's machine might be able to steal the private key(s).

Whether the token is a physical token or software, malware does not need to steal the private key(s) from the token in order to cause security issues. If malware can ask the token to authenticate on Alice's behalf, the malware doesn't need to know the token's secrets. Therefore, there is usually some sort of feedback to Alice when the device is acting on her behalf.

The most secure option would be for the token to have a display and display "Request to log in to BigBank?" Alice can then hit a button on the token for yes or no. Most implementations have Alice's device (rather than the token) display what the token is being asked to do on Alice's behalf, even if the permission button is on the token. The problem is that malware on Alice's device can ask Alice "Did you ask to log in to weather.com?" when in fact the malware will ask the token to authenticate to BigBank. Although it would be more secure to have a display on the token, this would make the token larger and more expensive.

The FIDO alliance (fidoalliance.org) has developed standards for communication between a browser or OS and such a device. The W3C organization has developed standards for how the credentials are used for web authentication. The standards include many variations. There are also public key authentication tokens that do not fit with the FIDO specifications. We will not describe the specific standards, but instead discuss some of the features such devices might have.

The token could, in theory, have a single private key for Alice, and she could use that to authenticate to all sites that support authenticating using the token. Some tokens, such as credit cards with embedded computer chips, U.S. government CACs (Common Access Cards), or PIV (Personal Identity Verification) cards, do not have different key pairs for each site that Alice will connect to. However, the PIV and CAC cards have several PIN-protected private keys, including one for authentication, one for signing documents, and one for decrypting documents. There is also a non-PIN protected private key for local authentication, say, for opening a door.

For consumer use, it is assumed that Alice will want to have a different private key for each service to which she will authenticate, so that it won't be easy to know which accounts at site A and site B are the same user (though people seem to think that using identity providers is not a privacy issue).

With the consumer type of public key token, when Alice creates an account at a site Steve that supports authentication using the token, Alice's token creates a new public/private key pair, and sends a **key handle** and public key to Steve. Later, when she needs to authenticate to Steve, she will tell Steve her username, and Steve will send the key handle associated with Alice's account to Alice's workstation, which will pass this information to the token. The token can then perform authentication using that private key.

A cute feature of the FIDO design is the option to have very little storage on the token, so that the token does not remember all the private keys it has created for all the sites. Instead, the token just needs to remember a single secret key S . The key handle the token sends to Steve can be the newly created private key for Alice's token, encrypted with S . When Alice logs into Steve, Steve sends the key handle, and the token can retrieve Alice's private key by decrypting the key handle.

Another issue is that Alice might purchase a counterfeit, possibly malicious token. How would she possibly know? The proposed solution is for a legitimate manufacturer (say M) to install an attestation key into each token, along with a certificate saying "This device, model number X, was manufactured by M." The characteristics of that model from that manufacturer (*e.g.*, how tamper-resistant the token is, or whether the token requires a fingerprint from Alice to activate it) can be looked up somewhere. Alice might be able to detect that the token is counterfeit because her device has the ability to query the token as to its heritage, or perhaps when Alice is enrolling at website Steve, Steve might query the token. Steve might only allow Alice to do certain operations if her token is extra-securely manufactured. Or Steve might warn Alice "I think your token is counterfeit" or "That token does not meet my standards or is not manufactured by one of the organizations I trust."

If every token had its own unique attestation key, then various sites would be able to figure out that Alice's account at one site was the same human (using the same token) as an account at another site. To prevent linking accounts this way, large batches of tokens can be configured with the same attestation key.

Another interesting issue is preventing evil Trudy from acting as a meddler-in-the-middle (MITM). Suppose Trudy can trick Alice into thinking that Trudy is BigBank.com. Perhaps Trudy has acquired a DNS name that looks similar to a human, like BiqBank.com (the letter q looks like the letter g if Alice doesn't look carefully). Alice connects to Trudy (intending to connect to BigBank). Suppose Trudy wants to impersonate Alice at BigBank. When Alice connects to Trudy, Trudy connects to BigBank, and says "I'm Alice." BigBank sends Trudy a challenge and key handle for Alice's token. Trudy forwards the challenge and key handle to Alice. Alice's token signs the challenge, and Trudy can then use it to complete the authentication as Alice to BigBank.

There are various ways of solving the MITM attack. One is for Alice's device to inform the token of the DNS name that the device thinks it is communicating with. BiqBank might look like BigBank to Alice, but it is a different string. If Alice's token signs a hash of the challenge and the DNS name BiqBank, then Trudy won't be able to forward this to BigBank, because the response will not be correct (it will be a hash containing the incorrect DNS name). Another trick is to have

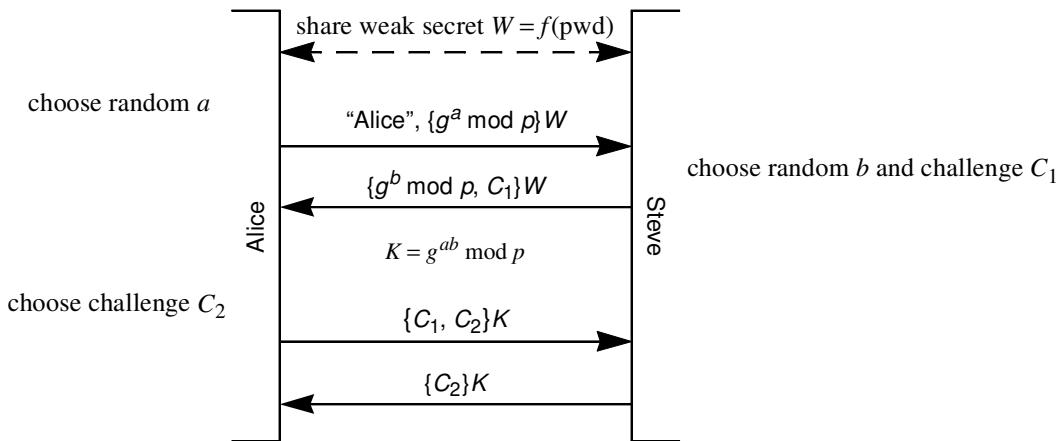
the endpoints of a secure connection (such as TLS or IPsec) sign a hash of the secret key established for the secure session. If Trudy were acting as a MITM, she would establish a different key between herself and BigBank than the key she established between herself and Alice's token. Trudy would not be able to trick Alice's token into signing the key between Trudy and BigBank. If the protocol enables Trudy to force the secret key to be a particular value, this technique would not detect her. Having the endpoints sign a transcript of all the messages so far is likely to be more secure. Additional issues:

- What can be done if Alice loses her token, or the token breaks after Alice leaves it in her pocket and sends it through the laundry? There should be some way for Alice to recover, by purchasing another token, preferably without having to re-enroll at all the Internet sites she has been using. This might be solved by having her token back up all the private keys it has created, but if there is an interface for reading the private keys from the token, this will make the token less secure.
- Can Alice have multiple tokens (because she is nervous she might lose one, or because she keeps one token plugged into her laptop USB port and the other taped to her phone)? Can protocols manage to allow her to use her multiple tokens interchangeably at all her sites?

9.17 STRONG PASSWORD PROTOCOLS

Strong password protocols are designed so that, even though Alice is authenticating to a remote resource using a password and not using something like TLS, someone who eavesdrops on an authentication exchange or impersonates either end will not obtain enough information to do off-line verification of password guesses. An eavesdropper should not be able to gain any information from observing any number of legitimate exchanges. Someone impersonating either endpoint will be able to do a single on-line password guess. There's really no way to avoid that. If someone correctly guesses the password, they will be able to successfully authenticate. If they guess incorrectly, they will know that they have not successfully authenticated, and therefore their guess must not be the user's password. A false guess will result in an authentication failure, which should generate alarms if they occur in large numbers.

These are sometimes referred to as **PAKE** (password-authenticated key exchange). The first such protocol, named **EKE** (for **Encrypted Key Exchange**), was published by Bellovin and Merritt [BELL92a]. (See Protocol 9-3.) There are other protocols that are conceptually similar. The idea of EKE is that Alice and server Steve share a weak secret W that is a hash of Alice's password. Steve knows W because at some point Alice set her password at Steve, and Steve stores $(Alice, W)$ in a database. The user Alice knows her password. The device from which Alice is authenticating to Steve learns W because it computes it based on the password Alice typed.



Protocol 9-3. Basic EKE Protocol

The two devices do a Diffie-Hellman exchange, encrypting the Diffie-Hellman numbers with W , and then do mutual authentication based on the agreed-upon Diffie-Hellman shared secret, which is a strong secret. After the authentication, they can use the agreed-upon Diffie-Hellman value K to encrypt the remainder of their conversation.

This protocol is quite subtle. The reason it is secure from an eavesdropper is that a Diffie-Hellman transmitted value looks like a random number. An eavesdropper doing a trial decryption of $\{g^a \text{ mod } p\}W$ and $\{g^b \text{ mod } p\}W$ cannot verify a password guess because decrypting with any password will still just look like random numbers. And someone impersonating one side or the other can verify a single password guess as incorrect or correct, but this is an on-line, auditable guess. There is no way to do an off-line dictionary attack. The Diffie-Hellman secret K is a strong secret because an attacker would both have to guess the password and break Diffie-Hellman.

The reason it is secure from someone, say, Trudy, impersonating either Alice or Steve is that Trudy only knows x for one value of $g^x \text{ mod } p$. Once Trudy encrypts with W , she is committing to a single password guess. (See Homework Problem 16.)

Several years later, more or less simultaneously, two other strong password protocols were invented. One was done by Jablon and was called SPEKE (*simple password exponential key exchange*) [JABL96], and the other was done by Wu and known as SRP (*secure remote password*) [WU98]. We'll describe SRP in section §9.17.3. Several years after SRP and SPEKE, we_{1,2} designed PDM (*password derived moduli*) [KAUF01].

SPEKE uses W (the weak secret derived from the password) in place of g in the Diffie-Hellman exchange, so rather than transmitting $\{g^a \text{ mod } p\}W$ and agreeing upon $K = g^{ab} \text{ mod } p$ (as would be done in EKE), SPEKE transmits $W^a \text{ mod } p$ and $W^b \text{ mod } p$ and agrees upon the key $K = W^{ab} \text{ mod } p$.

PDM chooses a modulus p derived deterministically from the password and uses 2 as the base, so the Diffie-Hellman numbers transmitted are $2^a \bmod p$ and $2^b \bmod p$, and the agreed-upon Diffie-Hellman key is $2^{ab} \bmod p$.

9.17.1 Subtle Details

There is more to making these schemes secure than the basic idea. The original EKE paper proposed many variants of the protocol, many of which were later found to be flawed. The successor protocols have had similar difficulties. To be secure, a protocol must carefully specify some implementation details to avoid an eavesdropper being able to eliminate password guesses. Subsequent papers noted other potential implementation issues. For instance, assume a straightforward encryption of $g^a \bmod p$ with W . Since $g^a \bmod p$ will be less than p , an eavesdropper that does a trial decryption with a guessed password and obtains a value greater than p can eliminate that password. If p were just a little more than a power of 2, an incorrect password would have almost a 50% chance of being eliminated. Each time an eavesdropper saw a value $W\{g^a \bmod p\}$ (each time presumably with a different a), the eavesdropper could eliminate almost half of the passwords. With a dictionary of, say, 50000 potential passwords, the eavesdropper would only need to see about twenty exchanges before narrowing down the possibilities in the dictionary to a single choice.

If SPEKE were not designed carefully, it would also have a flaw whereby an eavesdropper might be able to eliminate some password guesses based on seeing $W^a \bmod p$. The flaw can be eliminated by making sure that W is a perfect square mod p . Some numbers are generators mod p (g is a generator if $g^1, g^2, g^3, \dots, g^{p-1} \bmod p$ cycles through all the values from 1 through $p-1$). If g is a generator mod p , then its even powers are the perfect squares mod p (so half of all numbers mod p are perfect squares, and any power of a perfect square is also a perfect square). If some of the W s generated from passwords for use in SPEKE were perfect squares and some not, then if an eavesdropper saw a value $W^a \bmod p$ that was not a perfect square, she would know that none of the passwords that resulted in W s that were perfect squares could have been Alice's password (since such a password could not have generated a value that was not a square). (To tell if a number is a perfect square mod p , raise it to the power $(p-1)/2$ and see if the result is $1 \bmod p$.) This is a less serious vulnerability than the EKE vulnerability in the previous paragraph, because in each EKE exchange a different half of the passwords could be eliminated. But in this SPEKE vulnerability, half the passwords (the ones for which W would be a square) would be eliminated if a value $W^a \bmod p$ that was not a perfect square were seen by an eavesdropper, but there would be no further narrowing down the possibilities no matter how many exchanges were observed.

Both the vulnerabilities mentioned are easily avoided. The EKE vulnerability is avoided by choosing a p that is just a little less than a power of two. The SPEKE vulnerability is avoided by ensuring that W is a perfect square—hash the password and then square it mod p to get W .

To build a workable protocol from the basic idea of PDM (generating the modulus deterministically from the password with reasonable performance) involves some math that we won't go into in detail because it's not that important for the purpose of this chapter. (For subtle reasons, using 2 for g , the modulus p has to be a safe prime, *i.e.*, $(p-1)/2$ must also be prime, and p must be equal to $11 \bmod 24$.)

Adoption of all these protocols was slowed down by patent issues. However, the relevant patents have expired at this point.

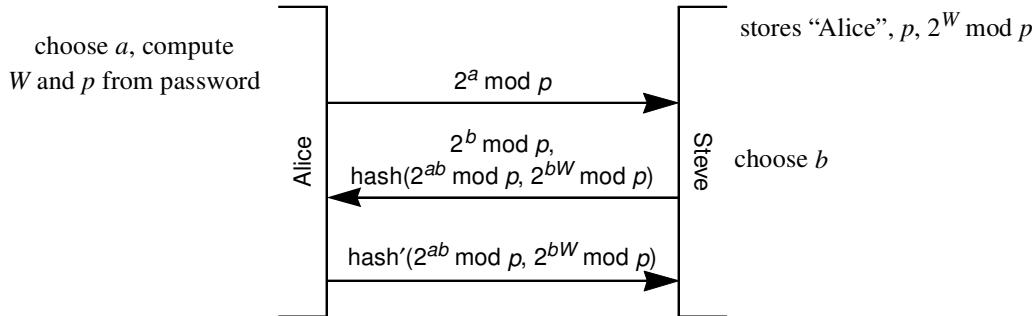
9.17.2 Augmented Strong Password Protocols

With the schemes in the previous section, if someone stole the server database and therefore knew W , they could impersonate the user. Bellovin and Merritt designed a strong password protocol with an additional security property that they called *augmented EKE* [BELL93]. The additional property prevents someone who has stolen the server database from being able to impersonate the user. The information in the server database would still allow an attacker, Trudy, to do a dictionary attack. If Trudy's dictionary attack found the user's password, then Trudy could impersonate the user. But using an augmented form of a strong password protocol, if Trudy's dictionary attack on the stolen server database was unsuccessful, she would not be able to impersonate the user to the server.

All the basic schemes (EKE, SPEKE, and PDM) can be modified to have the augmented property. Another protocol that we'll describe (SRP) only has an augmented form. The idea is for the server to store a quantity derived from the password that can be used to verify the password, but the client machine is required to know the actual password (in addition to the derived quantity stored at the server). The scheme in the published augmented EKE is a bit complicated, so we will instead show a simpler scheme with the same properties. The strategy in this simpler protocol will work with any of the schemes (EKE, SPEKE, and PDM), but we'll show it with PDM and let the reader adapt EKE and SPEKE in Homework Problem 11.

In the augmented form of PDM, the server (Steve) stores p , the safe prime derived from the user Alice's password. The server will also store $2^W \bmod p$, where W is a hash of Alice's password. The exchange is as follows (see Protocol 9-4):

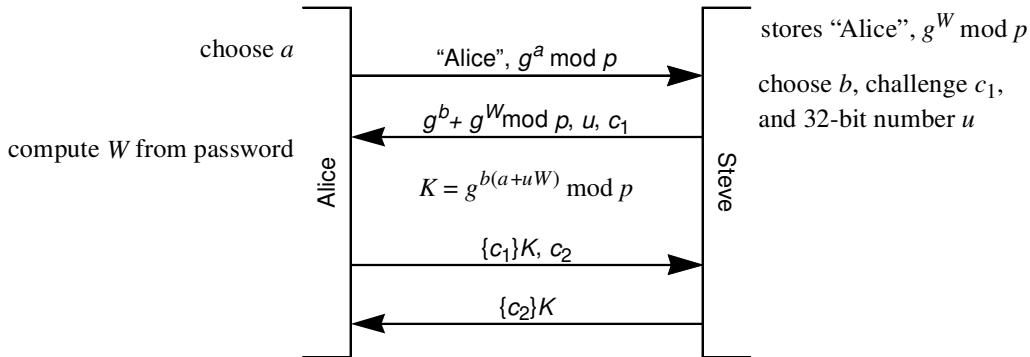
After the Diffie-Hellman exchange (using PDM's modulus p derived from Alice's password), both Alice and Steve can compute $2^{ab} \bmod p$ and $2^{bW} \bmod p$. (see Homework Problem 12). Notice that each side sends a hash of $(2^{ab} \bmod p, 2^{bW})$, so it has to be a different hash or else whoever speaks second can just repeat what they received. Having a different hash can be achieved in many ways, such as by concatenating a constant to the quantities being hashed.

**Protocol 9-4.** Augmented Form of PDM

9.17.3 SRP (Secure Remote Password)

SRP was invented by Tom Wu [WU98] and is a popular choice by the IETF for strong password protocols. It is documented in RFC 2945. It is harder to understand than the others, but we describe it here because it does appear in many IETF protocols. Unlike EKE, SPEKE, and PDM, there is no basic form of SRP. The augmented property is an intrinsic part of the protocol.

SRP is shown in Protocol 9-5. Steve stores $g^W \text{ mod } p$, where W is a function of Alice's password. Alice calculates W from the password. The tricky part is how Alice and Steve each manage to compute the session key K (see Homework Problem 14).

**Protocol 9-5.** The Secure Remote Password Protocol (SRP)

9.18 CREDENTIALS DOWNLOAD PROTOCOLS

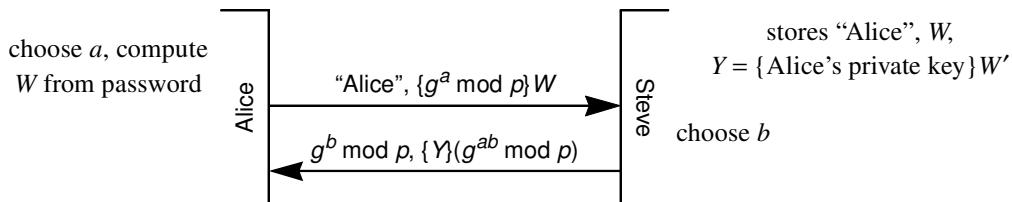
A **credential** is something that can be used to prove who you are or prove that you are authorized to do something. In this case, it is easiest to think of it as a private key. It might be nice to assume that Alice has a smart card, that she has remembered to bring her smart card to work, that it is still functional after she accidentally ran it through the wash, and that the workstation she is accessing the Internet from has a way to communicate with her smart card.

But suppose Alice does not have a smart card. All she knows is her name and password. If Alice walks up to a workstation that has trusted software but no user-specific configuration, then if she can somehow obtain her private key, she can obtain all the other information necessary to recreate her environment by downloading it from the cloud. Any information she has stored in the cloud, such as cookies or browser bookmarks that need to be kept private, can be stored in the cloud encrypted with her private key. So, all she really needs to do is securely retrieve her private key from the cloud.

We can use strong password protocols to do this. The private key is kept in the directory, encrypted with the user's password. Call that quantity Y . You don't want to make Y world-readable, since someone that has Y can test passwords against it. And you can't use traditional access control since Alice can't prove she's Alice until she obtains Y (and decrypts it with her password). Strong password protocols are ideal for downloading credentials.

For credentials download, the augmented protocols provide no added security. The only purpose of a credentials download protocol is to download Y . If someone has stolen Steve's database, then they already know Y .

[PERL99A] includes an analysis of credential download protocols, along with a two-message version that can be built upon any basic strong password protocol. Other properties are explored, such as the ability for Steve to save computation by reusing his Diffie-Hellman exponent b . In Protocol 9-6 we show a simple two-message credentials download protocol built upon EKE.



Protocol 9-6. Two-Message Credential Download Protocol

Steve cannot tell whether Alice really knew the password, but Alice can only guess one password in each on-line query, since once she encrypts by W , she is committing to a single password. She only knows the a for the quantity she encrypted with the chosen W . So Steve can audit

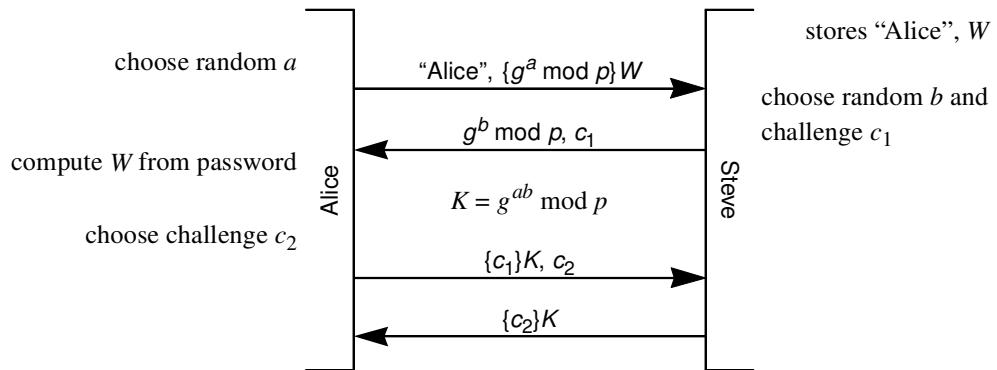
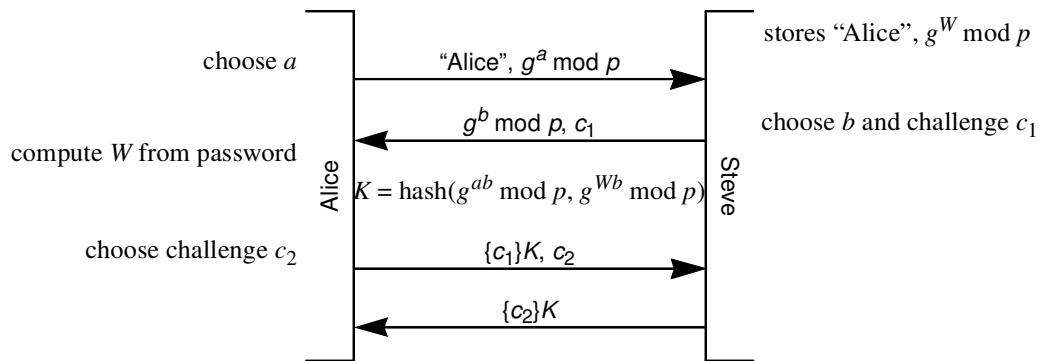
download requests and get suspicious if the credentials for the same user were requested too many times. Note that the key W' used to encrypt Alice's private key must be different from W , or else someone stealing Steve's database (not to mention Steve itself) would know Alice's private key.

9.19 HOMEWORK

1. Given that the Lamport hash (see §9.12 *Lamport's Hash*) value is sent in the clear over the network, why is it more secure than a password?
2. Is the Lamport hash protocol vulnerable to dictionary attack by an eavesdropper (assuming communication is not using an encrypted channel such as TLS)? Can someone impersonating Steve do a dictionary attack?
3. Design a variant of Lamport's hash using k times more storage at the server but needing only $1/k$ as much processing, on average, at the client.
4. Suppose we are using Lamport's hash, and Steve crashes before receiving Alice's reply. Suppose an intruder, Trudy, can eavesdrop and detect that Steve crashed (maybe Trudy can even cause Steve to crash). Then Trudy has a quantity (whatever Alice replied that Steve did not receive) that Trudy can use to impersonate Alice, if Trudy logs in before Alice attempts to log into Steve again. How can we modify Steve's behavior to prevent this threat? (Exactly when does Steve overwrite its database, and with what?)
5. Suppose in Lamport's hash, neither Alice's name nor Steve's name is part of the hashed quantity, so what is installed for Alice at Steve is $\langle n, \text{hash}^n(\text{password}) \rangle$. Suppose Alice uses the same password at server Carol (also using Lamport's hash). Suppose an eavesdropper can listen to Alice authenticating to Steve. Why might this enable the eavesdropper to impersonate Alice at Carol?
6. Suppose you want to use Lamport's hash for authentication but want to allow Alice to reset her password to the same password as before when n gets to 0. Alice might not even remember which passwords she used before. How can the Lamport hash protocol be modified so that it will be secure even if Alice reuses her password?
7. If there is a limit to the number of bad password guesses a user is allowed before she gets locked out of her account (say n), it is unreasonable to count it against her if she guesses the same wrong password multiple times (perhaps because she isn't sure whether it was the wrong password or she mistyped her password). Suppose the authentication protocol involves the user sending her password to the server. What can the server do to not count multiple guesses of the same wrong password against a user? Now suppose the authentication protocol

is a challenge/response protocol as in section §9.1.1. Since the server cannot see which password the user typed, what can the server do (without changing the authentication protocol) so that the user's account will only be locked out after n different wrong password attempts?

8. Suppose Trudy maliciously guesses n wrong passwords for Alice's account at service S, causing Alice to get locked out. This can be annoying to Alice, even though Trudy will not be able to impersonate her. What behavior could S do so that Alice will not be locked out, even though S still has a limit to the number of wrong password guesses? Hint: Consider having S use web cookies to differentiate which machine is trying to authenticate or having S remember IP addresses from which it received bad password guesses.
9. Suppose there are 1000 users. An attacker has stolen the database of hashed passwords and has a dictionary of common passwords. How much extra computation does the attacker require to find all users that have passwords in the dictionary if salt is included in the password hash (see §9.8)? Suppose the hash is slow to compute, e.g., consists of the password hashed 10000 times. How much extra computation does the attacker require to find all users that have passwords in the dictionary if both the expensive hash and salt are used?
10. In §9.8, we mention that it would be 10000 times as much work for the server if the password hash were $\text{hash}^{10000}(\text{password})$. Suppose the server still stores $\text{hash}^{10000}(\text{password})$, but when Alice logs in, her client machine first computes $\text{hash}^{9999}(\text{password})$ and sends that to the server. Will there still be the advantage gained by storing $\text{hash}^{10000}(\text{password})$? Will this save the server computation?
11. Show protocols for doing augmented forms of EKE and SPEKE.
12. Show how Alice and Steve can each compute $2^{ab} \bmod p$ and $2^{bW} \bmod p$ in Protocol 9-4.
13. Show how in Protocol 9-4 Alice can be assured that it is Steve, i.e., that the other side has the information stored at Steve. Explain why someone who has stolen Steve's database (but cannot find Alice's actual password through a dictionary attack) cannot impersonate Alice to Steve.
14. Explain how Alice and Steve each compute K in the SRP protocol (Protocol 9-5).
15. Show two-message credentials download protocols built upon SPEKE, PDM, and SRP.
16. Why is the EKE-based protocol in Protocol 9-7 insecure? (Hint: Someone impersonating Steve can do a dictionary attack, but show how.) How can you make it secure while still having Steve transmit $g^b \bmod p$ unencrypted?
17. Consider the protocol in Protocol 9-8. How would Alice compute K ? How would Steve compute K . Why is it insecure? (Hint: Someone impersonating Steve can do a dictionary attack, but show how.)

**Protocol 9-7.** For Homework Problem 16**Protocol 9-8.** For Homework Problem 17

10 *TRUSTED INTERMEDIARIES*

10.1 INTRODUCTION

If nodes Alice and Bob want to be able to communicate securely, they need to know keys for each other. Configuring each node with keys for every other node will not scale beyond a small number, so a **trusted third party** (someone that Alice and Bob trust) is used for introducing Alice and Bob to each other.

In this chapter we describe different types of systems based on trusted third parties. One is a system that uses only secret keys, in which case the trusted third party is usually known as a **KDC (Key Distribution Center)**. In a public key system, the trusted third party is usually known as a **certification authority (CA)**, and it signs **certificates**, which assert things such as the mapping between the name of something and its public key. In this chapter, we describe the concepts behind Kerberos (a KDC-based system) and **PKI (Public Key Infrastructure)**. A PKI includes components such as certificates, CAs, revocation mechanisms, and directories. We also describe DNSSEC (Domain Name System Security Extensions) that can be considered a form of PKI.

In a KDC-based system, the KDC has a database consisting of a secret key for each **principal**. A principal is any human or service for which the KDC facilitates secure communication. Each principal just needs to initially know the single secret that it shares with the KDC.

So the KDC knows keys K_A (Alice's secret) and K_B (Bob's secret). If Alice wants to talk to Bob, she asks the KDC to help her talk to Bob by creating a new key K_{A-B} for Alice and Bob to share. K_{A-B} can then be securely sent to Alice by being encrypted with K_A and securely sent to Bob by being encrypted with K_B .

In contrast, with a PKI solution, the CA signs a certificate saying "Alice's public key is 928...38021." When Alice and Bob wish to communicate they can simply send each other their certificates, and then they can do mutual authentication and establish a shared secret key for encrypted communication. Note that with a PKI solution, the CA never sees the private key of any node, so compromising the CA cannot leak any private keys of users or services. But if a compromised CA creates a fraudulent certificate mapping Alice's name to a key belonging to an attacker,

that attacker can impersonate Alice. There might be a service that keeps Alice's private key, either because it's required by her employer or the government or in case she loses it and needs help recovering it. This might be a service provided by the CA, but logically it is a different function. This service might be called a **key escrow service** or a **key recovery service**.

10.2 FUNCTIONAL COMPARISON

The KDC solution requires an online server (the KDC). This had several disadvantages:

- If the KDC is broken into, the attacker learns the secret keys of all the resources. This enables the attacker to impersonate all the resources, and it also enables the attacker to eavesdrop on conversations between Alice and Bob.
- The fact that Alice is talking to Bob will be known by the KDC. The KDC can also remember the key that it created for Alice and Bob to communicate and so can decrypt all their communication.
- If the KDC is down, no new connections can be made between any resources. It is possible to have multiple KDCs, both for robustness and for load splitting, but the more places where the database of secrets is stored, the more difficult it will be to protect them.

These properties might actually be viewed as advantages over a PKI solution in some situations (*e.g.*, within a company). The KDC is a central place that can keep a record of which resources have communicated. It is also easy to remove a resource, such as when the company finally decides to fire Trudy after all her eavesdropping and impersonation of other employees. Removing her from the KDC database will prevent Trudy from starting new conversations, but it will not immediately stop any ongoing conversations involving Trudy.

In contrast, in a public key design, the CA need not be online. This makes it easy to protect it from network-based attacks. It also makes it easier to physically protect it (*e.g.*, it can be kept in a vault). Bob or Alice might keep an audit log of who they have communicated with, but there is nothing like a KDC that will know they have communicated.

Although in a PKI system, the CA need not be online, revocation of either a key (Alice's key was compromised, but Alice is still a welcome member of the network) or a resource (Alice is no longer welcome on the network) is usually implemented with an online revocation service that advertises the list of revoked certificates or the list of valid certificates, or can be queried about a specific certificate. The revocation service is not as security-sensitive as a CA or KDC. If the revocation service is compromised, it might fail to accurately reflect the validity status of a certificate, but it will not be able to grant certificates to bogus keys or reveal users' private keys.

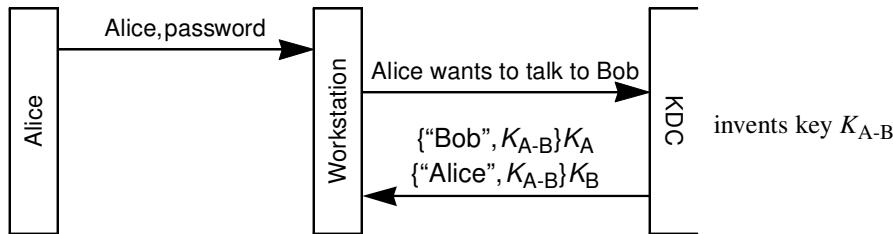
10.3 KERBEROS

Kerberos is a KDC-based design. It was originally designed at MIT [MILL87] based on a design by Needham and Schroeder [NEED78]. Because of patents on public key cryptography, Kerberos was designed to avoid any use of public key cryptography. Public key cryptography is a more natural solution, but even though patents have long expired, Kerberos (and designs that use it such as Microsoft Active Directory) are still in widespread use because of a large installed base.

For trust and scalability reasons, there would not be a single KDC for the world. Instead, a KDC serves some set of nodes. The KDC and the nodes it serves are known as a **realm**.

10.3.1 KDC Introduces Alice to Bob

The simplest way that Kerberos can be used is for Alice to directly request that the KDC help her talk to Bob (Protocol 10-1). In §10.3.3 *Ticket Granting Ticket (TGT)* we will discuss how it is usually done. Note that if Alice is a human, her secret will be derived from her password. This means that human Alice's secret is likely to be cryptographically weak, and, as we'll see in §10.3.5 *Making Password-Guessing Attacks Difficult*, the Kerberos design attempts to compensate for that.



Protocol 10-1. Alice Requests an Introduction to Bob

Alice types her password, and her workstation converts it into her secret, K_A . We'll show that step in Protocol 10-1, but for simplicity we will usually refer to Alice's workstation as "Alice". The KDC invents a new key, K_{A-B} , for Alice and Bob to share, and sends two items to Alice:

- $\{\text{``Bob'', } K_{A-B}\}K_A$
- $\{\text{``Alice'', } K_{A-B}\}K_B$

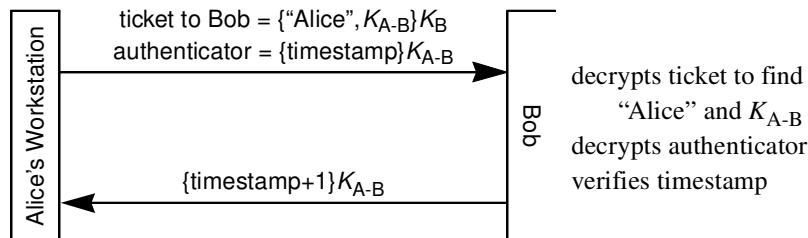
Note that when we refer to encryption, we mean some sort of mode (see Chapter 4 *Modes of Operation*) that provides both encryption and integrity protection. The quantity $\{\text{``Bob'', } K_{A-B}\}K_A$ can be decrypted by Alice, since it is encrypted with K_A . That quantity informs Alice that if she wants to talk to Bob, she should use the key K_{A-B} . The quantity $\{\text{``Alice'', } K_{A-B}\}K_B$ cannot be decrypted by Alice, but it can be decrypted by Bob. That message tells Bob that to communicate with Alice, he

should use the key K_{A-B} . In theory, the KDC could have sent $\{\text{Alice}, K_{A-B}\}K_B$ to Bob, but looking up Bob's IP address and opening a connection to Bob is a hassle for the KDC. Alice is going to be talking to Bob soon anyway, so the KDC sends the message intended for Bob to Alice. This message ($\{\text{Alice}, K_{A-B}\}K_B$) is known as a **ticket** to Bob. When Alice starts the conversation with Bob, she sends the ticket, which informs Bob that if someone who contacts him can prove they know the key K_{A-B} , they are "Alice".

For Kerberos purists, the entire response from the KDC is actually encrypted with Alice's key K_A . There is no security reason for the ticket to Bob to be encrypted with Alice's key—any eavesdropper can see it in the clear when Alice contacts Bob (see §10.3.2).

10.3.2 Alice Contacts Bob

In Protocol 10-2, Alice starts a connection with Bob. Notice that this protocol does mutual authentication. Bob knows, based on the information in the ticket, that if the thing talking to him knows K_{A-B} , it is "Alice". When Bob decrypts the authenticator with K_{A-B} , if the timestamp is close enough to what Bob thinks is the current time, he assumes the thing he is talking to knows K_{A-B} .



Protocol 10-2. Logging in to Bob from Alice's Workstation

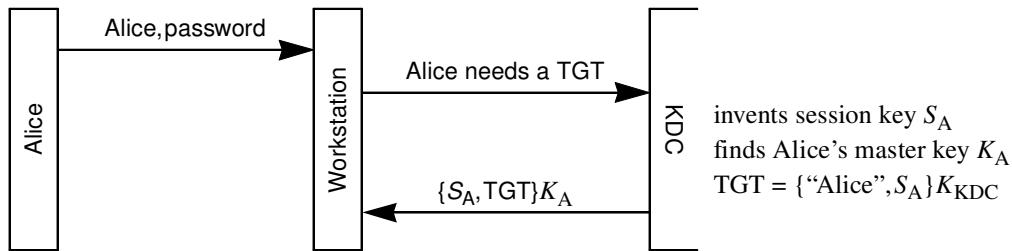
The timestamp in the authenticator field has to be reasonably close to the time on Bob's clock (say within five minutes or so), and Bob needs to remember all the timestamps it has received from Alice within that time window. This field prevents someone who eavesdropped on the conversation from replaying the conversation. For example, if the result of the Alice-Bob conversation were that Alice tells her bank (Bob) to transfer money to Carol, then it would be bad if Carol could replay the entire conversation to Bob, even if Carol could not decrypt the conversation.

To authenticate to Alice, Bob needs to prove he also knows K_{A-B} . It wouldn't be secure to have Bob just send the same encrypted timestamp back, so in Kerberos v4, Bob increments the time that Alice sent (which is why we incremented the timestamp in Protocol 10-2). In Kerberos v5, Bob does send the same timestamp, but it's part of a message that has other information inside (such as that this is a response to Alice's message), so the encrypted value sent by Bob will be different from what Alice sent. After the mutual authentication, it is optional whether Alice and Bob want to

encrypt and/or integrity-protect the remainder of their conversation. If they do want to cryptographically protect their conversation in Kerberos v4, they'd use the key K_{A-B} . In Kerberos v5, Alice chooses a new secret key S_{A-B} for this conversation and includes S_{A-B} with the timestamp in the authenticator.

10.3.3 Ticket Granting Ticket (TGT)

It is good security practice for Alice's workstation not to remember Alice's password or master secret K_A for any longer than necessary (so as to minimize exposure to malware). For that reason, Alice's workstation requests the KDC to give it a session key S_A and a TGT. The TGT is a ticket to the KDC that informs the KDC that Alice's current session key is S_A . (See Protocol 10-3.)



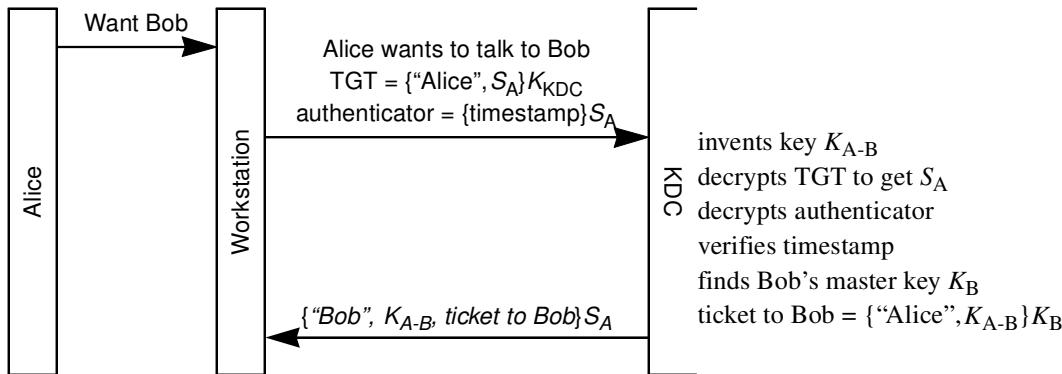
Protocol 10-3. Obtaining a TGT

Why isn't it just as much of a problem for someone to steal Alice's session key S_A and TGT as it would be if they stole her long-term secret K_A ? The reason is that the TGT expires, usually in a few hours, and usually includes limits on how it can be used. For instance, the TGT might include the IP address from which the KDC received the TGT request. This means that someone who stole S_A and TGT would not be able to use it to impersonate Alice unless they could also impersonate her IP address.

Note for purists: The KDC is usually described as being implemented by two different services—one that gives out TGTs and one that gives out tickets. We find that detail just makes the description more complicated, so we will use the name KDC for both services (see Homework Problem 4). So, Alice's workstation requests a TGT from the KDC. The KDC invents a session key S_A for Alice and sends two items to Alice's workstation, both encrypted with Alice's master key K_A —the session key S_A and the TGT. (Note that there is no reason for the TGT to be encrypted with Alice's key when transmitted to Alice, but Kerberos happens to do that.)

Alice can't decrypt the TGT. Only the KDC can read the information in the TGT, because it is encrypted with K_{KDC} (the KDC's key). After receiving the session key and TGT, Alice's workstation then forgets Alice's password and her master key K_A . The KDC does not remember the session

key it gave Alice, but when Alice needs a ticket to Bob, she sends the TGT along with the request for a ticket to Bob, and the KDC will be able to figure out Alice's session key by decrypting the TGT (see Protocol 10-4). Note that the ticket to Bob is encrypted with Bob's master key K_B . Suppose Bob, like Alice, wants to convert his master key into a session key and TGT and then forget K_B . How could that work? (See Homework Problem 8.)



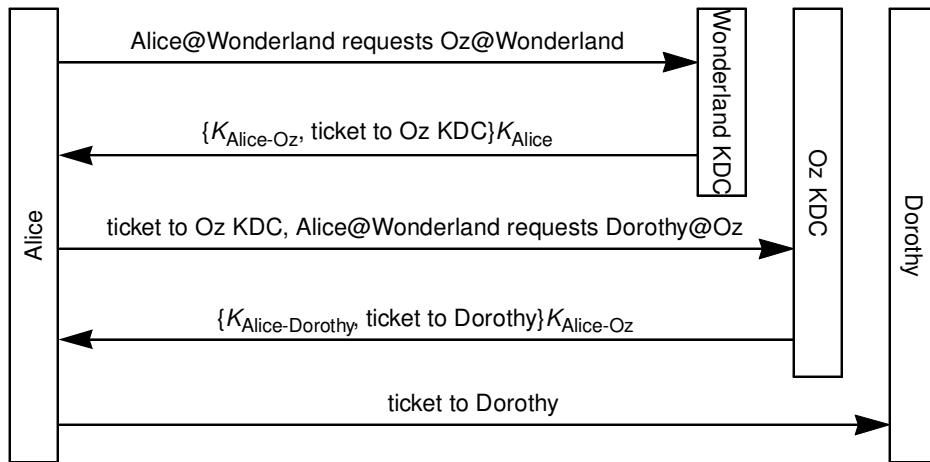
Protocol 10-4. Alice Uses a TGT to Get a Ticket to Bob

10.3.4 Interrealm Authentication

Suppose the world is partitioned into n different Kerberos realms. It might be the case that principals in one realm need to securely communicate with principals in another realm. This is supported by Kerberos. The way it works is that the KDC in realm B can be registered as a principal in realm A . This allows users in realm A to access realm B 's KDC as if it were any other resource in realm A , and once a user in realm A can access realm B 's KDC, the user can ask for tickets to principals in realm B .

Suppose Alice, in realm Wonderland, wishes to communicate securely to Dorothy in realm Oz. Alice's workstation notices that Dorothy is in a different realm (because Dorothy will have a name like `Dorothy@Oz`). Alice asks her KDC for a ticket to the KDC in realm Oz (see Protocol 10-5). If the managers of Wonderland and Oz have decided to allow this, the KDC in Oz will be registered as a principal in Wonderland. Now Alice can communicate with Oz's KDC, and Alice can then ask the Oz KDC to give her a ticket to Dorothy. The Oz KDC then issues a ticket for Alice to talk to Dorothy (see Protocol 10-5).

After Alice connects to Dorothy using this ticket, Alice and Dorothy will know they are talking to each other, and they will share a key K_{A-D} , the same as if they were in the same realm. The ticket to Dorothy will include the list of realms that were traversed to get from Alice to Dorothy.



Protocol 10-5. Interrealm Authentication

The Oz KDC will have a different secret for each realm that it connects to. There will be a key $K_{Oz-Wonderland}$ stored for the principal “Oz KDC” in Wonderland. And if the realm Oz cooperates with the realm Mordor, the key for the Oz KDC in realm Mordor would be $K_{Oz-Mordor}$. In the third message in Protocol 10-5, the Oz KDC will know it should decrypt the ticket with $K_{Oz-Wonderland}$, because the requestor name is Alice@Wonderland. If there were several intermediate realms Alice traversed to get to Oz, the Oz KDC will use its key in the last realm specified in the list of realms.

10.3.5 Making Password-Guessing Attacks Difficult

If Alice is a human, her master key K_A is derived from a password. There is no way in Kerberos to prevent Eve (an eavesdropper) from doing a password guessing attack, based on observing the protocol (Protocol 10-3) in which Alice asks for a TGT. The protocol involves the KDC sending a TGT encrypted with K_A . The TGT will have recognizable fields so Eve will be able to recognize a likely password.

But it's even easier to merely ask for a TGT for Alice than to try to eavesdrop during the exchange where Alice asks for a TGT. If the protocol for requesting a TGT required no authentication of the requester, Trudy could easily say “Hey, I'm Alice; send me a TGT” and the KDC would send Trudy the TGT encrypted with K_A , and then Trudy could do an offline password guessing attack.

To make it harder for Trudy to just ask for a TGT, Kerberos added (in version 5) a mechanism called **preauthentication** to prove that whoever is asking for a TGT for Alice knows her password.

This is done by having Alice include a timestamp, encrypted with Alice's master key K_A , in the TGT request. The KDC will not issue a TGT unless the preauthentication field is valid (the timestamp decrypted with K_A is close enough to the current time).

Another potential opportunity for password guessing is for Alice to request a ticket to human Bob. Bob's master key, K_B , is likely to be derived from a password, because Bob is a human. Kerberos prevents this attack by allowing principals in the database to be marked as *do not issue tickets to this principal*.

10.3.6 Double TGT Protocol

There might be cases where it would make sense for human Alice to request a ticket to human Bob, for instance, so she could send him encrypted email, where she could include the ticket in the header of the encrypted email. There also might be cases where non-human principals might wish to exchange their master secret for a TGT and session key. For example, a server might have its super-secret master key stored in protected hardware, so only accessible through the hardware and thus slow to use. Therefore, the server might want to obtain a time-limited, easily accessible session key that it feels safe storing in software to use for most of its operations.

Suppose Bob no longer knows his master key, because he forgot it after he used it to obtain a TGT and session key. Since the KDC does not keep track of session keys, if Alice asks for a ticket to Bob, the KDC will give her a ticket encrypted with Bob's master key. But Bob will not be able to decrypt the ticket, since he no longer knows his master key. If Bob is a user at a workstation, the workstation could at this point prompt him to retype his password, but this would be inconvenient for him.

Kerberos assumes Alice knows that Bob is the kind of thing that is likely to have exchanged his master key for a session key. In a method unspecified in Kerberos, Alice is supposed to ask Bob for his TGT (see Homework Problem 6). Alice then sends Bob's TGT as well as her own TGT to the KDC (hence the name **double TGT authentication**). Since Bob's TGT is encrypted under a key that is private to the KDC, the KDC can decrypt it. It then issues a ticket to Bob for Alice that is encrypted with Bob's session key rather than Bob's master key.

Another way to do this is for Bob to send the KDC the ticket Alice sent him and his TGT and have the KDC reissue the ticket encrypted with Bob's session key (see Homework Problem 10).

10.3.7 Authorization Information

There is a field in Kerberos tickets and TGTs named AUTHORIZATION-DATA. This can contain Alice's groups and roles. In Microsoft's version of Kerberos, this information is configured into the KDC under principal Alice's information. By default, the KDC puts this information into TGTs and

tickets. Alice can ask for a subset of this information, which will limit her rights (principle of least privilege), or she can ask for a subset of this information when delegating to Bob (see §10.3.8 *Delegation*).

10.3.8 Delegation

Suppose Alice wishes for Bob to be able to act on her behalf. For instance, Bob might be a backup service that will scan her file system for any files that have changed since the last backup. She might want to give Bob only read permission on her files so he can't corrupt them. She could send Bob her master secret. Or she could send Bob her session key and TGT. Security people generally frown on giving out your secrets and letting someone impersonate you.

To make tickets and TGTs more secure so only the requester can use them, Kerberos (version 4) always includes the IP address of the requester in the ticket or TGT. This means that even if Trudy were to somehow obtain Alice's TGT and session key, Trudy would not be able to use these unless she could impersonate Alice's IP address. In version 5, the IP address in a ticket or TGT is optional, because requiring a specific IP address is too restrictive in cases where something had multiple IP addresses or has moved.

However, suppose Alice really wants Bob to be able to act on her behalf. Kerberos allows Alice to send her TGT to the KDC and request a new TGT with the name "Alice" but with Bob's IP address or without an IP address. Also, Alice can limit what Bob can do on her behalf by requesting that the list of groups and roles in the ticket's AUTHORIZATION-DATA be shrunk from the total list of groups and roles Alice belongs to.

10.4 PKI

A **public key infrastructure (PKI)** consists of the components necessary to securely distribute public keys—certificates, a method to revoke certificates, and a method to evaluate a chain of certificates from public keys that are known and trusted in advance (**trust anchors**) to the target name. There have been some public key systems deployed that leave out components such as revocation, or even certificates. Instead, Alice might be configured with Bob's public key, or Bob might send his public key to Alice (rather than sending a certificate), and Alice will store it, taking it on faith that she was talking to the real Bob during the interaction where he sent his public key. The industry terminology for this strategy is either **trust on first use**, or **leap of faith**. If an active attacker noticed the initial message from Bob to Alice, the attacker could replace Bob's public key with the

attacker's public key and then be able to impersonate Bob to Alice. But in many cases, having Bob just send Alice his public key is reasonably secure in practice.

10.4.1 Some Terminology

A certificate is a signed message vouching that a particular name goes with a particular public key, as in [Bob's public key is 829348]_{Carol}. If Carol signs a certificate vouching for Bob's name and key, then Carol is the **issuer** and Bob is the **subject**. If Alice wants to discover Bob's key, then Bob's name is the **target**. A **trust anchor** is a public key that the verifier has decided is trusted to sign certificates. A **chain of certificates** is a sequence of certificates where each certificate is signed with the key that is certified in the previous certificate. If Alice is to believe a chain, the first certificate would be one of Alice's trust anchors. If Alice is evaluating a certificate or a chain of certificates, she is the **verifier**, sometimes called the **relying party**. Anything that has a public key is known as a **principal**.

There is often other information in the certificate that might determine whether Alice considers the certificate valid. This information includes an expiration date, where to look for revocation information about this certificate, whether the subject should be trusted to act as a CA, policies such as how carefully the subject was checked (*e.g.*, multifactor authentication, security clearance), and name constraints.

Sometimes public keys are distributed in the form of **self-signed certificates**. A self-signed certificate says [Bob's public key is 829348]₈₂₉₃₄₈. There is no security difference between a self-signed certificate asserting what Bob's public key is (signed by the key asserted as being Bob's key) and an unsigned message claiming the same information. The reason information is sometimes exchanged in this form (self-signed certificates) is that there is already code to parse a certificate. But it's important to realize that there is no security gained from the signature, and the only reason to verify the signature is if your computer is bored and needs something to do. Some people claim that a self-signed certificate should be considered invalid if it is signed using an out-of-favor cryptographic algorithm such as MD5. If someone says that to you, just smile and nod.

10.4.2 Names in Certificates

A certificate mapping a name to a key is an incredibly simple concept. The most widely deployed standard for certificates used on the Internet is PKIX (Public Key Infrastructure using X.509) (RFC 5280). X.509 was jointly published by ITU, as ITU-T X.509, and by ISO, as ISO/IEC 9594-8. X.509 certificates map an X.500 name to a public key, rather than mapping a DNS name to a key. X.500 names are perfectly reasonable hierarchical names, similar in spirit to the DNS names used on the Internet, but they have different syntax and are administered by a different organization. This

makes things unnecessarily complicated when used on the Internet, since X.500 names are not used for Internet applications.

Some people were unhappy with basing Internet certificates on X.509 certificates, and they started a working group within the IETF called SPKI (Simple Public Key Infrastructure). If SPKI certificates simply mapped DNS names to public keys, that would have been a nice simple form of certificate for the Internet, but instead the SPKI group tried to be really innovative and have certificates not be based on names at all, or have the names be relative to whoever is using the name (*e.g.*, Nicki's cousin's friend). So SPKI did not catch on, and the Internet is using PKIX certificates.

In PKIX, both the ISSUER field and the SUBJECT field in certificates are X.500 names. Internet applications do not care about X.500 names. When a CA certifies a DNS name, PKIX uses an extension known as the SUBJECT ALTERNATIVE NAME field. Usually, the party purchasing a certificate creates the unsigned certificate and sends it to the CA to be signed. So, to get the name `example.org` certified, `example.org` puts `example.org` into the SUBJECT ALTERNATIVE NAME field. There were other places one could put the DNS name, but they have been deprecated. It was also legal for the DNS name to be encoded in the CN field, which is a component of the X.500 name known as “common name”). There was another encoding where the components of the X.500 name were labeled as DC (domain component), and a DNS name such as `labs.example.com` could be encoded as `DC=com`, `DC=example`, `DC=labs`. Having multiple places where the DNS name could be encoded is a potential security vulnerability. Before the CA signs a certificate, it needs to verify that all the information in the certificate is accurate. For instance, the CA should make sure that Bob really owns all names encoded anywhere in the certificate.

Even more complicated, it would be nice to have certificates for human users. Users have lots of different types of names, *e.g.*, email addresses, social media handles, and legal names. In theory, any of these could be stored in the SUBJECT ALTERNATIVE NAME of a PKIX certificate.

Some people get excited about the syntax of certificates and complain about PKIX because its syntax is ASN.1, which is somewhat verbose and a bit computationally difficult for a computer to parse. We don't care about syntax, as long as anything necessary can be stated in the syntax and the result is unambiguous. But, interestingly, the people who tend to get upset about ASN.1 seem not to complain about XML, which is even more verbose.

10.5 WEBSITE GETS A DNS NAME AND CERTIFICATE

A website needs a DNS name. It chooses a top-level domain from which to purchase the name (*e.g.*, `.com`, `.org`), contacts the registrar associated with that domain, and requests a name that it likes. If the registrar says the name is already purchased, the aspiring website needs to choose another name. When the website finds a name that it can purchase, say, `example.org`, it purchases the

name. The DNS registrar for .org adds an entry in its domain for `example.org` and puts in information for it such as its IP address.

It would make a lot of sense if the same registrar from which the website is purchasing a DNS name also issued a certificate. While the website is purchasing the name, it has a secure connection to the registrar so that it can send information such as a credit card number. However, that is not how it is done. Instead, there are other organizations, unaffiliated with DNS registrars, that issue certificates. So, after purchasing the DNS name, `example.org` contacts a CA and says “My DNS name is `example.org` and I’d like you to certify that my key is 947289143.” How is the CA supposed to know this is the legitimate owner of the name `example.org`?

There is no standard for doing this, but one method is for the CA to look up the name `example.org` in DNS and find the associated IP address. The CA sends a secret number to that IP address (similar to when a bank sends a PIN to your cellphone). If whoever is requesting a certificate for `example.org` can then tell the CA what the secret number is, then the CA assumes that they can receive at the IP address listed in DNS, and then the CA is willing to issue a certificate to the name `example.org`.

Note that if being able to receive at a particular IP address were secure, there would be no need for certificates. This is an example of a leap of faith for the CA. Again, what would make the most sense is for the DNS registrar to also be the CA associated with names that it issued, but the CA organizations want to have standards for how securely a CA stores its key, how often the CA operators are drug tested, and so on, and the DNS organizations might not find these rules acceptable.

10.6 PKI TRUST MODELS

In this section we will explore various strategies for getting trust anchors and finding chains of certificates that lead to a target name. This section is about how a verifier would operate under various models.

10.6.1 Monopoly Model

In this model, the world chooses one organization, universally trusted by all countries, companies, and organizations, to be the single CA for the world. That organization’s public key is embedded in all software and hardware as the single PKI trust anchor. Everyone must get certificates from that one organization.

This is a wonderfully simple model, mathematically. This is probably the model favored by organizations hoping to be chosen as the monopolist. However, there are problems with it:

- There is no single universally trusted organization.
- Given that all software and hardware would come preconfigured with the monopoly organization's key, it would be infeasible to ever change that key in case it were compromised, since that would involve reconfiguration of every piece of equipment and software.
- It would be expensive and insecure to have a remote organization certify your key. How would they know it was you? How would you be able to securely send them your public key?
- Once enough software and hardware were deployed so that it would be difficult for the world to switch organizations, the organization would have monopoly control and could charge whatever it wanted for granting certificates.
- The entire security of the world rests on that one organization never having an incompetent or corrupt employee who might be bribed or tricked into issuing bogus certificates or divulging the CA's private key, since that one CA can impersonate the entire world.

10.6.2 Monopoly plus Registration Authorities (RAs)

This model is just like §10.6.1 *Monopoly Model*, except that the single CA trusts other entities, known as **registration authorities (RAs)**, to securely check identities of certain principals and obtain and vouch for their public keys. For instance, an RA might be run by the IT department of a company for vouching for the keys of that company's employees. The RA then securely communicates with the CA (because the CA and RA have a relationship and know how to authenticate each other), and the CA can then issue a certificate.

This model's advantage over the monopoly model is that it is more convenient and secure to obtain certificates, since there are more places to go to get certified. However, all the other disadvantages of the monopoly model apply.

RAs can be added to any of the models we'll talk about. Some organizations have been convinced that it is better for their organization to run an RA and pay a CA organization to create certificates. They believe the CA organization will be more expert at what it takes to be a CA (*e.g.*, protecting the CA private key and maintaining a revocation infrastructure). However, in practice, the CA just rubber-stamps whatever information is verified by the RAs. It is the RA that has to do the security-sensitive operations of ensuring the proper mapping of name to key. The CA might be better able to provide a tamper-proof audit trail of certificates it has signed, though.

RAs are invisible to a verifier. Certificates would still be signed by the CA, so verifiers only see certificates issued by CAs.

10.6.3 Delegated CAs

In this enhancement to other models, a trust anchor CA can issue certificates to other CAs, vouching for their keys and trustworthiness as CAs. Principals can then obtain certificates from one of the delegated CAs instead of having to get a certificate directly from the verifier's trust anchor CA.

The difference between a delegated CA and an RA comes down to whether a verifier sees a chain of certificates from a trust anchor to Bob's name or sees a single certificate.

10.6.4 Oligarchy

This is the model commonly used in browsers. In this model, instead of having products preconfigured with a single trust anchor key, the products come configured with hundreds of trust anchors. A certificate issued by any of the trust anchors, or a chain originating with one of them, is accepted by the browser. Sometimes in such a model it is possible for the user to add or delete trust anchors. The oligarchy model has the advantage over the monopoly model that the organizations chosen as trust anchors will be in competition with each other, so the world might be spared monopoly pricing. However, it is likely to be even *less* secure than the monopoly model:

- In the monopoly model, if the single organization ever has a corrupt or incompetent employee, the entire security of the world is at risk. In the oligarchy model, *any* of the trust anchor organizations getting compromised will put the security of the world at risk.
- The trust anchor organizations are trusted by the product vendor, not necessarily by the user. Why should the vendor decide which organizations the user should trust?
- It might be easy to trick a user into adding a bogus trust anchor into the set. This depends on the browser implementation. One implementation that was common, but no longer behaves this way, displayed a pop-up box if the server the browser was talking to presented a certificate signed by a public key that was not in the browser's list of trust anchors. The language in the pop-up box was much more confusing than our rewording of the questions:

Warning. This was signed by an unknown CA. Would you like to accept the certificate anyway? (The user will almost certainly say OK.)

Would you like to always accept this certificate without being asked in the future? (OK.)

Would you like to always accept certificates from the CA that issued that certificate? (OK.)

The first *OK* says the user is happy to go to that site anyway. The second *OK* says the user is willing to always trust that certificate for that one site. The third *OK* installs the unknown CA's public key into the set of trust anchors. It would be an interesting psychology exercise to see how outrageous you can be before the user stops clicking *OK*. *Would you like to always accept certificates from any CA? (OK.) Since you're willing to trust anyone for anything,*

would you like me to make random edits to the files on your hard drive without bothering you with a pop-up box? (OK.)

Note that if a user is sufficiently sophisticated and careful, she can ask for information about the certificate before clicking *OK* to accept it. She will be informed of the name of the signer, say, **Mother Teresa** (the most trustworthy imaginable signer). But this does not necessarily mean it was really signed by Mother Teresa. It just means that whoever signed it (for example, *SleazeInc*) put the string **Mother Teresa** into the ISSUER NAME field.

- Users will not understand the concept of trust anchors. If they have been assured that the application they are using does encryption, they will assume that it will be secure even if they're using a public workstation, perhaps in a hotel room or at an airport. Although it will always be an issue if a user can be tricked into using a public workstation with malicious code, it would be easier for the previous user of the workstation to modify the set of trust anchors (probably not a privileged operation) than to change the software.
- There is no practical way for even a knowledgeable user to be able to examine the set of trust anchors and tell if someone has modified the set. Browsers today come shipped with hundreds of trust anchors. A user can look at the set of trust anchors. Each entry has a name and a key, but someone could delete the key of *TrustworthyInc* and put in a new key claiming that it belongs to *TrustworthyInc*. You might even be able to look at the public keys, but what user will be sufficiently paranoid to have printed out all the certificate hashes displayed in the list of trust anchors in order to compare them with the set currently displayed?

Today, most browser implementations make it difficult or impossible for a user to modify the set of trust anchors, and users just have to trust that the browser vendor is making sure the list only has trustworthy trust anchors. Many company IT departments manage the list of trust anchors for their employees' devices.

10.6.5 Anarchy Model

This model is also sometimes called the **web of trust**. Each user, say Alice, is responsible for configuring some trust anchors, for instance, public keys of people she has met and who seem trustworthy and who send her their public key in some reasonably secure way.

In this model, anyone can sign certificates for anyone else. In many gatherings of nerds, there is a PGP key-signing party with some sort of ritual where keys are exchanged in email first, and then people get up and state their name and a hash of their key, and other people can vouch for that person having that name. People at the party can then sign certificates for the person who has just announced his name and key. Some organizations volunteer to keep a certificate database into which anyone can deposit certificates. These databases can be read by anyone. To get the key of someone whose key is not in user Alice's set of trust anchors, she can search through the public

database of certificates to try to find a path from one of her trust anchors to the target name. This eliminates the monopoly pricing, but it is really unworkable on a large scale:

- The database would get unworkably large if it were deployed on Internet scale. If every user donated, say, ten certificates, the database would consist of billions of certificates. It would be impractical to search through the database and construct paths.
- Assuming somehow Alice could piece together a chain from one of her trust anchors to the name **Bob**, how would she know whether to trust the chain? Say Carol (her trust anchor) vouches for Ted's key. Ted vouches for Gail's key. Gail vouches for Ken's key. Ken vouches for Bob's key. Are all these individuals trustworthy?

As long as this model is used within a small community where all the users are trustworthy, it will appear to work. However, on the Internet scale, when there are individuals who will purposely add bogus certificates and naive users who will be tricked into signing bogus certificates, it would be impossible to know whether to trust a path. Some people have suggested that if you can build multiple chains to the target name you can be more assured of the trustworthiness. But once someone decides to add bogus certificates, they can create arbitrary numbers of fictitious identities and arbitrary numbers of certificates signed by those entities. So, sheer numbers will not be any assurance of trustworthiness.

10.6.6 Name Constraints

The concept of name constraints is that the trustworthiness of a CA is not a binary value where a CA would either be completely untrusted or trusted for everything. Instead, a CA should only be trusted for certifying some subset of the users, and in particular, the name by which I know you implies whom I trust to certify the key for that name. If I want the key for the name `roadrunner@socialnetworksite.com`, I would trust a CA associated with `socialnetworksite.com` to certify the key. If I want the key for `Bob.Smith@example.com`, I would trust a CA associated with `example.com` to certify the key for that name. If I want to know the key for `creditcardnumber4928749287@bigbank.com`, I would trust a CA associated with `BigBank.com` to certify the key for that name. These names might all refer to the same human, but that is irrelevant. User Alice might use different public keys for each of her names, or she might use the same public key for some of her names. But that does not affect whom I should trust to certify the binding of a particular name to a key.

Note: Usually the DNS name of the registrar that can vouch for a name (*e.g.*, a social media handle) will somehow be derivable from the name the human is using, even if the human name is not the same syntax as an email address.

10.6.7 Top-Down with Name Constraints

This model (top-down) is similar to the monopoly model in that everyone must be configured with a single trust anchor—the key of the one chosen root. That root CA delegates to other CAs, meaning that the root CA might sign a certificate to some other CA, but the certificate indicates that this CA is only trusted for issuing certificates in a portion of the namespace, *e.g.*, names of the form `*.com`, or `*.edu`.

In a hierarchical namespace such as DNS, there would be a CA associated with each node in the namespace tree. The CA associated with a parent node would certify the key of the CA associated with the child node and also indicate that this child CA is only trusted to issue certificates in the tree rooted at the name of that child node.

With the top-down name-constraint model, it allows the organization associated with a namespace to have policies that CAs in their organization should follow. For example, the organization associated with `*.cia.gov` is likely to want to have different policies for CA operators than the organization associated with the namespace of `*.mit.edu`, which will likely be managed by playful undergraduates unlikely to pass a drug test.

In this model, it is easy to find the path to a name (just follow the namespace from the root down). But it has the other problems of the monopoly model, in that everyone has to agree upon a root organization, the security of the entire world depends on that one organization never being compromised, and that organization and its key would be prohibitively expensive to ever replace.

10.6.8 Multiple CAs for Any Namespace Node

In any of these models (the ones we have described so far and the ones we will describe), it is possible to have multiple CAs representing any node in the namespace. For example, in the top-down model, if there are two nodes in competition to provide the service of being the root CA, then verifiers would either be configured with both CAs as trust anchors, or principals certified by a CA associated with that namespace will need to be certified by both CAs, and the principal will need to ask the verifier which CA is its trust anchor.

For links in a certificate chain other than the first link, the verification will work with whichever CA has been delegated to.

10.6.9 Bottom-Up with Name Constraints

This is the model we recommend. It was originally proposed for Digital's security architecture in the late 1980s by Charlie Kaufman. This model is created with two enhancements to the model in §10.6.7 *Top-Down with Name Constraints*. The two enhancements are:

- **up-links**, where a CA associated with a child node in the name space hierarchy certifies the key of the CA associated with the parent node
- **cross-links**, where any node in the namespace can certify the key of the CA associated with any other node in the namespace

In this model, Alice's trust anchor can be any node in the namespace. For example, if she works at a company, the IT department is likely to configure her device with the CA associated with her company's name as her trust anchor. If Alice isn't associated with a company, she might copy a list of trust anchors from a place she trusts or not even notice when her browser comes pre-configured. The rule for finding the key of a target name is that Alice starts at her trust anchor and, if that CA represents a node that is an ancestor (in the namespace) of the target name, she follows down-links. Otherwise, she tries to find a cross-link to an ancestor of the target name. If so, she follows that cross-link to the ancestor node in the namespace and then follows down-links to the target name. Or if there is no cross-link, she follows the up-link to go up one level in the namespace. If she's at an ancestor of the target name, she follows down-links. If not, she looks for a cross-link to an ancestor of the target name.

10.6.9.1 Functionality of Up-Links.

Up-links are shown in Figure 10-6.

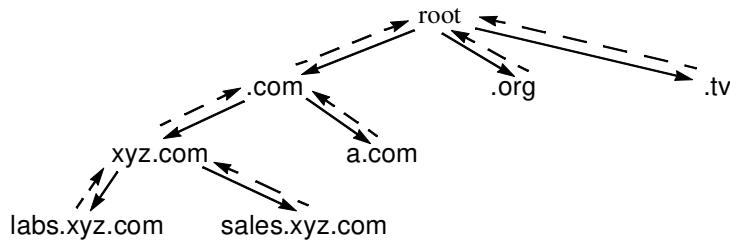


Figure 10-6. Up-links Allow Starting with Any Node as Your Trust Anchor

There are two advantages up-links provide over the top-down model:

- A root CA is no longer hard to replace. If the root CA misbehaves or its key gets compromised, the root key can be replaced by a different key reasonably painlessly, since it is only the child nodes (*e.g.*, the CAs associated with the top level domains in DNS) that need to revoke the old key and issue a new certificate. The vast majority of principals on the Internet would not have the root CA as their trust anchor, so replacing the root organization, or changing the key for that root, does not affect their key or their traversal rules.
- If the target name and trust anchor are in the same organization's namespace subtree, a compromised CA outside that namespace subtree will not be on the path of certificates between

principals in that namespace. For example, for principals in xyz.com (assuming they are configured with a CA associated with a node in the namespace xyz.com), if .com or the root were malicious, it could not impersonate principals in xyz.com to each other.

10.6.9.2 Functionality of Cross-Links

There are two advantages gained by cross-links:

- There is no need to wait for the entire PKI of the world to be connected. Organization a.com can deploy its own internal PKI, and xyz.com can deploy its own internal PKI. If the two organizations want the principals in their namespace to be able to find keys for principals in the other organization, they just need to cross-certify each other's keys. (See Figure 10-7.)

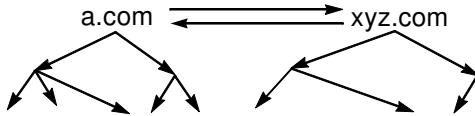


Figure 10-7. Cross-links Connect Two Organizations

- Another advantage of cross-links is that they allow bypassing CAs that are not trusted. (See Figure 10-8.) If the organization associated with the name tree under sales.xyz.com creates a cross-link to a.org, then the path of CAs will go directly from sales.xyz.com to a.org. For a principal whose trust anchor is sales.xyz.com or below, compromise of any of the CAs associated with xyz.com, .com, root, or .org will not affect the security of the chain from the namespace sales.xyz.com to a.org, since those CAs will not be in the chain. Note that the cross-link created by sales.xyz.com will not create a bi-directional cross-link unless a.org creates its own cross-link to sales.xyz.com. Principals in the namespace under a.org will have to follow the complete chain of certificates from a.org to its parent .org, to its parent root, and then down the namespace (through .com and xyz.com) to get to sales.xyz.com.

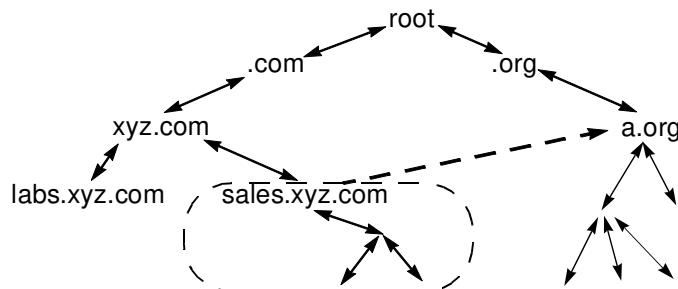


Figure 10-8. Cross-links for Added Security

To review, the advantages of the bottom-up model are:

- It is easy to find out if a path exists.
- The policy of assuming that the name by which something is known implies which CA you'd trust to certify the name is something people can understand, and it is sufficiently flexible and simple that it might actually work.
- PKI can be deployed in any organization independently of the rest of the world. There is no reason to pay a commercial CA to build a PKI for your own organization. There is no reason to wait for the entire world-encompassing PKI to be in place before you can use PKI in your own organization or between cooperating organizations.
- Since CA certificate chains between principals in your own organization never go outside of your own organization, security of what is presumably the most security-sensitive operation— authenticating users in your own organization—is entirely in your own hands. Compromise of any CA outside of your own organization will not allow anyone to impersonate one of your own principals to another in your name space.
- Replacing any key is reasonably easy. If a root service's key gets compromised, then it only affects the top CA of each of the root service's customers. Each such CA has to revoke the old certificate it issued to the root service and issue a new certificate containing the new key, and automatically all the users in the CA's subtree are using the new key in place of the old key.
- No organization gets so entrenched that it can start charging monopolistic prices. It is easy to replace any key, and competition is always possible (see §10.6.8 *Multiple CAs for Any Namespace Node*).

10.6.10 Name Constraints in PKIX Certificates

PKIX has a field called NAME CONSTRAINTS, which allows the issuer CA to specify which names the subject CA is trusted to certify. The name constraints can specify names of various forms, such as X.500 names, DNS names, or email addresses. For simplicity, let's just assume that the name constraints are DNS names, and we will assume the CA is associated with a name in the DNS hierarchy. The NAME CONSTRAINTS field can contain allowed names with wildcards (*e.g.*, *.example.com, which means any name that ends in .example.com) and disallowed names, also with wildcards.

Any of the models we've mentioned can be enforced with name constraints. To build the anarchy model or the oligarchy model, there would be no name constraints. In the top-down model, a certificate in which a CA certifies the key of the child CA in the DNS hierarchy would state *only trusted for this name and everything below that name*. In the bottom-up model, a child or cross-certificate would specify *only trusted to certify names in the subtree below the DNS name*.

that the subject CA represents. A parent certificate in the bottom-up model (an up-link) would contain the constraint *any names except the DNS name I represent and below, or any other names I have issued a cross-link to.*

We'd still recommend mostly building the bottom-up model, but there is some amount of flexibility that the strict *up*-cross once-down** algorithm might not give. For instance, an organization might have a cross-link to `other-org.com`, but realizing that `other-org.com` also keeps cross-certificates to `yet-another.com` and `still-another.com`, the name constraint in the cross-certificate might say that the subject would be trusted to certify names in the namespaces of any of `{other-org.com, yet-another.com, still-another.com}`. Or there might be several root organizations that all cross-certify each other, with each having certified some subset of the organizations. Since two organizations might not have been certified by the same root, it might be necessary to go up to the root, then find a path across a mesh of roots to the target's root, and then go down. This could be accomplished by having roots cross-certify each other using the name constraint *trusted for all names*. The further one gets from the bottom-up model, and the closer one gets to the anarchy model, the more complex it will be to search all valid paths.

Although PKIX certificates can contain name constraints, they are rarely used. If a CA's certificate contains name constraints, verifiers are supposed to check whether any certificates that CA has issued follows the name constraints, but not all verifiers do this.

10.7 BUILDING CERTIFICATE CHAINS

In all these models, there is the problem of how Alice, the verifier, obtains the relevant certificates for the target name Bob. There are various strategies. In email, a strategy was for a signed email message from Bob to contain a certificate chain to his key, hopefully including one of Alice's trust anchors. When Alice receives this, she can verify the chain and perhaps cache his public key (assuming Bob's chain includes one of Alice's trust anchors).

If there were directories associated with each CA, Alice could follow the path from her trust anchor(s) asking each CA along the path for up, down, or cross certificates. Note the actual CA need not be online. A directory just needs to store information signed by the offline CA. The directory would not be able to give a verifier an incorrect key because the directory would not be able to forge the CA's signature.

In protocols such as IPsec and TLS, the assumption is a top-down/oligarchy model. During the initial handshake, Alice informs Bob what her trust anchors are, and Bob sends her a chain of certificates from one of her trust anchors.

10.8 REVOCATION

If someone realizes their key has been stolen, or if someone gets fired from an organization, it is important to be able to revoke their certificate. A certificate typically has an expiration date, but since it is a lot of trouble to issue a certificate (especially if the CA is off-line), certificate lifetime is typically years, which is too long to wait for a certificate to expire if it needs to be revoked.

This is similar to what happens with credit cards. They, too, have an expiration date. They are usually issued to be good for a year or more. However, if a credit card is stolen, it is important to be able to revoke its validity quickly. Originally, the credit card companies published books of bad credit card numbers and distributed these books to all the merchants. Before accepting the card, the merchant would check to make sure the credit card number wasn't listed in the book. This mechanism is similar to a CRL (certificate revocation list) mechanism (see §10.8.1).

Today, the usual mechanism for credit cards is that for each transaction, the merchant calls an organization that has access to a database of invalid credit card numbers (or valid credit card numbers), and the merchant is told whether the credit card is valid (and perhaps, if there is sufficient credit for the purchase). This is similar to an OLRS (on-line revocation service) mechanism (see §10.8.2). The IETF standard protocol for requesting revocation status of a certificate is called OCSP (on-line certificate status protocol) and is documented in RFC 6960.

Why do certificates have expiration dates at all? Assuming there is a method of revoking them, the only security reason to have them expire is to make the revocation mechanism more efficient, for instance, by avoiding the CRL becoming overly large. Cynics might think the reason for designing certificates with expiration dates is so that companies that want to collect revenue from issuing certificates can collect multiple times for the same certificate. The PKIX certificate format does allow very long lifetimes, such as until 31 December 9999, which is, in practice, the same as not having an expiration date.

There might be cases where it would make sense for a certificate to have a very short lifetime, such as if you know this is only to be used for a temporary period. For example, the certificate might be a visitor badge valid for a week or a one-day parking permit. It is simpler to issue a short-lived certificate than to have to revoke the certificate.

10.8.1 CRL (Certificate Revocation list)

The basic idea of a CRL is that the CA periodically issues a signed list of all the revoked certificates. This list must be issued periodically, even if no certificates have been revoked since the last CRL, since otherwise an attacker could post an old CRL (from before his certificate was revoked). If a timestamped CRL is issued periodically, then the verifier can refuse to honor any certificates if

it cannot find a sufficiently recent CRL. Each CRL contains a complete list of all the unexpired, revoked certificates.

Delta CRLs are intended to make CRL distribution more efficient. Let's say you want to have revocation take effect within one hour. With a CRL, that would mean that every hour the CA would have to post a new CRL, and every verifier would have to download the latest CRL. Suppose the CRL was very large, perhaps because the company just laid off ten thousand people. Every hour, every verifier would have to download a huge CRL, even though very few certificates had been revoked after that layoff.

A delta CRL lists changes from some full CRL. The delta CRL would say "These are all the certificates that have been revoked since 10 AM 7 February", which is the timestamp of the full CRL that this delta CRL is referencing. The delta CRL would hopefully be very short, often containing no certificates. Only when the delta CRL gets large would it be useful to issue a new full CRL.

An idea we_{1,2} designed for making the CRL small again after it has become too large is what we_{1,2} call **first valid certificate**. This scheme also allows certificates to not have a predetermined expiration date when issued. Instead, they are only marked with a serial number, which increases every time a certificate is issued (or the issue time could be used instead of a serial number). A CRL would have one additional field that is not included in X.509. The CRL would contain a FIRST VALID CERTIFICATE field. Any certificates with lower serial numbers (or issue times) are invalid.

Certificates in our scheme would have no predetermined expiration date. As long as the CRL is of manageable size, there is no reason to reissue any certificates. If it looks like the CRL is getting too large, the CA organization issues a memo warning everyone with certificate serial numbers less than some number n that they'll need new certificates by, say, a week from the date of the memo. The number n is chosen so that few of the serial numbers in the current CRL are greater than n . Revoked certificates with serial numbers greater than n must continue to appear in the new CRL, while valid certificates with numbers greater than n do not have to be reissued. Some time later, say, two weeks after the memo is sent, the CA issues a new CRL with n in the FIRST VALID CERTIFICATE field. Affected users (those with serial numbers less than n) who ignored the memo will thenceforth not be able to access the network until they get new certificates, since their certificates are now invalid.

There are cases when even with this scheme it might be reasonable to have expiration dates in certificates. For example, at a university, students might be given certificates for use of the system on a per-semester basis, with a certificate that expires after the semester. Upon paying tuition for the next semester, the student is given a new certificate. But even in those cases, it may still be reasonable to combine expiration dates in some certificates with our scheme, since our scheme would allow an emergency mass-revocation of certificates.

10.8.2 Online Certificate Status Protocol (OCSP)

OCSP (RFC 6960) is a protocol for querying an OLRS (on-line revocation server) about the validity of individual certificates. If Bob is verifying Alice's certificate, Bob would ask the OLRS if Alice's certificate is valid. You might think that introducing an on-line server into a PKI eliminates an important security advantage of public keys because you now have an on-line trusted service. But the OLRS is not as security sensitive as a CA (or KDC). The worst the OLRS can do is claim that revoked certificates are still valid, so the damage is limited. It does not have a vulnerable database of user secrets (like a KDC does). Its key should be different from the CA's key, so if its key is stolen, the CA's key would not be compromised. Surprisingly, it is not uncommon to have the OLRS key be the same as the CA key.

An OLRS variant is for Alice to query the OLRS server about her own certificate. The OLRS response will be signed (by the OLRS server) and timestamped, so Alice can send the OLRS response along with her certificates to Bob. Assuming Alice will be visiting many resources, this saves the OLRS the work of talking to multiple verifiers, saves the verifiers the work of querying the OLRS, and saves the network from the bandwidth used by having multiple verifiers query the OLRS.

Bob can decide how quickly revocation should take effect. If he wants revocation to take place within, say, one hour, then he can insist that Alice's OLRS response be time-stamped within the last hour. If he complains Alice's OLRS response isn't sufficiently recent, then Alice can obtain a new one, or Bob could query the OLRS himself.

Alice can proactively refresh her OLRS response, knowing that most servers would want one that is, say, less than an hour old. Then the round-trip querying of the OLRS does not need to be done at the time of a transaction.

Even with Bob (instead of Alice) querying the OLRS, it is possible to do caching and refreshing. Bob can keep track of the certificates in the chain for users that tend to use his resource and proactively check with the OLRS to see if any of them have been revoked.

Note that there's even a bigger performance gain if the server uses this strategy and collects an OLRS response to use with multiple clients.

10.8.3 Good-Lists vs. Bad-Lists

The standards assume that the CRL will contain all the serial numbers of bad certificates, or that the OLRS would have a database of revoked certificates. This sort of scheme is known as a **bad-list** scheme, since it keeps track of the bad certificates.

A scheme that keeps track of the good certificates is more secure, however. Suppose a CA operator is bribed to issue a certificate, using a serial number from a valid certificate, and that no audit log indicates that this bogus certificate has been issued. Nobody will know this certificate

needs to be revoked, since no legitimate person knows it was ever issued. It will not be contained in the CRL.

Suppose instead that the CRL contains a list of all the valid certificates (and not just serial numbers, but hashes of the certificate for each serial number). Then the bogus certificate would not be honored, because it would not appear in the list of good certificates.

There are two interesting issues with good-lists:

- The good-list is likely to be much larger than the bad-list and might change more frequently, so performance might be worse than with a bad-list.
- An organization might not want to make the list of its valid certificates public. This is easily answered by having the published good-list contain *only* hashes of valid certificates, rather than any other identifying information.

Note that usually the good-list or bad-list, especially if publicly readable, will contain only serial numbers and hashes of the certificates rather than any other identifiable information. Then the only information divulged is the number of valid certificates (in the good-list case) or invalid certificates (in the bad-list case). There is no reason to believe that the count of good certificates is more security sensitive than the count of bad certificates. If for some reason the count was security sensitive, the revocation service could claim additional fictitious certificates as being valid or invalid.

The X.509 standard says that it is not permitted to issue two certificates with the same serial number and that all certificates issued must be logged. But we shouldn't assume that a bad guy would be hindered from issuing bogus, unaudited certificates just because it would violate the specification! The standard assumes that a CA will be run in such a way that nobody would be able to sneak in and have it create a certificate with a duplicate serial number. This could be enforced to a high probability with hardware.

10.9 OTHER INFORMATION IN A PKIX CERTIFICATE

Some of the fields in a certificate vouching for Bob's key are what you'd expect. SUBJECT NAME is where you'd expect to see Bob's name. The other obvious field in a certificate is the CA's signature. Given that there might be a lot of different signature algorithms, a signature needs to include both a SIGNATURE TYPE that would specify the signature algorithm, and the actual SIGNATURE. However, there are other fields in the PKIX certificate. Given that the PKIX format is extensible, new fields can always be added.

- SUBJECT PUBLIC KEY INFO. This is an important field. It specifies two things: the type of key being certified, *e.g.*, RSA key or ECDSA key, and the value of the key.

- VALIDITY INTERVAL. This specifies both a NotBefore and a NotAfter time in units of seconds. There is no IssuedTime. You might think that the NOTBEFORE field would be when the certificate was signed, but the PKIX format allows post-dating a certificate (signing it on Monday, but saying it will not be legal until Friday). Also, interestingly, there are two different representations of time, both represented as ASCII strings. The first representation is what is used for any dates up until 2050—YYMMDDHHMMSSZ, where YY is the last two digits of the year, MM is the month represented as two digits, DD is the day represented as two digits, HH is the hour in 24-hour time, MM is the minute, SS is the second, and Z is a constant, short for Zulu, which means what used to be called Greenwich Mean Time. Since there are only two digits in the year, this format originally would have had a Y2K problem in 2000. The time committee bought themselves an extra 50 years by saying that if the value of the two digits was greater than or equal to 50, the year would be 19YY, and if the digits were less than 50, the year would be 20YY. The time representation committee people could have assumed they'd all be retired by 2050, and left it up to some future generation to notice the problem and do something. Instead, the forward-looking visionaries came up with a different representation for dates after 2050. That representation is YYYYMMDDHHMMSSZ, where the year is four digits. So, the next panic for time representation won't come until the year 9999, when all us of will be retired, so we don't need to worry about it.
- USAGE RESTRICTIONS. This contains name constraints (see §10.6.6 *Name Constraints*). It also contains restrictions such as whether the key should only be used for encryption, for signing (and if so, what types of information it should be trusted to sign), or for authentication. There is a bit in usage restrictions that indicates whether the subject is allowed to be a CA. If it is allowed, then a certificate signed by the subject name can be trusted to be a link in a certificate chain. And the usage restrictions also allow specifying how long the chain from the subject is allowed to be.
- WHERE TO FIND REVOCATION INFORMATION. This indicates where revocation information for this certificate can be found.

10.10 ISSUES WITH EXPIRED CERTIFICATES

When a public key is used for real-time authentication, the only thing that matters is that the certificate is valid at the time it is being used. But suppose a public key is signing something. Should the signature remain valid if the certificate has expired? What if the key has been revoked?

Assuming a signature includes a date, it might be tempting to say that the signature should be valid if the key was valid at the time the document was signed. However, suppose the document is

one in which Alice signs over the deed to her house to Bob. At the time she signed it, her certificate was current, and it was not revoked. So Bob trusts Alice's signature on the deed. But then suppose ten years later, Alice says to Bob, "Why are you living in my house?" Bob shows her the deed that she signed ten years ago. Alice can then say, "I reported my key stolen last week. Whoever stole the key must have created that document and backdated it to ten years ago."

A solution is to have a third party, usually referred to as a *notary*, sign the document any time while Alice's key is still valid. The notary is attesting to the fact that Alice's key was valid at the time the notary signed the document. But the notary's key might also expire or get revoked. So, multiple notaries should sign the document. A notary can revalidate the document even after Alice's key has been revoked or has expired, provided that the new notary trusts a previous notary (and its key) that has signed the document. Even after Alice's key expires, or some of the notary keys have expired or been revoked, the signatures on the document can still be validated provided that at least one notary who has signed it still has a valid key.

10.11 DNSSEC (DNS SECURITY EXTENSIONS)

DNS names are hierarchical. The details of DNS are somewhat arcane, due to the history of getting it deployed or optimizing its performance. We will simplify the concepts in DNS (for instance, we will assume each name has one IP address, and there is one DNS server for each domain in the DNS hierarchy) and focus instead on the conceptual aspects of DNSSEC.

There is an online server associated with each DNS domain. The DNS infrastructure allows someone to look up things about a DNS name, such as its IP address. For example, information about the name `dell.com` would be found by querying the server responsible for storing information and answering queries about the domain consisting of names of the form `*.com`. There are lots of things DNS stores about a particular name in addition to the IP address, for example, the DNS information about the name `dell.com` would probably include the name of the email server that handles mail for email addresses of the form `user@dell.com`.

If Alice does not authenticate the DNS server when she looks up information about a name, something impersonating a DNS server can give her false information. Even if Alice were somehow to authenticate the DNS server she is querying, it is still better to minimize the amount of trust in an online server and have the information the online server sends you be digitally signed by something that is better protected and offline. For performance you would want many online servers that you could query about names of the form `*.com`, and it would be better not to require them all to be physically secured.

The main security features of DNSSEC are that it adds a public key to each name's information, and it enables information in DNS to be digitally signed, with those digital signatures stored in

DNS. The digital signature is preferably created by a physically secured offline entity, and the DNS server storing the information and answering queries about names in a domain will not know the private key for the domain.

DNSSEC is a bit more complicated than you'd expect because

- People want to allow information about some names in a domain to be signed, and others not to be signed. The probable motivation for this is so that the owner of a name could be charged by the domain manager for requesting that its information be digitally signed.
- People have decided that it should be difficult to figure out all the names in a domain. You are not allowed to ask for a list of names in the domain. You can ask for information about a specific name, and the reply will either be information for that name or that the specific name does not exist in the domain.

With the simplest imaginable design, if the information about a name was signed, the DNS server would return the signed information, and otherwise, return information and say "This was not signed." If that were the design, a dishonest DNS server could give you false information even if the information for the name was signed, because the DNS server can say "Here is the information, and it wasn't signed." Likewise, a dishonest DNS server could claim a name does not exist in the domain even if it does.

There are three cases that DNSSEC must accommodate:

- The name does not exist in the domain.
- The name exists, but the information for that name is not signed.
- The information for the name is signed.

DNSSEC has a clever mechanism for preventing a dishonest DNS server from lying. Associated with a domain (in which at least some of the entries are signed) is a list of hashes of names for which information is signed. This list of hashes is sorted numerically, *e.g.*, $h_1, h_2, h_3, \dots, h_n$, where $h_1 < h_2 < h_3 < \dots < h_n$. The DNS manager (which signs information for that domain) signs each adjacent pair of hashes, *e.g.*, it will sign $h_1|h_2$, and sign $h_2|h_3$, If a query is made for a particular name, and the information is signed, the DNS server returns the signed information. However, if that name does not exist or the information for that name is unsigned, the DNS server hashes the name, determines which hash range the name belongs in (*e.g.*, the hash of the name might fit between h_7 and h_8), and returns the signature on $h_7|h_8$. This proves that any name whose hash fits in that range either does not exist in the zone or is unsigned. A dishonest DNS server cannot trick you into believing incorrect information about an entry that was signed, but for entries that are not signed, a dishonest DNS server can claim the name doesn't exist, even if it does, or it can give you false information for the name, whether the name exists or not.

DNSSEC could work as a PKI. It would allow Alice to find Bob's key through querying online servers (unlike the way TLS and IPsec work today, which depend on Bob sending Alice its certificates when Alice tries to talk to Bob). DNSSEC is a top-down model, in that everyone is

configured with the key for the root domain, and by traversing through each child domain, you discover the key for that domain. In theory, it could implement the bottom-up model if a domain allowed an entry for the key for its parent (an up-link) or allowed an entry for a cross-link. The original version of DNSSEC allowed storing keys for server names, and it would be conceptually easy to have email and authentication keys for users at an organization to be stored in a DNS directory managed by that organization. Unfortunately, the DNSSEC committee removed the specification for how to represent any keys stored in DNS other than DNS signing keys for child directories. There are ongoing efforts to add them back in, at least in certain cases. DANE (DNS-based Authentication of Named Entities), RFC 6698, is one such effort.

10.12 HOMEWORK

1. What could a malicious CA do compared with a malicious KDC? Consider scenarios such as decrypting conversations between Alice and Bob or impersonating Bob to Alice.
2. In Protocol 10-1, the first message to the KDC “Alice wants to talk to Bob” is not cryptographically protected, so an active attacker could change the message to “Alice wants to talk to Trudy.” How does the protocol ensure that Alice will not be tricked into thinking Trudy is Bob?
3. Design a variant of Kerberos in which the workstation generates a TGT rather than asking the KDC for a TGT. Hint: The TGT will be encrypted with the user’s master key rather than the KDC’s master key. How does this compare with standard Kerberos in terms of efficiency, security,...?
4. Suppose there were two different services: TGT-server, that only gives out TGTs, and Ticket-server, that gives out tickets. Assume only human users get TGTs, other principals keep their master key and never get TGTs, and tickets are never granted to human principals. What information needs to be configured into the TGT-server? What information needs to be configured into Ticket-server?
5. Suppose all principals get TGTs and forget their master key, and the double-ticket protocol as described in §10.3.6 *Double TGT Protocol* is used. What information needs to be configured into the TGT-server? What information needs to be configured into the Ticket-server?
6. In §10.3.6 *Double TGT Protocol*, we say that Alice asks Bob to send her his TGT. If Alice knows Bob’s TGT, what prevents her from impersonating Bob?

7. Suppose to avoid disruption in case the KDC failed, a realm had several redundant KDCs. Do the KDC databases need to be synchronized when a new principal is added to or deleted from the realm? What about when a KDC creates a session key and TGT?
8. Suppose all Kerberos principals (not just humans) obtained session keys and TGTs so they could avoid keeping their master keys around. Design a system that allows Alice to get a ticket to Bob even though Bob has forgotten his master key and only remembers his session key and TGT, and the KDC does not keep track of session keys.
9. Why is the authenticator field not of security benefit when asking the KDC for a ticket for Bob but useful when logging into Bob?
10. Suppose Alice gets a ticket to Bob (which will be encrypted with Bob's master secret K_B). However, Bob has requested a session key and TGT and has forgotten K_B . What mechanism could a system like Kerberos use to enable Bob to find out the name of the principal contacting him ("Alice") and the shared secret K_{A-B} for communicating with Alice (K_{A-B})?
11. How could you use Kerberos for securing electronic mail? The obvious way is for Alice, when sending a message to Bob, to obtain a ticket for Bob and include that in the email message, and encrypt and/or integrity-protect the email message using the key in the ticket. The problem with this is that then the KDC would give Alice a quantity encrypted with Bob's password-derived master key, and then Alice could do off-line password guessing. How could you modify Kerberos to support email without allowing off-line password guessing? (Hint: Issue human users an extra, unguessable master key for use with mail, and extend the Kerberos protocol to allow Bob to safely obtain his unguessable master key from the KDC.)
12. Suppose human Alice is a principal in two different realms, say, Wonderland and Oz, and she wants to use the same password in each realm. How can she ensure that her master key in each realm is different?
13. To build the anarchy model with PKIX, what name constraints should go into a certificate? In a top-down model (where the only certificates are signed by a parent for a child), what name constraint should go into the certificate? To build the bottom-up model, what name constraint should go into the up-link certificate (where a child signs a certificate for its parent)? What name constraint should go into a cross-link?
14. In Figure 10-7 there are no up-links. What trust anchor would principals in `a.com` need to be configured with in order to reach all the principals in the figure?
15. In Figure 10-8, which CAs need to be trustworthy for the certificate path from principals in `sales.xyz.com` to principals in `a.org` to be secure? How about the path from principals in `a.org` to principals in `sales.xyz.com`?
16. Why must a CRL be reissued periodically, even when no new certificates have been revoked?
17. If there is a revocation mechanism, why do certificates need an expiration date?

18. Compare performance (*e.g.*, bandwidth, latency, timeliness of revocation) of various revocation schemes: relying on the expiration date in certificates, verifiers downloading complete CRLs, verifiers downloading delta CRLs, verifiers querying an online revocation server, principals obtaining “my certificate is still valid” certificates, and “first valid certificate” in a CRL. Consider factors such as a company suddenly laying off thousands of employees, long periods of time in which no certificates have been revoked, a verifier serving huge numbers of principals, or a principal visiting many services.
19. Why is it important in a good-list revocation scheme to keep hashes of the valid certificates, rather than just their serial numbers?

11

COMMUNICATION SESSION ESTABLISHMENT

*Knock Knock!
Who's there?
Alice.
Alice who?*

...and you'll have to read on to find secure ways of continuing...

This chapter analyzes various considerations when designing real-time communication handshakes. We start with very simple example handshakes that do authentication only (rather than also creating a session key and cryptographically protecting the data). These types of protocols are useful for simple scenarios, such as opening a door, and were common when people just wanted to replace sending a password in the clear with the least amount of effort. Even though most Internet communication today is done with TLS, it is still instructive to start with analysis of very simple handshakes. The second half of this chapter explains a lot of design considerations that went into the design of the handshakes for IPsec (Chapter 12 *IPsec*), TLS, and SSH (Chapter 13 *SSL/TLS and SSH*), widely deployed standards for doing authentication, establishing secret keys for cryptographically protecting a session, and sending cryptographically protected data.

Reminder of our notation: If Bob and Alice share a secret key we'll call that key K_{A-B} . The notation $f(K_{A-B}, R)$ means that some function f cryptographically transforms two inputs—the shared secret K_{A-B} and a challenge R . When we explicitly mean encryption, we'll write $\{R\}K_{A-B}$. When we explicitly mean a hash, we'll write $\text{hash}(K_{A-B}, R)$. K_{A-B} might be a high-quality key configured into both Alice and Bob, or, if Alice is a human, it is likely to be a low-quality secret derived from a password typed into her device.

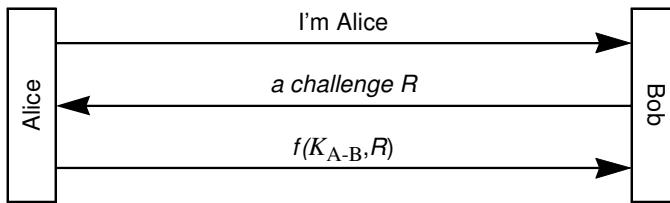
Also, as described in §9.8 *Off-Line Password Guessing*, a dictionary attack is where an attacker can capture some data that will allow them to verify whether a password is the user's password. Since it is offline, the number of guesses the attacker tries cannot be audited and is limited only by the compute power of the attacker and the time they have.

11.1 ONE-WAY AUTHENTICATION OF ALICE

A lot of older protocols were designed in an environment where eavesdropping was not a concern (rightly or wrongly), and bad guys were (rightly or wrongly) not expected to be very sophisticated. The authentication in such protocols generally consists of

- Alice (the initiator) sends her name and password (in the clear, *i.e.*, with no cryptographic protection) across the network to Bob.
- Bob (who has a password database consisting of usernames and passwords) verifies the name and password, and then communication occurs with no further attention to security—no encryption, no cryptographic integrity protection.

A common enhancement to such a protocol is to replace the transmission of the cleartext password with a cryptographic challenge/response. Consider Protocol 11-1.



Protocol 11-1. Bob Authenticates Alice Based on a Shared Secret K_{A-B}

An eavesdropper will see both R and $f(K_{A-B}, R)$. This protocol is a big improvement over passwords in the clear. However, there are some weaknesses to this protocol:

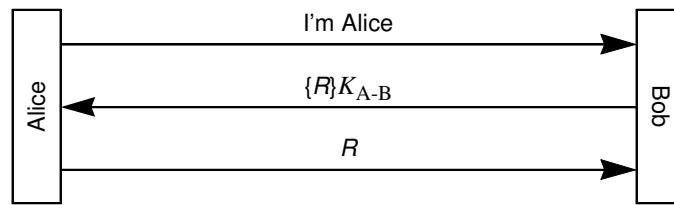
- Authentication is not mutual. Bob authenticates Alice, but Alice does not authenticate Bob. If Trudy can receive packets transmitted to Bob's network address and respond with Bob's network address (or through other means convince Alice that Trudy's address is Bob's), then Alice will be fooled into assuming Trudy is Bob. This is especially convenient for Trudy if she is on the path between Alice and Bob, *e.g.*, Trudy is malware on a router along the path. Trudy doesn't need to know Alice's secret in order to impersonate Bob—she just needs to send any old number R to Alice and ignore Alice's response.
- If this is the entire protocol (*i.e.*, the remainder of the conversation is transmitted without cryptographic protection), then Trudy can hijack the conversation after the initial exchange, assuming she can impersonate Alice's IP address, and Bob will assume he's talking to Alice. **Hijacking** the conversation means that Trudy steps in and starts communicating with Bob after Alice has completed the authentication to Bob. This is analogous to Trudy waiting for Alice to insert her ATM card and type her PIN at an ATM machine, and then pushing Alice away and withdrawing money for herself from Alice's account. In a network, assuming Trudy can impersonate the IP address from which Alice authenticated, Trudy can send some TCP

packets before Alice does, and then when Alice starts trying to continue her conversation, Bob will ignore Alice's packets because the sequence numbers Alice is using are smaller than what Trudy is using.

- An eavesdropper, seeing R and $f(K_{A-B}, R)$, can mount an off-line password-guessing attack (assuming K_{A-B} is derived from a password).
- Someone who steals the password database at Bob can impersonate Alice.

Note that if Mallory (he/him) has stolen Bob's password database, he will know K_{A-B} . If Mallory is able to modify the client software to bypass the step where the human types a password and the client device converts that into a key, then Mallory can directly impersonate Alice without needing to do a dictionary attack to find Alice's password. However, if Mallory cannot modify the client software, then he will need to do a dictionary attack using Bob's password database in order to find a password that will translate to K_{A-B} .

A minor variant on Protocol 11-1 is Protocol 11-2:



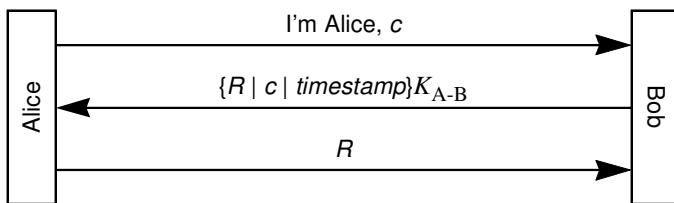
Protocol 11-2. Bob Authenticates Alice Based on a Shared Secret Key K_{A-B}

In this protocol Bob chooses a random challenge R , encrypts it, and transmits the result. Alice then decrypts the received quantity, using the secret key K_{A-B} to get R , and sends R to Bob. This protocol has only minor security differences from Protocol 11-1:

- Protocol 11-1 can be done using a hash function. But in Protocol 11-2, Alice has to be able to reverse what Bob has done to R in order to retrieve R .
- Suppose K_{A-B} is derived from a password and therefore vulnerable to a dictionary attack. If R is a recognizable quantity, for instance, a 96-bit random number padded with 32 zero bits to fill out an encryption block, then Trudy can, without eavesdropping, mount a dictionary attack by merely sending the message "I am Alice" and obtaining $\{R\}K_{A-B}$ from Bob. If Trudy is eavesdropping, however, and sees both R and $\{R\}K_{A-B}$, she can mount a dictionary attack with either protocol. It is often the case that eavesdropping is more difficult than sending a message claiming to be Alice. Kerberos V4 is an example of a protocol that allows someone to send a message claiming to be Alice and obtain a quantity that can be used to mount an offline dictionary attack. Kerberos V5 made it so that only an eavesdropper would be able to do an off-line dictionary attack.

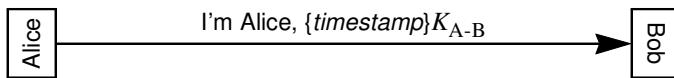
11.1.1 Timestamps vs. Challenges

If R is a recognizable quantity with a limited lifetime, such as a random number concatenated with a timestamp, Alice can somewhat authenticate Bob because generating $\{R\}K_{A-B}$ requires knowing K_{A-B} . However, if Alice attempts a new connection to Bob (within the acceptable lifetime of the timestamp), eavesdropper Trudy, (impersonating Bob's address, tricking Alice into connecting to Trudy), could replay $\{R\}K_{A-B}$ and trick Alice into thinking she is connecting (again) to Bob. A way to fix this weakness is to have Alice send a challenge c in her first message and have the quantity $(R|c|timestamp)$ be what Bob encrypts, as in Protocol 11-3.



Protocol 11-3. Mutual Authentication with Alice Sending a Challenge

Another variant on Protocol 11-1 is to shorten the handshake to a single message by having Alice use a timestamp (instead of an R that Bob supplies), as in Protocol 11-4:



Protocol 11-4. Timestamp-based Authentication of Alice, with a Shared Secret K_{A-B}

Using timestamps requires that Bob and Alice have reasonably synchronized clocks. Alice encrypts the current time. Bob decrypts the result and makes sure the result is acceptable (*i.e.*, within an acceptable clock skew, *e.g.*, ten minutes). The implications of this modification are

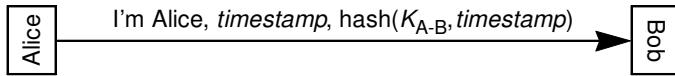
- Protocol 11-4 can be added very easily to a protocol designed for sending cleartext passwords, since it does not add any additional messages—it merely replaces the cleartext password field with the encrypted timestamp in the first message transmitted by Alice to Bob.
- The protocol is now more efficient. It goes beyond saving two messages. It means that a server, Bob, does not need to keep any volatile state (such as R in Protocol 11-1) regarding Alice (but see next bullet). This protocol can be added to a request/response protocol (such as RPC) by having Alice merely insert the encrypted timestamp into her request. Bob can authenticate the request, generate a reply, and forget the whole thing ever happened.
- Someone eavesdropping can use Alice's transmitted $\{timestamp\}K_{A-B}$ to impersonate Alice, if done within the acceptable clock skew. This threat can be foiled if Bob remembers all timestamps sent by Alice until they expire (*i.e.*, they are old enough that the clock skew check would consider them invalid). However, if there are multiple servers for which Alice uses the

same secret K_{A-B} , an eavesdropper who acts quickly can use the encrypted timestamp field Alice transmitted, and (if still within the acceptable time skew) impersonate Alice to a different server. This vulnerability can be fixed by concatenating the server name in with the timestamp, *e.g.*, Alice sends $\{\text{Bob}|\text{timestamp}\}K_{A-B}$. However, in the case where there are multiple instances of what appears to Alice to be a single server Bob, putting the name Bob into the encrypted timestamp will not help. Having all the instances of Bob coordinate the database of timestamps used by Alice within the acceptable timestamp window would help but would be very expensive, and it would be essential that the instances of Bob learn of a used timestamp faster than an eavesdropper could replay Alice's encrypted timestamp. Alternatively, there could be one main Bob instance that keeps track of used timestamps, and before an instance of Bob accepts an encrypted timestamp from Alice, he checks with the main Bob to ask if that timestamp has yet been used.

- Assuming Bob only bothers remembering used timestamps within the validity window (*e.g.*, ten minutes), if our bad guy Trudy can convince Bob to set his clock backwards, she can reuse encrypted timestamps she had overheard more than ten minutes ago (since Bob would have forgotten those). In practice there are systems that are vulnerable to an intruder resetting the clock. If the security protocols are not completely understood, it might not be obvious that clock-setting could be a serious security vulnerability.
- If security relies on time, then setting the time will be an operation that requires a security handshake. A handshake based on time will fail if the clocks are far apart. If there's a system with an incorrect time, then it may be difficult to log in to the system in order to manage it (*perhaps* to correct its clock). A plausible solution to this is to have a different authentication handshake based on challenge/response (*i.e.*, not dependent on time) for managing clock setting.

In Protocol 11-1, computing $f(K_{A-B}, R)$ may be done by encrypting R using K_{A-B} as a key or by concatenating K_{A-B} with R and doing a hash. When we're using timestamps the same is true, except for a minor complication. How does Bob verify that $\text{hash}(K_{A-B}, \text{timestamp})$ is acceptable? Suppose the timestamp is in units of minutes, and the believable clock skew is ten minutes. Then Bob would have to compute $\text{hash}(K_{A-B}, \text{timestamp})$ for each of the twenty possible valid timestamps to verify the value Alice sends (though he could stop as soon as he found a match). With a reversible encryption function, all he had to do was decrypt the quantity received and see if the result was acceptable. While checking twenty values might have acceptable performance, this approach would become intolerably inefficient if the clock granularity allows a lot more legal values within the clock skew. For instance, the timestamp might be in units of microseconds, in which case there would be 1.2 billion valid timestamps within a ten-minute clock skew. This would be unacceptably inefficient for Bob to verify. The solution (assuming you wanted to use a microsecond clock and a hash function rather

than a reversible encryption scheme) is to have Alice transmit the actual timestamp unencrypted, in addition to transmitting the hashed value, as in Protocol 11-5:



Protocol 11-5. Alice Sends the Timestamp in the Clear, As Well As $\text{hash}(K_{A-B}, \text{timestamp})$

11.1.2 One-Way Authentication of Alice using a Public Key

With protocols in the previous section, which are based on shared secrets, Trudy can impersonate Alice if she can read Bob's password database. If the protocols are based on public keys instead (where Bob's user database consists of public keys for each user), this can be avoided, as in Protocol 11-6.



Protocol 11-6. Bob Authenticates Alice Based on Her Public Key Signature

The notation $[R]_{\text{Alice}}$ means that Alice signs R using her private key. Bob will verify Alice's signature $[R]_{\text{Alice}}$ using Alice's public key. This is very similar to Protocol 11-1. The advantage of this protocol is that the public key database at Bob is no longer security-sensitive to an attacker reading it. Bob's public key database must be protected from unauthorized modification, but not from unauthorized disclosure. And, as before, the same minor variant works, as in Protocol 11-7:



Protocol 11-7. Bob Authenticates Alice if She Can Decrypt a Message Encrypted with Her Public Key

In this variant, Bob chooses R , encrypts it using Alice's public key, and Alice proves she knows her private key by decrypting the received quantity to retrieve R . Note that some public key schemes

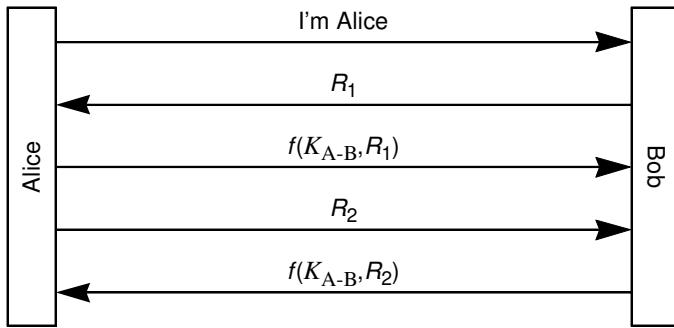
can only do signatures, not reversible encryption. This variant requires a public key scheme that can do reversible encryption.

In both Protocol 11-6 and Protocol 11-7, there is a potential serious problem. In Protocol 11-6, you can trick someone into signing something. Suppose Trudy has a quantity on which she'd like to forge Alice's signature (*e.g.*, the hash of a message that says "I agree to pay Trudy a million dollars"). If Trudy can impersonate Bob's network address and wait for Alice to try to connect, Trudy can give Alice that quantity as the challenge. Alice will sign it, and now Trudy knows Alice's signature on that quantity. Protocol 11-7 has Alice decrypting something. So, if there's a quantity encrypted with Alice's public key, such as the AES encryption key of a message, encrypted with Alice's public key, and Trudy wants to decrypt the message, Trudy can again impersonate Bob's address, wait for Alice to connect, and then give the encrypted quantity to Alice as the challenge.

How can we avoid getting in trouble? The general rule is that you should not use the same key for two different purposes unless the designs for all uses of the key are coordinated so that an attacker can't use one protocol to help break another. An example method of coordination is to ensure that R has some structure. For instance, if you sign different types of things (say an R in a challenge/response protocol versus an electronic mail message), each type of thing should have a structure so that it cannot be mistaken for another type of thing. For example, there might be a type field concatenated to the front of the quantity before signing, with different values for *authentication challenge* and *mail message*. PKCS #1 (see §6.3.6) defines enough structure to distinguish between using an RSA key for signing and for encryption. Note that merely having a specification doesn't magically cause all implementations to be careful. There are software implementations that do not do all the checks and might be able to be tricked into signing or decrypting something that does not have the appropriate structure. Also, PKCS #1, unfortunately, does not distinguish between different uses of signing, *e.g.*, signing as part of an authentication handshake *vs.* signing a message, or different uses of encryption. The basic idea of PKCS #1 is that typically an RSA key is used to sign the hash of something. A hash is likely to be at most 512 bits, and an RSA key is likely to be about 4096 bits, so there is plenty of room to encode extra information in the RSA-keysized block to be signed. And, likewise, when an RSA key is used for encryption, it is likely to just be encrypting a secret key that is at most 512 bits, again, leaving plenty of room in the padding of the block to encode extra information.

11.2 MUTUAL AUTHENTICATION

Suppose we want to do mutual authentication so both Alice and Bob will know who they are talking to. We could just do an authentication exchange in each direction, as in Protocol 11-8:



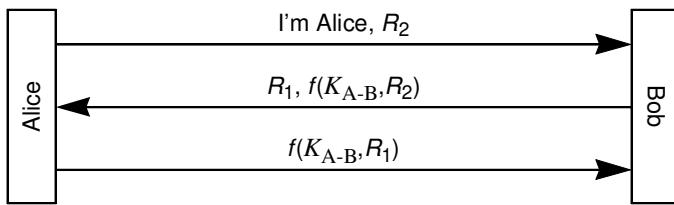
Protocol 11-8. Mutual Authentication Based on a Shared Secret K_{A-B}

11.2.1 Reflection Attack

"I can't explain myself, I'm afraid sir," said Alice, "because I'm not myself, you see."

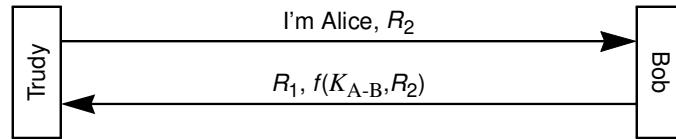
—Alice in Wonderland

The first thing we might notice is that Protocol 11-8 is inefficient. We can eliminate two messages by putting more than one item of information into each message, as in Protocol 11-9:

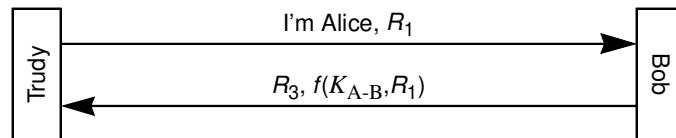


Protocol 11-9. Optimized Mutual Authentication Based on a Shared Secret K_{A-B}

Protocol 11-9 has a security pitfall known as the **reflection attack**. Suppose Trudy wants to impersonate Alice to Bob. First Trudy starts Protocol 11-9, but when she receives the challenge from Bob in the second message, she cannot proceed further, because she can't encrypt R_1 :



However, note that Trudy has managed to get Bob to encrypt R_2 . So at this point, Trudy opens a second session to Bob. This time she uses R_1 as the challenge to Bob:



Trudy can't go any further with this session, because she can't encrypt R_3 . But now she knows $f(K_{A-B}, R_1)$, so she can continue the first session to complete Protocol 11-9 and impersonate Alice.

This is a serious security flaw, and there are deployed protocols that contain this flaw. In many environments it is easy to exploit this, since it might be possible to open multiple simultaneous connections to the same server, or there might be multiple servers with the same secret for Alice (so Trudy can get a different server to compute $f(K_{A-B}, R_1)$ so that she can impersonate Alice to Bob).

We can foil the reflection attack if we are careful and understand the pitfalls. Here are two methods of fixing the protocol, both of which are derived from the general principle *don't have Alice and Bob do exactly the same thing*:

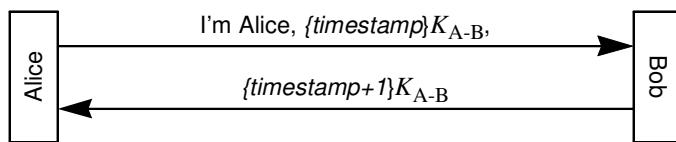
- different keys—Have the key used to authenticate Alice be different from the key used to authenticate Bob. We could use two totally different keys shared by Alice and Bob at the cost of additional configuration and storage. Alternatively, we could derive the key used for authenticating Bob from the key used to authenticate Alice. For instance, Bob's key might be $K_{A-B} + 1$, or $-K_{A-B}$, or $K_{A-B} \oplus F0F0F0F0F0F0F0F0_{16}$. Any of these would foil Trudy in her attempt to impersonate Alice to Bob since she would not be able to get Bob to encrypt anything using Alice's unmodified key, and we mention these examples because these choices have been used in protocols such as PEM (an early secure email standard). But some cryptographic algorithms have what is known as **related key weaknesses**, where if an attacker can see ciphertext encrypted with keys with simple known mathematical relationships (as in our examples), the workfactor for cryptanalyzing the key is less expensive than brute force. So, a preferable scheme for modifying K_{A-B} is hash(*constant*, K_{A-B}).
- different challenges—Insist that the challenge from the initiator (Alice) look different from the challenge from the responder. For instance, we might require that the initiator challenge be an odd number and the responder challenge be an even number. Or the party encrypting a challenge might concatenate their name to the challenge before encryption, e.g., if the challenge from Alice to Bob was R , Bob would encrypt $\text{Bob}|R$.

Notice that Protocol 11-8 did not suffer from the reflection attack. The reason is that it follows another good general principle of security protocol design: *The initiator should be the first to prove its identity*. Ideally, you shouldn't prove your identity until the other side does, but since that wouldn't work, the assumption is that the initiator is more likely to be the bad guy.

...if you only spoke when you were spoken to, and the other person always
waited for you to begin, you see nobody would ever say anything...
—Alice (in *Through the Looking Glass*)

11.2.2 Timestamps for Mutual Authentication

We can reduce the mutual authentication down to two messages by using timestamps instead of random numbers for challenges, as in Protocol 11-10:

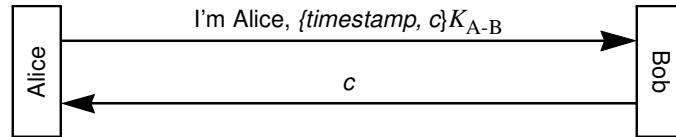


Protocol 11-10. Mutual Authentication Based on Synchronized Clocks and a Shared Secret K_{A-B}

This two-message variant is very useful because it is easy to add onto existing protocols (such as request/response protocols), since it does not add any additional messages. But it has to be done carefully. In Protocol 11-10 we have Bob encrypt a timestamp larger than Alice's timestamp. Obviously, Bob can't send the same encrypted timestamp back to Alice, since that would hardly be mutual authentication. (Alice would be assured that she was either talking to Bob or someone smart enough to copy a field out of her request!) So in the exchange, Alice and Bob have to encrypt different timestamps, use different keys for encrypting the timestamp, concatenate their name to the timestamp before encrypting it, or use any other scheme that will cause them to be sending different things. And the issues involved with the one-way authentication done with timestamps apply here as well (time must not go backwards, they must remember values used within the clock skew, and so on).

Note that any modification to the timestamp would do. We use $timestamp+1$ in our example because that's what Kerberos V4 uses, but $timestamp+1$ is probably not the best choice. Incrementing $timestamp$ has the potential problem that Trudy eavesdropping could use $\{timestamp+1\}K_{A-B}$ to impersonate Alice (unless Bob remembers both $timestamp$ and $timestamp+1$ as having been used). A better choice would be concatenating a value with the timestamp indicating whether the initiator or responder is transmitting.

Yet another variant that has the advantages of Protocol 11-10, but is simpler, has Alice concatenating a challenge c with the timestamp and Bob returning c , as in Protocol 11-11:



Protocol 11-11. Alice Encrypts a Timestamp and a Challenge; Bob Decrypts and Returns the Challenge

We haven't found a deployed protocol that uses this variant.

11.3 INTEGRITY/ENCRYPTION FOR DATA

Even if Alice and Bob have been configured with a long-term shared secret K_{A-B} , it is better to encrypt each conversation with a different secret key rather than cryptographically protecting the data with K_{A-B} . This per-conversation key is known as a **session key**. So we'll want to enhance the authentication exchange so that after the initial handshake, both Alice and Bob will share a session key.

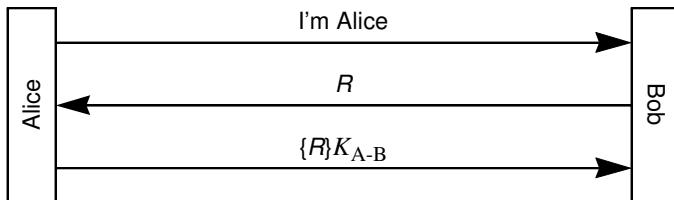
In a communication session, a sequence number is typically used to prevent replay or reordering of packets. A new session key should be established for each new session as well as during a session if the sequence number might be reused (*i.e.*, wrap around). Changing to a new key, because the previous key has been used for too much time or because too much data has been encrypted with the old key, is known as **key rollover**.

A good security rule is that both communicating parties should contribute to the session key. One way to do this is by having each side send a value encrypted with the other side's public key, and then using a hash of the two values as the session key. This rule makes it less likely that the protocol will have flaws in which someone will be able to impersonate one side in a replay attack. For instance, in a protocol in which Alice sends the session key to Bob, encrypted with his public key, someone impersonating Alice can simply replay all of Alice's messages from a previously recorded session.

With our example of using a hash of the values sent by the two sides, if *either* side has a good random number generator, then the session key will be sufficiently random. Note that this is not true with every scheme in which both sides contribute. For example, in a Diffie-Hellman exchange, if one side sends $g^1 \bmod p$ as their Diffie-Hellman value, the resulting Diffie-Hellman key will not be at all secure, no matter how great the other side's random number generator is.

11.3.1 Session Key Based on Shared Secret Credentials

Alice and Bob have a shared secret key K_{A-B} . The authentication exchange is shown in Protocol 11-12. Perhaps mutual authentication was done, in which case there are two challenges: R_1 and R_2 .

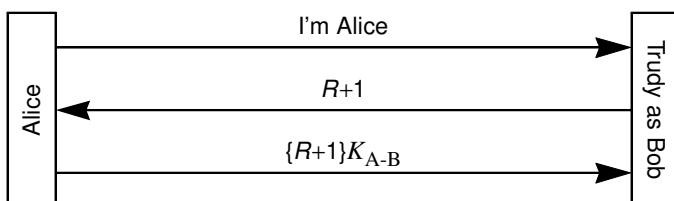


Protocol 11-12. Authentication with Shared Secret

At any rate, there is sufficient information in this protocol so that Alice and Bob can establish a shared session key at this point in the conversation. They can, for example, use $\text{hash}(R, K_{A-B})$.

There are some protocols that modify K_{A-B} and use that to encrypt R , for instance, using $\{R\}(K_{A-B}+1)$ as the session key. Why would such protocols need to modify K_{A-B} ? Why shouldn't they use $\{R\}K_{A-B}$ as the key? The reason they shouldn't use $\{R\}K_{A-B}$ is that $\{R\}K_{A-B}$ is transmitted by Alice as the third message in the authentication handshake, so an eavesdropper would see that value, and it certainly would not be secure as a session key.

How about using $\{R+1\}K_{A-B}$ as the session key? There's a more subtle reason why that isn't secure. Suppose Alice and Bob have started a conversation in which Bob used R as the challenge. Perhaps Trudy recorded the entire Alice-Bob conversation, encrypted using $\{R+1\}K_{A-B}$. Later, Trudy can impersonate Bob's network layer address to Alice, thereby tricking Alice into attempting to communicate with Trudy instead of Bob, and Trudy (pretending to be Bob) can send $R+1$ as a challenge, to which Alice will respond with $\{R+1\}K_{A-B}$:



Then Trudy will be able to decrypt the previous Alice-Bob conversation because she will see $\{R+1\}K_{A-B}$. She won't be able to continue the current conversation with Alice (because Trudy will not know $\{R+2\}K_{A-B}$), but Trudy will know $\{R+1\}K_{A-B}$, which is the session key for the Alice-Trudy conversation.

So, Alice and Bob, after the authentication exchange, know K_{A-B} and R , and there are many combinations of the two quantities that would be perfectly acceptable as a session key, but there are also some that are not acceptable. What makes a good session key? It must be different for each

session, unguessable by an eavesdropper, and not be a quantity X encrypted with K_{A-B} , where X is a value that can be predicted or extracted by an intruder (as just discussed for $X = R+1$). See Homework Problem 1.

11.3.2 Session Key Based on Public Key Credentials

Suppose we are doing two-way authentication using public key technology, so that Alice and Bob know their own private keys and know each other's public keys. How can they establish a session key? We'll discuss various possibilities, with their relative security and performance strengths and weaknesses.

1. One side, say, Alice, could choose a random number R_1 to be used as the session key, encrypt R_1 with Bob's public key, and send $\{R_1\}_{Bob}$ to Bob as an extra field in one of the messages in the authentication exchange. This scheme has a security flaw. Since we are assuming there is no integrity protection on the initial authentication exchange, our intruder, Trudy, could hijack the conversation by picking her own random number R_T , encrypt it with Bob's public key, and replace Alice's $\{R_1\}_{Bob}$ with $\{R_T\}_{Bob}$ in the message to Bob. Then Trudy (acting as a MITM; see Figure 1-1) will be able to communicate with Bob using R_T as the session key. Bob will think he's talking to Alice. Alice won't be able to talk to Bob after the initial exchange, but she'll assume the network or Bob is flaky and try again.
2. Alice could, in addition to encrypting R_1 with Bob's public key, sign the result. So she'd send $[\{R_1\}_{Bob}]_{Alice}$ to Bob. Bob would take the received quantity, first verify Alice's signature using Alice's public key, and then use his private key on the result to obtain R_1 . If Trudy were to attempt the same trick as in item 1, namely choosing her own R_T and sending it to Bob, she wouldn't be able to forge Alice's signature on the encrypted R_T . This scheme has a minor security weakness that can be fixed partially (in 3, below) or completely (in 4, below). The flaw is that if Trudy records the entire Alice-Bob conversation, and then later breaks into Bob and learns Bob's private key, she will be able to decrypt the conversation she'd recorded.
3. This is like 2, above, but Alice picks R_1 and Bob picks R_2 . Alice sends $\{R_1\}_{Bob}$ to Bob. Bob sends $\{R_2\}_{Alice}$ to Alice. The session key will be $R_1 \oplus R_2$. Breaking into Alice and stealing Alice's private key will enable Trudy to retrieve R_2 . Breaking into Bob and stealing Bob's private key will enable Trudy to retrieve R_1 . But in order to retrieve $R_1 \oplus R_2$, she'll need to break into both of them. (This is assuming that Trudy only manages to break into Alice and Bob after their conversation has terminated, and Alice and Bob forget R_1 , R_2 , and $R_1 \oplus R_2$ when the conversation is over.) Note that in item 2 we had Alice sign her quantity (*i.e.*, she sent $[\{R_1\}_{Bob}]_{Alice}$ instead of merely $\{R_1\}_{Bob}$). Why isn't it necessary for Bob and Alice to sign their quantities here? See Homework Problem 6.

-
4. Alice and Bob can do a Diffie-Hellman key exchange where each signs the Diffie-Hellman value they are sending (Alice transmits $[g^{R_A} \text{ mod } p]_{\text{Alice}}$; Bob transmits $[g^{R_B} \text{ mod } p]_{\text{Bob}}$). In this scheme, even if Trudy breaks into both Alice and Bob, she won't be able to decrypt recorded conversations because she won't be able to deduce either R_A or R_B . This property is known as **perfect forward secrecy** (PFS) and will be discussed further in §11.8 *Perfect Forward Secrecy*. Note this is not the only way to achieve PFS.

11.3.3 Session Key Based on One-Party Public Keys

In some cases, only one of the parties in the conversation has a public/private key pair. Commonly, as in the case of TLS, it is assumed that servers will have public keys, and clients will not bother obtaining keys and certificates. Cryptographic authentication is one way. The protocol assures the client that she is talking to the right server Bob, but if Bob wants to authenticate Alice, after the cryptographic session is established, Alice will send a name and password. Here are some ways of establishing a shared session key in this case.

1. Alice could choose a random number R , encrypt it with Bob's public key, send $\{R\}_{\text{Bob}}$ to Bob, and R could be the session key. A weakness in this scheme is that if Trudy records the conversation and later breaks into Bob and steals his private key, she can decrypt the conversation.
2. Bob and Alice could do a Diffie-Hellman exchange where Bob signs his Diffie-Hellman quantity. Alice can't sign hers because she doesn't have a public key. This achieves perfect forward secrecy even though only Bob signs his Diffie-Hellman value.

Note that neither of these schemes assures Bob he's really talking to Alice, but in either scheme Bob is assured that the entire conversation is with a single party. Bob can identify that single party as Alice by, for instance, having Alice send Bob a password or other shared secret encrypted with the session key.

11.4 NONCE TYPES

A **nonce** is a quantity any given user of a protocol uses only once. Many protocols use nonces, and there are different types of nonces with different sorts of properties. It is possible to introduce security weaknesses by using a nonce with the wrong properties. Various forms of nonce are a timestamp, a large random number, or a sequence number. What's different about these quantities? A large random number tends to make the best nonce, because it cannot be guessed or predicted (as

can sequence numbers and timestamps). This is somewhat unintuitive, since non-reuse is only probabilistic. But a random number of 128 bits or more has a negligible chance of being reused. A timestamp requires reasonably synchronized clocks. A sequence number requires nonvolatile state (so that a node can be sure it doesn't use the same number twice even if it crashes and restarts between attempts). When are these properties important?

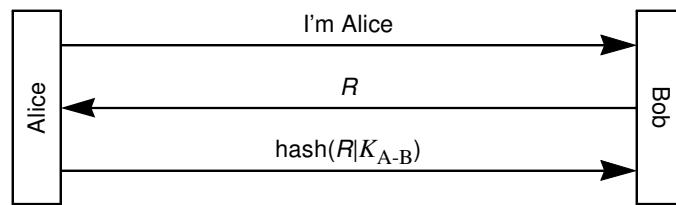
Protocol 11-13 is a protocol in which the unpredictability of the challenge is important.



Protocol 11-13. Protocol in Which R Must Be Unpredictable

Suppose Bob is using sequence numbers, so when Alice attempts to log in, Bob encrypts the next sequence number and transmits it to Alice who decrypts the challenge and transmits it to Bob. Suppose our eavesdropper, Eve, watches Alice's authentication exchange and sees Alice return an R of 7482. If Eve knows Bob is using sequence numbers for the challenge, she can then claim to be Alice, get an undecipherable pile of bits from Bob (the encrypted challenge), and return 7483. Bob will be suitably impressed and assume he's talking to Alice. So it is obvious in this protocol that Bob's challenge has to be unpredictable.

How about if we do it the other way, *i.e.*, make Alice compute a function of R and the shared key, as in Protocol 11-14? Must the challenge then be unpredictable?



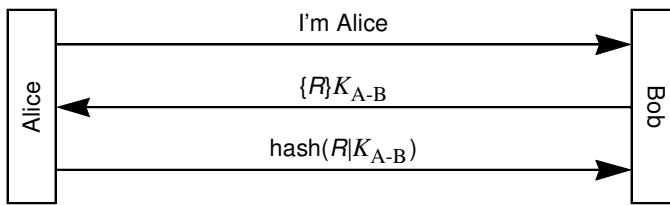
Protocol 11-14. Another Protocol in Which R Must Be Unpredictable

Suppose again that Bob is using sequence numbers. Eve watches Alice's authentication exchange and sees that R is 7482. Then Eve lies in wait, impersonating Bob's network address, hoping to entrap Alice into authenticating herself to Eve. When she does, Eve sends her the challenge 7483, and Alice will return the function of 7483 and the shared key. Now Eve can impersonate Alice to Bob, since Bob's challenge will be 7483, and Eve will know the answer to that challenge.

These protocols are also insecure if timestamps are used. Eve has to guess the timestamp Bob will use, and she might be off by a minute or two. If the timestamp has coarse granularity, say sec-

onds, Eve has a good chance of being able to impersonate Alice. But if the timestamp has nanosecond granularity, then it really does become just like a random number and the protocol is secure.

Here's Protocol 11-15, in which it would be perfectly secure to use a predictable nonce for R :



Protocol 11-15. Protocol in Which R Need Not Be Unpredictable

Even if Eve could predict what R would be, she can't predict either the value sent by Bob or the appropriate response from Alice.

11.5 INTENTIONAL MITM

We introduced MITM attacks in §1.6 *Active and Passive Attacks* and §6.4.1 *MITM (Meddler-in-the-Middle) Attack*. Sometimes MITM is not an attack, but it is intentionally deployed. Some companies intentionally deploy a proxy (let's call it Fred) that acts as a MITM for communication between their employees and external websites. There are many ways Fred might work, but an example is that the corporate IT department configures all the employee devices with a trust anchor (see §10.4 *PKI*) with a public key belonging to Fred. When Alice (inside the corporate network) tries to connect to Bob (an external site), Fred intercepts the request from Alice, creates a certificate for "Bob", signed by Fred, with a key known to Fred and completes the TLS session with Alice. Alice will believe that the certificate newly created by Fred is valid because Fred is one of her trust anchors. Fred will open a second TLS connection to Bob and relay messages between Alice and Bob. If the only thing Fred does is forward messages between Alice and Bob, that wouldn't be very useful. Fred can offer services such as having a deny-list of dangerous sites to prevent employees from connecting to them, save bandwidth and lower latency for Alice by caching web images that have been fetched by other users and send cached images to Alice, scan the data stream for viruses, or check whether employees are sending confidential information.

Note this form of Fred works when only Bob has a certificate. If Alice also has a certificate (client authentication), this will not work. All the devices managed by the company can be configured by the IT department to have Fred as a trust anchor (so that Fred can impersonate Bob), but the

world is not going to allow a company to install their trust anchor at all the servers, which would enable Fred to create certificates to impersonate all users.

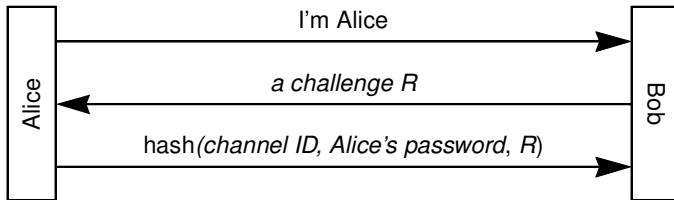
There might be some external websites for which Fred is configured not to act as a MITM and to instead allow direct connections to those websites. For example, if Alice is doing business at a bank, Alice's company would prefer not to see her banking information for liability reasons. And the bank would be a site that the company trusts not to be doing anything malicious. So for those sites, Fred is configured to not impersonate Bob. Instead, Fred just forwards messages from Alice to Bob without decrypting and re-encrypting. He does not create a certificate for Bob. He does not complete the TLS session with Alice and really just acts like a router. There will only be a single TLS connection in this case—from Alice to Bob—and Fred will not be able to see what Alice and Bob are talking about.

11.6 DETECTING MITM

In some cases, there is a way for Alice and Bob to detect a MITM. Suppose Alice and Bob are humans, communicating using special telephones. The telephones start a call by doing an anonymous Diffie-Hellman exchange. They also have a display that displays a small hash of the session key. Alice and Bob read aloud the value they see displayed. If there were a MITM, Alice's session key would be different from Bob's key. This assumes that the humans Alice and Bob recognize each other's voices. This technique—providing channel users with a channel identifier that would be different at the two endpoints if there were a MITM—is known as **channel binding**.

Channel binding can be used without depending on humans recognizing voices. Suppose there is a layer (*e.g.*, IPsec or TLS) that does anonymous Diffie-Hellman to create an encrypted tunnel, but perhaps with a MITM. Assume that layer has an interface in which it can inform an upper layer of some sort of session identifier, such as a hash of the session key. So, after an anonymous Diffie-Hellman with Trudy in the middle, the application at Alice's computer will be informed that the channel identifier is $\text{hash}(g^{at} \bmod p)$, and the application at Bob's computer will be informed that the channel identifier is $\text{hash}(g^{tb} \bmod p)$. If Alice merely sent a message saying “I think the channel identifier is $g^{at} \bmod p$ ”, Trudy could replace that field in the message with $g^{tb} \bmod p$.

However, suppose that layer has some sort of credential for Alice, for instance, a password that Bob has stored for Alice. In that case, that credential, along with the channel identifier, can be used cryptographically in a way that Trudy cannot impersonate. For example, see Protocol 11-16, which foils a MITM.

**Protocol 11-16.** Detecting a MITM Using Channel Binding

11.7 WHAT LAYER?

Definition of a layer n protocol: anything defined by a committee whose charter is to define a layer n protocol. —Radia Perlman (in *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*)

The concept of layers for network protocols (see §1.8.1 *Network Layers*) comes from the OSI (Open Systems Interconnection) model. Although the OSI 7-layer model is a useful way of learning about network protocols, implementations seldom conform to that model. TLS and SSH (see Figure 11-17) are said to be “implemented at layer 4”, whereas IPsec is said to be “implemented at layer 3”. What does it mean that a real-time communication security protocol is implemented at layer 3 vs. layer 4, and what are the implications?

Many IP stacks are implemented so that layer 4 (*e.g.*, TCP) and below are implemented in the operating system, and anything above is implemented in a user process. The philosophy of TLS is that it is easier to deploy something if you don’t have to change the operating system, so these protocols act “above TCP”. It requires the applications to interface to TLS instead of TCP and requires no changes to TCP. TLS is really a new version of SSL, or Secure Sockets Layer, whose name comes from the most popular API (Application Programming Interface) to TCP, which is called “sockets”. The API to TLS is a superset of the API to TCP sockets. Modifying an application to work on top of TLS requires minimal changes, but the security benefits are limited if the richer API is not used.

Although with TLS (and SSH) the applications have to be modified (albeit minimally), the operating system, which includes TCP and the layers below it, does not need to be modified. “Transport Layer Security” is somewhat of a misnomer because rather than being at layer 4, these protocols tend to act like a layer on top of layer 4.

In contrast, the philosophy behind IPsec is that implementing security within the operating system automatically causes all applications to be protected without the applications having to be modified.

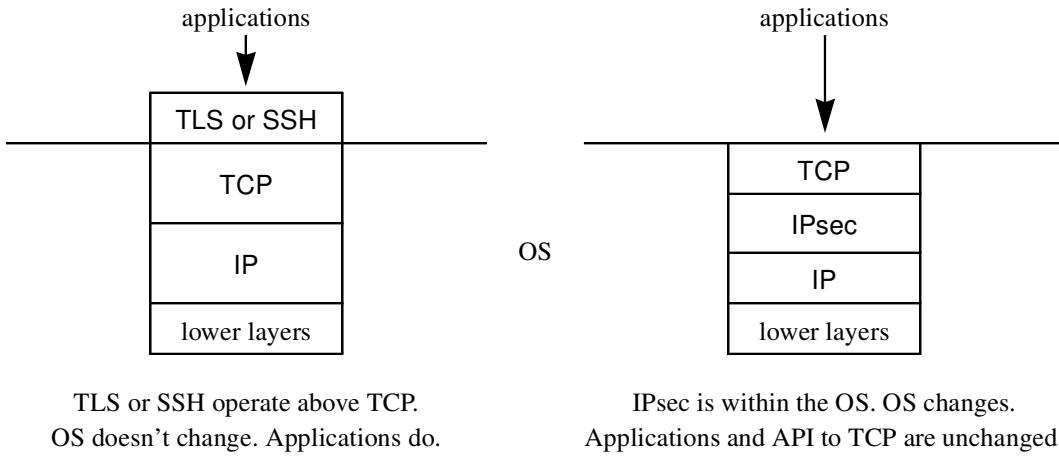


Figure 11-17. Implementing at “Layer 3” vs. “Layer 4”

There is a problem in operating above TCP. Since TCP will not be participating in the cryptography, it will have no way of noticing if malicious data is inserted into the packet stream, as long as the bogus data passes the (noncryptographic) TCP checksum and has the correct TCP sequence numbers (which is likely to require some eavesdropping by the attacker). TCP will acknowledge such data and send it up to TLS. The bogus data will fail the TLS integrity check, so TLS will discard the data, but there is no way for TLS to tell TCP to accept the real data at this point. When the real data arrives, TCP will assume it is duplicate data and discard it, since it will have the same TCP sequence numbers as the bogus data. TLS has no choice but to close the connection, since it can no longer provide the service that the API claims TLS offers, namely a lossless stream of cryptographically protected data. An attacker can launch a successful denial-of-service attack by inserting a single packet into the data stream. IPsec’s approach of cryptographically protecting each packet independently can better protect against such an attack.

In contrast, IPsec’s constraint of not modifying the applications winds up with its own serious problem. With the current commonly used API, IP tells the application only what IP address it is talking to, not what user is on the other end. So IPsec (with the current API) cannot tell the application anything more than which IP address is on the other end, even though IPsec is capable of authenticating an individual user. Most applications need to distinguish between users. If IPsec has authenticated the user, it could in theory tell the application the user’s name, but that would require changing the API and the application.

Implementing IPsec without changing the application has the same effect as putting firewalls between the two systems and implementing IPsec between the firewalls. It accomplishes the following:

- It causes the traffic on the path between the communicating parties to be encrypted, hiding it from eavesdroppers.
- As with firewalls, IPsec can access a policy database similar to what a firewall can access. For instance, it can specify which IP addresses are allowed to talk to which other IP addresses, or which TCP ports should be allowed or excluded. This is true whether the endpoint of the IPsec connection is a firewall or an endnode.
- Some applications do authentication based on IP addresses (see §9.2 *Address-based Authentication*). The API informs the application of the IP address from which information was received. Before IPsec, the source IP address was assumed based solely on the value of the SOURCE ADDRESS field in the IP header. With IPsec, address-based authentication becomes much more secure because one of the types of endpoint identifiers IPsec can authenticate is an IP address.

What IPsec (with the current API and an unmodified application) does *not* accomplish for the application is authentication of anything other than IP addresses. Most principals would have some identity, such as a name, and be allowed to access the network from a variety of IP addresses. In these cases, the most likely scenario for using IPsec is that IPsec would do its highly secure and expensive authentication, authenticating based on the user's public key and establishing a secure session to the user's name, but would have no way of telling the application who is on the other side. The application would have to depend on existing mechanisms, most likely a name and password, to determine which user it is talking to. IPsec is still of value in this scenario where the (unmodified) application authenticates the user based on name and password, since the name and password will now be encrypted when transmitted.

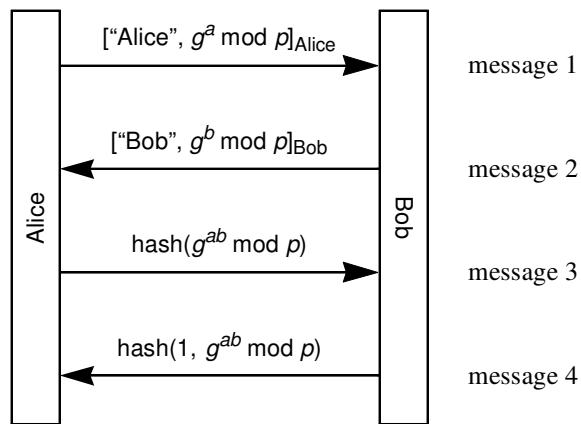
To take full advantage of IPsec, applications will have to change. The API has to change in order to pass identities other than IP addresses, and the applications have to change to make use of this information. So the best solution is one where both the operating system *and* the applications are modified. This illustrates why security would best be done by being designed in from the start, rather than being added in with the least modification to an existing implementation.

One other advantage of the packet-by-packet cryptographic handling of IPsec is that it is easier to build a network adapter that does IPsec processing. To implement a TLS-type protocol that operates over TCP in such a device, it would have to implement TCP, including buffering out-of-order packets.

DTLS (Datagram Transport Layer Security, RFC 6347) does not require TCP. Rather it will work over a datagram protocol such as UDP. This gives DTLS the best of both worlds—IPsec's invulnerability to connections breaking due to an attacker injecting bogus packets, and TLS's ability to assure an application of the authenticated identity of the endpoint. IPsec could, in theory, be modified to have the best of both worlds as well, by modifying the API so it could pass up the authenticated identities.

11.8 PERFECT FORWARD SECRECY

A protocol is said to have **perfect forward secrecy** (PFS) if it is impossible for an eavesdropper Mallory to decrypt a conversation between Alice and Bob even if Mallory records the entire encrypted session, and then subsequently breaks into *both* Alice and Bob and steals their long-term secrets. The trick to achieving perfect forward secrecy is to generate a temporary session key, not derivable from information stored at the node after the session concludes, and then forget it after the session concludes. If the session will last for a long time, it is common to generate and forget keys periodically so that even if Mallory seizes Alice's and Bob's computers while the conversation is still going on, he will not be able to decrypt messages received before the last key rollover. Protocol 11-18 is an example of a protocol with perfect forward secrecy. It uses Diffie-Hellman to agree on a session key, which achieves perfect forward secrecy assuming both sides generate an unpredictable Diffie-Hellman private number and forget both the private number and the agreed-upon session key after the session ends.



Protocol 11-18. Diffie-Hellman for Perfect Forward Secrecy Using Signature Keys

In the first two messages, each side identifies itself and supplies a Diffie-Hellman value signed by its private key. In the next two messages, each side proves knowledge of the agreed-upon Diffie-Hellman value $g^{ab} \text{ mod } p$ by sending a hash of it, with each side sending a different hash. If each side forgets $g^{ab} \text{ mod } p$ and its private Diffie-Hellman number (a or b) after the session, there is no way for anyone to reconstruct $g^{ab} \text{ mod } p$ from knowledge of both long-term private keys and the entire recorded conversation.

- What kind of protocol would not have perfect forward secrecy? Examples include
- Alice sends all messages for Bob encrypted with Bob's public key, and Bob sends all messages for Alice encrypted with Alice's public key.

- Kerberos (since the session key is inside the ticket to Bob and is encrypted with Bob's long-term secret).
- Alice chooses the session key, and sends it to Bob, encrypted with Bob's public key.

Perfect forward secrecy might seem like it only protects against a fairly obscure threat. However, protocols designed with perfect forward secrecy usually have another property, which is particularly popular with the IETF crowd, which we'll call **escrow-foilage**. This means that even if the forces of darkness (and we make no value judgments here) have required Alice and Bob to give their long-term private keys to some benign, completely trustworthy organization, the conversation between Alice and Bob will still be secret between only Alice and Bob. In other words, even with prior knowledge of Alice and Bob's long-term keys, a passive eavesdropper cannot decrypt the conversation.

Of course, if Mallory has prior knowledge of all of Alice's and Bob's secrets, then he can impersonate Alice or Bob and perhaps trick them into divulging what they would have divulged in the conversation. Maybe you'd think Alice and Bob could start off the conversation by asking each other a few personal questions like, "What café did we meet at in Paris?" but Mallory could be acting as a MITM, decrypting and re-encrypting the traffic, relaying the personal questions and answers, and this would be very difficult to detect.

Anything with perfect forward secrecy will also have escrow-foilage against a passive attack, since anything you can do by having recorded the conversation and learned the secret later you can also do knowing the secrets in advance and eavesdropping in real-time. But often with escrowed systems a user has two public key pairs, one for encryption and one for signatures. And in those cases, only the encryption key is escrowed, since law enforcement would like to decrypt data but does not need the ability to forge signatures. It would be counterproductive for them to have a user's private signature key, because then the user can repudiate his signature, claiming someone else with access to his key signed the message. So assuming the signature keys are not escrowed, then Protocol 11-18 will have escrow-foilage even against active attacks.

There are other ways of achieving PFS. An **ephemeral public key** is a public key created for short-term use, and the private key is then forgotten. For example, Alice or Bob could create an ephemeral RSA public key pair, sign the ephemeral public key with their long-term RSA key, and the session key could be sent encrypted with the ephemeral public key. This strategy was actually used in SSL for exportability reasons. At the time, an implementation was not exportable if it used a long RSA key for encryption, but it was okay to have a long (and therefore secure) RSA key for signing. So SSL allowed creating a shorter ephemeral RSA key pair, signing the ephemeral public key with the long RSA key, and using the ephemeral RSA key for encrypting session secrets. But RSA key generation is very expensive. Some of the post-quantum algorithms (see Chapter 8 *Post-Quantum Cryptography*) create ephemeral keys inexpensively and achieve PFS.

11.9 PREVENTING FORGED SOURCE ADDRESSES

In a denial-of-service attack, Trudy does not successfully impersonate Alice, but she might successfully use up enough of server Bob's resources to lock out legitimate users. If Trudy sends malicious packets from her own IP address, she is likely to get caught, or people might notice the IP address of malicious traffic and have firewalls delete anything from her address. It is much easier to send a packet with a forged source address than to be able to receive a packet to that address when the other side replies.

If Trudy can inject packets into the Alice-Bob TCP connection that will be accepted by Bob, her data will be accepted by Bob, and since Bob's TCP only looks at the sequence number to determine whether the next portion of the data from Alice is new or retransmission, Alice's real data will be ignored by Bob. Even if there is a cryptographic protocol such as TLS running on top of TCP, TCP will have no idea that Trudy's packets are fraudulent, and there is no way for TLS to ask TCP to ignore part of the data and let the real data (with the same sequence numbers) to go through. So, this will cause Alice's connection to Bob to break.

After many denial-of-service attacks on TCP, TCP implementations have been modified to prevent someone from sending TCP packets from a forged IP address unless they can receive traffic sent to that address. TCP could have been modified with extra fields, but instead the industry made clever use of the TCP sequence number, so that new implementations still followed the specification and could interwork with unmodified implementation. TCP sequence numbers are 32 bits long and refer to octets in the stream (as opposed to numbering packets). A TCP packet contains two sequence numbers—one numbers the first octet the sender is sending in this packet, and the other is the last octet that the receiver is acknowledging. If Trudy attempts to inject a TCP packet from Alice's address, Bob will not accept it if the sequence number from "Alice" is not close to what he expects. So TCP implementations attempt to make it hard for Trudy to guess the sequence numbers in the Alice-Bob conversation. Instead of starting each connection with sequence number 0, an implementation may choose a random sequence number as the initial sequence number. RFC 6528 discusses various techniques.

11.9.1 Allowing Bob to Be Stateless in TCP

There is another TCP denial-of-service attack, where Trudy opens lots of connections to Bob, with forged IP addresses, and we assume she can't see Bob's replies. TCP connections start with a three-way handshake, with the first message called a SYN. Alice sends a SYN announcing her initial sequence number. Bob replies with a SYN-ACK, announcing his initial sequence number and acknowledging Alice's sequence number. Alice responds with an ACK, which contains both her sequence number and Bob's sequence number.

The simplest implementation would be for Bob to keep track, after a SYN is received from Alice, of Alice's sequence number and what Bob had randomly chosen as his initial sequence number. However, if Trudy just sends lots of SYNs from lots of forged IP addresses, she might exhaust all the space Bob's TCP implementation has set aside for remembering partially completed TCP connections.

The defense is to let Bob be **stateless**, meaning that Bob does not need to devote any space to remembering anything about a TCP connection unless the three-way handshake completes, which it will only do if the client is sending from an IP address on which it can receive traffic.

TCP implementations had to be clever in order to accomplish allowing Bob to be stateless without changing the protocol. Other protocols (*e.g.*, IPsec) have different techniques. But TCP can only use the sequence number. If Bob chose a random number for each connection, he'd have to remember what sequence number he chose. But instead, Bob can compute his initial sequence number so that when he receives the third message in a TCP handshake, he can decide "is that the sequence number I would have chosen?"

Bob can't use the same initial sequence number on all connections, so it needs to be secret and (with reasonable probability, given that the sequence number is only 32 bits) different for each connection. An example technique is for Bob to have a secret S_i that he changes every minute. If Bob's current secret is S_n , he will not know whether he generated his sequence number using S_n or S_{n-1} (because he might have switched secrets between the first message and the third message). When he receives a SYN from IP address x , Bob chooses as his sequence number $\text{hash}(x, S_n)$, sends his SYN-ACK, and then can forget about this connection. If Bob receives the third message of a connection attempt (the ACK) from IP address x , the ACK might say that Alice's sequence number is p , and Bob's is q . Bob is willing to believe anything Alice wants for her own sequence number, but Bob will sanity check q by computing what he would have chosen as his initial sequence number. Does $q = \text{hash}(x, S_n)$? If not, maybe Bob had rolled over his secret, so does $q = \text{hash}(x, S_{n-1})$? If there are few enough possibilities for Bob's secret within conceivable network delays, Bob could try all potential secrets he might have used when calculating his initial sequence number. Or Bob could use a few bits within the sequence number to encode which of the recent secrets he used. For instance, if there might at any point be eight different secrets Bob might have used, he could use three of the 32 bits in his sequence number to specify to himself which secret to use.

None of this is cryptographically secure, of course, but people did launch denial-of-service attacks of this sort, and TCP implementations deployed defenses similar to this.

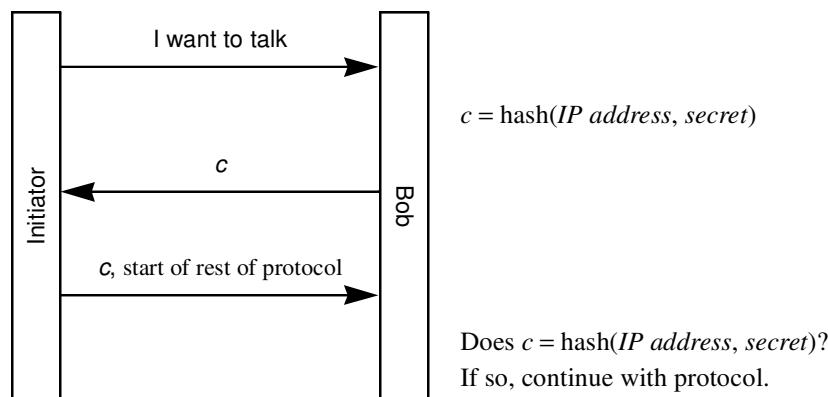
11.9.2 Allowing Bob to Be Stateless in IPsec

The designers of Photuris (RFC 2522, an early key-management protocol for IPsec) provided for denial-of-service protection by adding a feature the designers called *cookies*. These have nothing to do with web cookies, so that is a confusing name. But the best way to get the attention of IETF peo-

ple and have them like your protocol is to mention “cookies”. This feature is similar to the initial sequence number in TCP.

The cookie is a number chosen by Bob, reproducible by Bob, and unpredictable to the side initiating communication with Bob. When Bob receives a connection initiation from IP source address S , Bob computes a cookie that is a function of a secret Bob knows and the IP address from which he received a connection attempt. Bob keeps no state and does no significant computation until he receives a connection attempt with a valid cookie value.

This feature makes it somewhat harder for Trudy, using a forged IP address, to use up resources at Bob. The cookie feature doesn’t hurt, other than making the protocol slightly more complex. In theory, cookies could be used only when a node is getting swamped, saving the round-trip delay in the usual case. The cookie protocol might look like Protocol 11-19:



Protocol 11-19. Stateless Cookie Protocol

11.10 ENDPOINT IDENTIFIER HIDING

Another feature in some of these protocols is the ability to hide the identities of the two communicating parties from eavesdroppers. The IP addresses will still be visible, but this feature hides their names. Assume Alice wants to talk to Bob without letting an eavesdropper know that Alice wants to talk to Bob. A mechanism for accomplishing this is to first do an “anonymous” Diffie-Hellman exchange, which establishes an encrypted tunnel, but to an unknown endpoint. The tunnel might have a MITM.

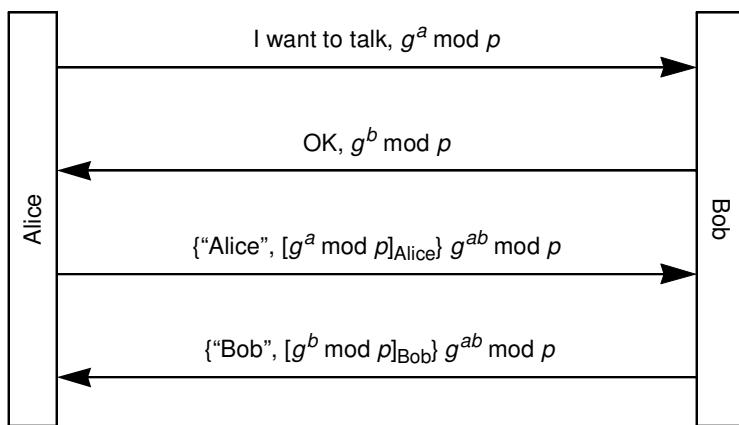
The next step will be for Alice to divulge her identity to Bob (but within the encrypted tunnel). Eavesdroppers won’t see her name, but a MITM would see it. Then Alice and Bob can do

mutual authentication. A MITM will not be able to successfully impersonate Alice to Bob or Bob to Alice. So all the MITM will have accomplished is learning that Alice wanted to talk to Bob.

Note that by carefully designing the protocol, you can arrange for the MITM to only be able to learn one of the two identities before being discovered by the other side as an impostor. Which identity is better to hide from an active attacker? One argument says that it is better to hide the initiator's identity (Alice) than the responder's identity (Bob), because Bob's identity is probably already known. He has to be sitting at a fixed IP address waiting to be contacted, whereas Alice might be dialing in from anywhere, and her identity could not be guessed from her IP address.

But a different argument says that it is better to hide Bob's identity. If Bob divulges his identity first, then anyone can initiate a connection to Bob and get him to divulge his identity. Unless there is a strict client/server model in which clients never accept connections and only initiate them, having a protocol in which the responder divulges his identity first makes it trivial to find out who is at a given IP address. In contrast, for an active attacker to trick Alice into revealing her identity, it requires impersonating Bob's address and waiting for Alice to initiate a conversation.

An example protocol, assuming the two sides have public signature keys, might be Protocol 11-20:



Protocol 11-20. Identity Hiding

In this protocol, an active attacker will be able to discover Alice's identity, but not Bob's. It is easy to arrange instead to hide Alice's identity instead (see Homework Problem 10).

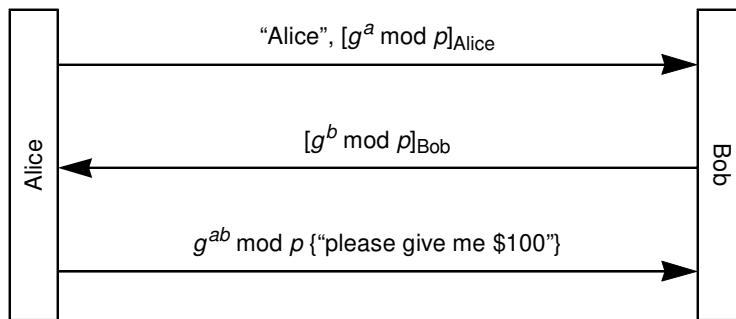
If Alice and Bob know in advance to whom they will be talking (perhaps they are two spies who will be contacting each other at a specific time), then a protocol based on a shared secret key will hide both identities. This is accomplished by authenticating based on the secret key and not sending identities at all (see Homework Problem 11).

If Alice already knows Bob's public encryption key, it is possible to hide both identities from active attackers (see Homework Problem 12).

There are cases where Alice needs to tell Bob his own name. Suppose Bob is one of many services provided at a web server. For example, all the services (like book seller, dictionary, hire-a-hitman) reside at the same address and each has a different certificate. When Alice makes a connection to that address, Alice needs to inform Bob to which service she wants to connect. This feature is sometimes called **You-Tarzan-Me-Jane**, because Alice is telling the other side who she wants him to be. TLS has this feature (called SNI, for Server Name Indicator). IPsec, unfortunately, does not.

11.11 LIVE PARTNER REASSURANCE

If Trudy can replay messages from previous conversation negotiations, she might be able to get Bob to waste space on a connection, or worse yet, she might be able to replay the subsequent data messages and, even if she can't decrypt the conversation, she might be able to cause Bob to repeat actions. For instance, when Bob (an ATM machine) talked to Alice, she might have requested Bob to put \$100 into the money tray, as in Protocol 11-21:

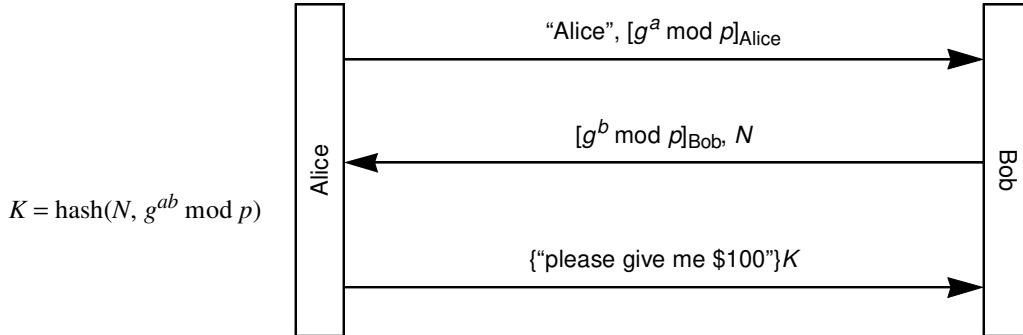


Protocol 11-21. A Protocol Vulnerable to Replay if Bob Reuses b

An hour later, if Trudy replays Alice's messages, it is important that Bob realize that this is not a conversation with the live Alice. If Bob chose a different b in each Diffie-Hellman exchange, then there wouldn't be a problem, but it is computation intensive to compute g^b , so it might be nice for Bob to be able to reuse b .

A way to allow Bob to reuse b and still avoid replay attacks is for Bob to choose a nonce for each connection attempt and have the session key be a function of the nonce as well as the Diffie-Hellman key. So the protocol might be modified to look like Protocol 11-22. The session key is a function of the nonce N as well as the Diffie-Hellman value. This seems similar to a cookie, but it is desirable for a cookie to be stateless so that Bob does not have to keep state until he's at least sure

the other end can listen at the IP address from which it claims to be sending. With the most straightforward implementation of a stateless cookie, the cookie will be reused, so wouldn't work as a nonce. It is possible to design a protocol that will allow something to work both as a nonce and as a stateless cookie.



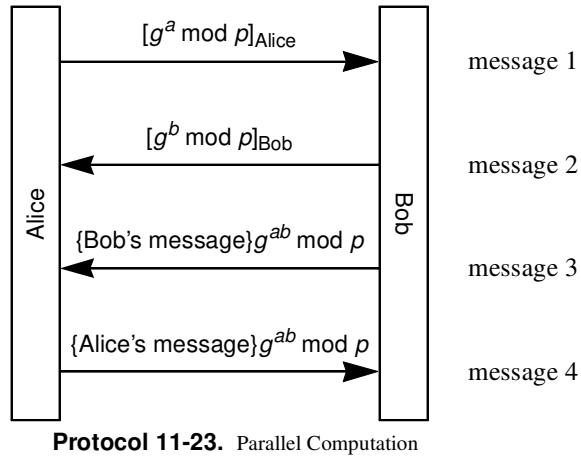
Protocol 11-22. Using a Nonce so Bob Knows It's Not Replayed Messages from Alice

Note that we've only ensured that Bob knows it's the live Alice (and not replayed messages). How would Alice know it's the real Bob? If Alice chooses a different a each time, and if Alice receives proof from the other side that it knows K (for instance, by acting on her request, which was encrypted with K), then she knows it's the real Bob. But suppose Alice, like Bob, would like to reuse a to save herself from computing $g^a \text{ mod } p$ (see Homework Problem 14).

11.12 ARRANGING FOR PARALLEL COMPUTATION

A lot of protocols require both Alice and Bob to compute a shared Diffie-Hellman key. This might take a long time. It can speed up the total elapsed time for an exchange if Alice and Bob can be computing at the same time, as in Protocol 11-23.

This exchange might seem silly. Why not combine message 2 with message 3? The reason is that telling Alice $g^b \text{ mod } p$ gives Alice a head start on computing g^{ab} . She can be computing g^{ab} at the same time Bob is computing it. Al Eldridge probably was the first to invent this trick of sending an extra message in order to allow the computation-intensive calculations to be done in parallel. He implemented it in Lotus Notes (an early public key messaging system that we discuss because it had some interesting innovations). In Lotus Notes, Bob sent something encrypted with Alice's public key in message k and then later sent his signature on that message in message $k+1$. This let Alice do the expensive private key decryption while Bob was doing the expensive signature.

**Protocol 11-23.** Parallel Computation

Notice that although this adds a message it doesn't add any round-trip times, so it can be faster even if Alice and Bob are very far apart, talking to each other via, say, carrier pigeons (see RFC 1149 or RFC 2549).

11.13 SESSION RESUMPTION/MULTIPLE SESSIONS

Protocols such as IPsec and TLS use public key operations for the initial handshake in which Alice and Bob authenticate and establish a secure session. Because this initial step is expensive, a trick some protocols use is to leverage the initial expensive handshake to cheaply resume a secret session that has been idle or spawn multiple secure sessions in parallel. For example, a common use of TLS is for Alice to be a browser and Bob a server with a web page that has many images. Alice will create multiple secure sessions to Bob to retrieve the images in parallel.

In IPsec (where they refer to secure sessions as SAs), Alice and Bob might want to spawn multiple SAs for different types of traffic (§12.1.3 *IKE-SAs and Child-SAs*). A resumed SA or a new SA, leveraging the handshake of the original SA, will use different secrets computed by hashing in new nonces exchanged during the SA resumption or SA creation handshake.

A simple mechanism for resuming a secure session or cheaply spawning a new secure session is for Bob, during the initial expensive handshake, to send Alice an ID for the session that he chooses. If Bob remembers the information for that session, he can inexpensively resume the session or create a new session. If he has forgotten the state associated with the session, then he tells Alice they'll have to start over, and they establish a new session.

But what if there are many instances of Bob? If Alice attempts to resume a session with a different instance of Bob, then the other instance will not remember the session, unless all the Bob instances do some sort of synchronization so that they all know the IDs, secrets, and other information (such as crypto algorithms negotiated) for all sessions.

An elegant alternative is for Bob to encrypt all the state associated with the session with a secret S that all the Bob instances know (and nobody else knows, hopefully), and use that quantity as the ID. Then when Alice sends the “ID”, Bob decrypts it and knows everything he needs to know in order to resume the session. This mechanism works even if there are multiple instances of Bob.

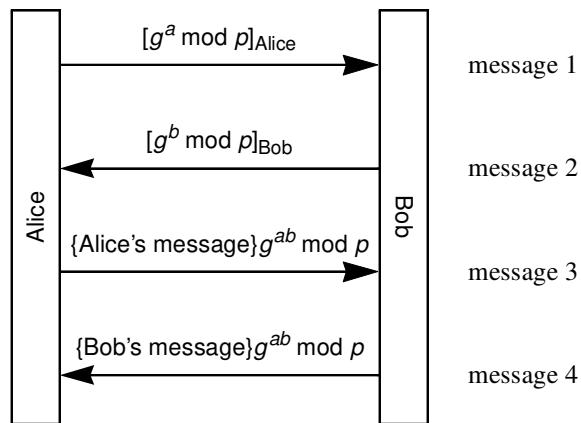
Alice does not need to know whether the session ID Bob sends her is the ID of a database entry where the session state is kept or is the encrypted session state itself. The ID is just a large random-looking number to Alice. The only issue is that the protocol must allow the ID to be large enough. If the ID were only a database entry ID, then a few octets would be long enough. To encode all the state associated with a session would require a much larger ID.

There were other interesting mechanisms used in older protocols. In Lotus Notes, Bob (the server) had a secret S that he shared with nobody and changed periodically (*e.g.*, once a month). After Bob authenticated Alice, he sent her a secret, S_{A-B} , which was a hash of her name and S . (He sent S_{A-B} encrypted with Alice’s public key.) Until Bob changed S , he’d give Alice the same S_{A-B} each time she successfully authenticated. The actual session key (used for encryption and integrity protection) was a function of S_{A-B} and nonces sent by each side. If Alice told Bob her name “Alice” and showed knowledge of the S_{A-B} that would result from hashing S and “Alice”, then Bob assumed he’d authenticated her in the recent past, and they’d skip the expensive public key operations and create secrets for the session by exchanging nonces. The actual session key (used for encryption and integrity protection) was a function of S_{A-B} and the nonces. If Bob had changed S , then Alice’s attempt to bypass the expensive authentication step would fail, and Alice and Bob would start from the beginning, exchanging certificates, signing things, etc. The Lotus Notes scheme did not require Bob to keep state, and it therefore worked with multiple instances of Bob.

Digital’s DASS scheme (RFC 1507) has a different interesting method of bypassing the public key cryptography. During the handshake, Alice sends Bob the session secret S , encrypted with Bob’s public key and signed with Alice’s private key, *i.e.*, $\{S\}_{Bob}Alice$. If Bob remembers $\{S\}_{Bob}Alice$ and S from a recent Alice-Bob session, then he merely compares the received $\{S\}_{Bob}Alice$ from Alice with the stored $\{S\}_{Bob}Alice$. If they match, he doesn’t have to bother decrypting it to extract S . But if he doesn’t remember $\{S\}_{Bob}Alice$, or if Alice has decided to create a session with a different secret S , then Bob verifies Alice’s signature on $\{S\}_{Bob}Alice$ and decrypts $\{S\}_{Bob}$ with his private key to obtain the new S . If Bob doesn’t remember the previous session, but Alice does, then Alice still saves time. If they both remember the previous session, then they both save time. What’s interesting about this protocol is that the protocol messages look the same whether state has been kept or not.

11.14 PLAUSIBLE DENIABILITY

If a protocol involves having Alice sign something containing Bob's name, then it offers proof that Alice intentionally talked to Bob (though it still gives no indication of what they talked about). In some cases Alice would like to assure Bob that it is her but not provide proof that she talked to Bob. If Alice and Bob are authenticating each other based on a shared secret, then there is no way to prove to a third party that Alice and Bob communicated with each other, because the entire conversation could have been constructed by Alice or Bob. If Alice and Bob authenticate each other using public encryption keys, *anyone* could create an entire conversation that looks like a conversation between Alice and Bob. (For example, consider Protocol 11-23 and change the first two messages from being signed by the sender to being encrypted with the recipient's public key. No knowledge of either side's private key is required to create such messages.) If Alice and Bob authenticate each other using public signature keys, then it is possible to create a protocol in which each signs information including the other's identity, in which case there is no plausible deniability. But it is also possible to avoid signing the other side's identity, and therefore preserving plausible deniability (see Protocol 11-24).



Protocol 11-24. Plausible Deniability with Signature Keys

11.15 NEGOTIATING CRYPTO PARAMETERS

It's fashionable today for security protocols to negotiate cryptographic algorithms, rather than simply having the algorithms be part of the specification. It certainly makes the protocols more

complex. Examples of things to negotiate are the algorithm for encryption, the size of the key, the algorithm for integrity protection, the algorithm for hashing, the algorithm for key expansion, and the group to use in a Diffie-Hellman exchange.

One reason for allowing choice of cryptographic algorithms is so that over time, systems can migrate to stronger but slower crypto as attackers' and defenders' devices get faster. Also, it allows migration from a cryptographic algorithm that gets broken. In order to allow for a smooth migration to new algorithms when endpoints are being updated independently, support is added for the new algorithms while the old ones continue to be supported until all the endpoints have been updated. Only then can support for the old algorithms be removed. Typically, algorithm negotiation takes place by having one side announce all the algorithms it supports while the other side chooses the best ones that it also supports.

11.15.1 Suites vs. à la Carte

There are several types of cryptographic algorithms a protocol might use, *e.g.*, encryption, integrity protection, hashing. Suppose there are four types of cryptographic algorithms to be negotiated, and for each of those, there are five choices. It is feasible for Alice to say, "Here are my acceptable encryption algorithms. Here are my acceptable encryption keysizes. Here are my acceptable hash algorithms. Here are my acceptable algorithms for integrity protection." Four types of algorithms times five choices for each is twenty items to list when negotiating.

But what if not all the algorithms work with each other? AES-GCM, for example, provides both authentication and encryption, so it would not make sense to choose AES-GCM for one of those functions and some other algorithm for the other. TLS before TLS 1.3 solved this by proposing suites of cryptographic algorithms, where each suite identifier implied all the algorithms that would be used. The problem there was that if an endpoint wanted to support all five choices for four types of algorithms, it would need to offer 625 different suites! And as people wanted to standardize new cryptographic algorithms, they wanted to be able to allow it to work with all other algorithms, so the list of all possible suites became unworkably large. IPsec's IKE took a different approach. It offered lists of each type of algorithm and allowed its peer to mix and match. But when there were objections that an endpoint might not support all combinations, they introduced a complicated additional mechanism that allowed alternate whole sets where mix and match was allowed within them. This evolved into a very complex solution to what seemed like a very simple problem. TLS 1.3 greatly reduced the number of algorithms it would support and went to more of an à la carte approach where endpoints were required to support arbitrary combinations of the algorithms they offered. At the time of this writing, it is yet to be seen whether they will be able to make that simple approach work or whether people wanting to support different variations will cause the offer structure to balloon in complexity again.

11.15.2 Downgrade Attack

One potential security flaw, if the negotiation isn't done correctly, is that an active attacker Trudy might be able to trick Alice and Bob into using weaker cryptography by removing from Alice's suggestions the ones that Trudy can't break. Alice and Bob would like to agree on the strongest possible cryptography that they both support. If Alice is willing when necessary to use weak crypto and a weak algorithm is among the choices, and Bob is willing when necessary to use weak crypto, removing the strong choices from Alice's list may cause them to agree on weak crypto if the protocol is not carefully designed.

The reason this is a common vulnerability is that while Alice and Bob are negotiating cryptographic algorithms, they probably do not yet have a shared secret with which to do integrity protection of the packets. The solution is to wait until they have established a shared secret and then detect the tampering by having Alice reiterate (in a cryptographically protected way) what proposals she had made.

The way both IPsec and TLS defend against this attack is to accumulate a hash of all the messages sent during connection setup before a key is established and then send that hash (encrypted and integrity protected) once the key is established. That is effective unless Trudy managed to get Alice and Bob to agree to such weak algorithms that Trudy can defeat the session authentication mechanism in real time. But the only way Trudy will succeed is if both Bob and Alice support a choice that is absurdly weak. It is easy for either Alice or Bob to defend against that case—just don't support such a weak cryptographic algorithm.

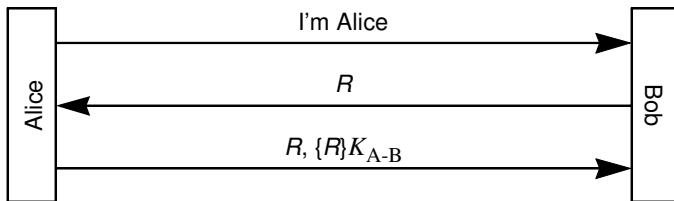
11.16 HOMEWORK

1. Consider Protocol 11-12. R is the challenge sent by Bob to Alice, and S is the secret Alice and Bob share. (S is shorthand in this problem for K_{A-B}) Which of the following are secure for a session key?

$$S \oplus R \quad \{R+S\}_S \quad \{R\}_{R+S} \quad \{S\}_S \quad \{R\}_S \quad \{S\}_R$$

2. Suppose we are using a three-message mutual authentication protocol, and Alice initiates contact with Bob. Suppose we wish Bob to be a stateless server, and therefore it is inconvenient to require him to remember the challenge he sent to Alice. Let's modify the exchange so that Alice sends the challenge back to Bob, along with the encrypted challenge. So the

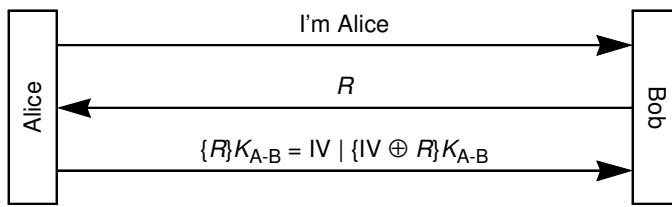
protocol is:



Is this protocol secure?

3. In the discussion of Protocol 11-4, Bob remembers all the timestamps he's seen within the last ten minutes. Why is it sufficient for him to remember only ten minutes' worth of timestamps?
4. In Protocol 11-4, assume there are multiple instances of Bob, say, Bob₁, Bob₂, Bob₃. Let's assume one of the instances, the main Bob, that we'll call Bob_M, is informed by the other Bobs when timestamps have been used. When Bob_i receives an encrypted timestamp from Alice, Bob_i informs Bob_M of the used timestamp and asks Bob_M if the timestamp has been used already. Bob_M tells Bob_i whether the timestamp should be accepted or not. Suppose Trudy eavesdrops on Alice's message to Bob_i and immediately repeats that encrypted timestamp to a different Bob instance, Bob_j. Is it possible that Bob_j's query to Bob_M will arrive at Bob_M before Bob_i's query? If so, Trudy will be able to impersonate Alice, and Alice's authentication will fail. What kind of mechanism might be used to lower the probability that Trudy will successfully impersonate Alice? Is there any way to make it impossible for this scenario to occur? (Hint: Consider Bob_i crashing before informing Bob_M.)
5. Design a two-message authentication protocol, assuming that Alice and Bob know each other's public keys, which accomplishes both mutual authentication and establishment of a session key.
6. In §11.3.2 *Session Key Based on Public Key Credentials*, in item 2, Alice signed her quantity (*i.e.*, she sent $\{R_1\}_{Bob}Alice$ instead of merely $\{R_1\}_{Bob}$) to foil Trudy from doing a MITM attack and to avoid Trudy being able to impersonate Alice. In item 3, she doesn't sign her quantity, *i.e.*, she sends $\{R_1\}_{Bob}$ to Bob. Bob chooses R_2 and sends $\{R_2\}_{Alice}$ to Alice. They use $R_1 \oplus R_2$ as a session key. Why isn't it necessary for Bob and Alice to sign their quantities in item 3 when they are using $R_1 \oplus R_2$ as the session key?
7. There is a product that consists of a fancy telephone that, when talking to a compatible fancy telephone, does a Diffie-Hellman key exchange in order to establish a secret key, and the remainder of the conversation is encrypted. Suppose you are an active wiretapper capable of modifying messages before relaying them. How can you listen to a conversation between two such telephones?

8. Suppose the telephones in Homework Problem 7 display (to their users) a hash of the session secret. How can Alice and Bob, using these telephones to talk, detect whether someone is eavesdropping?
9. Consider this protocol and assume R fits into a single block and Alice encrypts R in CBC mode, so $\{R\}K_{A-B}$ is computed (using CBC mode) by having Alice choose an IV and sending IV, $\{IV \oplus R\}K_{A-B}$. Suppose Eve sees an exchange between Alice and Bob, where the challenge is R_1 . How can Eve then impersonate Alice, if Bob sends a different challenge R_2 ?



Protocol 11-25. Another Protocol in Which R Must Be Unpredictable

10. Referring to §11.10 *Endpoint Identifier Hiding*, modify Protocol 11-20 to hide the initiator's identity rather than the target's identity.
11. As mentioned in §11.10 *Endpoint Identifier Hiding*, it is possible to hide both identities from active attackers if Alice and Bob share a secret key and there is a small set of entities that might initiate a connection to Bob. Show such a protocol.
12. As mentioned in §11.10 *Endpoint Identifier Hiding*, it is possible to design a protocol that will hide both identifiers from an active attacker, assuming that Alice (the initiator) already knows Bob's public key. Show such a protocol.
13. Assuming private key operations are very slow, show a protocol that runs faster with an extra message. Suppose transmission delay is longer than the time it takes to do a private key operation. Would the protocol still complete faster with an extra message?
14. In Protocol 11-22, explain why Bob knows that Alice is the real Alice and not someone replaying Alice's messages. How does Alice know that it's the real Bob if she uses a different a each time? Modify the protocol to allow both Alice and Bob to reuse their a and b values and yet have both sides be able to know they are talking to a live partner.
15. Assume a stateless session resumption scheme as described in §11.13 *Session Resumption/Multiple Sessions*. Suppose Bob changes his S every ten minutes, and yet you'd like to be able to resume sessions that have been idle for longer than that, say, two hours. How might that work?
16. In the DASS session resumption protocol described in §11.13 *Session Resumption/Multiple Sessions*, under what circumstances does Bob save computation? Under what circumstances does Alice save computation? Is there a case where Bob saves computation but Alice does not?

12 IPSEC

As we said in Chapter 11 *Communication Session Establishment*, IPsec is a secure session protocol that runs on top of network layer 3 (see §11.7 *What Layer?*). The implication of running directly on layer 3 (*e.g.*, IP) is that each packet is independently cryptographically protected. IPsec does not guarantee that all packets will arrive or that those that do arrive will be delivered in the order they were sent. IPsec only guarantees that packets that do not meet the integrity check will be discarded, and packets that are duplicates will be discarded. This design makes it easy to implement in network adapters. IPsec does not need to buffer packets. IPsec can process and deliver packets independently, even if they arrive out of order. IPsec could in theory work on top of any layer 3 protocol, and, indeed, works with both IPv4 and IPv6.

IPsec consists of several pieces. One is the initial authentication handshake, which is specified by IKE (Internet Key Exchange, RFC 7296). Another is the encoding of data packets, which is in one of two formats: ESP (Encapsulating Security Payload, RFC 4303) or AH (Authentication Header, RFC 4302). IPsec was originally envisioned to also be able to protect multicast traffic, and ESP and AH are designed to handle both unicast and multicast. But IKE can only establish keys for a connection between two endpoints, so there is no standard mechanism for setting up multicast IP streams (other than perhaps manual configuration).

12.1 IPSEC SECURITY ASSOCIATIONS

As we mentioned in §1.5 *Cryptographically Protected Sessions*, IPsec refers to a cryptographically protected session between Alice and Bob as a **security association** or **SA**.

Associated with each end of an SA is a cryptographic key and other information such as the identity of the other end, the sequence number currently being used, and the cryptographic services being used (*e.g.*, integrity only, or encryption+integrity, and which cryptographic algorithms should be used). An IKE-SA (the SA Alice and Bob use to do mutual authentication, signaling, and creating child-SAs) is bidirectional. All the expensive public key operations are performed in the IKE-SA. The IKE-SA is then leveraged to create what IPsec refers to as **child-SAs**, on which actual

application data is sent. Child-SAs between Alice and Bob are unidirectional and always created in pairs (Alice-to-Bob traffic and Bob-to-Alice traffic). A child-SA will either use ESP or AH format. **ESP** (encapsulating security payload) has optional integrity protection and optional encryption. **AH** (authentication header) only has integrity protection.

The IPsec data header (ESP or AH) includes a field known as the SPI (SECURITY PARAMETER INDEX) that identifies the security association, allowing Bob to look up the necessary information (such as the cryptographic key) in his SA database for a received packet. The SPI value is chosen by the destination (Bob). It would seem as though the SPI alone should allow Bob to know the SA, since Bob can ensure that the SPI for the SA is unique with respect to all the sources that Bob has SAs with. But the IPsec design was intended to also allow IPsec to protect multicast data (where the network delivers a packet to multiple recipients). If Bob is receiving multicast data, Bob would not have chosen the SPI, and the SPI value for the multicast group might be equal to an SPI value that Bob already assigned for an SA between him and a single other principal. Or two different multicast groups might choose the same SPI. Therefore, the SA is identified by both the SPI and the destination address. (The destination IP address of a packet received by Bob will be Bob's own IP address for unicast or a group address if it's multicast.) Furthermore, IPsec allows the same SPI values to be assigned to different SAs if one SA is using AH and one is using ESP, so the SA is defined by the triple $\langle \text{SPI}, \text{destination address}, \text{flag for whether it's AH or ESP} \rangle$.

IKE-SAs are also distinguished by SPIs, but the SPIs are handled a little differently than in child-SAs. In IKE, both Alice's chosen SPI (SPI_A) and Bob's chosen SPI (SPI_B) are in each IKE packet. It would have been simpler to have the pair be $\langle \text{SPI}_A, \text{SPI}_B \rangle$ when Bob is sending to Alice, and $\langle \text{SPI}_B, \text{SPI}_A \rangle$ when Alice is sending to Bob. If the recipient's SPI value were always first, then the recipient could just look at the first SPI value in the pair and find it in his table (since he assigned the value). However, IKE chose to always have Alice's (the original initiator of the IKE) SPI first and Bob's (the responder's) second. There is a flag in the IKE header indicating whether this packet is from the original SA initiator or the original SA responder. (This was how it was done in IKEv1. It would have been nice to have fixed this in IKEv2, because then IKEv2 would have been a bit simpler.)

12.1.1 Security Association Database

A system implementing IPsec keeps a security association database. When transmitting to IP destination X , the transmitter looks up X in the security association database, and that entry will tell it how to transmit to X , *i.e.*, it will provide the SPI, the key, the algorithms, the sequence number, which IP addresses are acceptable to send on this SA, and so on. When receiving an IPsec packet, the SPI of the received packet is used to find the entry in the security association database that will tell the receiver which key, which algorithms, which sequence numbers within a range have already been used, and so on to use to process the packet.

12.1.2 Security Policy Database

Firewalls are configured with tables telling them what type of traffic to allow, based on information such as the IP header source and destination addresses and TCP ports. IPsec is assumed to have access to a similar database specifying which types of packets should be dropped completely, which should be forwarded or accepted without IPsec protection, and which should be protected by IPsec, and if protected, whether they should be encrypted and/or integrity-protected. Decisions could, in theory, be based on any fields in the packet, *e.g.*, source IP address, destination IP address, protocol type in the IP header, and layer 4 (TCP or UDP) ports. When Alice and Bob negotiate parameters for a child-SA, they also negotiate which ranges of IP addresses and ports are acceptable to send on that SA. If Bob is configured to only accept a certain set of IP addresses, and Alice sends packets to a different IP address over the SA, Bob will drop the packet. But it would be more polite to warn Alice which IP addresses he is willing to handle packets for on that SA. Being polite in this case saves bandwidth, but more importantly, if Alice has an alternative path (perhaps a different SA to Bob) that will not drop the traffic, it is certainly better for Alice to know, so she is not sending traffic just to be dropped.

12.1.3 IKE-SAs and Child-SAs

Alice and Bob use the IKE protocol to do mutual authentication and establish a session secret. But the SA created by IKE is not actually used for sending application data. Instead, the IKE-SA is used to inexpensively create child-SAs, and it is over a child-SA that application data will be transmitted. The session key for a child-SA spawned from an IKE-SA is a function of the secret created by the IKE handshake, along with other information such as nonces that are exchanged during the handshake to create a child-SA.

Why would Alice and Bob want more than a single SA to communicate with each other? Wouldn't it be simpler to create a single SA and send data on that, rather than to first create an IKE-SA and then create a child-SA for sending data? The rationale for this feature is that perhaps Alice and Bob would want to create many different SAs for sending data between them. Each child-SA has separately negotiated cryptographic algorithms. Alice and Bob might create one child-SA with only integrity protection because the data is not security-sensitive, and they don't want to spend extra computation encrypting the data. They might create another child-SA for normal security-sensitive data and use 128-bit AES. And for super-sensitive data, they might create another child-SA and use 256-bit AES. They can also roll over the session key on a child-SA by creating a new child-SA with a new key and closing the old child-SA. And some customers feel more secure if their data-in-transit is sent with a different encryption key than some other customer's data, so they would like separate SAs.

Also, IPsec puts a sequence number on data packets and is supposed to discard duplicate packets. Note that is the only purpose of the IPsec sequence number. IPsec does not put packets back in order or ask for retransmission of lost packets. But IPsec is not supposed to deliver a replay of a packet to the application. So Bob (the receiver) needs to remember all the used sequence numbers, but only for sequence numbers that are close enough to the most recent one received that network delays could account for receiving a packet with that sequence number. Bob can discard packets with sequence numbers older than that.

The scenario the IPsec designers envisioned that would benefit from many child-SAs between Alice and Bob is that Alice is sorting data into streams that she sends on different paths. Alice might do this because one application requires a different quality of service than another, and so the paths that different applications are transmitted on might have very different delay characteristics. Another scenario is if Alice and Bob are firewalls multiplexing traffic for many customers between themselves, and some customers might have paid for higher quality of service. If streams multiplexed over the same IPsec SA are sent on very different paths, the delays on the different paths might be very different, and Bob would have to remember sequence numbers from a larger range. If all the packets on a single child-SA are handled by the network similarly, Bob does not need to remember a very large range of used sequence numbers (see Figure 12-1).

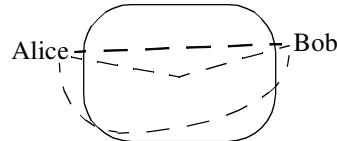


Figure 12-1. Multiple Paths between Alice and Bob

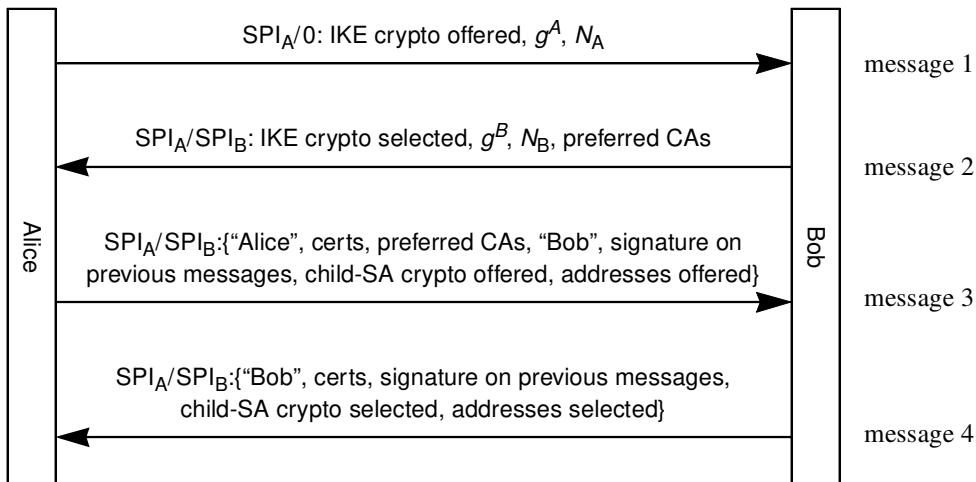
Instead of IPsec's design of creating lots of child-SAs, Alice and Bob could have independently created separate SAs for each class of traffic. However, creating child-SAs is very inexpensive once the IKE-SA has been established. All the expensive computation (*e.g.*, private key operations, Diffie-Hellman) is done during the IKE-SA (although, optionally, Alice and Bob can do a Diffie-Hellman handshake when creating a child-SA). Without the optional Diffie-Hellman handshake, creating the IKE-SA and a single child-SA is not more expensive than doing a single handshake and establishing a single SA, because the creation of the first child-SA is piggybacked on the IKE exchange. The concept of two different SAs (IKE and child) makes it a bit confusing for understanding IPsec, but even in the case where Alice and Bob just want to have a single SA for sending data, the IPsec design doesn't require additional round trips or significantly more computation.

12.2 IKE (INTERNET KEY EXCHANGE PROTOCOL)

The original version of IKE was overly complex, documented in several different documents (RFCs 2407, 2408, and 2409) with conflicting terminology and technical details, and vague about things like which end of the connection should be responsible for lost handshake messages. It was while trying to document IKE for the second edition of this book that we realized how baroque, and in some cases, broke, the first version of IKE was. We_{1,2} suggested to the working group various ways of simplifying IKE and found out from them which features they really wanted to retain. I₁ was the chief designer of IKE version 2, which was much simpler, more efficient, and retained all the features that people had plausible reasons to want. The goal of preserving as much of the flavor of the design of IKEv1 as possible made IKEv2 not quite as clean as it might have been if it had been designed from scratch. We will only describe IKE version 2 here.

IKE runs over UDP (see §1.8.2 *TCP and UDP Ports*). Unlike TCP, UDP does not retransmit messages or give sequence numbers to messages. The only things it really provides are ports to enable the receiver of a UDP message to know which process to give the message to. UDP packets can be very large, and IP will, if necessary, fragment a large packet into chunks that can traverse the network. The IP implementation at the other end will reassemble all the pieces of the UDP packet before sending the UDP packet to the destination process. IKE uses UDP port 500 and/or 4500.

All messages in IKE consist of a request and a response. The usual case of creating the IKE-SA between Alice and Bob consists of two request/response pairs (*e.g.*, four messages total; see Protocol 12-2). Additionally, the first child-SA can be negotiated in the same set of messages.



Protocol 12-2. IKEv2 Initial Exchange

There might be more messages, for instance, if Alice uses a Diffie-Hellman group that is unacceptable to Bob (in which case she has to restart the negotiation), or if Bob wants Alice to prove she can receive at the IP address she is sending from (see §11.9.2 *Allowing Bob to Be Stateless in IPsec*), or if messages are lost and the requester times out and retransmits the request.

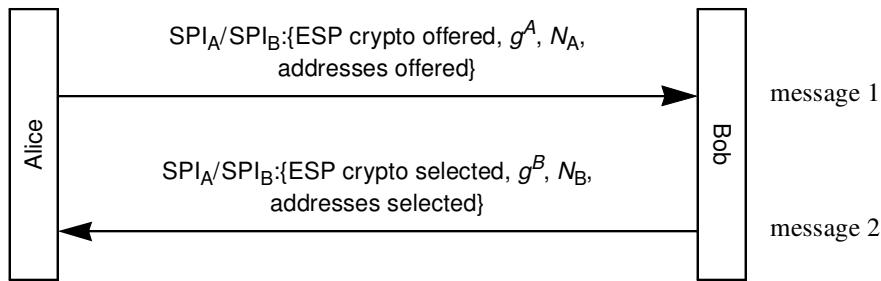
In the first two messages, Alice and Bob accomplish the following:

- Negotiate cryptographic algorithms for cryptographic protection of the IKE-SA. Alice suggests which algorithms she's willing to use in message 1, and Bob chooses from that set and informs Alice of his choice in message 2. Note that she also sends a Diffie-Hellman value (g^A). If Bob doesn't like that choice of Diffie-Hellman parameters, Bob might reject the first message. Alice will start over, using a Diffie-Hellman group that Bob will accept. This rare case will result in an extra two messages.
- Alice chooses her SPI value SPI_A (for the IKE-SA) in message 1. Since she doesn't yet know Bob's chosen SPI value, she sets that field to 0. Note that SPI_B works as a stateless cookie for Bob (see §11.9.2 *Allowing Bob to Be Stateless in IPsec*). Suppose Bob is under attack (swamped with garbage requests for IKE-SA creation from bogus IP addresses). Bob can avoid doing any significant computation or keeping any state until he confirms that the initiator can receive at the IP address in the source address of the IP packet. Bob confirms that the initiator can receive at that IP address by responding to a create IKE-SA request containing a zero SPI_B with a message that says "repeat your request, but include this nonzero SPI_B when you repeat your request." Bob will have chosen a value of SPI_B that he can reconstruct but that nobody else will be able to predict. Then Bob can forget he received that request.
- Negotiate a master session secret S_M . S_M is a function of the Diffie-Hellman exchange and the randomly chosen nonces (N_A and N_B). S_M is used, along with the same nonces (N_A and N_B) and the SPIs to generate five more keys: one encryption key for Alice-to-Bob IKE traffic, another for Bob-to-Alice IKE traffic, an integrity key for Alice-to-Bob IKE traffic, a different integrity key for Bob-to-Alice IKE traffic, and another master secret we'll call S_C that will be used to generate session keys for child-SAs. The keys for a child-SA (two encryption keys and two integrity keys) are derived from S_C , the nonces exchanged during creation of the child-SA, and the Diffie-Hellman value (if an optional Diffie-Hellman was done during creation of the child-SA).
- Alice and Bob let the other side know which CAs they trust, so that the other side knows which certificates to send. Note that IPsec allows for other ways to authenticate, e.g., with a configured pre-shared secret key, but usually authentication is done with certificates.
- Alice and Bob each show knowledge of their private key and detect whether a MITM meddled with messages 1 and 2 (where there was no integrity check because Alice and Bob had not yet established a session key) by signing fields from the previous messages.

- Alice's and Bob's identities are hidden from an eavesdropper because messages 3 and 4 are encrypted.
- They create the first child-SA (in messages 3 and 4), along with negotiating which IP addresses are acceptable to send on this child-SA. When creating a child-SA, each side chooses the SPI they would like.

12.3 CREATING A CHILD-SA

Once the IKE-SA is established, either side of the IKE-SA can request establishing an additional pair of child-SAs (the first pair of child-SAs is piggybacked on the initial IKE-SA establishment). Protocol 12-3 illustrates creating a pair of ESP-SAs, including doing the optional Diffie-Hellman.



Protocol 12-3. IKEv2 Rekey or Extra Child SA

SPI_A/SPI_B are the pair of SAs from the IKE-SA. Alice offers a set of cryptographic algorithms she is willing to support. She also suggests the list of IP addresses she would like to send over this SA. Bob chooses from the sets suggested by Alice. There is no issue with a MITM because these messages are cryptographically protected by the IKE-SA. The SPIs chosen for the child-SAs are included in the CRYPTO OFFERED and CRYPTO SELECTED fields.

The handshake in Protocol 12-3 can also be used to roll over the IKE-SA key by creating a new IKE-SA and then deleting the old one.

12.4 AH AND ESP

AH and ESP are the two types of IPsec headers for data packets. AH provides integrity protection only. ESP provides encryption and/or integrity protection. To understand why there are two headers defined, and why they look so different, it is important to understand the history. The AH committee had a lot of IPv6 enthusiasts. The ESP committee was composed of security people who didn't care what kind of layer 3 protocol they ran over. Mostly the two committees ignored each other.

Given that ESP optionally provides integrity protection (in addition to optional encryption), it's natural to wonder why AH is needed. In fact many people argue (and we concur) that AH is not necessary.

ESP and AH provide integrity protection differently. Both provide integrity protection of everything beyond the IP header, but AH provides integrity protection for some of the fields inside the IP header as well. We explain why it is not necessary to protect the IP header, and the real motivation for AH doing so, in §12.4.2 *Why Protect the IP Header?*

12.4.1 ESP Integrity Protection

Originally, AH was intended to provide integrity protection, and ESP was intended to provide encryption only. The original IPsec vision was that if people wanted both encryption and integrity protection, they would use both headers. But at some point the ESP people (correctly) decided it was dangerous to provide encryption without integrity protection, so they added optional integrity protection. The AH people didn't notice, or at least, they didn't object. They could have argued that anyone who wanted to get integrity protection should also use the AH header.

So, given no objections, ESP added optional integrity protection. But later, the ESP people said, "People might want integrity only, perhaps for performance reasons." So they wanted to make encryption optional in ESP. But this time, the AH people were paying attention and objected, saying that if you want only integrity protection, you should use AH. They insisted that ESP must provide encryption.

So the ESP people said, "Okay. Encryption is mandatory. We can live with that. But we can use any encryption algorithms we want, right?" The AH people said, "Of course." This led to my favorite specification ever, RFC 2410, "The NULL Encryption Algorithm and Its Use With IPsec". This defines a new encryption algorithm (null encryption). The RFC brags about how efficient and flexible it is. It will work with any key size. There is no reason even for the two ends to agree on a key. There is even some test data to test that your implementation is implementing null encryption properly.

Since null encryption is one of the algorithms that can be negotiated, this means that encryption is effectively also optional.

12.4.2 Why Protect the IP Header?

AH advocates claim AH is needed because it protects the IP header. But nobody has proposed any plausible reason why it's important to protect the IP header. Some people claimed to have really good reasons, but couldn't divulge the reasons because they were secret. Routers along the path cannot enforce AH's integrity protection, because they would not know the session key for the Alice-Bob security association. And once Bob receives the packet, all he needs to know is that the sender knew the session key and that the packet's integrity check verifies properly.

AH's desire to protect the IP header makes the protocol complicated to implement and compute-intensive. Some fields in the IP header get modified by routers, so they can't be included in AH's end-to-end integrity check. For example, the TTL field (which counts how many hops remain before the packet should be dropped) must be decremented by every router. So, AH specifies which fields in the IP header are mutable (*e.g.*, TTL), and therefore not included in the AH integrity check; which ones are immutable (and therefore included in the integrity check); and which fields are mutable but predictable (such as fragmentation information, which might change along the path, but what the packet will look like when reassembled is predictable). This means that an implementation can't simply compute an integrity check based on the packet but must zero out some fields (the mutable fields) before computing the integrity check.

A motivation for some of the AH people to want to protect the IP header was that they (being IPv6 enthusiasts) were frustrated that the world didn't immediately deploy IPv6. Note that if they really wanted the Internet to have bigger layer 3 addresses, they should have adopted CLNP, as recommended in 1992 by the IAB (Internet Architecture Board). CLNP had 20-octet addresses, was widely deployed at the time, and, in fact, is technically superior to IPv6 in many ways. But especially switching the Internet to bigger addresses in 1992, when the Internet was much smaller, would have been very easy.

So the IPv6 proponents hoped that IPsec (among other features) would be the motivator for moving to IPv6. Some IPv6 advocates proposed making it illegal to make any improvements (including IPsec) to IPv4, so that if the world wanted any of the stuff IETF designed (since the mid-1990s or so) it would have to move to IPv6. And they especially viewed NAT (Network Address Translation) as one of the reasons people continued to use IPv4 (because, indeed, NAT greatly expands the size of network that can be supported with IPv4). NATs allow networks to have addresses that are only locally meaningful, and the same block of locally meaningful addresses can be used in multiple networks. A node Alice inside such a network cannot be spoken to from outside unless Alice first initiates communication with an external node. If Alice does send a packet to Bob, where Bob's address is globally meaningful, the NAT box between Alice's network and the rest of the Internet will assign Alice a temporary globally reachable address. The NAT box will translate the source address on packets from Alice to Bob to Alice's temporary globally reachable address and translate the destination address on packets from Bob to Alice to Alice's internal address.

So, the AH designers saw an opportunity to “break NATs”. NATs need to modify the addresses in the IP header. If an AH-protected packet traversed a NAT box, the integrity check would fail. The vision of the AH designers was that as soon as AH was deployed, people would say, “NAT boxes make AH not work. Get rid of the NAT boxes.” But what people did say, after their network was working fine with NAT, and then AH got deployed and things stopped working, was “Turn off this AH stuff. It’s making things break.”

Note that there is some complication involved in making ESP and IKE work through NATs. The simplest strategy is to have IKE and ESP run over UDP port 4500 instead of directly over IP. Then both IKE and ESP will work through the NAT. The IKE spec specifies some elaborate mechanisms for Alice and Bob to figure out whether there is a NAT box between them, and if not, Alice and Bob can save the eight octets of the UDP header.

12.4.3 Tunnel, Transport Mode

The IPsec specification talks about two modes of applying IPsec protection to a packet. **Transport mode** refers to adding the IPsec information between the IP header and the remainder of the packet. **Tunnel mode** refers to keeping the original IP packet intact and adding a new IP header and IPsec information (ESP or AH) outside. (See Figure 12-4.)

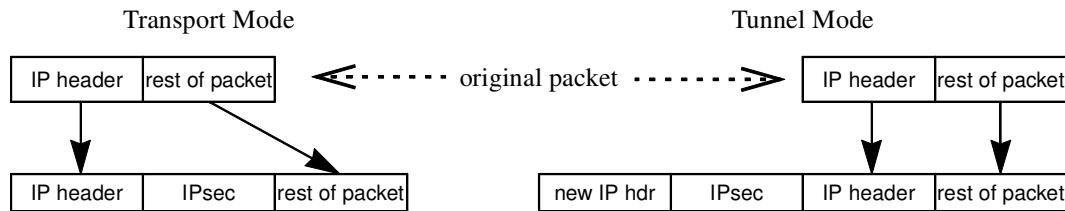
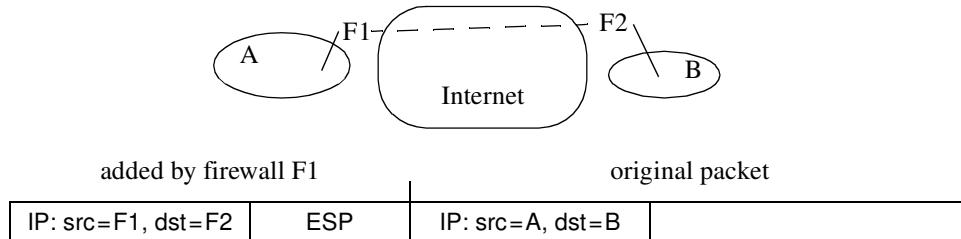


Figure 12-4. Transport Mode and Tunnel Mode

Transport mode is most logical when IPsec is being applied end-to-end. A common use of tunnel mode is firewall-to-firewall, or endnode-to-firewall, where the data is only protected along part of the path between the endpoints. Suppose two firewalls establish an encrypted tunnel to each other across the Internet (see Figure 12-5). They treat the tunnel as if it is an ordinary, trusted link. In order to forward packets across that link, F1 adds an IP header with destination=F2. When A launches an IP packet to destination B, it will have, in the IP header, source=A and destination=B.

When F1 forwards the packet to F2 across the encrypted tunnel, it will use IPsec tunnel mode. F1 will not modify the inner header, other than doing what any router would do when forwarding a packet, such as decrementing the hop count. The outer IP header added by F1 will have source=F1 and destination=F2. The inner header will be unmodified by the routers along the path between F1 and F2. Those routers will only look at the outer IP header.

**Figure 12-5.** IPsec, Tunnel Mode, Between Firewalls

Transport mode is not strictly necessary, since tunnel mode could be used instead. Tunnel mode just uses more header space since there will be two IP headers.

The same packet might have multiple layers of IPsec (ESP and/or AH) headers and might be multiply encrypted (see Figure 12-6). Suppose A and B are talking with an encrypted end-to-end connection. Their packets will contain an ESP header. When F1 forwards it across the tunnel to F2, F1 takes the entire packet (including the IP+ESP header) and adds its own IP+ESP header. F1 encrypts the entire packet it received, including the IP header, with the key that F1 shares with F2.

original packet as launched by A, encrypted with the F1–F2 key			
IP: src=F1, dst=F2	ESP	IP: src=A, dst=B	ESP

Figure 12-6. Multiply Encrypted IP Packet

Tunnel mode is essential between firewalls in order to preserve the original source and destination addresses; and as we said earlier, tunnel mode can be used instead of transport mode at the expense of adding a new IP header. Given that IPsec is too complex, many have argued that getting rid of transport mode would be one way of simplifying IPsec. But transport mode is such a small piece of the complexity of IPsec that we don't feel it's worth worrying about. Far more useful would be to get rid of AH.

12.4.4 IPv4 Header

The IPv4 header is defined in RFC 791. Its fields are

size	
4 bits	version
4 bits	header length (in 4-octet units)
1 octet	type of service
2 octets	length of header plus data in this fragment
2 octets	packet identification
3 bits	flags (don't fragment and last fragment)
13 bits	fragment offset
1 octet	hops remaining, known as TTL (time to live)
1 octet	protocol
2 octets	header checksum
4 octets	source address
4 octets	destination address
variable	options

50=ESP, 51=AH

For the purposes of this chapter, the most important field is the PROTOCOL field, which indicates what follows the IP header. Common values are TCP (6), UDP (17), and IP (4).

IPsec defines two new values for the PROTOCOL field in the IP header: ESP=50 and AH=51. For example, if TCP is on top of IP without IPsec, the PROTOCOL field in the IP header will be 6. If TCP is used with IP using AH, for instance, then the PROTOCOL field in the IP header will equal 51, and the PROTOCOL field in the AH header will be 6 to indicate that TCP follows the AH header. If the packet is encrypted using ESP, then the PROTOCOL field in the IP header will be 50 but the actual PROTOCOL field, the one that would have appeared in the IP header if encryption with ESP was not being used, will be encrypted and therefore not visible until the packet is decrypted.

12.4.5 IPv6 Header

The IPv6 header is defined in RFC 2460, and its fields are

# octets	
4	version (4 bits) type of service flow label
2	payload length
1	next header
1	hops remaining
16	source address
16	destination address

In IPv6, the equivalent field to IPv4's PROTOCOL is NEXT HEADER. It has the same values defined as the IPv4 PROTOCOL field, so ESP=50 and AH=51. IPv6-style extension headers (roughly equivalent to OPTIONS in the IPv4 header) are encoded as

# octets	
1	next header
1	length of this header
variable	data for this header

LENGTH OF THIS HEADER is in units of 8-octet chunks, not counting the first 8-octet chunk. AH looks like an IPv6 extension header, but its PAYLOAD LENGTH is in units of 4-octet chunks instead of 8-octet chunks (and, like other IPv6 extension headers, doesn't count the first eight octets). This violates one of the protocol folklore rules described in Chapter 17 *Folklore*, which is that the LENGTH field should always be defined the same way for all options, so that it is easy to skip over unknown options.

DATA FOR THIS HEADER is a sequence of options, each one **TLV-encoded**, which means a TYPE field, a LENGTH field, and a VALUE field. The TYPE field is one octet long, and one of the bits in the TYPE field for options that appear in some extension headers indicates whether the option is **mutable** (might change along the path) or **immutable** (relevant for AH; see §12.5). The mutable flag is only useful for AH, and if AH ever goes away, the flag in IPv6 will be very mysterious.

12.5 AH (AUTHENTICATION HEADER)

The AH header provides authentication only (not encryption) and is defined in RFC 2402. Its format is patterned after IPv6 extension headers, which all start with NEXT HEADER and PAYLOAD LENGTH (which gives the length of the AH header), except, as we said in the previous section, AH's PAYLOAD LENGTH is in different units than the equivalent field in an IPv6 extension header. AH is intended not only to protect the data but the IP header as well. In IPv4, the AH header must be a multiple of 32 bits, and in IPv6 it must be a multiple of 64 bits. So the AUTHENTICATION DATA field must be an appropriate size to make the header size be the right length.

Some integrity checks require the data to be a multiple of some blocksize. If the data is not a multiple of the blocksize, then AH is computed as if the data were padded to the proper length with 0s, but the 0s are not transmitted. The AH header looks like

# octets	
1	next header
1	payload length
2	unused
4	SPI (Security Parameter Index)
4	sequence number
variable	authentication data

The fields in AH are

- NEXT HEADER. Same as PROTOCOL field in IPv4. For example, if TCP follows the AH header, then NEXT HEADER is 6.
- PAYLOAD LENGTH. The size of the AH header in 32-bit chunks, not counting the first eight octets.
- SPI. Discussed in §12.1 *IPsec Security Associations*.
- SEQUENCE NUMBER. The sequence number has nothing to do with TCP’s sequence number. This sequence number is assigned by AH and used so that AH can recognize replayed packets and discard them. So, for example, if TCP retransmits a packet, AH will just treat it like a new packet and give it the next sequence number. AH will not know (or care) that this is a retransmitted TCP packet.
- AUTHENTICATION DATA. This is the cryptographic integrity check on the data.

12.6 ESP (ENCAPSULATING SECURITY PAYLOAD)

ESP allows for encryption and/or integrity protection. If you want encryption, you must use ESP. If you want integrity protection only, you could use ESP or AH. If you want both encryption and integrity protection, you could use both ESP and AH, or you could do both integrity protection and encryption with ESP. The security association database would tell you what to use when transmitting to a particular IP address. It’s a little odd to call ESP a “header” because it puts information both before and after the encrypted data, but everyone seems to call it a header so we will, too.

Technically, ESP always does encryption, but if you don’t want encryption, you use the special “null encryption” algorithm.

The presence of the ESP header is indicated by having the PROTOCOL field in IPv4 or the NEXT HEADER field in IPv6 equal to 50. The ESP envelope itself consists of

# octets	
4	SPI (Security Parameters Index)
4	sequence number
variable	IV (initialization vector)
variable	data
variable	padding
1	padding length (in units of octets)
1	next header/protocol type
variable	authentication data

The fields are

- SPI. Same as for AH, discussed in §12.1 *IPsec Security Associations*.
- SEQUENCE NUMBER. Same as for AH, explained in §12.5 *AH (Authentication Header)*.
- INITIALIZATION VECTOR. An IV is required by some cryptographic algorithms, such as encryption with CBC mode. Although the IV is variable-length (it may even be zero-length), it's fixed-length for a particular cryptographic algorithm. Once the SA is established, the cryptographic algorithm is known, and therefore the length of the IV field is fixed for the duration of the SA. This is also true of the field AUTHENTICATION DATA.
- DATA. This is the protected data, probably encrypted. If it is a tunnel-mode packet, then the beginning of the data is an IP header. If it is a TCP packet, and ESP is being used in Transport mode, then the beginning of the data is the TCP header.
- PADDING. Padding is used for several reasons: to make the data be a multiple of a blocksize for cryptographic algorithms that require it; to make the encrypted data be a different size than the plaintext so as to somewhat disguise the size of the data (limited because PADDING LENGTH is only one octet); and to ensure that the combination of the fields DATA, PADDING, PADDING LENGTH, and NEXT HEADER is a multiple of four octets.
- PADDING LENGTH. Number of octets of padding.
- NEXT HEADER. Same as PROTOCOL field in IPv4 or NEXT HEADER in IPv6, or NEXT HEADER in AH.
- AUTHENTICATION DATA. The cryptographic integrity check. Its length is determined by the authentication function selected for the SA; it is zero-length if ESP is providing encryption only.

If encryption is used, the fields DATA, PADDING, PADDING LENGTH, and NEXT HEADER are encrypted. The AUTHENTICATION DATA appears only if the security association requests integrity protection with ESP. If ESP integrity protection is used, all fields in the ESP (starting with SPI and ending with NEXT HEADER) are included in the ESP integrity check.

12.7 COMPARISON OF ENCODINGS

AH was designed by IPv6 fans and looks similar to IPv6 extension headers. (The only difference is that its LENGTH field is expressed in different units.) The ESP designers didn't really care about IPv6 and designed ESP to be as technically good as it could be, unconstrained with having to look like something else.

There are two wasted octets in AH (the “unused” octets) in order for all the fields to be on 4-octet boundaries. But AH can cleverly avoid padding the data. The integrity check is calculated as if the data were padded with zeroes to a multiple of a blocksize, but the padding is not transmitted. Therefore, if the data needed more than two octets of padding, AH winds up being smaller. Of course, this is assuming that IPsec is being used for integrity only, which as we said, we think will be rare. If the data is encrypted, then it requires much more overhead to use AH and ESP simultaneously than just using ESP for both integrity protection and encryption.

Having the MAC (the integrity check) appear before the data (as it is in AH) means that the data needs to be buffered and the integrity check computed before the packet can be transmitted. In contrast, in ESP, the MAC appears after the data.

Attempting to protect the IP header and needing to classify every field and every option according to whether it's mutable or immutable makes AH very complicated.

At one of the final IETF meetings before AH and ESP were finalized, someone from Microsoft got up and gave an impassioned speech about how AH was useless given the existence of ESP, cluttered up the spec, and couldn't be implemented efficiently (because of the MAC in front of the data). Our_{1,2} impression of what happened next was that everyone in the room looked around at each other and said, “Hmm. He's right, and we also hate AH, but if it annoys Microsoft, let's leave it in.”

12.8 HOMEWORK

1. Why isn't the SPI value sufficient for the receiver to know which SA the packet belongs to?
2. How is the integrity check processing more efficient with ESP than with AH?
3. Suppose Alice is sending packets to Bob using IPsec. Suppose Bob's TCP acknowledgment gets lost, and Alice's TCP, assuming the packet was lost, retransmits the packet. Will Bob's IPsec implementation notice that the packet is a duplicate and discard it?
4. Suppose you wanted the transmitter to assign the SPI rather than the receiver. What problems might this cause? Can it be made to work?

5. When sending encrypted traffic from firewall to firewall, why does there need to be an extra IP header? Why can't the firewall simply encrypt the packet, leaving the source and destination as the original source and destination?
6. Referring to Figure 12-5, assume that A and B are using IPsec in transport mode, and F1 and F2 have established an encrypted tunnel using IPsec. Assume A sends a TCP packet to B. Show the relevant fields of the IP header(s) as given to A's IPsec layer, as transmitted by A, as transmitted by F1, and as received by B.

13

SSL/TLS AND SSH

The concepts in TLS (Transport Layer Security) have been covered in Chapter 11 *Communication Session Establishment*, and the concepts are similar to IPsec. Alice and Bob authenticate and establish cryptographic keys for the session.

TLS grew out of Netscape's SSL (Secure Sockets Layer) protocol. When the IETF took it over to improve and standardize it, they renamed it TLS (Transport Layer Security). Since being called TLS, it has gone through three revisions, so the latest version is 1.3 (RFC 8446). There is no real logic to why the TLS versions were named TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3 rather than TLS version 1, TLS version 2, TLS version 3, and TLS version 4 or even why TLS 1.0 was not named SSL version 4. Most of the revisions were quite minor, but the changes from version TLS 1.2 to 1.3 are interesting, and we'll describe them here. We will mainly discuss TLS versions 1.2 and 1.3.

Credentials in TLS are usually PKIX certificates (§10.4.2 *Names in Certificates*), but because the first protocol to use TLS was HTTP (web browsing) and users in general don't have public keys or certificates, TLS allows for only the server to have a certificate. In most deployed scenarios, if the user needs to authenticate, she will authenticate herself using something like a username/password after the TLS session is established. As with IPsec, the TLS specification also allows for credentials between Alice and Bob to be a pre-shared secret key rather than certificates, but in TLS, this is a rarely used scenario. However, because authentication based on pre-shared keys was in the specification, it was a convenient way to efficiently resume a session (using the previous session secret as a pre-shared key).

13.1 USING TCP

TLS is designed to run in a user-level process and runs on top of TCP. As discussed in §11.7 *What Layer?*, running on top of layer 4 allows deployment of TLS in a user-level process rather than requiring operating system changes. Using TCP (the reliable layer 4 protocol) rather than UDP (the datagram layer 4 protocol) makes TLS simpler, because TLS doesn't have to worry about issues

such as timing out and retransmitting lost data, frame size limitations, and congestion avoidance. TLS could have retained the advantage of being easily deployable as a user-level process and still avoided the rogue packet problem discussed in section §11.7 *What Layer?* by running on top of UDP and doing all the timeout/retransmission work of TCP within SSL/TLS, but the decision was made to live with the rogue packet problem and keep SSL/TLS simpler. In fact, there is a version of TLS known as DTLS (Datagram Transport Layer Security, RFC 6347), which runs on top of UDP, and is based on TLS 1.2. There will be a version of DTLS based on TLS 1.3.

13.2 STARTTLS

Some protocols (for instance, email) were designed and deployed long before anyone thought about security. New implementations can be upgraded to run over TLS, and the goal is for the new implementations to talk over TLS if possible but still interoperate with legacy implementations. There are various solutions one could imagine:

1. Get a new port assigned to the protocol that TLS-capable implementations would listen on. However, getting new well-known ports is sometimes difficult.
2. Advertise in DNS whether a service is TLS capable.
3. Use a single port, but after the TCP connection is established, send a message, carefully crafted for that protocol that will be ignored by old implementations but recognized by new implementations as a desire to speak over TLS.

Some protocols (e.g., SMTP) have deployed option 3. In SMTP, the message is known as *StartTLS*. Since the StartTLS command is sent before there is any cryptographic protection, an attacker could remove that message and cause two TLS-capable nodes to speak without TLS.

Note that TLS itself can protect itself from a downgrade attack, where an attacker removes the strong cryptographic choices from the choices Alice offers (assuming Alice has not offered a set of cryptographic algorithms so weak that an attacker can break them before Alice and Bob time out the handshake). But there is no similar defense against an attacker stripping the StartTLS command (see Homework Problem 2).

Note that option 1 also has the problem of having an attacker prevent two TLS-capable implementations from using TLS. The attacker, on the path between the two nodes, can throw away packets sent to the new port, which will cause the initiator to time out the attempt to speak over TLS and, instead, revert to non-TLS communication.

13.3 FUNCTIONS IN THE TLS HANDSHAKE

The handshake accomplishes a variety of functions, some of which are optional:

- Negotiating version numbers (the client gives a list, the server chooses).
- Negotiating which cipher suite to use (the client gives a list, the server chooses).
- Having the client specify which DNS name the client wishes to speak to (if multiple services are sharing the same IP address). This field is known as SNI (Server Name Indication).
- Sending Diffie-Hellman values for perfect forward secrecy.
- Having the server send its certificates (either a single certificate signed by a CA that the server assumes the client has as a trust anchor or a chain of certificates from (hopefully) a client's trust anchor).
- Having the server send a session ID so the client can resume a session, or create a parallel session cheaply (similar in spirit to an IPsec child-SA).
- Having each side send nonces so the session key will be different each time the session is resumed.
- Optionally authenticating the client. The server specifies whether it accepts client authentication, and, optionally, what trust anchors the server recognizes and which extensions the server would like to see in the client certificate.
- Having the server prove knowledge of its private key (by decrypting something or signing something, depending on the TLS version number and cipher suite) and cryptographically protecting all messages in the exchange to avoid a downgrade attack (having an attacker remove offered cipher suites or version numbers).

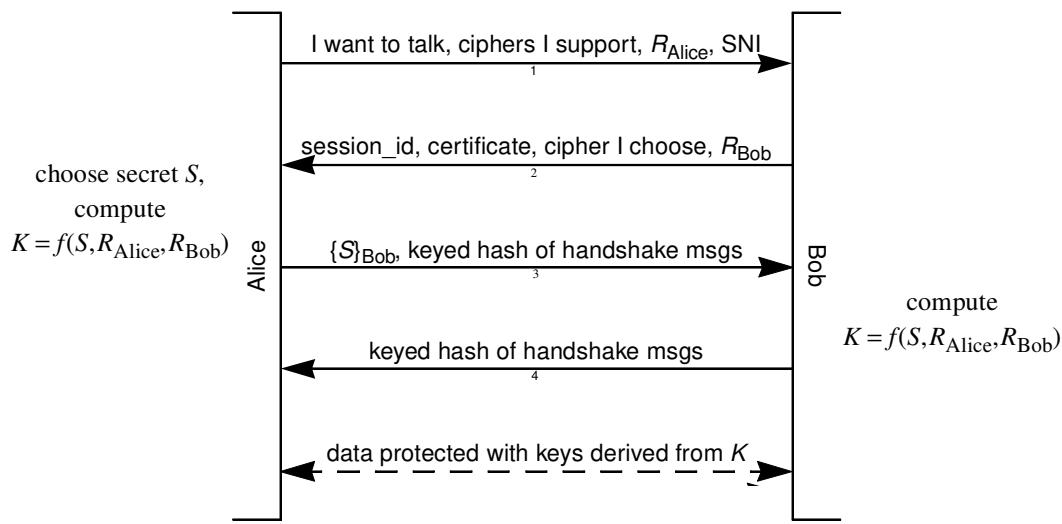
13.4 TLS 1.2 (AND EARLIER) BASIC PROTOCOL

Using the reliable octet stream service provided by TCP, SSL/TLS partitions this octet stream into records, with headers and cryptographic protection, to provide a reliable, encrypted, and integrity-protected stream of octets to the application. There are four types of records: user data, handshake messages, alerts (error messages or notification of connection closure), and change cipher spec (which should be a handshake message, but they chose to make it a separate record type).

In the basic protocol, the client (Alice) initiates contact with the server (Bob). Then Bob sends Alice his certificate. Alice verifies the certificate, extracts Bob's public key, picks a random number S from which the session keys will be computed, and sends S to Bob, encrypted with Bob's

public key. Then the remainder of the session is encrypted and integrity protected with those session keys. If Bob correctly computes the session key, Alice can know that Bob knows the private key associated with his certificate. Note that depending on the cipher suite selected and the TLS version, there might be many secrets derived from S . For instance, there might be encryption keys and integrity keys in each direction (for CBC mode).

First, we'll present a simplified form of the protocol (Protocol 13-1), then discuss various issues in the full protocol, and finally discuss the details.



Protocol 13-1. (simplified) SSLv3 through TLS 1.2

The simplified protocol consists of four messages that establish a shared master key:

- Message 1. Alice says she would like to talk (but doesn't identify herself) and gives a list of cryptographic algorithms she supports, along with a random nonce R_{Alice} , that will be combined with the S in message 3 to form the various keys.
- Message 2. Bob sends a session ID, his certificate, a nonce R_{Bob} that will also contribute to the keys, and Bob's choice of cipher suites from among those listed by Alice in message 1.
- Message 3. Alice chooses a random number S (known as the **pre-master secret**) and sends it, encrypted with Bob's public key. She computes the **master secret K** , which is a function of S and the two nonces R_{Alice} and R_{Bob} . She sends a keyed hash of the previous messages (using key K) both to prove she knows the same key K as Bob will compute and to ensure that tampering of the previous handshake messages will be detected. To ensure that the keyed hash Alice sends is different from the keyed hash Bob sends, each side includes a constant ASCII string in the hash, with a different string for the client than the server. Surprisingly (and

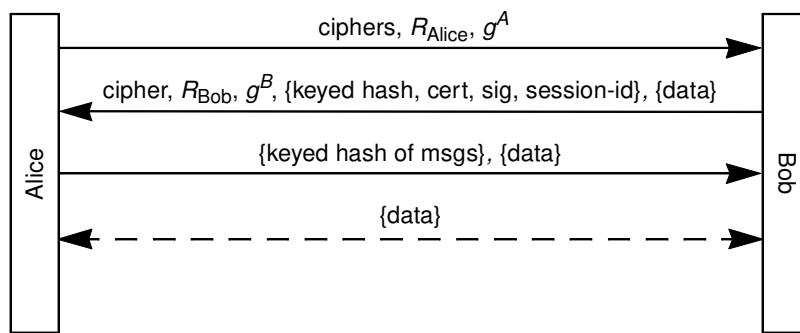
unnecessarily), the keyed hash is sent encrypted and integrity protected. The keys used for encrypting the keyed hash and the rest of the data in the session are derived from hashing K , R_{Alice} , and R_{Bob} . The keys used for transmission are known as *write* keys, and the keys used for receipt are known as *read* keys. So, for instance, Bob’s write-encryption key is Alice’s read-encryption key.

- Message 4. Bob proves he knows the session keys and ensures that the early messages arrived intact, by sending a keyed hash of all the handshake messages, encrypted with his write-encryption key and integrity protected with his write-integrity key. Since the session keys are derived from S , this proves he knows Bob’s private key because he needed it in order to extract S .

At this point, Alice has authenticated Bob, but Bob does not know Alice’s identity. As deployed today, authentication is seldom mutual—the client authenticates the server, but the server does not authenticate the client. The protocol allows optional authentication of the client if the client has a certificate. But in the most common case today, if the application on the server wishes to authenticate the user, authentication information (such as a username/password) is sent as data over the established TLS session.

13.5 TLS 1.3

Unlike the version changes from SSLv3 to TLS 1.0 to TLS 1.1 to TLS 1.2, where the differences were largely incremental in response to some subtle security vulnerability, the changes in TLS 1.3 were more dramatic and aimed at improving performance. TLS 1.3 rearranges the fields within messages in order to reduce the total number of round trips required to complete a handshake and begin sending application data. (See Protocol 13-2.) TLS 1.2 introduced cipher choices that



Protocol 13-2. TLS 1.3 Session Initiation If No Previous State

allowed perfect forward secrecy by having a Diffie-Hellman exchange at the beginning of the session using ephemeral keys and signing them instead of having Alice encrypt the pre-master secret using Bob's long term public key. TLS 1.3 eliminates all cipher suites without perfect forward secrecy. That allows encryption of messages to begin before the handshake has completed, keeping more information away from the prying eyes of at least passive eavesdroppers. There are several different Diffie-Hellman groups that can be negotiated for use with TLS. The most common and best performing choice is one of the ECDH curves [§6.7.1 *Elliptic Curve Diffie-Hellman (ECDH)*]. For the best performance, Alice has to correctly guess one that will be acceptable to Bob and send a g^A value on speculation. If it is not acceptable to Bob, he will tell Alice what group she should use at the cost of an extra pair of messages. Bob will send his chosen cipher in message 2, and Alice will send a new message 1, using a Diffie-Hellman group acceptable to Bob.

13.6 SESSION RESUMPTION

SSL/TLS assumes that a session is relatively long-lived, from which many connections can be cheaply derived. This is because it was designed to work with HTTP 1.0, which had a tendency to open lots of TCP connections between the same client and server (one per item on the web page).

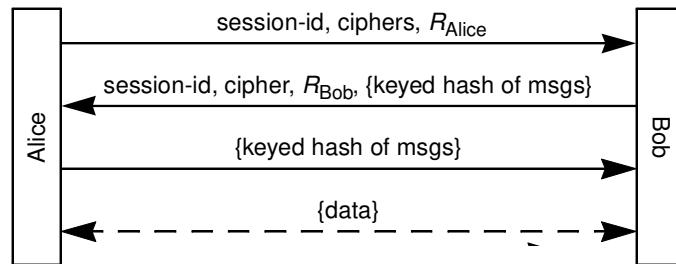
The per-session master secret is established using expensive public key cryptography. Multiple connections can be cheaply derived from that master secret by doing a handshake that involves sending nonces (so that session keys will be unique) but avoids public key operations.

SSL/TLS calls this *session resumption*, though it does not require that one connection end before the next is started. It is common for many connections to run in parallel. During the initial (expensive) handshake, Bob sends Alice a `session_id` that can be used later if Alice wants to set up a second connection to Bob with the same cryptographic algorithms but with fresh keys and bypassing the public key operations.

In TLS 1.2 and earlier, resumption requires that both Alice and Bob remember the `session_id` and master key from the previous connection. (See Protocol 13-3.) In TLS 1.3, the `session_id` provided by Bob is large enough to carry all the state that Bob needs. The `session_id` is sent in the clear, so it would be poor security practice if things like the session key were transmitted in the clear. So Bob should encrypt all the state he needs with a key only he knows and use that quantity as the `session_id`. It is not necessary for Bob to use the `session_id` this way, but TLS 1.3 allows it.

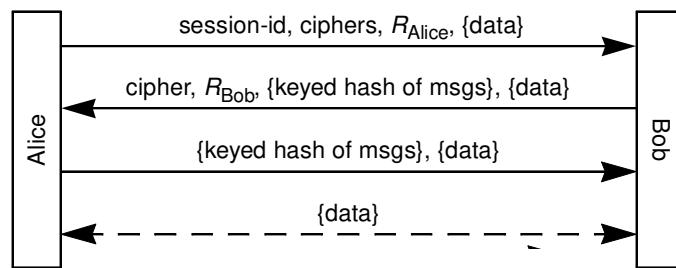
If Bob has forgotten the `session_id` (perhaps because he crashed or because it has timed out), he ignores the `session_id` that Alice presents and proceeds with the full handshake and likely presents a new `session_id` for Alice to use in the future.

It might seem odd for Alice, when resuming the session, to send a set of cipher suites rather than just the suite used in the session. Even if Bob has lost state about the session, wouldn't he

**Protocol 13-3.** Session Resumption for TLS 1.2 and Prior

make the same choice of cipher suite, given the same choices, as he made when the session was created? Not necessarily, because perhaps his policy has changed. So Alice is allowed, when resuming a session, to send a set of choices, but the set must at least contain the cipher suite that had been chosen previously for that session.

Session resumption for TLS 1.3 looks like Protocol 13-4:

**Protocol 13-4.** TLS 1.3 Session Resumption

13.7 PKI AS DEPLOYED BY TLS

As deployed today, the client typically comes configured with public keys of various “trusted” organizations (trusted by the browser vendor, not necessarily by the user). (See §10.6.4 *Oligarchy*.) In some implementations, the user at the client machine can modify this list, adding or deleting keys. The server sends a certificate (or certificate chain) to the client, and if it’s signed by one of the CAs on the client’s list, the client will accept the certificate. If the server presents a certificate signed by someone not on the list (such as a self-signed certificate), the user is typically presented with a pop-up box informing him that the certificate couldn’t be verified because it was signed by

an unknown authority, and would the user like to go to the site anyway? We discuss this model of PKI along with others in §10.6 *PKI Trust Models*.

If the server wishes to authenticate the client using certificates, it sends a certificate request in which it specifies the X.500 names of the CAs it trusts and the types of keys it can handle (*e.g.*, RSA or DSS). This was a strange piece of asymmetry in TLS 1.2 and earlier, since the client did not get to specify to the server what sort of certificate chain or key type it wanted. TLS 1.3 does allow the client to specify this, although most implementations don't.

13.8 SSH (SECURE SHELL)

SSH has a lot in common with SSL/TLS, and since they are both standardized by the same standards body (IETF), there has been a converging of their characteristics over time. When the IETF was choosing an algorithm to be TLS, SSL and SSH were two of the candidates. In the end, TLS was based almost entirely on SSL. It is possible that all use of SSH will convert to using TLS eventually, but there are some important differences that allow SSH to maintain its ecological niche, so it probably will not go away.

While SSL was originally designed to protect HTTP traffic carried between web browsers and web servers, SSH was originally designed to protect remote login sessions. In Unix, the command line processing application is known as the shell, and SSH secured connections to the shell, hence the name Secure SHell.

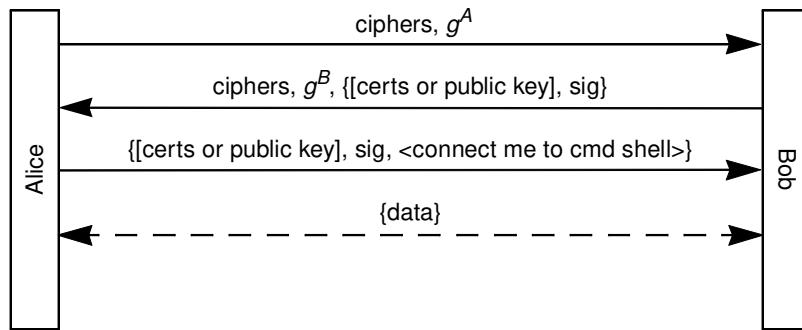
Like SSL/TLS, SSH runs over TCP/IP in order to get its reliability through retransmission. As with SSL/TLS, and unlike IPsec, an attacker can disrupt an SSH connection by injecting a single packet that confuses the TCP connection state. The cryptographic integrity check will ensure that the invalid data is not processed, but the two ends of the conversation do not preserve enough state to recover, so the connection will close.

Unlike SSL/TLS, which carries a single TCP connection as its payload, SSH is capable of initiating multiple TCP connections that are carried over a single SSH session. That means that all the clever things SSL/TLS does to efficiently create multiple parallel connections between a single pair of nodes are unnecessary with SSH. SSH can create additional connections within the cryptographic envelope of an existing SSH session. They will all be encrypted and integrity-protected with the same keys, but they are labeled to keep the connections from getting confused with each other, and there are no security weaknesses from cryptographically protecting them all with the same keys.

When SSL was first developed, a second TCP port was allocated for HTTP connections that were made using SSL. By convention, HTTP connections were made to port 80, and HTTP over SSL connections were made to port 443. When people wanted to use SSL to protect connection

types other than HTTP, they needed to allocate a second port to each protocol that wanted to optionally use SSL, and about 60 ports have been so assigned. Seeing that this was going to allocate a lot of ports from a scarce resource pool, many more protocols start up unencrypted and negotiate use of SSL/TLS. They then switch over to an SSL/TLS initial exchange on the same port.

The message sequence (Protocol 13-5) is a lot like that of TLS, but with some differences.



Protocol 13-5. SSH Session Initiation

Alice begins by sending the set of cipher suites she supports, and (assuming the cipher suite uses Diffie-Hellman for PFS) speculatively can send g^A . These lists include not just cryptographic algorithms, but also the forms of authentication she can use to prove her identity and the forms of authentication she will accept from Bob. If Alice guessed wrong as to what group to use, then when she sees Bob's cipher suite list, she will figure that out and immediately send a g^A from the correct group. So there is no need for an error message and a retry. Once Bob sees Alice's lists and g^A , he can not only send g^B but can start encrypting and send his certificates or public key and a signature on a collection of fields from previous messages. Since SSH does not engage in session resume, there is no need for nonces. If Alice or Bob want to rekey in the middle of a connection, they exchange new Diffie-Hellman values (or whatever PFS mechanism is used in the chosen cipher suite).

SSH can multiplex many connections over a single SSH session. After Alice and Bob create the SSH tunnel, Alice and Bob can negotiate connections that will be transmitted over the tunnel. SSH has a header comprising a 1-octet command followed by a 4-octet connection identifier, followed by a 4-octet length. As with IPsec, Alice and Bob will tell the other side what to use as a connection identifier. For example, assume Bob assigned two connections to be identifier C_1 and C_2 , respectively. If Alice is sending data from C_1 , the octet stream she will send will consist of an octet identifying what follows is data, followed by C_1 , followed by the length of the data, followed by the data. Then if she wants to send data for C_2 , she will send the octet identifying this as data, followed by C_2 , followed by the length of data to send on C_2 .

Another type of command is “close a connection”, followed by the connection identifier.

13.8.1 SSH Authentication

While some implementations of SSH support authentication using X.509 certificates (as with TLS), that is not its most common configuration. SSH does not depend on any sort of PKI, and can instead authenticate using raw public/private key pairs (rather than certificates). This is useful in the original use of SSH, where a user connects to a remote machine and starts a terminal session with a command prompt. When Alice connects to Bob for the first time, the software on Alice's machine has no idea what Bob's public key should be, nor is Alice configured with trusted CA keys. So when Bob presents his public key, Alice gets a prompt "Bob says his public key is 2832734...3623. Do you want to trust it?" Alice could look up the public key from some trusted source, but in practice she just says "Yes", taking a leap of faith. She doesn't really know whether she's connected to Bob or some attacker impersonating Bob, so she will be suspicious. But the software on her machine will remember Bob's public key and make sure it matches the next time she connects. In theory, this exchange is not very secure. But a Bob impersonator would have to impersonate Bob every time Alice tries to connect and would have to respond enough like Bob so as to not make Alice suspicious. So in practice, this mechanism is quite secure unless Alice is always using fresh clients so that Bob's public key is never remembered. It is also common for administrators to configure public keys of well-known hosts, and that mechanism is even more secure.

This prompting assumes that Alice is a person and not some automated process. If an automated process is going to be routinely connecting to Bob, it will have to be configured with Bob's public key. This can be done by having the administrator configure the key or by initially opening a connection to Bob and answering "Yes" to "Do you want to trust the key?"

There are many options for Alice to authenticate to Bob. If Alice is an automated process (or a human needing enhanced security), she will probably be configured with a public/private key pair where her public key is configured at Bob. If she is a human, she will likely use a username and password or some single sign-on mechanism [see §9.15 *Identity Providers (IDPs)*]. If she has a private key and certificate, Bob can also be configured to accept that (though this is uncommon).

13.8.2 SSH Port Forwarding

Another important application of SSH is called **port forwarding**. It can be used with any application that uses TCP or UDP ports, transparently adding encryption and integrity protection while tunneling the application data over an insecure network and frequently tunneling traffic through corporate firewalls that would block it if they knew what was being tunneled. This makes SSH port forwarding a favorite tool of users and attackers alike!

It works as follows. Alice opens an SSH session to Bob. To allow a legacy process (that is not SSH capable) to benefit from SSH protection when traversing the network between Alice and Bob, the legacy process is configured to speak to Alice's IP address and a port on Alice's machine allo-

cated for this purpose. Alice tunnels the packet over the SSH connection to Bob. Bob's SSH process strips off the cryptography and forwards the tunneled legacy packet to the appropriate destination process.

This functionality is built into the SSH protocol. This functionality could be implemented on top of TLS by defining a TLS client demon that listened on selected ports and forwarded connections over the TLS connection to a TLS-based service that opened outbound connections and forwarded the traffic. But that would be a lot of work. The port forwarding functionality is part of the SSH base specification, which is one reason SSH maintains its ecological niche.

Firewall administrators don't want to block SSH because it is useful for reaching out to remote nodes and opening terminal services, but once the connection is open, the firewall can't tell what it is being used for. In general, anyone can tunnel anything over anything (*e.g.*, HTTP or DNS lookups), but allowing SSH through makes it easy.

13.9 HOMEWORK

1. Some certificates specify that a public key should only be used for signatures or only for encryption. Some public key algorithms only work for signatures or only work for encryption. In TLS 1.2, does Bob's public key have to work for encryption, for signatures, or both? How about TLS 1.3?
2. How does TLS defend against a downgrade attack (where an attacker removes the most secure cryptographic algorithm from Alice's suggestions, causing Alice and Bob to communicate using a less secure cryptographic algorithm)? Why would this defense not work for StartTLS (see §13.2 *StartTLS*)?
3. In TLS 1.3 (Protocol 13-2), if Alice guesses a Diffie-Hellman group that Bob does not support, Alice and Bob start from the beginning, this time with Alice knowing (based on Bob's message) a Diffie-Hellman group that Bob will support. However, note that Bob knows after receiving Alice's first message which Diffie-Hellman group he will choose, so he could send his Diffie-Hellman value in his reply, even though he will ignore Alice's Diffie-Hellman value. Redesign the protocol to take advantage of having Bob immediately send his Diffie-Hellman value in this case.

14

ELECTRONIC MAIL SECURITY

The first thing that springs to mind with the phrase *email security* is a message from Alice to Bob signed by Alice with her private key and encrypted with Bob's public key. Lotus Notes had a proprietary, widely deployed implementation of encrypted email in 1989. PGP was created and released as open source by Phil Zimmermann in 1991. Various email standards for user-to-user signed and encrypted email were developed over twenty years ago, including

- PEM [RFC 1421—Feb 1993]
- PGP/GPG [RFC 2015—October 1996]
- S/MIME [RFC 2311—March 1998]

In the early 1990s, there were deployment barriers such as export controls and patents, but these have been resolved. So it is somewhat astonishing that none of these, or anything implementing the model signed-by-Alice and encrypted-for-Bob, are widely used today. Encrypted and signed email is used within some organizations such as the U.S. Government, which has deployed a hardware-based PKI system using S/MIME encryption, but end-to-end encrypted email is not used by the vast majority of Internet users. Why didn't it catch on as the default mechanism for sending mail? Perhaps it was too difficult for user Alice to find Bob's public key or to maintain her own private key across multiple devices. Perhaps Bob was unhappy with losing all of his received email when he forgot his password. Perhaps large companies discouraged their employees from using end-to-end encrypted email. Perhaps the effort of using end-to-end encrypted email was never worth it, because most people Alice might send email to didn't support receiving encrypted email. Or perhaps users just didn't care.

There are other email-specific security challenges that are not addressed by end-to-end encryption and signing. Without defenses, our inboxes would be buried in spam. Buggy email processing software allows malicious email to carry malware that could infect the user's device. Often, blame is placed on the user for having “opened a dangerous email” or “clicked on a suspicious link in an email”. However, it should be possible (and the world should aspire) to have safe email-handling software. Opening an email or clicking on a link in an email should not infect user Alice's device or do other horrible things such as emailing something to everyone in her address

book. We suspect that it's just easier to blame the user than to develop safe email client software. The industry is aware of the problem, and there is ongoing research, including DARPA's SafeDocs initiative.

Another impediment to deploying end-to-end encrypted email is that companies require access to the plaintext of all email from or to the organization, perhaps for legal reasons, or perhaps to attempt to prevent company confidential information from being sent out of the company. Theoretically, this could be accomplished by having the company keep copies of the private keys of all their employees, but that would be inconvenient for the IT department and distract them from what they really like to do, such as forcing users to change their passwords every sixty days. Some security features interfere with each other. For example, end-to-end encryption interferes with a middle-box's filtering spam before it gets to the client device.

This chapter deals with the conceptual challenges and potential solutions to various email security issues. Although there are products that address some of the issues we discuss, we are not intending to describe specifics of any particular product. Also, the exact deployed mechanisms and issues depend on current implementations (including their bugs). So we will concentrate instead on broad conceptual issues.

14.1 DISTRIBUTION LISTS

Electronic mail allows a user to send a message to one or more recipients. The simplest form of electronic mail message is where Alice sends a message to Bob.

```
To: Bob  
From: Alice  
  
Care to meet me in my apartment tonight?
```

Usually, a mail system allows a message to be sent to multiple recipients, for example:

```
To: Bob, Carol, Ted  
From: Alice  
  
Care to meet me in my apartment tonight?
```

Sometimes it is impossible or inconvenient to list all recipients. For this reason, mail is often sent to a **distribution list**, a name that stands for a set of recipients. There are two ways of implementing distribution lists:

The first way (Figure 14-1) involves sending the message to a site at which the list is maintained, and that site then sends a copy of the message to each recipient on the list. We'll call that the **remote exploder method**. Note that a more commonly used term is a **mail reflector**, but that term sounds to us like a service that will send any messages you transmit back to you.

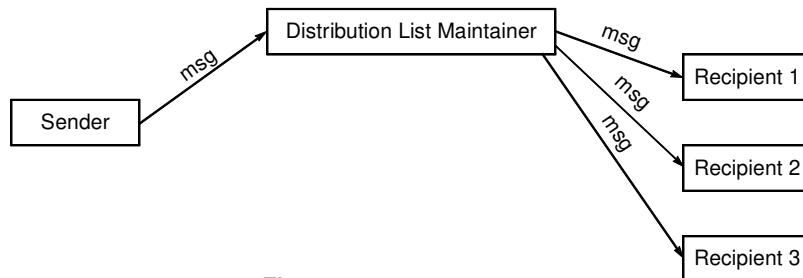


Figure 14-1. Remote Exploder

The second method (Figure 14-2) is for the sender to retrieve the list from the site where it is kept and then send a copy of the message to each recipient on the list. We'll call that the **local exploder method**.

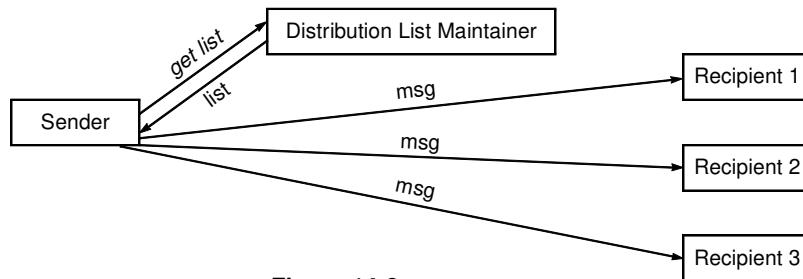


Figure 14-2. Local Exploder

Sometimes a member of a distribution list can be another distribution list. For instance, the mailing list **Security Customers**, used to advertise security products, might include **law enforcers**, **bankers**, **locksmiths**, and **members of organized crime**. It is possible to construct a distribution list with an infinite loop. Suppose someone is maintaining a mailing list for cryptographers. Someone else is maintaining one for cryptanalysts. The cryptanalysts point out that they also want to hear the information sent to the cryptographers mailing list, so the distribution list **Cryptanalysts** is added to the **Cryptographers** mailing list. And for similar reasons, **Cryptographers** is added to **Cryptanalysts**. The mail system must handle infinite loops in distribution lists in a reasonable manner, *i.e.*, it must send each recipient at least one copy of each message but not an

unreasonable number of copies of any. Loops like this effectively merge the mailing lists. (See Homework Problem 1.)

The advantages of the local exploder method are:

- It is easier to prevent mail forwarding loops.
- If there are multiple distribution lists, it is possible for the sender to prevent duplicate copies being sent to individuals on multiple lists.
- It is easier for the sender to know in advance how much bandwidth will be consumed to transmit the message.

The advantages of the remote exploder method are:

- It allows you to send to a list whose membership you are not allowed to know. (*To U.S. spies living abroad from the IRS: friendly reminder—the tax deadline is April 15. Being caught or killed is not one of the grounds for automatic extension.*)
- If distribution lists are organized geographically, you need send only one copy of a message over an expensive link to a remote area. (*To Citizens of France from the U.S. government: Thanks for the big statue.*)
- When distribution lists are included on distribution lists, it would be time-consuming to track down the whole tree to get to all the individuals. Instead, the message can be making progress as it is sent to the various exploders. Parallelism is exploited.
- When the distribution list is longer than the message, it is more efficient to send the message from the sender to the distribution list site than to send the distribution list to the sender. (*To people of planet earth: Greeting. Unless you stop transmitting reruns of I Love Lucy to us we will be forced to destroy your planet.*)

14.2 STORE AND FORWARD

It might seem simplest for electronic mail from Alice to Bob to be sent directly from Alice's personal device to Bob's personal device. However, in order for a message to be successfully delivered under that scenario, it is necessary for both Alice's and Bob's devices to be running and reachable from each other on the network at the time Alice wants to send the message. This might be especially inconvenient if the user devices are only occasionally connected to the network, e.g., laptops that the users only occasionally plug into the network. Also, most client devices do not have stable globally addressable IP addresses. A client device's address is often only valid within its local network, and so it is not addressable from outside that network. Communication with that client from outside its network is only possible if the client device initiates a connection to the external node, in

which case NAT (Network Address Translation) assigns the client a temporary globally reachable address (IP address plus port) so that it can receive return traffic. If both source and destination have only local addresses in different networks, neither side would be able to initiate a connection to the other unless they have both registered with some sort of relay service with a global address. Instant messaging services deploy such infrastructure, but typically email does not. Even if IPv6 were universally deployed, firewalls will most likely block most incoming connections from the Internet.

Mail forwarders known as **Message Transfer Agents**, or **MTAs** solve these issues. Instead of user Alice sending mail directly to user Bob's client device, the mail is instead sent to an MTA that is more or less permanently on the network. When user Bob's device attaches to the network, it retrieves his mail from the server that handles mail for his account. The mail processing at the source and destination devices is done by a program known as the **User Agent** or **UA**. Mail gets forwarded from UA to MTA to MTA to MTA to UA.

MTAs hold email for their users and send the email to the user when the user connects. But an MTA might provide additional services. It might provide spam filtering or virus checking or might examine email sent to addresses outside the company to detect and perhaps delete emails containing company confidential information.

14.3 DISGUIISING BINARY AS TEXT

When email was originally designed, the assumption was that its sole purpose was for human-to-human text using the English alphabet. Although most email infrastructure has been modernized to allow sending more than 7-bit ASCII text, it still can't reliably send arbitrary data (such as images or encrypted data) without performing some encoding of the data.

Even with English ASCII text messages, different platforms used somewhat different text formats. For instance, different platforms used different line delimiters. They might use <LF> (line feed, ASCII 10) or <CR> (carriage return, ASCII 13) or <CR><LF> (<CR> followed by <LF>). Or they might have line-length limitations, and, if there were too many characters between delimiters, a platform might add line delimiters or truncate. Or if there was white space at the end of a line (spaces or tabs), a platform might delete the white space. Some systems expected parity on the high-order bit of each octet. Others wanted the high-order bit to always be 0, so if they saw an eighth bit set to 1, they would change it to 0.

These sorts of transformations are mostly harmless when trying to send simple English character text messages from one user to another across diverse systems, but it causes problems when trying to send arbitrary data between systems. Certainly, if you're trying to send arbitrary binary data, having all your high-order bits cleared or having <CR><LF> inserted periodically will hopelessly mangle the data. But even if you are sending unencrypted text, security features can get

broken if the data is modified. For instance, a digital signature covering the contents of an email will no longer verify if any of the message is modified. It's difficult to predict exactly what kinds of modifications the email infrastructure, designed around the third day of Creation, could make to your file in its attempt to be helpful.

To send data other than simple text (*e.g.*, diagrams, voice, or pictures), there are various encoding standards such as BASE64 [RFC 4648] that will pass through all known mailers. These schemes treat the input as an arbitrary-length string of octets, partition the string into 6-bit chunks, and encode each group of six bits into an octet that represents only “safe” ASCII characters, with an occasional end-of-line delimiter to keep the mail infrastructure happy. There are obviously fewer than 256 safe values for an octet (otherwise, arbitrary binary data would work just fine). There are at least 64 safe characters to choose from (A–Z, a–z, 0–9, +, /, _, -, __,...). The encoding packs six bits of data into each 8-bit character by translating each of the possible 6-bit values into one of the 64 chosen safe characters, adding line delimiters sufficiently often. This encoding expands the data by about a third. Note that it is possible to be more efficient. By using more safe characters, BASE85 expands the data by a quarter, using five characters to encode four octets of data rather than BASE64, which uses four characters to represent three octets. This overhead is often mitigated by doing compression, and sometimes the BASE64-encoded compressed data is smaller than the original.

Most email infrastructure today supports UTF8, which enables sending most of the 256 octet values without modification. This became absolutely necessary once the Internet community realized that not everything is written in English. There are over a hundred thousand international characters that need to be expressed in order to say things in any of the currently supported languages. That takes more than eight bits per character, and there are various encoding schemes for using multiple octets to express a character. However, even if the email infrastructure could support all 256 octet values, mail gateways might insert end-of-line characters or remove whitespace, so BASE64 encoding is commonly used when transmitting arbitrary binary data such as images or digital signatures.

14.4 HTML-FORMATTED EMAIL

Email can be encoded in HTML, which means that the email has much of the power of a web page. It can embed URLs to fetch images, links to be clicked on, and JavaScript code that will execute. Suppose Alice is sending a message to user Bob. Most email clients will ask Bob if he wants to download pictures. If Bob does not say yes, legitimate email may be unintelligible, or at least look ugly, since many legitimate senders are using HTML to create “user friendly” email. If Bob does

agree to download pictures, and if Alice had cleverly embedded a recipient-specific URL containing Bob's email address, Alice will be able to know if and when Bob opened the message.

If Bob clicks on a link in a received email, Bob's device will perform whatever action the link specifies, and Bob will be exposed to any evil that a web page can dish out, including bugs that might infect his device or malicious content that might trick him into entering security-sensitive information.

The very rich functionality of HTML also means that it is likely there will be bugs in user Bob's email client that can be exploited by malicious email messages.

14.5 ATTACHMENTS

Email messages can contain attachments, and although it ought to be safe to open something like a Microsoft Word or PowerPoint document, it unfortunately is not. Clicking on an attachment may run the application associated with that document type. Some types are dangerous because they contain arbitrary code, such as .exe or .cmd. Others ought to be safe—a Word or PowerPoint document should merely display stuff, but, because of bugs or super fancy features that enable wondrous things like having the recipient digitally sign a document, it is not safe.

Many email clients or MTAs are configured to delete attachments with certain file name extensions (*e.g.*, .exe). Word and PowerPoint documents now have different file types depending on whether they should be harmless (*i.e.*, no dangerous features) or scary. Should-be-harmless Word and Powerpoint files have the type .docx and .pptx, whereas scary ones are of type .docm and .pptm, where m stands for *macro* (or, if you prefer, *malicious*).

14.6 NON-CRYPTOGRAPHIC SECURITY FEATURES

14.6.1 Spam Defenses

Spam is unwanted, unsolicited email, and there is so much of it that, without something identifying and deleting most of it before it gets to you, all resources involved in handling email for you (*e.g.*, your own time, your inbox storage, your MTA, the network delivering traffic) would become saturated with spam. Example reasons why a sender would want to send spam are to cheaply advertise a product or service to a large audience, trick people into divulging personal information (such as credit card numbers), trick people into sending money (a “small processing fee” in order to receive

their prize of millions of dollars), or to distribute malware by tricking a recipient into running malicious software. Even if only one in ten thousand recipients can be tricked by an email saying

I am the widow of a prince in Elbonia and need to find someone trustworthy to move my money out of the country. I will pay you 10% of my fortune of \$315 million if you help me.

if a sender targets a hundred million recipients, they will successfully reach many gullible people who will respond.

It is very easy and inexpensive to send email to huge numbers of recipients. If spammers needed to send individual copies of email from their own accounts, they would be limited in how much they could send. They can magnify the bandwidth they have available by using distribution lists, renting a name and address on a cloud, or using a bot army from which to send spam.

Spammers, including supposedly legitimate commercial and political organizations that want to advertise inexpensively to large numbers of potential customers, would love to have a huge database of live email addresses. Better yet would be email lists that are sorted for specific types of products or political affiliation.

How would a spammer know email addresses to send spam to? Email addresses can be fairly easily collected on the Internet (*e.g.*, in documents or posts on mailing lists), or they can be purchased from sites that collect email addresses (“for a discount coupon, sign up with your email address”). Or, since it is so inexpensive to send, a spammer can just try many combinations of potential username strings at some email domain (aaaa1@hotmail.com, aaaa2@hotmail.com). If someone wants to sell lists of email addresses, the list will be more valuable if the majority of addresses are legitimate. Verification of email addresses can be done by sending an email to a wide set of guessed email addresses and giving recipients the option to “unsubscribe” and saying what their email address is. Many MTAs will not send an explicit error message “username not found” to avoid giving hints about which email addresses at their domain are legitimate.

What can be done to combat spam? Some people attempt to make it slightly harder for programs that search the Internet to find their email address but still allow humans to find the email address by posting their email address as something like radia at alum dot mit dot edu.

Unfortunately, any service that attempts to identify spam will have some false negatives (*i.e.*, spam that gets through the filter) and false positives. Often, instead of deleting email that an automatic filter thinks might be spam, the email system puts it into a special spam folder. If legitimate email gets misclassified, a user could (and should) scan the spam folder to see if any legitimate email was misclassified as spam.

What techniques would an anti-spam service look for to try to classify an email as spam? Any technique will become known to spammers shortly after it is deployed, and they can tailor their emails to fool the filter. A filter might guess something is spam if the same email is sent to a very large set of recipients. Or it might look for keywords like *free* or *prize* or apply various AI techniques. If spammers sent email from their actual email addresses, those addresses could be easily

added to a deny list, but, unfortunately, spammers usually do not send from legitimate email addresses (or they forge legitimate addresses that are not theirs).

Or, if a lot of spam is detected from some IP address, that address might get deny-listed. There is a problem with that, though. If a spammer rents space on a public cloud, they might eventually get evicted for misbehavior (such as sending spam), but then the innocent next tenant will have their IP address deny-listed. Or perhaps the spam filter might deny-list an entire block of IP addresses, in which case all tenants of a particular public cloud will be punished for the misbehavior of some by having their IP addresses included in the deny-listed set.

14.7 MALICIOUS LINKS IN EMAIL

Spam email wastes users' time, storage, and network bandwidth and can trick gullible users into purchasing bogus products, sending money, or divulging personal information. But another issue is that an email can contain a malicious link that, if clicked by the user, can install malware on the user's device.

Some advice given to users is *don't click on links in email*, but that is really not practical. Legitimate email messages contain links. A user might attempt to access the content pointed to by the link without clicking the link, *e.g.*, by typing the DNS name of the legitimate entity, starting at their home page, and navigating their website. This would be time consuming and sometimes impossible.

There is at least one company that provides a service to help a company protect its employees from malicious links. Let's call the service [phishingprotection.org](#) (intentionally not a real name). To use [phishingprotection.org](#), a company has their MTA look for links in email. The MTA prepends [phishingprotection.org/](#) to the URL of each link. Then, when the email user clicks on the link, the request goes to [phishingprotection.org](#). Since the URL contains the original link, [phishingprotection.org](#) can retrieve the page specified by the original URL and scan it for viruses, or perhaps merely check the DNS name of the original URL against a deny-list or allow-list of known bad or good sites.

14.8 DATA LOSS PREVENTION (DLP)

The term *data loss prevention* is, in our opinion, a very confusing term. It sounds like the data actually gets lost, and the company should have made backups. What the industry means by DLP is pre-

venting a company's intellectual property or other confidential information from being sent to unauthorized recipients.

A company's MTA could check messages that are destined for addresses outside of the company and scan for anything marked company confidential. Or it could look for keywords such as product names or credit card information. It could delete messages that might leak information or hold them for inspection by a human who can decide whether the messages are consistent with company policy.

Note there are other ways of stealing company confidential information besides emailing it. For example, the information could be loaded onto a USB stick or sent using a file transfer service.

14.9 KNOWING BOB'S EMAIL ADDRESS

Alice wants to send email to user Bob, and for a simple example, let's assume they both work at the same company. The first step in ensuring that only Bob sees the message is to enable Alice to learn Bob's email address. At a large company, say `company.com`, there will likely be many people named Bob Smith. The first one hired might get the email address `bob@company.com`. The next one might be `bob.smith@company.com`. The next one might be `robert.q.smith@company.com`, and so forth. When someone looks up Bob Smith in the company directory, they get several choices, and they have to guess. Invariably email gets sent to the wrong Bob Smith, and the unintended recipient has to read these carefully in order to guess which email address the sender should have used. In some cases, this can place more of a burden on the unfortunate Bob Smith than receiving obvious spam. He can quickly delete a message claiming that Bill Gates has chosen him to receive a million dollars. However, an email saying "the company CEO really wants you to attend this meeting in Minneapolis in January" will take more thought to decide if the message was actually intended for one of the other seven Bob Smiths at the company. The original recipient could forward the message to all seven of the other Bob Smiths, and then they could all have a great time together in Minneapolis in January, even if there were no actual meeting.

14.10 SELF-DESTRUCT, DO-NOT-FORWARD, ...

A sender might want a message to be destroyed shortly after the recipient reads it or prevent the recipient from saving or forwarding the message. This can be implemented by marking the message with the sender's desires and having the destination email system follow the instructions. However,

as with all DRM (digital rights management), enforcement in a software-only system can usually be circumvented by modified versions of the client email application or taking a photo of the screen.

14.11 PREVENTING SPOOFING OF FROM FIELD

It is easy to place any string into the FROM field of an email. Some email clients allow you to set what you want in the FROM field of all outgoing email.

If Alice (the sender) were to digitally sign the email she sends to Bob, and Bob expects all email from Alice to be signed by Alice, he couldn't be fooled by someone else putting Alice's name into the FROM field. However, client-signed email is not widely deployed, perhaps because software to verify signatures is not widely deployed. So there are various proposals for making it harder to spoof the FROM field.

SPF (Sender Policy Framework) [RFC 7208] allows an email domain to declare a limited set of IP addresses (or IP address ranges) authorized to send mail from that domain. (An email domain is the DNS name after the @ in an email address.) The declaration is made by placing data in the DNS, which can be retrieved securely through use of DNSSEC. A receiving MTA that supports SPF will refuse mail claiming to be from a domain if it is not coming from one of the listed IP addresses for that domain. It is in practice difficult, but certainly not impossible, to impersonate a connection from someone else's IP address, so although SPF adds security, it is not as strong as cryptographic authentication. Also, the security depends on the sending MTA authenticating the user transmitting the email and checking that the authenticated user really does own the email address in the FROM field.

SPF causes some problems for mail forwarders. For example, if Alice sent email to a distribution list exploder, it would forward copies to each recipient. The IP address from which the forwarded email would arrive at a destination MTA would be the distribution list's IP address and not Alice's MTA. This sort of deployment challenge discourages deployment of SPF.

DKIM (DomainKeys Identified Mail) [RFC 5585] allows an email domain to digitally sign outbound email messages associated with its domain and to declare (in DNS) that mail coming from that domain will be signed with one of a list of signing keys. Most MTAs will respect such declarations and refuse mail claiming to be from such a domain if it is not digitally signed with one of the listed public keys. Since mail forwarders often modify email messages in various ways, this will cause the DKIM signature to fail. An example is that a forwarder might add a "forwarded by" to the subject line or the body.

14.12 IN-FLIGHT ENCRYPTION

Instead of having Alice encrypt an email with Bob's public key, she might send the email to her MTA using an authenticated and encrypted channel such as TLS. The MTA might authenticate to Alice using a certificate. Alice might authenticate using username and password, or there might be a Kerberos ticket stored on her device. Furthermore, the MTAs might communicate with each other using TLS. And Bob might log into his MTA to retrieve his email, and that connection might use TLS. This is often marketed as "email encryption", but it is not end-to-end. It requires trusting all the MTAs along the path and requires Bob to store the email in some protected way.

If an MTA is capable of communicating using TLS, it has to know which other MTAs can communicate using TLS and assumes the standard Internet PKI with X.509 certificates for learning each other's public keys. Otherwise, anyone could spoof an MTA by saying "I am the MTA for this email domain, but I don't do TLS, so you should just believe me." DANE (DNS-Based Authentication of Named Entities) [RFC 7672] specifies how to announce in DNS (hopefully using DNSSEC) that you (an MTA) accept TLS connections. MTA-STS (MTA Strict Transport Security) [RFC 8461] is similar to DANE but announces at the mail domain level, so the announced policy applies to all MTAs serving that mail domain. There is nothing wrong with using both mechanisms. STARTTLS [RFC 3207] allows an MTA to tell another MTA with which it is in the process of forming a connection that it supports TLS. Note that without DANE or MTA-STS, an active attacker can remove the "I can speak TLS" from the message and trick two TLS-capable MTAs into communicating without cryptographic protection.

Sometimes this hop-by-hop in-flight encryption is referred to as "encrypted email", even though it is not end-to-end encrypted.

14.13 END-TO-END SIGNED AND ENCRYPTED EMAIL

Although, as we said, end-to-end cryptographically protected email (such as envisioned by PGP, PEM, and S/MIME) is not widely used, the concept is interesting. If Alice wants to send a signed message to Bob, she can just sign the message and send it, hoping that if Bob actually wants to validate her signature, he'll already have her certificate or can obtain it if necessary. Alternatively, Alice can be pessimistic and assume she needs to send her certificate to Bob, in which case she can include her certificate in the email message.

A signed email together with Alice's certificate will work whether Alice is sending to a specific individual or sending to a distribution list (provided that none of the fields included in the sig-

nature is modified by a distribution list exploder). The signature will also work if the signed email is forwarded (in its entirety) by Bob.

However, if Alice wants to send an encrypted message to Bob, she needs to know his public key. There are various methods by which she might discover Bob's public key:

- She might have received Bob's public key through some secure out-of-band mechanism and installed it on her workstation.
- She might obtain it through a PKI.
- The email system could allow piggybacking of certificates on email messages. For instance, if Bob sends an email to Alice, he could include his certificates so Alice will know his public key. Or if Alice has not received prior email from Bob, she can send an email to Bob requesting a return message containing Bob's certificates.

How would Alice send an encrypted email to several recipients, say, to Bob and Carol? Each recipient has a different public key. To encrypt such an email, Alice chooses a random secret key S to be used only for encrypting that one message. She encrypts the message with S . Alice need only encrypt the message once. But she encrypts S once for each recipient, with the appropriate key, and includes each encrypted S with the encrypted message. She might send a single message to all three recipients, in which case the header would contain $\{S\}_{Bob}$, $\{S\}_{Carol}$, and $\{S\}_{Ted}$, or each copy might be customized for each recipient (Carol's copy will only contain $\{S\}_{Carol}$, for instance).

How would Alice send an encrypted email to a distribution list or to a group? Note that very few, if any, email systems implement this, but here is a description of how it could be done. Suppose Alice is sending a message to a distribution list that will be remotely exploded, and Bob is only one of the recipients. Assume the distribution list is stored at some node remote from Alice, and Alice does not even know the individuals on the distribution list. So she won't know the public keys for all the members of the distribution list. Instead, she has a key only for the distribution list exploder. Alice does not need to treat the distribution list exploder any differently than any other recipient of a message. It is merely a recipient with a public key. The distribution list exploder will need keys for all the individuals on that distribution list. It will decrypt S (which was encrypted with the distribution list exploder's public key) and encrypt S with each recipient's public key as it forwards the encrypted email to each member of the distribution list. Note that the distribution list exploder *could* see the plaintext of the message, but it need not decrypt the message; it only needs to decrypt and re-encrypt S for each recipient.

Another possible way of sending email (or sharing a file) with a group of recipients is to encrypt the message with a key for the group. The distribution list exploder might not be able to decrypt the message and will just forward the ciphertext to all the recipients. Again, the message would be encrypted with a secret key S just for that message. The encrypted message would be $\{message\}S$, and associated with the encrypted message would be $\{S\}_{group-key}$. Perhaps all members of the group have obtained and stored group-key in advance, and they can immediately decrypt messages for the group. Alternatively, there might be a group server to which they authenticate, and

the member could send $\{S\}_{\text{group-key}}$ to the group server, which would authenticate the member, decrypt $\{S\}_{\text{group-key}}$, and send the member S .

14.14 ENCRYPTION BY A SERVER

There are some products that will encrypt email for a user. There are various ways this could work. It is usually awkward if the receiver has not implemented the proprietary mechanism and receives encrypted email. Such email will usually come with instructions for how to sign up for the service or download software.

One mechanism might be for a server to create public key pairs for all the users it supports. If user Alice wants to send to user Bob, and the service does not have a public key for Bob, it creates one and stores the key pair. Later, when Bob, following the instructions in the received encrypted email, goes to the service and authenticates somehow as owning that email address, the service can decrypt the email for him or give him the opportunity to download software and his private key so that he can decrypt future emails.

Another mechanism is for Alice to tell the service that she will want to send an encrypted email. The service can create a secret key and a key ID and tell Alice to put the key ID in the header and encrypt with that secret key. When Bob receives the email, the service can decrypt the email for him or send Bob the secret key (assuming Bob has downloaded the proprietary software for that service).

Another mechanism is to use identity-based encryption (IBE). This is a form of public key cryptography introduced as a concept by Shamir in [SHAM84]. A practical solution was invented by Boneh and Franklin and published in [BONE01]. Rather than using a certificate that maps a name to a public key, the public key for a name in a domain can be derived from the name.

- All the users in the domain trust a server known as a **Private Key Generator (PKG)**.
- Associated with the domain are **domain-specific parameters** that are known by all the nodes in the domain and will enable them to convert a name into a public key.
- The PKG knows a **domain secret** that allows it to convert a public key into a private key.

If a domain (for instance, a company) uses a product based on IBE, the server will know the domain secret, and all users will be configured with the domain-specific parameters. When Alice wants to send email to Bob, she derives his public key from his name and the domain-specific parameters. When Bob receives the email, if he already has authenticated to the server (somehow) and received his private key, he can decrypt the email. Otherwise, his email client needs to contact the server and receive his private key. Both Alice and Bob need to have special software for the encryption and decryption.

With all of these systems, although the email is encrypted by Alice with a key known to Bob, the server will be able to decrypt all email. Therefore, this sort of server-mediated email encryption is usable for corporate email when a corporation wants to be able to access plaintext of all email that goes to or from an email address in the corporate domain.

14.15 MESSAGE INTEGRITY

When Bob receives a message from Alice, how does he know that Carol did not intercept the message and modify it? For instance, she might have changed the message “Fire Carol immediately” to “Promote Carol immediately”. It makes little sense to provide for authentication of the source without also guaranteeing integrity of the contents of the message. Who would care that the message originated with Alice if the contents might have been modified en route?

For those used to cryptography, the solution is simple. Have Alice digitally sign the content of the message. However, there are complications. Suppose Bob would like to forward to Carol a subset of an email he got from Alice. Once Bob modifies Alice’s email in any way, her signature will no longer verify.

Another complication with having a message digitally signed by the source (Alice) is that it would interfere with services that the MTA might provide that involve modifying the message. For instance, the MTA might look for fields that look like credit card numbers and delete or encrypt them. Or it might modify URLs in links in order to have a service that inspects links in emails (see phishing protection in §14.7).

Yet another complication is what fields Alice’s signature would cover. Email systems can’t encrypt most of the header because that would interfere with delivering the email to the right destination. The secure email standards PEM and PGP only provided integrity protection or privacy on the *contents* of a message. S/MIME allows either behavior—where the header (*e.g.*, SUBJECT, TO, FROM, or TIMESTAMP) is not protected or where a copy of the header is included with the protected message and some other header, possibly a copy of what was in those fields before, would appear outside the cryptographic envelope. Most humans would expect, knowing that they were using email that used encryption and digital signatures, that the entire email message would be protected.

There are interesting security flaws that can be exploited if the signature does not protect the SUBJECT, TO, FROM, or TIMESTAMP field. A naive user might put private information into the subject line, and expose the information to an eavesdropper (for instance, a subject line of “Our company is getting acquired at twice the market price!” with a forwarded email about the acquisition). Or suppose Alice sees a message from Fred, in which Fred suggests some outrageous idea. Alice forwards Fred’s message, integrity protected, with the subject line “Fire this Bozo immediately”, but someone modifies the unprotected subject line to “Great idea! Implement this suggestion immediately”.

If the subject line is not protected, the message will arrive signed by Alice, but the modified subject line completely changes what she intended to say.

The header fields cannot be encrypted because the mail infrastructure needs to see them. Theoretically they could be included in the integrity protection even though they couldn't be privacy-protected.

Note that even if email protects all the header fields, it would be possible to misuse electronic mail. For instance, if Alice sends Bob the signed message "I approve", Bob can't use the message later to prove that Alice okayed his action, since the message isn't explicit about what Alice was approving.

Perhaps if digitally signed email were widely deployed, users would get used to understanding how to use the feature securely.

14.16 NON-REPUDIATION

Repudiation is the act of denying that you sent a message. If the message system provides for **non-repudiation**, it means that if Alice sends a message to Bob, Alice cannot later deny that she sent the message. Bob can prove to a third party that Alice did, indeed, send the message. For instance, if Alice sends a message to the bank that she wishes to transfer a million dollars to Bob's account, the bank should not honor the transaction unless it is sent in such a way that not only can the bank know the message was from Alice, but they can prove it to a court if necessary.

The traditional method of having Alice sign the message with her private key provides non-repudiation. Not only does the recipient know that the message came from Alice, but they can prove to anyone else that Alice signed the message.

Note that non-repudiation sounds like it has a legal meaning. It does not. Even if something is signed by Alice's private key, she can claim that malware on her device signed on her behalf without her permission or that her private key was stolen.

14.17 PLAUSIBLE DENIABILITY

It is not always desirable to have non-repudiation. For instance, Alice might be the head of some large organization and want to give the go-ahead to her underlings for some scheme like selling arms to Iran to raise money to fund the Contras. The underlings must be absolutely certain the orders came from Alice, so it is necessary to have source authentication of the message. But Alice

wants plausible deniability (what a great phrase!), so that if any of her underlings are caught or killed, she can disavow any knowledge of their actions. How can Alice send a message to Bob in such a way that Bob knows it came from Alice, but Bob can't prove to anyone else that it came from Alice?

First, we'll review our notation. We use curly braces {} for encrypting something with a public key, with a subscript specifying the name of the individual whose public key we are using. We use square brackets [] for signing something with a private key, with a subscript specifying the name of the individual whose private key is being used.

1. Alice picks a secret key S , which she will use just for m .
2. She encrypts S with Bob's public key, getting $\{S\}_{Bob}$.
3. She signs $\{S\}_{Bob}$ with her private key, getting $[\{S\}_{Bob}]_{Alice}$.
4. She uses S to compute a MAC for m .
5. She sends the MAC, $[\{S\}_{Bob}]_{Alice}$, and m to Bob.

Bob will know that the message came from Alice, because Alice signed the encrypted S . But Bob can't prove to anyone else that Alice sent him m . He can prove that at some point Alice sent some message using key S , but it might not have been m . Once Bob has the quantity $[\{S\}_{Bob}]_{Alice}$, he can construct any message he likes and construct a MAC using S .

14.18 MESSAGE FLOW CONFIDENTIALITY

This feature allows Alice to send Bob a message in such a way that no eavesdropper can find out that Alice sent Bob a message. This would be useful when the mere fact that Alice sent Bob a message is useful information, even if the contents were encrypted so that only Bob could read the contents. (For instance, Bob might be a headhunter. Or Bob might be a reporter, and Alice might be a member of a secret congressional committee who is leaking information to the press.)

If Alice knows that intruder Carol is monitoring her outgoing mail to look to see if she is sending messages to Bob, Alice can utilize a friend, Fred, as an intermediary. Alice can send an encrypted message to Fred with the message she wants to send to Bob embedded in the contents of the message to Fred. So the message Fred would read (after he decrypts it) is "This is Alice. Please forward the following message to Bob. Thanks a lot." followed by Alice's message to Bob. If Alice were sufficiently paranoid, she might prefer a service that is constantly sending encrypted dummy messages to random recipients. If Fred forwards Alice's message after a random delay, then even if Carol knows Fred provides this service, Carol cannot know to which recipient Alice was asking Fred to forward a message.

To be even more paranoid, a path through several intermediaries could be used. Suppose Bob and all the intermediaries have public keys. When we say *encrypt a message with a public key*, we mean to encrypt the message with a randomly chosen key K and include K encrypted with the recipient's public key along with the encrypted message. To send a message to Bob without divulging that she is communicating with Bob, Alice chooses a path of intermediaries, say, R1, R5, R2 (so the message will travel from Alice to R1, then from R1 to R5, from R5 to R2, and then from R2 to Bob). R1 will know that the message came from Alice, but not who the destination is. R2 will know the destination is Bob but won't know who the source is. Alice does the following:

- She encrypts the message with Bob's public key.
- She takes the result and encrypts that (plus instructions for R2 to forward to Bob) with R2's public key.
- She takes the result and encrypts that (plus instructions for R5 to forward to R2) with R5's public key.
- She takes the result and encrypts that (plus instructions for R1 to forward to R5) with R1's public key.
- She takes the multiply encrypted result and sends it to R1.

R1 decrypts what it receives, and the result is an encrypted message and instructions to forward it to R5. R5 decrypts the result and gets an encrypted message and instructions to forward it to R2. R2 decrypts the result and gets an encrypted message and instructions to forward to Bob. Bob now receives a message that he can decrypt, and Alice can, if she chooses, divulge her identity to Bob.

This design, in which each hop uses public key encryption and includes in each encrypted message the IP address of the next hop, makes the message longer for each hop, and private key decryption for each intermediary for each message is expensive.

It is possible to instead use secret key encryption by having Alice set up a path through a chosen series of intermediaries between Alice and Bob. There are several advantages of using secret key encryption for data forwarding:

- Public keys will be used for connection setup, but for data forwarding, only secret keys will be needed. So data can be fixed-size, and the computation burden on forwarding data is reduced. This is especially advantageous if a lot of traffic will be sent between Alice and Bob (for instance, a web browsing session).
- With this mechanism, Bob can return traffic to Alice even if he does not know who he is communicating with.

Again, assume the path Alice chooses uses intermediaries R1, R5, and R2. Alice can establish a connection to R1 and agree on a shared secret key, say, K_{A-R1} , for the Alice–R1 connection. She then can use that connection to secretly establish a connection to R5 and agree upon a shared secret key K_{A-R5} with R5. Then she can leverage her secret connection with R5 to create a secret

connection with R2 and agree upon a shared secret key K_{A-R2} . Since the intermediaries keep state about ongoing connections, there is no need to include forwarding instructions. Each intermediary keeps a connection table that tells them, for each connection, what key to use to decrypt with and which connection to forward the resulting message on.

To send a message to Bob, Alice encrypts the message with a secret key she has established for this connection for Bob. She takes the result and encrypts it with K_{A-R2} , encrypts that with K_{A-R5} , and encrypts that with K_{A-R1} . When she sends this quadruply encrypted message to R1, R1 recognizes the connection (*e.g.*, from the TCP port), does a table lookup for the connection, and finds that it should decrypt with K_{A-R1} and forward the packet on a connection to R5. The next intermediary (R5) looks up information that it stored about this connection during connection setup and finds that it should decrypt with K_{A-R5} and forward on a connection to R2. And so on.

Bob can send a message on the return path without setting up a new path to Alice. He encrypts the message with a key he shares with Alice, and forwards the result on the connection to R2. R2 looks up the information about the connection, which tells R2 to encrypt the message with K_{A-R2} and forward to R5. And so on.

With this approach, not only do we avoid expensive private key decryption operations, but the message does not increase in size at each hop.

Using the multiple intermediary technique, even if someone bribed one of the intermediaries to remember the directions from which it received and sent messages, Alice could not be known to be communicating with Bob. To discover that Alice had sent a message to Bob would require the cooperation of all the intermediaries.

There are several deployed systems using approaches such as this. An email-specific system was conceived by Chaum in 1981 [CHAU81]. There are two deployed systems inspired by Chaum's work—Mixmaster and Mixminion. Another system, aimed mostly at anonymous web browsing, is **onion routing (TOR)**. Onion routing was published by Goldschlag, Reed, and Syverson in [GOLD99], and a system is deployed and maintained by The Tor Project, Inc. (tor-project.org).

14.19 ANONYMITY

There are times when Alice might want to send Bob a message but not have Bob be able to tell who sent the message. One might think that would be easy. Alice could merely not sign the message. However, most mail systems automatically include the sender's name in the mail message. One could imagine modifying one's mailer to leave out that information, but a clever recipient can get clues. For instance, the network-layer address of the node from which it was sent might be sent with the message. Often the node from which the recipient receives the message is not the actual source

but instead is an intermediate node that stores and forwards mail. However, most mail transports include a record of the path the message took, and this information might be available to the recipient.

If Alice really wants to ensure anonymity, she can use the same technique as for message flow confidentiality. She can give the message to a third party, Fred, and have Fred send the unsigned message on to Bob.

If Fred does not have enough clients, he can't really provide anonymity. For instance, if Alice is the only one who has sent a message to Fred, then Bob can guess the message came from her, even if Fred is sending out lots of messages to random people. Fred can have lots of clients by providing other, innocuous services, such as weather reports, bookmaking, or pornographic screen savers (where you are required to transmit a fixed-length, encrypted message in order to get a response), or by having the user community cooperate by sending dummy messages to Fred at random intervals to provide cover for someone who might really require the service. We specified fixed-length messages to Fred so that people could not correlate the lengths of messages into and out of Fred. Someone wanting to send a longer message would have to break it into multiple messages. Short messages would have to be padded to the fixed length. This extra traffic is known as **cover traffic**.

Another concept is **pseudonymity**. That means that although Bob will not know who sent a message, he can know that multiple messages all came from the same entity. This is done by creating a pseudonym associated with a public key and always signing messages associated with that pseudonym with that same public key.

If you send to Bob with a pseudonym, Bob won't be able to reply to that pseudonym through normal DNS lookups, so if replies are required, the remailer intermediaries all have to remember the previous hop from which they received email for a pseudonym. Only if someone breaks into all the remailers will they be able to trace who owns that pseudonym. Anonymous email would presumably not allow Bob to reply.

14.20 HOMEWORK

1. Outline a scheme for sending a message to a distribution list, where distribution lists can be nested. Attempt to avoid duplicate copies to recipients who are members of multiple lists. Discuss how it could be done in both the local exploder and remote exploder methods of distribution list expansion.
2. Suppose Alice sends an encrypted, signed message to Bob via the mechanism suggested in §14.17 *Plausible Deniability*. Why can't Bob prove to third party Fred that Alice sent the

message? Why are both cryptographic operations on S necessary? (Alice both encrypts it with Bob's public key and signs it with her private key.)

3. Using the authentication without non-repudiation described in §14.17 *Plausible Deniability*, Bob can forge Alice's signature on a message to himself. Why can't he forge Alice's signature on a message to someone else using the same technique?
4. Suppose you changed the protocol in §14.17 *Plausible Deniability* so that Alice first signs S and then encrypts with Bob's public key. So instead of sending $\{\{S\}_{Bob}\}_{Alice}$ to Bob, she sends $\{\{S\}_{Alice}\}_{Bob}$. Does this work? (Can Bob be sure that the message came from Alice but not be able to prove it to a third party?)
5. Which security features (privacy, integrity protection, repudiability, non-repudiability, source authentication, anonymity) would be desired, and which ones would definitely not be used, in the following cases:
 - submitting an expense report
 - inviting a friend to lunch
 - sending a mission description to the Mission Impossible team
 - sending a purchase order
 - sending a tip to the IRS to audit a neighbor you don't like but are afraid of
 - sending a blackmail letter
6. Suppose Alice wants to use remailers (§14.18) to hide that she is communicating with Bob. The path of remailers she chooses to get to Bob is R1, R5, R11, R2. Suppose R1, R11, and R2 all collude with each other to try to discover which parties are communicating, but R5 is honest. Can the colluding set discover that Alice is communicating with Bob? (Assume there are lots of messages passing through the system, they are all the same size, and there are random delays introduced at each hop).

15

ELECTRONIC MONEY

Money has a long and storied history as a medium of exchange and a mechanism for measuring and storing wealth. Nations control the minting of coins, the printing of paper currency, and the quantity of money in circulation, trying hard through physical means to avoid counterfeit bills and coins. Nowadays, almost all but the smallest payments are made using credit cards, checks, and electronic funds transfers. Large cash transactions (suitcases full of money) are usually used only for illicit or illegal transactions.

Forms of electronic money have been common for many years. The Internet is used to manage bank accounts, make purchases with credit cards, do wire transfers, and pay bills.

New protocols for dealing with money, such as the ones described in this chapter, typically aim at achieving things that are not easily done with current banking systems, such as cheaply and quickly transferring money across international borders, hiding assets, and anonymously receiving money for services (such as restoring a victim's system after attacking it with ransomware). We will discuss two very different electronic money protocols:

- **eCash**, an elegant design by David Chaum [CHAU82], was designed to allow spending money electronically with the anonymity of cash. It is a centralized design, in that a particular bank allows users to purchase anonymous coins. Many banks could independently create their own anonymous currency using this scheme. eCash was launched as a company (Digi-Cash) and was implemented by a handful of banks between 1995 and 1998. However, it was not commercially successful and is no longer deployed.
- **Bitcoin** is currently deployed, along with hundreds of similar cryptocurrencies. Bitcoin does not involve any central bank or country. Instead, money is created and spent using a network of thousands of nodes that prevent double-spending. The nodes can be anonymous. New nodes can start participating, existing nodes can depart, and the total number of nodes can only be estimated. Users can also be reasonably anonymous but not quite as anonymous as with eCash.

We will concentrate on the interesting security concepts in the designs, rather than on the specific details of these as deployed.

15.1 ECASH

The goal of eCash is to electronically create a form of money that, like physical cash, can be spent anonymously. Let's call that an **eCash coin**. A financial institution (which we'll call "TheBank") converts traditional money into eCash coins that can be paid by a user ("Alice") to a payee ("Bob").

Unlike traditional cash, which uses esoteric materials and printing techniques to make it hard to counterfeit, an electronic message (such as an eCash coin) can be easily duplicated. Therefore, eCash has mechanisms to assure that an eCash coin only gets spent once.

The eCash scheme uses blind signatures, invented by Chaum for this purpose (see §16.2 *Blind Signature*).

To create an eCash coin, Alice chooses a unique identifier (UID)—a large random number that will, with high probability, distinguish that coin from all other eCash coins. She constructs a message that contains that UID along with other formatting information. The extra formatting prevents a random number from looking like a valid signature. Then Alice pays TheBank to blindly sign what she has created. Alice then unblinds the result, and the signed message is an eCash coin. Although TheBank will know how many eCash coins Alice has obtained, it will not know the UIDs of the coins that belong to Alice.

Alice pays Bob by sending him an eCash coin. Although Bob can immediately verify that the formatting of the coin is legitimate, and that TheBank's signature is authentic, Bob must communicate with TheBank to be assured that the coin has not already been spent. TheBank must remember all the UIDs it has seen in spent coins in order to detect double-spending. Once Bob forwards the coin to TheBank, TheBank verifies that nobody has already deposited a coin with that UID, and then adds the value of the coin to Bob's account (or allows Bob to create a new coin).

Suppose Alice uses the same eCash coin to pay both Bob and Carol. Whichever of Bob or Carol turns it into TheBank first will be paid, and the other will be told the coin is not valid because it will have the same UID as a coin that has already been spent.

Note that the cryptographic aspects of this scheme do not require Bob to know Alice's identity (although attributes such as the IP address or the ship-to address might allow Bob to know something about Alice). Bob should probably not deliver the services to Alice until he has verified with TheBank that the coin has not already been spent.

To actually have eCash users be anonymous requires having many users purchasing coins. TheBank knows which users have purchased coins, so if there are only a few that have purchased coins, the identity of the spender of a coin would be limited to a small set.

Note also that Bob need not have an account with TheBank. When Bob verifies with TheBank that he holds a valid coin, TheBank could pay Bob with physical currency or allow Bob to create a new coin that he could spend later.

If all eCash coins were worth the same amount, say, a penny, then it would require a thousand coins for Alice to pay Bob ten dollars, which would be inefficient. Therefore, it is desirable that

eCash coins be issued in multiple denominations. To accomplish this, TheBank needs to use a different public key for each denomination (see Homework Problem 3). For instance, if Alice pays TheBank ten dollars, then TheBank will blindly sign her coin with the public key associated with ten-dollar coins.

15.2 OFFLINE ECASH

The offline eCash scheme, published by Chaum, Fiat, and Naor in [CHAU88], has properties somewhat different from the online eCash scheme and is more complicated. This variant of eCash is intended to allow Bob to somewhat safely accept an anonymous payment of eCash, even if he temporarily cannot communicate with TheBank to check for double-spending. If Alice cheats by paying Bob a coin she had already paid to Carol, TheBank will tell Bob that the coin has already been spent and will reveal Alice's identity. Bob can then presumably sue her. If Alice has not cheated, she remains anonymous.

Although the added practical benefit over the online scheme is small at best (see §15.2.1), some of the cryptographic tricks to achieve this variant are interesting enough that we will give an overview of it.

- To create a single coin, Alice creates many (for concreteness, let's say 256) **coinlets** with a particular format, each of which embeds her identity. However, Alice's identity will not be readable from any of the coinlets unless Alice has cheated and double-spent a coin. Alice blinds the 256 coinlets and sends them all to TheBank.
- A coin consists of 128 coinlets, and the collection is blindly signed by TheBank. Note that Alice has created more coinlets than needed for a coin (in our example, she created 256, but the coin will only consist of 128). Before blindly signing the set of 128 coinlets in a coin, TheBank will ensure that (with high probability) Alice has generated the coinlets honestly. This is done by having TheBank choose a subset, say, 128 out of the 256 blinded coinlets that Alice presented to TheBank and having Alice unblind and reveal those coinlets. If the 128 coinlets that Alice is asked to reveal are, indeed, in the correct format, TheBank will trust that the coinlets it did not ask Alice to reveal are also honestly generated, and then TheBank will sign the remaining hidden coinlets. This technique of having someone other than Alice choose a subset of the items and forcing Alice to prove she is following the rules on all those chosen items is known as **cut and choose**.
- Each coinlet has a number associated with it, computed according to information that Alice must remember about that coinlet until she has spent the coin. TheBank's signature on a

collection of coinlets consists of a signature on the product of the 128 numbers associated with each of the coinlets in the collection.

- The number for each coinlet can be computed using either of two different subsets of the information Alice is required to remember about that coinlet. We will refer to the two subsets as **left-info** and **right-info**. During the cut and choose step, Alice reveals both left-info and right-info about the chosen coinlets to TheBank. Although the number associated with the coinlet can be computed using either left-info or right-info, Alice's identity is only revealed if someone knows both left-info and right-info for a coinlet.
- To pay Bob, Alice sends TheBank's signature on the product of the 128 coinlets to Bob. However, in order to verify TheBank's signature, Bob needs more information (left-info or right-info for each coinlet) that he must obtain from Alice in order to calculate the number for each of the 128 coinlets. He can then multiply the 128 coinlet numbers together and verify TheBank's signature on that product.
- Bob chooses a random 128-bit challenge and sends it to Alice. Each of the bits in the challenge corresponds to one of the 128 coinlets. If the bit in the challenge corresponding to a coinlet is a 0, then Alice must disclose left-info for that coinlet to Bob. If the bit is 1, Alice must disclose right-info for that coinlet to Bob. Note that Bob will receive exactly one of left-info or right-info for each of the 128 coinlets. Knowledge of either right-info or left-info will allow Bob to calculate the number for a coinlet. However, knowledge of *both* left-info and right-info for any coinlet will reveal Alice's identity.
- In order to be paid, Bob sends the right-info or left-info he got from Alice about each of the coinlets to TheBank. TheBank calculates the number for each coinlet based on this information and remembers the number for each coinlet, along with the right-info or left-info that someone turning in a coin has told TheBank. TheBank then checks its database to see if any of the coinlets have already been spent. If TheBank has no record of any of those coinlets having been spent, it tells Bob the coin is valid.
- If Alice had already paid the coin to someone else, say Carol, the combination of what TheBank got from Bob and what TheBank got from Carol will allow TheBank to calculate Alice's identity. The reason is that with high probability, when Alice paid Carol, Carol's 128-bit challenge will differ from Bob's challenge. For any bit where Carol's and Bob's challenge differs, one of them will have collected left-info for a coinlet, and the other will have collected right-info for the same coinlet. When both have sent their information about the coin to TheBank, TheBank will have both left-info and right-info for at least one coinlet, which will reveal Alice's identity.

The actual math for accomplishing this is arcane and not of practical importance. However, if you are curious, you can read [CHAU88] or do Homework Problem 4, which specifies all the details.

15.2.1 Practical Attacks

The offline scheme has some security issues. If someone can create a bank account in a fictitious name (and, in practice, people succeed in doing this at real banks), there is no limit to the amount of multiple-spending they can do, since tracing the fraudulent activity to a bank account will not help catch the actual culprit. As a defense, TheBank could publish a list of coinlet numbers that have been found to be from such fraudulent activity, assuming the list is not prohibitively large. But once a criminal starts multiple-spending, they can steal a huge amount until TheBank alerts all eCash-accepting entities about these bad coinlet numbers. The online eCash model is far more efficient than requiring TheBank to promptly alert all entities that might accept offline eCash.

Another attack is possible if someone were to break into Alice's computer and steal the information (left-info and right-info) for all the coinlets in any of her coins (spent or unspent). They could then spend these coins as many times as they want, and it would only be Alice who would look guilty. Presumably once she reports the theft, she wouldn't be prosecuted for the coins being spent. However, to protect merchants from being cheated with multiply spent stolen coins, merchants must immediately check with TheBank to make sure the coins are valid before shipping the merchandise, which is the same use model as for the online scheme.

With the online scheme, if Alice's information were stolen from her computer, all her unspent coins could be spent by the attacker, once, but no further damage could be done.

Another downside of the offline scheme (as compared with the online scheme) is that Bob cannot be anonymous to TheBank and simply exchange a coin he receives for a fresh coin he could spend elsewhere. That's because he must prove his identity to TheBank to ensure the coinlets he creates correctly record his identity.

Yet another downside is that if TheBank were malicious, it could easily frame Alice.

The original paper suggests fixes to some of these issues.

So, given how complex the offline scheme is, and given the questionable added benefit over the online scheme, we described it only because of its interesting design. The interesting ideas are:

- cut and choose
- having Alice's identity embedded in the coinlet in a way that is revealed only if both left-info and right-info are known for that coinlet
- having a coin consist of many coinlets, and having a recipient send a challenge, whose bits specify which of left-info or right-info Alice should reveal for each coinlet

15.3 BITCOIN

The Bitcoin concept was introduced in 2008 in a paper [NAKA09] published under the pseudonym Satoshi Nakamoto. The goal is to have a version of electronic cash in which parties can pay each other without having the transaction mediated through a financial institution. Double-spending is prevented by having a ledger of all transactions recorded in a publicly readable data structure known as a **blockchain**, maintained by an anonymous group of nodes called **miners**.

In contrast with eCash, the value of a bitcoin is not tied to the value of any other currency. Moreover, unlike national currencies, there is not (and by design there cannot be) any institution like the Federal Reserve or European Central Bank tasked with keeping the value of a bitcoin stable, so its price can and does fluctuate wildly. While government-issued currencies can also fluctuate, they rarely fluctuate as much as Bitcoin (or other cryptocurrencies), and large fluctuations of national currencies generally only occur in cases of severe political instability. Bitcoin currency might be purchased in the belief that someone else will accept it later for similar (or higher) value. Or a transaction might be done with bitcoins rather than traditional currency because it is more convenient (*e.g.*, no need to incur the overhead of currency exchange). Or a criminal might want to be paid in bitcoins because it is fast, hard to reverse, and mostly untraceable.

In the following description, we will concentrate on the intuition behind the mechanisms rather than the implementation details. Currently, Bitcoin is an open source project, and the community can change the details.

Some Bitcoin terminology:

- **address**—the hash of a public key of the sender or receiver of an amount of bitcoin
- **transaction**—a signed message from one address, transferring bitcoins received previously, to another address
- **ledger**—a record of all transactions that have ever occurred
- **block**—a data structure consisting of a list of valid transactions, the hash of the previous block in the chain, and other information
- **blockchain**—the ledger, formatted as a sequence of blocks
- **miner**—a node that gathers transactions and attempts to create the next block in the chain

We will now explain various interesting aspects of the design: why the ledger is difficult to forge; how the algorithm can adjust the difficulty of adding a new block so that on average, the community of miners adds a block about every ten minutes; why Bitcoin consumes so much power; how new bitcoins are created; and how the design ensures there is a maximum number of bitcoins that will ever be created.

15.3.1 Transactions

A transaction specifies that the owner of a public key that received some amount b of bitcoin in a previous transaction is transferring that amount to one or more payee addresses. Bitcoin received in a transaction must be spent all at once. A transaction is identified by the hash of the information in the transaction.

If Alice needs to pay Bob some amount, say b bitcoins, she needs to find a transaction x in which a public key she owns was paid at least b bitcoins (that she has not yet spent). If she received more than b bitcoins in transaction x , then she can list two payees for the new transaction—an address that Bob tells her to pay to, and a public key that she owns (to receive the difference between what she had received in transaction x and what she is paying Bob—like receiving change at a store). That is an example of a transaction that has multiple outputs. It is also possible for a transaction to have multiple inputs, if the amount being paid needs to combine smaller amounts received in previous transactions, or if several individuals pool resources to purchase something. So, a transaction includes:

- a set of inputs, each including a hash value of a previous transaction together with a sequence number indicating which output of that previous transaction is being spent in this transaction
- a set of outputs, each including an amount of bitcoin and an address of the recipient. The bitcoin amounts in the outputs will add up to no more than the value of the inputs, and any difference is a transaction fee
- a public key and a signature for each distinct address among the inputs

All this complexity is hidden behind the user interface of software a user would download. The software is called a **wallet**.

15.3.2 Bitcoin Addresses

Unlike eCash, Bitcoin doesn't provide complete anonymity. Identities in Bitcoin are public keys. Users are encouraged to create a new public key for each transaction to make it difficult to link transactions to the same individual. If a user reuses her public key for multiple transactions, someone that knew her identity (for instance, because she purchased something that needed to be shipped to her home address) might be able to figure out what else she was purchasing. Also, if someone stole the private key associated with one of her addresses, they would be able to spend all the unspent coins under that address.

Using a different public key for each transaction still doesn't guarantee anonymity. The ledger will be world-readable, and a lot of information can be mined by following the trail of coins (*e.g.*, "X1 pays X2; X2 pays X3" or "X1, X2, and X3 all pay X4 the same amount on the first of each month").

The address to which a payer pays is the hash of a public key, rather than the actual public key of the recipient. This makes the address shorter. It also may make Bitcoin more secure if the public key algorithm is weak, assuming that a user, say Bob, changes his public key on every transaction so only needs to sign once with each public key. If Carol tried to steal bitcoin paid to address $\text{hash}(B)$ by computing the private key associated with B , she'd have to first find a public key that hashed to $\text{hash}(B)$ before she could start trying to break the public key scheme by finding the private key for B . However, Bob must reveal his actual public key when he spends bitcoin so that his signature on the spending transaction can be verified. There might be a very long time (*e.g.*, months) between the time Bob received the coins and when he spends them. If the transaction in which Bob received the coins specified Bob's actual public key, the attacker would have a very long time to try to break the key and spend the money. But if Bob's actual public key is only divulged when Bob spends the coins, an attacker would have to break the key between the time the attacker sees the transaction and when the transaction appears in the ledger, which would likely be a few minutes, rather than months.

15.3.3 Blockchain

The ledger is in the format of a chain of blocks (the **blockchain**—Figure 15-1), where each block contains the hash of the previous block (in addition to other information, such as new transactions).

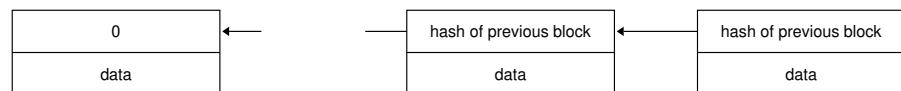


Figure 15-1. Blockchain

This format makes it impossible to replace a block in the middle, say block n , without replacing all the blocks following that block. If block n were replaced with a different block, the hash of block n would be different, which would cause block $n+1$ to have to be changed (since block $n+1$ contains the hash of block n). Likewise for all subsequent blocks.

15.3.4 The Ledger

The ledger records all transactions that have ever occurred. Having all transactions visible allows nodes to prevent double-spending. A transaction is identified by its hash. Suppose A pays B some number of bitcoins in a transaction T_1 , and then, perhaps years later, B pays C what B received in transaction T_1 . To verify that the B to C transaction is valid, the ledger must contain transaction T_1 (which must show that B indeed received that amount of bitcoin), and also there must be no other transaction later in the ledger in which B paid T_1 to anyone else.

To save time when validating a transaction, most nodes keep a separate, internal database (that is not shared with other nodes) that keeps track of all the unspent transactions in the ledger. This database is usually called the list of UTXOs (Unspent Transaction Outputs). If a valid transaction T_i has input T_j , then T_j is removed from the unspent transaction database (because it is being spent), and T_i is added to the UTXO database. If a transaction T_x has multiple outputs, say three outputs, then three entries will be placed in the UTXO database: $\langle T_x, 1 \rangle$, $\langle T_x, 2 \rangle$, and $\langle T_x, 3 \rangle$. If transaction T_x has multiple inputs, all those inputs will be removed from the UTXO database. If transaction T_1 is in the UTXO list, and T_1 specifies that B received the necessary amount of bitcoin, then the node does not need to look through the ledger to see if B had paid T_1 already, because if B had spent T_1 , that transaction would have been removed from the list of UTXOs.

Each block in the blockchain contains some number of transactions plus other information. The list of transactions includes this information:

input	payees/amounts	signer	transaction hash
transaction x_i (where P was paid)	A/133	P	x_1
transaction x_1	B/78, C/51	A	x_2
transaction x_2 , output #2	Q/50.88	C	x_3
transaction x_2 , output #1	Z/77.95	B	x_4
transaction x_4	M/49, K/28.6	Z	x_5
transaction x_5 , output #2	D/15, K/13	K	x_6
transaction x_6 , output #2	N/12.8	K	x_7

The first row of the table says that P had paid A 133 coins from a previous transaction with hash x_i . If P had received less than 133 coins in transaction x_i , this transaction would not be valid. If P received more than 133 coins in transaction x_i , then the remainder is a transaction fee paid to the miner that includes this transaction in a block. In the second row, A signs over the proceeds of the transaction with hash x_1 to B and C. Of the 133 coins A receives in transaction x_1 , A pays 78 to B, 51 to C, and the remainder goes to the miner. In the third row, the input says “transaction x_2 , output #2”, meaning that the second payee in transaction x_2 (namely C, who is seen receiving 51 coins in the second row) pays Q 50.88 coins.

To prevent double-spending, it is essential that the community agrees on an order for transactions. In other words, if A attempts to double-spend what it received in transaction x_1 , there will be two transactions—one where A signs the output of x_1 to B, and one where A signs the output of x_1 to C. Once one of those transactions is recorded in the ledger, the other transaction is considered invalid, and a miner will not record the other transaction in the ledger because any block that contained an invalid transaction would be ignored by the other miners.

It is possible for the transaction that occurred later to be the one that is recorded in the ledger, because which transactions to include in a block is completely at the discretion of the miner that creates the next block in the chain. As long as all the transactions that are included are valid, the

block will be accepted by the other miners. For instance, since a block has a fixed maximum length, if not all pending transactions will fit into a block, a miner might choose to record transactions with the highest transaction fee.

15.3.5 Mining

A miner that successfully creates the next block in the chain is rewarded with bitcoins. The rule is that a block includes a set of new valid transactions, the address of the miner that found this block, the hash of the previous block, and a random number known as a nonce. The hash of the new block must be smaller than some value. For example, if the hash must have 77 leading zeroes (the approximate required size of the hash in 2022), a miner has only a 1 in 2^{77} chance that a candidate block (including the random number the miner chose) will have a hash that is sufficiently small. If the candidate block has a hash bigger than the maximum value (which it almost certainly will), then the miner chooses a new random number and tries again.

Coming up with a block with a sufficiently small hash is like winning the lottery. The more hashes the miner can compute, the more likely it is to find a block that will be accepted by the other miners before any other miner finds one. In a money lottery, the more tickets you buy, the more likely it is you will win. In Bitcoin, the more computation you can devote to calculating hashes, the more likely you will find the next block in the chain.

If a miner finds a block with such a hash, that miner distributes the new block to the peer-to-peer network of miners, and the rest of the miners now start building on that block. Since the new block takes some time to propagate, it is possible for multiple miners to find a new block with a sufficiently small hash, and those blocks might contain different transactions. If a miner sees two valid blockchains, if they are of equal size, the miner keeps trying to build on the one it saw first. However, if one is longer, the miner ignores the shorter blockchain and attempts to build on the longer blockchain. The miner that finds a block is not rewarded unless the mining community agrees on the chain that includes that miner's block, so it would not make sense for a miner to build on a shorter blockchain. Note that having multiple valid chains is known as a **fork** (see §15.3.6 *Blockchain Forks*).

Bitcoins are created and awarded to the miner that is lucky enough to find a block that is included in the blockchain. The lucky miner is rewarded not only with the newly minted bitcoins, but also with all the transaction fees in transactions included in that block. Some interesting details:

- In 2009, the miner of a block was awarded 50 bitcoins, in addition to any transaction fees. The award for a block decreases by half about every four years, because the designers wanted there to be a finite amount of bitcoin that can ever be generated. In 2022, the award for a block was 6.25 bitcoins. Eventually, after the amount is halved many more times, the reward

of newly minted bitcoins per block will be negligible, and miners will only be rewarded based on transaction fees of the transactions in the block.

- The desire is that a hashed block is found approximately every ten minutes. Every 2016 blocks (nominally two weeks, since $2016=6\times24\times7\times2$), the time it took to create those blocks is computed. If it took too little time (*i.e.*, it took less than ten minutes, on average, for each block), the difficulty of finding a hash is increased by making the maximum hash size smaller. If too much time has elapsed, the difficulty of finding a hash is decreased. For instance, to increase the difficulty by a factor of two, the maximum hash value is divided by two.

It might seem problematic to halve the reward for finding a hash. Wouldn't there at some point be no incentive for a miner to spend all that electricity to compute hashes? The hope is that people will add enough transaction fees to their transactions to provide continuing compensation.

15.3.6 Blockchain Forks

It is possible to have two valid blockchains that start with the same first n blocks, but diverge afterwards (see Figure 15-2).

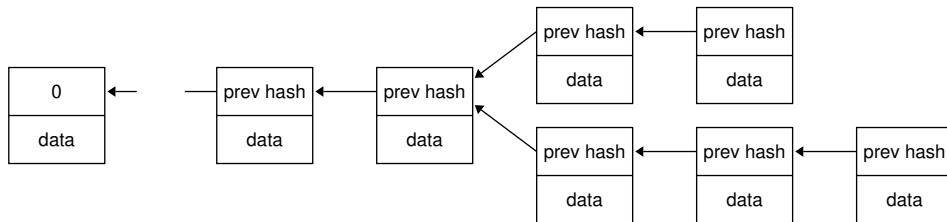


Figure 15-2. Forked Blockchain

This can happen due to several reasons. For instance, two miners might find a next block at approximately the same time, and it takes some time for the different chains to propagate through the peer-to-peer network. Also, if the network of miners were to partition into subgroups, where there is no communication between the subgroups, each subgroup will continue adding blocks to their blockchain. When the partition is repaired, whichever subgroup has the longest chain at that point will have their chain accepted, and the shorter blockchains will be forgotten.

The consequence of forking is that transactions in the shorter chain are no longer recorded, and coins paid in those transactions can be spent again. It also means that miners that thought they'd won the lottery by finding blocks in the shorter chain no longer own the bitcoins they thought they'd earned.

15.3.7 Why Is Bitcoin So Energy-Intensive?

This design, computing all those hashes (known as **proof of work**), is very expensive. Estimates (in 2022) are that the energy consumed by mining bitcoins is about twice the output of the largest U.S. nuclear power plant. The energy use is despite the fact that Bitcoin is a very low-volume application (in number of transactions per second) compared to credit cards.

If the maximum hash size has 77 leading zeroes, it means that the total mining community needs to compute, on average, 2^{77} hashes every ten minutes. That is a huge number, but that very expense is what Bitcoin relies on for security. If an attacker could amass more compute power than the Bitcoin mining community, the attacker could, for example, erase a transaction in block n by creating an alternate block n , and then compute enough subsequent blocks in a chain to be longer than the chain the rest of the community had computed. Once the attacker introduces the longer chain, the shorter chain is replaced. It could very well be that the Bitcoin mining community is so large that no nation-state or well-funded criminal enterprise could amass more compute power. However, there are many cryptocurrencies being created, and most of them will not have a very large mining base.

15.3.8 Integrity Checks: Proof of Work vs. Digital Signatures

In traditional cryptography, someone who is authorized to create an integrity check on data (a digital signature) knows a secret (a private key). With the magic of cryptography, there is an enormous gap between the computation needed to create a signature (by an authorized party that knows the private key) and the computation needed to forge a signature (create a valid-looking signature without knowing the private key). Furthermore, increasing the size of the key increases that gap.

In contrast, as a consequence of the Bitcoin design goal of having no known authorized parties, the integrity check (blocks that hash to a small number) requires an *equal* amount of compute power to create as it would take for someone to maliciously create an alternate chain that looked valid.

What do we mean by “large gap” between signing and forging for traditional cryptography, and what do we mean by increasing the gap by increasing the key size? For example, with a 1024-bit RSA key, forging a signature is about 2^{63} times harder than creating a signature. With a 2048-bit RSA key, forging a signature is about 2^{94} times harder. It is hard to appreciate these numbers, so a tangible way of saying this is that with a 1024-bit RSA key, signing takes about one millisecond on a typical CPU, whereas forging the signature would require the compute power of today’s entire Bitcoin miner community for about an hour. This is certainly an enormous gap, but by doubling the key size to 2048-bit RSA, signing increases from one millisecond to six milliseconds, but forging would require the compute power of today’s Bitcoin miner community for the next million years!

In contrast, as a consequence of the Bitcoin design goal of having no known authorized parties, the integrity check (constructing blocks that hash to a small number) requires an *equal* amount of compute power to create as to forge!

15.3.9 Concerns

The Bitcoin infrastructure is incredibly expensive in terms of computation, and moderately expensive in terms of storage and network bandwidth. Its rules (*e.g.*, halving the reward to miners every four years) seem arbitrary and likely to cause problems. Given the fixed blocksize, and blocks being created at a fixed interval (ten minutes), the number of transactions per unit time is limited. One of the supposed benefits of Bitcoin is low transaction fees, but when miners are only rewarded through transaction fees, miners will only choose to include transactions with the largest transaction fees. How high are these fees likely to get?

To respond to these issues, people have deployed mechanisms such as the Lightning Network [POON16] to allow multiple small transactions to occur off of the main blockchain and then have multiple accumulated transactions be summarized with just a few big transactions listed on the main blockchain.

Another issue could be that if there are independent implementations of miners, implementations might have incompatibilities such that the community partitions, ignoring each other's blockchains as invalid. This actually occurred (in March 2013). There was a block B that looked valid to one version and looked invalid to the other. All the nodes that thought B was valid built upon the blockchain that included B, and the nodes that thought B was invalid built a chain that forked starting at the block before B. This situation lasted six hours but could have lasted for much longer before it was detected and fixed. The rule of thumb is that if the blockchain contains six blocks after the block containing a transaction, then the transaction can be assumed to be safely committed. But this obviously would not be true if there were frequent incidents of subtly incompatible miner implementations.

Note that when “someone” makes the decision to choose one fork in a case like this, the bitcoins earned by miners that thought they had won the lottery by having found blocks in the losing chain lose those bitcoins. And transactions that seemed to be safely recorded in the losing fork are no longer recorded.

As for the complaint about storage and network bandwidth, the Bitcoin proponents claim that both storage and bandwidth capacity will increase faster than the growth of the use of Bitcoin.

15.4 WALLETS FOR ELECTRONIC CURRENCY

Most people prefer keeping their (traditional) money in a bank rather than hidden in their house, because they assume that the bank has better security measures and that the money is insured even if the bank is robbed or goes out of business.

Electronic money presents a challenge as to how a person, say, Alice, can keep it safe. If a thief steals the information associated with a coin, they can spend the coin, and Alice will have no recourse. Electronic money is not the same as physical coins—it is information stored in Alice’s computer. For instance, it might be a set of signed eCash coins, or it might be the private keys associated with received bitcoins. So, there are at least two things that can go wrong:

- Alice’s computer could get lost or broken, and if Alice did not properly back up the contents, all the stored electronic money would be lost. This is analogous to her money burning up in a house fire.
- Alice’s computer could get broken into, and the money could be stolen. This is analogous to her house being robbed.

15.5 HOMEWORK

1. If Bob receives an eCash coin from Alice, and Bob trusts Alice not to double-spend, could Bob pay Carol with the same coin (without turning it into TheBank and converting it into a new coin)?
2. In the online eCash scheme, we say that a coin must contain not only a large random number, but also some formatting information. Suppose instead an eCash coin consisted solely of a random number x , signed by TheBank’s RSA private key (meaning that the coin would consist of $x^d \bmod n$). Why would this not be secure? Why does the formatting make it secure?
3. If TheBank issues eCash coins in different denominations, why does it need a separate public key for each denomination? For instance, why couldn’t the eCash coin contain both the UID and the value of the coin?
4. Here are (slightly simplified) details for the offline eCash scheme. For each of the 256 coinlets, say coinlet T_i , Alice’s ID is **Alice**. She chooses three random numbers: a_i , c_i , and d_i . Coinlet T is constructed out of the values **Alice**, a , c , and d as follows:
First Alice computes the hashes of two values: $\text{hash}(a|c)$ and $\text{hash}((a \oplus \text{Alice})|d)$, which she then concatenates and hashes to obtain $T = \text{hash}(\text{hash}(a|c) \mid \text{hash}((a \oplus \text{Alice})|d))$. Then left-info for the coinlet is the triple $\langle a, c, \text{hash}((a \oplus \text{Alice})|d) \rangle$, and right-info for the coinlet is

the triple $\langle \text{hash}(a|c), a \oplus \text{Alice}, d \rangle$.

- (a) Show how to compute T from left-info.
 - (b) Show how to compute T from right-info.
 - (c) Show how knowing only left-info or right-info does not divulge Alice's identity.
 - (d) Show how knowing both left-info and right-info does divulge Alice's identity.
5. In the offline eCash model, how could a malicious bank make it look like innocent Alice is double-spending?
 6. In Bitcoin, assume Alice changes her public key on every transaction. When she spends what she received in a transaction, she has to list her public key (so that her signature can be verified). Given that she has to divulge her public key when spending, why does it add security for her to only reveal the hash of her key to receive money?
 7. Suppose someone replaced the hash algorithm currently used in Bitcoin with a hash algorithm that took a tenth as much compute power per hash. Would the Bitcoin community therefore use less electricity?

16

CRYPTOGRAPHIC TRICKS

We've covered the major cryptographic building blocks, such as secret key encryption and integrity checks, public key encryption and signatures, and hashes/message digests. In this chapter we'll talk about some other functions. We won't go into great detail about the math, or proofs, but we will at least demystify what these functions are attempting to do, give some intuition about how they accomplish it, and give examples of those that are used in actual real-world protocols.

16.1 SECRET SHARING

Secret sharing is a way for someone, say, Alice, to store a piece of information, say, S , that must be kept secret from all except Alice but must also be retrievable by Alice. There is a tradeoff between robustly storing S so that it can always be retrieved or risking having S stolen.

Alice will compute n numbers (known as **shares**) such that any k of those numbers will enable computing S , but any subset of size smaller than k will give no information about S . Alice can store the n shares in different locations. The parameters k and n should be chosen so that

- k is larger than the number of locations Alice thinks could be broken into, while
- $n-k$ is at least as large as the number of locations Alice thinks could lose the information she stores or be unavailable when needed.

There are various schemes for accomplishing secret sharing. Shamir's scheme [SHAM79] is easiest to understand, although the concept of secret sharing, using a different mechanism, was also invented by Blakley [BLAK79].

First, let's see how Shamir's scheme handles $k=2$, meaning it will require two out of the n shares to recover S . Alice chooses a random number b and creates an equation for a line $y=bx+S$. Each share of S consists of a point $\langle x, y \rangle$ on the line. The secret S is the value of y when $x=0$. If Alice (or an attacker) were to be told any two points $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$, they could compute the line and therefore know S . But any single point will yield no information.

For general k and n (n shares, k of which can reconstruct S), the equation would be a polynomial of degree $k-1$. Shares can be any n distinct points on the curve represented by the equation (other than the point where $x=0$, of course).

Usually when we think of equations and curves, we think of points on the curve as being pairs of real numbers (the x and y values) and the equation having coefficients that are real numbers. That would be very messy—digital computers are notoriously imprecise at computing with real numbers. Instead, in a system used for secret sharing, an example implementation might have the coefficients being integers mod p for some suitably large prime p , and points fitting the equation would be tuples of integers mod p . This is possible because integers mod p form a field. For n -bit secrets, the obvious field of choice would be $\text{GF}(2^n)$ (see §2.7.1 *Finite Fields*).

16.2 BLIND SIGNATURE

The concept of a **blind signature** was invented by David Chaum [CHAU82]. The idea is for Alice to get Bob to sign a message m without Bob being able to see what he's signing. It is rather surprising that such a protocol exists, that it would be useful for anything, and that anyone would have thought of it!

Alice chooses two functions *blind* and *unblind* that interact in a special way with Bob's public key pair. She applies the *blind* function to m , which disguises m so that Bob cannot know m . Then Bob signs the blinded m . Alice then applies the *unblind* function, and the result is Bob's signature on m . (See Figure 16-1.)

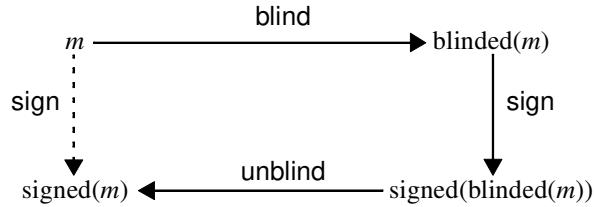


Figure 16-1. Blind Signature

An example application for blind signatures is anonymous electronic cash (see §15.1 *ECASH*). Another is for one of the variants of group signatures (see §16.5.1.4 *Blindly Signed Multiple Group Membership Certificates*).

Blind signature schemes exist for many public key signature algorithms, but the one that is easiest to understand is based on RSA. Let Bob's RSA public key be $\langle e, n \rangle$ and private key be $\langle d, n \rangle$. Alice wants m signed by Bob's private key $\langle d, n \rangle$, meaning she wants $m^d \bmod n$.

To blind m , Alice chooses a random number R and applies Bob's public key to R (meaning that she computes $R^e \text{ mod } n$). She then multiplies the message m by $R^e \text{ mod } n$, and the result is $mR^e \text{ mod } n$, which is the blinded message.

Next, she asks Bob to sign the blinded message, which means he will exponentiate $mR^e \text{ mod } n$ to the power d , and reduce the result mod n . The result is $m^d R^{ed} \text{ mod } n$. Since e and d are inverses, $R^{ed} \text{ mod } n$ is equal to $R \text{ mod } n$, so the blinded signed message is $m^d R \text{ mod } n$, which Bob sends to Alice.

Now Alice unblinds by multiplying $m^d R \text{ mod } n$ by $R^{-1} \text{ mod } n$, and the result is $m^d \text{ mod } n$ —in other words, m signed by Bob, even though Bob was not able to see what he was signing.

16.3 BLIND DECRYPTION

Blind decryption is very similar to blind signatures, and for RSA, the same math works. Alice has $\{X\}_{\text{Bob}}$ and wants to retrieve X . She chooses functions (blind and unblind), applies the blind function to $\{X\}_{\text{Bob}}$, and sends the result to Bob. Bob applies his private key, which reverses the encryption with Bob's public key, resulting in X encrypted with the blind function. Alice can then decrypt with the unblind function to retrieve X .

An example use of blind decryption is for a concept known as **oblivious transfer**, which was introduced in [RABI81]. The problem being solved is that Bob has several items that he can send to Alice, but Alice does not want Bob (or an eavesdropper) to know which item she has retrieved.

For instance, Bob might be a content provider, where Alice pays to download a movie but does not want anyone to know which movie she has purchased, such as was described in [PERL10]. Bob might post a bunch of encrypted movies, each encrypted with its own secret key K . Associated with each movie is a header consisting of K encrypted with a public key that Bob created for that purpose, say, Bob-movies.

So, associated with each movie is $\{K_i\}_{\text{Bob-movies}}$.

Alice can download any encrypted movie. However, to decrypt it, she will need the key K_i . She needs to pay Bob for each movie but does not want Bob to know which movie she has purchased.

The solution is simple. Alice selects $\{K_i\}_{\text{Bob-movies}}$ for the movie she wants and asks Bob to blindly decrypt it.

Of course, to build a complete solution and preserve Alice's privacy requires some sort of mechanism for Alice to obtain the encrypted movie without anyone noticing which movie she downloaded. One possibility is that the movies are broadcast, and Alice, knowing the decryption key, can watch the movie when it is broadcast. Another is that Alice might download the encrypted movie through some sort of anonymizer such as TOR [DING04].

16.4 ZERO-KNOWLEDGE PROOFS

The concept of zero-knowledge proofs (ZKP) was introduced in [GOLD85]. The basic goal is for Alice to prove to Bob that Alice knows a secret without revealing the secret to Bob. Furthermore, the *zero knowledge* part means that Bob should gain no information from the interaction other than that the party he is talking to knows the secret.

Traditional ZKPs are interactive protocols, where there is a per-round probability (in most examples, $\frac{1}{2}$) that someone impersonating Alice would be able to fool Bob. If Bob quizzes Alice n times, the probability that an imposter could fool him all n times would be $\frac{1}{2}^n$.

ZKP should also have the property that although Alice can convince Bob that she knows Alice's secret, Bob would not be able to show the transcript of the conversation to a third party to prove that Bob, indeed, had talked to Alice, because Bob could have created the entire conversation himself.

16.4.1 Graph Isomorphism ZKP

A graph is a set of vertices plus a set of links connecting pairs of vertices. If we label the vertices and represent each link as an unordered pair of vertex labels, then two graphs G_1 and G_2 are **isomorphic** if there is a way of relabeling the vertices in G_1 to produce G_2 . It is assumed difficult to determine in all cases, whether two graphs are isomorphic. (See Figure 16-2.) The best-known solution is harder than polynomial. But if presented with a renaming of vertices in G_1 to G_2 , it is easy to show that they are isomorphic.

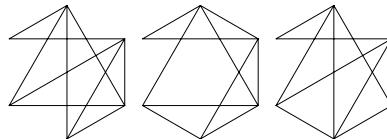


Figure 16-2. Isomorphic Graphs?

A ZKP for graph isomorphism is for Alice to prove that she knows that G_1 and G_2 are isomorphic, meaning she knows how to rename the vertices of G_1 to produce G_2 . She can produce two such graphs by taking a large graph G_1 and renaming the vertices to produce G_2 . She will now know how to map from G_1 to G_2 , and her “public key” is the pair of graphs $\langle G_1, G_2 \rangle$. But it is difficult for Bob to find a relabeling to be sure that G_1 is isomorphic to G_2 .

For Alice to prove to Bob that she knows that G_1 and G_2 are isomorphic, Alice chooses a random relabeling of one of G_1 or G_2 to produce a new graph G_3 . Alice presents G_3 to Bob. This step is known as Alice's **commitment**.

Bob then challenges her by choosing either G_1 or G_2 and asking Alice to show how it maps to G_3 . If Alice indeed knows how to map G_1 to G_2 , she can correctly answer his question, whether he chose G_1 or G_2 . Suppose an imposter, say Trudy, attempts to impersonate Alice. Trudy does not know how to map G_1 to G_2 . If Trudy created G_3 by renaming the vertices in G_1 , she will be able to show the mapping to G_3 if Bob challenges her with G_1 . However, if Bob challenges her with G_2 , Trudy will not be able to answer correctly. If Trudy were able to pick a G_3 and be able to correctly answer whether Bob challenged her with G_1 or G_2 , then Trudy could compose the mappings to get a map from G_1 to G_2 .

This means that Trudy has a $\frac{1}{2}$ chance per round of fooling Bob. Given that Bob needs to be more than 50% sure he's talking to Alice, he can perform more rounds. At each round, Alice must choose another graph, G_i , and Bob will challenge her to show a mapping between G_i and his choice of G_1 or G_2 .

16.4.2 Proving Knowledge of a Square Root

Another example of a ZKP protocol was published in [FIAT86]. The rules are as follows. We assume there is a composite modulus, n , but nobody (except possibly Alice) knows n 's factorization. Alice creates a public key $s^2 \bmod n$ by picking a random s and squaring it mod n . In order to authenticate to Bob, Alice must prove to Bob that she knows s (her private key).

- Message 1: (Alice's commitment) Alice chooses a random r and sends $r^2 \bmod n$ to Bob.
- Message 2: (Bob's challenge) Bob asks Alice either for r or for rs .
- Message 3: If Alice really knows s , it will be easy for her to give either r or rs . If Bob asked for r , he squares it and verifies that the result equals Alice's commitment. If Bob asked for rs , he squares that and verifies that the result is equal to the product of Alice's commitment and Alice's public key.

If Trudy is impersonating Alice, then she has a $\frac{1}{2}$ chance of fooling Bob. If she thinks Bob will ask her for r , then she sends r^2 in message 1. However, if she thinks Bob will ask for rs , then Trudy picks a random number, say, t , and sends $t^2/s^2 \bmod n$, in message 1. When Bob asks for rs , Trudy sends t . Bob will square t (to get t^2) and verify that what he received from Trudy (t^2/s^2) multiplied by s^2 is indeed t^2 , so Bob will be fooled. However, if Trudy made the wrong prediction for whether Bob will ask for r or for rs , she will not be able to answer correctly.

16.4.3 Noninteractive ZKP

Any ZKP can be turned into a noninteractive ZKP by having the prover (Alice) simulate the challenger (Bob) with challenges that she cannot control. A typical way of doing this is by deriving the challenges from a cryptographic hash of Alice's commitments.

So, for instance, let's take the protocol in §16.4.2 *Proving Knowledge of a Square Root*. Let's say that Alice wants to simulate k rounds. She creates k inputs $(r_1^2, r_2^2, \dots, r_k^2)$. Then she takes a hash of $r_1^2|r_2^2|\dots|r_k^2$. If the i th bit of the hash is 1, then she must provide r_i s. If the i th bit of the hash is 0, then she must provide r_i .

If someone, say, George, wanted to impersonate Alice without actually knowing s , George can choose $r_1^2, r_2^2, \dots, r_k^2$, but he will only know one of r_i or $r_i s$, for each i . He could hash $r_1^2|r_2^2|\dots|r_k^2$, and if he doesn't like the challenges selected by that hash, he can choose new r_i s, rehash, and hope for the best. The assumption is that the hash would be large enough that it would be computationally infeasible for George to be lucky enough to find a set of r_i s that result in a set of challenges that he can answer. Note that twenty rounds might be sufficiently secure in an interactive ZKP where a forger can answer each round half the time. With a noninteractive ZKP you'd need many more rounds than that. (See Homework Problem 8.)

This form of noninteractive ZKP can be used as a type of signature if, instead of simply hashing the set of commitments as described above, Alice hashes the concatenation of the message and the commitments.

The formal definition of **zero knowledge** requires that a third party, seeing a transcript claimed to be an interaction between Bob and Alice, will not be convinced that Bob actually interacted with Alice, since Bob could have created that transcript without Alice. This strict definition is useful in some security proofs, but it is not necessary for any plausibly practical application of ZKP that we know of.

The transformation of a zero-knowledge proof into a non-interactive ZKP is known as the **Fiat-Shamir construction**. Most proposed applications of ZKP use the Fiat-Shamir construction. These only require a weaker notion of ZKP (**honest verifier zero knowledge**) where it only needs to be zero knowledge if Bob chooses his challenges randomly.

Suppose an impersonator has a 50% chance of answering correctly on each simulated round in a noninteractive ZKP. Given that the input to each round (the commitment) is many bits long, the Fiat-Shamir construction would be highly inefficient. Therefore, to make a practical non-interactive ZKP, the challenges need to have a far smaller probability that an imposter can respond correctly.

Here is an example protocol original published by Schnorr [SCHN89] with a small probability of an imposter creating a correct response. In this protocol, Alice proves that she knows the discrete log, x , of a public key $g^x \bmod p$.*

*This protocol is a secure honest verifier zero-knowledge proof system to the extent that the underlying discrete log problem is hard. This will depend on the factorization of $p-1$ in various ways that aren't worth getting into here.

-
1. Alice picks a random number r and sends Bob $g^r \bmod p$ (Alice's commitment).
 2. Bob sends Alice a random number c (Bob's challenge).
 3. Alice sends Bob $r - cx \pmod{p-1}$ (Alice's response).
 4. Bob verifies that $(g^{r-cx})(g^x)^c = g^r \bmod p$.

The above protocol (where Alice proves she knows the discrete log of $g^x \bmod p$) and other such proofs that have very small probabilities of an imposter guessing a single correct response are honest verifier zero-knowledge proofs (see Homework Problem 3 and Homework Problem 4).

16.5 GROUP SIGNATURES

The goal of group signatures is to have a member of a group of individuals sign something, such that a verifier can know that it was one of the individuals in the group that signed it but not know which member of the group generated the signature.

One application of group signatures is to enable someone (known as a whistleblower) to safely divulge some wrongdoing to a news outlet. The news outlet needs to know that the whistleblower is genuinely one of the people who would have accurate information (rather than some troublemaker fabricating a story). But it is also important to keep the whistleblower's identity secret.

Another example application is for a device to prove it is a genuine device, while preserving the privacy of the owner of the device. An example use is for DRM (digital rights management). A corporation selling content might want to be assured that the device purchasing the content was one of a set of devices designed to follow the DRM rules. Users might be hesitant to purchase content if the corporation could determine which device was purchasing the content.

The concept of group signatures was introduced in [CHAU91]. It should be impossible to forge a signature if you're not part of the group. Seeing a signature should not leak information, so even if you see a lot of them, you shouldn't be able to forge a signature. Some group signature schemes involve a **group manager** that creates the credentials for the group.

There are various schemes that have been proposed and various potential properties they might have, such as:

- unlinkable signatures—the verifier does not know whether two signatures were signed by the same group member
- a choice between two properties—either nobody would be able to identify which member generated a given signature, or there is one entity (the group manager) that is able to identify the member that generated a given signature

- if the private key of a compromised member becomes known, it is possible to revoke that member's membership, so signatures from that member will no longer be verified as valid by verifiers
- if a signature is done in bad faith (*e.g.*, someone agrees to purchase something but doesn't pay), even though the particular member who generated that signature cannot be identified (even by a group manager), that signature can be revoked, so that in the future a prover can prove that they were not the one that generated a particular signature

16.5.1 Trivial Group Signature Schemes

In this section we will look at simple ways of achieving some of the features that are generally desired of group signature schemes. The sorts of schemes we discuss in this section might be considered too trivial to be considered "true" group signature schemes. We will also discuss the drawbacks of these simple approaches to see why (aside from getting papers published) cryptographers might want to design more complicated schemes for group signatures.

16.5.1.1 Single Shared Key

In this trivial approach, all the members of the group share the same private key. This simple scheme does have the property that a verifier would not be able to tell which member signed something or to tell that two signatures were created by the same device.

However, sharing a private key with many individuals is not very secure. If many individuals (or devices) know the same private key, it is likely someone outside the group would eventually figure out how to extract the private key.

To revoke a single member of the group, the group key would need to be revoked, a new group key created, all verifiers notified that the group key has changed, and the new private key would need to be shared by all the remaining group members.

16.5.1.2 Group Membership Certificate

This simple approach to group signatures is to have each member create a new public key pair specifically for signing group signatures for this group. Member Bob creates a new public key P and sends it (signed with his normal private key) to the group manager. The group manager signs a certificate for P , " P is part of this group". The group manager would know which public key is associated with each member. The verifier would not know which public key goes with each member, but the verifier would know whether two items were signed by the same individual. Revocation of a member would be easy. The group manager revokes the certificate for the public key associated with that member.

16.5.1.3 Multiple Group Membership Certificates

This variation provides unlinkability of signatures (except by the group manager). Each member creates lots of public key pairs and has the group manager certify each of those keys. The group manager will know which public keys are associated with each member. However, a verifier will not know which public keys are associated with each member. Therefore, if a member uses a different private key for each signature, then a verifier will not be able to link two signatures as being from the same member. To revoke a member, the group manager will need to revoke all certificates for that member.

16.5.1.4 Blindly Signed Multiple Group Membership Certificates

This variation makes it impossible for the group manager to know which member is associated with a given public key. It uses blind signatures (§16.2). Let's say Bob is a member of the group. Bob will authenticate to the group manager, create a public key P , embed P in a blinded message that says "public key P is a member of the group". The group manager signs the blinded message. If Bob uses a different public key pair and certificate (blindly signed by the group manager) each time he creates a signature, then nobody, including the group manager, will be able to know which member created a signature, nor be able to link two signatures.

With this variant, if the group manager wants to revoke Bob's membership, it can refuse to blindly sign any new certificates for Bob, but will not be able to revoke the certificates the group manager had blindly signed before Bob's membership was revoked. Instead, to revoke a member, the group manager will need to change its own public key pair, declare all certificates signed by the old key to be invalid, and issue new certificates for the still-valid members, blindly signed with the group manager's new private key.

Alternatively, instead of only changing the group manager's key when a member is revoked, the group manager key could be changed frequently, for instance, every hour. To avoid problems for a member who receives a group membership certificate right before the group manager changes its key, there should be an overlap in time when verifiers accept certificates signed either by the current group manager key or the previous one. Perhaps each member might request a single blindly signed certificate at the point when it needs to sign something, in which case there is a risk that the group manager can correlate when Bob asked for a certificate and when it was used for a signature.

16.5.2 Ring Signatures

This elegant scheme was presented in [RIVE01]. It is particularly suited for the whistleblower scenario. Each individual has its own public key pair, and the set of individuals in the group does not have to be determined in advance. Anyone, say, Bob, can choose a set of $n-1$ individuals and be

able to hide his identity (as the signer) in the group of n individuals. The verifier will know that one of those n users signed the message but would not know which one.

This scheme is described in the literature in a much more general way, and needs to be tuned to the specific cryptographic algorithms used. For simplicity of explanation, we'll choose unpadded RSA, SHA2-256, and AES-256 as the functions.

Assume there are n individuals, M_1, M_2, \dots, M_n . Each individual in this group has a public key pair (say a 2048-bit RSA key). The public keys of all the members are known to everyone. We'll use the notation that a 2048-bit quantity, say R , encrypted with M_i 's public key is $\{R\}_i$. We'll be using unpadded RSA encryption so that any 2048-bit quantity smaller than the modulus can be encrypted or decrypted.

Let's say there is a message m that a member, say M_3 , would like to sign. The SHA2-256 hash of m is h . This hash will be used as an AES key in some mode such as CBC with zero IV so that the encryption of a 2048-bit quantity by h results in a 2048-bit quantity.

The ring signature on message m (with n individuals in the group) includes $n+1$ seemingly random 2048-bit numbers $y_1, R_1, R_2, \dots, R_n$. (See Figure 16-3.) The signature could start with any of the y values, but it's simpler to always start with y_1 .

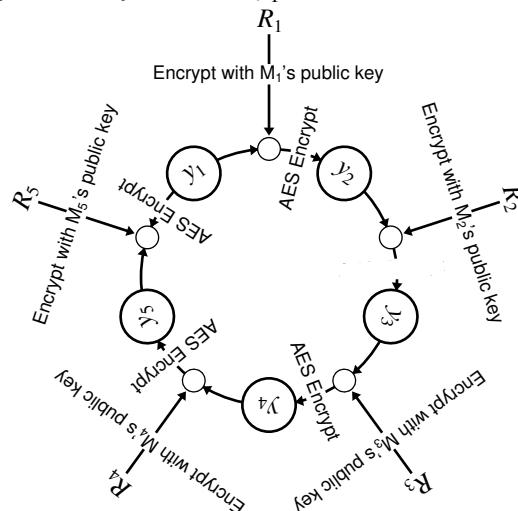


Figure 16-3. Ring Signature

To verify a signature, begin by computing the SHA2-256 hash h of m . h will be the key we use for each of the AES encryption steps. Then, starting at, say, $i=1$, compute around the ring for n steps:

- Encrypt R_i with M_i 's public key to get $\{R_i\}_i$. Compute $y_i \oplus \{R_i\}_i$
- Set y_{i+1} to the AES-encryption (using key h) of $y_i \oplus \{R_i\}_i$

After completing the above for each member, you will have computed y_2, \dots, y_6 . If $y_6 = y_1$, then the signature will be considered valid.

Suppose Bob is member M_3 . To *compute* a signature for message m with hash h , Bob picks a random 2048-bit number for y_4 and $n-1$ random numbers $R_1, \dots, R_2, R_4, \dots, R_n$. (Note that R_3 will be computed later; see Figure 16-4.) Bob encrypts R_4 with member M_4 's public key and \oplus s the result with y_4 . Then he AES-encrypts the result with h , and the result is y_5 . He continues around the ring computing each y_i until he gets to y_3 . Now he computes the AES-decryption of y_4 using key h . Let's call that X . To complete the ring, Bob computes $y_3 \oplus X$ and decrypts the result using his own private key. If Bob is unlucky, and his calculation results in a number that is larger than his modulus, he will have to change one of the random numbers and repeat the calculation. The result will be the R_3 that he supplies as part of the group signature.

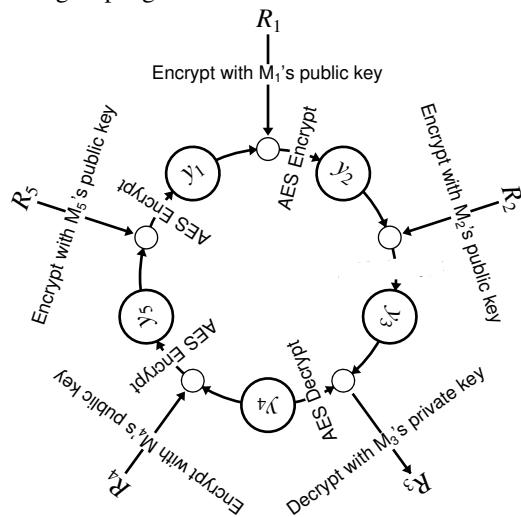


Figure 16-4. Ring Signature Creation

Someone who knows any one of the private keys can compute a ring of values that satisfies the verification algorithm, but the verifier will have no way of knowing which private key was used.

16.5.3 DAA (Direct Anonymous Attestation)

DAA [BRIC04] is a group signature scheme that has been adopted by the TCG (Trusted Computing Group) for use by TPMs (Trusted Platform Modules). An example application that would be suited for DAA is DRM (Digital Rights Management). Attestation is a signed statement of something such as, in this case, the configuration of the hardware and software.

An overview of DAA (without the math) is that there are three entities—a DAA issuer (that issues DAA credentials), a platform (that is doing the attestation using the credentials it has received), and a verifier.

If the credential were simply a certificate, the platform would present the certificate to the verifier, along with proof that the platform knows the private key associated with the public key in the certificate. However, DAA does not do this. The dialog between the platform and the prover consists of a (noninteractive) zero-knowledge proof that the platform has a valid credential issued by the DAA issuer.

If the private key of a platform becomes known (someone has broken into the platform and obtained the key and posted it somewhere), then all previous attestation proofs using that private key can be identified, and future attestations can be refused.

If the private key is not known, even the DAA issuer would not be able to know which platform has issued a particular proof, and the only way to revoke it would be to rekey all valid devices.

16.5.4 EPID (Enhanced Privacy ID)

EPID [BRIC10] is an enhancement to DAA that allows an additional form of revocation. The only form of revocation possible in DAA is revocation of a private key, and it requires that the private key of the member to be revoked be known in order to do revocation of that member.

With EPID, there is a way to revoke “whoever generated this signature”. Suppose there were a revocation list of “bad signatures”, where it’s not known who generated those signatures, but the signatures were created by some platform in the past that was later found to have acted badly in some way. With EPID, the verifier asks the platform to prove it is a valid member of the group (as it would with DAA). In addition, the verifier can present a list of bad signatures to the platform, and the prover can prove (again, using a zero-knowledge non-interactive proof) that its signature was produced by a different key than each of the signatures on the list of bad signatures.

16.6 CIRCUIT MODEL

A concept that we’ll need for §16.7 *Secure Multiparty Computation (MPC)* and §16.8 *Fully Homomorphic Encryption (FHE)* is that of a **circuit**. A circuit is a type of algorithm for computing an output from an input, consisting of a sequence of simple operations called gates. The output of a gate may either be the final output of the circuit, or it may be an intermediate value. The gates can act on the inputs of the circuit, intermediate values, or constants (*e.g.*, a value that is always 1). One way that circuits differ from more general kinds of programs is that the sequence of gates and

which values they act on cannot depend on the input to the circuit, so circuits cannot have branches, loops, or references to a memory location specified by the input. Nonetheless, it is a theorem that any program with a fixed-size input and a known maximum number of computation steps can be converted into a circuit acting on 0s and 1s and consisting of only two types of gates:

- bitwise AND (\wedge) (mod 2 multiplication)
- bitwise XOR (\oplus) (mod 2 addition)

Executing a program as a circuit can be expensive. For instance, if there is a branch, you have to append two circuits—one for computing based on the computation that would be done if the condition were known to be satisfied and the other for the computation to be done if the condition were known to be not satisfied. If the program has a loop that executes a variable number of times, the circuit has to execute the loop the maximum number of times. Also, to look up something in a large database (assuming the index of the database cannot be predicted without knowing the input to the program), the cost of looking up a single item in the database will be proportional to the total size of the database.

16.7 SECURE MULTIPARTY COMPUTATION (MPC)

Secure multiparty computation [BENO88] (sometimes abbreviated as MPC) consists of having a group of n participants, each with its own input, collectively compute a function of the n inputs without divulging its own input to the other participants. One simple way of accomplishing this is to have all participants send their inputs to someone, say, Tom, that all the participants trust, and have Tom compute the function of all the inputs. But the cryptographers that have been devising protocols for secure multiparty computation are assuming there is no trusted Tom, and so the participants have to somehow collectively compute the function. One model is “honest but curious”, where the participants do the computations honestly, but the only mischief the system is designed to prevent is participants learning information about other participants’ inputs. That is what we’ll describe here. There are additional research papers that discuss how to detect and protect against maliciously incorrect computation.

An example function where participants might want to keep their inputs secret is an auction, where the participant with the highest bid will win the auction (and have to pay what they have bid). If you knew all the other bids, you could bid just slightly higher than the highest bid, so the participants do not want their bids known by the other participants, even after the auction is completed. There are several published protocols for computing a result based on inputs from several participants, where no one can see any inputs other than their own. Indeed, one of these protocols has been used in the Danish sugar beet auction [BOGE09]. It is an expensive protocol. But it only happens once a year, so the Danish sugar beet farmers are reportedly happy with the result.

There are various schemes, but most secure multiparty computation is based on secret sharing (see §16.1 *Secret Sharing*). The computation needs to be converted into a circuit (see §16.6). Then each participant's input is divided into shares that are distributed to the other participants.

After a process where the participants compute on the shares they have been given and communicate with one another to produce shares of intermediate values, the final output can be constructed from a quorum of the participants combining their shares of the output.

Each participant P_i has an input p_i that will be used in computation by the community of n participants. We'll use the term t -share to refer to a share of an input based on a degree- t polynomial. Participant P_i uses secret sharing, with a degree t polynomial, to create and distribute t -shares of its input to each of the other participants. Now each participant has n numbers, each of which is a t -share of the input of each of the n participants. Note that a group of $t+1$ participants could collude to recover the input of any of the participants.

Suppose a computation that the group must perform is to add the inputs of P_k and P_j . Addition is easy. Each participant takes its share of p_k and adds it to its share of p_j . Everyone now has a share of $p_k + p_j$.

Multiplication is more complicated. To compute $p_k p_j$, everyone multiplies their t -share of p_k to their t -share of p_j . This, unfortunately, doubles the degree of the polynomial, so they now all have a $2t$ -share of $p_k p_j$. It will now take $2t+1$ participants to recover $p_k p_j$. It will be necessary to reduce the degree of the polynomial for the item for which the participants have $2t$ -shares. This is done with two rounds of communication, where every participant sends a message to every other participant, followed by computation.

- Round 1: Each P_i creates n t -shares of its $2t$ -share of $p_k p_j$ and then sends the corresponding t -share to each of the other participants. After this step, each of the participants will have n values, each of the n values being a t -share of one of the participants' $2t$ -share of $p_k p_j$. Then each participant takes the vector of these n values and multiplies that vector by a matrix that turns it into a different set of n values, which are each participant's t -share of their t -share of the product $p_k p_j$. In other words, the first of the n values is P_1 's t -share of P_1 's t -share of $p_k p_j$. The second of the n values is P_1 's t -share of P_2 's t -share of $p_k p_j$, and so on.
- Round 2: Each participant distributes the first of their n numbers to P_1 , the second number to P_2 , and so forth. What each participant will receive in round 2 is n t -shares of its own t -share of the product $p_k p_j$. From these (actually only $t+1$ of these are needed), each participant can now construct its own t -share of $p_k p_j$. So now, instead of having a $2t$ -share of $p_k p_j$, each participant has a t -share of $p_k p_j$.

Note that if t were much smaller than n , perhaps several multiplications could be done before the degree reduction step. However, since most applications don't involve very many participants, and since small values of t would be insecure (too easy for t colluding participants to find each other), most implementations set t to be the largest value for which $2t+1$ is no greater than n .

16.8 FULLY HOMOMORPHIC ENCRYPTION (FHE)

With homomorphic encryption, the data can be encrypted (in a special way), computation can be done on the encrypted data, and the result is the encrypted answer. So, computing on the plaintext data will yield the same answer as computing on the encrypted data and decrypting the result. One application of homomorphic encryption is to store your encrypted data in a public cloud and do computations on the encrypted data, without needing to trust the cloud not to leak your data. Another reason for doing this might be if Alice has secret data, and Bob has a proprietary program to do computation on the data. Alice doesn't want Bob to see her data, and Bob doesn't want Alice to learn about his program.*

There are alternative approaches that get some or all of the advantages sought by homomorphic encryption:

- Copy all your encrypted data to a location considered more secure, such as your private network, decrypt it, and then do your computation on the plaintext.
- Use **secure enclaves**, which are special hardware features in chips, such as Intel's SGX or AMD's SEV, designed to hide user-level data and code from potentially malicious hypervisors, operating systems, or cloud administrators with admin privileges. Secure enclaves do not significantly impact performance, and it is far easier to adapt a program to take advantage of secure enclaves than to turn a program into a circuit of \wedge and \oplus operations (which is what computing on homomorphically encrypted data requires). Secure enclaves may not give the provable security that homomorphic encryption could, especially since, historically, there have been side-channel attacks from which data could leak. However, this approach will always be more efficient than homomorphic encryption.

Homomorphic encryption has been a passion of the cryptography community for many years. As we said in §16.6, any computation can be done with a circuit consisting of \wedge (and/multiply) and \oplus (xor/add). So, all that is necessary to achieve a homomorphic encryption scheme is to figure out some mathematics that allows both operations on encrypted data.

There are schemes where it is possible to do one operation but not the other. For example, RSA encryption (without padding) can do multiplication homomorphically. If you have two messages m_1 and m_2 , a public key $\langle e, n \rangle$, and private key $\langle d, n \rangle$, multiplying the encryption of m_1 ($m_1^e \bmod n$), by the encryption of m_2 ($m_2^e \bmod n$) will yield $(m_1 m_2)^e \bmod n$. Then if you decrypt $(m_1 m_2)^e \bmod n$ (by mod n exponentiation to the power d), it will have the same result $(m_1 m_2)$ as having multiplied the plaintexts m_1 and m_2 . But that's only one operation (multiplication).

*Even if Alice cannot see Bob's program, because she sends her encrypted data to Bob to run his computation, there is the worry that the pattern of noise that Alice sees in the result might divulge something about Bob's program. To avoid this potential issue, an extra property called **circuit privacy** is needed. FHE schemes with circuit privacy have been proposed, and they are not markedly less efficient than schemes that are not designed to have circuit privacy.

There are also schemes where you can do one operation many times and the second operation only a limited number of times. These schemes are called “somewhat homomorphic encryption”. An example we’ve already seen that behaves similarly to somewhat homomorphic encryption is the scheme in §16.7 (if you don’t do the degree reduction step when you multiply). In most somewhat-homomorphic-encryption schemes, the number of additions and multiplications that can be performed on ciphertext is limited due to “noise” that must be combined with ciphertext in order to make the homomorphic encryption secure. Addition and more so multiplication increase the amount of noise. If there is too much noise, decryption is no longer possible.

16.8.1 Bootstrapping

Craig Gentry [GENT09] came up with a trick that eliminates the issue of a homomorphic operation that can only be done a limited number of times. He calls the trick *bootstrapping*, which reduces the amount of noise that ciphertext has accumulated after some computation steps.

We assume the cloud, which is not allowed to see the plaintext or private key, needs to do the bootstrapping. In order to do this, the cloud needs to know

- the user’s public key, (which enables the cloud to convert plaintext into minimally noisy ciphertext), and
- the user’s private key, encrypted with the user’s public key.

If the cloud knew the actual private key, it could decrypt the noisy ciphertext C and re-encrypt it with the public key. But the cloud is not allowed to see the actual private key or the plaintext. The cloud *is* allowed to see the user’s private key homomorphically encrypted with the user’s public key, so we will provide that.

The cloud first encrypts the noisy ciphertext C with the user’s public key, resulting in doubly encrypted data. Typically, each bit of C will be independently encrypted, so the doubly encrypted data will be vastly larger than C . The cloud then homomorphically evaluates the decryption circuit using the encrypted private key and the encrypted ciphertext (*i.e.*, the doubly encrypted plaintext). This results in a new singly encrypted plaintext. As long as the noise in the original singly encrypted plaintext (C) was sufficiently small that C was decryptable, the new singly encrypted plaintext will be an encryption of the same plaintext, and it will have a fixed amount of noise that will not depend on how much noise the ciphertext had accumulated.

As long as the bootstrapping operation itself plus any single operation doesn’t introduce too much noise, you can do arbitrarily many computations on the encrypted data, so long as you run the bootstrapping operation often enough. Because of the bootstrapping trick, many fully homomorphic encryption schemes have been proposed.

16.8.2 Easy-to-Understand Scheme

Most of the FHE schemes involve tedious math. Just for intuition, we will present one scheme [VAND10] that is surprisingly easy to understand. It is definitely not practical (*e.g.*, the encrypted data would expand by a factor of about a billion), but our purpose is to give the reader intuition.

This scheme requires encryption to be done on each bit. So it is necessary to know how to encrypt 0 and how to encrypt 1. The private key is a large odd integer n .^{*} This scheme will do ordinary integer arithmetic and not modular arithmetic.

To encrypt a bit if you know n :

- Choose some very large multiple of n .
- Add or subtract a relatively small even number (which we'll call *noise*). We'll call the value at this point a *noisy multiple* of n . If you are encoding a 0, you are now done.
- If the bit to be encrypted is a 1, add or subtract 1.

In order to allow someone who does not know n to encrypt, we create a public key that is a list of encryptions of 0. To encrypt a 0, someone would add together a randomly chosen subset of the encryptions of 0. To encrypt a 1, they would do the same thing but add or subtract 1 at the end.

Only someone who knows n will be able to decrypt. To decrypt a value x , find the nearest multiple of n and if the difference between x and the multiple of n is even, x decrypts to a 0. If the difference is odd, x decrypts to a 1.

Adding two encoded bits results in the \oplus (exclusive or) of the two plaintext bits. Multiplying two encoded bits results in the \wedge (and) of the two bits. Adding or subtracting 1 to an encoded bit computes the \neg (not) of the encoded bit. Note that the noise increases each time two encrypted bits are added or multiplied. If the noise ever gets bigger than $n/2$, decryption will no longer be guaranteed to produce the right answer (see Homework Problem 10). So, we can do additions and multiplications, but we have to avoid letting the noise get bigger than $n/2$. Another annoying issue is that the size of the encrypted bit doubles each time a multiplication is performed. If you multiply two billion-bit numbers together, you get a two-billion-bit number. It doesn't take too many multiplications before the numbers (which started out as ridiculously huge) to get super-absurdly ridiculously huge.

The paper has a very cute trick for reducing the size of an encrypted bit using a value similar to the public key. It is a value that is safe to give to the cloud, which will be doing the computation and the number reductions. The reduction list contains noisy multiples of n of different sizes, say m_1, m_2, m_3, \dots

To reduce x , take the largest m_j that is smaller than x and reduce $x \bmod m_j$. The result, say x' , can be reduced further by looking for the largest m_k that is smaller than x' and reducing $x' \bmod m_k$,

^{*}The paper uses p for the secret odd number, but since usually p is a prime, and it doesn't need to be a prime in this case, we use n .

and so forth. Although it's nice that this operation makes the encoding of x smaller, modular reduction increases the noise.

Additions will increase the noise somewhat, but multiplications and modular reductions will very quickly increase the noise. It is essential to bootstrap before the noise gets bigger than $n/2$.

Bootstrapping requires using the public key (the multiple encryptions of 0) to encrypt each bit of the noisy value, resulting in about a billion billion bits). Then this value needs to be homomorphically decrypted, using a circuit that homomorphically decrypts the result with a circuit that has access to a homomorphically encrypted version of the private key n .

As we said, we present this scheme only because it is easy to understand. There are less impractical homomorphic schemes proposed, and this is an area of active research.

16.9 HOMEWORK

1. In Shamir's secret sharing scheme (§16.1 *Secret Sharing*), if $k=7$ and $n=37$, what degree polynomial would you need to create shares?
2. In the graph isomorphism ZKP, why must Alice choose a different graph each time? (In other words, what would happen if Bob asked Alice to show how to map G_3 to G_1 in one round and asked Alice to map the same G_3 to G_2 in another round?)
3. Show how in the protocol in §16.4.3 *Noninteractive ZKP*, Bob could create a transcript without knowing x . (Hint: Have Bob choose random numbers for messages 2 and 3, and then compute what Alice would have needed to send in message 1.)
4. Suppose in the protocol in §16.4.3 *Noninteractive ZKP*, Bob were a dishonest verifier. How could he convince a third party that he actually communicated with Alice? (Hint: Instead of choosing a random number for message 2, his message 2 could be, say, a hash of message 1. Once committed to message 1, Bob can no longer derive message 1 from messages 2 and 3 without knowing x .)
5. In §16.5.2 *Ring Signatures*, we described the ring signature computation as working in the forward (clockwise) direction. How would you compute a signature working in the backward (counterclockwise) direction? Describe how you could start at any point in the ring and still compute a signature.
6. For each of the simple group signatures schemes in §16.5.1 *Trivial Group Signature Schemes*, specify its properties (linkability of signatures, whether the group manager can determine which member signed something, whether a verifier can know which member signed something, whether it is possible to revoke a group member).

7. Show how a ring signature scheme could be done with an irreversible function such as HMAC instead of a reversible function such as AES.
8. Why would it be secure to use twenty rounds in an interactive ZKP where an impersonator had a one-in-two chance of answering correctly on each round, but many more rounds would be required for a noninteractive ZKP?
9. Section §16.6 *Circuit Model* asserts that the only gates that are required are \oplus and \wedge . How would one create a \neg gate (a NOT gate)?
10. Consider the homomorphic encryption scheme described in §16.8.2. Why must n be odd?
11. In §16.8.2, why would an encrypted 0 look like an encrypted 1 if the noise ever got bigger than $n/2$?

17

FOLKLORE

Whenever I made a roast, I always started off by cutting off the ends, just like I'd seen my grandmother do. Someone once asked me why I did it, and I realized I had no idea. It had never occurred to me to wonder. It was just the way it was done. Eventually I remembered to ask my grandmother. "Why do you always cut off the ends of a roast?" She answered "Because my pan is small, and otherwise the roasts would not fit." —anonymous

Many things have become accepted security practice. Most of these are to avoid problems that could be avoided in other ways if you really knew what you were doing. It's fine to get in the habit of doing these things, but it would be nice to know at least *why* you're doing them. A lot of these issues have been discussed throughout the book, but we summarize them here. First, though, in §17.1 *Misconceptions* we debunk some things we often hear. The rest of the chapter covers things that are considered good to do. We explain why they came to be accepted practice and when they are actually important.

17.1 MISCONCEPTIONS

We list here several common security-related beliefs and describe what's really true:

- *Any program would run zillions of times faster on a quantum computer than on a classical computer.* In fact, classical programs would not run faster on a quantum computer. It is only a narrow set of problems that can benefit from algorithms that are designed for quantum computation, and these algorithms are very different from classical algorithms. A program designed for a classical computer would not run faster on a quantum computer.
- *Quantum computers would break all of cryptography.* In fact, hashes and secret key algorithms are not broken by any known quantum attack (aside from Grover's algorithm, which, even in the most optimistic scenarios for quantum computation, can be counteracted by doubling keysizes and hashsizes). It is only public key algorithms that are threatened, and they

will (hopefully) be long replaced by post-quantum algorithms before the world might have a quantum computer capable of breaking our currently deployed public key algorithms. Also, hopefully, any encrypted data that used current public key algorithms will no longer be sensitive at that point.

- *Quantum computers can factor prime numbers.* Indeed they can, but so can classical computers. In fact, you can probably factor a prime number in your head. We assume this oft-repeated sentence that quantum computers can factor prime numbers is just a misstatement, but people really ought to get in the habit of saying “quantum computers can factor numbers.”
- *It is good to make users change passwords every few months.* It depends on your goals. If your goal is to annoy users, this is a good strategy, but if your goal is to make things more secure, periodic password changes make systems *less* secure. If a bad guy stole a user’s password, they can do enough damage in the weeks between forced password changes that having the user change her password every few months will not enhance security. Also, forcing users to remember even more passwords will lower security. Given that users tend to be human, they will resort to a simple strategy such as ending their password with a number and incrementing that number at each cycle.
- *Any system using MD5 is insecure.* We are mentioning MD5 as an example, but we are really referring to any cryptographic algorithm that cryptographers have shown to have vulnerabilities. In fact, just because an algorithm is insecure in one usage does not mean that it will be insecure in all cases. If you are implementing a new system, you should definitely use algorithms that the cryptographic community believes in unless there is some really good reason (such as backwards compatibility) to use a deprecated algorithm. For an existing system, even if the deprecated algorithm would actually be secure in that context, there is the possibility that when analyzing its security, you might have missed some subtle property. Also, once a customer realizes you are using an algorithm that they have read is insecure, it will be time-consuming to explain to the customer why it is secure in that particular context. Explaining might be more trouble than modifying the working system to use more current algorithms.

17.2 PERFECT FORWARD SECRECY

Perfect forward secrecy (PFS) (see §11.8) is a protocol property that prevents someone who records an encrypted conversation between Alice and Bob from being able to later decrypt the conversation, even if the attacker has since learned the long-term cryptographic secrets of either or both

sides. A protocol designed with PFS cannot be deciphered by a passive attacker, even one with knowledge of Alice and Bob's long-term keys. However, an active attacker with knowledge of, say, Bob's long-term key can impersonate Bob to Alice or act as a MITM. And even if attacker Trudy doesn't know Bob's long-term key, if she can convince Alice that a different key belongs to Bob, Trudy can act as a MITM (see §11.5). PFS is considered an important protocol property to keep the conversation secret from

- an escrow agent who knows the long-term key
- a thief who has stolen the long-term key of one of the parties without this compromise being detected
- someone who records the conversation and later manages to steal the long-term keys of one or both parties
- law enforcement that has recorded the conversation and subsequently, through a court order, obtains the long-term key of one or both parties.

17.3 CHANGE ENCRYPTION KEYS PERIODICALLY

It is good to change keys (**do key rollover**) before a key is used on more than some amount of data, or for more than a certain amount of time. When a key needs to be rolled over depends on the encryption mode and the block size. For example, in CBC mode, due to the birthday problem (see §5.2), it is likely that after $2^{n/2}$ blocks have been encrypted with the same key, two ciphertext blocks will be equal. In that case (a collision of two ciphertext blocks) it will be possible to compute the \oplus of two plaintext blocks, which might leak information. If you'd like the probability of a ciphertext collision to be much smaller than 1/2, then the key should be changed more frequently than every $2^{n/2}$ blocks. For example, if you want the probability of a collision to be less than 2^{-32} , you should change keys before you've encrypted $2^{n/2-16}$ blocks.

In GCM mode, it is essential that the 96-bit per-message portion of the IV is not reused (with the bottom 32-bits of the 128-bit counter incremented for blocks within a message). If the application can definitely keep track of all IVs used, then it could keep the same key and encrypt 2^{96} messages with the same key. If, however, the application chooses the 96-bit per-message portion of the IV at random, then the application should not encrypt more than 2^{32} messages with the same key (making the probability of reusing a randomly chosen IV less than one in a billion).

Another reason to do key rollover (with perfect forward secrecy in the case of communicated data, rather than encrypted data-at-rest) is in case the data encryption key were stolen without your knowledge in the middle of the conversation. This would limit the damage from a stolen key.

With encrypted data-at-rest, if the data is directly encrypted with the data encryption key, it will be expensive, and probably not that helpful, to decrypt all the ciphertext and encrypt with a new key, because it is likely that there will be backups of the old ciphertext. However, if each message is encrypted with a randomly chosen data encryption key K , and associated with the data is K encrypted with a long-term encryption key S_1 , it is inexpensive to rollover S_1 , since only the metadata $\{K\}S_1$ needs to be decrypted with the old S_1 and encrypted with a new key, say S_2 .

17.4 DON'T ENCRYPT WITHOUT INTEGRITY PROTECTION

It is common misconception that if something is encrypted, it would be impossible to modify the ciphertext without it being detected. There are some encryption modes that include integrity protection, but if the mode does not include integrity protection, there are many attacks possible.

- In a mode such as CTR mode, if the attacker knows or can guess the plaintext, it is trivial for them to \oplus the desired changes to the plaintext into the ciphertext. Imagine a secret key network authentication system such as RADIUS (RFC 2058) in which an authentication server is configured with user secrets and shares a secret key with each server that a user might log in to. A server Bob will communicate with the authentication server over a secure session (and we are assuming here, using a stream cipher). Trudy connects to server Bob, claiming to be Alice. Bob sends Trudy a challenge and gets her response, then sends to the authentication server, “Alice sent Z to challenge X .” The authentication server responds either **success** or **failure**. Without integrity protection, Trudy (who is impersonating Alice) can intercept the message between the authentication server and Bob and \oplus into the message (**success** \oplus **failure**).
- In other modes, modifying the ciphertext would corrupt the plaintext without the attacker being able to cause a predictable change to the plaintext. Without integrity protection, the system might accept the corrupted plaintext, with unknown consequences. Who knows what the nuclear power plant would do when asked to execute the command `&(Hk2'#zAzq*$fc_)2@c` instead of `reduce output by 10%`?
- Integrity protection would prevent the vulnerabilities discussed in §17.5.

17.5 MULTIPLEXING FLOWS OVER ONE SECURE SESSION

Folklore says that different conversations should not be multiplexed over the same secure session. For instance (Figure 17-1), suppose two machines, F1 and F2, are talking with an IPsec SA and forwarding traffic between A and B and between C and D (where A and C are behind F1, and B and D are behind F2). F1 and F2 might be firewalls, with A, B, C, and D machines behind those firewalls. Or A and C might be processes on F1 and B and D might be processes on F2. Some people would advocate creating two secure sessions between F1 and F2, one for the A-B traffic and one for the C-D traffic, and using different keys.

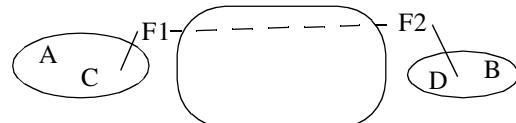


Figure 17-1. Multiplexing A-B and C-D Traffic over One Secure Session

Creating multiple secure sessions is obviously more expensive in terms of state and computation. In §17.5.1 *The Splicing Attack*, §17.5.2 *Service Classes*, and §17.5.3 *Different Cryptographic Algorithms*, we explain some cases where it is advantageous to create multiple secure sessions.

17.5.1 The Splicing Attack

In a splicing attack, an attacker who can see multiple encrypted conversations replaces ciphertext blocks from one encrypted conversation with ciphertext blocks from a different encryption conversation. This sort of attack is foiled if the secure session is integrity protected as well as encrypted, but let's assume the F1-F2 secure session in Figure 17-1 is only doing encryption and assume F1 is multiplexing A-B traffic and C-D traffic over a single secure session to F2.

Suppose C is capable of eavesdropping on the F1-F2 link, as well as injecting packets. It is possible for C to do a splicing attack and see the decrypted data for the A-B conversation. Assume that the beginning of the plaintext, say, the first sixteen octets, identifies the conversation to F2, most likely by specifying the source and destination. The splicing attack involves the following steps:

- Record an encrypted packet from the C-D conversation.
- Record an encrypted packet from the A-B conversation.
- Overwrite the first sixteen octets of ciphertext from the C-D packet onto the first sixteen octets of the encrypted A-B packet.
- Inject the spliced packet into the ciphertext stream.

When F2 decrypts the packet, it will observe from the first sixteen octets that the packet should be delivered to D. The remainder of the packet is the data from the A-B conversation. If it were encrypted in ECB mode, all the data from the A-B plaintext packet will be delivered to D. But if it were encrypted in, say, CBC mode, the first block of data will be garbled, but the remainder will be the plaintext from the A-B conversation.

Note that this flaw is only relevant if the F1-F2 link does encryption only. If it is using integrity protection, this attack is not possible. However, since people found the flaw with encryption only, some people think it would be safer to use separate secure sessions between F1 and F2 for each conversation, just in case a similar flaw is found when integrity is also used.

Even if there is no security flaw due to multiplexing traffic from different conversations over an encrypted and integrity protected tunnel, if F1 is a machine at an ISP (Internet Service Provider) serving multiple customers, the customers often feel safer if their traffic is carried over its own SA. Since the ISP needs the customers, it is advantageous for it to make the customers feel safer.

17.5.2 Service Classes

Suppose some types of traffic being forwarded between F1 and F2 get expedited service or some different routing that would make it likely for packets transmitted by F1 to get very much out of order. If F2 is protecting against replays based on a sequence number in the packet, a common implementation to detect replays is for F2 to remember the highest sequence number seen so far, say, n , and remember which sequence numbers in the range $n-k$ to $n-1$ have already been seen. If F1 marks the packets with different classes of service, then packets with a lower priority might take a lot longer to arrive at F2. If more than k high-priority packets launched by F1 can arrive before a lower priority packet launched earlier by F1, then the low-priority packet will be discarded by F2 as out of the sequence number window.

For this reason, some people advocate creating a different secure session for each class of service, so that each service class has sequence numbers from a different space.

17.5.3 Different Cryptographic Algorithms

Some people advocate different secure sessions for different flows, because each flow might require a different level of security. Some flows might require integrity only (without encryption). Some encrypted traffic might require more security, and thus a longer key, than other traffic. One might think this could be solved by using the highest level of security for all traffic. But the higher security encryption might be a performance problem if it was used for all the traffic.

Another reason to use different cryptographic algorithms is if some of the flows are for customers who would, for their own reasons, like to use particular cryptographic algorithms. It might

be vanity crypto developed by their own company or country, they might feel safer using different algorithms than what others use because they might have heard rumors about potential weaknesses, or there might be legal reasons why particular algorithms might only be usable for traffic of certain customers.

17.6 USING DIFFERENT SECRET KEYS

It is good practice to use different secret keys for establishing a secure session, as well as for protecting data during the session.

17.6.1 For Initiator and Responder in Handshake

Often a secure session between Alice and Bob is established using a shared key K_{A-B} , but one must be somewhat careful. Using different keys for the initiator than the responder in an authentication handshake avoids attacks such as in §11.2.1 *Reflection Attack*. There are other ways of avoiding reflection attacks, for instance:

- Have the initiator generate odd challenges and the responder even challenges.
- Have the response to the challenge consist of a function of the challenged side's name in addition to the key and challenge, e.g., $\text{hash}(\text{name}, \text{key}, \text{challenge})$ or $\{\text{name}|\text{challenge}\}\text{key}$.

17.6.2 For Encryption and Integrity

With CBC residue (see §4.3.1 *CBC-MAC*) as an integrity check, there are problems when the same key is used for encryption as well as integrity protection, as described in section §4.4 *Ensuring Privacy and Integrity Together*. But if the integrity protection were, say, a keyed hash, then there are no known weaknesses of using the same key. However, people are worried that since there was a problem with using the same key for both purposes with CBC residue, perhaps someone will later find a weakness with other schemes, and using two different keys avoids the issue.

Another reason using two keys became popular in protocols was because of U.S. export laws. The allowed keysize for exportable encryption was 40 bits, which was definitely very weak. But the U.S. government did allow stronger integrity checks. They only wanted to read the data, not tamper with it, so it was okay with them if the integrity check were reasonably strong. As a result, protocols often wound up with two keys: a 40-bit one for encryption and one of adequate size for integrity protection. But today, strong encryption keys are allowed.

Today, there are cryptographic modes (*e.g.*, GCM and CCM) that are secure without requiring two different keys.

17.6.3 In Each Direction of a Secure Session

If different keys are used for the two directions of a secure session, it prevents an attacker from reflecting traffic to one of the parties to get them to interpret the message as if it came from the other side.

17.7 USING DIFFERENT PUBLIC KEYS

People often refer to “Bob’s public key”, but having Bob use a single public key can be problematic.

17.7.1 Use Different Keys for Different Purposes

If the same key is used for, say, decrypting a challenge in an authentication protocol, and for decrypting headers of encrypted messages, it is possible for an attacker (*say*, Trudy) to trick Bob into decrypting something that will give Trudy information or signing something that will allow Trudy to impersonate Bob. An extreme example is a blind signature (see §16.2). By definition, with a blind signature, Bob has no idea what he’s signing. So that key had better not be used for any other purposes.

But even when the application isn’t purposely designed to prevent Bob from knowing what he’s doing, it is very likely that one application might not recognize an action that would have meaning in another application. For example, if the private decryption key is used to decrypt a challenge in a challenge/response protocol, the “challenge” given could be an encrypted data encryption key extracted from an email header.

One method of preventing such cross-application confusion is to encode something application-specific in the padding of the information to be signed or encrypted using public key cryptography. So in a challenge/response protocol (*let’s call the protocol QXV*) in which Bob would be asked to sign the challenge, the challenge could be a 128-bit number, and Bob would pad the challenge, before signing, with an application-specific constant such as a hash of the string **authentication protocol QXV challenge-response**. If all applications using that same key were carefully coordinated, no confusion would arise. However, if any application didn’t insist on those

rules, there could be vulnerabilities. Also, if any application were using blind signatures or blind decryption, the padding would not be visible to the private key owner (Bob). Therefore it would be safest, in theory, to certify a key as only applicable for a given application. The downside of this is that maintenance of so many different keys could present its own vulnerabilities as well as performance issues.

17.7.2 Different Keys for Signing and Encryption

There are are different consequences for forgetting a key and for having a key stolen. The consequences for a signing key are different from the consequences for an encryption key. This argues that the same key should not be used for both purposes, so that different mechanisms can be used for storage and retrieval.

If a signature private key is forgotten, it's not a big deal. You can just generate a new one. But if an encryption private key is lost, you'd lose all the data encrypted with it. Therefore, it is common for Alice to make sure she doesn't lose her private encryption key by keeping extra copies of the key in lots of different places. But this makes the key easier to steal, a trade-off considered acceptable in some cases for an encryption key, given the high cost of losing all the encrypted data.

Note that if the signature key is stolen, the obvious solution is to revoke it. But that would cause all previous signatures to become invalid, even those that were validly signed before the signature key was stolen. You might think you could avoid invalidating all signatures by including a timestamp in the revocation saying "any signatures earlier than this date are still valid". However, the attacker who stole the signature key can put any date he wants into the signature, so specifying a date in the revocation will not help. So, it's dangerous to make copies of a signature private key, because that makes it more likely to be stolen.

If a signature key is only used for authentication, it is much simpler, since it isn't important to maintain its validity for previous authentications. But if the signature key is used for signing documents, you want to avoid invalidating all signatures.

Another reason for separate keys for encryption and signatures is for law enforcement or for a company that wants to be able to decrypt anything encrypted for any of its employees. They would like to make Alice's private encryption key readily accessible to the IT department (or the government). But they have no need to be able to forge Alice's signature, so they do not need to have a copy of Alice's signature private key. In fact, it's to their advantage not to have access to the signature key. Then Alice cannot claim that messages signed by her key were an attempt by law enforcement to frame her. (Though she could still claim that an attacker or malware used her key without her knowledge.)

Yet another reason is that you'd like to treat old expired encryption keys differently than old signature keys. There's no reason to archive old private signature keys. Once the key is changed, just throw away the old one. Irretrievably discarding a private signature key once it isn't supposed

to be used anymore safeguards against someone stealing the old key and backdating a document. But old encryption keys are likely to still be needed to decrypt old data.

17.8 ESTABLISHING SESSION KEYS

17.8.1 Have Both Sides Contribute to the Master Key

It is considered good cryptographic form for both sides of a communication to contribute to a key. For instance, consider the protocol in which Alice and Bob each choose a random number, send it to the other side encrypted with the other side's public key, and use a hash of the two values as the shared secret. Although this protocol does not have PFS, it has "better" forward secrecy than something like SSL (where one side sends the secret encrypted with the other side's public key). By having both sides contribute to the key and hashing the two values, an attacker would need to learn *both* side's private keys to decrypt the recorded conversation. Another reason for having both sides contribute to the master key is that if *either* side has a good random number, the result will be a good random number.

17.8.2 Don't Let One Side Determine the Key

In addition to ensuring that each side contributes to the key, folklore says it is a good idea to ensure that neither side can force the key to be any particular value. For instance, if each side sends a random number encrypted with the other side's public key, if the shared key were the \oplus of the two values, then it would be easy for whichever side sent the last value to ensure that the result would be a particular value. If instead, the shared key were a cryptographic hash of the two values, this would be impossible.

Why is this an important property? In most cases it wouldn't matter. However, here is a subtle example where it would matter. Suppose the application is doing an anonymous Diffie-Hellman exchange to establish a session key, because there are no credentials for Alice or Bob at that protocol layer. However, because an upper layer protocol has the ability to authenticate, Alice and Bob can use channel binding (§11.6 *Detecting MITM*), using the session key agreed-upon by the lower layer. If Trudy were acting as a MITM and could force the Alice-Trudy key to be the same as the Trudy-Bob key, then channel binding would not detect the MITM.

17.9 HASH IN A CONSTANT WHEN HASHING A PASSWORD

Users tend to use the same password in multiple places. Consider a protocol in which a secret key is derived from the user's password and used in a challenge-response protocol (for example, see Protocol 11-1). In such a protocol, the server will contain a database of each user's hashed password, and the user's hashed password can be used to directly impersonate the user. Note we are not talking about a protocol in which the client establishes a TLS session to Bob, Alice sends her password over the secure session, and Bob hashes Alice's password and compares it against Bob's database of user passwords. We are specifically referring to a protocol in which the client converts Alice's password into a shared secret key with Bob and uses that shared secret key in a challenge-response protocol.

To prevent theft of one server's database from allowing someone to impersonate users using the same password on a different server, have the server hash a constant into the password hash, such as the server name. In that way, If Alice used the password `albacruft` on both server X and server Y, her hashed password stored at X would be different from what would be stored at Y, because at X, the value "X" would be hashed with her password, and at Y, the value "Y" would be hashed with her password. If someone were to steal server X's database, they could impersonate Alice on server X. If they did a dictionary attack on X's stolen database and discovered Alice's actual password, they could impersonate her on server Y. However, if Alice's password were good enough to withstand a dictionary attack, X's stolen database would not allow the attacker to impersonate Alice on server Y.

This strategy, of course, requires the client machine to hash in the same constant when creating the user's shared secret with that server. Since Alice has requested to talk to X, the client will know what constant to hash with the password.

This strategy solves the problem of Alice using the same password on multiple servers. It should be used in addition to approaches like using salt (§9.8 *Off-Line Password Guessing*) or making the hashing computation artificially expensive (*e.g.*, hashing the user's password many times), which are both solutions to a different problem, namely making off-line password guessing more computationally expensive for an attacker.

17.10 HMAC RATHER THAN SIMPLE KEYED HASH

Folklore today says that in order to do a keyed cryptographic hash, you should use HMAC (see §5.4.11). It is computationally slower than, say, doing $\text{hash}(\text{message}|\text{key})$. But it is possible, as we explain in §5.4.10 *Computing a MAC with a Hash*, to do the keyed hash incorrectly. With some

hash algorithms, if you do $\text{hash}(\text{key}|\text{message})$ and divulge the entire hash, then it is possible for someone who sees the message and the entire keyed hash to append to the message and generate a new keyed hash, even without knowing the key with which the hash is produced. With HMAC, this attack is impossible. There are other ways of avoiding that attack that are simpler and faster and likely to be as secure. However, HMAC comes with a proof, so it is popular despite its (minor) performance disadvantage. Today there are encryption modes that include integrity protection such as AES-GCM that are also provably secure, and more efficient, and these are gradually replacing HMAC.

17.11 KEY DERIVATION

Key derivation is the technique of using a small number of random bits as a seed and from it deriving lots of bits of keys. For instance, from a 128-bit random seed, you might want to derive 128-bit encryption keys and 128-bit integrity keys in each direction. Or you might want to periodically do key rollover (without PFS) lots of times, using the same seed.

The alternative to key derivation is obtaining independent random numbers for each key. It might be expensive to obtain that many random bits. For instance, if the random number is sent encrypted with the other side's public RSA key, it is convenient to limit the size of the random number to fit within an RSA block. And if an unpredictably many future keys also need to be derived from the same keying material, you wouldn't know how large a random number you'd need.

If the random number is sufficiently large (say, 128 bits or more), then it can be used as the seed for an unlimited number of keys. It is important to do this in such a way that divulging one key does not compromise other keys. So each key is usually derived from the random number seed and some other information (such as a key version number or timestamp), using a one-way function.

The random number from which all other keys are derived is often known as the **master key** or **key seed**.

In many protocols, creating and/or communicating the initial secret is expensive, because it involves private key operations. Reusing the original secret for generating new keys will be less expensive than creating and sending a new random seed.

In the cryptography community, companies with outrageous claims and bogus security products are known as “snake-oil” companies. A common claim made by the snake-oil marketeers is that the company's algorithm uses keys of some huge size, perhaps millions of bits, and therefore their product is much more secure than, say, the 128-bit keys used by other companies. (And they usually have a “patented”, proprietary encryption scheme.) There are many aspects of such a statement that should raise the suspicion that the claims are snake oil. Often the million-bit key is gener-

ated from a very small seed, perhaps 32 bits, which, through key derivation, turns into many bits. So even though the key used might be a million bits, if the seed from which it is computed is 32 bits, then the key itself is equivalent to a 32-bit key.

17.12 USE OF NONCES IN PROTOCOLS

Nonces are values that should be unique to a particular running of the protocol. Protocols use nonces as challenges (Bob gives Alice a nonce as a challenge, and she proves she knows her secret by returning a function of the challenge and her secret), or perhaps as one of the inputs into the session key derived from the exchange. Some protocols would be insecure if the nonces were predictable, whereas other protocols only require that the nonce be unique. If you don't want to bother analyzing whether your protocol requires unpredictable nonces, it is usually safest to generate them randomly.

See §11.4 *Nonce Types* for examples of both types of protocol (those that require unpredictable nonces and those that do not). It has become fashionable to put lots of nonces into protocols, and furthermore, to require them all to be randomly chosen. However, in some cases, such as when the nonce must not be reused, it is more secure to use a sequence number, because it requires a lot less state to remember all the sequence numbers used recently, since the application can delete messages with sequence numbers that are too old. If the nonce were randomly chosen, the application would need to remember all nonces ever used.

17.13 CREATING AN UNPREDICTABLE NONCE

If it's inconvenient to obtain large random numbers, a unique nonce such as a sequence number can be turned into an unpredictable nonce by hashing it with a secret known only by the sender.

17.14 COMPRESSION

If compression is desired for encrypted data, compression must occur before the data is encrypted. This is because compression algorithms depend on the data being somewhat predictable. With any secure encryption algorithm, ciphertext looks random and therefore would not compress.

Obviously, compression saves bandwidth for transmitted data and saves storage for stored data. This advantage might be offset by the disadvantage of additional computation to compress before encryption and decompress after decryption. On the other hand, compression might provide a computation benefit because there are fewer octets to encrypt/decrypt after the data has been compressed. On the third hand (we_{1,2,3,4} have eight hands between us), there is sometimes hardware support for the encryption algorithm but not for the compression algorithm.

In olden times when encryption algorithms were weaker, the folklore belief was that compression made things more secure. The assumption was that when an attacker did brute-force search of the keyspace, it would be harder to recognize valid compressed plaintext than valid uncompressed plaintext. However, often compressed data starts with a fixed easily recognizable header, so it would be likely easier for a brute-force attacker to recognize valid compressed data.

Recently, though, cryptographers have been recommending against doing compression. That might seem surprising. What do they have against saving bandwidth and storage? The reason that compression has fallen out of favor among cryptographers is that in many cases, compression leaks information about the plaintext. For example, in a block storage system, the plaintext is chopped into fixed-size blocks. With compression and encryption, although an attacker would not be able to decrypt an encrypted block, the attacker would be able to see the size of the ciphertext block. The compression algorithm would be well-known, so based on the size of the resulting ciphertext block, the attacker will be able to know which candidate plaintext blocks would compress to that size.

Another example where compression leaks information was published in [WRIG08]. The natural way of encoding audio is to break it into fixed time-unit chunks and compress each chunk. If encryption is used, then each compressed chunk is encrypted with a length-preserving encryption mode. The paper shows that given the size of each compressed encrypted chunk, an attacker can learn information about the plaintext, such as what language was spoken and where phrases of interest were said.

Another attack is known as CRIME (Compression Ratio Info-leak Made Easy) [en.wikipedia.org/wiki/CRIME], which is a vulnerability relevant to HTTP and cookies. As a result of this and other vulnerabilities due to compression, support for compression was removed in TLS 1.3. Since very few implementations of TLS had implemented compression, removing support for compression was not controversial.

Compression is only problematic when it is performed on chunks whose compressed lengths leak more information than knowing lengths of the chunks of plaintext would. There is nothing

wrong with compressing large volumes of information before encrypting and sending fixed size chunks of the resulting compressed data.

17.15 MINIMAL VS. REDUNDANT DESIGNS

Some protocols are designed so minimally that everything in them is essential. If you cut corners anywhere and don't follow the spec carefully, you will wind up with vulnerabilities. In contrast, it has become fashionable to over-engineer protocols, in the sense of accomplishing what is needed for security several different ways, when in fact $n-1$ of those ways are unnecessary as long as any one of them are done. An example is throwing in lots of nonces and other variables, each required to be random, whereas for security only one of them would be required to be random. We believe this fashion of redundancy is a dangerous trend, since the protocols become much more difficult to understand. And as the person writing the code for choosing the nonce notices that there is no reason for the nonce to be random, and therefore doesn't bother, and the person writing the code for a different portion of the protocol notices that their nonce doesn't need to be random, you wind up likely to have a protocol with flaws. It's okay to cut $n-1$ corners, but not n corners.

17.16 OVERESTIMATE THE SIZE OF KEY

U.S. law enforcement was always trying to find the “right” keysize, sufficiently large to be “secure enough” but sufficiently small that it would be breakable, if necessary, by law enforcement. Of course, such a keysize couldn't be really secure, because “secure” means nobody, including them, would be able to break it. Anyway, the assumption was that they were smarter or had more computation power than the bad guys from whom you were trying to protect yourself. Even if that were true (*e.g.*, that law enforcement had ten times as much computation power as organized crime), given that computers keep getting faster, and encrypted data often needs to be protected for many years, a system breakable by law enforcement today would be breakable by organized crime in a few years. But it's also absurd to assume that the bad guys (whoever they are) would have significantly less computation power at their disposal than law enforcement. So it's dangerous to try to pick the smallest “sufficiently secure” keysize.

17.17 HARDWARE RANDOM NUMBER GENERATORS

Generating random numbers can be tricky. Typically, computers are designed to be deterministic, and it isn't obvious how to cause software to choose a different number each time a random number is required. Various software-only strategies include recording timing of user keystrokes or message arrivals on a network, or statistics on attached devices. There have been many examples of implementations that were insecure because of faulty random number generators. As a result, many people wish that computers would include a source of true randomness, which can be done at the hardware level by measuring noise of some sort. Indeed, it would be convenient and enhance security to have a hardware source of random numbers. However, instead of depending on it as the sole source of randomness, it should be used to enhance other forms.

Suppose an organization was designing a hardware random number generator that was likely to become widely deployed. It would be very tempting to purposely (and secretly) design it so that it generated predictable numbers, but only predictable to the organization that designed the chip. It would be difficult or impossible for anyone to detect that this was done. For instance, each chip could be initialized with a true random 128-bit number as the first 128 bits of output, and each subsequent 128 bits of output could be computed as the previous 128 bits of output hashed with a large secret known only to the organization that designed and built the chips. Assuming the hash was good, the output would be indistinguishable from truly random. It would be difficult to embed such a trap in software, since it is more easily reverse engineered.

Even if a hardware random number generator might have such a trap, it is still useful, and can be used in such a way that would foil even the organization that embedded the trap. The way it should be used is to enhance any other scheme for generating random numbers. So random numbers should be generated as they would be without the chip (for instance, using mouse and keyboard inputs and low-order bits of a fine-granularity clock). But the output of the hardware random number generator should also be hashed into the random number calculation.

17.18 PUT CHECKSUMS AT THE END OF DATA

Some protocols (for example, §12.5 *AH (Authentication Header)*) use a format for integrity-protected data where the integrity check is before the data, or possibly even in the middle of the data. It is better instead to have the integrity check at the end [for example, §12.6 *ESP (Encapsulating Security Payload)*]. If the integrity check is at the end, then it is possible to be computing it while transmitting the data and then just append the computed integrity check at the end. The alternative, where the integrity check comes before the data, requires the data to be buffered on output

until the integrity check is computed and tacked onto the beginning. This adds delay and complexity. This rule is mainly for transmitted data. For received data, most likely it would be undesirable to act on the data before the integrity check is verified, so the data would need to be buffered no matter where the integrity check was stored.

Having the integrity check in the middle of the data (as is done in AH) complicates both the specification and the implementation. Space for the integrity check must be provided, with the field set to zero, and then the integrity check calculated and stored into the field. On receipt, the integrity check must be copied to some other location, and then zeroed in its location in the message, so that the integrity check can be verified.

Especially complex (again, as is done in AH) is when an integrity check is computed based on selected fields, especially when some are optional and ordering is not strictly specified. The integrity check must be computed identically by the generator and verifier of the check.

17.19 FORWARD COMPATIBILITY

History shows that protocols evolve. It is important to design a protocol in such a way that new capabilities can be added. This is a desirable property of any sort of protocol, not just security protocols. There are some special security considerations in evolving protocols, such as preventing an active attacker from tricking two parties into using an older, possibly less secure version of the protocol (see §11.15.2 *Downgrade Attack*).

17.19.1 Options

It is useful to allow new fields to be added to messages in future versions of the protocol. Sometimes these fields can be ignored by implementations that don't support them. Sometimes a packet with an unsupported option should be dropped. In order to make it possible for an implementation that does not recognize an option to skip over it and parse the rest of the message, it is essential that there be some way to know where the option ends. There are two techniques:

- Having a special marker at the end of the option. This tends to be computation-intensive, since the implementation must read all the option data as it searches for the end marker.
- TLV encoding, which means each option starts with a TYPE field indicating the type of option, a LENGTH field indicating the length of the data in this option, and a VALUE field, which gives option-specific information.

TLV encoding is more common because it is more efficient. However, the “L” must always be present, and in the same units, in order for implementations to be able to skip unknown options. Sometimes protocol designers who don’t quite understand the concept of TLV encoding do clever things like notice that the option they are defining is fixed length, so they don’t need the LENGTH field. Or that one option might be expressed in different units. For instance, although AH (see §12.5 *AH (Authentication Header)*) is designed to look like an IPv6 extension header, its length is expressed in units of 32-bit words, when all the other IPv6 options are expressed in units of 64-bit words.

It is also useful to be able to add some options that should simply be skipped over by an implementation that does not support it and to add other options that must be understood or else the packet must be dropped. But if an implementation does not recognize the option, how would it know whether it could be safely ignored or not? There are several possible solutions. One is to have a flag in the option header known as the **critical bit**, which if not set on an unknown option, indicates the option can simply be skipped over and ignored. Another possibility is to reserve some of the type numbers for critical options and some of them for noncritical options (those that can be safely skipped if unrecognized).

17.19.2 Version Numbers

A lot of protocols have a field for version number but don’t specify what to do with it other than to specify what the implementation should write into that field. The purpose of a version number field is to allow the protocol to change in the future without confusing old implementations. One way of doing this without a version number is to declare the modified protocol to be a “new protocol”, which would then need a different multiplexor value (a different TCP port or Ethertype, for instance). With a version number, you can keep the same multiplexor value, but there have to be rules about handling version numbers so that an old implementation won’t be confused by a redesigned packet format.

17.19.2.1 Version Number Field Must Not Move

If versions are to be differentiated based on a version number field, then *the version number field must always be in the same place in the message*. Although this might seem obvious, when SSL was redesigned to be version 3, the version number field was moved! Luckily, there is a way to recognize which version an SSL message is (in version 2’s client hello message, the first octet will be 128, and in version 3’s client hello message, the first octet will be something between 20 and 23).

17.19.2.2 Negotiating Highest Version Supported

Typically, when there is a new version of a protocol, the new implementations support both versions for some time. If you support both versions, how do you know what version to speak when talking to another node? Presumably the newer version is superior for some reason. So you typically first attempt to talk with the newer version, and if that fails, you attempt again with the older version. With this strategy, it is important to make sure that two nodes that are both capable of speaking the new version wind up speaking the newest version they both support and not getting fooled into speaking the older version because of lost messages or active attackers sending, deleting, or modifying messages. Why would an active attacker care enough to trick two nodes into speaking an earlier version of the protocol? Perhaps the newer version is more secure or has features that the attacker would prefer the nodes not be able to use. (We would hope there was *some* benefit to be gained by having designed a new version of the protocol!)

The right thing to do if you see a message with a higher version number than you support is to drop the message and send an error report to the other side indicating you don't support that version. But there will be no way for that error message to be cryptographically integrity protected since the protocol has not been able to negotiate a key. So unless care is taken, nodes could be tricked into speaking the older version if an active attacker or network flakiness deleted the message of the initial attempt or an attacker sent an `error: unsupported version number` message.

One method of ensuring that two nodes don't get tricked into talking an older version is to have two version numbers in the packet. One would be the version number of the packet. The other would be the highest version the sender supports. But a single bit suffices, indicating that the sender can support a higher version number than the message indicates. If you establish a connection with someone, using version n , and you support something higher than n , and you receive authenticated messages with the HIGHER VERSION NUMBER SUPPORTED flag, then you can attempt to reconnect with a higher version number.

17.19.2.3 Minor Version Number Field

Another area in which protocol designers get confused is the proper use of a MINOR VERSION NUMBER field. Why should there be both MAJOR VERSION NUMBER and MINOR VERSION NUMBER fields? The proper use of a minor version number is to indicate new capabilities that are backward compatible. The major version number should change if the protocol is incompatible. The minor version number is informational only. If the node you are talking to indicates it is version 4.7 (where 4 is the major version number and 7 is the minor version), and you are version 4.3, then you ignore the minor version number. But the version 4.7 node might use it to know that there are certain fields you wouldn't support, so it won't send them.

Most likely the confusion about the proper use of the minor version number is because software releases have major and minor version numbers, and the choice as to which to increment is a marketing decision.

