6.1600 STAFF

6.1600 FOUNDATIONS OF COMPUTER SECURITY

0.1 Disclaimer

This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

0.2 Contributors

These notes are based on lectures by the 6.1600 course staff:

- **2024:** Srini Devadas and Yael Kalai
- **2023:** Henry Corrigan-Gibbs and Nickolai Zeldovich
- **2022:** Henry Corrigan-Gibbs, Nickolai Zeldovich, and Yael Kalai
- **2021:** Henry Corrigan-Gibbs, Nickolai Zeldovich, Srini Devadas, and Yael Kalai

Ben Kettle, our course TA in 2022, was responsible for transcribing the first set of these lecture notes in Fall 2022.

Contents

1
What is Security?

1.1 Overview

The goal of this course is to give you an overview of the most important "Big Ideas" on securing computer systems. Throughout the course, we will touch on ideas from the fields of computer security, cryptography, and (to some extent) computer systems.

1.2 What is Security?

Security is a very broad property, but generally the goal of computer security is to ensure that a particular computer system is *behaves correctly* even in the face of an *adversary* (or *attacker*) whose goal is to foil the system.

We will use the terms "adversary" and "attacker" interchangeably throughout this course.

To achieve this goal, we will need some kind of systematic plan. That is, we will have to carefully define what it means for our system to *behave correctly* and we will have to specify the class of *adversaries* against which we want to defend.

For the purposes of this course, we will typically structure our plan in terms of three components: a *model* of the system and the adversary, a security *goal*, an *implementation*.

- **Model:** The system model specifies:

  1. what the system is that we are trying to defend,
  2. what the attacker is, and
  3. how the system and attacker interact.

  For example, in network security we might think of the system and the attacker as being two computers that interact over a network. In this model, the attacker can send network packets to the victim system. (The attacker in this model cannot, for example, swap out the hard drive of the victim system.)

Sometimes people call this the "threat model." For the purposes of this chapter, think of the model as describing *how* the attacker and system interact and the security goal as defining *what* our implementation is trying to achieve.

When we are studying hardware security, we might consider a different model: The system we are trying to defend is a CPU, and the adversary is an external device that can read all of the contents of the victim system's RAM. (The attacker in this model cannot, for example, read the internal state of the victim's CPU.)

When we are studying ATM security, the system we are trying to defend is an ATM machine. For an attacker, we might consider a person interacting with the ATM through it's normal interface: the attacker can insert a malformed ATM card into the machine and type arbitrary PINs into the device. (The attacker in this model cannot, for example, take a jackhammer to the ATM to extract the cash.)

Modeling the adversary and the system in question is almost always the first step of thinking about system security. While we often only have an informal system model in mind, the more precise you—as a system designer—can be about your system model, the clearer your security properties can be.

- **Goal:** The security goal defines what we want our system to achieve in our specified model.

  For example, in network security, we might want the property that "only someone knowing Alice's secret password can execute shell commands on the machine." In hardware security, we might want the property that "the attacker learns nothing about the data stored in RAM, apart from its size." In ATM security, we might want the property that "an attacker can fraudulently authenticate as a victim with probability at most 1/10000."

  As you will learn throughout the course, figuring out exactly what your security goal should be is often quite subtle and challenging.

- **Implementation:** The implementation is how we achieve the goal. For example, in securing a computer system on a network, we might use password-based authentication to protect access to a computer system.

Together, the model and the goal model create our *definition* of security. As such, the model and goal cannot be "wrong", but it may be that the threat model might not have captured all of the attacks that a real-world adversary can mount, and goal might turn out to not be exactly what we needed. Often, the process of designing the model and goal is iterative: when the system designer discovers a surprising gap in the security goal (we will see some shortly), she patches the goal and the implementation accordingly.

When you read about security failures in the news, it is worth trying to understand whether the failure arose from a problem with the model, the goal, or the implementation.

The implementation, on the other hand, can definitely be wrong—if the implementation does not guarantee the goal under the model

in question, due to bugs or oversights or supply chain vulnerabilities or anything else, the implementation has a mistake.

The implementation is always public: Kerckhoff's principle. Throughout this course, we will always assume that the attacker knows the implementation of whatever security mechanism we are using. The only thing that we keep secret from the adversary are the system's secret keys. This is known as *Kerckhoff's principle*. The logic behind this way of thinking is that: (1) it is often relatively easy for the attacker to learn bits of information about the system design and (2) it is much easier to replace a set of cryptographic keys (if the attacker learns them) than to redesign the entire system from scratch.

1.2.1 *Security is Hard*

Building secure systems is challenging. There are at least two broad reasons for this.

Secure systems must defend against worst-case behavior. First, a secure system must defend against *all possible attacks* within the scope of the system model. In contrast, when we are just concerned about functionality or correctness, we are often satisfied with a system that performs well for the cases that users care about. In other words, security is concerned with behavior in *worst-case situations*, while correctness is often about behavior in *expected situations* (i.e., *average-case situations*).

Some developers do worry about correctness of their system for all possible inputs and corner cases; this is is indeed the mindset that is often necessary for security.

For example, suppose that you are a car manufacturer and you want to test a car stereo functions correctly. Testing that the stereo works well *on average* (i.e., in expected situations) is easy: turn the stereo on and off 10,000 times, try playing some music through it, and accept it as working if all of these checks pass. Testing that the stereo works well *in the worst case* is not as easy: it is possible that if someone connects an adversarial USB device that sends some specially crafted malicious packets to the stereo, they can hijack the car and cause it to explode. But you will never find these malicious packets by random testing—only by careful inspection. A secure system must defend against all possible attacks within the threat model; being certain that a system satisfies this strong security property is a challenge.

These attacks are actually possible in practice! See: Karl Koscher et al. "Experimental security analysis of a modern automobile". In: *IEEE Symposium on Security and Privacy*. 2010

An implementation can never defend against all possible threats. When we specify a system model, we delimit the set of adversaries against which our implementation must defend. But real-world adversaries can behave in ways that are outside of our model and thereby violate

our security goals.

For example, someone besides a TA might be able to access the grades file for our class by:

- finding a bug in the server software,
- breaking into a TA's office,
- compromising the TA's laptop,
- stealing the password to an administrator's account,
- tricking a TA into disclosing grades,
- breaking the server's cryptography,
- getting a job at the registrar and making herself a TA.

And the list never ends. Because our threat model cannot capture all possible threats, **security is never perfect**. There will essentially always be *some* attacker than can break your system.

This is why we need a threat model: the threat model defines what kinds of attacks we worry about and which we decide are out of scope.

1.2.2 Designing security goals and a threat model

Specifying security goals and a threat model is all about comparing the cost of defending against an attack with the cost of that attack if it were to happen. It is almost always impractical to exactly calculate these costs, but this framework is useful conceptually. Cheap defenses that block major holes are likely to be worth implementing, but defending against an esoteric RF side channel that could leak unimportant information is likely not.

Building a threat model always requires iterating—you will not get it right on the first try. There is likely to be some type of attack that you didn't consider at first that ends up being important.

1.2.3 Designing an implementation

We will focus largely in this class on techniques that have a big payoff—methods of developing software and tools to use that eliminate entire classes of attacks (or make them much harder).

1.3 Examples

We give a handful of examples of security failures arising from poor choices of security goals or threat model.

1.3.1 Insufficient Attack Models

Assuming specific strategy: CAPTCHA. CAPTCHAs were designed to be expensive to solve via automation, but easy for a human to read. Indeed it might be expensive to build an optical-character recognition system for CAPTCHAs in general, but attackers who want to bypass CAPTCHAs do not do this. Instead, they set up computer centers in countries where the cost of labor is cheap. Attackers then pay people working in these centers to solve CAPTCHAs.[1] The result is that it costs some fraction of a cent to solve a CAPTCHA. The cost of solving a CAPTCHA is still non-zero, but the cost is much lower than the system designers may have intended.

[1] Marti Motoyama et al. "Re: CAPTCHAs—Understanding CAPTCHA-Solving Services in an Economic Context". In: *Proceedings of the 19th USENIX Security Symposium*. Washington, DC, Aug. 2010.

Assuming low limit on computational power: DES. There used to be an encryption standard called DES that had 2^{56} possible keys. At the time that it was designed, the U.S. government standards agencies asserted that it was secure against even powerful attackers, but today a modern computer can try all 2^{56} keys at only modest cost. (Academic researchers even at the time of DES's design understood that 56-bit keys were not large enough to prevent exhaustive cryptanalysis.[2])

For example, `https://crack.sh/` uses an array of FPGAs to provide a service that exhaustively checks all possible keys.

[2] Whitfield Diffie and Martin E. Hellman. "Exhaustive Cryptanalysis of the NBS Data Encryption Standard". In: *Computer* 6.10 (1977), pp. 74–84.

Because of the weakness of DES against modern computers, everyone using the standard had to upgrade their block ciphers. For example, MIT had to switch from using DES for authentication to newer block ciphers with longer keys, such as AES.

Assuming a secure out-of-band channel: Two-factor authentication (2FA) via SMS. Many 2FA systems use a text message for authentication, but an attacker then just needs to convince the clerk at the AT&T store to give them a new SIM card for your phone number.

These attacks are often called "SIM swapping" or "SIM hijacking".

Assuming a correct compiler: Xcode. iPhone apps are normally created and compiled on a developer's machine, sent to Apple's App Store, and sent to iPhones from there. iPhone apps are created using a tool called Xcode that is normally downloaded from Apple servers. However, Xcode is a big piece of software and for developers behind China's firewall, it was very slow to download. Someone within China set up a much faster mirror of Xcode, and lots of developers in China used the version of Xcode from this mirror. However, this mirror was not serving exactly Apple's version of Xcode—instead, it was serving a slightly modified version of Xcode that would inject some malicious code into every app that was compiled with it. This took a long time to detect.

This prescient paper from 1984 anticipated the XcodeGhost attack: Ken Thompson. "Reflections on trusting trust". In: *Communications of the ACM* 27.8 (1984)

The Wikipedia article on XcodeGhost, `https://en.wikipedia.org/wiki/XcodeGhost`, provides more details about this attack.

1.3.2 *Insufficient security goal*

Business-class airfare. An airline tried to add value to their business-class tickets by allowing ticket-holders to change the ticket (i.e., the departure date, origin, and destination) at any time with no fee. One customer realized that they could board the flight *then* change their ticket. The customer could then take an unlimited number of business-class flights for the price of one.

In this case, the airline's goal did not meet their real needs—perhaps they needed to add an additional goal of the form "every time someone takes a flight, we get paid."

Sarah Palin's Email. Sarah Palin had a Yahoo email account, and Yahoo used security questions for password reset—their goal may have been something like "no one can reset a user's password unless they know all of the answers to the user's security questions." (The security questions are typically things like "What is your mother's maiden name?") As it turned out, it was possible to find the answer to all of Palin's account-recovery security questions on the Internet.[3] Yahoo's implementation may have been perfect, but their goal did not provide any meaningful security for certain users.

[3] Kim Zetter. *Palin e-mail hacker says it was easy*. https://www.wired.com/2008/09/palin-e-mail-ha/. Sept. 2008.

Instruction Set Architecture (ISA) Specification. When defining ISAs for processors, computer architects thought it would be acceptable for each instruction to take a different number of cycles to execute. This had big benefits for performance and for compatibility, but as we'll talk about later in the semester, researchers have recently exploited this timing variability to perform sophisticated attacks on wide ranges of processors.[4] Even if a processor's implementation faithfully implements the specification, the specification itself allows for certain types of timing side-channel attacks.

[4] Mark D. Hill et al. "On the Spectre and Meltdown Processor Security Vulnerabilities". In: *IEEE Micro* 39.2 (2019), pp. 9–19.

Complicated access-control policies. A school in Fairfax, Virginia used an online course-management software with a somewhat complex access-control structure: each teacher is in charge of some class, each class has many students, and each student has many files. Teachers cannot access student's files, and there is also a superintendent that has access to all the files. Teachers are able to change their students' passwords, and are able to add students to their class. It turned out that a teacher could add the superintendent as a student, change the superintendent's password, and then access all files via the superintendent's account. While each of the access-control policies individually sounds reasonable, together they lead to a bad outcome.

Mat Honan's Gmail Account. A journalist for wired named Mat Honan had his Gmail account compromised via a clever attack.[5] Honan had a Gmail account. Gmail's reset-password feature avoided using security questions, and instead used a backup email account. The attacker triggered the reset-password feature, which sent a reset link to Honan's Apple email account.

[5] Mat Honan. *How Apple and Amazon Security Flaws Led to My Epic Hacking*. `https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/`. Aug. 2012.

The attacker then attempted to gain access to Honan's Apple email account. The attacker triggered the reset-password feature on the Apple account. Apple's reset-password feature, in turn, required Honan's address and the last four digits of the credit card number. The attacker was able to find Honan's address publicly, but could not easily find his credit-card number.

The attacker found this credit-card information through Honan's Amazon account. Amazon, which knew his credit card number, required a full credit-card in order to reset an account. However, Amazon allowed buying something for a certain account without logging in, so long as you provide a new credit card number. It also allowed *saving* this new credit card number to the user's Amazon account. So, the attacker made a purchase on Amazon with the attacker's credit card. Then, the attacker saved their own credit card number to Honan's Amazon account. Next, the attacker triggered Amazon's reset-password feature, and—using the credit-card number saved to the account—were able to reset Honan's Amazon password and access his Amazon account. The attacker was then able to see the last four digits of Honan's real credit card within his Amazon account, use that to reset his Apple mail account, and then use that to reset Honan's Gmail account.

Complex chains of systems like this can be very hard to reason about, but these interactions ultimately are security-critical.

1.3.3 Buggy implementations

Bugs, misconfigurations, and other mistakes are the most common cause of security issues. A rule of thumb to keep in mind is that every 1000 lines of code will have around one bug. This is a very rough estimate, but the basic idea is that more code will have more bugs. An effective strategy to reduce security vulnerabilities is to reduce the amount of code in your system.

Missing Checks: iCloud. Apple's iCloud performs many functions—email, calendar, storage, and Find my iPhone. Each of these had their own way of logging in, but across all of them a common goal was to limit the attacker's ability to guess a user's password. To do this, they added rate limiting to all the login interfaces, allowing something like

only 10 login attempts per hour—but they forgot the Find my iPhone login interface.[6] Because this authentication code was duplicated all over the place, there were many places to remember to add this rate limiting, but the attacker only needed one weak login interface to brute-force a password. In general, avoiding this repition will make it much easier to build a secure system.

[6] J. Trew. *'Find My iPhone' exploit may be to blame for celebrity photo hacks (update)*. https://www.engadget.com/2014-09-01-find-my-iphone-exploit.html. Sept. 2014.

Insecure Defaults. When you set up a new service, they almost always come with some defaults to make the setup simpler. Wi-Fi routers come with default passwords, AWS S3 buckets come with default permissions, and so on. These defaults can be convenient, but they are very important to security because many people will forget or neglect to change the default. Because of this, the default becomes the way that the system operates. In order to build a secure system, it is important that the default is secure.

1.4 What are the general principles for secure system design?

1.4.1 Threat Models and Goals

- Create simple, general goals.
- Avoid assumptions (such as "no one else will be able to get a user's SIM card") through better designs.
- Learn and iterate.
- Practice Defense in Depth: don't rely on one single defense for all your security—it is useful to use backup defenses to guard against bugs that will inevitably come up in one defense.

1.4.2 Implementation

- A simpler system will lead to fewer problems.
- Factor out the security-critical part into a small separate system or piece of code (for example, hardware security keys).
- Reuse well-designed code, such as well-tested crypto libraries.
- Understand and test the corner cases.

Part I

Authentication

2
Authenticating People

In this class, we will talk a lot about requests going to a computer system. And a lot of security comes down to looking at that request and deciding how to handle it. For this, it is crucial to know *who* issued the request. Then, we can decide whether the system should allow the request.

Typically, a computer system performs two steps before processing a request:

1. **Authenticate:** Identify the person or machine (the "*principal*") making the request.
2. **Authorize:** Decide if the principal is authorized to make the request.
3. **Audit:** Log some information about what requests your system authorized, so that you can identify malicious requests after the fact and/or clean up your system after attacks on the authentication system. (For example, a user who accidentally reveals their password to an attacker.)

This chapter focuses on authentication. We will first discuss the attack model and security goals. Then we will describe common implementations that aim to achieve these goals.

2.1 Authentication: Security goals

In the simplest model of authentication, we have a client and server—two machines communicating over a network.

Another common scenario has a human being authenticating to a computer: think about typing a PIN into a phone or a password into a computer terminal. We will discuss this setting more when we come to passwords.

All authentication schemes try to prevent an attacker from impersonating an honest user. To precisely define the security goal of an authentication system though, we have to specify the attacker's power: against which types of attack are we trying to defend?

Figure 2.1 sketches three types of attacks on authentication systems. In more detail these are, in increasing order of strength are:

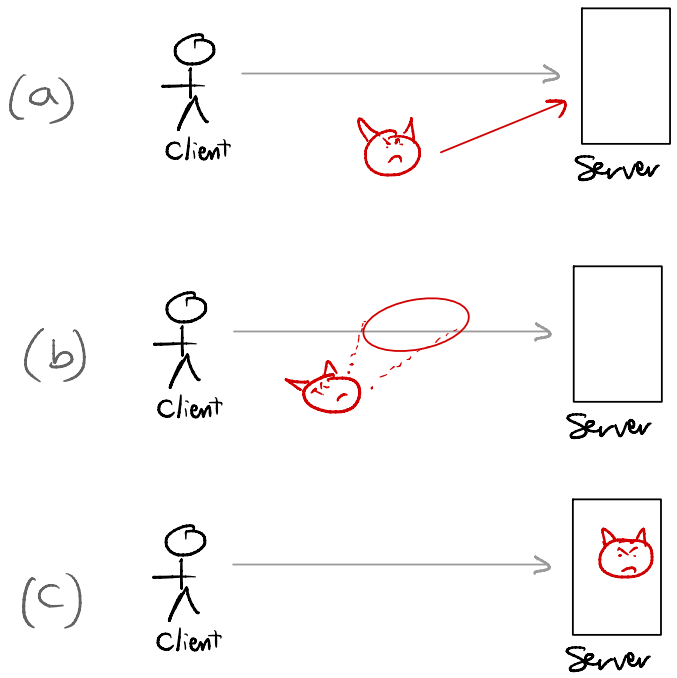


Figure 2.1: There are (at least) three interesting security goals for client-server authentication systems: (a) direct attack, (b) eavesdropping attack, and (c) active attack.

- *Direct attack.* The attacker never sees the target user authenticate and then tries to impersonate the honest user. PIN-based authentication systems, e.g., on your phone or on an ATM machine, often aim to defend only against direct attacks. The screen-lock password that protects your laptop is also an authentication system that just attempts to protect against direct attack.

  That is, these systems do not protect against an attacker that can look over your shoulder while you are typing your password. These systems only aim to provide security when the attacker *never* sees you (the honest user) authenticate.

- *Eavesdropping attack.* The attacker observes an honest user authenticating many times—i.e., the attacker sees all of the traffic between the client and server—and then tries to impersonate that user. One-time passwords, such as the six-digit authentication codes that the Google Authenticator app uses, aim to protect against eavesdropping attacks: an attacker who sees one of your one-time passwords will not be able to use it to authenticate as you; it is a *one-time* password.

- *Active attack.* The attacker that compromises the server, interacts with the honest user, and after the server is restored to a good

state, tries to authenticate as the honest user (*active attack*).

U2F security keys, and other schemes based on digital signatures (Chapter 5), aim to protect against active attacks.

Systems that defend against active attacks provide the strongest form of security, in that they also protect against eavesdropping attacks and direct attacks. Systems that defend against eavesdropping also defend against direct attacks. Systems that defend against direct attacks are the weakest—they do not necessarily provide any protection against the other types of attack.

2.2 *Protecting against direct attacks: Bearer tokens, PINs, and passwords*

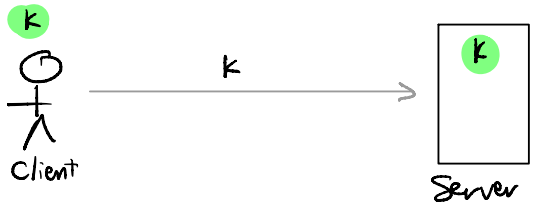


Figure 2.2: A bearer-token-based authentication scheme. To authenticate, the client sends its secret key k to the server. WARNING: In practice, the server should store the secret k under a slow hash function (see **??**).

The simplest form of authentication uses secret passwords or PINs. We sometimes call these secret values *bearer tokens*; whoever bears (holds) a user's token can authenticate as that user. Authentication with such schemes works as follows:

1. The server holds a password (or PIN or random token string) for the client.
2. To authenticate, the client sends their password to the server.
3. The server checks the password against its stored password and accepts if they match.

The benefit of password-based authentication schemes is that they are simple and easy to implement. In addition, a human can play the role of the client in a password-based authentication system (e.g., as you do when you type a PIN into your phone). Fancier authentication systems require the client to compute non-trivial cryptographic functions—not functions that normal humans can compute in their brains.

Bearer-token-based schemes do provide security against direct attacks: if an attacker has never seen the user authenticate, the attacker's best strategy is to just guess the user's password. Thus, the

security of these schemes against direct attacks depends entirely on the adversary's uncertainty about the password.

In some bearer-token-based systems, the server can assign a random password to each user. For example, when you create an account for certain web APIs, the API provider will give you a random secret key—a bearer token. You will have to include this secret key with each API request. Modern APIs use the stronger authentication mechanisms we describe later in this chapter.

In the vast majority of password- and PIN-based login systems, the user may choose their own password. This creates all sorts of headaches...

2.2.1 *What makes a good password?*

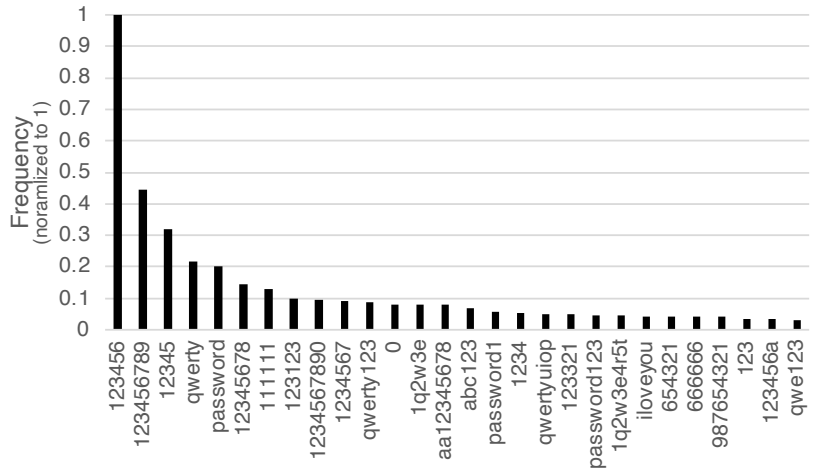


Figure 2.3: The most common passwords, according to NordPass (`https://nordpass.com/most-common-passwords-list/`), sorted by their frequency descending. A small number of common passwords dominate.

Rank	Password
1	123456
2	123456789
3	12345
4	qwerty
5	password
6	12345678
7	111111
8	123123
9	1234567890
10	1234567
11	qwerty123
12	000000
13	1q2w3e
14	aa12345678
15	abc123
16	password1
17	1234
18	qwertyuiop
19	123321
20	password123

Table 2.1: The most popular passwords in 2021, according to NordPass, `https://nordpass.com/most-common-passwords-list/`.

The security of a password-based authentication system rests entirely on the attacker's inability to guess the password in a small number of guesses.

Entropy is a way to quantify an adversary's uncertainty about a value sampled using a random process, or from a particular probability distribution. If a distribution has b bits of entropy, then it will take at least roughly 2^b guesses for an attacker to correctly guess a value sampled from this distribution.

The uniform distribution over 128-bit strings has 128 bits of entropy. The distribution from which humans typically choose their passwords has much less entropy—empirically, more like 20 bits.

Ideally, we would want all passwords to be equally as likely, from the adversary's perspective. (These would be "high-entropy" passwords.) If a system generated a truly random password for each user, each password would be indeed equally likely. But then it

becomes very difficult for a user to remember their password, much less many random passwords for many different services.

Typically, people have to remember their passwords, so we let them pick their own passwords. When they do, it turns out that many people are likely to choose the same password.

A typical password might be sampled from a distribution with roughly 20 bits of entropy—if an adversary is able to make 2^{20} guesses at the password, they can expect to guess the password correctly. One can computer easily make 2^{20} authentication requests to another in a few minutes.

2.2.2 *Dealing with weak passwords*

Every system that uses password- or PIN-based authentication must contend with the fact that most passwords are not that difficult for an attacker to guess.

In this section, we describe some mitigation strategies: all are flawed, but each is better than nothing. The goal of each is to make the attacker's job slightly more difficult; by stacking a few of these defenses on top of each other, we can substantially strengthen the end-to-end system.

Aggressively limit the number of guesses. Therefore, when using passwords as an authentication mechanism, an authentication system must *always* somehow limit the number of password guesses.

For example, some phones allow 10 guesses at the screen-lock PIN before the device resets itself. Limiting the number of guesses effectively prevents a *single* account from being compromised—provided that the password is not too too weak. One downside is guess limits create the possibility for denial-of-service attacks: an attacker can potentially make 10 guesses at your password and lock you out of your phone or online-banking account.

In addition, in many physical computer systems have multiple authorized users, each with their own password. If the guess limit is enforced only on a per-user basis, then an attacker can often compromise *some* account on the machine if it is allowed 10 guesses at *every* user's password. Preventing these types of attacks requires some additional measures: websites that use password authentication rate-limit guesses by IP, or use CAPTCHAs, etc.

Try to coerce users into picking stronger passwords. Modern websites will often provide the user with a "password-strength checker" that tries to give the user some sense of how strong or weak their password is. These strength meters are completely heuristic and can

be wildly wrong: they might say that `6175551212` is a great password; if the attacker knows that 617-555-1212 is my phone number, it is probably not a great password. These strength meters sometimes check a user's password against public lists of popular passwords. Ensuring that your password isn't in the million most popular ones gives you at least some protection against untargeted attacks.

Two common strategies for encouraging users to choose strong passwords that don't work terribly well are:

- *Require longer passwords.* If someone tries to use `abc123` as a password but it's not long enough, they might use `abc123456`—but this doesn't really add much uncertainty. There are standard ways to lengthen passwords, and a clever attacker will try these first.
- *Prohibit using common English words in passwords.* It's not clear that this is a good idea. Five randomly chosen words from the dictionary will form a strong password, and prohibiting English words in passwords may make passwords much more difficult to remember.

2.2.3 *Avoiding weak passwords with a password manager*

When using passwords to authenticate to a website, a user can install a password manager on their computer that will generate random passwords for them. Since the user doesn't need to remember these passwords, they can be sampled truly at random from a high-entropy distribution. Once the user authenticates to their computer (using a password, typically), they can then access their randomly generated passwords and use them to log in to their websites.

Internally, the password-manager software maintains a table of servers and the corresponding passwords:

server	user	pw
`amazon.com`	`alice`	`3xyt42...`
`mit.edu`	`alice4`	`a21\$z...`

Even when using a password manager, password-based authentication schemes provide *no security* against eavesdropping or active attacks. If an attacker can observe you sending your password to the server (e.g., with a phishing attack) it can still authenticate as you.

2.2.4 *Password hashing: Trying to get some protection against server compromise*

Password-based authentication schemes provide no security against active attacks, in which the attacker compromises the server. And yet,

since attackers manage to breach web servers quite often, we would really like to provide some defense against server compromise.

Since, as we have seen, passwords are easy to guess, avoiding password-based authentication entirely is the safest option where possible. When a system must use passwords for authentication, the safest way to store them (e.g., on a server) is using a *salted cryptographic password-hashing function*. The goal is to make it as difficult as possible for an attacker to recover the plaintext passwords, given the hashed values stored on the server.

Forcing password changes. A system may force their users to change passwords on a regular schedule (e.g., every six months). If an attacker has compromised the password database on the server, it only has a limited amount of time to access the system before the passwords change and it will get locked out. It is not clear that the cost of requiring frequent password changes is worth the benefit.

To describe how this works: when a user creates an account with password pw, the server chooses a random 128-bit string, called a *salt*, and the server stores the salt and the hash value $h = H(\text{salt}\|\text{pw})$, where H is a special password-hashing function.

The server then stores a table that looks like this:

user	salt	$H(\text{salt}\|\text{pw})$
alice	r_a	h_a
bob	r_b	h_b

A *rainbow table* is a common data structure that an attacker can use to invert unsalted password hashes. A rainbow table is essentially a compressed table $(\texttt{passwd}, H(\texttt{passwd}))$ pairs, where H is a common hash function, and `passwd` ranges over a large set of common passwords.

An attacker can download rainbow tables for common cryptographic hash functions, such as MD5 or SHA1, from the web. If the password hash is salted with a 128-bit salt, it will be infeasible to produce a table that covers any reasonably large fraction of the $(\texttt{salt}\|\texttt{passwd})$ pairs.

Later on, when the user sends a password pw' to the server to authenticate, the server can use the salt and hash function to compute a value $h' = H(\text{salt}\|\text{pw}')$. If this hash value h' matches the server's stored value h for this user, the server accepts the password.

To explain the rationale for this design:

- The password-hashing function H is designed to be relatively expensive to compute—possibly using a large amount of RAM and taking a second or more of computation. This makes it more difficult for an attacker to brute-force invert the hash value, since each guess at the password requires a second of computation (instead of the microseconds required to compute a standard hash function, such as SHA256).
- The use of a per-user random salt ensures that guesses at one user's password are useless in inverting another user's password hash. Salting also defeats *precomputation attacks*, in which an attacker precomputes the hashes of many common passwords to speed up this hash-inversion step later on.

2.2.5 *Biometrics*

Biometrics are physical features like your fingerprints, your face, etc. These are essentially a type of bearer token: whoever is able to produce a face that looks like yours is able to authenticate as you.

Biometrics are very convenient to use for authentication, since you will not forget them and cannot easily lose them. Biometrics most useful when authenticating in person to a device, such as for

phone unlock, or to grant a person access to a secure vault. In these settings, the device performing the authentication has a "trusted input path" that can provide some assurance that a real human who owns that biometric is on the other end. Biometrics are not so useful for authenticating over a network because the network typically does not provide a trusted input path (i.e., does not provide any assurance that the biometric readings are coming from a real human), and the biometric data itself is not particularly secret. In particular, if we used biometrics for network authentication, an adversary who knows what your fingerprints looks like could log in to your account. (Since biometrics are essentially impossible to change, this is a major drawback.)

2.3 *Protecting against eavesdropping attacks: Challenge-response protocols*

We have so far been talking about a human manually authenticating to a device (ATM, phone, laptop, etc.) by physically entering a PIN or password into the device. But we often log in to some server on the network—Facebook, Gmail, MIT, and so on. In this scenario, we can get much more creative with the authentication mechanism we use and the security properties we can demand.

We now assume that our computer has some key k (e.g., a random 128-bit string), and the server also holds the same key k. In this setting, we can hope to provide security against eavesdropping attacks: even if an attacker can observe the traffic between the client and server, the attacker learns no information that can help it authenticate as the client later on.

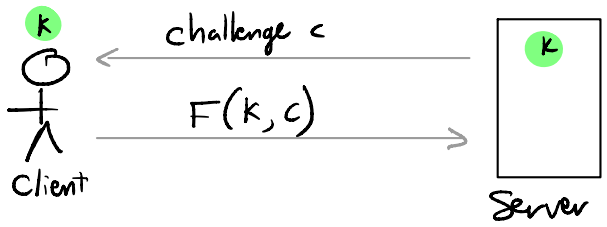


Figure 2.4: A challenge-response-based authentication system. The server and client share a secret key k. To authenticate, the client sends a function F of its secret key and the server's challenge c.

Figure 2.4 describes an über-simplified challenge-response authentication scheme that provides security against eavesdropping attacks. The protocol takes place between a client and server holding a shared secret key k.

1. The server chooses a long random string c, which we call a *challenge* and sends it to the authenticating client.

2. The client computes an authentication "tag" $t \leftarrow F(k, c)$, where $F(\cdot, c)$ is hard to compute without knowing k. (The function F here is a Message Authentication Code, which we will talk more specifically about in Chapter 4.) The sends the MAC tag t to the server.

3. The server receives a tag t' from the client and ensures that $t' = F(k, c)$. If so, the server considers the authentication successful.

The security of this scheme derives from the fact that an attacker cannot produce tags $F(k, c)$ on new challenge values c. (An attacker can always try to replay an old tag it has seen, but since the challenge changes with every authentication request, the server will always reject the old tag.)

2.3.1 *Time-based One-Time Passwords (TOTP)*

Time-based one-time passwords are a type of challenge-response authentication protocol. The only difference from Fig. 2.4 is that in a TOTP scheme, the client and server derive the challenge from the current time. The user has a device, such as a phone, that shares a secret key k (e.g., a random 128-bit string) with the server. Both parties agree on a protocol by which to generate this code—something like $F(k, \texttt{gettimeofday() / 30})$. The phone can generate the code, display it to the user, and the server can then verify the code by recomputing it.

2.3.2 *Authenticating requests*

Often, a client will want to send an *authenticated message* to a server. That is, the client often wants to simultaneously authenticate to the server and send a request req, such as $\mathsf{req} = \texttt{rm file.txt}$. To accomplish this, the client can compute the challenge value as the hash of the server-provided challenge c and the client's request. So the tag looks like: $t_{\mathsf{req}} \leftarrow F(k, c \| \mathsf{req})$. Then the client sends the pair $(t_{\mathsf{req}}, \mathsf{req})$ to the server. In this way, the server can simultaneously authenticate the client and be sure that the request req came from the client.

An **unsafe** way for the client to simultaneously authenticate to the server and send a request would be for the client to compute the MAC tag $t \leftarrow \mathsf{MAC}(k, r)$ and then send (t, req) to the server. A network attacker could modify the client's request to (t, req') en route to the server without the server being able to detect this attack.

2.3.3 *Phishing attacks (attacker-in-the-middle attacks)*

A *phishing* attack is one in which an attacker tricks a user into giving away their Gmail password, for example, by creating a website that looks, for example, like the `gmail.com` login page. TOTP passwords have a similar vulnerability: an attacker can simply ask the user to give her the one-time code by pretend to be tech support, or the user's employer, or a customer-service representative. In this setting, TOTP codes are slightly better than standard passwords since the attacker must use a stolen TOTP code within ≈30 seconds of stealing it, which requires a much more sophisticated attack.

Phishing attacks take advantage of the fact that in password- and TOTP-based authentication schemes, there is no binding between the authentication process and the server's subsequent communication with the client. The attacker here doesn't really break the authentication scheme; the problem is that the authentication scheme didn't authenticate enough. U2F, which we now discuss, handles that issue.

2.4 *Protecting against active attacks: Signatures and U2F*

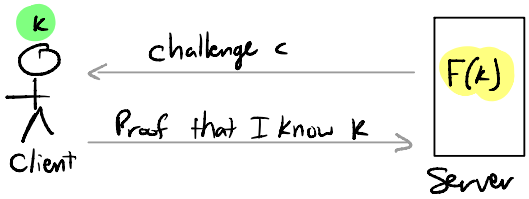


Figure 2.5: An authentication scheme based on digital signatures.

To provide security against active attacks, we can use an authentication scheme based on digital signatures, which we will discuss in Chapter 5. With these schemes, the client has a secret key k and the server stores some hard-to-invert function of the key $F(k)$. (We call $F(k)$ the "public key.") In particular, the server *does not store any secrets*—even if the attacker can compromise the server and/or interact with the honest client, it cannot learn the client's secret key k nor learn any information that it can use to later authenticate as the client. To authenticate, the server sends the client a challenge and the client produces a digital signature on the challenge c—essentially a proof that the client knows the secret key k and that it intended to sign the challenge c.

The U2F USB security tokens that you may have seen use this form of authentication. As an added bonus, they prevent phishing attacks by binding the authentication process to the name of the server that the client is trying to authenticate to. In particular, the U2F software

on the client passes the name of the server (e.g., `amazon.com`), in addition to a server-provided random challenge c, to the U2F token. The token then produces a signature on the string $c \| \texttt{amazon.com}$. If the attacker sets up `amason.com` and gets the user to visit it, the U2F device will only generate a code that is good for `amason.com` and not the real `amazon.com`.

2.5 *Two-Factor Authentication*

Many systems use multiple forms of authentication to try to boost security. In particular, as we have already seen, passwords are a weak authentication mechanism: humans are bad at choosing strong passwords and attackers have become good at stealing password databases and recovering many users' passwords at once.

A common technique to harden password-based authentication systems is to combine passwords with a second method of authentication—one with a different failure mode. Common authentication schemes are:

- Something you know: password, PIN, etc
- Something you have: USB key, phone, etc
- Something you are: biometrics (fingerprint, face ID)...

3
Collision Resistance and File Authentication

In the last chapter, we focused on authenticating people—ensuring that a person (or a request on behalf of that person) is likely who they claim to be. In this chapter, we will focus on authenticating files, code, and other data. When we say that we want to authenticate a file, we mean that we want to verify that the file's contents are exactly as they were when we or someone we trust last viewed them. The key new tool we use to do so is a *collision-resistant hash function*.

3.1 Intuition: Collision resistance

For our purposes, a hash function H maps a bitstring of any length onto a fixed-size space of outputs, so the type signature is $H : \{0,1\}^* \to \{0,1\}^\lambda$.

In order for a hash function to be *collision-resistant*, we want it to be the case that for any input, the generated output should be "unique." Of course, it cannot really be unique—we are mapping infinitely many inputs onto finitely many outputs—but we want it to be *computationally infeasible* to find a pair of distinct inputs that have the same hash values (a "collision").

> **Security goal:** A hash function H is *collision resistant* if it is "computationally infeasible" to find two distinct strings x and x' such that $H(x) = H(x')$.

Given a long message m, it's hash $H(m)$ under a collision-resistant hash function is like a short "fingerprint" of the message—the hash essentially uniquely identifies the message m. For that reason, collision-resistant hash functions let you authenticate a long message m by authenticating the short fixed-length string $H(m)$. We often call the hash value $H(m)$ a *digest*.

3.1.1 Applications

Secure File Mirroring. Often a user wants to download large files (e.g., software updates) from a far-away server. To speed up this process, a company or Internet-service provider may set up local *mirrors* of the remote files. Users can then download the files from the nearby mirror instead of the far-away server. However, without additional security measures, the mirror may server users a different file than the one the mirror fetched from the origin server. If the mirror is malicious, it can, for example, trick the user into installing a backdoored software update. (We saw an attack based on mirrors in Section 1.3.1.)

To protect against a malicious mirror, we can add some authentication on the file that the mirror hosts. Say that the origin server publishes a large software update f. The origin server will send the file f to its mirrors and the origin server itself will serve the hash digest $d \leftarrow H(f)$ to anyone who asks for it. A user who wants to fetch the update can download d from the origin server directly—this will be fast since the digest is tiny. Then, the client can fetch the update itself from a (potentially untrustworthy) mirror. When the client receives a file $\hat{f}$ from the mirror, it can check that $d = H(\hat{f})$ to ensure that $\hat{f}$ is the true software update. If H is collision resistant, then if the hash value $H(\hat{f})$ matches the origin server's digest d, the files are almost certainly identical.

Subresource Integrity. If a program fetches a file from some content delivery network, it can store the hash of that file locally and use it to verify that the contents of the file did not change since the application was developed.

Outsourced File Storage. If you want to store your files on a cloud provider, you want to be sure that the cloud provider does not maliciously modify the files without you noticing. To make sure of this, you can store $H(\text{files})$ locally, which takes very little storage space. Then, when you redownload your files locally, you can recompute the hash to verify that they were not tampered with.

3.2 Defining collision resistance (slightly more formally)

An adversary's goal in breaking a collision resistant hash function is to find a collision—a pair of values $m_0, m_1 \in \{0,1\}^*$ such that $m_0 \neq m_1$ and $H(m_0) = H(m_1)$.

Definition 3.2.1 (Collision Resistance)**.** A function $H : \{0,1\}^* \to \{0,1\}^\lambda$ is collision-resistant if for all "efficient" adversaries $\mathcal{A}$, we

have that:

$$\Pr[H(m_0) = H(m_1), m_0 \neq m_1 : (m_0, m_1) \leftarrow \mathcal{A}()] \leq \text{“negligible”}$$

In words, this means that the probability of finding a collision is so small that no efficient adversary could hope to do it.

There are two ways of thinking about the terms “efficient” and “negligible” that we use in this definition—one mindset we use in practice and the other mindset we use in theory.

- In theory...
  - All of our cryptographic constructions are parameterized by an integer $\lambda \in \{1, 2, 3, \dots\}$ that we call the *security parameter*. So instead of a single collision-resistant hash function H, we have a family of functions $\{H_1, H_2, H_3, \dots\}$, where the function H_λ has λ-bit output.
  - An “efficient” algorithm is a randomized algorithm that runs in time polynomial in λ.
  - A “negligible” function is one that grows slower than the inverse of every polynomial—a function that is $O(\frac{1}{\lambda^c})$ for all constants $c \in \mathbb{N}$.
- In practice...
  - We use a fixed hash function H with a fixed-length output, which might be as 256 or 512 bits.
  - An “efficient” adversary is one that runs in time $\leq 2^{128}$.
  - A “negligible” probability is some very small constant, like one smaller than 2^{-128}.

3.2.1 *Understanding which attacks are feasible*

Typically, we think of an attack that runs in more than 2^{128} time as infeasible and an event that happens with probability less than 2^{-128} is one that will never happen. These seemingly magic constants come from empirical considerations:

2^{30}	operations/second on a laptop
2^{58}	ops/sec on Fugaku supercomputer
2^{68}	hashes/second on the Bitcoin network (as of Fall 2022)
2^{92}	hashes/yr on the Bitcoin network
2^{114}	hashes required to use enough energy to boil the ocean
2^{140}	hashes required to use one year of the sun’s energy

See Lenstra, Kleinjung, and Thomé for an entertaining discussion of these constants.[1]

[1] Arjen K Lenstra, Thorsten Kleinjung, and Emmanuel Thomé. “Universal security”. In: *Number Theory and Cryptography*. 2013.

For most cryptosystems, there is a tradeoff between the attacker’s running time and success probability. For example, an attacker running in time T can find a collision in a hash function with n-bit output with probability $T^2/2^n$. So, as the attack runs for more

2^{-1}	fair coin lands heads
2^{-13}	probability that a randomly sampled MIT grad is a Nobel Prize winner
2^{-19}	probability of being struck by lightning next year
2^{-28}	probability of winning the Mega Millions jackpot
2^{-128}	will essentially never happen

The takeaway is that if an attacker finds a collision with probability 2^{-128}, we can be extremely sure that a collision will never occur.

3.3 *Constructing a collision-resistant hash function*

The current standard for fast collision-resistant hashing is SHA256 (a.k.a. SHA2), which was designed by the NSA in 2001. The SHA2 hash functions are designed using the following common two-step approach:

We can also build collision-resistant hash functions that are secure under "nice" cryptographic assumptions, such as the assumption that factoring large numbers is hard. Unfortunately, hash functions based on these nice assumptions tend to be very slow and, as a result, are almost never used in practice.

1. Build a small collision-resistant hash function on a fixed-size domain $H_{small} : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$. This step is, to some degree, "more art than science". The standard practice is to design a hash function that defeats all known collision-finding attacks. If no known attack works well, we declare the candidate function to be collision resistant.
2. Use H_{small} to construct $H : \{0,1\}^* \to \{0,1\}^{\lambda}$. This can be done very cleanly using the "Merkle-Damgård" approach described below. This step requires no additional assumptions: we can prove unconditionally that if H_{small} is collision resistant, then H is as well.

Another way to build collision-resistant hash functions is to use the so-called "sponge" construction. It is similar to the approach described here in that we start with a small primitive, which we assume secure in some sense, and then we use the small primitive to build a hash function on a large domain.

3.3.1 *Merkle-Damgård*

The Merkle-Damgård construction gives a way to construct a collision-resistant hash function for all bitstrings (i.e., $\{0,1\}^*$) from a collision resistant hash function that maps 2λ-bit strings down to λ-bit strings, sketched out in Figure 3.1.

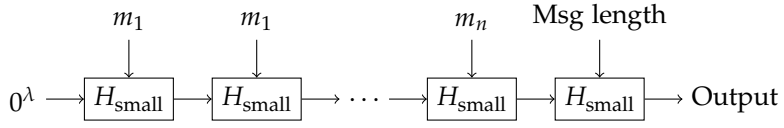


Figure 3.1: Sketch of the Merkle-Damgård construction for a collision-resistant hash function.

The Merkle-Damgård construction first splits the message into λ-sized blocks $[m_1, \ldots, m_n]$ and successively hashes them together. In

the following pseudocode, the function ToBlock converts an integer, representing the length of the input message in blocks, into a λ-bit string. Then the Merkle-Damgård construction is: (Here, we are

$H(m_1, \ldots, m_n)$: // Merkle-Damgård construction

- Let $b \leftarrow 0^\lambda$.
- For $i = 1, \ldots, n$:
  - Let $b \leftarrow H_{\text{small}}(b, m_i)$.
- Let $b \leftarrow H_{\text{small}}(b, \mathsf{ToBlock}(n))$.
- Output b.

Figure 3.2: The Merkle-Damgård construction of a large-domain collision-resistant hash function H from a small-domain collision-resistant hash function H_{small}.

assuming that the message is at most $\lambda 2^\lambda$ bits long.)

In practice, standard hash functions have limits on the length of the messages that they can hash. For example, SHA256 can hash messages of length up to $2^{64} - 1$ bits.

We won't prove it here, but we can use the fact that H_{small} is collision-resistant to prove that H must also be collision-resistant. The basic idea of the proof is to show that given a collision in H, we can easily compute a collision in H_{small}.

Note: In the Merkle-Damgård construction of Fig. 3.2, we initialize the variable b to the all-zeros string. The construction is collision-resistant if we omit the all-zeros string and start by setting $b \leftarrow m_1$ and then continue by hashing $m_2, m_3, \ldots$. The construction is *not* collision resistant if we omit the length block $\mathsf{ToBlock}(n)$ that we hash in at the end.

3.3.2 *The Birthday Paradox*

If you sample $2^{\lambda/2}$ random 10λ-bit strings and hash them with a hash function that has λ-bit outputs, you will find a collision among these inputs with constant probability.

An important thing to understand when dealing with hash functions is the "Birthday Paradox," which states that given a hash function with λ-bit output, you can always find a collision in time $O(\sqrt{2^\lambda}) = O(2^{\lambda/2})$. So, if you want to force an attacker to use at least 2^{128} to find a collision, you must use a hash function with at least 256 bits of output.

3.3.3 *Domain Separation*

In many applications, we have a one-input CRHF (such as SHA256) $H : \{0,1\}^* \to \{0,1\}^{256}$ and we need to construct a two-input CRHF $H_2(x, y)$.

Bad idea. An obvious solution to construct the two-input hash function H_2 is to concatenate the two values, so that $H_2(x, y) =$

$H(x||y)$. However, this construction allows two different pairs of messages to hash to the same value:

$$H_2(\text{"key"}, \text{"value"}) = H_2(\text{"ke"}, \text{"yvalue"}).$$

Both Amazon and Flickr had a bug arising from this—they concatenated all parameters before hashing, and had parameters such that two different intents had the same concatenation.[2]

[2] Thai Duong and Juliano Rizzo. *Flickr's API Signature Forgery Vulnerability*. `https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html`. Sept. 2009.

3.3.4 *Length-Extension*

Recall the concept of Message Authentication Codes (MAC) from the last lecture—a code that can be sent along with a message to verify that the message was not changed. (We will see the formal definition in Chapter 4.)

Bad idea. Poorly designed software uses $\mathsf{MAC}(k, m) = H(k\|m)$ as a very simple construction of a MAC. However, this construction has an easy attack—given $\mathsf{MAC}(k, m)$, it is easy to compute $\mathsf{MAC}(k, m\|m')$ without knowing the key k if H is a hash function built with the Merkle-Damgård construction. To do this, the attacker hashes the output of $\mathsf{MAC}(k, m)$ with two more blocks—a new message m'' and another length block. Now, we have computed $\mathsf{MAC}(k, m\|m')$ where m' is the original length block plus some custom message without knowing the key k.

This problem here is that we were using a hash function that was *only* guaranteed to be collision resistant, but we assumed that it had other properties (such as that it is guaranteed to be difficult to compute the hash of an extension of the original message). Figure 3.3 sketches out the length-extension attack on the Merkle-Damgård construction from Figure 3.1.

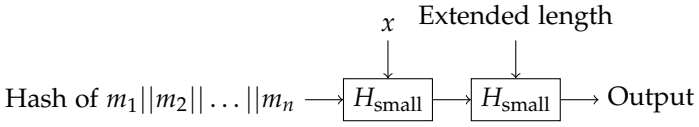


Figure 3.3: Sketch of the extension attack on the Merkle-Damgård construction, starting with a hash of $m_1||m_2||\ldots||m_n$ to compute the hash of $m_1||m_2||\ldots||m_n||\mathsf{ToBlock}(n)||x$.

3.4 *Applications: Merkle Trees*

In many settings, an origin server has N files (e.g., Android app binaries) and wants to serve these files from potentially untrustworthy mirror servers (e.g., Akamai servers) distributed around the globe.

To do this, the origin server can put the N files at the leaves of a binary tree. Then the server hashes together pairs of files, then hashes each pair of hashes and so on until it eventually ends up with

a single root hash h. The client fetches the root hash h from the origin server directly.

Later on, the client can download any one of the N files from the untrustworthy mirror server. The mirror can produce the file, along with $O(\log N)$ hashes—the sibling nodes of each node on every path from the file's leaf to the root. The client can use the root hash h it got from the origin server, along with the additional hashes from the mirror server, to be convinced that the mirrored file it downloaded was authentic.

TODO: Add diagram from lecture.

4
Message Authentication Codes

So far, we have talked about authenticating *people* and authenticating *files*. In this section, we will discuss authenticating *communication*. If we have two parties that are communicating over the network, we want some way to guarantee to each party that the message they received really came from the other party and was not tampered with along the way.

At a first glance, this seems impossible. If there is some eavesdropper Eve in between the two parties, they can just replace the message with one of their own choosing and the other party will have no idea. To make this possible, we need to relax the scenario a bit and add an assumption—that the two parties share some secret key k.

With this shared key k between the two parties, our goal will be to add some "tag" onto the message that validates its authenticity. Necessarily, this tag will be a function of this shared key k. If this were not the case, the eavesdropper would be able to compute a valid tag herself—the secret k is the only information in this scenario that Eve does not know.

4.1 Defining message authentication codes

Syntax. A message authentication code (MAC) over key space $\mathcal{K}$, message space $\mathcal{M}$, and tag space $\mathcal{T}$ is an efficient algorithm $\mathsf{MAC}\colon \mathcal{K} \times \mathcal{M} \to \mathcal{T}$. In order for a MAC to be useful, it must be *secure*, in the following sense. We first give the definition and then explain why it is a useful one:

Definition 4.1.1 (MAC Security: Existentially unforgeability against adaptive chosen message attacks)**.** A MAC MAC over key space $\mathcal{K}$ and message space $\mathcal{M}$ is secure (existentially unforgeable against adaptive chosen message attacks) if any poly-time adversary $\mathcal{A}$ wins the following game with at most negligible probability:

- The challenger samples a MAC key $k \xleftarrow{R} \mathcal{K}$.
- For $i = 1, 2, \ldots$ (polynomially many times)

In practice, "a poly-time adversary" means "any real-life adversary". But we need to place some mathematical bound on real-life to make the proofs work out.

  - The adversary sends any message $m_i \in \mathcal{M}$ to the challenger
  - The challenger responds with $\mathsf{MAC}(k, m_i)$.
- The adversary sends the challenger a message-tag pair (m^*, t^*).
- The adversary wins the game if $\mathsf{MAC}(k, m^*) = t^*$ and $m^* \notin \{m_1, m_2, \ldots, m_n\}$.

A subtlety of this definition is that, even if the MAC scheme is secure under this definition, it is possible for an adversary, given a valid message-tag pair (m, t) to produce a second valid message-tag pair (m, t') on the same message without knowing the secret key.

4.1.1 *Intuition for the security definition*

To formulate our security notion, we need to define the adversary's goal and the adversary's power.

The adversary's goal in this definition is to compute a valid MAC of *any* message $m \in \mathcal{M}$ of its choice. It's not entirely obvious why we care about the adversary producing a valid MAC on *any* message: "If the adversary MACs a message that is jibberish, they are unlikely to be able to do any harm with it," you might think. But there will certainly be applications that authenticate messages that violate whatever definition of "non-jibberish" we define. So allowing the adversary to forge a MAC tag on any message makes the definition as broadly applicable as possible.

This has some interesting implications—importantly, the adversary can store these messages along with their MAC and replay them later.

As far as the adversary's power goes: we, as usual in cryptography, restrict the adversary to be efficient (i.e., to run in polynomial time). But in the MAC security game we also allow the adversary to obtain MAC tags on messages of its choice. This captures the reality that in many systems, an adversary can trick an honest system into MACing adversarial messages. For example, if an email-backup system MACs every email that a user receives, an adversary may be able to obtain MAC tags on messages of its choice by sending emails to the backup system.

4.1.2 *MACs require pseudorandomness*

The fact that it is even possible to construct a MAC seems a bit surprising—in effect, for a MAC to satisfy the definition, the tag has to effectively be random. But the only "randomness" that we have is the key k—to generate tags for arbitrarily many messages, we need much more randomness than one key's worth. This seems impossible. How can we generate a large number of random-looking tags from only a single short random key?

We get ourselves out of this conundrum by observing that the adversary must be an *efficient* algorithm. So while we cannot generate a large number of truly random bits from a short key, we can—under appropriate and reasonable cryptographic assumptions—generate a large number of bits that *look* truly random from the perspective of any efficient algorithm. We call these bits *pseudorandom*.

This surprising and powerful idea leads us to our next cryptographic primitive...

4.2 *Pseudorandom Functions*

A pseudorandom function is defined over a keyspace $\mathcal{K}$, and input space $\mathcal{X}$ and output space $\mathcal{Y}$. To be useful a pseudorandom function must satisfy the following security definition:

Definition 4.2.1 (Pseudorandom Function, PRF)**.** A function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a pseudorandom function if all efficient algorithms $\mathcal{A}$ win the following game with probability $\frac{1}{2}$ + "negligible":

- The challenger samples a random bit $b \leftarrow \{0,1\}$ and a key $k \xleftarrow{R} \mathcal{K}$.
- If $b = 0$, the challenger sets $f(\cdot) := F(k, \cdot)$.
- If $b = 1$, the challenger sets $f(\cdot) \xleftarrow{R} \mathsf{Funs}[\mathcal{X}, \mathcal{Y}]$.
- Then for $i = 1, 2, \ldots$ (polynomially many times):
  - The adversary sends the challenger a values $x_i \in \mathcal{X}$.
  - The challenger responds with $y_i \leftarrow f(x_i) \in \mathcal{Y}$.
- The adversary outputs a guess $\hat{b}$ at the bit b.
- The adversary wins if $b = \hat{b}$.

Here, Funs is the set of all functions from $\mathcal{X}$ to $\mathcal{Y}$.

First, the challenger will sample a random $b \leftarrow \{0,1\}$ and a key $k \leftarrow \mathcal{K}$.

The adversary can trivially win this game with probability $\frac{1}{2}$ by just guessing the bit b at random. This definition asserts that no efficient adversary can do much better than that.

If we have such a pseudorandom function F, we could easily construct a MAC—we can just use the message as the input to the pseudorandom function along with the key: $\mathsf{MAC}(k, m) := F(k, m)$.

4.2.1 *Constructing pseudorandom functions from one-wayness*

It is not at all obvious that pseudorandom functions should exist at all! They seem like a very magical primitive indeed.

One surprising fact is that if there exists *any* function that is "hard to invert," in a sense we will define, then pseudorandom functions exist. For example, if you believe that factoring large numbers is difficult (as many people do), then pseudorandom functions exist.

In particular the following definition captures the notion of a function that is hard to invert:

Definition 4.2.2 (One-Way Function)**.** A function $f\colon \mathcal{X} \to \mathcal{Y}$ is a *one-way function* if for all efficient adversaries $\mathcal{A}$,

$$\Pr[f(\mathcal{A}(f(x))) = f(x)\colon x \xleftarrow{R} \mathcal{X}] \leq \text{"negligible"}.$$

Having defined one-way functions, we now have the following surprising and non-obvious result:

Notice that if $\mathsf{P} = \mathsf{NP}$, one-way functions do not exist, and therefore psuedorandom functions do not exist.

Theorem 4.2.3. *Psueodorandom functions exist if and only if one-way functions exist.*[1]

[1] J. Hastad et al. "A Pseudorandom Generator from any One-way Function". In: *SIAM Journal on Computing* 28.4 (1999), pp. 1364–1396.

In practice, we assume that:

- the function $f(x) := \text{SHA256}(x)$ is a one-way function where the domain is the set of 256-bit strings,
- the function $f(x) := \text{AES}(x, 0^{128})$ is a one-way function, where the domain is the set of 128-bit strings, and
- the function $f(x) := 2^x \bmod p$ is a one-way function on domain $\{1, \ldots, p\}$, for a sufficiently large prime p.

4.2.2 *Pseudorandom functions in practice*

In practice, we use the Advanced Encryption Standard (AES) as a pseudorandom function. The AES function on key length $\kappa \in \{128, 192, 256\}$ has the type signature $\mathsf{AES} : \{0,1\}^\kappa \times \{0,1\}^{128} \to \{0,1\}^{128}$. That is, it takes a 128-bit input and generates a 128-bit output.

We don't have any mathematical proof that AES is a pseudorandom function. However, it has undergone a tremendous amount of cryptanalysis and the best attacks on AES are only marginally better than the obvious brute-force attacks.

4.3 *From pseudorandom functions to MACs*

MACs for short messages. Using AES as a pseudorandom function on a 128-bit domain, we can build a MAC for 128-bit messages as described above : $\mathsf{MAC}(k, m) := \mathsf{AES}_k(m)$. However, since AES takes only 128 bits as input, using AES directly, we can only authenticate 128-bit messages.

Insecure ways to construct a MAC for long messages. A bad way to construct a MAC for long messages from a pseudorandom function F for 128-bit messages is just to chop our message m up into 128-bit blocks $m = (m_1, m_2, \ldots)$ and MAC each block separately. Our tag, then, would look something like $(F(k, m_1), F(k, m_1))$. However, there is a problem! Given the tag $t = (t_1, t_2)$ for a message $m = (m_1, m_2)$, we can easily generate a valid tag $t' = (t_2, t_1)$ for a different message $m' = (m_2, m_1)$.

MACs for long messages: The easy way. If we have a pseudorandom function F with an input space of 256-bits, we can construct a MAC on arbitrary-length messages using the "hash-and-sign" paradigm. In particular, we use a collision-resistant hash function $H\colon \{0,1\}^* \to$

Notice that we cannot use AES as the pseudorandom function F in this construction, since AES only takes a 128-bit input. In this case, we would need a collision-resistant hash function $H\colon \{0,1\}^* \to \{0,1\}^{128}$, but it is always possible to find collisions in hash functions with 128-bit output in time 2^{64}. So such a MAC can never be secure against attackers running in time 2^{64}.

$\{0,1\}^{256}$ (Theorem 3.2.1) and we define the MAC on message space $\{0,1\}^*$ as:

$$\mathsf{MAC}(k, m) := F(k, H(m)).$$

In practice, we typically do not construct MACs in this way because collision-resistant hash functions are typically more expensive to compute (per bit of input) than pseudorandom functions, such as AES.

4.3.1 *MACs for long messages: Cipher-Block Chaining MAC*

A common and secure way to construct a MAC for long messages from a MAC for short messages is to *chain* the output of each of these calls to the pseudorandom function. Given our chopped message $(m_1, m_2, \ldots, m_n)$, we will generate $t_1 = F(k, m_1)$ as before. When generating t_2, we will first XOR t_1 into the input: $t_2 = F(k, m_2 \oplus t_1)$. This continues until the end of the message, at which point have a tag t_n. Finally, we apply the PRF *with a different key* k' to the value t_n and output this tag $t \leftarrow F(k', t_n)$. This construction is called CBC-MAC or CMAC.

Applying the PRF to the last block using an independent random key is important. If we do not use a new key, an adversary can mount a length-extension attack. That is, if the adversary asks for $t = \mathsf{MAC}(k, m_1)$ and $t' = \mathsf{MAC}(k, t)$, t' is also a valid key for the original message with two zero blocks attached $\mathsf{MAC}(k, m_1 \| 0 \| 0)$. The chain of AES applications becomes equivalent, since zero blocks are equivalent to skipping the XOR and adding AES applications.

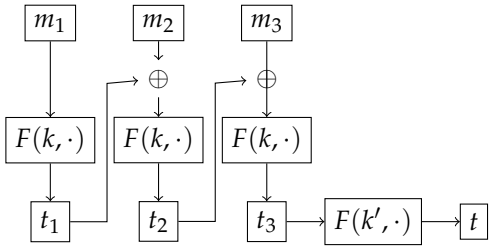


Figure 4.1: The CBC-MAC construction.

CBC-MAC is going out of favor for two reasons:

1. It is impossible to parallelize the MAC computation: the chaining procedure is inherently sequential so you cannot speed it up, even if you have a computer with many CPU cores.
2. Computing the MAC requires one PRF invocation *per block* of the message. There are even faster MACs that require only one PRF invocation *per message* total, plus a number of fast "non-cryptographic" operations per message block. These MACs can be faster than CBC-MAC on some processors. The GMAC construction we will see next is one example.

4.3.2 *A parallelizable MAC: Carter-Wegman MAC*

We now describe a different way to authenticate long messages. This MAC scheme is parallelizable and also requires only one single PRF invocation per message authenticated (independent of the message

length). The construction is named the Carter-Wegman MAC, after its inventors.[2] Modern encryption schemes, including AES-GCM (Section 9.2) use a Carter-Wegman-style MAC as a key ingredient.

[2] Mark N Wegman and J Lawrence Carter. "New hash functions and their use in authentication and set equality". In: *Journal of computer and system sciences* 22.3 (1981).

For this construction, we will use the notation $\mathbb{Z}_p$ to indicate the set of integers modulo p with addition and multiplication modulo p. So $x + y \in \mathbb{Z}_p$ means that we add x and y as integers and reduce the result modulo p. Typically, we will think of p as a prime—of 64 bits, for example. The construction uses a fixed a prime number p as a parameter, where $p \approx 2^n$ for security parameter n.

So in practice, we take $p \approx 2^{128}$ for 128-bit security.

Universal hash function. Before we look at the construction of the Carter-Wegman MAC, we first define an important building block: the notion of a *universal hash function*, or UHF for short. A universal hash function is keyed, and provides collision-resistance when the adversary does not know the key. Specifically, we say that H is a universal hash function if, when an adversary chooses two messages m and m' where $m \neq m'$,

$$Pr[H(k, m) = H(k, m')] \leq \text{negl.}$$

Intuitively, a universal hash function is a weaker primitive than a collision-resistant hash function: the adversary does not know the precise hash function that will be applied to their messages, because the adversary does not know what key will be used.

One simple and practical construction of a universal hash function is based on polynomials. Given a long message m, break it up into fixed-size chunks $m_0, m_1, \ldots, m_{l-1}$. Then, the hash of that message is defined as:

In practice, these fixed-size chunks are going to be 128 or 256 bits long.

$$H(k, m_0||m_1||\ldots||m_{l-1}) = (m_0 + m_1 k + m_2 k^2 + \ldots + m_{l-1} k^{l-1}) \mod p$$

We can give some intuition for why H is a universal hash function (i.e., collision-resistant for a randomly chosen key). In order for a pair of messages m and m' to collide, it must mean that $H(k, m) = H(k, m')$, which in turn means that $H(k, m) - H(k, m') = 0$. Expanding the definition of H as a polynomial, this means that

$$(m_0 - m'_0) + (m_1 - m'_1)k + (m_2 - m'_2)k^2 + \ldots + (m_{l-1} - m'_{l-1})k^{l-1} = 0$$

which is another way of saying that k is a root of that degree-$l-1$ polynomial. We know that there can be at most $l-1$ roots of a degree-$l-1$ polynomial, but there are $p \approx 2^n$ possible choices for k, so the probability that our randomly-chosen k happens to be one of those roots is $\frac{l-1}{p}$, which is negligible.

MAC construction. The MAC uses a pseudorandom function $F\colon \mathcal{K} \times \mathbb{Z}_p \to \mathbb{Z}_p$. The keyspace for the MAC is $\mathcal{K}$, so the MAC key consists of a key for the pseudorandom function. The message space for the MAC is $\mathcal{M} = \mathbb{Z}_p^{\leq L}$, the set of vectors of integers of $\mathbb{Z}_p$ elements of length at most L where $L \ll p$. Here, assume that the message vector has length at least 1.

Here, the input space of the pseudorandom function F is the set of integers in $\{0, \ldots, p-1\}$. Given a pseudorandom function on bitstrings, it is indeed possible to construct one that operates on numbers in $\mathbb{Z}_p$ like this by interpreting each number as a bitstring.

One other difference is that this MAC construction is *randomized*. So there are now two algorithms:

- $\mathsf{MAC.Sign}(k, m) \to t$, which takes as input a key k and message m and outputs a MAC tag t, and
- $\mathsf{MAC.Verify}(k, m, t) \to \{0, 1\}$, which takes as input a key k, message m, tag t, and outputs an accept/reject bit.

The security definition here is essentially the same as for deterministic MACs, except that we use different algorithms to generate and verify the MAC tags.

The Carter-Wegman MAC construction is then:

$\mathsf{MAC.Sign}(k, m \in \mathbb{Z}_p^{\leq L}) \to t$.

- Compute $v \leftarrow F(k, 0) \in \mathbb{Z}_p$.
- Parse the message into chunks as $(m_1, \ldots, m_\ell) \leftarrow m \in \mathbb{Z}_p^\ell$.
- Compute $M(v) \leftarrow m_1 v + m_2 v^2 + m_3 v^3 \cdots + m_\ell v^\ell \in \mathbb{Z}_p$.
  TODO: HCG: Check the definition of the message polynomial M. Should there be an additional v^{t+1} monomial?
- Sample a nonce $r \xleftarrow{R} \mathbb{Z}_p$.
- Output $t \leftarrow \big(r, F(k, r) + M(v)\big) \in \mathbb{Z}_p^2$ as the MAC tag.

Essentially we are viewing the blocks of the message m as coefficients of a degree-t polynomial $M(\cdot)$. We then evaluate this polynomial at the secret point v determined by the MAC key.

$\mathsf{MAC.Verify}(k, m, t) \to \{0, 1\}$.

- Compute $v \leftarrow F(k, 0) \in \mathbb{Z}_p$.
- Parse the message into chunks as $(m_1, \ldots, m_\ell) \leftarrow m \in \mathbb{Z}_p^\ell$.
- Compute $M(v) \leftarrow m_1 v + m_2 v^2 + m_3 v^3 + \cdots + m_\ell v^\ell \in \mathbb{Z}_p$.
- Parse the tag $(r, z) \leftarrow t \in \mathbb{Z}_p^2$.
- Output "1" if and only if $z - F(k, r) = M(v)$.

Security intuition. The security argument here goes as follows:

For a detailed treatment of Carter-Wegman security see Boneh and Shoup's textbook, *A Graduate Course in Applied Cryptography*, Section 7.4.

- First, we appeal to the PRF security property to argue that we can replace the values $F(k_F, r)$ used to generate the tags with truly random values.
- Next, we show that as long as the MAC.Sign algorithm never samples the same nonce r twice, the masking values $F(k, r)$ are independent random values that complete hide the values $M(v)$. So, the adversary learns no information on the secret point v by making MAC queries.

- Now, say that the adversary finds a forged message-tag pair (m^*, t^*). There are two cases:
  - Either the forgery uses a fresh random nonce r^* that did not appear as the response to any of the adversary's MAC queries. In this case, the forgery is only valid with probability $1/p$.
  - Alternatively, the forger could use a random nonce r^* that is equal to the nonce r returned from one of the adversary's MAC queries. In this case, we have the following relations, where message m polynomial M was the message the adversary queried of the challenger:

    $$\begin{aligned} F(k,r) &= M(v) - z \\ F(k,r) &= M^*(v) - z^* \\ 0 &= \big(M(v) - M^*(v)\big) + (z - z^*). \end{aligned}$$

    Since $m \neq m^*$, we know $z \neq z^*$. So $(M(\cdot) - M^*(\cdot)) + (z^* - z)$ is a non-zero polynomial of degree at most t. Since such a polynomial can have at most $\ell \leq L$ zeros in $\mathbb{Z}_p$, and since the adversaries view is independent of the evaluation point $v \in \mathbb{Z}_p$, the probability that the adversary's forgery is valid is at most ℓ/p.

  In either case, the adversary's probability of forging is $O(L)/p = \text{poly}(\lambda) \cdot \text{negl}(\lambda) = \text{negl}(\lambda)$ on security parameter λ.

5
Digital Signatures

In the last section, our strategy for authentication depended on two parties sharing a secret key. In that discussion, we completely left out of the picture how these parties should exchange this secret key. Our implication was that they went to some private room and exchanged the key in secret, but in many cases this is not practical: if they could whisper a key, why not just whisper the message?

Luckily, there is a way to get around this requirement for a shared secret using *public-key cryptography*.[1]

5.1 Definitions

The basic idea of public-key cryptography, applied to authentication, is that each party will generate two linked keys—a secret signing key and a public verification key. The verification key will be good enough to verify that a signature is valid, but not to generate new signatures.

Definition 5.1.1 (Signature Scheme)**.** A signature scheme is associated with a message space $\mathcal{M}$ and three efficient algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$.

- $\mathsf{Gen}(\lambda) \to (\mathsf{sk}, \mathsf{vk})$. The key-generation algorithm as input a security parameter $\lambda \in \mathbb{N}$ and outputs a secret signing key sk and public verification key vk. The algorithm Gen runs in time $\mathrm{poly}(\lambda)$.
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$. The signing algorithm takes as input a secret key sk and a message $m \in \mathcal{M}$, and outputs a signature σ.
- $\mathsf{Ver}(\mathsf{vk}, m, \sigma) \to \{0,1\}$. The signature-verification algorithm takes as input a public verification key vk, a message $m \in \mathcal{M}$, and a signature σ, and outputs $\{0,1\}$, indicating acceptance or rejection.

For a signature scheme to be useful, a correct verifier must always accept messages from an honest signer. Formally, we have:

[1] Whitfield Diffie and Martin E Hellman. "New Directions in Cryptography". In: *Transactions on Information Theory* 22.6 (1976).
The original Diffie-Hellman paper from 1976, which introduced public-key cryptography, is a fascinating read.

In theoretical papers, people will write $\mathsf{Gen}(1^\lambda)$ to indicate that the key-generation algorithm takes as input a length-λ string of ones. This is just a hack to make the input given to Gen λ bits long so that the Gen algorithm can run in time polynomial in its input length: $\mathrm{poly}(\lambda)$. If we express λ in binary, then $\mathsf{Gen}(\lambda)$ gets a $\log_2 \lambda$-bit input and can only run in time $\mathrm{poly}(\log \lambda)$. This distinction is really unimportant, but if you see the 1^λ notation, you will now know what it means.

Definition 5.1.2 (Digital signatures: Correctness)**.** A digital-signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ is *correct* if, for all messages $m \in \mathcal{M}$:

$$\Pr\left[\mathsf{Ver}(\mathsf{vk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1 \colon (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(\lambda)\right] = 1.$$

The standard security notion for digital signatures is very similar to that for MACs (Theorem 4.1.1). The only difference here is that a digital-signature scheme splits the single secret MAC key into two keys: a secret signing key and a public verification key. Otherwise the definition is essentially identical.

Definition 5.1.3 (Digital signatures: Security – existential unforgeability under chosen message attack)**.** A digital-signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ is *secure* if all efficient adversaries win the following security game with only negligible probability:

- The challenger runs $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(\lambda)$ and sends vk to the adversary.
- For $i = 1, 2, \ldots$ (polynomially many times)
  - The adversary sends a message $m_i \in \mathcal{M}$ to the challenger.
  - The challenger replies with $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}, m_i)$.
- The adversary outputs a message-signature pair (m^*, σ^*).
- The adversary wins if $\mathsf{Ver}(\mathsf{vk}, m^*, \sigma^*) = 1$ and $m^* \notin \{m_1, m_2, \ldots\}$.

Notice that this security definition does not guarantee that an attacker cannot forge a new signature on a message that it has already seen a signature of. Namely, given a valid message-signature pair (m, σ) an adversary may be able to produce additional valid message-signature pairs on the same message: $(m, \sigma'), (m, \sigma''), \ldots$.

In some applications, we want to prohibit an attacker from finding *any* new message-signature pair. We call this security notion "*strong* existential unforgeability under chosen message attack." The definition is the same as in Theorem 5.1.3 except that we require the adversary to find a valid-message signature pair (m^*, σ^*) such that $(m^*, \sigma^*) \notin \{(m_1, \sigma_1), (m_2, \sigma_2), \ldots\}$. Standard digital-signature schemes, such as the elliptic-curve digital signature algorithm (ECDSA) or the RSA algorithm with full-domain hashing (RSA-FDH), are believed to have this strong security property.

5.2 *Constructing a Signature Scheme*

In the following sections, we will show how to construct a digital-signature scheme from any one-way function (Theorem 4.2.2).

We will generate a signature scheme that is secure, but that has relatively large signatures and public keys: to achieve security against

attackers running in time 2^λ, this signature scheme has signatures of $O(\lambda^2)$ bits. Widely used modern digital signature schemes (e.g., EC-DSA) have signatures of $O(\lambda)$ bits.

We will construct this scheme in three stages:

1. Construct a signature scheme for signing a *single bit*.
2. Construct a *one-time secure* signature scheme for signing a *fixed length* messages. With this scheme, an attacker who sees two signatures under the same signing key can forge signatures. In addition, the secret signing key for this scheme will be larger than the size of the message being signed.
3. Construct a *one-time secure* scheme for *arbitrary-length messages*. Here, we construct a one-time signature scheme whose secret signing key is independent of the length of the signed message.
4. Construct a *many-time secure* scheme (i.e., a fully secure one under Theorem 5.1.3) for *arbitrary-length messages*. This last scheme is a fully secure and fully functional digital-signature scheme.

One benefit of the signature scheme that we present here is that—unlike EC-DSA, RSA, DSA, and other widely used signature schemes—this one is plausibly secure even against *quantum* adversaries. There is ongoing work to standardize signature schemes secure against quantum adversaries; see `https://csrc.nist.gov/projects/pqc-dig-sig`

5.3 Constructing a Signature Scheme for Signing a Single Bit

This signature scheme is not useful on its own, and is given only as a step towards the final construction. It uses as a building block a one-way function $f : \mathcal{X} \to \mathcal{Y}$. Recall that f a one-way function if it is easy to compute but hard to invert; namely there is an efficient algorithm that given $x \in \mathcal{X}$ outputs $f(x)$, and at the same time any efficient algorithm given $y = f(x)$ for a random $x \leftarrow \mathcal{X}$, finds an inverse $x' \in \mathcal{X}$ such that $f(x') = f(x)$ with only negligible probability.

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{vk})$. Choose two random elements x_0, x_1 from $\mathcal{X}$ and let $(y_0, y_1) = (f(x_0), f(x_1))$. Output $\mathsf{sk} = (x_0, x_1)$ and $\mathsf{vk} = (y_0, y_1)$.
- $\mathsf{Sign}(\mathsf{sk}, b) \to \sigma$. Parse $\mathsf{sk} = (x_0, x_1)$ and output $\sigma = x_b$,
- $\mathsf{Ver}(\mathsf{vk}, b, \sigma) \to \{0, 1\}$. Parse $\mathsf{vk} = (y_0, y_1)$ and output 1 if and only if $f(\sigma) = y_b$. (Otherwise, the signing routine rejects.)

In this construction, we leave the security parameter λ implicit. To be fully formal, Gen would take λ an input. The one-way function f and its domain $\mathcal{X}$ would both depend on λ. So we would write f_λ and $\mathcal{X}_\lambda$.

5.4 One-time-secure Signatures (Lamport Signatures)

In this section we give a very simple and elegant construction of a one-time-secure digital signature scheme, due to Lamport.[2] The construction is a straightforward generalization of the signature scheme constructed above: Each message is signed bit-by-bit, where each bit is signed using a fresh and independently generated secret key.

[2] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. Oct. 1979.

Before giving the construction, we define one-time security for digital-signature schemes. This signature scheme is not generally useful on its own, but is useful as a building block.

Definition 5.4.1 (Digital signatures: One-Time Security)**.** A digital-signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ over message space $\mathcal{M}$ is *one-time secure* if all efficient adversaries win the following game with negligible probability:

- The challenger generates $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(\lambda)$ and sends vk to the adversary.
- The adversary sends the challenger *single* message $m \in \mathcal{M}$.
- The challenger responds with $\sigma = \mathsf{Sign}(\mathsf{sk}, m)$.
- The adversary outputs (m^*, σ^*).
- The adversary wins the game if $\mathsf{Ver}(\mathsf{vk}, m^*, \sigma^*) = 1$ and $m^* \neq m$.

Lamport signatures. We now construct a one-time secure signature scheme for messages in $\{0,1\}^n$, for some fixed message length $n \in \mathbb{N}$. To do this, we will define the following algorithms, which make use of a one-way function $f\colon \mathcal{X} \to \mathcal{Y}$:

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{vk})$. Choose $2n$ random elements from $\mathcal{X}$, the domain of the one-way function f. Arrange these values in to a $2 \times n$ matrix, which forms the secret signing key sk. The public verification key just consists of the $2n$ images of these values under the one-way function f:

$$\mathsf{sk} \leftarrow \begin{pmatrix} x_{10} & \dots & x_{n0} \\ x_{11} & \dots & x_{n1} \end{pmatrix}, \quad \mathsf{vk} \leftarrow \begin{pmatrix} f(x_{10}) & \dots & f(x_{n0}) \\ f(x_{11}) & \dots & f(x_{n1}) \end{pmatrix}.$$

- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$ outputs $(x_{1m_1}, \dots x_{nm_n})$, where $m_1 \dots m_n$ are the individual bits of the length-n message $m \in \{0,1\}^n$.

- $\mathsf{Ver}(\mathsf{vk}, m, \sigma) \to \{0,1\}$ parses the the message into bits $m = m_1 \dots m_n \in \{0,1\}^n$ and the signature σ into its individual symbols $\sigma = (x_1^*, \dots x_n^*) \in \mathcal{X}^n$. The signing routine accepts if, for all $i \in \{1, \dots, n\}$:

$$f(x_i^*) = \mathsf{vk}_{i,m_i}. \tag{5.1}$$

  In other words, the routine accepts if applying the one-way function f to each symbol of the signature matches the corresponding value in the verification key. (Otherwise, the signing routine rejects.)

This signature scheme has relatively large keys: the verification key, in particular consists of $2n$ values, where each is of length $\Omega(\lambda)$ bits. So the total length is roughly $2n\lambda$ bits—much longer than the n-bit message being signed.

In addition, notice that an adversary who sees signatures on even two messages can forge signatures on messages of its choice. In particular:

- The adversary first asks for a signature on the message $m_0 = 0^n$. It receives $\sigma_0 = (x_{10}, \ldots, x_{n0})$.
- The adversary then then asks for a signature on the message $m_1 = 1^n$. It receives $\sigma_1 = (x_{11}, \ldots, x_{n1})$.
- At this point, the adversary has the entire secret signing key!

However, we will show that this scheme is indeed one-time secure.

Claim. *The Lamport signature scheme is one-time secure under the assumption that f is a one-way function.*

Remember that if $\mathsf{P} = \mathsf{NP}$, one-way functions, and also digital signature schemes, do not exist. So any proof of security of a digital-signature scheme will require some sort of cryptographic assumption.

In cryptography, we generally prove these security claims by *reduction*: we will show that if there exists an efficient adversary $\mathcal{A}$ that breaks the security of our scheme, then we can construct an efficient adversary $\mathcal{B}$ that breaks one of our assumptions. If we do this, we have reached a contradiction to one of our assumptions, so the first adversary cannot exist.

Proof of Claim. Suppose there exists an adversary $\mathcal{A}$ that wins the one-time-security game of Theorem 5.4.1 with non-negligible probability ϵ. That is, the adversary can produces (m^*, σ^*) such that $\mathsf{Ver}(\mathsf{vk}, m^*, \sigma^*) = 1$ and $m \neq m^*$ given only $\sigma = \mathsf{Sign}(\mathsf{sk}, m)$. We can then construct an adversary $\mathcal{B}$ that can use $\mathcal{A}$ to invert the one-way function.

Lamport's construction shows that if one-way functions exist, then so do digital signatures. Can you show that if digital signatures exist, then so do one-way functions?

In particular, our adversary $\mathcal{B}$ will use algorithm $\mathcal{A}$ as a subroutine to invert the one-way function. We will show that if $\mathcal{A}$ wins in the one-time signature security game often, then algorithm $\mathcal{B}$ will invert the one-way function often, which is a contradiction.

Assume our one-way function is of the form $f\colon \mathcal{X} \to \mathcal{Y}$ and that the Lamport signature scheme is on n-bit messages. The one-way-function adversary $\mathcal{B}$ operates as follows:

- The adversary $\mathcal{B}$ is given a point $y \in \mathcal{Y}$ and its task is to produce a preimage of y under f.
- The adversary $\mathcal{B}$ generates a signing keypair as follows:
  - It runs the key-generation algorithm for the Lamport signature scheme $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}()$.
  - The adversary chooses a random value $i^* \xleftarrow{R} \{1, \ldots, n\}$ and a random bit $\beta^* \xleftarrow{R} \{0, 1\}$.
  - The adversary sets $\mathsf{vk}_{i^*,\beta^*} \leftarrow y$. That is, it inserts the one-way-function point it must invert into a random location in the verification key.

- The adversary then sends the verification key vk to the Lamport-signature adversary $\mathcal{A}$.
- The adversary $\mathcal{A}$ asks for the signature on a message $m = m_1 m_2 \ldots m_n \in \{0,1\}^n$.
- If $m_{i^*} = \beta^*$, then algorithm $\mathcal{B}$ cannot produce a valid signature on the message m and it outputs FAIL.
- Otherwise, the algorithm $\mathcal{B}$ returns the signature $\sigma = (\mathsf{sk}_{1,m_1}, \ldots, \mathsf{sk}_{n,m_n}) \in \mathcal{X}^n$ to algorithm $\mathcal{A}$.
- Algorithm $\mathcal{A}$ then produces a forged message-signature pair (m^*, σ^*), where $m \neq m^*$.
- Algorithm $\mathcal{B}$ parses $m^* = m_1^* \ldots m_n^* \in \{0,1\}^n$ and $\sigma^* = \sigma_1^* \ldots \sigma_n^* \in \mathcal{X}^n$. Then:
  - If $m_{i^*} = m_i$, algorithm $\mathcal{B}$ outputs FAIL.
  - Otherwise, algorithm $\mathcal{B}$ outputs $x \leftarrow \sigma_{i^*}^* \in \mathcal{X}$.

First, notice that whenever (m^*, σ^*) is a valid message-signature pair and whenever algorithm $\mathcal{B}$ does not output FAIL, algorithm $\mathcal{B}$ outputs a preimage $x \in \mathcal{X}$ of point $y \in \mathcal{Y}$ under the one-way function f. That is because, by the verification relation (5.1) for Lamport signatures,

$$f(x) = f(\sigma_{i^*}^*) = \mathsf{vk}_{i^*, m_{i^*}^*} = \mathsf{vk}_{i^*, 1-m_i} = \mathsf{vk}_{i^*, \beta^*} = y.$$

Now, we must show that algorithm $\mathcal{B}$ does not output FAIL too often. Since algorithm $\mathcal{B}$ chooses the values i^* and β^* at random, and since the adversary $\mathcal{A}$ behavior is *independent* of these values, we can say:

- the probability of the first failure event is $1/2$, since there are two possible choices of m_{i^*} and only one of these is bad, and
- the probability of the second failure event is at most $1/n$, since m and m^* must differ in at least one of n bits, and there is a $1/n$ probability that this differing bit is at index i^*.

The events that $\mathcal{A}$ breaks the signature scheme and that either of these failures occur are all *independent*. Then if $\mathcal{A}$ breaks the one-way function with probability ϵ, our one-way-function adversary $\mathcal{B}$ inverts the one-way function with probability

$$\epsilon_{\text{one-way}} = \epsilon \cdot \frac{1}{2} \cdot \frac{1}{n}.$$

The probability of either bad is at most $1/2 + 1/n$, by the union bound. Therefore if algorithm $\mathcal{A}$ breaks one-time security of Lamport's scheme with probability ϵ, If ϵ is non-negligible, then $\epsilon_{\text{one-way}} = \epsilon/2n$ is also non-negligible, and we have a contradiction. □

5.5 A one-time signature scheme for arbitrary-length messages

In the Lamport signature scheme (Section 5.4), the length of the keys scales with the size of the message being signed. To adapt our scheme from above into a scheme that works on arbitrary-length messages without the key growing arbitrarily large, we will use a strategy called *hash-and-sign*. In essence, the signing algorithm will pass the message through a hash function to generate a fixed-size digest before applying a signature scheme that works only on fixed-length messages.

Essentially all signature schemes used in practice use this hash-and-sign construction.

This paradigm is called "hash and sign," and is very common. In practice, hashing is computationally cheap operation while signing turns out to be computationally relatively expensive. So it is common to hash a message before signing it in order to reduce the size of the message that must be signed.

The following claim gives the general construction:

Claim (Hash-and-sign paradigm)**.** *Given a collision-resistant hash function* $h : \{0,1\}^* \to \{0,1\}^n$ *and a signature scheme* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *for message space* $\mathcal{M} = \{0,1\}^n$ *(such as the one in Section 5.4), there exists a signature scheme* $(\mathsf{Gen}', \mathsf{Sign}', \mathsf{Ver}')$ *for* $\mathcal{M}' = \{0,1\}^*$ *as follows:*

- $\mathsf{Gen}' := \mathsf{Gen}$. *The key-generation algorithm is unchanged.*
- $\mathsf{Sign}'(\mathsf{sk}, m) := \mathsf{Sign}(sk, h(m))$. *We hash the message using the hash function h before passing the hashed message to the original signing function.*
- $\mathsf{Ver}'(\mathsf{vk}, m, \sigma) := \mathsf{Ver}(\mathsf{vk}, h(m), \sigma)$. *We use the original* Ver *to check that the tag matches* hash *of the original message.*

Security Intuition. Suppose that there exists an efficiency adversary that breaks $(\mathsf{Gen}', \mathsf{Sign}', \mathsf{Ver}')$. In particular, given $((m_1, \sigma_1, \ldots, (m_t, \sigma_t))$, the adversary is able to construct a valid message-signature pair (m^*, σ^*) such that $m^* \notin \{m_1, \ldots, m_t\}$. There are then two cases:

1. $h(m^*) \in \{h(m_1), \ldots, h(m_t)\}$. If this is the case, there is some $i \in [t]$ such that $h(m^*) = h(m_i)$. However, h is collision-resistant as in the definition, so this is a contradiction!
2. $h(m^*) \notin \{h(m_1), \ldots, h(m_t)\}$. Since the message that we pass to the underlying signature scheme is $h(m)$, this means that the adversary has found a valid signature for $h(m)$ under the original scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ after seeing only signatures of $h(m_1) \neq h(m), \ldots, h(m_t) \neq h(m_t)$. This breaks the security of the underlying signature scheme, which is a contradiction!

Application to Lamport. We can apply the hash-and-sign paradigm to the Lamport signature scheme from Section 5.4. We can fix our input to the Lamport scheme at, for example, 256 bits, and then run messages through a hash function that outputs 256 bits before passing them to the Lamport scheme. This gives a *one-time-secure* signature scheme for messages of arbitrary length.

Security implications of hash and sign. In practice, hash-and-sign can actually *increase* the security of our signature scheme, in a certain sense. As shown in case 2 above, it is absolutely crucial that the hash function used is collision-resistant: if not, an adversary can find messages that cause collisions, and then a signature for one message will also be a valid signature for the other. However, in practice we often think of hash functions like SHA256 as behaving like *random oracles*. That is, for a hash function $h\colon \{0,1\}^* \to \{0,1\}^\lambda$ and a string $x \in \{0,1\}^*$ we think of the value $h(x)$ as being an independently sampled and uniformly random value from the co-domain of the hash function, $\{0,1\}^\lambda$. (Of course, a real-world hash function is *never actually* a random oracle. A random oracle from $h\colon \{0,1\}^* \to \{0,1\}^\lambda$ would take infinitely many bits to describe, while real-world hash functions have finite size (and polynomial-size descriptions).)

Recall that we faced a similar problem in our MAC construction. Why not use hash and MAC there? The reason is that we used AES as our PRF, which takes input of $\{0,1\}^{128}$. As explained by the birthday paradox, it is possible to find a collision in an output space of size 2^{128} in only time 2^{64}! This does not provide sufficient security for practical use, as it would be quite practical to find collisions. If we had a version of AES that outputted 256 bits, we could indeed apply hash and sign.

Another reason to not use hash and MAC is that MACs can be faster to compute than collision-resistant hash functions.

Recall that the standard security definition for digital signatures (Theorem 5.1.3) allows the attacker to request signatures on messages of its choice. If we pass a message through a hash function before signing it using an underlying signature scheme scheme, however, we effectively randomize the message—the adversary can no longer control the input to the underlying signature scheme. This allows us to define another meaningful definition of security:

Definition 5.5.1 (Digital signatures: security against random message attacks)**.** Any efficient adversary given the public verification key and a list of random message-signature pairs $((m_1,\sigma_1),\ldots,(m_t,\sigma_t))$ cannot generate a forged message-signature pair (m^*,σ^*) such that $\mathsf{Ver}(\mathsf{vk},m^*,\sigma^*) = 1$ and $m^* \notin \{m_1,\ldots,m_t\}$.

Note that this definition is *not* good enough on its own—an adversary often does have the ability to generate signatures for messages of his choice. However, paired with a hash function modeled as a random oracle, this definition becomes very useful—if the inputs are passed through the hash function before they are passed to the signature scheme, they become effectively random. We can even relax the definition further without losing practicality: by the same logic, with hash function in front of the signature scheme, what the adversary needs to sign is really not a message of their choise, but is the *hash* of a message of their choice—effectively a random value.

Definition 5.5.2 (Digital signatures: random security against random message attacks)**.** Any efficient adversary given vk and a list of random message-signature pairs $((m_1, \sigma_1), \ldots, (m_t, \sigma_t))$ and random $m^* \notin \{m_1, \ldots, m_t\}$ cannot generate σ^* such that $\mathsf{Ver}(\mathsf{vk}, m^*, \sigma^*) = 1$.

It is possible to formally argue that given a signature scheme satisfying Theorem 5.5.2 and a random oracle, we can construct a scheme satisfying existential unforgeability under chosen message attacks.

5.6 From one-time security to many-time security

After applying the hash-and-sign strategy above to our Lamport scheme, we have a signature scheme that is one-time secure for messages of arbitrary length. In order for the scheme to be useful and satisfy our security definition, we need to be able to sign polynomially many messages with a single key pair. To do this, we will use a construction very similar to the Merkle tree construction we have seen before.

Informally, we will build up a binary tree of Lamport keys of depth 256. The signing key in each of the 2^{256} leaf nodes will be used to actually sign messages; we will use a random leaf node to sign a message, so the fact that there are 2^{256} of them means that the probability of accidentally choosing the same leaf twice is negligible (2^{-128}). The signing key in an intermediate nodes (and in the root) will be used to sign the (public) verification keys of the two corresponding child nodes in the tree. Fig. 5.1 shows a sketch of this tree.

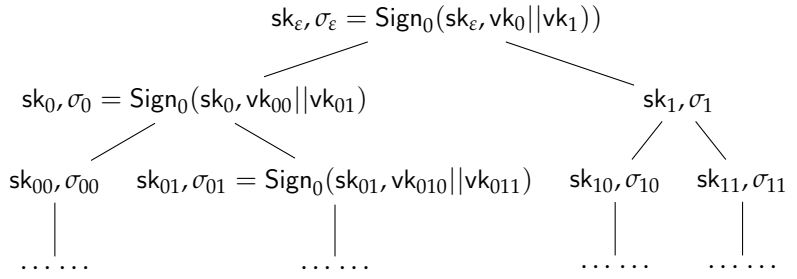


Figure 5.1: Sketch of the tree of Lamport signature keys used for the many-time secure signature construction.

Importantly, every signing key in the tree will be used to sign only one message ever: the key at non-leaf nodes will only ever sign the pair of its children, and the key at leaf nodes will only ever be used to sign a single message.

The signature, in this scheme, will consist of the signatures and verification keys along the path from the root to the chosen leaf node. The signature will also include information about which child node

was chosen.

This tree, of course, is impractically large, but we can solve that problem by lazily constructing it using a pseudorandom function. That is, instead of actually building up all of the leaves of the tree, we will build up the tree (i.e., generate the signing keys, verification keys, and signatures) on-demand, and furthermore, we will build it in a deterministic way using the pseudorandom function, so that we don't have to remember what parts we might have already computed in the past.

To make the construction more precise, we will assume that we are given:

- a pseudorandom function f with keyspace $\mathcal{K}$, and
- a one-time secure signature scheme $(\mathsf{Gen}_0, \mathsf{Sign}_0, \mathsf{Ver}_0)$.

We will need the ability to run the (non-deterministic) Gen_0 algorithm on specific randomness, so as to make it deterministic. For a PRF key $k \in \mathcal{K}$ and string s, we will write $\mathsf{Gen}_0^{F(k,s)}()$ to indicate the process of running the key-generation algorithm Gen_0 using randomness derived from the output of the PRF $F(k,s)$. We will assume the use of SHA256 (with a 256-bit digest length) as a collision-resistant hash function. Using these building blocks, we will construct a many-time secure signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ for arbitrary-length messages (i.e., on message space $\{0,1\}^*$), where all of the algorithms are efficient (poly$(\cdot)$ running time).

Our construction is as follows:

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{vk})$:
  - Sample a fresh PRF key $k \xleftarrow{R} \mathcal{K}$.
  - Set $(\mathsf{sk}_\epsilon, \mathsf{vk}_\epsilon) \leftarrow \mathsf{Gen}_0^{F(k,"")}()$.
  - Output $(\mathsf{sk}, \mathsf{vk}) \leftarrow (k, \mathsf{vk}_\epsilon)$.
- $\mathsf{Sign}_t(k, m) \to \sigma$:
  - Choose a random 256-bit value $r = (r_1 \ldots r_{256}) \in \{0,1\}^{256}$).
  - Compute $(\mathsf{sk}_r, \mathsf{vk}_r) \leftarrow \mathsf{Gen}_0^{F(k,r)}()$
  - Compute $\sigma_r \leftarrow \mathsf{Sign}_0(\mathsf{sk}_r, \mathrm{SHA256}(m))$.
  - Compute σ_ε, vk_0, vk_1, σ_{r_1}, $\mathsf{vk}_{r_1 0}$, $\mathsf{vk}_{r_1 1}$, $\sigma_{r_1 r_2}$, …, $\sigma_{r_1 r_2 \ldots r_{255}}$, $\mathsf{vk}_{r_1 r_2 \ldots r_{255} 0}$, $\mathsf{vk}_{r_1 r_2 \ldots r_{255} 1}$ as shown in Fig. 5.1.
  - Output $\sigma \leftarrow (r, \sigma_\varepsilon, \mathsf{vk}_0, \mathsf{vk}_1, \sigma_{r_1}, \mathsf{vk}_{r_1 0}, \mathsf{vk}_{r_1 1}, \ldots, \sigma_r)$.
- $\mathsf{Ver}_t(\mathsf{vk}_\epsilon, m, \sigma) \to \{0,1\}$:
  - Parse $(r, \sigma_\varepsilon, \mathsf{vk}_0, \mathsf{vk}_1, \sigma_{r_1}, \mathsf{vk}_{r_1 0}, \mathsf{vk}_{r_1 1}, \ldots, \sigma_r) \leftarrow \sigma$.

- Output "1" if and only if
  - $\mathsf{Ver}_0(\mathsf{vk}_r, \sigma_r, \mathrm{SHA256}(m)) = 1$ and
  - $\mathsf{Ver}_0(\mathsf{vk}_x, \sigma_x, \mathsf{vk}_{x0}||\mathsf{vk}_{x1}) = 1$ for every prefix x of r (from ε to $r_1 r_2 \ldots r_{255}$).

5.7 *Choosing Signature Schemes*

The signature scheme we presented in Section 5.6 is not particularly efficient in terms of signature size.

Algorithm	vk size	σ size	signatures/sec	verifications/sec
SPHINCS+-128	32 B	8000 B	5	750
RSA 2048	256 B	256 B	2,000	50,000
ECDSA256	32 B	64 B	42,000	14,000

Table 5.1: Statistics about various signature schemes used in practice

Many deployed systems today use the ECDSA256 signature scheme. Legacy application still use RSA signatures, though because of their relatively large public-key and signature sizes, few new applications use these schemes.

Hashing is much, much faster than signing—the commonly used SHA256 hashing algorithm can compute around 10,000,000 hashes per second. This is one reason the hash-and-sign paradigm is so useful.

6
RSA Signatures

In this chapter, we will discuss the RSA digital-signature scheme. The RSA paper[1] was tremendously influential because it gave the first constructions of digital signatures and public-key encryption. (We will talk about public-key encryption in detail later on.)

[1] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

The RSA cryptosystem is going out of style for a few reasons: generating RSA keys is relatively expensive and the keys are relatively large (4096 bits for RSA versus 256 bits for more modern elliptic-curve-based cryptosystems). In addition, a large-scale quantum computer could—in theory, at least—break RSA-style cryptosystems.

The RSA cryptosystem is worth studying for a few reasons:

- RSA's security is related to the problem of factoring large integers, which is (arguably) the most natural "hard" computational problem out there.
- RSA gives the only known instantiation of a *trapdoor one-way permutation*, which we will define shortly.
- RSA has a number of esoteric properties that are useful for advanced cryptographic constructions. For example, it gives a "group of unknown order." See Boneh-Shoup, Chapter 10.9 for details.
- RSA signatures are used on the vast majority of public-key certificates today.[2]

  The most commonly used type of RSA signatures ("PKCS #1 v1.5") is more complicated—and no more secure—than the construction we describe here, but that construction is still used for historical reasons.

[2] As of today, around 94% of certificates in the Certificate Transparency logs use RSA signatures: `https://ct.cloudflare.com/`.

6.1 Trapdoor one-way permutations

RSA implements a *trapdoor one-way function*. Informally, a trapdoor one-way function is a function that is easy to compute in the forward

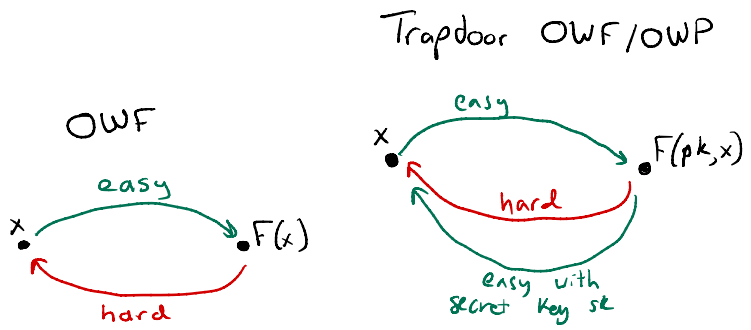


Figure 6.1: A one-way function (at left) is hard to invert on random inputs. A trapdoor one-way function/permutation (at right) is a function keyed by a public key. The function is easy to invert given the secret key and is hard to invert otherwise.

direction but that is hard to invert *except* to someone knowing a secret key. So it is like a one-way function with a "trapdoor" that allows efficient inversion.

RSA actually implements a trapdoor one-way *permutation*—that is, it maps an input space onto itself with no collisions.

6.1.1 Definition

Formally, a trapdoor one-way permutation over input space $\mathcal{X}$ is a triple of efficient algorithms:

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$. The key-generation algorithm takes as input the security parameter $\lambda \in \mathbb{N}$, expressed as a unary string, and outputs a secret key and a public key.
- $F(\mathsf{pk}, x) \to y$. The evaluation algorithm F takes as input the public key pk and an input $x \in \mathcal{X}$, and outputs a value $y \in \mathcal{X}$.
- $I(\mathsf{sk}, y) \to x'$. The inversion algorithm I takes as input the secret key sk and a point $y \in \mathcal{X}$, and outputs its inverse $x \in \mathcal{X}$.

If we wanted to be completely formal, the input space would be parameterized by the security parameter λ. So we would have a family of input spaces $\{\mathcal{X}_\lambda\}_{\lambda \in \mathbb{N}}$—one for each choice of λ. This way the input space can grow with λ.
In the RSA construction, the input space $\mathcal{X}$ depends on the public key, but we elide that technical detail here.

Correctness. Informally, we want that for keypairs $(\mathsf{sk}, \mathsf{pk})$ output by Gen, we have that $F(\mathsf{pk}, \cdot)$ and $I(\mathsf{sk}, \cdot)$ are inverses of each other. More formally, for all $\lambda \in \mathbb{N}$, $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$, and $x \in \mathcal{X}$, we require:

$$I(\mathsf{sk}, F(\mathsf{pk}, x)) = x.$$

Security. Security requires that $F(\mathsf{pk}, \cdot)$ is hard to invert (in the sense of a one-way function) on a randomly sampled input in the input space $\mathcal{X}$, even when the adversary is given the public key pk. That is, for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathrm{negl}(\cdot)$ such that

$$\Pr\left[\mathcal{A}(\mathsf{pk}, F(\mathsf{pk}, x)) = x : \begin{matrix}(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda) \\ x \xleftarrow{R} \mathcal{X}\end{matrix}\right] \leq \mathrm{negl}(\lambda).$$

When we use the RSA cryptosystem, we make the assumption that the RSA function is hard to invert given only the public key:

Definition 6.1.1 (RSA Assumption)**.** The RSA function (Gen, F, I) is a trapdoor one-way permutation.

IMPORTANT: Just as a one-way function is only hard to invert on a *randomly sampled input*, a trapdoor one-way function is only hard to invert on a randomly sampled input. Many of the cryptographic failures of RSA come from assuming that the RSA one-way function is hard to invert on non-random inputs.

6.1.2 Digital signatures from trapdoor one-way permutations

This construction is called "full-domain hash."[3] The construction makes use of a hash function H and resulting signature scheme is secure, provided that we model the hash function H as a "random oracle."

[3] Mihir Bellare and Phillip Rogaway. "Random oracles are practical: A paradigm for designing efficient protocols". In: *ACM Conference on Computer and Communications Security*. 1993.

In other words, to argue security, it is not sufficient to show that H is, for example, collision resistant. Instead, we can only prove security provided that we pretend that H is a truly random function—i.e., in the random-oracle model. When we instantiate the hash function H with some concrete cryptographic hash function, such as SHA256, we hope that the resulting signature scheme is still secure. In practice, this approach works quite well.

One way to think about it is that if a signature scheme is secure in the random-oracle model, then the concrete signature scheme is in some sense secure against attacks that do not exploit the peculiarities of the hash function.

In the construction, we use:

- a trapdoor one-way permutation (Gen, F, I), and
- a hash function $H\colon \{0,1\}^* \to \mathcal{X}$, which we model as a random oracle in the security analysis.

Construction. We construct a digital-signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ as follows:

- Gen – Just run the key-generation algorithm for the trapdoor one-way permutation.
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$. Hash the message down to an element h of the input space $\mathcal{X}$ of the trapdoor one-way permutation using the hash function H. Then invert the trapdoor one-way permutation at that point:

  - Compute $h \leftarrow H(m)$.
  - Output $\sigma \leftarrow I(\mathsf{sk}, h)$.
- $\mathsf{Ver}(\mathsf{pk}, m, \sigma) \to \{0,1\}$.
  - Compute $h' \leftarrow H(m)$.
  - Accept if $F(\mathsf{pk}, \sigma) = h'$.

Notice that the use of a hash function here is **critical** to security, since (in the random oracle) it means that forging a signature is as hard as inverting F on a random point in its co-domain. Without the hash function, forging a signature is only as hard as inverting F on an attacker-chosen point in its co-domain, which could be easy.

Correctness. For all $\lambda \in \mathbb{N}$, $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$, and $m \in \{0,1\}^*$, we have:

$$\begin{aligned}\mathsf{Ver}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) &= 1\{F(\mathsf{pk}, I(\mathsf{sk}, H(m))) = H(m)\} \\ &= 1\{I(\mathsf{sk}, F(\mathsf{pk}, I(\mathsf{sk}, H(m)))) = I(\mathsf{sk}, H(m))\}\end{aligned}$$

and by correctness of the trapdoor one-way permutation:

$$= 1\{I(\mathsf{sk}, H(m)) = I(\mathsf{sk}, H(m))\} = 1.$$

Security. The intuition here is that if the adversary cannot invert F, it cannot find the preimage of $H(m)$ under F for any message on which it has not seen a signature. See Boneh-Shoup Chapter 13.3 for the full security analysis.

6.2 *The RSA construction: Forward direction*

The algorithms for key-generation and for evaluating the RSA permutation in the forward direction are not too complicated.

In what follows, we present RSA with public exponent $e = 5$. The same construction works with many other choices of e, just by replacing all of the "5"s below with some other small prime: 3, 7, 13, etc. A popular choice of the public exponent e in practice is $e = 2^{16} + 1$. The complexity of computing the RSA function in the forward direction scales with the size of e, so we prefer small choices of e.

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$.
  - Sample two random λ-bit primes p and q such that $p \equiv q \equiv 4 \pmod 5$.
  - Set $N \leftarrow p \cdot q$.

In practice, we usually take the bitlength of primes to be $\lambda = 1024$ or $\lambda = 2048$.

Standard RSA implementations require the weaker condition that the public exponent e shares no prime factors with $p-1$ and $q-1$. Using the stronger condition here simplifies the inversion algorithm.

  - Output $\mathsf{sk} \leftarrow (p, q)$, and $\mathsf{pk} = N$.

- $F(\mathsf{pk} = N, x) \to y$.
  - The input space for the RSA function is $\mathcal{X} = \mathbb{Z}_N^*$—the set of elements in $\{0, 1, 2, \ldots, N-1\}$ relatively prime to N.
  - Output $y \leftarrow x^5 \bmod N$.

Remark 6.2.1. The key-generation algorithm relies on us being able to sample large random primes. One perhaps surprising fact is that there are many many large primes. In particular, if you pick a random λ-bit number, the probability that it is prime is roughly $1/\lambda$.

For more on this, look up the Prime Number Theorem.

We can sample a random λ-bit prime by just picking random integers in the range $[2^\lambda, 2^{\lambda+1})$ until we find a prime. We can test for primality in $\approx \lambda^4$ time using the Miller-Rabin primality test. We also need that there are infinitely many primes congruent to 4 mod 5, but fortunately there are. Generating RSA keys is expensive—it can take a few seconds even on a modern machine.

Notice that computing the RSA function in the forward direction is relatively fast: it just requires three multiplications modulo a 2048-bit number N. That is, to compute $x^5 \bmod N$, we compute:

$$(x^2)^2 \cdot x = x^5 \mod N.$$

Before describing the RSA inversion algorithm, we discuss why the RSA trapdoor one-way permutation should be hard to invert without the secret key.

6.2.1 *Why should the RSA function be hard to invert?*

To invert the RSA function, the attacker's is effectively given a value $y \xleftarrow{R} \mathbb{Z}_N$ and must find a value x such that $x^5 = y \bmod N$. Or, put another way, the attacker's task is essentially the following:

- **Given:** A polynomial $p(X) := X^5 - y \in \mathbb{Z}_N[X]$, for $y \xleftarrow{R} \mathbb{Z}_N$.
- **Find:** A value $x \in \mathbb{Z}_N$ such that $p(x) = 0 \in \mathbb{Z}_N$.

So the attacker must find the root of a polynomial modulo a composite integer N.

The premise of RSA-style cryptosystems is that we only know of essentially two ways to find roots of polynomials modulo N:

- **Factor N into primes** and find a root modulo each of the primes. (We will say more on this in a moment.) Since the best algorithms for factoring run in time roughly $2^{\sqrt[3]{\log N}} = 2^{\sqrt[3]{\lambda}}$, this approach is infeasible at present without knowing the factorization of N.

In Chapter A we present a factoring algorithm that runs in sub-exponential time $2^{\sqrt{\log N \log \log N}}$.

- **Find a root over the integers** and reduce it modulo N. For example, it is easy to find a root of polynomials such as:

Actually, it suffices to find a root over the rational numbers, but the distinction isn't important here.

$$\begin{aligned} X + 4 &= 3 && \bmod N, \\ X + 2Y &= 5 && \bmod N, \\ X^2 &= 9 && \bmod N, \text{ and} \\ X^2 - 3x + 2 = (X-2)(X-1) &= 3 && \bmod N. \end{aligned}$$

When $y \xleftarrow{R} \mathbb{Z}_N$, the probability that y is a perfect 5-th power, and thus that there is an integral root to $X^5 - y$, is $\sqrt[5]{N}/N \approx 2^{-4\lambda/5}$, which is negligible in the security parameter λ. So solving this equation over the integers is a dead end.

There are many clever attacks for solving polynomial equations modulo composites that work in certain special cases, but for most purposes these are the two known attacks.

Is inverting the RSA function as hard as factoring the modulus? No one knows—the question has been open since the invention of RSA. We do know that finding roots of certain polynomial equations, such as $p(X) := X^2 - y \bmod N$ for random $y \xleftarrow{R} \mathbb{Z}_N$ *is* as hard as factoring the modulus N. But for RSA-type polynomials, the answer is unclear.

6.3 The RSA construction: Inverse direction

To understand how the inversion algorithm works, we will need some number-theoretic tools.

6.3.1 Tools from number theory

For a natural number N, let $\phi(N)$ denote the number of integers in $\mathbb{Z}_N = \{1, 2, 3, \ldots, N\}$ that are relatively prime to N. When p is prime $\phi(p) = p - 1$. The function $\phi(\cdot)$ is called *Euler's totient function*.

Two natural numbers are *relatively prime* if they share no prime factors.

When $N = pq$ is the product of two distinct primes, $\phi(N) = (p-1)(q-1)$. That is so because all numbers in $\mathbb{Z}_N$ are relatively prime to N except N and the multiples of p and q:

$$p, 2p, 3p, \ldots, (q-1)p, \quad q, 2q, 3q, \ldots, (p-1)q.$$

So there are $N - (q-1) - (p-1) - 1 = (p-1)(q-1)$ numbers in $\mathbb{Z}_N$ relatively prime to N.

Theorem 6.3.1 (Euler's Theorem)**.** *Let N be a natural number. Then for all $a \in \mathbb{Z}_N^*$,*

$$a^{\phi(N)} = 1 \mod N.$$

Proof. Consider the sets $\mathbb{Z}_N^*$ and $\{ax \bmod N \mid x \in \mathbb{Z}_N^*\}$. These sets are equal, so the product of the elements in the two sets is equal. Let

$X \leftarrow \prod_{x \in \mathbb{Z}_N^*} x \bmod N$. Then

$$X = a^{\phi(N)} X \bmod N \quad \Rightarrow \quad 1 = a^{\phi(N)} \bmod N.$$

□

Lemma 6.3.2. *Let p and q be distinct primes congruent to 4 modulo 5. Define the integer $d = \frac{\phi(N)-4}{5} + 1$. Then $5d \equiv 1 \bmod \phi(N)$.*

Proof. Observe that

$$p \equiv 4 \bmod 5 \quad \Rightarrow \quad \phi(N) - 4 \equiv 0 \bmod 5,$$

so $\frac{\phi(N)-4}{5}$ is an integer and thus d is well defined. Then $5d = \phi(N) - 4 + 5 = 1 \bmod \phi(N)$. □

6.3.2 *Inverting the RSA function*

With the number theory out of the way, we can now describe how to invert the RSA function. All we have to do is to show how to compute a fifth root of $y \bmod N$.

- $I(\mathsf{sk}, y) \to x$.
  - The secret key sk consists of the prime factors p, q of N. Recall that $\phi(N) = (p-1)(q-1)$.
  - Compute the integer $d \leftarrow \frac{\phi(N)-4}{5} + 1$, as in Theorem 6.3.2.
  - Return $y^d \bmod N$.

We sometimes call d the *private exponent* in RSA.

It is not obvious why the inversion algorithm is correct. Say that $y = x^5 \bmod N$. Then:

$$\begin{aligned}
y^d &= (x^5)^d && \bmod N && \\
&= x^{5d} && \bmod N && \\
&= x^{k \cdot \phi(N)+1} && \bmod N, && \text{for some } k \in \mathbb{Z}, \text{ by Theorem 6.3.2} \\
&= x \cdot (x^{\phi(N)})^k && \bmod N && \\
&= x && \bmod N, && \text{by Theorem 6.3.1.}
\end{aligned}$$

We could write $5d = k\phi(N) + 1$ because from Theorem 6.3.2, we know that $5d \equiv 1 \bmod \phi(N)$.

Using other public exponents. For our RSA-inversion algorithm to work, we need only to compute the multiplicative inverse e modulo $\phi(N)$. That is, we need to compute an integers d such that $ed \equiv 1 \bmod \phi(N)$. Such an inverse always exists when e and $\phi(N) =$

$(p-1)(q-1)$ are relatively prime. RSA implementations typically use the extended Euclidean algorithm to compute the multiplicative inverse of e modulo $\phi(N)$. That algorithm is more general, but the one we used in Theorem 6.3.2 is simpler and is self-contained.

Inverting RSA is easy on a negligible fraction of points. Recall the RSA is If the preimage under the RSA function of a point y is very very small, then If $x < N^{1/5}$, then computing x given $y = x^5 \bmod N$ is *easy*.

Is inverting RSA as hard as factoring the modulus N? The inversion algorithm we showed here requires knowing the prime factors of the modulus N. Inverting RSA is thus *no harder than* factoring N.

Is inverting RSA *as hard as* factoring N? In particular, if we have an efficient algorithm $\mathcal{A}$ that inverts RSA, can we use $\mathcal{A}$ to factor the modulus N? No one knows!

Most cryptographers, I would guess, believe that inverting the RSA function is as hard as factoring. But for all we know, it could be that computing fifth roots modulo N is *easier* than factoring the modulus.

7
Public-key Infrastructure

In the last chapter, we discussed digital signatures, which allow us to authenticate messages without a shared secret. For example, if I have the public signature-verification key of the university dean, I can verify that signed emails from the dean really came from her and not from someone pretending to be her. But to verify the signature on the dean's message, I need to know her signature-verification key vk. How can I (the recipient) obtain this verification key without a secure channel to the dean (the sender)?

Unfortunately, there are no perfect solutions to this problem. In this section, we will discuss some of the approaches that we use in practice.

7.1 Public-key infrastructure

The goal of a public-key infrastructure is to facilitate the mapping of **human-intelligible names** to **signature-verification keys**. Examples of human-intelligible names that we map to keys are: email addresses, domain names, legal entities, phone numbers, and usernames (e.g., within a company).

We can think of the public-key infrastructure as implementing the following (grossly simplified) API:

$$\mathsf{IsKeyFor}(\mathsf{vk}, \mathtt{name}) \to \{0,1\}.$$

That is, given a verification key vk and a name $\mathtt{name}$, the public-key infrastructure gives a way to check whether this mapping is valid.

We now discuss some ways to implement this API.

7.2 Option 1: Use verification keys as names

One option is to just refer to everyone by the bytes of their signature-verification key. This way, there is no need to do a messy name-to-verification-key translation at all.

This is not practical for humans generally: it would be difficult to remember your friends' names if you had to call them by random 32-byte strings! However, some digital services such as Bitcoin indeed use verification keys as identities: when you want to transfer Bitcoin to someone, you send the coins to an account identified by their public key. The public key *is* name of that account.

Using keys as names has a two major problems:

- Verification keys are hard to remember. Things like email addresses, domain names, kerberos usernames, phone numbers, and so on, are much easier for humans to remember.
- There is no way to update the name-to-key mapping. If you lose the secret key associated with your name/account, there is no way to "update" the key to a new value. In practice, people lose their secret keys all the time, so supporting key updates is critical in most systems.

7.3 Trust on first use (TOFU)

Another strategy is to avoid having any global mapping from names to verification keys. Instead, a client can just accept the first verification key that it sees associated with a given name. The secure shell system (`SSH`) uses TOFU for key management by default.

In particular, the key-validation logic looks like this:

> keymap $\leftarrow \{\}$.
>
> IsKeyFor(vk, name) :
> - If keymap[name] is undefined:
>   - Set keymap[name] $\leftarrow$ vk.
>   - Return true..
> - Else: Return keymap[name] $==$ vk.

That is, the client will accept the *first* verification key it sees associated with a particular name. Later on, the client will only accept the same verification key for that name.

TOFU is very simple to implement and provides a meaningful security guarantees. There are two drawbacks:

- If the first key that client receives for a particular name is incorrect/attacker-generated, the attacker can forge signatures under that name.
- It is not clear with TOFU how to handle key updates. In most systems that use TOFU, whenever the sender's key changes, the system notifies the user and allows them to accept or reject the

new key. The burden is then on the user to figure out whether the sender really did change their signing keypair, or whether there is an attack in progress.

7.4 Certificates

Another option is to rely on a few parties to manage the mapping of names to public keys. These entities are called *Certificate Authorities* (CAs). Your operating system and web browser typically come bundled with a set of roughly 100 public signature-verification keys, owned by each of 100 CAs.

Whenever the owner of website `example.com`, for example, generates a new public key $\mathsf{vk}_{\texttt{example.com}}$, the website owner can ask a certificate authority to certify that $\mathsf{vk}_{\texttt{example.com}}$ really belongs to `example.com`. The certificate authority does this by signing the pair $(\texttt{example.com}, \mathsf{vk}_{\texttt{example.com}})$ using its own signing keypair vk_{CA} to generate a signature σ_{CA}. This signed attestation $(\texttt{example.com}, \mathsf{vk}_{\texttt{example.com}}, \sigma_{\text{CA}})$ is called a *certificate*.

In practice the structure of certificates is much more complicated than we are showing here, and include all sorts of additional metadata. But the basic idea is the same as we describe here.

When a client connects to `example.com`, the server at `example.com` will supply the client with the certificate $(\texttt{example.com}, \mathsf{vk}_{\texttt{example.com}}, \sigma_{\text{CA}})$. As long as this certificate was signed by a CA that the client trusts (i.e., a CA for which the client has a verification key), the client can validate the certificate and conclude that the verification key $\mathsf{vk}_{\texttt{example.com}}$ really belongs to `example.com`.

In pseudocode the logic for verifying certificates looks like this:

$\mathsf{caKeys} \leftarrow \{\mathsf{vk}_{\text{Verisign}}, \mathsf{vk}_{\text{Let's Encrypt}}, \ldots\}$.

$\mathsf{IsKeyFor}((\mathsf{vk}, \sigma), \mathsf{name})$:

- For each vk_{CA} in caKeys:
  - If $\mathsf{Ver}(\mathsf{vk}_{\text{CA}}, (\mathsf{name}, \mathsf{vk}), \sigma) = 1$: Return true.
- Return false.

A very nice feature of certificate-based public-key infrastructure is that the client does not need to communicate with the CA to validate a name-to-key mapping. The client only needs to perform one signature-verification check.

Certificates in practice works quite well:

- The client only needs to store ≈ 100 CA verification keys, and yet the client can validate the name-to-key mappings for millions of websites.
- A client can choose which CAs to trust (though in practice, clients typically delegate this decision to software vendors).
- The client never needs to interact with the CA.

In order to use TLS on a website you own, you need to convince one of the certificate authorities to give you a certificate—i.e., to sign your $(\mathsf{name}, \mathsf{vk})$ pair. To do so, the CA will have some protocol to follow—typically, you will send your $(\mathsf{name}, \mathsf{vk})$ pair to the CA, who will then ask you to verify that you own the name somehow. This is called *domain validation*. In the case of web certificates, the CA may verify ownership by requiring you to upload a file file to your server, to add a new DNS record with a random value, or something similar. Once the CA is convinced that you own the domain, the CA will reply with a certificate: a signature over the tuple $(\mathsf{name}, \mathsf{vk})$. This $(\mathsf{name}, \mathsf{vk}, \sigma_{\text{CA}})$.

However, certificates still have some drawbacks:

- If an attacker compromises *any* CA, they can generate certificates for any domain.
- Certificate authorities often perform quite minimal validation of domain ownership.
- If a server's private key gets stolen, there is no great plan for *revoking* or updating a name-to-key mapping.

7.4.1 Domain Validation

Domain Validated (DV) certificates are quite common. CA validation uses a simple technical check for domain ownership. For example, the CA asks the requester amazon.com to put a nonce in a file on an amazon.com server. Then, the CA retrieves that file, and checks the nonce. The (reasonable) assumption is that only the server owner could have created such a file. Alternately, the CA could ask the requester to create certain Domain Name Services (DNS) records under amazon.com, e.g., acme-challenge.amazon.com.

This check is often automated, and is fast, convenient, cheap. An example DV CA is **Let's Encrypt**. It is easy to use and free; the requester doesn't have to be a real company.

DV certificates only have low-level DNS ownership guarantees: "CA validated that owner of certificate owns amazon.com". However, the guarantee closely matches what CAs can be expected to validate in real life.

A standard protocol for validating domain ownership is ACME[1].

[1] https://tools.ietf.org/html/rfc8555#section-8.3

7.4.2 Revocation

In many cases, a CA will want to delete or change a name-to-key mapping. This process is called *certificate revocation*. There are several possible reasons for this:

- The owner of a verification key may have their corresponding secret key be lost or stolen.
- A company may want to rotate keys, for example to update to a new cryptographic algorithm.
- A website may go out of business and another entity buys their domain name.
- Software bugs may lead a user to generate an insecure keypair that they later want to revoke.[2]

[2] Scott Yilek et al. "When private keys are public: Results from the 2008 Debian OpenSSL vulnerability". In: *SIGCOMM*. 2009; Matus Nemec et al. "The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli". In: *CCS*. 2017.

In a scheme that uses certificates, this seems like a hard problem: since there is no interaction with the CA to verify a certificate, there is no way for a CA to "take back" a certificate. There are again no

excellent solutions to this, but there are a few strategies used in practice.

Expiration Dates One pragmatic way to handle revocation is to add an expiration date to each generated certificate—if this expiration date has passed, the client will reject the certificate. This way, a server will need to re-authenticate to the CA that they own the name that they claim to own periodically. So, for example, an attacker that steals a website's secret key will only be able to use it until the certificate expires. In practice, certificates used on the Internet typically expiration dates between 90 days and 1-2 years.

Software Updates Another solution is for the browser (or client, more generally) to maintain a list of revoked certificates. On every connection, the browser will check whether the provided certificate is in this local revocation list and refuse it if so. Since browsers today check for updates very frequently, this strategy can respond to a stolen secret key quickly. However, there is a large storage cost since now every browser needs to store this (potentially large) list of revoked certificates.

CA Revocation List To avoid depending on browser manufacturers to update this revocation list, another strategy is to ask the CA for it directly. One way to do this is similar to the above: periodically query the CA to download its updated revocation list and check that each certificate is not in this list. This method has fallen out of favor, in part because clients (e.g., behind corporate firewalls) cannot connect directly to the CAs to download these revocation lists.

Part II

Transport Security

8
Introduction to Encryption

So far, we have explored methods for authenticating data—verifying that it has not been modified in transit. However, the integrity-protection mechanisms we have discussed provide no *confidentiality*: a network eavesdropper can still view everything that we send over the network.

In this chapter, we will discuss *encryption*, which allow two parties to exchange messages over an insecure network while hiding the contents of their communications.

We will cover encryption in a sequence of steps:

- First, we will construct an encryption scheme with a *weak form of security* for *fixed-length messages* for settings in which the sender and recipient have a *shared secret key*.
- Next, we will show how to extend this scheme to support *variable-length messages*.
- Then, we will show how to improve the scheme to have a *strong form of security*.
- Finally, we will show how to implement encryption in settings in which the sender and recipient *have no shared secrets*.

At the conclusion of this part, we will discuss how deployed systems use encryption, and we will think about some problems that encryption does not solve.

8.1 Background

The need for encryption The internet is a massive network of wifi access points, routers, switches, undersea cables, DNS servers, and much more. There are many, many devices for a potential adversary to compromise and many vantage points from which an attacker can observe network traffic. Every single hop your packets take is a potential point of compromise.

To make matters worse, most standard network protocols provide *no authentication or encryption*: in Ethernet, IP, DNS, email, HTTP, and others, an adversary is free to modify and read the traffic we send and receive. Protecting confidentiality typically requires augmenting these standard network protocols with some form of authentication and encryption.

Systems using encryption Encryption shows up in a large number of deployed systems. A few examples are:

- **Messaging apps**, such as WhatsApp, Signal, and iMessage, encrypt traffic between app users so that the server cannot easily read it.
- **Network protocols**, such as SSH and HTTPS use encryption to protect traffic between a service's clients and servers.
- **File-storage systems** use encryption to protect data at rest. So if a thief steals your laptop, they will not easily be able to read the encrypted files on your hard disk.

8.2 Encryption Scheme Syntax

Encryption schemes are defined with respect to a key space $\mathcal{K}$, a message space $\mathcal{M}$, and a ciphertext space $\mathcal{C}$. For now, think of $\mathcal{K} = \mathcal{M} = \{0,1\}^n$ and $\mathcal{C} = \{0,1\}^{2n}$, for a security parameter n. An encryption scheme then consists of two algorithms:

Remember that in practice, we will often take the size of the keyspace $|\mathcal{K}|$ to be at least 2^{128} to prevent brute-force key-guessing attacks.

- $\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$
- $\mathsf{Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$

As we will see later on, the decryption algorithm Dec can sometimes output "FAIL" or $\perp$.

The definition of correctness for an encryption scheme just states that if you encrypt a message m with a key k, and then you decrypt the resulting ciphertext with the same key k, you end up with the same message m that you started with:

Definition 8.2.1 (Encryption Scheme, Correctness)**.** An encryption scheme is correct if, for all keys $k \in \mathcal{K}$ and all messages $m \in \mathcal{M}$, $\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$.

Defining security for encryption scheme is tricky business. Many of the most obvious security definitions are insufficient:

- **Bad definition:** *"An encryption scheme is secure if it is infeasible for an attacker to recover the plaintext message given only a ciphertext.* This definition admits encryption schemes in which the ciphertext leaks half of the plaintext bits.

- **Bad definition:** *"An encryption scheme is secure if it is infeasible for an attacker to recover any bit of the plaintext message given only a ciphertext.* This definition admits encryption schemes in which the ciphertext leaks the parity of the plaintext bits.

The starting-point (weak) security definition we will use is called *indistinguishability under adaptive chosen plaintext attack* (IND-CPA). Intuitively, a scheme is CPA-secure if an attacker cannot tell which of two chosen messages are encrypted, even after seeing encryptions of many attacker-chosen messages.

Definition 8.2.2 (Encryption Scheme, CPA Security (weak))**.** Formally, an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ over message space $\mathcal{M}$ and key space $\mathcal{K}$ is CPA-secure if all efficient adversaries win the following game with probability at most $\frac{1}{2}$ + "negligible:"

- The challenger samples $b \xleftarrow{R} \{0,1\}$ and $k \xleftarrow{R} \mathcal{K}$.
- Polynomially many times: *// Chosen-plaintext queries*
  - The adversary sends the challenger a message $m_i \in \mathcal{M}$
  - The challenger replies with $c_i \leftarrow \mathsf{Enc}(k, m_i)$.
- The adversary then sends two messages $m_0^*, m_1^* \in \mathcal{M}$ to the challenger. (We require $|m_0^*| = |m_1^*|$.)
- The challenger replies with $c^* \leftarrow \mathsf{Enc}(k, m_b^*)$.
- The adversary outputs a value $b' \in \{0,1\}$. The adversary wins if $b = b'$.

Standard encryption systems do not hide the length of the message being encrypted. So, if the message space $\mathcal{M}$ contains messages of different lengths, our security definition requires the adversary to distinguish the encryption of two messages of the *same* length.

One potentially surprising consequence of the CPA-security definition for encryption schemes is:

For an encryption scheme to be secure in any meaningful sense, the encryption algorithm ***must*** *be randomized.*

In contrast, secure MACs can be—and typically are—deterministic!

If the encryption algorithm is deterministic (i.e., not randomized), an attacker can win the CPA security game in Theorem 8.2.2 with probability 1. To do so:

- The attacker first asks for encryption of a message m_0 and receives a ciphertext c_0.
- Then, the attacker attacker choose a message $m_1 \neq m_0$ and sends $(m_0^*, m_1^*) = (m_0, m_1)$ to the challenger and receives the challenge ciphertext c^*.
- If $c^* = c_0$, the attacker outputs 0. Otherwise, the attacker outputs 1.

Deterministic encryption schemes are not only broken in theory, they are also broken in practice. For example, if you encrypt the pixels of an image using a deterministic encryption scheme, the encrypted image essentially reveals the plaintext image.

Why CPA security is a "weak" form of security There are two reasons why CPA security is a weak or insufficient definition of security for an encryption scheme:

- First, the CPA security definition guarantees nothing about message *integrity*: an attacker can modify the ciphertext and potentially change the meaning of the encrypted message in a meaningful way (even if the attacker does not know the encrypted message!).
- Second, the CPA security definition guarantees nothing in the event that the adversary can obtain *decryptions* of ciphertexts of its choosing. In practice, attackers can often obtain decryptions of chosen ciphertexts. for example, in a system where a client sends encrypted messages to a server and the server does something in response, an attacker can send encrypted queries to the server and observe its behavior to learn some function of the decrypted contents of the message. Later on, we will expand our definition to include these *chosen ciphertext attacks*.

8.3 One-time Pad

The one-time pad is perhaps the simplest encryption scheme. Its keyspace, message space, and ciphertext space are all the set of n-bit strings. The algorithms are then:

- $\mathsf{Enc}(k, m) \to c$. Compute $c \leftarrow (k \oplus m)$.
- $\mathsf{Dec}(k, c) \to m$. Compute $m' \leftarrow (k \oplus c)$.

The encryption scheme is correct since $\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = k \oplus (k \oplus m) = m$. A less obvious fact is that it is also *one-time secure*: if an attacker sees only one message with encrypted a one-time-pad key k, it learns nothing about the underlying plaintext.

The one-time pad is actually one-time secure in a very strong sense: it protects confidentiality even against a computationally unbounded attacker—one that can perform arbitrary amounts of computation.

The "two-time pad" attack The one-time pad is insecure if the same key k is every used to encrypt two messages. In particular, if two ciphertexts are ever computed using the same key, we have:

$$\begin{aligned} c_1 &= k \oplus m_1 \\ c_2 &= k \oplus m_2 \\ c_1 \oplus c_2 &= (k \oplus m_1) \oplus (k \oplus m_2) = m_1 \oplus m_2. \end{aligned}$$

The attacker then learns the XOR of the two encrypted messages, which is often enough to leak all sorts of sensitive information about the plaintext. For example, if the attacker knows some bits of m_1, it can learn some bits of m_2.

Why the one-time pad is useful The one-time pad encryption scheme feels in some sense useless: if a secure channel exists through which Alice and Bob can exchange an n-bit secret key, they may as well use that channel to exchange the message itself! There is some merit still in the one-time pad: it is possible to exchange the key ahead of time, and then send encrypted messages later on. Diplomats indeed used the one-time pad in this way throughout the 20th century. They would exchange huge books of keying material (random strings) and then use these keys to communicate securely over long distances.

In practice, it is much more convenient to be able to exchange a *short* key and then use it to encrypt many *long* messages.

8.4 A Weak Encryption Scheme

What we effectively need is a way to generate many bits of random-looking keys (i.e., keys for the one-time pad encryption scheme) from a single short random string.

To generate this, we can use a pseudorandom function! In particular, we would like a pseudorandom function of the form $F\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$.

If you forget what a pseudorandom function is, refer back to Theorem 4.2.1.

Using such a pseudorandom function, we can construct a new encryption scheme that replaces the truly random keys with pseudorandom keys generated from the pseudorandom function. The keyspace and message space for this encryption space are $\{0,1\}^n$ and the ciphertext space is $\{0,1\}^{2n}$. The algorithms are:

In practice, we will use AES or another block cipher as a pseudorandom function, so we will have $n = 128$ or so.

- $\mathsf{Enc}(k, m) \to c$:
  - Sample a random value $r \xleftarrow{R} \{0,1\}^n$. We call this the "nonce."
  - Compute a one-time key $k \leftarrow F(k, r)$ using the pseudorandom function.
  - Use this key to compute the ciphertext using the one-time pad: output $c \leftarrow (r, k \oplus m) \in \{0,1\}^{2n}$.
- $\mathsf{Dec}(k, c)$:
  - Parse the ciphertext c into (r, c').
  - Compute $m \leftarrow c' \oplus F(k, r)$.

Correctness holds by construction. The security argument goes in three steps:

While this encryption scheme is CPA-secure, it provides no message integrity. For any string $\Delta \in \{0,1\}^n$, an attacker can modify a ciphetext (r, c') to $(r, c' \oplus \Delta)$. This ciphertexts now decrypts to the original message XORd with the attacker-chosen string Δ.

- First, we argue that if n is large enough, the probability that the encryption algorithm ever chooses the same r value twice is negligible.

By the Birthday Paradox, after encrypting T messages, the probability of a repeated r value is $O(T^2/2^n)$, which is negligible in the key length n.

- Second, we argue that as long as the encryption algorithm never chooses the same r value twice, we can replace the values $F(k, r)$ with truly random strings. By the security of the pseudorandom function, the adversary will not notice this change.
- At this point, we can appeal to the security of the one-time pad scheme to argue that the adversary has no chance of winning the security game (Theorem 8.2.2).

For security to hold, it is crucial that the probability that the encryption algorithm uses the same encryption nonce r twice be negligibly small. If the encryption algorithm ever selects the same random nonce r twice, the pad $F(k, r)$ will be identical in two ciphertexts. An attacker can then apply the two-time-pad attack to recover the XOR of the two messages.

By the Birthday Paradox, if we sample the nonce from a space of size 2^{128}, we can expect a collision in 128-bit random values once around 2^{64} have been generated. Therefore, any single encryption-key must be used for $\ll 2^{64}$ messages. Cryptographic standards typically limit the number of bytes that users can encrypt with the same key to prevent these sorts of problems.

8.5 *Encrypting longer messages: Counter mode*

The CPA-secure encryption scheme of Section 8.4 only allowed encrypting messages of a fixed length. We now show how to use *counter-mode encryption* to extend this scheme to support messages of arbitrary length. That is, we will construct an encryption scheme $\mathsf{Enc}\colon \mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$ for messages of any length.

In practice, encryption schemes place some (large) bound on the length of encrypted messages. For example, the AES-GCM cipher has a maximum message limit of just under 64 GiB.

Counter-mode encryption works much as the CPA-secure encryption scheme we have already seen in Section 8.4, except that we split the message into blocks and encrypt each block separately.

We will use the function $\mathsf{ToString}\colon \{0, \ldots, 2^n - 1\} \to \{0,1\}^n$, which converts an integer in $\{0, \ldots, 2^n - 1\}$ to an n-bit string in the natural way.

The encryption scheme uses a pseudorandom function $F\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. The secret encryption key is a key $k \in \mathcal{K}$ for the pseudorandom function. To encrypt a message, we choose a fresh random value $r \xleftarrow{R} \{0, \ldots, 2^n - 1\}$ and XOR the ith block of the message with the value $F(k, \mathsf{ToString}(r + i \bmod 2^n))$.

- $\mathsf{Enc}(k, m)$:
  - Split the message m into blocks of n bits: $(m_1, m_2, m_3, \ldots, m_\ell)$. The last message block m_ℓ may be shorter than n bits.
  - Sample a random nonce $r \xleftarrow{R} \{0, \ldots, 2^n - 1\}$.

The nonce is sometimes called an "initialization vector" or "IV."

  - For $i = 1, \ldots, \ell$: Compute $c_i \leftarrow m_i \oplus F(k, \mathsf{ToString}(r + i \bmod 2^n))$. (If the last message block m_ℓ is less than n bits long, truncate the last ciphertext block c_ℓ to the length of m_ℓ.)
  - Output the ciphertext $c = (r, c_1, \ldots, c_\ell)$.

- $\mathsf{Dec}(k, c)$:
  - Parse the ciphertext c as $(r, c_1, \ldots, c_\ell)$, where all values but the last are exactly n bits long.
  - For $i = 1, \ldots, \ell$: Compute $m_i \leftarrow c_i \oplus F(k, \mathsf{ToString}(r + i \bmod 2^n))$. (Truncate m_ℓ to the length of c_ℓ.)
  - Output the message $m = (m_1 \| \ldots \| m_\ell)$.

As long as the space of random values r is large enough to ensure that the encryption routine never evaluates the pseudorandom function $F(k, \cdot)$ on the same input twice, this scheme will be CPA secure.

You may notice that this encryption scheme reveals the length of the encrypted message to the attacker! This indeed is a potential risk, but in some sense it is required: if we were to hide the length of the message, we would need to set some maximum message length and pad it up to this length. If we did this, encrypting a single word would necessarily result in a ciphertext equal in length to the ciphertext of encrypting a movie! This would greatly decrease the practicality of our encryption scheme. For applications where hiding the length is especially important, the messages can be padded to ensure they are all the same length before they are passed to the encryption scheme.

9
Authenticated Encryption

We have just constructed an encryption scheme with weak (CPA) security: one that provided security given that the adversary could see *encryptions* of messages of her choice. The "gold standard" security notion for encryption schemes allows the attacker to receive both encryptions of messages of its choice *and* decryptions of ciphertexts of its choice. Our security notion then says that even an attacker with this power should not be able to distinguish which of two chosen plaintext message a given ciphertexts encrypts. This new and strong notion of security for encryption schemes is called *security against adaoptive chosen-ciphertext attacks*.

Motivation: Chosen-ciphertext security It is not clear why chosen-ciphertext security is the right security notion to consider. In real-life applications, why would we ever allow an attacker to obtain *decryptions* of ciphertexts of its choosing? It turns out that, in many settings, attackers can indeed trick honest parties into decrypting adversarially ciphertexts and revealing their contents.

As a simple example, imagine a server that receives encrypted requests from the network, decrypts them, and either:

- returns an error if the decrypted request is malformed, or
- silently processes the request otherwise.

In this case, an attacker can send ciphertexts to the server and learn information about their decryptions by noticing whether or not the server returned an error message.

If the server returns an error when the decrypted message is illformed, we often call it a *padding oracle*. Many many real-world protocols using non-chosen-ciphertext-secure encryption schemes fall victim to this sort of attack.

CPA-secure encryption schemes provide *no* security guarantees in this setting. Even if the server leaks a single bit to the attacker about the decrypted value (such as whether the decrypted ciphertext is a well-formed request or not), the attacker could potentially learn the entire secret key!

As a concrete application-level example, consider what could happen if SSH (the secure shell protocol) used the counter-mode

encryption scheme, as described in the previous lecture, without any authentication. A user might send the command `echo secret > secret-file` to write their secret string, `secret`, to their own private file `secret-file`. The encryption of this command, sent over the network, would be the XOR of the command and the PRF-generated pseudorandom bytes. Suppose that the adversary knows the user is running this command, but doesn't know the contents of the 6-byte secret string. Consider what happens if the adversary XORs the encrypted message with the hexadecimal bytes `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 5c 11 0e 02 4a 04 58 04 05 05 06`. The leading zeroes mean that the first part of the command, `echo secret > `, will remain unchanged. When the server decrypts the latter part of the command (originally `secret-file`), however, it will obtain the XOR of `secret-file` and the bytes that the adversary XOR'ed in, which happens to turn into the string `/tmp/public`. As a result, if the server now runs this command, the user's secret data will be written to a file `/tmp/public`, which might be available to an adversary that also has an account on that same server and can look at files in `/tmp`.

Chosen-ciphertext security guarantees that such attacks are ineffective.

9.1 *Defining Authenticated Encryption*

Authenticated-encryption schemes simultaneously provide message integrity (as a MAC does) and confidentiality (as CPA-secure encryption does). Additionally, authenticated-encryption schemes remain secure even when an attacker can see encryptions and decrypts of ciphertexts of her choice. Perhaps unsurprisingly, the standard way to construct an authenticated-encryption scheme is to combine a CPA-secure encryption scheme with a MAC in a careful way.

We now formally define our strong security notion: *indistinguishability under chosen ciphertext attacks*, also known as IND-CCA2 security or CCA security.

Definition 9.1.1 (CCA security for encryption (strong))**.** An encryption scheme is *secure against adaptive chosen-ciphertext attacks* if every efficient adversary wins the following game with probability at most $\frac{1}{2}$ + "negligible":

- The challenger samples $b \xleftarrow{R} \{0,1\}$ and $k \xleftarrow{R} \mathcal{K}$.
- The adversary can make either of the following queries to the challenger repeatedly:
  - *Chosen-plaintext queries*
    - The adversary sends the challenger a message $m_i \in \mathcal{M}$

    * The challenger replies with $c_{m_i} \leftarrow \mathsf{Enc}(k, m_i)$.
  - *Chosen-ciphertext queries*
    * The adversary sends the challenger a *ciphertext* $c_j \notin \{c_{m_0}, \ldots, c_{m_i}\}$
    * The challenger replies with $m_{c_j} \leftarrow \mathsf{Dec}(k, c_j)$.
- The adversary then sends two messages $(m_0^*, m_1^*) \in \mathcal{M}^2$ to the challenger, where $|m_0^*| = |m_1^*|$.
- The challenger replies with $c^* \leftarrow \mathsf{Enc}(k, m_b^*)$.
- The adversary can make more chosen-plaintext queries and more chosen-ciphertext queries. (The adversary may not make a chosen-ciphertext query on the challenge ciphertext c^*.)
- The adversary outputs a value $b' \in \{0, 1\}$.
- The adversary wins if $b = b'$.

9.1.1 *Encrypt then MAC*

We typically achieve CCA security using the "encrypt-then-MAC" construction:

- First, encrypt the message using a CPA-secure encryption scheme on key k_{Enc}.
- Next, MAC the *ciphertext* using a secure MAC scheme and an independent key k_{MAC}.
- Output the ciphertext and the MAC tag.

As we discuss below, it is possible to derive both keys k_{Enc} and k_{MAC} from a single key k using a pseudorandom function.

The decryption routine first checks the MAC tag, then decrypts the ciphertext.

Using independent keys $(k_{\mathsf{Enc}}, k_{\mathsf{MAC}})$ is important in encrypt-then-MAC, as in many other cryptographic constructions. For example, a CPA-secure encryption scheme using n-bit keys can reveal the low order $n/2$ bits of its secret key in the ciphertext. And a secure MAC scheme using n-bit keys can reveal the high-order $n/2$ bits of its secret key in each MAC tag. Used independently, the encryption scheme and the MAC scheme are both secure. Used together with the same key k, the attacker learns all n bits of the key n and can break both primitives!

So, in general, you should always use independent keys for different primitives. To reduce the amount of keying material parties need to store, it is actually sufficient to store a single secret key k for a pseudorandom function F and derive all subsequent keys from the pseudorandom strings $F(k, 0), F(k, 1), \ldots$.

Theorem 9.1.2 (Informal)**.** *The Encrypt-then-MAC construction yields a CCA-secure encryption scheme, provided that: the underlying encryption scheme is CPA-secure and the underlying MAC scheme is secure (existentially unforgeable against adaptive chosen message attacks).*

Warning! ***Only*** *use encrypt-then-MAC* There are a number of bad ways to combine encryption and MACs to attempt to build authenticated-encryption schemes. MAC-then-encrypt is one way. Encrypt-and-MAC (i.e., MAC the message instead of the ciphertext) is another. Neither of these constructions is necessarily CCA-secure when used with a CPA-secure encryption scheme and a secure MAC scheme. So the ***only*** flavor of authenticated encryption you should use is encrypt-then-MAC.

Reminder: You should never need to implement authenticated-encryption schemes yourself. Instead use an off-the-shelf implementation that does the hard work for you. AES-GCM is one popular and widely implemented authenticated encryption scheme

9.2 *AES-GCM (Galois Counter Mode)*

One of the most widely used authenticated-encryption constructions is AES-GCM. It follows the encrypt-then-MAC paradigm. It uses AES as a pseudorandom function for counter-mode encryption from the previous lecture. It uses a Carter-Wegman-style MAC – see previous lecture – as the MAC scheme.

There are a few optimizations that AES-GCM uses beyond what we have described:

- AES-GCM derives both the encryption and MAC keys from a single short key using a pseudorandom function.
- AES-GCM implements a fast form of the Carter-Wegman MAC that does not need arithmetic modulo a big prime p, as the scheme described in Section 4.3.2 does. Instead, of defining the MAC using $\mathbb{Z}_p$ (integers mod a 128-bit prime p), GCM works with 128-bit strings. The GCM mode of operation replaces addition modulo p with XOR of 128-bit strings and it replaces multiplication modulo p with a somewhat complicated operation on 128-bit strings. (Formally, the scheme works over the field $\mathbb{F}_{2^{128}}$ of order 2^{128}.) This gives a big performance boost with no loss in security.

If you are interested, to implement the multiplication operation: think of both 128-bit strings as polynomials with 128 coefficients in $\mathbb{Z}_2 = \{0, 1\}$. Multiply the polynomials, reducing the coefficients modulo 2. Then reduce the resulting polynomial modulo some fixed polynomial of degree-128. Then interpret the result as a 128-bit string.

10
Key Exchange and Public-key Encryption

So far, we have talked about encryption systems that require the sender and recipient to *share a secret key*. In this chapter, we discuss how the sender and recipient can agree on a shared secret even if they only ever communicate over an open (insecure) network.

10.1 Key exchange

A key-agreement scheme over keyspace $\mathcal{K}$ is defined by efficient functions $(\mathsf{Gen}, \mathsf{Derive})$:

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{pk})$. The Gen algorithm generates a secret key and a public key for one party.
- $\mathsf{Derive}(\mathsf{sk}_A, \mathsf{pk}_B) \to k$. The Derive algorithm takes as input one party's secret key sk_A and the other party's public key pk_B and outputs a shared key $k \in \mathcal{K}$.

Formally, Gen also takes as input the security parameter.

Given this syntax, let us see how two parties can use a key-agreement scheme $(\mathsf{Gen}, \mathsf{Derive})$ to agree on a shared secret:

1. Alice runs $(\mathsf{sk}_A, \mathsf{pk}_A) \leftarrow \mathsf{Gen}()$ and sends pk_A to Bob.
2. Bob runs $(\mathsf{sk}_B, \mathsf{pk}_B) \leftarrow \mathsf{Gen}()$ and sends pk_B to Alice.
3. Alice and Bob both then compute a key $k \in \mathcal{K}$ as:
   - Alice computes $k \leftarrow \mathsf{Derive}(\mathsf{sk}_A, \mathsf{pk}_B)$.
   - Bob computes $k \leftarrow \mathsf{Derive}(\mathsf{sk}_B, \mathsf{pk}_A)$.

Correctness. Correctness for a key-agreement scheme just says that the two parties should always agree on the same shared secret:

Definition 10.1.1 (Key Agreement Correctness)**.** We say that a key-agreement scheme is *correct* if for all $(\mathsf{sk}_A, \mathsf{pk}_A) \leftarrow \mathsf{Gen}()$ and $(\mathsf{sk}_B, \mathsf{pk}_B) \leftarrow \mathsf{Gen}()$, it holds that:

$$\mathsf{Derive}(\mathsf{sk}_A, \mathsf{pk}_B) = \mathsf{Derive}(\mathsf{sk}_B, \mathsf{pk}_A).$$

Security. The standard notion of security for a key-agreement scheme only considers *passive attacks*: we consider adversaries that can view the network traffic but cannot modify it. In practice, we can combine key-agreement schemes with authentication schemes (e.g., digital signatures) to prevent active attacks by in-network adversaries.

Our security definition for key agreement says that even if an adversary sees both parties' public keys, it should not be able to distinguish the shared secret from random. That is, the following probability distributions D_{real} and D_{random} should be computationally indistinguishable:

When we say that two distributions are *computationally indistinguishable*, we mean that if we give the adversary a sample from one or the other, it can guess which sample it got with probability at most $1/2 +$ "negligible".

$$D_{\text{real}} := \left\{ (\mathsf{pk}_A, \mathsf{pk}_B, k) : \begin{array}{r} (\mathsf{sk}_A, \mathsf{pk}_A) \xleftarrow{R} \mathsf{Gen}() \\ (\mathsf{sk}_B, \mathsf{pk}_B) \xleftarrow{R} \mathsf{Gen}() \\ k \xleftarrow{R} \mathsf{Derive}(\mathsf{sk}_A, \mathsf{pk}_B) \end{array} \right\}$$

$$D_{\text{random}} := \left\{ (\mathsf{pk}_A, \mathsf{pk}_B, k) : \begin{array}{r} (\_, \mathsf{pk}_A) \xleftarrow{R} \mathsf{Gen}() \\ (\_, \mathsf{pk}_B) \xleftarrow{R} \mathsf{Gen}() \\ k \xleftarrow{R} \mathcal{K} \end{array} \right\}$$

10.2 *Diffie-Hellman key exchange*

We now give a simple and beautiful key-exchange protocol, due to Diffie and Hellman.

So far, we have been able to construct our cryptographic entities from constructs like a PRF, which meant that we could use "unstructured" algorithms like AES to compute them. We so far only know how to construct key-exchange schemes from more structured problems (e.g., based on number theory).

The version we will see uses large primes p and q a public parameters, along with a number $g \in \mathbb{Z}_p^*$, called the "generator." (**Warning:** The parameters p, q, and g must have some particular relation. So do not attempt to pick these parameters yourself.) Typically, we will have $p \approx q \approx 2^{2048}$.

For a prime p, the notation $\mathbb{Z}_p^*$ just denotes the non-zero integers modulo p. So when we write $ab \in \mathbb{Z}_p^*$, we mean $a \cdot b \bmod p$.

The keyspace of the Diffie-Hellman scheme is $\mathcal{K} = \mathbb{Z}_p^*$ and the algorithms are as follows:

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{pk})$.
  - Sample $\mathsf{sk} \xleftarrow{R} \{1, \ldots, q\}$.
  - Set $\mathsf{pk} \leftarrow g^{\mathsf{sk}} \in \mathbb{Z}_p^*$.
  - Output $(\mathsf{sk}, \mathsf{pk})$.
- $\mathsf{Derive}(\mathsf{sk}_A, \mathsf{pk}_B) \to k$.
  - We have $\mathsf{sk}_A \in \{1, \ldots, q\}$ and $\mathsf{pk}_B \in \mathbb{Z}_p^*$.
  - Output $k \leftarrow (\mathsf{pk}_B)^{\mathsf{sk}_A} \in \mathbb{Z}_p^*$.

Before we argue correctness and security, let us consider the computational efficiency of the scheme:

Efficiency. In order for the algorithm to be useful, Alice and Bob must be able to compute $g^x \in \mathbb{Z}_p^*$ efficiently, for $x \in \{1, \ldots, q \approx$

$2^{2048}\}$. However, trying to compute g^x where x is 2048 bits long naively would certainly not be efficient: it would require $x \approx 2^{2048}$ multiplications! However, we can compute this exponentiation much more efficiently using the following strategy:

- **Compute powers of g.** Write $\ell \leftarrow \lceil \log_2 p \rceil$. Then compute $(g, g^2, g^4, g^8, g^{16}, \ldots, g^{2^\ell})$, where all of these are in $\mathbb{Z}_p^*$. It is possible to compute g^{2^i} with a single multiplication modulo p as $g^{2^i} = (g^{2^{(i-1)}})^2 \in \mathbb{Z}_p^*$. So this step takes only $\ell = 2048$ multiplications.
- **Compute exponentiation.** Write the bits of the exponent as $x = x_0 \cdots x_{\ell-1}$. Then compute:

$$g^x = g^{\sum_{i=0}^{\ell-1} x_i 2^i} = \prod_{i=0}^{\ell-1} x_i (g^{2^i}) \in \mathbb{Z}_p^*$$

  This step again takes only ℓ multiplications.

In many applications, the generator g is fixed in advance. In this case, the implementation can precompute and store these powers of g.

Correctness. Correctness holds since $g^{xy} = g^{yx} \in \mathbb{Z}_p^*$ for all $x, y \in \mathbb{Z}$:

$$\mathsf{Derive}(\mathsf{sk}_A, \mathsf{pk}_B) = (\mathsf{pk}_B)^{\mathsf{sk}_A} = (\mathsf{pk}_A)^{\mathsf{sk}_B} = \mathsf{Derive}(\mathsf{sk}_B, \mathsf{pk}_A).$$

Security. To argue security, we must rely on a new computational assumption: essentially we just assume that the key-agreement scheme is secure.

Definition 10.2.1 (Decision Diffie-Hellman (DDH) assumption)**.** The *decision Diffie-Hellman assumption* states that, for a suitable choice of p, q, and g, the following distributions are computationally indistinguishable:

$$D_{\text{real}} := \{(g, g^x, g^y, g^{xy}) \in (\mathbb{Z}_p^*)^4 : x, y \xleftarrow{R} \{1, \ldots, q\}\}$$
$$D_{\text{ideal}} := \{(g, g^x, g^y, g^z) \in (\mathbb{Z}_p^*)^4 : x, y, z \xleftarrow{R} \{1, \ldots, q\}\}$$

To make the statement fully formal, we need to let p, q, and g grow with the security parameter.

In practice, we typically first run Diffie-Hellman key agreement, have the two parties run the shared secret that they get through a cryptographic hash function, and then use the hashed value as an encryption key. If we model the hash function as a random oracle, security can rely on a slightly weaker assumption—the "computational" Diffie-Hellman (CDH) assumption. The CDH assumption asserts, informally, that given (g, g^x, g^y), it is infeasible to compute g^{xy}. More formally, we have:

Definition 10.2.2 (Computational Diffie-Hellman (CDH) assumption)**.** The *computational Diffie-Hellman assumption* states that, for a suitable choice of p, q, and g, and for all adversaries $\mathcal{A}$:

$$\Pr[\mathcal{A}(g, g^x, g^y) = g^{xy} : x, y, \xleftarrow{R} \{1, \ldots, q\}] \leq \text{“negligible.”}$$

10.3 The discrete-log problem

The DDH assumption (Theorem 10.2.1) is no harder than the following problem, which asserts that computing x given $(g, g^x) \in (\mathbb{Z}_p^*)^2$ is computationally infeasible:

Definition 10.3.1 (Discrete-log assumption)**.** The *discrete-log assumption* states that, for p, q, and g, and for all efficient adversaries $\mathcal{A}$:

$$\Pr[\mathcal{A}(g, g^x) = x \colon x \xleftarrow{R} \{1, \ldots, q\}] \leq \text{“negligible”}.$$

Given an algorithm for the discrete-log problem, we can use it to solve the DDH problem. Given a DDH challenge (g, g^x, g^y, g^z), compute the discrete log of g^x and test whether $g^z = (g^y)^x$. For certain choices of p, q, and g, the best known algorithm for the DDH problem is to first solve the discrete-log problem in $\mathbb{Z}_p^*$.

How hard it to solve the discrete-log problem then?

1. The most basic attack is to enumerate all p possible values of x and check whether the corresponding g^x matches. This will take time $p \approx 2^{2048}$.
2. There is a slightly more clever algorithm, called “Baby Step Giant Step,” that is able to compute x in time $\sqrt{p}$.
3. In $\mathbb{Z}_p^*$, a much better attack is the Number Field Sieve. This algorithm is able to compute x in (roughly) time $\exp((\log p)^{1/3}(\log\log p)^{2/3})$—sub-exponential time! The existence of this attack is the reason we require p to be 2048 bits long to get 128-bit security.

An attack that runs in time $\sqrt{p}$ runs in time $2^{\frac{1}{2}\log p}$. In contrast, the Number Field Sieve runs in time roughly $2^{\sqrt[3]{\log p}}$. This is much much much faster than the $\sqrt{p}$-time algorithms, since the exponent grows much more slowly.

10.4 Generalizations of Diffie-Hellman

We have described the Diffie-Hellman protocol in terms of $\mathbb{Z}_p^*$—multiplication of integers modulo p. In particular, the protocol uses a *set* of elements (here, $\mathbb{Z}_p^*$) and a *binary operation on elements* (here, multiplication modulo p). There is a natural generalization of the Diffie-Hellman protocol that works with other sets of elements and binary operations that operate on them.

The most widely used version of the Diffie-Hellman protocol today uses *elliptic-curve groups*. The set of elements in an elliptic-curve group is a set of points $(x, y) \in \mathbb{Z}^2$ in two-dimensional space, where $0 \leq x, y \leq p$ for some prime $p \approx 2^{256}$. The binary operation on elements is some geometric operation on two points that yields a third point. Even though the underlying objects are now points instead of integers modulo p, the Diffie-Hellman protocol looks exactly the same in this setting.

More precisely, we can define Diffie-Hellman key exchange with respect to any *finite cyclic group*—a concept from abstract algebra. A cyclic group just consists of a *set* of elements and a *binary operation on elements*. The set and operation need to satisfy certain mathematical properties—associativity, etc.

Given a group $\mathbb{G}$, we can then define a discrete-log assumption on $\mathbb{G}$ and whenever discrete-log is hard on $\mathbb{G}$, we can use $\mathbb{G}$ to construct cryptosystems.

The appeal of elliptic-curve cryptosystems is that the best known discrete-log algorithm on certain elliptic-curve groups is Baby Step Giant Step, which runs in $\sqrt{p}$ time. So, we can use elliptic-curve groups of size 2^{256} and the Diffie-Hellman public keys take only ≈ 256 bits to represent. In contrast, when working in $\mathbb{Z}_p^*$, we need to work modulo a prime $p \approx 2^{2048}$ to defeat the Number Field Sieve attack, so Diffie-Hellman public keys in this setting take ≈ 2048 bits to represent.

10.5 Defining Public-Key Encryption

The definition for a public-key encryption scheme will be similar to the definitions we saw for symmetric-key encryption:

Definition 10.5.1 (Public-Key Encryption Scheme)**.** A public-key encryption scheme over message space $\mathcal{M}$ consists of three efficient algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$:

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{pk})$: Generates a keypair with secret key sk and public key pk.
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$: Uses public key pk to encrypts a message $m \in \mathcal{M}$ to a ciphertext c.
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$: Uses secret key sk to decrypt ciphertext c into message $m \in \mathcal{M}$.

Definition 10.5.2 (Public-Key Encryption - Correctness)**.** For all keypairs $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}()$ and for all messages $m \in \mathcal{M}$, it holds that

$$\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m.$$

The literature sometimes calls security against chosen-plaintext attacks ("CPA security") *semantic security*.

Definition 10.5.3 (Public-Key Encryption - Security against chosen-plaintext attacks (Weak))**.** A public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *secure against chosen-plaintext attacks* if all efficient adversaries $\mathcal{A}$ win the following game with probability $\leq \frac{1}{2} + \text{negl}$:

- The challenger generates $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}()$ and $b \xleftarrow{R} \{0,1\}$ and sends the public key pk to $\mathcal{A}$.
- The adversary $\mathcal{A}$ sends $m_0, m_1 \in \mathcal{M}$ to the challenger, where $|m_0| = |m_1|$.
- The challenger responds with $\mathsf{Enc}(\mathsf{pk}, m_b)$
- The adversary outputs b' and wins if $b' = b$.

Note that since this is a public-key encryption scheme, the adversary can use the public key to generate encryptions of any message of their choice. As in the secret-key setting, here it is crucial that the encryption algorithm be randomized—otherwise two encryptions of the same message are identical and the attacker can break CPA security.

For symmetric-key encryption, we also defined a stronger notion of security that we called security against chosen-ciphertext attacks (CCA security). We can extend this definition of CCA security to the public-key setting.

This security definition asserts that the attacker should not be able to distinguish the encryption c^* of two messages of its choice, even if the attacker can obtain decryptions of any ciphertext except c^*. This is a very strong notion of security (since the security definition gives the attacker a lot of power) and it is the notion of security that we typically demand in practice.

Definition 10.5.4 (Public-Key Encryption - Security against chosen-ciphertext attacks (Strong))**.** A public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *secure against chosen-ciphertext attacks* if all efficient adversaries $\mathcal{A}$ win the following game with probability $\leq \frac{1}{2} + \text{negl}$:

- The challenger generates $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}()$ and $b \xleftarrow{R} \{0,1\}$ and sends the public key pk to the adversary $\mathcal{A}$.
- The adversary may make polynomially many *decryption queries*:
  - The adversary $\mathcal{A}$ sends ciphertext c to the challenger.
  - The challenger responds with $m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$
- At some point, the adversary sends a pair of messages $m_0, m_1 \in \mathcal{M}$ to the challenger, where $|m_0| = |m_1|$.
- The challenger returns $c^* \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b)$.
- The adversary can continue to make decryption queries, provided that it never asks the challenger to decrypt the ciphertext c^*.
- The adversary outputs b' and wins if $b' = b$.

10.6 ElGamal Encryption Scheme

In public-key encryption, we want a sender to be able to encrypt a message to a recipient. The goal of public-key encryption is to perform encryption without a shared secret key.

ElGamal's encryption scheme essentially uses Diffie-Hellman key exchange to allow the sender and recipient to agree on a shared secret key k, and then has the sender encrypt her message with a symmetric-key cryptosystem under key k.

ElGamal's scheme was not the first public-key encryption scheme. The first scheme, RSA, was much more complicated despite Diffie-Hellman already having been published.
In practice, we typically use elliptic-curve groups to instantiate ElGamal encryption, instead of $\mathbb{Z}_p^*$. But the general principle is exactly the same.

Definition 10.6.1 (Hashed ElGamal Encryption)**.** Let p, q, and g be integers of the sort we use for Diffie-Hellman key exchange (Section 10.2). In particular, $p \approx q \approx 2^{2048}$ and $g \in \mathbb{Z}_p^*$.

Let $H : \mathbb{Z}_p^* \to \{0,1\}^*$ be a hash function (modelled as a random oracle). Let $(\mathsf{Enc}', \mathsf{Dec}')$ be a symmetric-key authenticated-encryption scheme. Then define:

- $\mathsf{Gen}() \to (\mathsf{sk}, \mathsf{pk})$:
  - Choose $a \leftarrow \{1, \ldots, q\}$.

  - Compute $A \leftarrow g^a \in \mathbb{Z}_p^*$.
  - Output $(\mathsf{sk}, \mathsf{pk}) \leftarrow (a, A)$.
- $\mathsf{Enc}(\mathsf{pk}, m) \rightarrow c$:
  - Choose $r \leftarrow \{1, \ldots, q\}$.
  - Compute $R \leftarrow g^r \in \mathbb{Z}_p^*$.
  - Compute $k \leftarrow H(\mathsf{pk}^r \in \mathbb{Z}_p^*)$.
  - Output $c \leftarrow (R, \mathsf{Enc}'(k, m))$.
- $\mathsf{Dec}(\mathsf{sk}, c) \rightarrow m$:
  - Parse $(R, c') \leftarrow c$.
  - Compute $k \leftarrow H(R^a \in \mathbb{Z}_p^*)$.
  - Output $m \leftarrow \mathsf{Dec}'(k, c')$.

10.6.1 Performance

The performance of this scheme is limited by the exponentiations—the symmetric encryption scheme is quite fast (gigabytes per second), but a single exponentiation can take a millisecond on modern processors. For encryption, this scheme requires two exponentiations (g^r and pk^r). Decryption requires one exponentiation (R^{sk}). To speed up the encryption routine, we can precompute powers of g: $g^2, g^4, g^8, g^{16}, \ldots$, which saves a factor of two in exponentiations.

Hashed ElGamal encryption is one of the most common public-key encryption schemes used today.

10.6.2 Security

If we model the hash function H as a random oracle, we can prove security of ElGamal encryption from (a) the computational Diffie-Hellman assumption (Theorem 10.2.2) and (b) the CCA security of the underlying authenticated-encryption scheme $(\mathsf{Enc}', \mathsf{Dec}')$.

11
Encryption in Practice

So far, we have established several constructions that allow us to hide the contents of transmissions: we created chosen-plaintext- and chosen-ciphertext-secure encryption schemes that worked with and without a shared key. We will now discuss a few practical applications of transport encryption, and why it is often difficult to get right.

11.1 File Encryption

Perhaps the most straightforward use of encryption is file encryption.

Example: WhatsApp Encrypted Backup. WhatsApp allows the app's users to back up their messages and contacts to the cloud. This way, a user can recover her messages if she loses or breaks her phone. To hide the user's data from WhatsApp's cloud servers, WhatsApp uses encrypted backup. To achieve this, the user's device generates a 128-bit AES key k at the time of backup and encrypt the message data (photos, messages, etc.) using AES-GCM$(k, \cdot)$ before sending the *ciphertext* to the server. In order to allow you to restore your backup on a new phone, the app allows you to export 64 decimal digits that encode the AES key used. When restoring your backup, you will enter these digits and your phone will fetch the ciphertext from the server and use these digits to reconstruct the key and decrypt your messages.

See **this document** for details on how WhatsApp encrypts backups. (The document also describes a more complicated backup scheme that uses passwords for encryption.)

This is a fairly simple application of file security. However, file encryption can be much more tricky: many applications require or desire features beyond simple encryption and decryption.

11.1.1 Case Study: PDF v1.5 Encryption

One instance of this desire for extra features that ended up going wrong was a previous version of the PDF standard, PDF v1.5.[1] This standard provided several features:

[1] Jens Müller et al. "Practical decryption exfiltration: Breaking pdf encryption". In: *ACM CCS*. 2019.

1. It is possible to password-encrypt some or all of the document. (The encryption scheme does not matter, but think of it as a secure authenticated-encryption scheme.) For example, a PDF could have an unencrypted title page and have the rest of the pages be encrypted.
2. A PDF document can contain a form that the PDF reader submits to a server via HTTP.
3. A form in a PDF document can reference other parts of a document.
4. The PDF reader may submit a form when an event happens: when the PDF is opened, closed, decrypted, etc.

Each of these features individually seems innocuous. However, when combined, there is a clever attack that allows an attacker to learn the contents of the encrypted portion of a PDF.

The attack works against a PDF document with an *unecrypted (public) title page* and an *encrypted (secret) body*. We assume that the attacker can modify the PDF file on its way to the victim.

To mount the attacker, the attacker intercepts the PDF on the way to the victim and replaces the title page with an "evil" title page. The evil title page:

- contains an invisible PDF form,
- reads the contents of the decrypted pages into a PDF form element, on the event that decryption of the PDF body succeeds, and then
- submits the form via HTTP to `evil.com`

So when the victim recipient enters her password into the PDF reader to decrypt the document, the evil title page is exfiltrates the decrypted content to `evil.com` via a PDF form.

What went wrong? The core issue here was that the unencrypted contents of the document were not authenticated. That is, an attacker could modify the unencrypted pages of the document without detection. A better design would have been to either use a MAC over all pages of the document or to use a primitive called "Authenticated Encryption with Associated Data" to authenticate the encrypted data together with the unencrypted data.

One downside of MACing the entire document is that, before rendering even a single page of the PDF, the reader would have to compute a MAC over the entire PDF. If the PDF consists of many thousands of pages, this could be expensive. Using a more sophisticated cryptographic construction: per-page MACs, plus MACs on the document metadata, etc., we might be able to construct a scheme that protects document integrity while allowing partial document

rendering. But the design of such a scheme would have to carefully defend document integrity against the sort of attacks we describe here.

11.2 Stream Encryption: Transport Layer Security (TLS)

The standard Internet data transfer protocols—TCP and UDP—provide no integrity or confidentiality whatsoever. To protect the data that we send over the network, we use the *transport-layer security* (TLS) protocol. The TLS protocol runs on top of TCP and aims to provide an encrypted "tunnel" between a client and server.

HTTPS is simply HTTP run over TLS.

While designing a stream-encryption protocol may seem straightforward at first glance, the task is much more subtle than it would seem. However, as is often the case in security, features and practical requirements make the situation much more complex.

11.2.1 Downgrade Attack

We now give one example of a *downgrade attack*—the sort of subtle security issue that can arise in protocol design.

The current version of TLS is TLS 1.3. An older version of the TLS standard, called SSLv3, is vulnerable to devastating attacks that allow an attacker to recover the plaintext traffic. However, for backwards compatibility, many TLS clients and servers still supported SSLv3 until quite recently.

When a TLS client connects to a TLS server, the two parties first exchange some messages to decide which version of the TLS protocol to use. In particular, the client will try to connect to the server using the latest version of TLS it supports. The server will respond back with either a confirmation (if the server supports the client's proposed TLS version) or with garbage (if not). If garbage, the client will try to connect to the server with an older version of TLS.

An important point is that *none* of these negotiation messages are authenticated—the client and server cannot start using authentication (MACs, signatures, etc.) until they agree on which version of TLS to use! So, an active in-network attacker can simply replace all of the server responses in this protocol-negotiation phase with garbage until the client downgrades all the way to SSLv3. Once the client and server agree to use SSLv3, believing that this is their best available option, the attacker can then monitor and decrypt their traffic using known attacks on SSLv3.

The best defense against this attack is for both parties to disable support for SSLv3 completely.

11.2.2 TLS Structure

TLS consists of two main phases:

1. **Handshake:** In this phase, the client and server use a key-exchange protocol to agree on a shared key to use to encrypt their application-layer traffic. This step uses public-key cryptography, since the client and server initially have no shared secret.
2. **Record protocol:** This phase is where the actual application-layer communication happens. This phases uses the secret key that the client and server agreed upon in the handshake phase for authentication and encryption.

11.2.3 TLS Handshake Properties

In our definitions of public-key encryption, we had only two (relatively simple) properties: correctness and security. The TLS handshake, however, has a much more complicated set of goals:

- **Correctness:** Both parties agree on the same session key at the conclusion of the handshake.
- **Security:** adversary "learns nothing" about the secret key that the parties agree upon at the conclusion of the handshake.
- **Peer authentication:** At the end of the handshake, each party believes that they are talking to the other party.
- **Downgrade protection:** The parties agree on the same version of TLS that they would agree on absent an in-network attacker.
- **Forward secrecy with respect to key compromise:** if an attacker steals the secrets stored on the client or the server, the attacker cannot decrypt past traffic.
- **Protection against key-compromise impersonation:** If an attacker steals a client's secret key, the attacker should not be able to impersonate other servers to the client.
- **Protection of endpoint identities:** The public keys of the two parties should never flow over the wire in the clear. For example, if a client is connecting to a website that uses a content-distribution network, such as Akamai or Cloudflare, an attacker should not be able to tell which website the client is connecting to—only that it is hosted on Akamai or Cloudflare.

To provide forward secrecy, modern cryptographic protocols use long-term secrets only for signing—not for encryption. These protocols use *ephemeral* (one-time-use) cryptographic keys for key exchange and encryption. Protocol participants delete these ephemeral keys on connection teardown and/or they rotate these keys often.

This way, if an attacker compromises a party's secret key, the attacker can only sign messages on behalf of that party; the attacker cannot use the secret to decrypt past messages.

11.2.4 TLS Handshake

The TLS handshake is very carefully designed to achieve these properties. A grossly simplified version looks something like the following:

1. At the start of the handshake, the TLS client holds the public

pk_{CA} of a certificate authority. The TLS server (for example, MIT) holds its secret signing key sk_{MIT} and a public-key certificate cert_{MIT} binding its public key pk_{MIT} to its domain `mit.edu`.

2. **Client Hello**: The client sends the following values to the server:
   - a random nonce,
   - list of supported ciphersuites, and
   - an ephemeral Diffie-Hellman public key. (The client constructs this Diffie-Hellman public key using its preferred ciphersuite. If the server does not support the ciphersuite the client picked, the client will have to re-run this step using a different ciphersuite.)
3. **Server Hello**: The server sends several values to the client, choosing a ciphersuite to use and completing the Diffie-Hellman key exchange. That is, the server sends:
   - a random nonce,
   - a ciphersuite to use,
   - and an ephemeral Diffie-Hellman public key
4. Both partices compute a shared session key k using Diffie-Hellman key agreement on the ephemeral keys they exchanged.
5. Under encryption using the keys derived from the session key k, the server sends the certificate for `mit.edu` as well as a signature over all messages sent so far, using its long-term secret key sk_{MIT}. The client then checks that:
   - the certificate has been signed by one of the client's trusted CAs, and
   - the signature from the server matches their own record of the messages.
6. Finally, the client and server run the TLS Record Protocol to exchange encrypted and authenticated application data.

A *ciphersuite* contains all of the cryptographic parameters needed to perform key exchange, hashing, authentication, and encryption. For example, one possible ciphersuite for TLS 1.2 is `ECDHE-RSA-AES256-GCM-SHA384`. This indicates use of ephemeral elliptic-curve Diffie-Hellman key exchange (`DHE`) with RSA signatures (`RSA`), 256-bit AES encryption in GCM mode (`AES256-GCM`), and SHA2-384 as the hash function (`SHA384`).

This simplified toy version of the TLS handshake does not provide many of the features that the real TLS handshake provides. But it should give you a flavor of what the real handshake looks like.

11.3 Properties that TLS does not provide

Authenticated End-of-File TLS does not provide any end-of-file authentication, or "clean closure." To explain what this means by example:

A popular tool to install the toolchain for the trendy systems programming language Rust is `rustup`. To use the tool and install the Rust toolchain, the recommanded method is to run the command

"`curl https://sh.rustup.rs | sh`." This downloads a shell script from the internet over HTTPS and immediately runs it using the shell `sh`.

Imagine that the contents of the downloaded script create a temporary directory, copy things into it, install some things, and finally delete the temporary directory with something like `rm -r /tmp/install`. An in-network attacker, who knows the structure of the `rustup` install script, could drop all of the packets in the stream immediately after the characters `rm -r /`. Eventually, the TLS connection will timeout, an shell will run the command `rm -r /`, deleting the user's entire file system.

To protect against this, script writers try to design their scripts such that if the stream is cut off in the middle of a download, nothing happens. For example, the install script might consist of a single function definition that is called at the very end of the function.

Plaintext Length Obfuscation As we have discussed, encryption reveals exactly the length of the plaintext. If there is data that is not encrypted that is then included inside the encrypted data as well, this can cause a vulnerability—see the CRIME attack.

12
Open Questions in Encryption

So far, we have established what may seem like a comprehensive set of tools to transmit data over the network: we have schemes for verifying the integrity of data and for hiding the contents of a transmission from an adversary, both with and without a shared key.

Transport-layer security (TLS) effectively builds and "encrypted pipe" between a client and a server. Through the encrypted pipe that TLS provides, we can run any of our favorite TCP-based protocols—HTTP, SMTP, POP, IMAP, etc.—and can thus hide our data from an in-network attacker. And yet, the security guarantees that TLS provides fall short of the strongest possible security notions we could desire.

If we were to imagine the best possible security we could ask for regarding network traffic, it might look (imprecisely) something like the following:

> *An attacker who controls many parties (clients and servers) as well as the network should "learn nothing" about who the client is talking to and what she is saying.*

Unfortunately, the protocols we have for secure communication today fall far short of this goal. In this section, we describe some of the shortcomings of today's transport-security tools and some imperfect solutions.

12.1 Problem: Encryption does not hide the source and destination of a packet

In order to send IP packets over the internet, the Internet's routing system relies on routers in the network knowing the source and destination IP addresses of each packet: these are included, unencrypted, in the packet header. (In some ways, the "pipe" analogy fits here: anyone can see where the pipe starts and where it ends.)

Solution Attempt: Tor The Tor system aims to allow a client to connect to a server over the Internet while hiding—from certain types of adversaries—which server the client is connecting to. For example, a Tor client should be able to browse the web without anyone learning which websites the client is visiting.

Tor's strategy is to bounce traffic around the Internet and hope that no real-world attacker can gather enough information to figure out which client is communicate with which server. Tor provides no precise security guarantees, and there are scores of research papers demonstrating various weaknesses in Tor's security plan. At the same time, Tor is publicly available, is well supported, is widely used, and seems to provide some meaningful privacy benefits in practice.

It is difficult to evaluate the security of a tool like Tor, since real-world attackers will not necessarily reveal that they can break the tool's security guarantees. So using a tool like Tor requires taking a leap of faith.

Tor works by nesting several of these encrypted pipes: when opening a connection, the Tor client will first select three *relays* from the Tor network (A, B, C). The Tor client will then:

1. Open an encrypted tunnel to the first relay A (the "guard").
2. Through that tunnel, open an encrypted tunnel to the second relay B.
3. Through that tunnel-inside-a-tunnel, open an encrypted tunnel to the third relay C (the "exit").

The client will then send its application-layer traffic through this tunnel-inside-a-tunnel-inside-a-tunnel. So each byte of application data will be encrypted first for relay C, then for relay B, then for relay A. When the client sends this ciphertext over the *circuit* from relay A to B to C to the real destination, relay A will first strip off its layer of encryption then forward the inner packet to relay B. Relay B will do the same, stripping off a layer of encryption and forwarding the packet to relay C. Finally, relay C will strip off the last layer of encryption and be left with a normal IP packet that it can then send to the destination server. As the response makes it back through the network, each relay node will add a layer of encryption. The end result of this is that no single relay can see the source and destination IP addresses.

However, the security that Tor provides is imperfect. First, if an attacker controls the guard node (relay A) and the exit node (relay C), the attacker can correlate the timing of when a packet enters the guard node and when a packet exits the exit node. Using this timing an attacker can make a guess at the route traffic is taking through the Tor network. Even without controlling relay nodes, if an attacker controls certain key points in the Internet (e.g., Internet exchange points or undersea fiber links) it may be able to perform this sort of traffic analysis even without controlling relays.

12.2 *Problem: Attacker sees packet sizes and timings*

As we discussed, practical encryption schemes necessarily reveal the length of the ciphertext. In the context of the Internet, this means that an attacker can learn the size of each TCP packet that a client sends, along with timing information. (Here the "encrypted pipe" analogy for TLS breaks down: an attacker can see how much traffic flows through the encrypted pipe and when.)

Even without seeing the destination and source of packets sent to and from a client's machine, an attacker can learn significant amounts about the client's traffic. Some examples are:

- *Watching a movie:* Video traffic has a distinct traffic pattern. By monitoring, for example, the length of time that a client spends watching a movie, a network attacker learn with fairly high accuracy *which movie* the client is watching.
- *Using `ssh`:* Different commands will have different traffic patterns. An attacker may be able to infer what type of commands a client is running by inspecting traffic patterns.
- *Downloading a file:* The bitlength of a downloaded file can uniquely identify the file in many cases.
- *Browsing the web:* sizes leak individual pages.

Example: New York Times The New York Times homepage `nytimes.com/` downloads 1.56 MB of content, along with 76 total assets (images, CSS, JavaScript). The webpage to submit a sensitive tip, `nytimes.com/tips`, downloads 41.92 KB and only 15 assets. By counting the number of HTTPS requests that a client makes over an encrypted connection to `nytimes.com`, an attacker can easily distinguish whether a client is visiting the homepage or the tips page, even if the attacker cannot decrypt even a single bit of the HTTPS traffic itself.

Attempts at a solution There are several common ways that people attempt to protect against this sort of traffic analysis. None of these solutions works well.

1. **Random Noise:** To try to hide the length of the packets it sends, a client can add a randomly chosen number of bytes of dummy data to the end of each packet. The hope is that by randomizing packet lengths, the client prevents the attacker from performing the traffic analysis.

   Unfortunately, a patient attacker can use *averaging* to effectively eliminate the effect of the random noise. That is, if the attacker can

trick the client into sending the same message a few times (as is often possible), the attacker can average the noised packet lengths to get a good estimate of the true length.

2. **Padding:** Another option is to just pad every packet (or webpage or encrypted message, etc.) to match the largest packet that the client will ever send. For example, whenever the client visits a page on `nytimes.com`, the client could download 50MB of page content and 100 fixed-size assets, even if the true page is tiny. This is somewhat secure, but incredibly costly and therefore not practical outside of very specific circumstances.

12.3 A Promising Direction: Metadata Privacy for Messaging

Messaging apps like WhatsApp and iMessage are end-to-end encrypted, but still may leak who you are talking to. This problem is more tractable due to the circumstances of messaging:

- Messages are approximately fixed length.
- Some latency is OK.
- Total daily traffic per user is small.
- Each user talks to few messaging partners.

Because of these constraints, it may be feasible to use techniques like padding to greatly reduce the amount of data that messaging metadata reveals and to do so in a way that provides strong formal guarantees about security. But still, no widely used messaging app provides any sort of metadata-privacy guarantees.

12.4 Problem: Endpoint Compromise

Say that we have a perfect scheme for transport security—one that hides all data and metadata. Such a scheme is still not enough to protect our data if an adversary can compromise the communication *endpoints*.

For example, in many applications, a client sends some sensitive data to a server (e.g., its Google search queries). The server is free to lose it in a breach, sell it, or turn it over to law enforcement agencies, etc. Later on, we will discuss how to protect against server compromise using software-engineering techniques. Here, we will give one example of how cryptography can protect user data even against a compromised server.

12.4.1 Private Information Retrieval

In many applications, a client must read a record from a database stored at a server. The client might like to perform such a database query without the server learning which record it accessed.

A concrete application of this is Google search—in order to give you search results, Google necessarily learns what you are searching. With a PIR scheme, it would be possible for Google to look up search results without learning what you are searching for!

In a *private information retrieval* scheme:

- the server holds a public database of n bits: $x_1, x_2, \ldots, x_n \in \{0,1\}$, and
- the client holds a secret index $i \in \{1, \ldots, n\}$.

The client and server interact. At the end of the interaction we want the following properties to hold:

- *Correctness:* The client outputs $x_i \in \{0,1\}$.
- *Security:* The server "learns nothing" about the client's secret index i. In particular, we demand that the message that the client sends to the server is a CPA-secure encryption of its index i.

Naïve private information retrieval. The simple scheme for private information retrieval is to have the server send all n bits of the database to the client. When the database is large, as it is for Google search, this would be an infeasible amount of communication. A surprising fact is that there are simple private-information-retrieval protocols that involve much less than n bits of client-server communication.

12.4.2 A non-trivial private-information-retrieval scheme.

To achieve this, we need a new tool called *additively homomorphic encryption*.

Additively homomorphic encryption A secret-key additively homomorphic encryption scheme is a CPA-secure secret-key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ over key space $\mathcal{K}$ and message space in $\mathcal{M} = \mathbb{Z}_p$ with the added property that for all keys $k \in \mathcal{K}$ and all messages $m, \hat{m} \in \mathcal{M}$,

$$\mathsf{Enc}(k, m) \star \mathsf{Enc}(k, \hat{m}) = \mathsf{Enc}(k, m + \hat{m}),$$

where "$\star$" is some fixed binary operation on ciphertexts.

In English: given two encrypted messages m and $\hat{m}$, encrypted under an additively homomorphic encryption scheme, anyone can compute the encryption of $m + \hat{m}$. Being able to add encrypted messages also allows multiplying encrypted messages by public constants, since $m + m = 2m$ and $2m + 2m = 4m$ and so on. Once we can add and multiply by constants, and we can compute a matrix-vector product of an encrypted vector and a public matrix.

It is possible to construct an additively homomorphic encryption scheme from the DDH assumption, with only a slight tweak to ElGamal encryption.

PIR Construction We can use additively homomorphic encryption to construct a private-information-retrieval scheme. To do so, the server represents its its database (the x_i values) as a $\sqrt{n} \times \sqrt{n}$ matrix D. The client then tells the server which column j it would like by supplying the encryption of a $\sqrt{n} \times 1$ vector $\mathbf{m}$ with a 1 in the jth location. The client sends this encryption (using a key only the client knows) as $\mathsf{Enc}(k, \mathbf{m})$. The server computes the matrix product $\mathsf{Enc}(k, D\mathbf{m})$, which gives the jth column of the matrix, using additively homomorphic encryption and returns the response to the client, where the client can find the bit they are interested in in the column.

This allows a client to retrieve a bit from a server's database without the server learning anything about the desired bit, and to do so at the communication cost of only $2\sqrt{n}$ ciphertext. The server computation cost is high—the server necessarily touches every bit of the database—but at least it shows that making private queries is feasible in theory.

Part III

Platform Security

13
Architecting a secure system

So far, we have been focusing on security for network communication. We have established many tools to achieve this, from message authentication codes to public-key encryption.

Ultimately, however, applications need to make use of these tools. And for our network security tools to provide meaningful security, the applications themselves must be reasonably secure.

In discussing platform and application security, there are two classes of problem that we want to defend against.

1. Mistakes of various types.

- Buggy systems: including hardware and software bugs
- User mistakes: phishing, misconfiguration

2. Malicious people or components.

- Malicious components: malware, supply-chain attacks
- Malicious users: what if the adversary gets the admin's password?
- Attacker gets access to the system: insider attacks, the adversary guesses credentials,

In computer security, we tend to treat mistakes/bugs and malicious software/components in the same way. We do that because (1) it's often difficult to specify what it means for a component to be "non-maliciously buggy" and (2) an attacker can often leverage what seems like a benign bug into full-fledged misbehavior.

Thus, a theme that will be present throughout this section is that we will consider mistakes to be malicious: if we are prepared to handle malicious components, we will similarly be prepared to handle our own buggy code. If we are prepared to limit damage of a malicious user with the admin password, we will also be limiting the damage that a mistake-making admin can cause.

This multitude of threats makes designing secure applications quite difficult. To make progress, we will seek to design systems that limit damage when things go wrong.

When we design for security, we have essentially three goals:

1. **Defend against known attacks.**
2. **Defend against unknown attacks.**
3. **Limiting damage.** In the cryptography part of this course, systems are either security or insecure. In systems security, things are much more gray. Attacks we care about often are outside of our threat model—even when this happens, we'd like to somehow contain the damage.

13.1 Isolation

One of the most effective strategies to limit damage is to split a system into isolated components. If one of these components becomes compromised, it should not be able to compromise the other components. For example, if you run code in one virtual machine, it should not be able to tamper with data in another virtual machine.

These components will typically run on top of some *host* that enforces isolation. Importantly, this host must be correct! If there are bugs in the host, malicious code in a component may be able to exploit a bug to escape its isolation. The success of an isolation mechanism depends on the correctness and configuration of the host.

When choosing what mechanism to use to isolate various components, we think a lot about the *performance overhead* of an isolation mechanism. The challenge of building a good isolation mechanism is ensuring strong isolation without slowing down the isolation components too much (or taking up too much extra memory).

Examples	Host
Docker Container	Operation System (e.g. Linux)
Browser tabs	Browser
Language-Level (JavaScript, Wasm)	Language Runtime
Process	Linux kernel
Virtual machines (VMs)	VM Monitor
Physical ("air gap")	Physics

Table 13.1: Some common types of isolation

In order for these isolated components to be useful, they will need to be able to talk to each other in some form. For example, a client component must be able to make requests to a database component, but we would like to limit the power of the client to do damage. For this, we would like to achieve *controlled sharing*.

13.1.1 Controlled Sharing

For an isolation mechanism to be useful, it additionally needs to have some way to interact with other isolated components. For example,

some JavaScript code isolated in a browser tab still needs some means by which to make requests over the network.

When a host decide whether to allow a request from a particular component, it typically needs to do three things with each request:

Since all three of these actions start with the letters "Au," we sometimes call this the *gold standard* for controlled sharing.

- Authenticate: Associate the request with some *principal*. A principal could be a user name, an "origin" in the web context (e.g., `google.com`), a program, or some other entity in the system.
- Authorize: Decide whether that principal is allowed to make the request.
- Audit: Keep track of requests that each principal makes. Auditing is about limiting damage: often a host will mistakenly allow requests it shouldn't; audit logs make it easier to discover such mistakes and to clean up afterwards.

It is crucial that an isolation mechanism perform these three checks on every single request—a single hold in the isolation boundary is often enough to completely break any benefits isolation that would have provided.

13.2 Authentication

Since we already had an entire module on authentication using signatures, MACs, passwords, and so on, we will not discuss authentication further here.

13.3 Authorization Policies

In order to authorize requests, we need some sense of permissions—a mapping from *objects* to *principals* that can access them. We call these permissions the authorization *policy*.

Storing policies. We can think of an authorization policy as a gigantic matrix with one row per object and one column per principal. For example, in a file system, we could have one row per file, and one column per user in the system. The entry in column i and row j lists the actions that user i can perform on file j: read, write, execute, etc. A common way of storing this gigantic matrix, for example in AFS, is via an *access control list* for each object.

Setting policies. There are many approaches to setting authorization policies. As always in computer systems, there is no one perfect solution:

- **Discretionary access control: "Owner" of each object sets the policy.** This approach is useful in file systems—each file has an owner and the owner can determine who has access to the file. A problem is that if an attacker hijacks the owner's account (or just one application that the owner runs), the attacker can tamper with the policy for all of the user's files. In addition, it may be difficult for non-expert users to set policies.
- **Mandatory access control: Administrator sets policy.** This approach often is useful in a large organization, when administrators have opinions about which user should have access to which files or systems. A classic example of this is for systems that handle classified data in government systems. Normal users of the system cannot give unprivileged users access to a classified files.

  One limitation of this approach is that it is very coarse grained: administrators may not know exactly who should have access to what.

Systems in practice often use some combination of both of these strategies.

Role-based access control tries to hit some midpoint between discretionary and mandatory access control. In these systems, there is a centrally defined set of "roles." In a university, these could be "Students," "Faculty," and "Staff." The security administrator assigns users to roles. Then application developers determine which roles have access to the application.

Common issues are:

- It is difficult to keep policies up to date as the set of users evolves. Expiring permissions is one strategy.
- Users will complain if they do not have enough permissions, but they will never complain if they have too many permissions. As a result, users often end up with more access than they need from a security standpoint.

13.4 Auditing

We have relatively little to say about this. The most important thing to remember about auditing is that a system should store the audit logs in a container that is separate from the container holding application logic. That is important because if the attacker compromises the application, it should be difficult for the attacker to compromise the logs as well.

13.5 Delegation and Chained Requests

Users often interact with systems *indirectly*. For example, when accessing Gmail, a user's browser first sends a request to the Gmail server asking for new messages. The Gmail server then sends a request to the database to fetch the message data.

For the first request, it is fairly clear that the principal should be Alice: the request is coming from Alice's browser, and therefore should have been initiated by Alice directly. Alice will send some credential to the server, and the server can use this credential to verify that it is really Alice on the other end. For the second request, however, it is not as clear who the request should be from.

One option is to have the request come from Alice. This protects against compromise of the Gmail server—the adversary cannot see all user data. Systems like SSH and AFS follow a strategy like this. Another option is for the principal of this second request to be the Gmail server itself. This helps with isolation among services access the same database: if the Google calendar code is buggy and gets compromised, the first plan would allow an adversary to view Alice's gmail data even is the gmail service was perfectly secure. However, it does not protect other users from a buggy Gmail service.

Compound Principal: "B for A." To achieve something stronger, we can create a new type of *compound principal* that combines a service or device with a user. For example, this server-to-database request could carry a principal of "Gmail Server for Alice". This provides protection against both gmail server compromise and against compromise of other services.

However, it is not as clear how to actually implement this. One option is to continue to have Alice send her credential to the server directly. However, then the server can totally impersonate Alice and we gain little protection. What we would really like is for Alice to give permission to the Gmail server to fetch her emails, but not to do anything else. This is called *delegation*.

Delegation with cryptography. In interacting with the Gmail server B, we may like for Alice (A) to give B permission to authenticate as "B for A" and to do so for only 60 seconds into the future. To achieve this, A can sign a message that outlines the permission it would like to give to B. This signature becomes the proof of authorization. As an example:

$$\mathsf{Sign}(\mathsf{sk}-A, \text{“A delegates to B”}, \mathsf{start} = \mathsf{now}, \mathsf{end} = \mathsf{now} + 60)$$

Google indeed uses a strategy like this. They have a global DoS-resilient HTTP front-end that performs initial authentication. This frontend is then responsible for generating these scoped delegation signatures for each operation that the user would like to do and sending them along to the individual services. These signatures are then used for all following operations.

Capabilities. We may want more fine-grained access control. For example, on Android, the Gmail app may like to delegate permission to a PDF viewer to view an attachment. However, if all the attachments are stored in some common database, we would like to avoid giving the PDF viewer access to view everything in the database. To achieve this, Android (and systems more generally) use a plan called cababilities.

A similar strategy can be seen, for example, in cloud file sharing: when you share a file in google drive, it generates a long random link that allows anyone with access to that link to view that file (and no others). This link itself becomes a *capability*—it allows anyone that posesses it to perform some related action.

14
Isolation

In many settings when building systems, it is useful to *isolate* different components of a computer system. For example, cloud providers such as Amazon Web Services' EC2 run multiple virtual machines, your phone runs several apps, and your browser runs many sites together. If one of these virtual machines (or apps or websites) is malicious or buggy, these platforms would like to protect the buggy component from interfering with the execution of other code in the system. We often refer to separate isolated components as running in separate *isolation domains*.

Ideally, we could have each isolation domain run on separate physical computer. If this were the case, (modulo physical side-channel attacks) one domain would clearly not be able to touch another's state. Unfortunately, the cost of this physical "air gap" isolation is far too high for most practical applications.

Computer systems for high-stakes applications (e.g., classified government data) do in fact use this type of "air gap" isolation.

So, when creating isolation methods, we aim to make it appear as if each domain is running on a separate computer but to do so all on the same computer.

14.1 Defining Isolation

In order to define isolation, we will think about two key properties: integrity and confidentiality.

14.1.1 (Weak) Integrity

One property that we would like an isolation scheme to provide is *integrity*: one domain cannot modifying the state of another domain. To formalize this, let's consider two domains running on a single host, an adversary domain A and a victim domain V. We would like to guarantee that A cannot change the execution of V. We can define something in terms of the *state* of each domain, S_A and S_V. For any pair of starting states (S_A, S_V), after running A, we would like the

new pair of states to be (S'_A, S_V):

$$(S_A, S_V) \xrightarrow{\text{run}A} (S'_A, S_V)$$

14.1.2 (Weak) Confidentiality

We would also like *confidentiality*: an adversarial domain should not be able to read a data from any other domain. We can formalize this by considering two worlds, each with a *different* victim state. After running A in each of these worlds, we would like the resulting S'_A to be identical. That is, for all pairs of victim states S^1_V, S^2_V, running (S_A, S^1_V) results in the same adversarial state as running (S_A, S^2_V):

$$(S_A, S^1_V) \xrightarrow{\text{run}A} (S'_A, -)$$

$$(S_A, S^2_V) \xrightarrow{\text{run}A} (S'_A, -)$$

This definition of confidentiality is often called "non-leakage".

14.1.3 Non-interference: Strong confidentiality and integrity

In our definitions of confidentiality and integrity so far, only the adversary domain runs. In a real system, both the adversary and the victim domain will run concurrently. Ideally, we would like to ensure that our confidentiality and integrity properties hold under interleaved execution of the adversary and victim.

To achieve this stronger notion of security, we can include an interleaving of A and V in one world—we would like for the resulting S'_A to be identical whether or not V runs.

$$(S_A, S_V) \xrightarrow[A,V,V,A,A]{\text{run } A,V} (S'_A, -)$$

$$(S_A, S_V) \xrightarrow[A,A,\ldots,A]{\text{run only } A} (S'_A, -)$$

This definition is often called *non-interference*. We can similarly strengthen our definition of integrity by requiring that S'_V is identical after running V whether or not A is run.

$$(S_A, S_V) \xrightarrow[A,V,V,A,A]{\text{run } A,V} (-, S'_V)$$

$$(S_A, S_V) \xrightarrow[A,A,\ldots,A]{\text{run only } V} (-, S'_V)$$

14.1.4 Non-interference is difficult to achieve

To achieve a non-interference style of isolation, an adversarial process A must not be able to determine whether there is a victim V process

running alongside it concurrently. The challenge, though, is that whenever the adversary and victim *share limited resources*—such as CPU, RAM, network bandwidth, hard disk space, etc.—it is almost always possible for the adversary to determine whether there is a victim process running concurrently.

The information leakage across isolation boundaries as a results of resource contention is one type of *side channel* or *covert channel*. There is a vast literature on how to construct and exploit various types of side channels that leak information from a victim to an adversary.

Example: Memory Allocation. A real system will have some bound on the amount of memory available to it. After this memory is used, the system will be unable to allocate any additional memory. Consider a system with 16GB of memory and a victim process that allocates memory based on the value of some secret:

```
int secret;
malloc(secret);
```

An adversary could repeatedly try to allocate memory until the system tells them they cannot. By keeping track of the amount of memory they were able to allocate, the adversary can learn the secret: if the adversary is able to allocate 15GB, the adversary will know that the secret is 2^{27}, as the victim must have allocated 1GB.

Example: Execution Time. Consider another victim that runs some computation that takes a variable amount of time to finish depending on the value of a secret. By keeping track of how long the adversary takes to finish, the adversary can learn how much execution time the victim running on the same system takes to finish. Using this information, the adversary may be able to learn information about the secret. This type of information transfer are often called "timing channels", and can be quite tricky to work with.

There are effectively three ways to deal with the fact that non-interference is generally impossible to achieve with shared limited resources:

- **Strictly partition resources to prevent contention.** Each isolation domain could run on a separate physical machine, or we can provision the resources on a machine are partitioned in such a way that there is never contention for resources between isolation domains.
- **Prevent isolation domains from detecting resource contention.** Another (more practical) approach is to restrict the types of programs that can runs in such a way that prevents the programs in the system from detecting contention in shared resources. For example, if all programs in an isolated system are deterministic functions with no access to the outside world—no system calls, no networking, etc.—then programs may not be able to *detect* resource

contention when it exists. In most implementations of isolation (e.g., virtual machines in a cloud environment), isolation domains absolutely need access to the outside world, so this approach is rarely useful.

- **Give up on non-interference.** Most isolation mechanisms opt for this solution. Rather than trying to achieve strict non-interference, we aim for some "good enough" notion of isolation. Linux, for example, does not attempt to achieve strict non-interference between processes running on the same physical machine.

14.2 Implementing Isolation

In principle, implementing isolation in a system involves three main steps:

1. identify the state for each domain,
2. identify operations that access state, and
3. ensure that state-modifying operations can only read/write the state within an isolation domain.

The challenge is performance. An isolation mechanism will have to perform checks to ensure that components in one isolate cannot influence another. The game in isolation is to provide strong isolation at the minimum possible cost.

We will start by considering simpler isolation mechanisms and then look at more sophisticated ones.

14.2.1 Emulation

One simple way to implement isolation is to have an interpreter that executes isolated programs (e.g., x86 programs). The interpreter inspects each opcode in the program one at a time, and then implements the operations on the isolated state that the opcode indicates. While processing each opcode, the interpreter enforces checks on the isolated program to ensure that it can only modify its local state.

Some JavaScript engines ("runtimes") in web browsers use emulation for isolation. The Python interpreter is another example of emulation-based isolation. The runtime for these languages is designed such that the code can access only memory that belongs to the domain.

Benefits Emulation can be simple to implement and provides "good enough" isolation for many applications.

Downsides Emulation can be slow: to execute each logical opcode in the emulated program, the emulator may have to run a large number of physical instructions. Emulation can be inflexible: emulated programs may not be able to take advantage of special-purpose hardware, unless the emulator explicitly grants access to these devices to emulated programs.

14.2.2 Time Multiplexing

Another effective isolation strategy is to only allow the code from one isolation domain to run on the hardware at once. When the isolation mechanism switches from one isolation domain to another, it writes the state of the hardware (e.g., registers, RAM) to storage, clears the state of the hardware (e.g., zeros the contents of memory), and loads the next program to run into the machine.

For example, gaming consoles implement this form of isolation: one game runs at a time and has almost full control of the hardware during this time. The state of the running game cannot tamper with the state of a non-running game.

The security of this isolation scheme requires the isolation mechanism to protect the stored state of each isolation domain, and to somehow protect the code that switches between them.

Benefits Time multiplexing is relatively simple to implement, and it can give programs in each isolation domain almost full control of the hardware (e.g., graphics hardware in game consoles)..

Downsides There can be a high cost of switching execution between between isolation domains. Since the isolation mechanism clears the state of the hardware during context switches, storing and restoring the state can be costly.

14.2.3 Translation (Naming)

Translation is another common isolation mechanism in computer systems. In a system using translation for isolation, an isolated program may not access hardware resources (such as memory or files) directly. Isolated programs can only access resources via pointers controlled by the isolation mechanism. By construction, each process in an isolation domain can only name resources inside of its isolated context.

The canonical example of naming/translation for isolation is *virtual memory*. In a system using virtual memory, the isolated domains cannot read/write physical memory directly—they can only read/write to virtual memory addresses..

To prevent one isolated program from accessing another's memory, the isolation mechanism ensures that the valid virtual addresses in each domain point to a separate portion of physical memory. Modern CPUs have special support for implementing virtual memory to make the virtual-address translation as fast as possible.

Other examples of naming/translation isolation in computer systems are: file descriptors in Linux and virtual LANs in networking.

14.2.4 Example: Isolation in Virtual Machines

When we run several virtual machines on one physical machine, we want each virtual machine to run as if it had its own physical CPU, memory, and devices, but we want to run all of these virtual machines on a single physical machine. For performance, we would like to run instructions from the VM directly on the host CPU—but we need to make sure, for example, that the VM does not access memory belonging to another VM. To achieve isolation and good performance, systems today use several effective techniques.

Here is how a virtual-machine monitor handles the three questions that an isolation mechanism must answer:

1. **What is the isolation domain's state?** A few important pieces are:
   - the contents of memory,
   - the values in the CPU registers,
   - the data on disk.
2. **What operations can a program perform on isolated state?** Virtual-machine monitors need to handle:
   - CPU instructions that modify register states ,
   - CPU instructions that modify memory, and
   - operations to read and write devices.
3. **How does the virtual-machine monitor ensure isolation?**
   - To handle updates to the register state, the monitor uses *time multiplexing*: one virtual machine runs on the CPU at a time and controls the CPU's registers.
   - To handle accesses to memory, the monitor uses *naming/translation* via virtual memory.
   - To handle device operations, the monitor uses *emulation*. When a virtual machine executes instructions that would cause device I/O, the CPU jumps into some dedicated code in the monitor that emulates these device operations. This is slow, but since device I/O is usually costly anyways, the overhead is isolation is tolerable.

14.2.5 Software interposition

A final isolation technique that we will discuss is *software interposition*. Say that an isolated program wants to run the following code:

```
var a = b[c];
var f = ....;
f();
```

When using software isolation, a compiler can insert checks into this isolated program to make sure that the memory access `b[c]` does not access out-of-bounds memory:

```
if c >= b.size: error;
load b+c -> a             (***)
```

After inserting these checks, the isolated code can run directly on hardware, since the checks ensure that the isolated code cannot touch any state outside of its isolated context.

A central challenge of using software interposition is handling function calls. When calling the function `f` in the code snippet above, the hardware might execute an instruction like:

```
jump *f
```

which would execute the code at the location stored in memory location `*f`.

If an isolated program can set the value in `*f` to the location `(***)` in the snippet above, the isolated program can skip the safety checks on the memory accesses.

15
Software Trust

The central question of this chapter is:
How do we know whether a system is running the software that we expect it to be running?

This question comes up both when we are interacting with a machine in person (e.g., typing a passcode into our phone) or across a network (e.g., when sending sensitive data to a far-away server).

A separate question, which we will discuss in future chapters is: How do we ensure that the software itself is "good" or "bug-free?"

Threats to software integrity. There are a number of threats that might cause a machine to be running unexpected software:

- **Malware.** An adversary may install bad software onto the laptop, such as a keylogger.
- **User error.** A user may inadvertently install malware onto a machine.
- **Software supply-chain problems.** An adversary may inject malware into a real app's libraries, by tricking or coercing a developer.
- **Malicious updates.** An adversary may trick the software update process, converting a real piece of software into a malicious one.

A classic way to trick users into installing malware is to show them a warning (e.g., on a webpage) that says "Your computer is infected. Please download and install this anti-virus software."

The software supply chain. There are many steps that take place between the development of a piece of software and its use:

1. Developers write code. Their code may use many third-party libraries.
2. Compilers build and package the application.
3. The software vendor distributes binaries over the network.
4. Users download new software.
5. Users download software updates.
6. Users launch applications on their devices.
7. Running applications interact with remote servers running code.

15.1 Library Imports

15.1.1 Example: Python Imports

In Python, importing a library requires downloading a library using the `pip` package manager:

```
pip install requests
```

After that a user can import the package into their code like this:

```
import requests
```

Behind the scenes, the PyPi service maintains a database that maps package names (e.g., `requests`) to a piece of code. When you type `pip install requests`, the `pip` program fetches the code from the PyPi repository and installs it on your machine.

Benefits Benefits of this approach are:

- The centralized service makes it easy for users to identify packages.
- It is relatively easily for users to discover and download software updates.
- Developers do not need to run their own code-distribution service.

Downsides Downsides of this approach are:

- The centralized update service is a single point of failure: if an attacker is able to change the code in PyPi, it can infect a large number of machines at once.
- The end user has no idea who actually produced the library code. The user only is able to specify the package name.
- In Python, the naming scheme is ambiguous: if there is a public package and a private package both with the name `requests`, it's not clear when a user imports `requests`, which one the user wants to import.

15.1.2 Example: Go Library Imports

The Go programming language takes a slightly different approach to package management. In the Go programming language, a user imports a library/package by specifying the URL of the package's Git repository. For example, an import might look like this:

```
import "github.com/grpc/grpc-go"
```

When compiling the code, the user's PC will contact the server at the given URL over HTTPS (verifying the server certificate via TLS) and download the software bundle. On the other end, when a library developer wants to update their library, they do so by interacting with the hosting server via HTTPS and whatever authentication the server has set up—credentials, maybe two-factor authentication, etc.

This has some good features: the server name is explicit so there is no ambiguity about packages and the decentralized nature of specifying individual URLs avoids the necessity for a central server that attracts attacks. However, this requires trusting the server hosting the library to secure the update process and distribute software honestly.

15.1.3 More Explicit Trust: Code Signing

In each of these approaches, if an attacker can cause the user to download a bad package *without* compromising the package developer. In particular, if the attacker can compromise Github, it can cause Github to distribute malware to end users.

To prevent this attack, a library developer could sign their software using their private key and include the signature with their software package. To verify that a package is authentic, the application developer's PC can check that the signature is valid.

Of course, with any signature-based plan the mechanism for public key distribution is crucially important. In the software distribution case, the only reasonable plan is likely a Trust-on-First-Use based one which accepts the first public key it sees but verifies that future software updates use that same key. This protects against an adversary taking control of, for example, the application's Github repository after the end user installs the softare once. However, key management is hard, so this is not widely used in practice.

15.2 Building Binaries

In order to run software on our computer, it is necessary to convert the source code (which is, at least in principle, manually auditable) into a binary that is much more difficult to audit. Since compiling software is computation-heavy, most application developers typically compile their software and distribute the binary to their users. If an attacker compromises the build server (or is able to backdoor the compiler), then the attacker can cause users to execute bad code, even if the attacker does not compromise the application developer itself.

The XCodeGhost attack is an example of how an attacker can insert a backdoor in a build system and exploit it to distribute malware.

Reproducible Builds. One promising approach to the problem of ensuring that a binary is the faithful compilation of a piece of software

is called "reproducible builds."

If a build process is reproducible, the function that turns a set of source-code files into a binary is a deterministic function: if two different people compile the same set of source-code files, they will get exactly the same binary—the two will be bit-for-bit identical. This allows anyone to *audit* a build: to check that a build server did its job correctly. In addition, having multiple independent parties build the same piece of software (and sign the result) can give an end user some assurance that the build server behaved correctly.

Implementing reproducible builds is not trivial. Traditional compilers introduced many sources of non-determinism—not necessarily for any particular reason, just for convenience. Creating reproducible builds requires eliminating all of these sources of non-determinism, even across multiple versions of the compiler.

As of today, the Go programming language now supports reproducible builds.

15.2.1 Juggling multiple versions of a library

Once a binary exists, the next step of the process is to distribute that software to user devices. Typically, there are many different versions of a piece of software around. When a user wants to install a piece of software, they typically need to specify which version of the software they want.

For example, in Python a user can specify a version of a package that they would like to install when they run `pip install`. If an attacker compromises the PyPi server, it can serve up any code it wants to a user asking for a particular version of a library.

In contrast, in the Go programming language, when a users imports a package, the `go get` software will store a hash of the downloaded code in a file called `go.sum`. If an attacker later compromises the server serving the package (e.g., Github), the `go get` command will refuse to install the package unless its code matches the stored hash value.

This is an example of "trust on first use" in the context of code installation.

15.3 Installing & Updating Software

Once a software developer finishes writing an application, it builds and distributes it. When a user installs an application—e.g., by downloading it from a website or fetching it using a package manager—how does the user know that it got the authentic version of the software? As usual in systems design, there are many possible strategies.

Application Developer Signs Package (Android Apps). One possible option is to have application developers sign the software that they produce. When application developers distribute their software, they attach a their signature to it. This way, it does not matter how a user obtains the software—a user can download an application bundle from any server and know that it came from the developer who owns the corresponding secret key.

When a user first installs a piece of software they need to somehow obtain the software developer's public key. Public-key distribution, as always, is messy: trust on first use is a common strategy.

Once the user has the software developer's public key, the user can easily verify that future updates to the software came from the same developer. (To do this, the user can just check signatures on the updates using the app-developer's public key.)

An important caveat is that signatures do not guarantee freshness: once signed, a package is always valid.

Repository Signs Packages. For systems with a central repository, another plan is for the repository to sign packages. This again allows the user to fetch the signed packages from untrustworthy sources—from a content-distribution network, for example.

In addition to signing the packages, the repository typically signs a timestamped manifest of the latest package versions. This allows a user to check that they are not only getting the right software but also that they are getting the most up-to-date software.

Many Linux package managers, such as `apt`, `pacman`, and `rpm`, use signatures to integrity-protect packages.

Third-Party Validator Signs Packages. Yet another option that does not require a single central repository is to have a trusted validator sign packages. This involves sending the source code and package to a third party, who will then perform some inspection of the package and, if it deems a package to be worthy, provide some signature over that package that verifies that the validator thinks the package is trustworthy.

A number of software platforms use this strategy for protecting binaries. On Android, there is no requirement to install apps from the Google Play Store, but Google provides a service that inspects packages and attaches these signatures if the package passes. Similarly, Windows uses a validation plan for its device drivers.

Binary Transparency. One different plan to help involves an audit log that keeps track of all published binaries.

This helps prevent in particular targeted attacks—for example, if

some adversary has a specific target in mind and compromised the distribution of the Linux kernel, they would likely be immediately noticed if they introduced a backdoor into Linux for the whole world. However, if they were able to introduce a backdoor and distribute that backdoored version only to their target, the adversary would be much more likely to evade detection. If clients check their received binary against the publicly available one before installing it, this personalized attacks can be avoided—if the attacker wants to change the binary for someone, they will need to change it for everyone.

15.4 Booting the System

15.4.1 Secure Boot

In order to actually run an application, we rely on large amounts of software running on our computer, from the applications themselves to the operating system that supports them. If the operating system itself is compromised, for example, the modified OS could undermine all of the defenses we just discussed. "Secure Boot" is one strategy for getting some partial protection against these attacks.

Devices using secure boot have a small amount of read-only memory (ROM) that contain a small piece of code that runs on boot. This boot-ROM code has a signature-verification key baked into it. There is no way to change this key—it is a fixed part of the hardware Booting then involves several steps:

1. On boot, the CPU will begin running the hardcoded Boot ROM code, which has a signature-verification key $\mathsf{vk}_{\mathrm{ROM}}$ hardcoded into it.
2. the Boot ROM will load the code for another layer called the *boot loader*. The boot ROM will then verify that the boot loader code carries a correct signature that verifies under $\mathsf{vk}_{\mathrm{ROM}}$. If the signature is valid, the boot ROM code will begin executing the boot loader. The bootloader has another signature-verification key ($\mathsf{vk}_{\mathrm{bootloader}}$)) baked into it.
3. The boot loader will load the code for the operating system and verify it using $\mathsf{vk}_{\mathrm{bootloader}}$. If the signature is valid, the bootloader will redirect control to the operating system.

This way, the system can verify that only boot loaders approved by the hardware manufacturer can run on the machine. These boot loaders then can verify that only trusted operating systems are executed.

Many systems use secure boot: iPhone, Android, chromebooks, game consoles, and UEFI secure boot on PCs.

In some cases (e.g., UEFI secure boot) secure boot is a mechanism to protect against malware that tampers with the operating-system code. While the malware may be able to compromise the running machine, after rebooting the machine, the user has some assurance that it is running an uncompromised operating system.

In other cases (e.g., game consoles) secure boot is a mechanism to prevent the device owner from installing a non-standard operating system on the device. Game-console vendors often sell the console hardware at a loss, and they make their money back by selling game software. They have a strong incentive then to prevent users from buying game consoles and using them for non-game purposes.

A number of researchers have used PlayStation 3 consoles for brute-force password-cracking and cryptanalysis (e.g., factoring). Game consoles often have a large number of CPU/GPU cores, which make them appealing hardware for applications that benefit from massive parallelism.

15.4.2 Measured Boot

Secure boot assumes you know which key is needed to sign the software. But what if the hardware cannot determine which software is good and which is bad? The idea is to measure what software is being booted. This involves hashing the bootloader, then hashing the OS kernel, and finally hashing the OS image. This idea is called measured boot.

While measured boot cannot prevent bad software from loading, different software generates different secrets. The system contains a secret that is durably stored in hardware and remains consistent across reboots. When the system boots up, it derives a secret key based on this hardware secret, using a derivation method that incorporates the hashes of the bootloader, and OS kernel, and perhaps other software.

The OS then uses this secret key to decrypt its data and authenticate to remote servers. Booting a different OS —- perhaps due to malware corruption —- will result in a different key being generated, and this is detected when the data cannot be decrypted.

15.5 Secure attention key

When to approach a terminal and type your bank password into it, how do you know that you are typing the password into the banking app or into some other app (e.g., the flashlight app) on the machine?

A traditional approach to address this problem is a *secure attention key*: there is a special button or combination of buttons that trap into the operating-system kernel code—interrupting whatever application that may be running.

Windows workstations, for example, required users to type the keyboard combination `CTRL-ALT-DEL` to bring up a login prompt. If a user entered this combination while an app was running, the

operating system would interrupt the application and open the legitimate login screen.

Pressing the "Home" button on many smartphones has the same effect.

16
Hardware Security

So far while discussing platform security, we have considered only our software. However, software must run on some hardware, and the security of this hardware is similarly vital—if an attacker can undermine the security of our hardware, it does not matter how strong our software security is. Luckily, hardware is typically much more difficult to attack than software, but powerful attacks are still possible.

As a running example for this chapter, consider the example of a certificate authority that signs certificates given that some policy is met.

An equivalent example is a cryptocurrency wallet—transactions in the cryptocurrency are authorized by a signature over a transaction message.

This server will accept requests and respond with a signature over that request if some policy is met. For example, an MIT CA might enforce a policy like "I will sign only certificates for `*.mit.edu` domains". For this CA server, we can achieve some nice security properties:

- We can prove the security of the signature scheme used under concrete assumptions such as the hardness of discrete log.
- We can verify that the cryptography implementation faithfully implements the signature algorithm on some ideal hardware model.

In order to actually be used, however, we must first buy a computer, load the code onto it, and run it (likely on a machine running other software). This process is outside of the nice security properties with achieved about our CA program. In cryptography and in software, we are able to develop clean characterizations of the power of the attacker. In cryptography, we assume that our adversaries are bounded by probabilistic polynomial time. In software, we can model our attacker as choosing arbitrary inputs to our software and accurately capture an attacker's power. Once we consider the hardware, however, it becomes much more difficult to meaningfully define the attacker's power.

When we consider the real hardware that the system is running on, there are many attacks that we must consider.

16.1 Hardware Bug

Much of what we have discussed so far has been cryptography schemes that rely on trusted parties knowing some randomness that the adversary does not know. We have assumed that we have some source of "true" randomness to use, for example, as a seed for a PRF. When actually implementing cryptographic algorithms in a computer, we need to actually materialize this "true" randomness. However, it is not clear where this randomness should come from: computers are designed specifically to behave as they are instructed by the programs they run. Common solutions are to measure statistics about the environment that should be hard for an adversary to predict. For example, devices may use combinations of:

- Keypress timings
- Packet timings
- Clock
- Temperature sensor

All of these require "accumulating" randomness from the environment. A common hardware bug on embedded devices is to generate keys when the randomness source has not accumulated enough random measurements from the environment. For example, if a network card generates a cryptographic key right after boot, this key may be predictable if an attacker is able to accurately guess at the values of the randomness source.

To help with this, many devices include specialized hardware that uses some special circuit to generate randomness by measuring randomness inherent to the universe.. Of course, developers must then use this randomness—a common error is to use insufficient randomness, such as the time, instead of this hardware randomness.

On Intel CPUs, this takes the form of the `RDRAND` instruction

16.2 Attacks without Physical Access

Perhaps the most concerning attacks are those that do not require physical access to a machine.

16.2.1 Cache Timing Attacks

One major goal of operating systems is to provide isolation between processes. Even if an attacker is able to run some software on the same machine as our signing process, we would like to guarantee that an attacker can not read, for example, the signing key used by

our signing process. However, the attacker and victim code both run on the same CPU, and the victim may leave traces of secrets in the state of the CPU.

For example, consider that our signing process runs and, depending on some secret value, either loads the value at memory address A or does not. If the victim loads this address,the CPU will copy the value into the cache to speed up future accesses to the value. An attacker process that runs next can try to access this same memory address and measure how long it takes to read the value. If the victim read that value, the access will be fast since it comes from the cache, but if not, the data will have to come from the much slower main memory. From the difference in this timing, the attacker can learn about the victim's access pattern, which may reveal data about the victim's secret.

Because of attacks like this and others, it is important to write secure code such that it does not branch on secret values.

This may seem like an unlikely attack since both the victim and attacker must have access to the same memory location, which process-level isolation should prevent. However, operating systems perform something called deduplication that can be cleverly taken advantage of to achieve this: if the victim process uses OpenSSL, the attacker process can also use OpenSSL. The operating system will see that both processes are linking the same library, and may map a section of virtual memory for each process to the same physical memory.

16.2.2 Rowhammer

Data in a computers memory is stored in what is effectively a grid of capacitors. These capacitors do not store values indefinitely, and so their values must be refreshed every so often (commonly every 64 milliseconds). Reading a chunk of memory drains the corresponding capacitors a bit, and they must then be rewritten. Memory is read one row f this grid at a time. Reading a row drains the corresponding capacitors, requiring them to be rewritten. This rewriting involves voltage fluctuations, and since modern memory is so dense, these voltage fluctuations can cause neighboring rows to discharge more quickly that the refresh interval is equipped to handle. Surprisingly, reading a single row repeatedly can cause bits in an adjacent row to flip.

An attacker could take advantage of this by "hammering" a memory location, causing a bit to flip in memory that belongs to another process or to the operating system. In some cases, this was enough to allow the attacker to learn a secret or bypass isolation, etc.

16.3 Physical Attacks

If an attacker has physical access to a device, an entirely new class of attacks becomes possible. They can measure the device, introduce faults to the device, and more.

16.3.1 Probing Attacks

An attacker with physical access to a device can measure many things about the device's behavior that a remote attacker could not. For example:

- Place probes on the pins of a chip
- Measure power consumption of a chip and watch for patterns
- Measure optical emissions of a chip with an electron microscope
- Measure RF emissions from a chip
- Monitor the blinking light on an internet router

The information that an attacker can learn from attacks like this may be limited, but even very slow information leakage can be enough to leak a key in a relatively short amount of time. Protecting against these types of attacks is difficult since the attacker can do such a broad range of things. However, if we make certain assumptions about the attacker's power, we can achieve principled solutions.

Defense against Probing Attacks. One assumption that may be reasonable to make is that an attacker can probe at most t wires of a circuit. By using techniques like secure multiparty communication, it is possible to build a circuit that implements something like a signature scheme, but that does so without leaking anything about the secret given this assumption.

16.3.2 Fault Attacks

An attacker can also introduce faults that the system was not designed to handle. For example, they can point a heat gun or a laser at the chip, hoping to cause some bit flips. If they are successful, these bit flips may leak a secret key or corrupt a kernel data structure, allowing the attacker to take over the system.

Designers of satellite systems have to think about similar attacks, but in their case the attacker is the sun! Cosmic rays carry enough energy to flip bits of CPU registers or memory.

16.3.3 Supply Chain Attacks

When we buy an device, we assume that the hardware inside is not working against us. However, there is a long chain of steps that happens before the device gets to us—it is built in the factory, packaged, mailed across the world to a retailer, stored in a warehouse, packaged and mailed again, and so on. An attacker that has control over any of these steps could intercept the device on its way to you and modify it somehow. For example, they could:

- Modify the randomness source to something predictable
- Preload keys that the attacker knows
- Add extra input/output interfaces
- Add or enable management interfaces

There are no great solutions to defend against this—inspecting the chips is impossible since they are so small, building a device yourself is much too hard, and so on. One solution that can improve the protection is to build a system out of n identical devices and use a strategy like secure multiparty computation to protect against cases where at most $n - 1$ of these devices is compromised by a supply chain attacker.

17
Case Study: iOS Security

There are several things we might worry about when it comes to the security of a smartphone, spanning software, platform, and hardware security:

- Malicious apps that steal contacts, eavesdrop on calls, or steal you credit card data
- Someone stealing a phone and extracting secret data
- A non-Apple operating system that is loaded on a phone (i.e. Jailbreaking)
- Malicious chips installed in the factory, en route to the store, or during repairs
- People selling fake phones as real iPhones

The iOS platform is a very well-designed integration of the topics that have been covered in the platform security section and which addresses many of these considerations effectively.

17.1 App Security

In any platform, there is a balance between the "openness" of the platform—who is able to run software on it—and the access that software has to sensitive data. Of course, the most secure platform is one that doesn't do anything at all: it is very closed and provides no access to sensitive data. More practical systems have a different balance:

- A typical laptop is very open—anyone can write software for it—and every application is able to access the entire filesystem.
- The web is still very open, as any website can include JavaScript code that will execute on your machine, but access to sensitive data is tightly locked down via language sandboxing.

iOS improves its security story by locking this down: applications are given some (checked) access to sensitive data, but are very

closed—applications must go through a review process and only approved developers can write software for iOS.

In addition to security policies, the review process checks that applications follow policies in place for business reasons, such as the restriction that digital purchases must go through apples In-App Purchase mechanism.

iOS apps run in a sandbox that does not provide access to a shared filesystem and that allows communication between apps only though limited APIs provided by the operating system. If an application wants to interact with data that is controlled by the operating system, such as VPN configurations or health data, the application developer must ask for access to special APIs and have that access approved as part of the review process.

17.1.1 Things Can Still Go Wrong

Even with these checks and the sandboxing, malware can slip through the gaps. As we discussed earlier, XCode Ghost was a compromised version of the XCode compiler distributed via mirrors behind China's firewall that inserted malware into apps developed by honest developers. This malware was not found in App Store reviews, allowing apps that people expected to be honest to run arbitrary malicious code. Even with the sandbox, an app has quite a bit of access to do potentially sensitive actions:

- Learn Country
- Learn Language
- Learn UUID, before recent privacy changes
- Read and modify clipboard contents, including copied passwords or credit card numbers
- Open a URL that points to a phishing webpage

Despite this, isolation buys a lot of security. A malicious app that makes it through review cannot access your text messages, browser history, etc.

17.2 iOS Secure Boot

For security and business reasons, Apple would like to ensure that an Apple-signed operating system is running on the phone. This prevents a malicious actor from distributing a backdoored operating system and convincing people to install it on their phone and against malware that tries to persistently modify the operating system. It also prevents users from installing a customized operating system on their phone, bypassing the Apple restrictions on apps and other policies.

To achieve this, iOS uses a secure boot system as described in the last chapter. Each phone ships with a Boot ROM that cannot be changed that is burned with some public key for a secret key that Apple knows. This boot ROM is responsible for verifying that this secret key signed the code for a low-level bootloader and running

that bootloader. This bootloader will then verify and check the operating system kernel. This allows the bootloader and the kernel to be updated as necessary, but places a root of trust in the boot ROM that cannot be updated. If the bootloader is updated by anyone besides Apple, however, the boot ROM's signature check will fail and the boot ROM will refuse to run the bootloader.

Many device owners that wanted to customize their devices sought to modify the operating system to add new features. This process generally is referred to as "jailbreaking".

17.2.1 Checkra1n Jailbreak

One set of exploits that constituted a jailbreak was named `checkra1n`. This took advantage of complex code in the unchangeable boot ROM—the devices supported running code directly via USB, bypassing the low-level bootloader and the OS kernel, which meant that the boot ROM contained code to act as a USB peripheral. USB code is quite complex, and as with most complex code, it had bugs. `checkra1n` took advantage of these USB bugs to trick the phone into executing arbitrary code.

Because the Boot ROM can never be updated, it was impossible for Apple to fix these bugs: this jailbreak will work forever on the devices that had the bug.

However, these bugs did not allow the jailbreak to bypass the signature check entirely, so this exploit needed to be run on every boot—if you tried to reboot your phone without running this, it would no longer be jailbroken.

17.3 iOS protection for data at rest

If a phone is stolen, it would be nice if the theif could not learn any sensitive information from the device. As with any time that we want to hide data, encryption is the answer here—a simple solution is to encrypt all data on the phone with 128-bit AES. However, this does not tell the whole story: in order to decrypt the data, the key for this encryption must be stored somewhere. We can't store the key in normal flash memory, since then anyone could use it to decrypt the data. We also can't use the only secret that the user knows, their 6-digit PIN, since it is much too short to be an AES key.

Even an approach like using a PRF based on the key to extend it into key for AES will not be secure—6 digits is so short that it can easily be brute-forced. Instead, recent iPhones use a special chip called a "secure enclave" that holds the key and provides access to it only if the user enters the correct PIN. Doing so allows the secure

enclave to enforce strict guess limits that prevent brute-forcing the PIN.

The secure enclave is essentially another processor that runs its own operating system. It uses a similar secure boot system to prevent tampering with the secure enclave's operating system, but it also uses *measured boot* to derive the secret encryption key from the contents of the OS being run—if an attacker modifies the secure enclave's operating system, the encryption key will change and the attacker will not be able to decrypt the phone data.

Importantly, the secure enclave has access to two things that the main application processor does not. First, on the first boot: the secure enclave generates a long-term secret unique ID and burns it into internal fuses. The enclave also has access to a secure NVRAM module that has a limited amount of secure storage with support for real deletion. This secure storage contains the root encryption key itself, a hashed version of the user's PIN salted with the UID, and a guess counter that keeps track of how many incorrect guesses have been made.

In typical storage, deleting data does not really delete the data—instead, it marks it as deleted and indicates that the operating system should overwrite it in the future. However, if someone inspects the storage directly, they are likely to be able to recover the data that was deleted. Apple's "effacable storage" that is used for the secure enclave supports deleting data such that it cannot be recovered.

The communication between the secure enclave and the secure storage is also encrypted with a key burned into the enclave and into the secure storage. This prevents an attacker with a probe on the wire from learning the data.

When put together, these elements enable the following process:

1. User enters a PIN
2. iOS passes the PIN to the secure enclave
3. The enclave enforces some delay after each guess
4. The enclave passes $H(\text{PIN}, \text{UID})$ to the secure storage.
5. Secure storage uses included logic to check whether the hashed PIN matches the stored hash.
   - If correct, return the root AES key and zero the guess counter
   - If incorrect, increment the guess counter. If the guess counter is too high, erase the key from the effacable storage.
6. If the PIN was correct, the enclave passes the key returned from the storage to the AES engine.
7. The application processor sends data to the AES engine to be encrypted or decrypted.

Note that the application processor never sees the AES key—it is seen only by the secure enclave and the (hardware) AES engine.

17.3.1 Biometric Unlock

For convenience, however, iPhones do not require entering a PIN on every unlock. A PIN unlock is always required on the first unlock after boot, but afterwards they allow unlocking with a Biometric such as Face/Touch ID. The phone includes dedicated biometrics hardware that is responsible for reporting the "hash" of the measured face to the secure enclave. The secure enclave then checks this against the stored one, and unlocks the device if there is a match.

A possible attack, then, might look something like the following: steal a phone that has been unlocked at least once and relocked (almost always the case). Then, before it runs out of battery and shuts down, replace the Face ID chip with a malicious one that always reports the correct hash. Without protection against replacing the biometric hardware, an attack like this would allow an attacker to access all the data on your phone. To address this, iPhone include yet another shared key between the secure enclave and the biometric hardware: if the Face ID module is replaced, the keys will not match and the phone will refuse to unlock.

Part IV

Software Security

18
Software Security

Software vulnerabilities are at the root of some of the most serious security failures in real-world software. A huge number of security issues come from software-implementation bugs. A rule of thumb to keep in mind is that, for reasonably carefully-written code, there will be around one bug for every 1000 lines of code. Many of these bugs may seem minor, but surprisingly, almost any kind of bug can be used in a security exploit and lead to compromise—even bugs in code that may not seem security-critical. One principle to remember is then:

"Any bug can be a security bug."

In terms of practical consequences on deployed computer systems, software issues are perhaps second only to phishing attacks.

A common strategy to exploit seemingly benign software bugs is to string together a number of benign bugs into an exploit that does something very bad from a security perspective.

Protecting the security of computer system thus requires eliminating software bugs.

Many bugs can lead to security exploits, but some of the most common types of exploited bugs include memory corruption bugs (buffer overrun, use after free, etc.), encoding and decoding errors, cryptographic implementation bugs, race conditions and resource-consumption bugs.

18.1 Memory Corruption

Memory-corruption bugs show up in languages, such as C and C++, that do not guarantee any sort of memory- or type-safety properites. The two most common types of memory-corruption bugs are: (1) buffer overflows and (2) use-after-free bugs.

18.1.1 Buffer overflow

As an example of a memory-corruption bug, consider the following C code:

```
void f() {
        char buf[128];
```

```
    gets(buf); // write bytes from stdin starting
               // at &buf[0], followed by a '\0'
}
```

This code allocates an array of 128 bytes and then uses `gets` to read a string from standard input into the buffer. The `gets` routine will read the input from until it reaches the end of the string (indicated in C with a zero byte), then it will write the corresponding string into the buffer. Arrays in C have no length information attached to them, so the `gets` code will happily accept an input strength of any length—possibly much larger than 127 bytes.

So, what happens if the input is longer than 128 bytes? The `gets` function simply writes until it finds a NULL character (`'\0'`) in the input. So if the input string is too long, `gets` will simply write past the end of the memory allocated for `buf`. If an attacker gives an input that is longer than 128 bytes, the attacker will be able to write bytes of its choice into the memory after `buf`.

In C, the `buf` array will be allocated on the call stack, which will be laid out in memory like this:

```
... rest of stack ...
---------------
return address of f's caller
---------------
buf[127]
...
buf[2]
buf[1]
buf[0]
---------------
```

Once the `f` function ends execution, the program will jump to the return address sitting after at the end of there buffer. So if an attacker can write data past the end of the `buf` array, the attacker can overwrite the return address of `f` on the stack and the attack can cause the program to jump to and begin executing code at an arbitrary location in memory.

You might think that the fact that this problem is caused in part by having the stack in C grow down, so that running off the end of the buffer can cause the attacker to overwrite the return address. It turns out that similar attacks are possible even on architectures in which the stack grows up. The fundamental problem is that the attacker can scribble over data in the stack.

One simple mitigation is to modify the compiler to refuse to compile programs that use routines such as `gets`. We discuss other mitigations below.

To avoid this kind of *buffer overlow* bug, we need to somehow ensure that `gets` only writes within the bounds of the buffer.

Mitigation: Runtime checks Modern compilers can try to check in real time whether a memory access goes past the end of the buffer

and will crash the program if so. These defenses are imperfect but prevent the most naive type of bug.

Mitigation: Bounds Checking Another mitigation is to ensure that the input is not too large before reading it in. To do this, we can insert a check before writing.

Consider the following slightly more complex code, which receives n records that are each 16 bytes long and writes them into the buffer:

```
void f() {
        char buf[256];
        uint32_t n = get_input();
        for (uint32_t i=0; i < n; i++) {
                // read record i into
                // buf[i*16] .. buf[i*16+15]
        }
}
```

This code will write beyond the end of the buffer if `n*16` is greater than `256`. We then may consider adding a check like the following:

```
#define sz 256

void f() {
        char buf[sz];
        uint32_t n = get_input();
        if (n * 16 > sz) {
                // input too long!
                return
        }
        for (uint32_t i=0; i < n; i++) {
                // read record i into buf[i*16] .. buf[i
                    *16+15]
        }
}
```

However, consider an adversary that inputs data such that `n = 2^30`. If we were computing on paper, our check would work just fine: $2^{30} \cdot 16 = 2^{34}$, which is certainly greater than `sz`. However, `uint32_t` is a 32-bit value and 2^{34} will not fit into the 32 bits allocated for that integer—the computation will *overflow*. It turns out that if you try to compute `n*16` in C when `n` is 2^{30}, the answer is zero! Thus, our check will pass but our code will still write beyond the end of the buffer.

To prevent this type of overflow, the program can explicitly check for overflow—it's tricky to do, but important when accepting user-provided input.

18.1.2 Use after free

Another common type of bug is a use-after-free bug, in which a programmer frees a chunk of heap memory and then reads or writes it after freeing it.

In C this happens when a programmer uses `malloc` to allocate some memory, then calls `free` to free it, and then accesses it. An example piece of code with this bug is here:

```
void f() {
  char *req = malloc(1024);
  int err = read(0, req, 1024);
  if (err) free(req);
                            // ***
  if (err) printf("Error␣%d:␣%s\n", err, req);
}
```

If some other thread in the program calls `malloc` when the code is a point `***`, the other thread of execution may use the array `req` for something else. Then the `printf` line could print out some other contents of memory—possibly exposing cryptographic keys or other sensitive data.

These bugs are very difficult to track down since it is difficult for a compiler to figure out which memory a piece of code should or should not have access to. One way to defend against these bugs is to use a programming language, such as Rust, that explicitly associates a "lifetime" with each piece of memory and can prevent code from accessing free'd memory.

18.2 Encoding Bugs

Another common source of security bugs comes from encoding and decoding data from and to language data structures.

18.2.1 SQL Injection

Most web sites that we interact with consist of some application code—for example, a Flask app—that communicates with a database via SQL queries. For example, a phone-number-to-name lookup site would likely use SQL queries that look like

```
'SELECT name FROM users WHERE phone = "6172536005"'.
```

When accepting a phone number, stored in variable `phone`, from a user, the same query might look like:

```
/* WRONG!!! */
'SELECT name FROM users WHERE phone = "' + phone + '"'.
```

The problem with using string interpolation is that an adversarial user can supply a phone number like

```
123"; DROP TABLE users; "
```

The SQL engine will then receive the query:

```
'SELECT name FROM users WHERE phone = "123"; DROP TABLE
users;',
```

which will have the effect of deleting the `users` table.

The principled way to solve this problem is to have a strategy for *unambiguously encoding* data. In SQL, if you have a quote character `"` in a data string, the programmer writes it as `\"`. This is called "escaping" a string. A SQL library can automatically escape characters such as quotes, but escaping is not as easy as replacing each quote with its escaped equivalent—you need to worry about escaping the escape character "`\\`" and all sorts of other subtleties.

Modern libraries for interacting with databases perform escaping automatically to avoid these "SQL injection" attacks.

18.2.2 Cross-Site Scripting

Another common behavior is to take input from the user via a form, save it to the database, and later render that input to another user as HTML. For example, a social media site will have each user enter their name when the sign up, and may use some code like the following to render another user's list of friends:

```
def render_friends(friends: List[str]):
        print("<h3>Friends</h3>")
        print("<ul>")
        for name in friends:
                print("<li>" + name + "</li>")
        print("</ul>")
```

If a friend's name is something expected, like "Alice" or "Bob", this works fine. However, what if a friends sets their name to something like `<script>send_to_adversary(document.cookie)</script>`? Now, the rendered HTML will look something like the following:

```
<h3>Friends</h3>
<ul>
        <li>Alice</li>
        <li>Bob</li>
        <li><script>
                send_to_adversary(document.cookie)
        </script></li>
</ul>
```

This `script` tag runs the contained Javascript code in the viewer's browser. Anyone who views a friends list with this adversary in it, then, will have their authentication cookie sent to the adversary, potentially allowing the adversary to log in as that user.

The core of this issue is similar to the SQL injection attack: an attacker is able to insert code that the victim's browser will run. To prevent this, the solution is again escaping: we typically replace the angle brackets (< and >) used to denote HTML tags with the sequences `<` and `>`. Now that `&` has a special meaning, we also must escape it as `&`. Modern web frameworks have "templating" systems that automate this escaping process.

18.2.3 Decoding: Android Apps

The application-installation process on Android also suffered from decoding errors. Apps on Android (`.apk` files) are just renamed ZIP files.

Apps shipped with a signature. When installing an app, Android would first check that the contents of the ZIP file match the signature. If the signature checked out, Android would then install the app. However, the signature checking code and the installation code used different ZIP decoders. An attacker was able to take advantage of a historical quirk of the ZIP format that meant that ZIP holds two lists of files: the signature checker used one list, while the installer used the other list. By pointing to real files from one list but to malicious files in the other list, an attacker was able to bypass this signature check and cause a user to install malicious code.

18.3 Concurrency Bugs

When systems have code running in parallel, things become much more difficult to reason about, and as a result, many bugs can occur. Consider the following code running on a bank server:

```
def xfer(src, dst, amt):
        s = bal[src]
        d = bal[dst]
        if s < amt:
                raise InsufficientBalanceError
        balances[dst] = d + amt
        balances[src] = s - amt
```

By executing two of these `xfer` requests in parallel, an attacker can cause unexpected behavior. For example, if an attacker tries to send money from a single source to two destinations at once, the check `s <`

`amt` may pass in both executions, and both destinations will then be updated to have money. However, the source will only be deducted once, since `s` is stored before any money is transferred.

The fix is to make sure that there is some kind of locking or concurrency-control strategy in place. The important high-level bit is to be thinking carefully about how concurrent execution can affect your software, in cases when multiple threads of execution can access the same data at the same time.

18.3.1 File system races / Time-of-check-time-of-use (TOCTOU) bugs

When deadline with files, symbolic links can cause all sorts of chaos. A piece of code might want to check that it is deadline with a regular file—rather than a symbolic link—before opening it. A bad way to implement would be like this:

```
// WRONG
if(lstat(path, &st) < 0) error();
if(!S_ISREG(st.st_node)) error();
// ***
int f = open(path, O_RDWR);
...
```

The problem is that if an attacker can replace the file when the code is at line ***, the attacker could swap out the regular file with a symbolic link that points to somewhere else.

The way to defend against this bug is to change the interface. Newer versions of the POSIX file-system APIs enable checking for this type of property in a way that defats race conditions.

The `openat()` API call gives a way to open a file while ensuring that the file is of a particular type.

18.4 Resource Usage

Other bugs allow attackers to consume many resources on a system, denying service to honest users.

For example, hash tables typically use a hash function designed to be very fast when deciding which bin to place an input in. These hash functions are typically not collision-resistant in the cryptographic sense. Hash tables work well when inputs get distributed evenly across all of the bins. However, if multiple inputs get mapped to the same hash value, the performance of a hash table deviates from the constant-time idea we have of a hash table—hash tables typically revert to using a linked list of all the values for a given hash value.

If an attacker is able to predict which bin their input will end up in, they can maliciously craft inputs that create a huge linked list, resulting in very poor performance for the hash table.

The way to defend against this type of attack is to use a *keyed* hash function—essentially a pseudorandom function—where the implementation does not reveal the secret key to the attacker (over the network, etc.).

18.5 Dealing with Software Bugs

As we have seen, there are many types of bugs so there are many ways to defend against them. Here are some general rules:

Clear Specification. One way to avoid design-level bugs is to have a clear and complete specification about what your program is supposed to do. In particular, for things such as encoders and decoders, a precise specification can make it easier to check that you have handled all of the important corner cases.

Design. A simpler design is easier to understand, and thus bugs are easier to find.

Limit Bug Impact. Some techniques that we will discuss, such as privilege separation, allow us to protect security even when software bugs arise.

Find and Prevent Bugs at Development Time. Techniques such as fuzzing can find bugs before they hit production systems.

Catch an Mitigate Bugs at Runtime

Deploy Bug Fixes Quickly A big advance in browser security came from mechanisms for pushing out software updates quickly.

19
Privilege Separation

The last chapter left off on a somewhat unsatisfying note: our software is bound to have bugs and those bugs are likely to be exploitable. Today, we will try to limit the impact that these bugs can have by *planning for compromise* and aiming to limit the damage that each component can cause if an attacker compromises it.

The philosophy that we will employ is called the *Principle of Least Privilege*. The idea is to give each component in the system the least privileges needed to do its job. By limiting the access that each component of our software has to do sensitive actions, we can prevent the compromise of our whole system when a single piece is attacked.

In real systems, it is often costly to implement least privilege in the strictest sense possible. You should think of the principle of least privileges as a difficult-to-achieve design goal, rather than a rule that every component of every system must satisfy.

To summarize our strategy:

Principle of Least Privilege: Each component should have the smallest set of privileges necessary to do its job.

As a concrete example, a web server might have some networking code, it might talk to a database server, and it might use some cryptographic keys. One way to architect the system with least privilege would be to split the server into multiple distinct processes—one that receives network requests, one that interfaces with the database, and one that uses the cryptographic keys. These components then would speak to each other using narrow, restrictive APIs.

In some sense, a system that perfectly implements the principle of least privilege gives one with the "best security we could ask for," in the face of component compromise. However, most real systems do not completely implement the principle of least privilege for a number of reasons:

An orthogonal, but important, strategy is to reduce the number of privileges that a component needs to do its job.

Challenge: Splitting Boundaries. A large piece of software has many components that often have many points of interaction. To effectively implement least privilege, we need to find boundaries at which we can separate our software. A few common ways to partition a system or application are:

If we don't split up a large piece of software into components, the software just consists of one über-component. An attacker who breaks into this one component can hijack the entire system.

- **Isolating by user:** In an operating system, different users of the systems have different privileges. This way, if an attacker compromises one user's account it does not compromise the entire system.
- **Splitting by feature:** In large business applications, different features (e.g., search, user management, mail) run as isolated components. This way, a bug in one component does not necessarily affect the entire system.
- **Splitting off buggy code:** The Firefox web browser uses special sandboxes to compartmentalizes bug-prone video codec code (Section 19.3). This way, if a malicious website is able to exploit the codec code, it is not so easy for the attacker to compromise the system.
- **Separating exposed versus internal code:** Google's front-end HTTP servers are isolated from the servers that run core application logic. This way, an attacker on the network is one step removed from the servers with access to Google's internal resources (databases, etc.).
- **Isolating sensitive data or keys:** Certificate authorities (CAs) use hardware security modules to isolate the cryptographic keys from all other code in the system. Laptops and servers sometimes have separate cryptographic co-processors for this purpose as well.

Challenge: Interface/API. If we chop our monolithic app into different domains, we also need to clearly define an API that the different domains can use to interact. This API must allow us to preserve the functionality of the system while making it difficult for an attacker that compromises one component from compromising an adjacent one.

Some strategies for interfacing between different components of a system might be

- remote procedure calls (RPCs) over a network,
- message queues,
- shared memory,
- shared database, or
- shared files or directories.

In designing interfaces between isolated components, we tend to worry about:

- **Functionality:** Does the isolation plan allow the application to work as it should?
- **Security:** Does the isolation strategy prevent the compromise of one component from affecting others?
- **Performance:** How much performance overhead does the isolation mechanism add?
- **Complexity:** How much code does the isolation mechanism require? If the compartmentalization strategy requires a large amount of code, this code might introduce new vulnerabilities.

19.1 Example: Logging

Most systems use logging to keep track of the actions that an application has performed. That way, if an attacker compromises the application, system administrators can look at the log to see what happened and how to mitigate it. For a logging system to be useful, it must be difficult for an attacker who compromises the application to erase the log.

It is very natural to separate out the log into a separate component: the app's functionality is almost entirely separate from the log, and the app's interface to the log can be very simple. The app should be allowed to append to the log and read from the log—the app should *not* be allowed to remove entries from the log. This way, compromise of the application or log server will not be enough to erase the log.

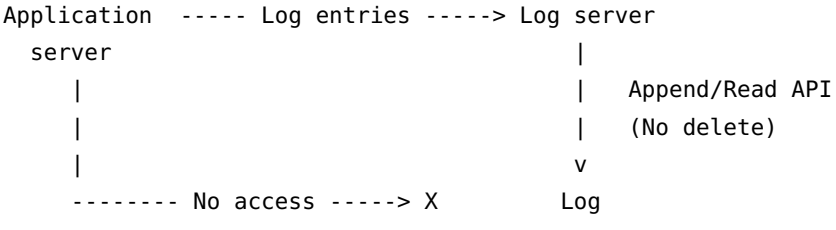


Figure 19.1: One way to architect a logging system. An attacker that compromises the application and/or log server may not be able to erase the logs.

19.2 Example: Cryptography Keys

Many applications use cryptographic keys for authentication: certificate authorities need to generate signed certificates for their clients, cryptocurrency wallets need to sign transactions, and WebAuthn requires an authenticator to sign a challenge from the relying party. In all of these applications, we worry a lot about an attacker stealing the application's secret signing key. Towards the goal of protecting cryptographic secrets, we often isolate the code that uses cryptographic

keys into a separate software (or even hardware) component. In particular, we might create a cryptography component whose only API is `sign(msg)` and `get_public_key()`. In this way, even if an attacker compromises the application, it cannot easily extract the secret key.

This design does not completely protect us against the compromise of the application. In particular, an attacker who compromises our app can still call `sign()` as much as they like to sign any message they like. At the same time, having the key isolated to this crypto component allows us to add checks inside the crypto component to, for example, reject messages of the wrong type. If our service is a certificate authority, we could set up our crypto component to verify that each message is an actual signature before signing it.

Although in many cases the things we are worried about an attacker asking for signatures of will pass this type check: for example, for a CA, this would still allow an attacker to generate a certificate for their malicious website.

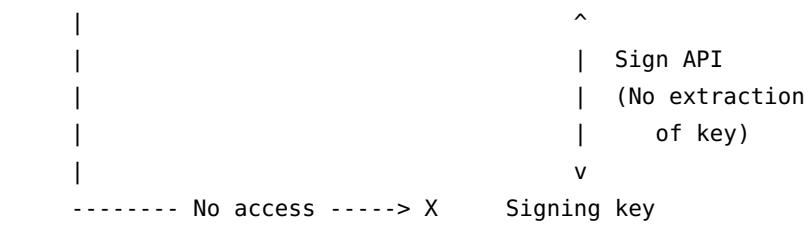


Figure 19.2: A common architecture for isolating cryptographic keys from an application.

This division also allows our crypto code to do some meaningful logging: imagine that the crypto module saves every signature it creates to a log. Compromise of the main app would allow an attacker to generate arbitrary signatures, but administrators could then see every signature that was generated during the compromise. This would be very helpful in recovery.

19.3 Example: Media Codecs in Web Browsers

Media codecs (e.g., for JPEG decoding) are notoriously complex and bug-prone: codec libraries have been at the source of many web-browser exploits. If we were able to isolate these codecs, we could make it less likely that a codec bug allows an attacker to access data other than the media file being decoded.

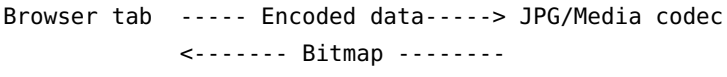


Figure 19.3: The Firefox browser isolates media codecs in a separate sandbox.

Isolating codecs may not always be as trivial as Fig. 19.3 makes it seem. In many cases, codecs require a sophisticated interface to the browser tab: the codec library may progressively decode videos as data arrives, for example. Even image decoders progressively decode

images, improving resolution as more data arrives. Supporting this functionality without adding vulnerabilities or excessive latency requires careful API design.

Firefox isolates these risky codecs using the language-level isolation that the WebAssembly language provides.

19.4 Example: Server for Network Time Protocol (NTP)

Operating systems use the network-time protocol to fetch the current time from time servers on the Internet, and to update the current time to match. Setting the time requires root privilege on Unix-like systems, but NTP also requires network accesses; the networking code can be complicated and bug-prone. To prevent an attacker who finds a bug in these network protocols from gaining root privilege on our machine, many systems separate the two into a process that handles talking to the NTP server on the network and a privileged process that accepts the time from this other process and sets it. This way, an attacker who find a bug in the network code can only set the time—they cannot perform arbitrary actions as root. In addition, the time service may impose some policy on time changes (e.g,. that time can never go backwards).

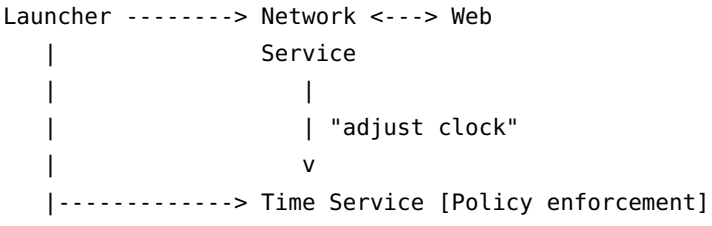


Figure 19.4: Modern operating systems implement a network-time protocol (NTP) client as separate processes. This way, an attacker who compromises the client over the network cannot arbitrarily corrupt the system time.

19.5 Example: OpenSSH Server

A secure shell (SSH) server has access to many sensitive resources: network port 22, a host secret key, the system's password file, and all users' data. When it starts, the SSH server runs a "monitor" process that listens for connections on port 22. When the monitor receives a connection, the first thing it does is to spawn a new per-connection worker process that communicates over the network. The worker client has no access to the host key or password database—the worker process can only ask the monitor for help in authenticating. The monitor-worker API supports a few operations:

- a *signing* operation, that instructs the monitor to sign a protocol transcript,

- a *password-authentication* operation, that instructs the monitor to check a password (this can be rate limited to prevent password guessing), and
- a *start-session* operation, that instructs the monitor to create a new process with a shell for the user.

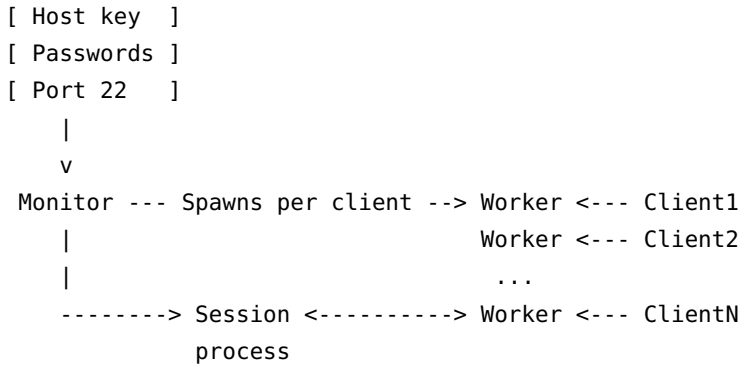


Figure 19.5: The OpenSSH server is split into multiple process to mitigate the compromise of the network-facing code.

While this architecture adds quite a bit of complexity to the OpenSSH server, it has paid off in terms of mitigating the impact of vulnerabilities in client-facing code.

19.6 Example: Web applications

Companies will often implement Web applications (e.g., a photo-sharing website) as a number of separate services, running on separate physical machines. Client connections come into a front-end server that terminates the TLS connection and proxies client data to a front-end application server. The front-end server then routes requests to one or more application services, each of which may have access to different databases. For example, the login service may have access to the password database, while the profile service may not.

19.7 Example: Web client

When you open a PDF attachment in Gmail , you might worry that the PDF could exploit some bug in your browser that could steal your sensitive Google data. To make this kind of attack more difficult, Gmail serves attachments and other suspect files from a separate domain ("origin')): `googleusercontent.com`. Code loaded from `googleusercontent.com` cannot access cookies or data for `google.com`. In this way, even if an attacker can somehow run JavaScript in your browser, it cannot easily steal your Google cookies.

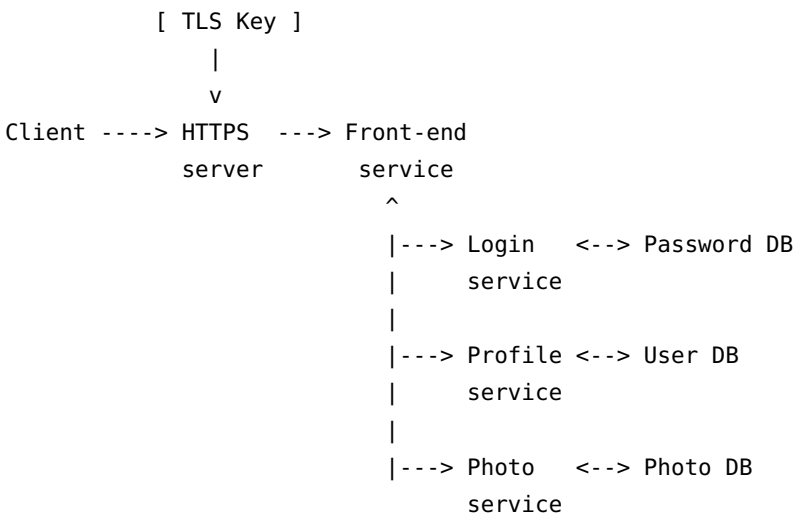


Figure 19.6: Large Web services tend to isolate different application features into different services, often on different physical machines. The system gates access to these services via minimal front-end client-facing servers.

19.8 Example: Web browser

Web browsers today are extraordinarily complicated pieces of software. The sensitive data that a browser is trying to protect are things, such as user cookies, cached data, browser history, and other user data. The browser may spawn new processes to handle rendering for each site from each distinct domain/origin. In this way, if an attacker from one origin can exploit a bug in the JavaScript engine, the attacker may still not be able to compromise sensitive user data from other domains/origins. GPU code, which is extremely complicated and bug-prone, may run in yet another process. Today, compromising a browser entirely often requires finding and exploiting a collection of bugs in multiple components.

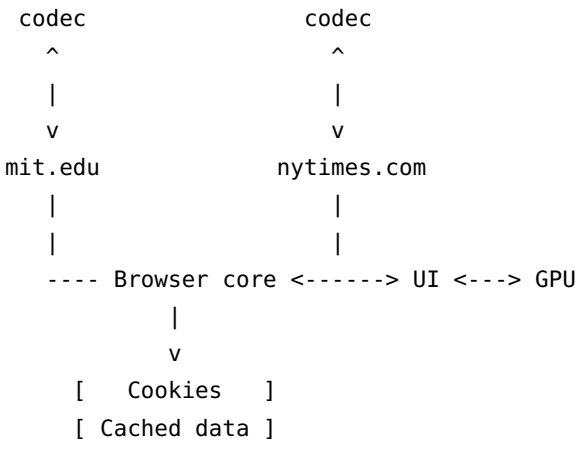


Figure 19.7: Web browsers may isolate the execution of each origin's code in a separate process. They further isolate complicated and bug-prone codecs and GPU code in separate processes.

19.9 Example: Payment Systems

Processing credit-card transactions in web applications is risky: if a vendor suffers a compromise, the credit-card network may fine them or kick them off the network. To avoid ever having to handle credit-card data, most websites use an external payment-processing service that handles credit-card information. When the user makes a purchase, the vendor redirects the user to the payment-processing service, who collects the user's credit-card data. After payment, the payment-processing service redirects the user back to the vendor's website.

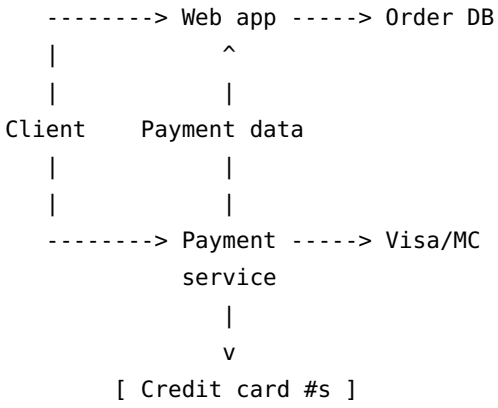


Figure 19.8: Online vendors often use a separate payment processor that handles the user's credit-card data.

20
Bug Finding

We are going to continue our treatment of security and bugs with a discussion of how to find bugs. As we already have seen, bugs are a big deal in terms of security problems. We have discussed how to architect a system using privilege separation so that bugs do not matter so much. But even with a very good privilege-separated design, we still want to make sure that our system is as bug-free as possible.

The topic of this chapter is then: *How do we find bugs?*
Step 1: Define what is a bug. Before we even begin talking about how to find bugs, we need to answer the question: *What is a bug?* There are a number of application-independent ways to determine when we have hit a bug:

- the program crashes,
- the program makes an out-of-bounds memory access, or
- the program jumps to an unknown or undefined point in the program.

More difficult types of bug to detect are ones that we can only detect with knowledge of what the application is supposed to do:

- the program's output is incorrect, or
- the program allows an attacker to access data it should not be able to access.

Step 2: Find bugs. To find a bug, we just need to identify a possible execution of the program that leads to one of the buggy outcomes we defined in Step 1. The reason this is difficult is:

- there are typically exponentially many possible inputs to the program and it may be difficult to find one the triggers the bug,
- concurrent systems exhibit non-deterministic behavior, and

- inputs from the environment (time, network state, etc.) can change the behavior of the program.

20.1 Bug finding: A concrete example

To frame our discussion of bugs, we will consider one specific example of a C program that parses a binary packet that has the format depicted in Fig. 20.1. The packet starts with a header byte and a length byte and then has a list of IDs.

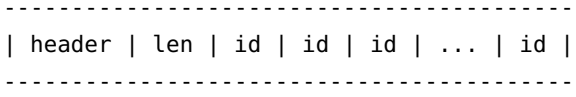


Figure 20.1: An example packet format.

```
1  // Parse header
2  char in[64];
3  int hdr = in[0];
4  if (hdr != s) returnl;
5  int n = in[1];
6  if (n > 64) return;
7
8  // Read ID fields
9  int ctr[32];
10 char *next = &in[2];
11 for (int i=0; i < n; i++) {
12   ctr[*next]++;
13   next++;
14 }
```

Figure 20.2: An example of some buggy parsing code.

20.2 Manual testing

When we manually test code, we consider a *specific execution* of the program and predefine an *expected result*. For the example of Fig. 20.2, we might tests to check that the following two inputs give the following behavior:

```
in = {6};         // Should have no effect.
in = {5, 1, 2};   // Should cause ctr[2] == 1
```

Benefits. The main advantage of manual testing is that tests can be targeted and can exercise application-specific logic. If you have a precise correctness condition that your program should ensure, it is often easiest to test it with manual testing.

An additional advantage is that manual tests are useful in *regression testing*: If you find a bug in your program today, you can write a manual test that tests that the buggy condition does not occur. As you update your program later on, this regression test can determine whether the same bug occurs again.

Downsides. The downsides of manual tests are that they are expensive to write, the test cases can themselves be buggy (especially when the program is complicated), it requires a lot of program-specific understanding, and it is difficult to write enough tests to cover a large fraction of the program's behavior.

20.3 Fuzzing

Fuzzing is the process of running a program on a very large number of *randomly generated inputs*. As soon as a random input causes the program to crash, we know that we have detected a bug.

When using a fuzzer to test a piece of code, we will typically ask the compiler to instrument the code with extra instructions to test whether the code behaved improperly. In C, for example, we will instruct the compiler to check for out-of-bounds memory accesses.

With the GCC compiler, you can compile your code with the `-fsanitize=address` flag to insert extra checks for memory-access errors.

To test for application bugs with a fuzzer, we can instrument our code with assertions that will crash the program if the program ever violates certain programmer-specified invariants.

Fuzzers may have to test a large number of inputs before finding one that triggers a bug. In the code of Fig. 20.2, if a well-formed packet has `n=64`, the program will write off the end of the `in` array. (The check in Line 6 should test whether `n >= 64` instead of `n > 64`.) For a fuzzer to hit this bug, it will need to choose a random input value of the form:

```
in = {5, 64, ... 64 arbitrary values ...};
```

The probability that a uniform randomly bitstring of the appropriate length hits this bug is $2^{-8} \cdot 2^{-8} = 2^{-16}$, which is quite small.

Coverage-guided fuzzing. Real fuzzers do not just feed uniform random bitstrings to programs that they are trying to fuzz. Instead, real fuzzers try to pick inputs in a way that maximizes the fuzzer's *code coverage*: the fraction of the lines of the program's code that the fuzzed programs have executed.

Most random bitstrings will probably not cause the program to exercise a large fraction of the program's code, since the program will reject them early most of the time.

To implement this strategy, the fuzzer maintains a corpus of bitstrings. Each time the fuzzer runs the program, it picks an input from the corpus and randomly mutates it in some way (e.g., by changing or adding a byte). If running the new string on the program causes the program to execute some new lines of code (i.e., the coverage increases), the fuzzer adds the newly mutated string to the corpus.

While this strategy does not have a robust theory to support it, coverage-guided fuzzing works shockingly well in practice. Many major software projects use fuzzing extensively to find bugs.

A coverage-guided fuzzer run on Fig. 20.2 might find the following input that causes the program to crash with an out-of-bounds write:

```
in = {5, 1, 100};
```

Benefits. A major benefit of fuzzers is that they are almost completely automated—they require very little input from the programmer. Since programmer time is more expensive than machine time, finding bugs using fuzzers is often much cheaper than finding bugs via manual test cases. Since fuzzers execute the program on billions of inputs, they will often find tricky bugs that a human might never find.

Drawbacks. A drawback of fuzzers is that they cannot find application-specific bugs: they are essentially limited to only finding violated assertions in a program. So while fuzzers are a useful tool for finding bugs, they are typically only useful in conjunction with manual tests.

Generalizations of fuzzing. The first fuzzers used random bitstrings as their initial pool of inputs. More recent fuzzers have application-specific logic for handling HTML, JSON, or other file formats—these fuzzers are better at catching higher-level logic errors. Some languages, such as Go, have support for fuzzing in their test infrastructure.

20.4 Symbolic execution

A weakness of fuzzing is that a fuzzer may not be able to trigger bugs that hide behind `if` conditions that are very very rarely true. Symbolic execution is a testing strategy that can find bugs in these difficult-to-fuzz programs.

The idea of symbolic execution is that we will run the program. But instead of running the program on actual concrete input values,

For example, if a parser first checks a CRC32 checksum on a packet, a fuzzer will almost never find an input that causes the program to run past the checksum check. Another example of difficult-to-fuzz code might be some HTML parsing code that checks that every < symbol is followed by an > symbol.

we will run the program on *symbolic variables* that represent arbitrary values. For example, we might want to use symbolic execution to run the following simple snippet of code:

```
c = a + b;
e = d + c;
f = a * d;
```

When executing this code, the state of memory might look like this, where we replace the value of the variable `d` with a variable X:

```
-------------------------------------------------
...    |  5  |  7  |  12  |  X  |     |     |  ...
-------------------------------------------------
          a     b      c     d     e     f
```

The major headache when using symbolic execution is *control flow*. For example, we might have code that looks like this:

```
if (e == f) {
  BUG();
} else {
  // Something else
}
```

If running with the symbolic variable `d` $= X$, then we will have the condition: `(d+c == a*d)`, which simplifies to: `(X+12 == 5*X)`.

To handle this sort of case, we can use a program called a *SAT Solver* to search for inputs that cause the condition to be true. For example, a SAT solver run on the branch condition `(X+12 == 5*X)` will likely find the value `X==3` that causes the condition to be true. Then the symbolic-execution engine can continue executing the program down the `true` branch of the program with the constraint `X==3` on the symbolic variable X. In parallel, the engine can search for values of `X` that cause the program to go down the `false` branch of the program. A SAT solver might find `X==908234` as one input that causes the program to traverse the `false` branch.

An important thing to know is that there is *no guarantee* that a SAT solver will actually be able to find an input that causes the program to execute one branch or another. The reason is that we know of no efficient algorithm for finding an assignment of the variables for an arbitrary condition that causes the condition to be true. (This problem is NP complete.) At the same time, for finding branch conditions in "reasonable" programs, SAT solves work surprisingly well.

Running a symbolic-execution engine on the code of Fig. 20.2 might produce the following output:

```
ERROR: buggy.c:10 memory error
ERROR: buggy.c:12 memory error
```

The symbolic-execution engine may consider the following tree of program executions, branching on each condition:

```
// Input data
in = {i0, i1, i2, i3, i4, ...};
```

```
[i0 == 5]
  /
  | FALSE -> return
  | TRUE  -> [i1 > 64]
  \           /
              | TRUE  -> return
              | FALSE -> [0 < i1]
              \           /
                          | FALSE -> return
                          | TRUE  -> [i2 < 0 || i2 >= 32]
                          \             /
                                        | TRUE  -> BUG!
                                        | FALSE -> [1 < i1]
                                        \           /
                                                    | ...continue ...
                                                    | ...execution...
                                                    \
```

Benefits. Symbolic execution can find tricky bugs involving complicated branch conditions that fuzzers and manual tests may not find. In addition, symbolic execution may be able to find bugs that do not crash the program. In addition, a symbolic-execution engine can in principle consider *all possible inputs* to a program and can give a guarantee that the program has no bugs of a certain type (e.g., out-of-bounds read).

Drawbacks. Symbolic-execution engines can be very slow to run and often work poorly on very large pieces of code. As the length of an execution grows, the symbolic-execution engine accumulates more and more symbolic variables (representing values in memory) and the number of constraints on each symbolic variables grows as well. When there are many variables and many constraints, the SAT solver may not be able to determine—in a reasonable amount of time—whether there is or is not a satisfying assignment to the variables. For these reasons, symbolic execution can work well for small snippets of code; in large programs (such as a web browser), symbolic execution may not be able to progress very deep into the program.

To address these drawbacks in symbolic executions, one approach is to write a *scheduler* that guides the search that the symbolic execution makes through the program state. A second approach is to define *loop invariants* or *function invariants* that aim simplify the job that the SAT-solver must perform by giving it more information about the program's expected behavior.

21
Runtime Defenses

We have considered a number of ways to improve software security. First, we explored *privilege separation* as a way to architect a system so that bugs do not lead to catastrophic security failures. Next, we looked at *bug finding*—techniques to find and eliminate bugs in source code before we use the code in production. Now, we will discuss *runtime defenses*: how to detect buggy behavior as it occurs in a piece of production software so that we can halt the program when a bug occurs.

There are a number of reasons why it is difficult to build runtime defenses. We must:

- determine the classes of bugs that we want to find,
- identify which components of the system that we want to monitor,
- figure out how to avoid *false positives* (erroneously halting a program when there is no bug), and
- try to implement these runtime defenses with minimal overhead, since we will apply these defenses to production code.

21.1 Defenses against buffer overflows

We have discussed the pervasive buffer overflow attack, which takes advantage of a missing bounds check to overwrite memory beyond the bounds of an array, often modifying the current function's return address to cause the attacked system to run attacker-specified code which is placed in the buffer itself.

An example of C code that is vulnerable to a buffer-overflow attack is this:

```
1 void f() {
2   char buf[128];
3   gets(buf);
4 }
```

The call-stack layout for this program will look like this:

```
|                 |
|-----------------|
| Return address  |
|-----------------|
| buf[127]        |
| ....            |
| buf[1]          |
| buf[0]          |
|-----------------|
|                 |
|                 |
|     ....        |
|-----------------|
```

The `gets` function in C will read from `stdin` until it finds a NULL (`\0`) character in the input string. If the string has length > 127, the `gets` function could copy adversarially generated input byte onto the stack, overwriting the return address of the function `f`.

There are three steps involved in executing a buffer-overflow attack:

1. write past the end of a buffer,
2. cause the victim process to jump to an adversary-controlled address, and
3. cause the victim process to run adversarial code.

Runtime defenses against buffer overflows can try to disrupt each of these three steps.

21.1.1 Non-Executable Stack

A first defense against buffer-overflow attacks is to prevent the CPU from executing code on the C call stack. In traditional buffer-overflow attacks, an attacker will somehow place some adversarial code on the stack (e.g., in the buffer `buf` in the example above) and then cause the victim process to jump to that code on the stack.

However, C usually puts normal program code in a separate region of memory—not on the stack. Modern CPUs allow us to add permissions to different memory regions: the OS can mark each region as read (R), write (W), and/or execute (X). To make buffer-overflow attacks more difficult, we can prevent the CPU from running code from the stack at all by marking the stack as RW only.

This is sometimes called an NX defense—for "no execute."

Marking the stack as non-executable seems to eliminate a major piece of an effective buffer overflow attack: the attacker can no longer supply code of their choice and point to it with the return address. However, modern attackers have worked around this with something called *return-oriented programming*: with the ability to supply their own code removed, attackers must find code that already exists to do what they like. This may be full existing functions, but more likely attackers will set the return address to point into the middle of some function and execute just a fragment that does something useful. It turns out that with more work, it is possible to perform many attacks using only code that already exists in a victim process.

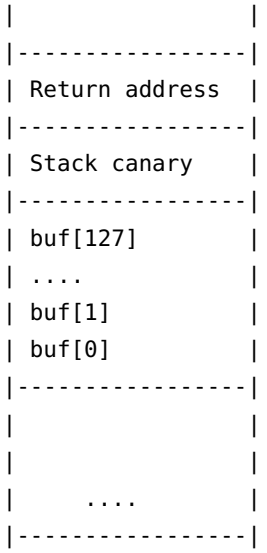
21.1.2 Stack Canary

To try to remove the adversary's ability to overwrite the return address in the presence of a buffer overflow, another defense is to insert a *stack canary* in every stack frame between the function variables and the return address. A stack canary is typically a secret random value, chosen when the program starts running.

A buffer-overflow attack does not directly allow the attacker to read arbitrary memory, so the attacker has no direct way to read the canary before overflowing the buffer.

At the start of each function, the compiler inserts instruction that write this canary to some value. At the end of the function before returning, the compiler also adds some code that checks that the canary value has not changed. If the canary has changed, the attacker must have overflowed a buffer and the program should exit to avoid running unknown code.

A clever non-random canary includes a collection of the string-terminating characters, such as `0` `n` `r`. Many C functions that read strings from input, such as `gets` will stop reading once they reach one of these values, so it could be difficult for an attacker to feed in an overflowing string that includes these characters.



This is effective because in a buffer-overflow scenario, the attacker needs to write memory sequentially until the address they care about writing is reached: if the canary is between the function variables

and the return address, the attacker must overwrite the canary to modify the return address. However, this is not a perfect defense: if the attacker writes the same value to the canary as was already there, it will go undetected. Therefore, the canary value must be hard for the attacker to guess.

This defense is still not perfect—for example, it does not prevent an attacker from overwriting function pointers. However, it does make a successful attack significantly harder.

There are a few ways to subvert canaries:

- An attacker can corrupt *data* on the stack, even if it does not corrupt the return address. This could be very bad for the program's behavior.
- An attacker might find a way to read the canary from memory (e.g., if there is some other bug in the program) and then execute the traditional buffer-overflow attack to overwrite the return pointer.
- In a forking web server, the child process may have the same canary value as the parent process. An attacker can potentially exploit this to learn the canary (See Andrea Bittau et al. "Hacking blind". In: *IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 227–242).

21.1.3 Address Space Layout Randomization (ASLR)

Another approach to defend against buffer overflow-style attacks is to make it more difficult for the adversary to guess a useful address to jump to. To do this, many modern systems randomize the locations of code, stack, and heap memory regions when a process starts. With this defense in place, an attacker needs to learn the location of the code memory region in order to mount a return-oriented programming attack (Section 21.1.1).

Implementing ASLR requires compiler support. Most modern compilers do.

A weakness of ASLR schemes is that they typically shift the location of an entire region of memory: the entire heap, stack, and code sections move around in memory, but the layout within each section is typically fixed at compile time. Thus if the adversary can learn the location in memory of the code for a single function, it can mount an effective return-oriented programming attack.

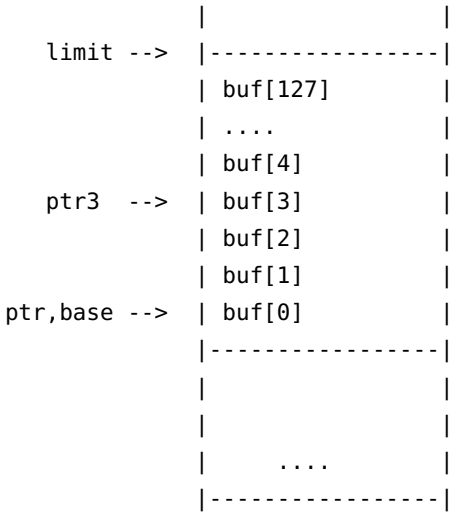
21.1.4 Bounds Checking with Fat Pointers

All of the defenses so far have attempted only to minimize the damage of a buffer overflow after an attacker has exploited it. However, we could prevent a more comprehensive suite of attacks if we could

make sure that our code never reads or writes a pointer that is outside the bounds of a given buffer. Memory-safe languages such as Go, Rust, and Python have this bounds checking built in. For older languages, such as C, we can try to retrofit the C compiler to achieve this bounds checking. As we now discuss, implementing bounds checking in C is challenging.

One way to implement bounds checking in C is a technique called *fat pointers*. When using fat pointers, the compiler changes the representation of a pointer to include not only an address, but also the *base* and the *limit* of the buffer the pointer points to. This base and limit are initialized on an allocation, and the compiler inserts bounds checks on each pointer dereference to guarantee that the dereferenced value is within the array bounds that the base and limit specify. Pointer arithmetic preserves the base and limit but modifies the pointer itself as before, allowing the pointer to possibly go out of bounds.

For example, if a programmer allocates an array of 128 bytes using `void* ptr = malloc(128)`, the compiler will associate values `base = ptr` and `limit = ptr+128` with the pointer `ptr`. If we then assign `ptr3 = ptr + 3`, then the new pointer `ptr3` will have the same base and limit as `ptr`:



When the program deferences `ptr3`, the compiler will insert checks to ensure that `base` $\leq$ `ptr3` $<$ `limit` and crash the program otherwise.

One limitation of fat pointers is that they only check overflowing an *allocated region of memory*—not overflows *within* a region of memory. For example, if we use `malloc` to allocate a C `struct`, overflowing the `buf` member of the struct could allow the attacker to overwrite the values of other elements of the struct:

```
struct {
```

```
  int x;
  char buf[16];
  void (*f)();
}
```

If there are function pointers in the `struct`, the result of this within-region overflow could be as bad as overflowing the return address.

Unlike a nonexecutable stack, stack canaries, and ASLR, fat pointers are not widely used. This is largely because the modified "fat" pointers can break the functionality of existing C code. In particular, a fat pointer on a 64-bit architecture will typically take more than 64 bits to represent. If the programmer cast a pointer to an `int` and back again, the behavior of the program could change when using fat pointers versus when using unmodified 64-bit pointers. In addition, a C program may implicitly require that a pointer takes exactly 64 bits to represent (e.g., as a field within a `struct`); a compiler using fat pointers will break such programs.

21.1.5 Control-Flow Integrity (CFI)

The goal of techniques for *control-flow integrity* is to limit the set of addresses that a program will jump to when returning. In this way, a victim process may be able to detect when it is about to jump to a return address that an adversary modified with a buffer overflow. Implementing CFI requires compiler support—many modern compilers support some form of CFI.

To implement CFI, we need to add several checks on different kinds of jumps.

- For direct jumps (e.g,. "`call gets`", we know at compile time that these jumps are valid since there is nothing that the attacker can control. There is no need to implement any CFI checks on these jumps.
- For indirect jumps that use some variable in the jump target (e.g., function pointers, function returns), the compiler inserts checks to insure that the return address is a valid jump target. To do this, the compiler inserts into the program a data structure that maintains a set of all valid indirect-jump targets. Checking whether a jump point is valid takes a bit of extra computation—and thus imposes some runtime cost—it may tolerable in practice.

Often a compiler will implement this data structure using some sort of simple Bloom filter.

21.1.6 Unsolved program: Use-after-free bugs

The defenses in this section have made buffer overflows extremely challenging to exploit in modern code on modern systems. These

defenses do very little to prevent against *use-after-free* bugs, in which a program `free`s allocated memory and then inadvertently reads or writes the `free`d pointer.

21.2 Taint Tracking to Defend against Input-Sanitization Bugs

We have discussed SQL injection and cross-site scripting, which allow an attacker to run code by adding special characters, such as "`"`" or "`>`," into their input.

Unlike buffer-overflow bugs, input-sanitization bugs arise not because a bug at one particular point in the program—they are more due to a systematic failure to consider certain types of inputs. As a result, the defenses are more systematic as well.

21.2.1 Taint tracking in libraries

A common approach to check for sanitization failures at runtime is called *taint tracking*. In taint tracking, some infrastructure in the program (e.g., a SQL library) marks any data coming from user input as *tainted*. At functions that perform escaping, the infrastructure removes the taint label. The infrastructure marks sensitive functions, such as the HTML renderer, are marked as *sinks*. Any time the program runs sink code, the taint-tracking infrastructure checks that the data headed into the sink is not tainted. If the data is tainted, some part of the input must not have been sanitized since it came from the user, and the infrastructure will halt the program to avoid an exploit. See Fig. 21.1 for an example.

```
name = read_from_user()     # name is tainted
first,last = name.split()   # first,last are tained
# query is tainted
query = "SELECT␣*␣FROM␣USERS␣WHERE␣last_name␣=␣'" + last "'"
query_database(query)       # will cause an exception

qesc = escape(query)        # qesc is not tainted
query_database(qesc)        # will succeed
```

Figure 21.1: A hypothetical Python example of how taint-tracking might prevent SQL injection

Many browsers implement taint tracking to prevent cross-site scripting using *Trusted Types*: for JavaScript calls that update the displayed HTML, such as `innerHTML = foo`, browsers may restrict the type of `foo` to ensure that the code explicity converts its type to something like `TrustedHTML`. This does not guarantee that the sanitization was done correctly, but does ensure that the programmer acknowledged the risk in their code.

21.2.2 *Taint tracking in language runtimes*

Some programming languages associate a *taint bit* with every string in the program. The application developer can set or clear the taint bit manually. In addition, the language runtime will automatically set the taint bit on strings that certain functions (e.g., those that read from user input) return. An application developer then can implement some taint-checking policy to systematically prevent against escaping bugs. For example, if there is one function that makes a SQL query to the database, the application could check the taint bit is cleared on any query string that a developer passes to this function.

21.2.3 *Taint tracking in operating systems*

Operating systems also use taint tracking, often to flag suspicious files—typically those that the user downloaded from the Internet. MacOS, for example, will set a taint bit on any executable that the user downloaded from the Internet. The OS will maintain this taint bit when the user copies or moves a tainted file. If the user ever tries to execute a tainted file, the OS will raise a warning to the user before executing it.

Part V

Privacy

22
Privacy and zero-knowledge proofs

So far, we have discussed several cryptography primitives, from hash functions to encryption schemes, and explored many applications of those primitives to systems security. These primitives have provided security, but in a very all-or-nothing sense: in order to provide broadly applicable security, our definitions required that a certain party (with the key) could either completely decrypt the message, learning the message contents, or cannot do anything with the message at all.

22.1 Zero-knowledge proofs of knowledge

Zero-knowledge proofs allow one party (called a prover) to convince another party (called a verifier) that the prover "knows" some secret, without revealing anything else about the secret. More formally, say that both parties hold a function f, and a value $y = f(x)$. The prover may want to convince the verifier that it "knows" an x such that $y = f(x)$. We call such a prover-verifier interaction a *proof system*. If the prover reveals no information about x, apart from the fact that $y = f(x)$, to the verifier, we say that the proof system is a *zero-knowledge proof of knowledge*.

The proof takes the form of an interaction between the prover and the verifier, in which the two parties exchange a sequence of messages. At the end of the interaction, the verifier either accepts the proof or rejects it. This is an *interactive proof*: the prover and verifier may exchange many messages. In contrast, in a classical (non-interactive) proof, the prover writes down a proof and sends it to the verifier in a single message.

Probably all of the proofs you have seen in your life up until this point, including the ones in your math textbooks, are classical proofs.

In our example, both parties hold a function f and a value y. The prover wants to convince the verifier that it knows a value x such that $y = f(x)$. To be a zero-knowledge proof of knowledge, the proof system must have three properties. We state this informally:

The literature usually refers to the secret value x as the *witness*.

- *Completeness.* If the prover "knows" x, it will always convince an

honest verifier to accept.

- *Soundness.* If the verifier accepts the proof, then prover must really "know" x. The soundness property is about protecting an honest verifier from a cheating prover: if the prover does not know x, then no matter how it cheats it will not be able to convince the verifier that it does.

- *Zero-knowledge.* The verifier "learns nothing" about x, apart from the fact that $y = f(x)$, as a result of interacting with the prover. The zero-knowledge property is about protecting the prover from a malicious verifier: if the prover is honest, then no matter how the verifier misbehaves, it should learn nothing about the prover's secret witness x.

TODO: HCG: Insert formal definitions here?

One important question here is: How do we define the notion of knowledge? That is, what does it mean for the prover to really "know" x? A naïve definition might say that x is stored somewhere in the prover's memory, but that might be too strong: a prover might "know" x without storing it literally in its memory, perhaps storing it in base64-encoded form. A more meaningful definition of knowledge, which is the one we will use, thinks of knowledge in terms of extractability. That is, we say that a prover knows x if there is an efficient algorithm that extracts x by interacting with the prover and observing the prover's internal state.

We also must pin down what it means for the verifier to learn nothing through an interaction with the prover. A good way to define this, it turns out, is to think of "learning nothing" in terms of the verifier being able to simulate its interaction with the prover without actually talking to the real prover. Specifically, if the verifier can produce a transcript of its hypothetical interaction with the prover—that is cryptographically indistinguishable from a real transcript—without actually communicating with the prover, we say the verifier has learned nothing about x.

There is a rich theory of zero-knowledge proof systems that we will not be able to cover here. One surprising fact is that there exist zero-knowledge proofs of knowledge for *any* choice of the public function f, provided that it is an efficiently computable function.

One immediate application of zero-knowledge proofs of knowledge is to authentication. The prover's witness x becomes the user's secret key, the verifier's y is the user's public key, and the function f is a public parameter of the authentication scheme. The authentication protocols we've seen so far, such as challenge-response protocols using MACs or signatures, do not quite satisfy the notions of soundness and zero-knowledge we use here. For example, in a MAC-based challenge-response protocol, the server learns the MAC of a server-chosen challenge value under the client's (prover's) secret key. Provided that the MAC is secure, the verifier could not produce

a simulated transcript that is identical to a real one without actually interacting with the real prover.

22.2 *Discrete-log problem and Schnorr signatures*

As an example of a zero-knowledge proof of knowledge, we will show how a prover can convince a verifier that it knows a discrete log, without revealing it.

22.3 *Reminder: The discrete-log problem*

We recall the discrete-log problem. The problem is parameterized by a group $\mathbb{G}$ of prime order q, generated by a value $g \in \mathbb{G}$. Then, given a group element $y = g^x \in \mathbb{G}$, for $x \xleftarrow{R} \mathbb{Z}_q$, the discrete-log problem is to find the value of $x \in \mathbb{Z}_q$. We have already seen the discrete-log problem in the context of Diffie-Hellman key exchange and elliptic-curve signatures.

In cryptographic settings, q is a big prime.

22.4 *Schnorr's protocol for proving knowledge of discrete logs*

In Schnorr's protocol, the prover and verifier both hold the generator $g \in \mathbb{G}$ and the group order q. In addition the prover holds a secret value $x \in \mathbb{Z}_q$, the verifier holds a public value $y \in \mathbb{G}$. The prover must convince the verifier that it knows the discrete log of y base g: that is, that it knows x such that $y = g^x \in \mathbb{G}$. The prover wants to prove this to the verifier this without revealing anything about else about its secret value x.

The protocol goes as follows:

- The prover picks a random $r \in \mathbb{Z}_q$, and sends $R = g^r \in \mathbb{G}$ to the verifier.
- The verifier picks a challenge bit c at random, either 0 or 1, and sends the challenge to the prover.
- The prover sends back a response $z \in \mathbb{Z}_q$, chosen as follows:
  - If $c = 0$, the prover sets $z \leftarrow r \in \mathbb{Z}_q$.
  - If $c = 1$, the prover sets $z \leftarrow r + x \in \mathbb{Z}_q$.

  Mathematically, $z = r + cx \in \mathbb{Z}_q$.
- The verifier checks that $g^z = Ry^c \in \mathbb{G}$. If equal, the verifier accepts, otherwise the verifier rejects.

In the true Schnorr protocol, the verifier samples $c \xleftarrow{R} \mathbb{Z}_q$, as we discuss in Section 22.6

Why is interaction necessary here? It turns out that this protocol critically depends on the prover not knowing the challenge bit c in advance. If the prover knew c in advance, the prover could pick $z \xleftarrow{R} \mathbb{Z}_q$ at random, and just compute R as $(g^z)(y^c)^{-1} \in \mathbb{G}$.

Reducing soundness error. In any given interaction of the protocol, the prover could falsely convince the verifier that it knows x, using the attack we just described. The prover only will succeed in this attack with probability $\frac{1}{2}$, which is exactly the probability that the prover guesses the verifier's challenge c in advance. By repeating this protocol λ times and accepting only if all λ iterations accept, the verifier can reduce the probability of accepting a cheating prover (called the "soundness error") to $2^{-\lambda}$.

A more efficient way to reduce the soundness error is to have the verifier sample the challenge value $c \xleftarrow{R} \mathbb{Z}_q$ as a random $\mathbb{Z}_q$ element instead of a random bit. With this transformation, the soundness error becomes roughly $1/q$, which is $\approx 2^{-256}$ for the cryptographic groups we use in practice. An important caveat is that if we increase the challenge space in this way, we cannot prove that the resulting protocol is zero knowledge. This modified protocol only satisfies a weaker flavor of zero knowledge called "honest-verifier zero knowledge." We won't discuss those details here.

22.5 Analysis of Schnorr's protocol

We will now argue that Schnorr's protocol is really a zero-knowledge proof of knowledge of discrete log. To do this, we must show that the protocol satisfies completeness, soundness, and zero knowledge, as we defined them in Section 22.1.

The completeness follows by construction. The more challenging steps are proving soundness and zero knowledge.

22.5.1 Soundness using an extractor

We first need to convince ourselves that the protocol is *sound*: that is, if the verifier accepts, the prover really knows x. We can prove soundness by showing that, if there is a (possibly adversarial) prover P^* that can convince our honest verifier to accept, there exists an efficient extractor that can use the cheating prover P^* to extract x with some high probability (say, $\gg 1/2$).

For simplicity, assume here that the prover P^* manages to convince the verifier to accept with probability 1—i.e., for all choices of the verifier's challenge bit c and all choices of the prover's randomness. Then, given a prover algorithm P^*, the extractor for Schnorr's

protocol works as follows:

- The extractor runs the prover P^* to get some initial protocol transcript with challenge $c = 0$. The transcript consists of $(R, c = 0, z)$.
- The extractor rewinds the state of the prover P^* to just after the point when it sent the first protocol message R to the verifier.
- The extractor runs P^* again, but this time feeds its the challenge $c = 1$. This produces a second transcript $(R, c = 1, z')$.
- The extractor outputs $z' - z$ as the discrete log x.

By our assumption—namely, that P^* can convince our honest verifier to accept with probability one—both runs of the prover P^* by the extractor must convince the verifier to accept. (Otherwise, the prover P^* would not convince the verifier in at least one of the two runs of the protocol.) That means that both $g^z = R \in \mathbb{G}$ and that $g^{z'} = RX \in \mathbb{G}$. Thus, $g^{z'-z} = y \in \mathbb{G}$, which is exactly the definition of what it means for something ($z' - z$ in particular) to be the discrete log of $y \in \mathbb{G}$.

Here we have glossed over the details of what happens if the prover convinces the verifier with some non-negligible probability that is nonetheless less than 1. A similar but more involved argument handles that case.

22.5.2 *Zero-knowledge using a simulator*

Our second task is to show that Schnorr's protocol satisfies zero knowledge. That is, that even a malicious verifier V^* that deviates from the protocol "learns nothing" about the prover's witness x as a result of interacting with the prover in the Schnorr protocol. To do this, given a malicious verifier algorithm V^*, we construct a *simulator* algorithm. The simulator's job is to produce a transcript of the prover-verifier interaction that is indistinguishable from a true prover-verifier interaction. Crucially, in the simulated interaction, the simulator *does not know* the witness x, while in the real prover-verifier interaction, the prover *does know* the witness x.

Given the verifier V^*, we construct this simulator as follows:

- Make a guess $c' \xleftarrow{R} \{0, 1\}$ of the verifier's challenge bit.
- Choose the third protocol message $z \xleftarrow{R} \mathbb{Z}_q$ at random.
- Set the first message to $R \leftarrow g^z y^{-c'} \in \mathbb{G}$.
- Run the verifier V^* on first message R. The verifier outputs a challenge c.

- If $c = c'$, output (R, c, z) as the simulated transcript. This happens with probability $1/2$.
- Otherwise, retry. This happens with probability $1/2$ as well.

Both the extractor and simulator constructions rely crucially on being able to rewind the prover or verifier and try again. In a real prover-verifier interaction, the verifier cannot rewind the prover and the prover cannot rewind the verifier. This is why the existence of an extractor does *not* mean that in a real protocol interaction the verifier can extract the witness x from the prover.

22.6 *Fiat-Shamir heuristic and Schnorr signatures*

One challenge with Schnorr's protocol as we have presented it, and zero-knowledge proof systems more generally, is that they are interactive. They require the prover and verifier to send messages back and forth. A clever trick, called the Fiat-Shamir heuristic, allows us to turn interactive zero-knowledge proofs into non-interactive proofs, provided that

- we use a cryptographic hash function that we model as a random oracle, and
- the verifier only sends the prover independent random values.

In Schnorr's protocol, the verifier's challenge is just a random bit, so it satisfies this second property.

The idea is that, whenever the protocol expects the verifier to send a challenge to the prover, we can instead derive the challenge just as a hash of the protocol transcript so far. The prover can then execute the zero-knowledge protocol against a synthetic verifier (the hash function), and produce a transcript. The prover can now send this transcript to the verifier, and if the verifier can confirm that indeed all of the challenges were correctly computed using a cryptographic hash function, such as SHA-256, it's sound to accept this transcript as a proof.

We can use this Fiat-Shamir heuristic to construct a (regular, non-interactive) signature scheme from an (interactive) zero-knowledge proof of knowledge for the discrete-log problem. The elliptic-curve signatures used in practice today effectively take the Schnorr protocol for proof of knowledge of discrete log using challenge space $\mathbb{Z}_q$, and apply the Fiat-Shamir heuristic to it. The resulting signature scheme is called the *Schnorr signature scheme*.

23
Differential Privacy

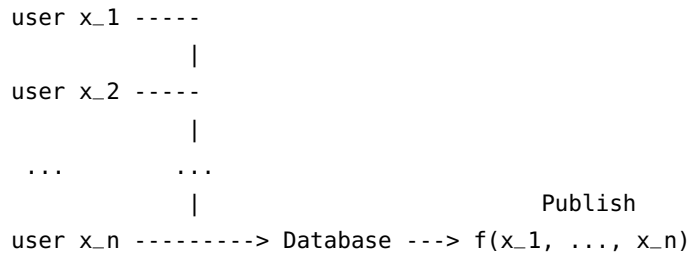
In many of the systems we have seen so far—encryption, iOS security, etc—we try to prevent an adversary from learning *any information* about our sensitive data. Today we will be talking about how to deal with situations in which we *want* to leak some information about sensitive user data, but we want to do it in a "privacy-preserving" way.

For example:

- the U.S. Census collects sensitive demographic information about U.S. citizens and then publishes aggregate statistics about it,
- Internet Service Providers may want to publish aggregate statistics about network traffic,
- Google may want to publish anonymized data about user search queries, or
- public-health officials may want to publish metrics about a population's health at large without revealing any individual user's health status.

In each of these settings, we many n users, where each user i has some data x_i. There is a central party that has collected $(x_1, \ldots, x_n)$ and wants to publish an aggregate statistic $f(x_1, \ldots, x_n)$ in a way that "protects the privacy" of each user's data.

The techniques we describe here also apply when there is only a single user and the user wants to publish some function about their private data.



Q1. What functions of private data can we publish without compromising user privacy? How do we even define "privacy" in this

context?

Q2. Once we have a definition of privacy, how do we achieve it in practice?

Differential privacy gives us *one possible* answer to the question Q1 here—it is a *definition* of privacy that has an number of appealing properties. For reasons we will discuss, it is not a perfect definition of privacy, but it is essentially the best one we have.

Once we buy the privacy definition that differential privacy gives us, there is a rich literature that explains mechanisms for achieving differential privacy (i.e., the answer to question Q2).

23.1 A bad idea: Attempt to anonymize the data

A common **bad** strategy for attempting to release data with privacy protections is to try to *anonymize* the data. For example, if a hospital wants to release medical-records data, they could publish the records with the names redacted. One surprise—that has bitten many companies and governments—is that it is shockingly easy to re-identify data in datasets that are supposedly anonymized.

Example: Re-identification by Linking (Sweeney 1997) For example, in Massachusetts, the Group Insurance Commission released a dataset that they believed to be anonymized. This dataset contained health data, including patient ethnicity, ZIP code, birth date, sex, date of visit, diagnosis, procedure, and medication given.

In 1997, Latanya Sweeney demonstrated that this anonymization strategy completely failed. In particular, she purchased (for $20!) the public voter-registration data from the government of Cambridge, MA. This second dataset included voter name, address, ZIP code, birth date, and gender. Crucially, this other database included each voter's zip code, birth date, and gender! Using the voter-registration dataset—which linked names to zip codes and birth dates—and the medical data set—which linked zip codes and birth dates to medical conditions—Sweeney was able to determine which people had which medical conditions. That is, Sweeney was able to completely de-anonymize the GIC dataset and reveal private medical information for everyone up to the governor of Massachusetts.

Netflix Competition In 2006, Netflix aimed to improve their movie recommendation system. To do this, they planned to have researchers compete to come up with the best movie-recommendation algorithms. For the purposes of running the competition, Netflix published a supposedly anonymized dataset that included a randomized user id,

movie id, rating, and date. Netflix released a portion of the dataset, and the research group that could produce a model that best predicted ratings for the unreleased set of movies would win a million dollars.

One group of researchers was able to link individual records in the Netflix database to records from the public IMDb database. This allowed them to de-anonymize the data from Netflix even though the database contained no identifying information whatsoever!

The bottom line. Anonymizing data sets simply does not work.

23.2 *A flawed idea: k-Anonymity*

The k-anonymity approach to privacy is to think of a dataset release as being sufficiently private if there are at least k users' data in the dataset that are identical. For example, if there are $k = 10$ people in a medical-records database with the same birth date, zip code, and medical condition, we might be okay with publishing the dataset with those three fields only when there are at least $k = 10$ users whose medical records are identical on those three fields.

The flawed intuition here is that even if there is still some leakage, maybe it is not so bad. For example, even if you can link the voter-registration database to the medical-records database, you might hope that it will be difficult to figure out exactly which user has exactly which medical condition.

There are a few issues with the k-anonymity approach to privacy:

- It is not robust to *side information*. For example, if an attacker knows a few people with a given medical condition, it can use a process of elimination to de-anonymize the dataset still.
- The anonymized dataset may still leak *some private information*. For example, say that every student in a class of 100 got the same grade. The teacher might be happy publishing the grades of all students (without names) since this release satisfies $k = 100$-anonymity. The problem is that the anonymized release reveals the *grade of every student in the class*—definitely not exactly what we would expect from a privacy-preserving release.

23.3 *A flawed idea: Publish only aggregate statistics*

Another approach we might consider is to publish only summary statistics with the aim that these summary statistics compress the data so much that it is impossible to learn anything meaningful

about an individual from them. However, we have to be careful: even a few statistics can reveal sensitive information.

For example, consider a company that released the average salary of its employees regularly. If the company releases this average before and after the resignation of one individual, anyone who knows that that individual resigned can learn his salary.

So, releasing only statistics does not cleanly protect privacy either.

23.4 A new definition: Differential privacy

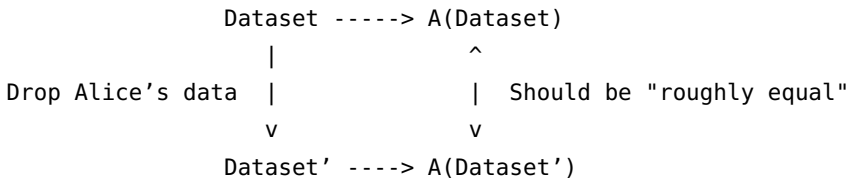
Differential privacy is a very mathematically clean definition of privacy that addresses many of the concerns of the flawed ideas above. It is not perfect—for reasons we will see—but many regard it as the best formal definition of privacy we have.

Say that $(x_1, \ldots, x_n)$ is a dataset of n users' data. Differential privacy, roughly, says that a it is safe to release an algorithm $A(x_1, \ldots, x_n)$—called a *mechanism* in the language of differential privacy—applied to the dataset if the value of the algorithm A's output is roughly the same if we release any user i's data x_i with some other data value x_i^*:

$$A(x_1, \ldots, x_n) \approx A(x_1, \ldots, x_{i-1}, x_i^*, x_{i+1}, \ldots, x_n).$$

In other words, dataset-release mechanism provides differential privacy if its output looks roughly the same, whether or not a particular user's data is included in the original dataset.

In picture form, the situation looks like this:



To formalize this, we will require that a pair of datasets that differ in only a single row—that is, one contains a row and the other does not—are *close*. The formal definition is as follows:

Notice that a mechanism A that outputs nothing trivially satisfies differential privacy. For a mechanism to be useful, it should add as little noise as possible to maintain a given level of differential privacy.

Definition 23.4.1 (ϵ-differential privacy)**.** An mechanism A provides ϵ-differentially private if, for all neighboring datasets x and x' that differ in only one row, for all subsets S of outputs of A:

$$\Pr[A(x) \in S] \leq e^{\epsilon} \cdot \Pr[A(x') \in S].$$

This is approximately saying that whether or not a given row is included in the dataset, the probability of the output being detectably different is less than ϵ.

We have that $e^{\epsilon} \approx 1 + \epsilon$ when ϵ is very small.

Notice that the definition of differential privacy is parameterized by a real number ϵ that specifies how close the two outputs of the mechanism A must be on neighboring datasets. The smaller ϵ is, the stronger the privacy guarantee is.

In the differential-privacy literature, this parameter ϵ is often called the "privacy budget."

One of the delicate points in practice is figuring out what actual value of ϵ a system should aim to achieve.

In order for a non-trivial mechanism to provide differential privacy, the algorithm must be randomized. Take our salary example from before: if we want to reveal an average salary of all employees, achieving differential privacy requires adding *noise* in order to make sure that the output of the release algorithm looks roughly the same whether or not a particular employee's salary is included in the computation.

Differential privacy is closed under post-processing. A really convenient property of the differential-privacy view of privacy is that the definition is *closed under post processing*. That is, if a mechanism A is ϵ-differentially private, then for *every* function B, the composed mechanism $B(A(\cdot))$ is also ϵ-differentially private.

Differential privacy is composes nicely. In many applications, we want to publish many functions of the same dataset. An very powerful property of the definition of differential privacy is that it can nicely handle this situation. In particular, say that a mechanism A_1 is a mechanism that satisfies ϵ_1-differential privacy. Say that a mechanism A_2 is a mechanism that satisfies ϵ_2-differential privacy. Then the mechanism $B(x) = (A_1(x) \| A_2(x))$ that publishes the output of both mechanisms is $(\epsilon_1 + \epsilon_2)$-differentially private.

There are more sophisticated composition theorems that give tighter bounds on the differential privacy parameter of the composed mechanism when the number of mechanisms you are composing is large.

The value of ϵ increases with each statistic we publish—which nicely captures our intuition that the more statistics you release about a dataset, the more information you are leaking about it.

23.5 *Achieving differential privacy: Adding noise*

A common approach to achieving differential is exactly this: adding random noise to the data in order to add uncertainty about the real data. The first instance of this was by Warner in 1965, who proposed using random noise to allow individuals to answer sensitive survey questions (i.e., revealing some private data) while also giving them some privacy protection. Warner's approach is called *randomized response*.

To give an example of randomized response: consider a professor that wants to learn the fraction of students who cheated on a test. Obviously no student wants to admit that they cheated on an exam. Using Warner's approach, the professor could ask students to answer

honestly with probably $2/3$, but to lie with probability $1/3$. This allows a student who did cheat to claim that they were lying, as per the directions. However, with many students, the noise should average out, and the professor can still learn approximately how many students cheated on the test.

We can apply this same approach if we want to make a certain algorithm differentially private. Instead of publishing a dataset (or a function of a dataset) directly, we can publish a *noisy version of it*. Much of the technical complexity in differential privacy goes into trying to design exactly how to choose the noise in a way that maintains as much of the "signal" of the underlying dataset as possible.

23.6 The Laplace mechanism

The *Laplace mechanism* gives a very general way to take an aggregate statistic f and modify it to have ϵ-differential privacy for any choice of ϵ.

First, we need to define the *global sensitivity* of the function f. This aims to capture the maximal change in f that results from changing one of the function's n inputs. Intuitively, the higher the sensitivity of a function is, the more each user's data can affect the functions output and therefore the more noise we will have to add to achieve differential privacy.

The definition of sensitivity and the Laplace mechanism generalize nicely to functions that output multiple real numbers.

Definition 23.6.1 (Global Sensitivity)**.** The global sensitivity of a function $f\colon \mathcal{D}^n \to \mathbb{R}$, for some domain $\mathcal{D}$, is:

$$GS_f = \max_{\text{neighbors } x,x' \in \mathcal{D}^n} \|f(x) - f(x')\|.$$

Here, "neighbors" inputs in $\mathcal{D}^n$ that differ in only a single coordinate.

We will also need to define the zero-mean real-valued Laplace distribution, which is parameterized by a value $b \in \mathbb{R}$. The probability distribution function is:

$$h(y) = \frac{1}{2b} e^{-\frac{|y|}{b}}$$

Now, the Laplace mechanism for achieving a differential privacy release of the function $f(x_1, \ldots, x_n)$ is:

1. Compute the true statistic $y \leftarrow f(x_1, \ldots, x_n)$.
2. Let $b \leftarrow \frac{GS_f}{\epsilon}$ be the global sensitivity of f.
3. Sample a random noise value ν from the Laplace distribution with mean 0 and parameter b.
4. Output the noised statistic $y + b$.

Handling non-numerical data. The Laplace mechanism gives a very clean way to provide differential privacy for releasing real-valued aggregate statistics about a dataset—where the aggregate statistic is just a number. But in many applications, we need to publish *strings*, such as names or zip codes or birth dates.

One way to handle strings is to convert them into numerical statistics and them apply differential privacy. For example, we could publish (1) the number of users with name "Alice," (2) the number of users with name "Bob," and so on. Then we are back to working in a world of numerical statistics and we can against use the Laplace mechanism.

23.7 *Challenges with differential privacy*

As we discussed in Section 23.4, the more statistics you publish about a database, the larger the differential-privacy parameter ϵ grows. A consequence is that if you want to publish many statistics about a dataset, you will have to add a huge amount of noise to the statistics you publish to achieve ϵ-differential privacy any reasonable value of ϵ. This is a major headache in practice.

A second issue is that the designer of the system has to choose what value of ϵ to use. What value is good enough? The theory of differential privacy cannot tell you—it's really a subjective question.

Yet a third issue is that in many applications, a company has to publish an aggregate statistic every day or week. Since the ϵ values "add up," after a short amount of time the effective ϵ value can end up being so large as to render the differential privacy guarantee meaningless.

Part VI

Conclusions

24
Conclusions

To conclude, we will explore five case studies that exemplify what we have covered so far.

24.1 Authentication: OPM Hack

In order to get security clearance to view classified documents, it is necessary to fill out a form called an SF86 that covers all kinds of personal details from relationships and mental health to drug use and finances—some of the most sensitive data there is. Many, many people have filled these out—around 2.8 million people have some level of current security clearance. The goal of this invasive background check was to understand people's exposure to blackmailing.

These records are stored in a database at the Office of Personnel Management. In 2015, the OPM announced that 20 million of these records have been exposed. This was a big problem for the US Government. Not only did it expose exactly how to blackmail every person with security clearance, but also records of CIA employees were not stored in this OPM database—this meant that if you knew someone had a security clearance but they were not in the database, you had an idea that they might be a CIA agent.

24.1.1 How it Happened

Learn Contractor Credentials. First, the attacker somehow got a contractor's credentials. This could have been through phishing or some other means. The system did not use two-factor authentication, and only around 1% of OPM users used smart cards. Importantly, the contractor did not need to have many privileges on the system—perhaps only enough permissions to log in.

Compromise Root Account. Then the attacker likely compromised the root account on a local machine. Given that these systems were old, this was likely easy—old version of Windows were not particularly

careful about protecting access to the root account. One way to do this was by scheduling a job for the future. Windows allowed unprivileged users to schedule jobs that would run as the system user, so a command like `at 16:05 /interactive "cmd.exe"` would open a command prompt as the system user at 4:05 PM.

Learn Administrator Credentials. Even with access to the root account on a local machine, the attacker needed to learn credentials to be able to log into one of the machines with access to the database. Windows did not store the password of the currently logged in user, but it did store the *hash* of the password and uses that to authenticate to the server. This means that the client does not need the cleartext password to log in!

As we have covered many times, passwords are terrible for authentication. Signatures, as used with standards like FIDO2/WebAuthn, provide much stronger security and should be used whenever possible.

24.2 Transport Security: POODLE

TLS is used all over the web. Many servers, from data centers to embedded hardware devices, like doorbells, implement TLS. Some of these servers take a long time to upgrade, so even if a client supports the latest version of TLS (e.g., TLS 1.3), it might be unable to use TLS 1.3 to communicate with some servers. To handle this situation, TLS implements version negotiation; the client and server should agree to use the most recent version of TLS supported by both sides. However, the way this negotiation was implemented was subtly insecure.

When an old server receives a message from a new client speaking a new version of TLS that the server does not support, the server might reply back with garbage data. This could be due to a bug in the server implementation, which was never found because the developers didn't think in advance to test against various ways that future clients sending messages from a future version of TLS. Despite the fact that this is a server bug, users want to still be able to communicate with this device; after all, communicating with the device worked fine before the client upgraded to support a new version of TLS. As a result, clients that receive garbage data in response to their TLS connection attempt will try to downgrade the version of TLS that they use to connect in their next attempt. However, the "garbage" that the client is reacting to was not authenticated. Therefore, an adversary could inject garbage to force a client to downgrade to an older TLS/SSL version.

Old versions of TLS/SSL have well-known weaknesses in their

cryptographic protocol. In particular, SSL 3.0 computed the MAC of the message (for authentication) before encrypting the message (MAC-then-encrypt), rather than the encrypt-then-MAC approach that we discussed in the authenticated encryption lecture. This allowed the attacker to send corrupted ciphertexts and see how the client or server respond to them. In combination with MAC-then-encrypt, SSL 3.0 also used a particular encryption construction, called cipher-block-chaining (CBC) mode, whereby changing bits in one part of the ciphertext caused corresponding changes to another part of the plaintext. Finally, the plaintext payload included padding, constructed with a well-known scheme, which was checked after decryption to make sure it was not corrupted; if the padding was corrupted, the connection was terminated. However, this gives a signal to the adversary as to whether their corrupted ciphertext happens to match the expected padding or not. We saw the CBC padding oracle attack earlier in the lecture on authenticated encryption.

The POODLE attack was a combination of these two weaknesses (downgrade attack and CBC padding oracle), together with running adversary-supplied code in Javascript in the victim user's web browser, as a way of sending partially-adversary-chosen requests over the TLS/SSL connection. The result is that the adversary can recover other data sent to the server over the same connection, such as the victim's cookie.

24.3 Platform Security: Sony PS3 Hack

The Sony PS3 originally could boot Linux and Windows. Since the hardware was subsidized by the games that they sold, PS3s were cheaper than comparable PCs, and for that reason PS3s were a popular option for a cheap PC. In a software update, Sony disabled the ability to run a custom operating system. Like the iPhone and other systems we discussed, PS3s then used secure boot to ensure that they only boot Sony-signed operating systems.

Sony used EC-DSA for their signatures, which resulted in signatures along the lines of:

$$\sigma = (g^r, r + H(\mathsf{pk}||g^r||m) \cdot \mathsf{sk}) \pmod q$$
$$\sigma' = (g^r, r + H(\mathsf{pk}||g^r||m') \cdot \mathsf{sk}) \pmod q$$

In EC-DSA, r is supposed to be a long random number, serving as a nonce. However, Sony re-used these nonces in pairs of signatures. This meant that their signatures revealed their secret key! This allowed others to sign their own operating systems.

Importantly, Sony had a plan for updating the PS3 firmware that allowed them to ship a fix for this attack. However, attackers quickly found flaws in every other update they shipped—it is very hard to secure a device that the attacker has unconstrained access to.

24.4 Software Security: WannaCry Ransomware

Ransomware is a type of malware that encrypt important-looking files on the infected system and demands a payment in Bitcoin to decrypt the files. This is inconvenient and upsetting for personal computers, but for enterprise computer systems this can cause huge monetary losses. For hospital systems, this can even lead to loss of life. This WannaCry randomware infected hundreds of thousands of computers and caused billions of dollars of damange, but did not make much money due to bad payment systems and slow decryption.

The bugs used to enable this were part of an exploit developed and kept secret by the NSA called EternalBlue. It took advantage of several C bugs, including an invalid cast, a parser bug, and an allocation bug, to eventually achieve remote code execution over the network on a Windows system.

The NSA intended to keep these exploits to themselves, and thus did not inform Microsoft (or anyone else) about the bugs in their software. However, an NSA contractor took terabytes of NSA data home with him, including this exploit. On his home computer, he ran Kaspersky antivirus. Importantly, Kaspersky sends suspicious files home for analysis. Wall Street Journal reported that this was likely how the exploit leaked and became a part of malware.

In order to spread to many computers, WannaCry looked something like the following:

1. Connect to a website at a random-looking address and exit if it succeeds. Security teams would often analyze software that they thought may include malware by running it in a VM, allowing network requests to succeed, and watching what happens. This random-looking domain did not really exist, so this may have been a way to try to detect when the software is running in a VM and make it behave "normally“.
2. Install Tor and connect to command-and-contral infrastructure.
3. Encrypt all files with a fixed set of extensions with RSA and AES.
4. Demand a ransome to be paid to one of four static Bitcoin addresses.

5. Spread itself by trying to perform the exploit on all IPs in the local network.

When designing a system, it is prudent to have the system as simple as possible, since less software leads to fewer bugs. And of course, any bug is a security bug—each of the bugs used in the EternalBlue exploit did not look like a security bug, but the combination of them allowed for a powerful exploit.

24.5 Privacy: US Census

The US Census, performed every decade, collects data used to allocate seats in the House of Representatives and by many researchers. The Census Bureau is mandated to make this information public, but is forbidden by law from publishing any data that allows individuals to be identified.

In the 2020 census, the bureau used differential privacy to protect released data from de-identification. However, they used $\epsilon = 19.61$ to avoid adding so much noise that the data lost its utility. This meant that if the probability of some event happening to an individual without the release of these data was p, the release of these data with $\epsilon = 19.61$ was guaranteed to make the probility at most $\approx e^{19.61}p$, or around a million times the original probability.

Part VII

Appendices

A
Factoring integers

The problem of integer factorization was central to 20th-century cryptography. Breaking the one-wayness of the RSA trapdoor one-way function (Chapter 6), for example, is no harder than factoring integers. In this chapter, we will see a couple of surprisingly powerful algorithms for factoring integers.

We will only consider factoring numbers of the form $N = pq$, for distinct odd primes p and q. (The general case is not too much more challenging.) Throughout, let $n = \lceil \log_2 N \rceil$ be the bitlength of the number to factor.

A.1 Background

Trial division. We can factor N by trying to divide N by each of the primes of size $\leq \sqrt{N}$ and checking whether the result is an integer. If so, we have found a factor of N. Since at least one of the two factors of N is in $\{1, \ldots, \sqrt{N}\}$, this algorithm ("trial division") runs in time roughly $\sqrt{N} = 2^{n/2}$.

Trial division is an *exponential time* algorithm, since it runs in time $2^{\Omega(n)}$, where the bitlength n is the size of the number to be factored. The best known factoring algorithms run in *sub-exponential time* $2^{O(n^c)}$, for some constant $c < 1$.

Euclid's algorithm. An important subroutine in almost all factoring algorithms is Euclid's polynomial-time algorithm for computing the greatest common divisor of two integers x and y.

The principle of Euclid's algorithm is that

$$\gcd(x, y) = \gcd(x, y \bmod x) \qquad \text{and} \qquad \gcd(x, 0) = x.$$

So, for example, if we want to compute $\gcd(46, 12)$, we can compute it as:

$$\gcd(46, 12) = \gcd(12, 10) = \gcd(10, 2) = 2.$$

In this discussion, we draw on Arjen Lenstra's very nice survey on factoring Arjen K Lenstra. "Integer factoring". In: *Designs Codes, and Cryptography* 19.2/3 (2000).

Difference of squares. The second key idea is that, if we can find two numbers $x, y \in \mathbb{Z}$ whose squares are congruent modulo N, we can use these numbers to factor N:

$$\begin{aligned} x^2 &= y^2 && \pmod N \\ x^2 - y^2 &= 0 && \pmod N \\ (x+y)(x-y) &= 0 && \pmod N \end{aligned}$$

If $x = \pm y$, then this relation is not helpful to us. But if $x \neq \pm y$, then we know that $x + y \neq 0 \bmod N$ and $x - y \neq 0 \bmod N$. So we have:

$$(x+y)(x-y) = kN \quad \in \mathbb{Z},$$

for some positive integer $k \in \mathbb{Z}$. Then $x + y$ must be a multiple of one of the factors of N (but not both), and $\gcd(x+y, N)$ reveals a factor of N.

The goal of many factoring algorithms—including the one we will see today—is finding these integers x and y whose squares are congruent modulo N.

It is not necessarily obvious that the useful pairs (x, y) will ever exist. The key idea is that, modulo $N = pq$, ever number in $\mathbb{Z}_N^*$ either has four square roots or has none. If an element in $\mathbb{Z}_N^*$ has four square roots then the roots are of the form $r, -r, s, -s$. In this case, a pair $(\pm r, \pm s)$ yields the sort of relation that we need to factor.

A.2 Dixon's algorithm

Dixon's algorithm is one of the simplest sub-exponential-time factoring algorithms. It gives a fast method for finding two numbers whose squares are congruent modulo N. Once we have these squares, we can use them to factor as in Section A.1

A.2.1 The idea

The principle of Dixon's algorithm is that we will pick many random numbers $r \in \mathbb{Z}_N^*$ and square them modulo the integer N we would like to factor.

Say that we are somehow able to find numbers r, r' such that

$$\begin{aligned} r^2 &= 2 \cdot 3^2 \cdot 5 && \pmod N \\ r'^2 &= 2 \cdot 5 && \pmod N, \end{aligned}$$

then we know that:

$$\begin{aligned} (rr')^2 &= 2^2 \cdot 3^2 \cdot 5^2 && \pmod N \\ (rr')^2 &= (2 \cdot 3 \cdot 5)^2 && \pmod N \end{aligned}$$

and now we have two numbers whose squares are congruent modulo N:

$$x = rr' \qquad \text{and} \qquad y = 2 \cdot 3 \cdot 5.$$

If we are lucky, this is the useful type of congruence that we can use to factor N (i.e., $rr' \neq \pm 2 \cdot 2 \cdot 5 \bmod N$).

The principle of Dixon's algorithm is to generate many such rs and then use linear algebra to find a subset of them whose product modulo N is a perfect square.

A.2.2 The algorithm

Input: An integer $N = pq$ for odd primes p and q. A parameter $B \in \mathbb{N}$, which we refer to as "the size of the factor base."

Output: The factors (p, q) of N.

1. **Collect linear relations.** Maintain a list L of pairs of (a) an element in $\mathbb{Z}_N^*$ and (b) vectors over $\mathbb{Z}_2^B$. Repeat until L contains $B + 1$ pairs:
   - Sample $r \xleftarrow{R} \mathbb{Z}_N^*$.
   - Compute $s \leftarrow (r^2 \bmod N)$.
   - Attempt to write s as a product of the first B primes:
   $$s = 2^{e_2} 3^{e_3} 5^{e_5} \dots$$
   - If successful, add the pair $(r, (e_2, e_3, e_5, \dots))$ to the list L.

   If a number completely splits into prime factors $\leq B$, we say that the number is "B-smooth."

2. **Solve linear system.** Let $L = \{(r_1, \mathbf{v}_1), (r_2, \mathbf{v}_2), \dots\}$. Find a non-zero combination of the vectors in L that sums to zero modulo 2. That is, find $S \subseteq [B + 1]$ such that
   $$\sum_{i \in S} \mathbf{v}_i = (0, 0, 0, \cdots, 0) \qquad \in \mathbb{Z}_2^B.$$
   Letting
   $$(e_2, e_3, e_5, \dots) \leftarrow \sum_{i \in S} \mathbf{v}_i,$$
   we then have a difference of squares:
   $$\left(\sum_{i \in S} r_i \right)^2 = (2^{(\frac{e_2}{2})} \cdot 3^{(\frac{e_3}{3})} \cdot 5^{(\frac{e_5}{2})} \cdots)^2 \qquad (\bmod N).$$

   We can find the set S using Gaussian elimination in roughly B^3 time. Since the set of vectors will be extremely sparse, there are faster methods that implementers use in practice.

3. **Use Euclid's algorithm to try to factor N.** We can take:
   $$x = \left(\sum_{i \in S} r_i \right)$$
   $$y = 2^{(\frac{e_2}{2})} \cdot 3^{(\frac{e_3}{3})} \cdot 5^{(\frac{e_5}{2})} \cdots)$$
   and compute $\gcd(x + y, N)$. With probability roughly $1/2$, over the random choice of the rs, this will yield a factor of N.

A.2.3 The analysis.

The costs of the three steps of the algorithms are:

1. Each iteration of the loop requires us to try to factor a number into primes $\leq B$. We can factor in this way by trial division using time roughly B.

   I'm ignoring any $\log B$ factors, which do matter very much in practice.

   The question then is how many trials it will take for us to find a single smooth number. For a smoothness bound B, let's say for now that it takes $T(B)$ trials—we will look into the precise value of $T(B)$ in a moment..

2. Solving the linear system using Gaussian elimination takes roughly B^3 time.

3. Run Euclid's algorithm—the time required here is negligible compared to the time of the first two steps. This step runs in time $\text{poly}(\log N) = \text{poly}(n)$.

Putting everything together, we have that factoring an n-bit number with a factor base of size B takes time:

$$B \cdot T(B) + B^3 + \text{poly}(n). \tag{A.1}$$

Smoothness probabilities. The key question that we need to answer to complete the analysis is

> "If we pick an integer uniformly at random from $\{1, \ldots, N\}$, what is the probability that the integer will be B-smooth?"

The convention is to denote the number of B-smooth numbers in $\{1, \ldots, N\}$ as $\Psi(N, B)$. When B is "not too small," we have:

$$\Psi(N, B) \approx N \cdot u^{-u+o(1)} \qquad \text{for } u = \frac{\log N}{\log B}.$$

For many more details on these estimates, take a look at Granville's very nice survey on smooth numbers. Andrew Granville. "Smooth numbers: computational number theory and beyond". In: *Algorithmic number theory: lattices, number fields, curves and cryptography* 44 (2008), pp. 267–323

The probability of a random number modulo N being B-smooth is then $\Psi(N, B)/N$ and the expected number of tries it will take for us to find a smooth number is:

$$T(B) = 1/\Psi(N, B).$$

We are being slightly imprecise—we actually need to know the number of squares (quadratic residues) modulo N that are smooth. But heuristically, we can assume that quadratic residues behave like random integers modulo N for the purposes of smoothness.

Now we can plug this estimate into the expression (A.1) for the running time of Dixon's algorithm and we can solve for the value of B that minimizes the running time. In particular, to minimize the running time we want:

$$B \approx T(B) \approx N/\Psi(N, B) = u^u,$$

for $u = (\log N)/(\log B)$.

$$\begin{aligned} B &= u^u \\ \log B &= u \log u \\ \log B &= \frac{\log N}{\log B} \log \frac{\log N}{\log B} \\ \log^2 B &\approx \log N \log\log N \\ \log B &\approx \sqrt{\log N \log\log N} \\ B &\approx \exp(\sqrt{\log N \log\log N}). \end{aligned}$$

If we plug this value of B into Dixon's algorithm, we get a running time of

$$\exp(O(\sqrt{\log N \log\log N}) = 2^{O(\sqrt{n \log n})}.$$

Bibliography

Bellare, Mihir and Phillip Rogaway. "Random oracles are practical: A paradigm for designing efficient protocols". In: *ACM Conference on Computer and Communications Security*. 1993.

Bittau, Andrea et al. "Hacking blind". In: *IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 227–242.

Diffie, Whitfield and Martin E Hellman. "New Directions in Cryptography". In: *Transactions on Information Theory* 22.6 (1976).

– "Exhaustive Cryptanalysis of the NBS Data Encryption Standard". In: *Computer* 6.10 (1977), pp. 74–84.

Duong, Thai and Juliano Rizzo. *Flickr's API Signature Forgery Vulnerability*. `https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html`. Sept. 2009.

Granville, Andrew. "Smooth numbers: computational number theory and beyond". In: *Algorithmic number theory: lattices, number fields, curves and cryptography* 44 (2008), pp. 267–323.

Hastad, J. et al. "A Pseudorandom Generator from any One-way Function". In: *SIAM Journal on Computing* 28.4 (1999), pp. 1364–1396.

Hill, Mark D. et al. "On the Spectre and Meltdown Processor Security Vulnerabilities". In: *IEEE Micro* 39.2 (2019), pp. 9–19.

Honan, Mat. *How Apple and Amazon Security Flaws Led to My Epic Hacking*. `https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/`. Aug. 2012.

Koscher, Karl et al. "Experimental security analysis of a modern automobile". In: *IEEE Symposium on Security and Privacy*. 2010.

Lamport, Leslie. *Constructing Digital Signatures from a One Way Function*. Tech. rep. Oct. 1979.

Lenstra, Arjen K. "Integer factoring". In: *Designs Codes, and Cryptography* 19.2/3 (2000).

Lenstra, Arjen K, Thorsten Kleinjung, and Emmanuel Thomé. "Universal security". In: *Number Theory and Cryptography*. 2013.

Motoyama, Marti et al. "Re: CAPTCHAs—Understanding CAPTCHA-Solving Services in an Economic Context". In: *Proceedings of the 19th USENIX Security Symposium*. Washington, DC, Aug. 2010.

Müller, Jens et al. “Practical decryption exfiltration: Breaking pdf encryption”. In: *ACM CCS*. 2019.

Nemec, Matus et al. “The Return of Coppersmith’s Attack: Practical Factorization of Widely Used RSA Moduli”. In: *CCS*. 2017.

Rivest, Ronald L., Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

Thompson, Ken. “Reflections on trusting trust”. In: *Communications of the ACM* 27.8 (1984).

Trew, J. *‘Find My iPhone’ exploit may be to blame for celebrity photo hacks (update)*. `https://www.engadget.com/2014-09-01-find-my-iphone-exploit.html`. Sept. 2014.

Wegman, Mark N and J Lawrence Carter. “New hash functions and their use in authentication and set equality”. In: *Journal of computer and system sciences* 22.3 (1981).

Yilek, Scott et al. “When private keys are public: Results from the 2008 Debian OpenSSL vulnerability”. In: *SIGCOMM*. 2009.

Zetter, Kim. *Palin e-mail hacker says it was easy*. `https://www.wired.com/2008/09/palin-e-mail-ha/`. Sept. 2008.