

Real-World Cryptography

David Wong



MANNING



MEAP Edition
Manning Early Access Program
Real-World Cryptography
Version 9

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Dear reader,

Getting into cryptography is a road paved with difficult challenges and many uncertainties about what there is to learn, or what's important to know. "What's the quickest path to get there?" you may have often asked yourself. And you were right to wonder. The amount of information out there can be intimidating, especially when most of it is outdated or goes deeply into theoretical rabbit holes.

Fear not, you've come to the right place. This book is here for you, the curious, the student, the engineer who wants to know (or needs to know) more about cryptography.

In this book, I first summarize the state of real world cryptography, which is the cryptography that is used every day by you and me, as well as the large companies of this world. Almost everything you learn in this book is practical and useful.

The rest of the book looks into what the real world cryptography of tomorrow is starting to look like. Many topics like cryptocurrencies and post-quantum cryptography are often ignored from the applied cryptography literature, but they are equally as important. The field of applied cryptography is currently booming and changing rapidly, and one has to keep up with its advances or fear being left out behind.

This is not a reference book, and so I have avoided most math (although not all) and deeply technical details. Most of what you will learn will first teach you about cryptographic objects as if they were black boxes that serve useful purposes if given the right inputs. The book also strives to give you the right amount of detail, just enough to push your curiosity forward and give you a peek of what these black boxes look like from the inside. Each chapter is accompanied with many real world examples of what you just learned.

There are no specific prerequisites for reading this book aside from having a taste for computer science and having heard the word "encryption" before. Students and engineers should be able to pick up this book and have most of their questions answered.

Prepare yourself, as I will now demystify the real world of cryptography before your eyes.

—David Wong

brief contents

1 *Introduction*

PART 1: PRIMITIVES - THE INGREDIENTS OF CRYPTOGRAPHY

- 2 *Hash functions*
- 3 *Message authentication codes*
- 4 *Authenticated encryption*
- 5 *Key exchanges*
- 6 *Asymmetric encryption and hybrid encryption*
- 7 *Signatures and zero-knowledge proofs*
- 8 *Randomness and secrets*

PART 2: PROTOCOLS - THE RECIPES OF CRYPTOGRAPHY

- 9 *Secure transport*
- 10 *End-to-end encryption*
- 11 *User authentication*
- 12 *Crypto as in cryptocurrency?*
- 13 *Hardware cryptography*
- 14 *Next generation and post-quantum cryptography*

PART 3: CONCLUSION

- 15 *Final words: The dangers of implementing and using cryptography*

Introduction

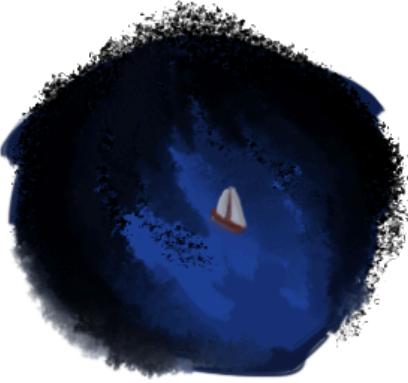
This chapter covers

- What cryptography is about.
- The difference between theoretical cryptography and real-world cryptography.
- What you will learn throughout this adventure.

Greetings traveler,

Sit tight as you're about to enter a world of wonders and mystery — the world of cryptography. Cryptography is the ancient discipline of securing the unfortunate situations that are troubled with malicious characters. In order to do so, many spells exist, spells you will have to master. Many have been there to learn the craft, but few have survived the challenges that stand in the way. Many exciting adventures await you. We'll uncover how cryptographic algorithms can secure communications, identify our allies, protect treasures from our enemies. Many great lessons you will learn, but difficult it will be to sail straight through the cryptographic seas. As cryptography is the foundation of all security in our world, the slightest mistake could be deadly.

REMEMBER If you find yourself lost, keep moving forward. It will all eventually make sense.



1.1 A Peek Into the World of Cryptography

Our journey starts here, with an introduction to cryptography, the science behind protecting protocols that adversaries could actively attempt to sabotage.

NOTE

A **protocol** is any scenario where multiple participants (including potentially malicious ones) are attempting to achieve something. Imagine the following premise: you want to leave your magic sword unattended for a few hours so you can take a nap. One protocol to do this could be the following:

1. Deposit weapon on the ground.
2. Take nap under a tree.
3. Recover weapon from the ground.

In ancient times, when kings and generals were busy betraying each other and planning coups, their biggest issue was finding ways to **share confidential information with those they trusted**. From these problems, the idea of cryptography was born. It took some time, and some effort, for cryptography to become a serious field of study. But today, it is used all around us in order to provide the most basic services in the face of our chaotic and adverse world.

The story of this book is about the practice of cryptography on our modern planet. It takes you on an expedition throughout the computing world to cover cryptographic protocols in use today, it shows you what are the parts they are made of, and how everything fits together. There will be (almost) no scary math formulas. The purpose of this book is to demystify cryptography, survey what is considered useful nowadays, and give intuition about how things around you are built. This book is intended for the curious people, the interested engineers, the adventurous developers and the inquisitive researchers.

Chapter 1 initiates a tour of the world of cryptography. We will discover the different types of cryptography, which ones matter to us, and how the world agreed on using them.

1.1.1 Symmetric Cryptography: Symmetric Encryption

One of the fundamental pieces of cryptography is **symmetric encryption**. It is used in a majority of cryptographic algorithms in this book, and it is thus extremely important. We introduce this new concept here via our very first protocol. Let's imagine that **queen Alice** needs to send a letter to **lord Bob** who is a few castles away. She asks her loyal messenger to ride his camel and battle his way through the filthy lands ahead in order to deliver the precious message to lord Bob. Yet, she is very suspicious; even though her loyal messenger has served her for many years, she wishes the message in transit to **remain secret from all passive observers, including the messenger**. You see, the letter most likely contains some controversial gossip about the kingdoms on the way.

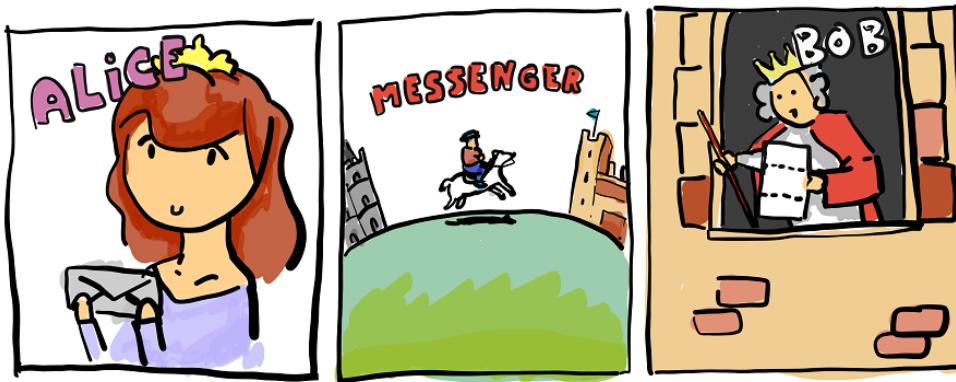


Figure 1.1 Alice asks her (potentially malicious) messenger to deliver a confidential message to Bob.

What queen Alice needs is a protocol that acts like she would effectively hand the message to lord Bob herself. No middle men. This is quite an impossible problem to solve in practice, unless we introduce cryptography into the equation. And this is what we ended up doing ages ago by inventing a new type of cryptographic algorithm called a **symmetric encryption algorithm** (or a **cipher**).

NOTE

By the way, a **type of cryptographic algorithm** is often referred to as a **primitive**. You can think of a primitive as the smallest useful construction you can have in cryptography, and it is often used with other primitives in order to build a protocol. It is mostly a term, and has no particularly important meaning, though it appears often enough in the literature that it is good to know about it.

Let's see next how an encryption primitive can be used to hide queen Alice's message from the messenger. Imagine for now that the primitive is a black box (we can't see what it's doing internally) that provides two functions:

- `ENCRYPT()`

- **DECRYPT()**

The first function, **ENCRYPT()** works by taking a **secret key** (usually a very large number) and a **message**. It then outputs a series of numbers that look like they were chosen randomly. We will call that output the encrypted message.

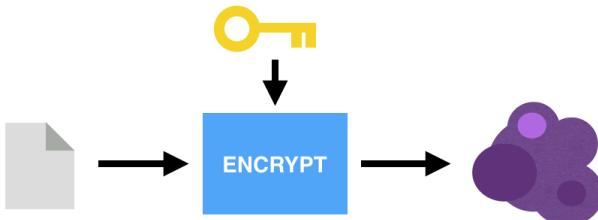


Figure 1.2 The ENCRYPT function takes a message and a secret key, and outputs the encrypted message — a long series of numbers that look like random noise.

The second function, **DECRYPT()**, is the inverse of the first one: by taking the same **secret key** and the random output of the first function (the encrypted message), it finds back the original message.

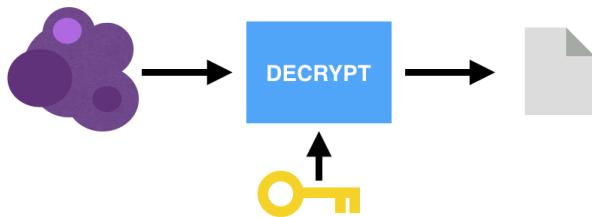


Figure 1.3 The DECRYPT function takes an encrypted message and a secret key, and returns the original message.

To make use of this new primitive, queen Alice and lord Bob have to first meet in real life and decide on what **secret key** to use. Later, queen Alice can use the provided **ENCRYPT()** function to protect a message with the help of the **secret key**. She then passes the encrypted message to her messenger, who eventually delivers it to lord Bob. Lord Bob then uses the **DECRYPT()** function on the encrypted message, with the same secret key, to recover the original message.

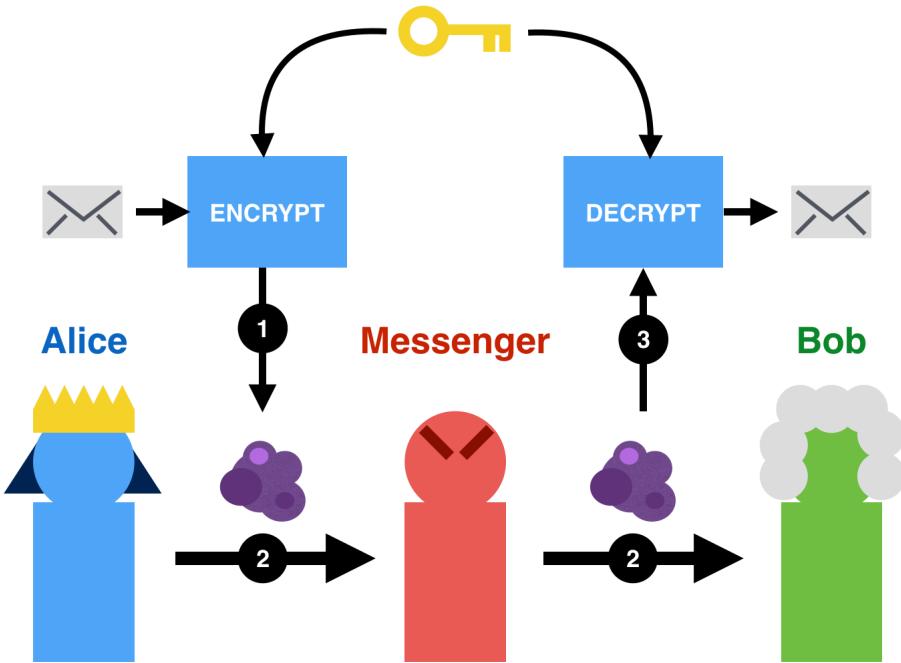


Figure 1.4 (1) Alice uses the ENCRYPT function with a secret key to transform her message into noise. (2) She then passes the encrypted message to her messenger, who will not learn anything about the underlying message. (3) Once Bob receives the encrypted message, he can recover the original content by using the DECRYPT function with the same secret key Alice used.

During this exchange, all the messenger had was something that looked very random, and that would not reveal anything about the original message.

Effectively, what we did is that we augmented our insecure protocol into a secure one thanks to the help of cryptography. The new protocol makes it possible for queen Alice to deliver a confidential letter to lord Bob without anyone (but lord Bob) learning the content of it. This process of adding noise to render something "indistinguishable from random" (thanks to the help of a secret key) is a common way of securing a protocol in cryptography. We will see more of this as we learn more cryptographic algorithms in the next chapters.

By the way, symmetric encryption is part of a larger category of cryptography algorithms called **symmetric cryptography** or **secret-key cryptography**. This is due to the **same key** being used by the different functions exposed by the primitive.

1.1.2 Kerckhoff's Principle: Only the Key is Kept Secret

To design a cryptographic algorithm (like our encryption primitive) is an easy task, but to design a **secure** cryptographic algorithm is not for the faint of heart. While we shy away from creating such algorithms in this book, we *do* learn how to recognize the good ones to use. This can be difficult, as there is more choice than one can ask for the task. Hints can be found in the repeated failures of the history of cryptography, as well as the lessons that the community has learned from them. As we take a look at the past, we will grasp at what turns a cryptographic algorithm into a trusted-to-be-secure one.

Hundreds of years have passed and many queens and lords have been buried. Since then, paper has been abandoned as our primary means of communication in favor of better and more practical technologies. Today we have access to powerful computers as well as the internet. More practical, sure, but this also means that our previous malicious messenger has become much more powerful. He is now everywhere: the Wi-Fi in the Starbucks cafe you're sitting in, the different servers making up the internet and forwarding your messages, the machines running our algorithms. Our enemies are now able to observe many more messages, as each request you make to a website might pass through the wrong wire, and become altered or copied in a matter of nanoseconds without anyone noticing.

With that influx of information, the security of nation states became of utmost importance in periods of war. For a long time, cryptography was enforced as a secret science, available only to highly classified circles. This has fortunately changed and cryptography is now studied in the open throughout the world. Yet, looking at our past, history contains many instances of encryption algorithms falling apart, being broken by secret state organizations and failing to protect their messages. From this, many lessons were learned and we slowly understood how to produce good cryptography.

NOTE

A cryptographic algorithm can be considered **broken** in many ways. For an encryption algorithm, you can imagine several ways to attack the algorithm: the secret key can be obtained somehow, messages can be decrypted without the help of the key, some information about the message can be revealed just by looking at the encrypted message, and so on. Anything that would somehow weaken the assumptions we made about the algorithm could be considered a break.

A strong notion that came out of this long process of discovery was that, to obtain confidence in the security claims made by such cryptographic primitives, the primitives themselves had to be analyzed in the open by experts. This is why cryptographers (the people who build) will usually use the help of **cryptanalysts** (the people who break) in order to analyze the security of a construction (see figure 1.5).

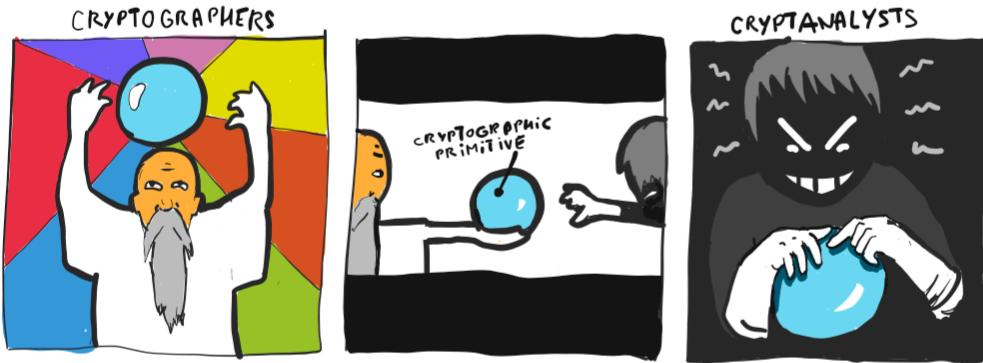


Figure 1.5 The importance of cryptographers and cryptanalysts in designing cryptographic algorithms: cryptographers build, cryptanalysts break.

Let's take the **Advanced Encryption Standard (AES)** encryption algorithm as an example. AES was the product of an international competition organized by the National Institute of Standards and Technology (NIST).

NOTE

NIST is a United States agency whose role is to define standards and develop guidelines for use in government-related functions as well as other public or private organizations. It has standardized many widely-used cryptographic primitives like AES.

The AES competition lasted several years, during which many volunteering cryptanalysts from around the world gathered to take a chance at breaking the various candidate constructions. After several years, once enough confidence was built by the process, a single competing encryption algorithm was nominated to become the Advanced Encryption Standard itself. Nowadays, most people have agreed that AES is a solid encryption algorithm and it is widely used to encrypt almost anything. For example, you use it every day when you browse the web.

While not every modern cryptographic algorithm came from such a competition, it has happened many times. The idea to build cryptographic standards in the open is related to a concept often referred to as **Kerckhoffs' principle**, which can be understood as: it would be foolish to rely on our enemies not to discover what algorithms we use, because they most likely will. Instead, let's be open about them.

If the enemies of queen Alice and lord Bob knew exactly how they were encrypting messages, how was our encryption algorithm secure? The **secret key** used by the encryption algorithm to encrypt and decrypt is the answer. The secrecy of the key makes the protocol secure, not the secrecy of the algorithm itself. This is a common theme in this book: all the cryptographic algorithms that we will learn about, and that are used in the real-world, are most often free to be studied and used. Only the secret keys that are used as input to these algorithms, when being used, are kept secret. "*Ars ipsi secreta magistro*" (an art secret even to its inventor) said Jean Robert du Carlet in 1644.

In the next section, we will talk about a totally different kind of cryptographic primitive, so let's use figure 1.6 to organize what we've learned so far.

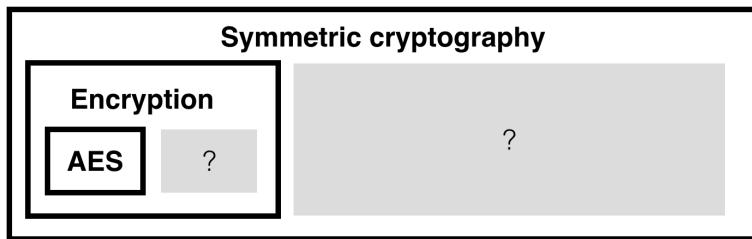


Figure 1.6 The cryptographic algorithms we have learned so far. AES is an instantiation of a symmetric encryption algorithm, which is a cryptographic primitive that is part of the broader class of symmetric cryptographic algorithms. There are many more of those (represented as question marks) that we will learn about in this book.

1.1.3 Asymmetric Cryptography

In our discussion about symmetric encryption, we said that Alice and Bob first met to decide on a symmetric key. This is a plausible scenario, and a lot of protocols actually do work like this. Nonetheless, this quickly becomes less practical as our scenario scales: do we need our web browser to meet with Google, Facebook, Amazon, and the other billions of websites before being able to securely connect to them? This problem, often referred to as **key distribution** has been a hard one to solve for quite a long time, at least until the discovery in the late 70's of another large and useful category of cryptographic algorithms called **asymmetric cryptography** or **public-key cryptography**. Asymmetric cryptography generally makes use of different keys for different functions (as opposed to a single key being used in symmetric cryptography), or provides different point of views to different participants. To illustrate what this mean, and how public-key cryptography helps to set up trust between people, we will now uncover a number of asymmetric primitives.

Note that this is only a glance at what we will learn in this book, as we will talk about each of these cryptographic primitives in more details in subsequent chapters.

KEY EXCHANGES

The first public-key algorithm discovered and published was a key exchange algorithm named after its authors — **Diffie-Hellman**. The Diffie-Hellman key exchange algorithm's main purpose is to establish a common secret between two parties. This common secret can then be used for different purposes for example, as a key to a symmetric encryption primitive.

In chapter 6, I will explain in details how Diffie-Hellman works, but for this introduction let's use a simple analogy in order to understand what a key exchange provides. Like many algorithms in cryptography, a key exchange must start with the participants using a common set of parameters. In our analogy, we will simply have them agree to use a square (as in figure 1.7).

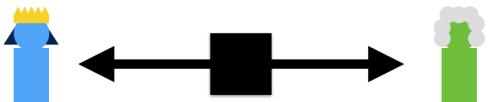


Figure 1.7 A Diffie-Hellman key exchange always requires two participants to first agree on a common set of parameters in order to perform a key exchange. Here, they agree on a square.

The next step is for Alice and Bob to choose their own random shape. Both of them go to their respective secret place, and out of sight, Alice chooses a triangle and Bob chooses a star. The objects they have chosen need to remain secret at all costs! The objects represent their **private keys** (see figure 1.8).

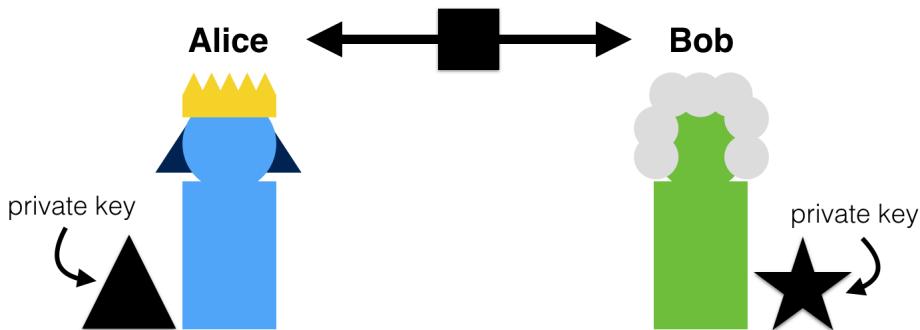


Figure 1.8 The first step of a Diffie-Hellman key exchange is to have both participants generate a private key. In our analogy, Alice chooses a triangle as her private key, whereas Bob chooses a star as his private key.

Once they have done that, they both individually combine their secret shape with the common shape they initially agreed on using. This gives out a unique shape, which represents their **public key** (see figure 1.9).

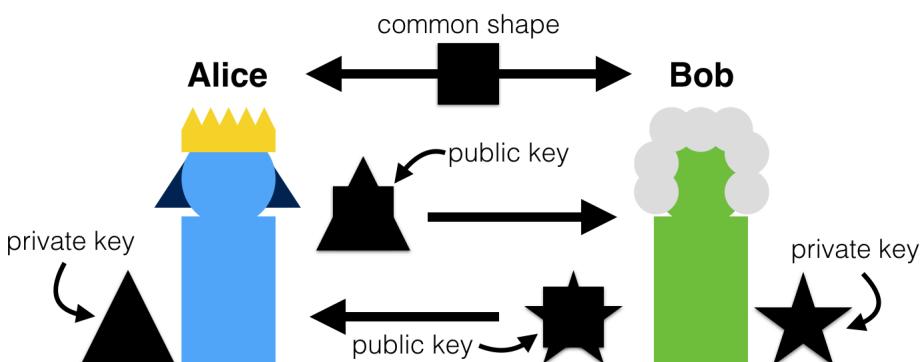


Figure 1.9 The second step of a Diffie-Hellman key exchange is to have both participants exchange their public keys. Participants derive their public keys by combining their private keys with the common shape.

We are now starting to see why this algorithm is called a public-key algorithm. It is because it requires a **key pair**: a private key and a public key.

What do they do with their public key? Well, they exchange it. These keys are now considered public information, as they could have been passively observed during the exchange.

The final step of the Diffie-Hellman key exchange algorithm is quite simple: Alice takes Bob's public key, and combines it with her private key. Bob does the same with Alice's public key, and combines it with his own private key. The result should now be the same on each side, in our example a shape combining a star, a square and a triangle (see figure 1.10).

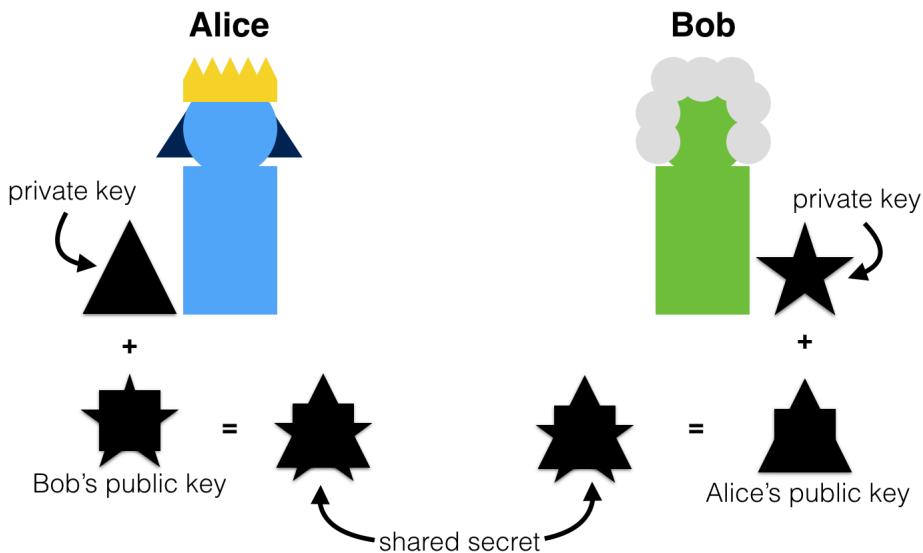


Figure 1.10 The final step of a Diffie-Hellman key exchange is to have both participants produce the same shared secret. To do this, Alice combines her private key with Bob's public key, and Bob combines his private key with Alice's public key. The shared secret cannot be obtained from solely observing the public keys.

This is now up to the participants of the protocol to make use of this **shared secret**. We will see several examples of this in this book, but the most obvious use case is to make use of it in an algorithm that requires a shared secret like symmetric cryptography algorithms. For example, Alice and Bob could now use the shared secret as a key to encrypt further messages with a symmetric encryption primitive.

To recap:

1. Alice and Bob exchanged their public keys, which is masking their respective private keys.
2. With the other participant's public key, and their respective private key, they can compute a shared secret.
3. An adversary who observes the public keys being exchanged doesn't have enough information to compute the shared secret.

NOTE

In our example, the last point is easily bypassable. Indeed, without the use of any private keys we can combine the public keys together to produce the shared secret. Fortunately, this is only a limitation of our analogy, but it works well enough for us to understand what a key exchange does.

In practice, a Diffie-Hellman key exchange is quite insecure. Can you take a few seconds to figure out why?

As Alice will accept any public key she receives as being Bob's public key, I could intercept the exchange and replace it with mine, which would allow me to impersonate Bob to Alice (and the same can be done to Bob). We say that a **man-in-the-middle** can successfully attack the protocol. How do we fix this? We will see in later chapters that we either need to augment this protocol with another cryptographic primitive, or we need to be aware in advance of what Bob's public key is.

But then, aren't we back to square one?

Previously Alice and Bob needed to know a shared secret, now Alice and Bob need to know their respective public keys. How do they get to know that? Is that a chicken-and-egg problem all over again? Well, kind of. As we will see, in practice public-key cryptography does not solve the problem of trust, but it simplifies its establishment (especially when the number of participants is large).

Let's stop here and move on to the next section, as you will learn more about key exchanges in chapter 5. We still have a few more asymmetric cryptographic primitives to uncover (see figure 1.11), to finish our tour of real-world cryptography.

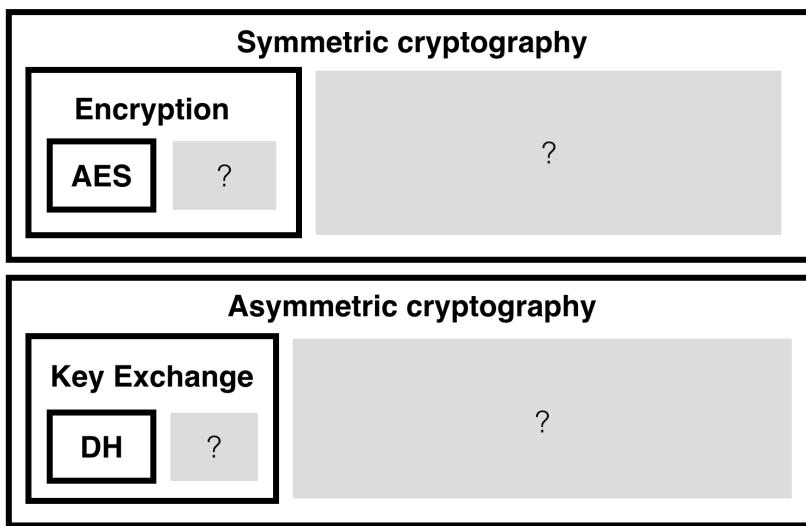


Figure 1.11 The cryptographic algorithms we have learned so far. Two large classes of cryptographic algorithms are symmetric cryptography (with symmetric encryption) and asymmetric cryptography (with key exchanges).

ASYMMETRIC ENCRYPTION

The invention of the Diffie-Hellman key exchange algorithm was quickly followed by the invention of the **RSA** algorithm named after Ron Rivest, Adi Shamir, and Leonard Adleman. RSA contains two different primitives: a **public-key encryption algorithm** (or **asymmetric encryption**) and a **(digital) signature scheme**. Both primitives are part of the larger class of cryptographic algorithms called asymmetric cryptography. In this section we will explain what these primitives do, and how they can be useful.

The first one, **asymmetric encryption**, has a very similar purpose to the symmetric encryption algorithm we've talked about previously: it allows one to encrypt messages in order to obtain confidentiality. Yet, unlike symmetric encryption, which had the two participants encrypt and decrypt messages with the same symmetric key, asymmetric encryption is quite different:

- it works with two different keys: a public key and a private key.
- it provides an asymmetric point of view: anyone can encrypt with the public key, but only the owner of the private key can decrypt messages.

Let's now use a simple analogy to explain how one can use asymmetric encryption. We start with our friend Alice again, who holds a private key (and its associated public key). Let's picture her public key as an open chest that she releases to the public for anyone to use (see figure 1.12).

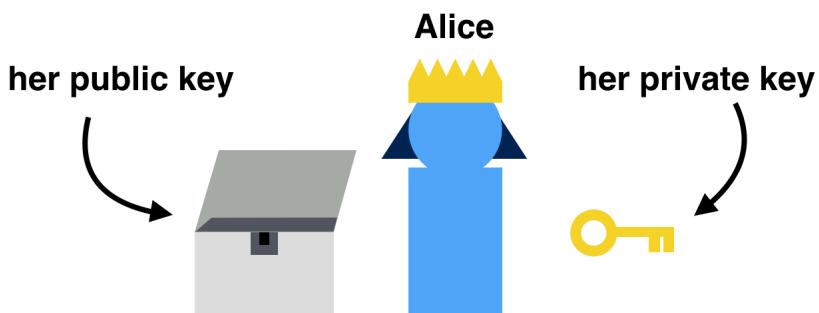


Figure 1.12 To use asymmetric encryption, Alice needs to first publish her public key (represented as an open box here). Now, anyone can use the public key to encrypt messages to her. And she should be able to decrypt them using the associated private key.

Now, you and I and everyone who wants can encrypt a message to her using her public key. In our analogy, imagine that you would insert your message into the open chest and then close it. Once the chest is closed, nobody but Alice should be able to open it. The box effectively protects the secrecy of the message from observers on the way.

The closed box (or encrypted content) can then be sent to Alice, and she can use her private key (only known to her, remember) to decrypt the ciphertext (see figure 1.13).

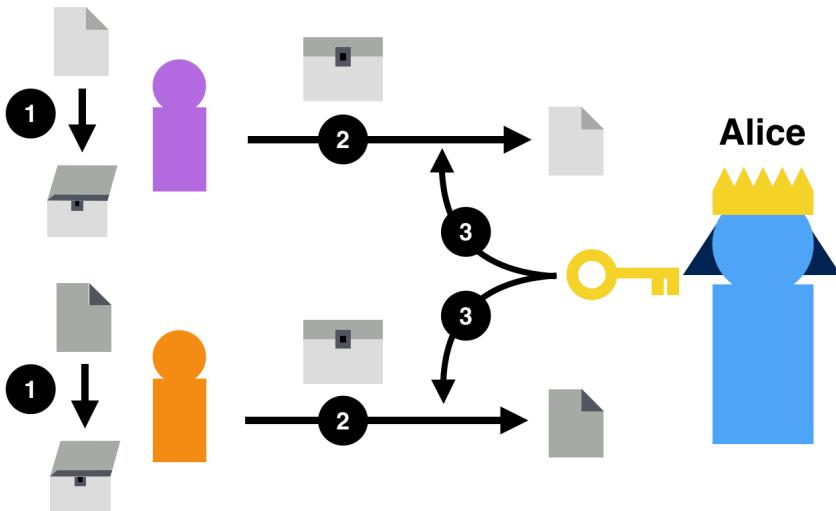


Figure 1.13 Asymmetric Encryption: (1) anyone can use Alice’s public key to encrypt messages to her. (2) After receiving them, (3) she can decrypt the content using her associated private key. Nobody is able to observe the messages directed to Alice while they are being sent to her.

In practice, using asymmetric encryption only works for small messages, and it is rather slow. For this reason, it is often used to encrypt a symmetric key, which is then used to further encrypt a message with a symmetric encryption primitive (which are usually quite faster). This latter construction is called **hybrid encryption** as it mixes public-key cryptography with symmetric cryptography.

Let’s summarize in figure 1.14 the cryptographic primitives we have learned so far. We are only missing one more to finish our tour of real-world cryptography!

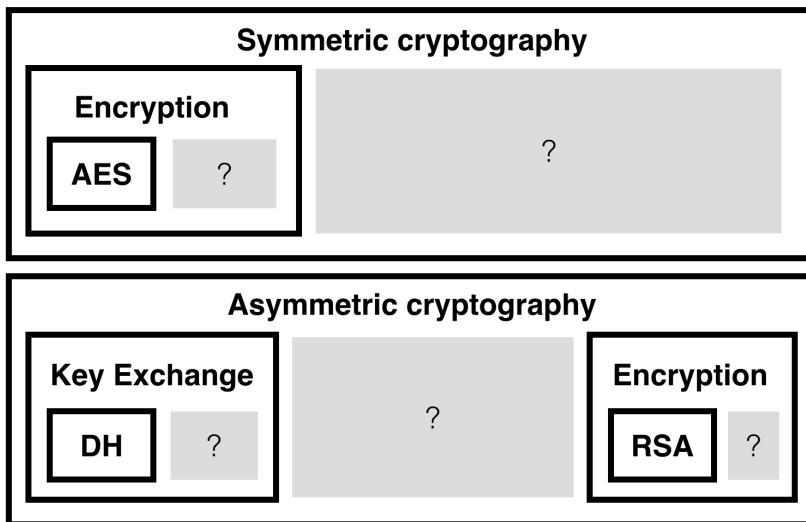


Figure 1.14 The cryptographic algorithms we have learned so far. Two large class of cryptographic algorithms are symmetric cryptography (with symmetric encryption) and asymmetric cryptography (with key exchanges and asymmetric encryption).

DIGITAL SIGNATURES

We've seen that RSA provides an **asymmetric encryption** algorithm, but as we've mentioned earlier, it also provides a **digital signature** algorithm.

NOTE

The word "RSA" is often used to mean either RSA encryption or RSA signatures depending on context, which is always a great source of confusion. You've been warned.

The invention of this digital signature cryptographic primitive has been of immense help to set up trust between Alice(s) and Bob(s). It is very similar to real signatures, you know, the one you are required to sign on a contract when you're trying to rent an apartment for example. "What if they forge my signature?" you may ask, and indeed real signatures don't provide much security in the real-world. On the other hand, cryptographic signatures can be used in the same kind of way, but provide a cryptographic certificate with your name on it. Your cryptographic signature is **unforgeable**, and can easily be verified by others. Pretty useful compared to the archaic signatures you used to write on checks!

In figure 1.15, we can imagine a protocol where Alice wants to show David that she trusts Bob. This is a typical example of how to establish trust in a multi-participant setting, and how asymmetric cryptography can help. By signing a piece of paper containing "*I, Alice, trust Bob*", Alice can take a stance and notify David that Bob is to be trusted. If David already trusts Alice and her signature algorithm, then he can choose to trust Bob in return.

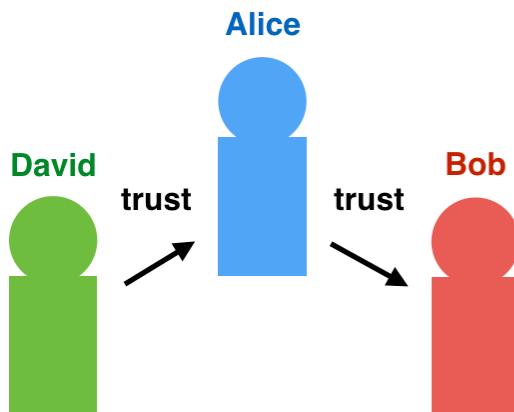


Figure 1.15 David already trusts Alice. Since Alice trusts Bob, can David safely trust Bob as well?

In more details, Alice can use the **RSA signature scheme** and her **private key** to **sign** the message "*I, Alice, trust Bob*". This generates a signature that should look like random noise (see figure 1.16).

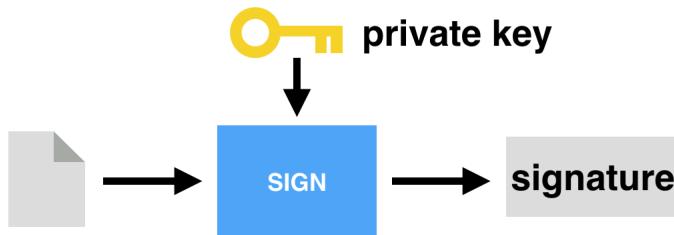


Figure 1.16 To sign a message, Alice uses her private key and generate a signature.

Anyone can then **verify the signature**, by combining:

- Alice's **public key**.
- The **message** that was signed.
- The **signature**.

The result is either `true` (the signature is valid) or `false` (the signature is invalid) as can be seen in figure 1.17.

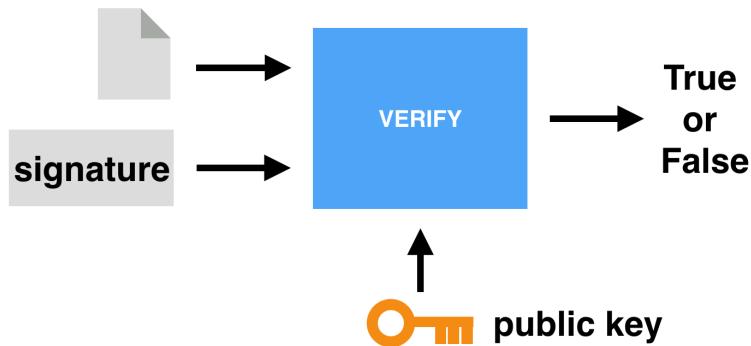


Figure 1.17 To verify a signature from Alice, one also needs the message signed and Alice's public key. The result is either validating the signature, or disproving it.

We have now learned about three different asymmetric primitives:

- Key exchange with Diffie-Hellman.
- Asymmetric encryption.
- Digital signatures with RSA.

These three cryptographic algorithms are the most known and commonly used primitives in asymmetric cryptography. It might not be totally obvious how these can help to solve real-world problems, but rest assured, they are used every day by many applications to secure things around them.

It is time to complete our picture with all the cryptographic algorithms we've learned about so far. See figure 1.18.

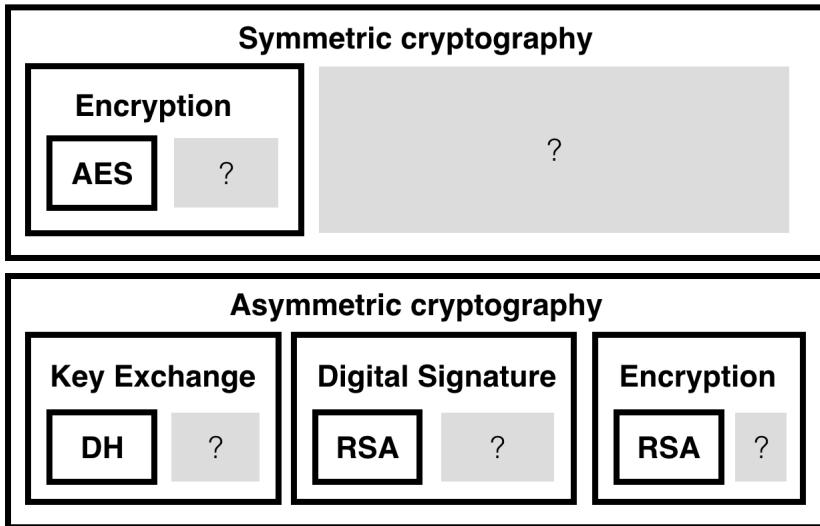


Figure 1.18 The symmetric and asymmetric algorithms we have learned so far.

1.1.4 A Map of Cryptography

In the previous section, we have surveyed two large classes of algorithms:

- **Symmetric cryptography** (or secret-key cryptography). A single secret is used. If several participants are aware of the secret, it is called a shared secret.
- **Asymmetric cryptography** (or public-key cryptography). Participants have an asymmetrical view of the secrets. For example, some will have knowledge of a public key, while some will have knowledge of both a public and private key.

Symmetric and asymmetric cryptography are not the only two categories of primitives in cryptography, and it's quite hard to classify the different subfields. But yet, as you will realize, a large part of our book is about (and makes use of) symmetric and asymmetric primitives. This is because a large part of what is useful in cryptography, nowadays, is contained in these subfields.

Another way of dividing cryptography can be:

- **Mathematic-based constructions.** These are constructions that rely on mathematical problems like factoring numbers. (The RSA algorithm for digital signatures and asymmetric encryption is an example of such a construction.)
- **Heuristic-based constructions.** These are the constructions that rely on observations and statistical analysis by cryptanalysts. (AES for symmetric encryption is an example of such a construction.)

There is also a speed component to this categorization, as mathematic-based constructions are often much slower than heuristic-based constructions. To give you an idea, symmetric constructions are most often based on heuristics (what seems to be working), while most asymmetric constructions are based on mathematical problems (what is thought to be hard).

Furthermore, even these subfields can be further divided. Permutation-based cryptography, for

example, is the set of constructions built on top of permutations. Lattice-based cryptography is, as you can guess, the field of cryptography based on lattices (a type of vector space in mathematics). It is thus hard for us to rigorously categorize all of what cryptography has to offer. Indeed, every book or course on the subject gives different definitions and classifications.

In the end, these distinctions are not too useful for us, as we will see most of the cryptographic primitives as unique tools that make unique **security claims**. Many of these tools can in turn be used as building blocks to create protocols. It is thus essential to understand how each of these tools work, and what kind of security claims they provide, in order to understand how they provide security in the protocols around us. For this reason, the first part of this book will go through the most useful cryptographic primitives and their security properties.

1.1.5 Two Goals: Confidentiality and Authentication

The definition of cryptography used to be simple (Alice and Bob want to exchange secret messages). It isn't anymore. What cryptography is nowadays, is quite complex to describe, and has grown organically around discoveries, breakthroughs and practical needs. At the end of the day, **cryptography is what helps to augment a protocol, in order to make it work in adversarial settings**. To understand exactly how cryptography can help, the set of goals that these protocols want to achieve is what matters to us. That's the useful part. Again, it is hard to classify these goals, but most of what we will talk about in this book provides the following security properties:

- **Confidentiality.** It is about masking and protecting some information from the wrong eyes. For example, encryption masks the messages in transit.
- **Authentication.** It is about identifying who we are talking to. For example, this can be helpful in making sure that messages we receive indeed come from Alice.

NOTE

This is a heavy simplification of what cryptography can provide. In most cases, the details are in the security claims of the primitives. Depending on how a cryptographic primitive is used in a protocol, it will achieve different security properties.

Throughout this book, we will learn new cryptographic primitives and how they can be combined to expose security properties like confidentiality and authentication. For now, appreciate the fact that cryptography is about providing insurances to a protocol in adversarial settings. While the "adversaries" are not clearly defined, we can imagine that they are the ones who attempt to break your protocol: a participant, an observer, a man-in-the-middle. They reflect what a real life adversary could be. Because eventually, cryptography is a practical field made to defend against bad actors in flesh and bones and bits.

1.2 Real-world Cryptography

This book focuses on real-world cryptography. Thus, a large amount of topics are omitted if they aren't useful to us. We hereby define what we mean by non-real-world (theoretical) cryptography and real-world cryptography, and survey what real-world cryptography has to offer.

1.2.1 Theoretical Cryptography Versus real-world Cryptography

Theoretical cryptography is what cryptographers and cryptanalysts do. They are mostly from the academia, working in universities, but sometimes from the industry or in specific departments of the government. They research everything and anything in cryptography. Results are shared internationally through publications and presentations in journals and conferences. Yet, not everything they do is obviously useful or practical. Often, no "proof of concept" or code is released, and it wouldn't make sense anyway as no computer exists that could run their research. Having said that, it does sometimes become so useful and practical that something makes its way to the other side (real-world cryptography).

We will avoid digging too deep into this territory, as it would take us too much time. We will still take a peek at what is brewing there in the last chapters of this book. Be prepared to be amazed, as you might catch a glance of the real-world cryptography of tomorrow.

real-world cryptography is the foundation of the security you find in all applications around you. Although it often seems like it's not there, almost transparent, it is here when you log into your bank account on the internet, it is with you when you message your friends, it helps protect you when you lose your phone. It is ubiquitous, because unfortunately attackers are everywhere and actively do try to observe and harm our systems. Practitioners are usually from the industry, but will sometimes vet algorithms and design protocols with the help of the academia community. Results are often shared through conferences, blog posts, and open-source software.

real-world cryptography usually cares deeply about real-world considerations: what is the exact level of security provided by an algorithm? How long does it take to run the algorithm? What is the size of the inputs and outputs required by the primitive? real-world cryptography is, as you might have guessed, the subject of this book.

1.2.2 From Theoretical to Practical

One might wonder, how do people decide what algorithms to use in the real-world? The answer is complicated. Before a primitive is massively adopted by developers, it usually follows some amount of standardization:

Standardization from national institutions. A large number of cryptographic algorithms in use in the past or today came from standards created by government bodies. AES for example, was first standardized by the NIST.

While many of these standards were success stories, there is unfortunately a lot to say about failures. In 2013, it was discovered following revelations from Edward Snowden, that the NSA had purposefully and successfully pushed for the inclusion of backdoors algorithms in standards (like the Dual EC random number generators,¹ which included a secret switch that allowed the NSA and only the NSA to predict your secrets). Following this, the cryptographic community has lost a lot of confidence in standards and suggestions coming from government bodies. Recently, in 2019, it was found that the russian standards GOST had been victim of the same treatment.²

Cryptographers have long suspected that the agency planted vulnerabilities in a standard adopted in 2006 by the National Institute of Standards and Technology and later by the International Organization for Standardization, which has 163 countries as members.

Classified N.S.A. memos appear to confirm that the fatal weakness, discovered by two Microsoft cryptographers in 2007, was engineered by the agency. The N.S.A. wrote the standard and aggressively pushed it on the international group, privately calling the effort “a challenge in finesse.”

– *New York Times about Dual EC following Edward Snowden's revelations 2013*

Standardization from open contests. The process cryptographers love the most is an open competition! A lot of the important algorithms in use today come from such events. AES is again such an example! To participate, researchers from all over the world gather to form research groups and propose their algorithms to compete. After rounds of analysis and help from cryptanalysts (which can take years) the list is reduced to a few candidates (sometimes only one) who then become part of a standard.

Enforcement from regulations. Some industries are forced to use specific algorithms, which makes cryptographic algorithms popular by default. For example, the Federal Information Processing Standards (FIPS) mandates what cryptographic algorithms can be used by companies that deal with the US government, another example is the Payment Card Industry Data Security Standard (PCI-DSS) which does the same thing with companies dealing with credit card numbers.

Organic standardization. It sometimes happens that an algorithm or protocol becomes popular due to the popularity of a research white paper, or a webpage, or from being spread from word of mouth. At some point, groups of volunteers decide that it is a good idea to write a **Request For Comment (RFC)** for it. An RFC is a free document that follows guidelines from the Internet Engineering Task Force (IETF), an organization behind many standards on the internet (like TCP, UDP, TLS, and so on). Many of the algorithms you will learn about in this book are standardized via RFCs.

To reinforce that we do not vote, we have also adopted the tradition of "humming": When, for example, we have face-to-face meetings and the chair of the working group wants to get a "sense of the room", instead of a show of hands, sometimes the chair will ask for each side to hum on a particular question, either "for" or "against".

– RFC 7282: *On Consensus and Humming in the IETF 2014*

Standardization from alliances. Groups of private companies sometime create alliances in order to agree on standards to use. This is what happens with your Wi-Fi, which works on protocols standardized by the Wi-Fi Alliance. This way of standardizing algorithms and protocols is often criticized because of the opaque process, the quality of the protocols produced (Wi-Fi has had a lot of vulnerabilities in its protocols) and the sometimes paid-access to the standards.

Standardization from companies. The most popular examples are the standards produced by the RSA Security LLC company (funded by the creators of the RSA algorithm). The Public Key Cryptography Standards (PKCS) are a series of 15 documents published in order to legitimize algorithms and techniques the company was using at the time. Nowadays, this is pretty rare and a lot of companies will go through the IETF to standardize their protocols or algorithms via an RFC instead of a custom document. On the other hand, companies sometimes slow down the adoption process by patenting cryptographic algorithms. The most popular example is probably Schnorr signatures, which were the first contender to become the most popular signature scheme until Schnorr himself patented the algorithm in 1989. This led to the NIST standardizing a poorer algorithm (DSA), which became (at the time) the most popular signature scheme. The patent over Schnorr signatures has expired in 2008, and the algorithm has started re-gaining popularity since then.

1.3 A Word of Warning

Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.

– Bruce Schneier 1998

I must warn you, the art of cryptography is difficult to master and it would be unwise to build complex cryptographic protocols once you're done with this book.

This journey should enlighten you, show you what is possible, and show you how things work, but it will not make you a master of cryptography. This book is not the holy grail. Indeed, the last pages of this book will have you go through the most important lesson: do not go alone on a real adventure. Dragons can kill, and you need some support to accompany you in order to defeat them. In other words, cryptography is complicated, and this book alone does not permit you to abuse what you learn. To build complex systems, experts who have studied their trade for years

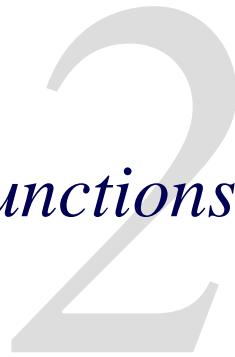
are required. Instead, what you will learn is to recognize when cryptography should be used or if something seems fishy, what cryptographic primitives and protocols are available to solve the issues you're facing, and how all these cryptographic algorithms work under the surface.

Now that you've been warned, go to the next chapter.

1.4 Summary

- A protocol is a step by step recipe where multiple participants attempt to achieve something, like exchanging confidential messages.
- Cryptography is about augmenting protocols to secure them in adversarial settings. It often requires secrets.
- Two large categories of cryptographic primitives stand out from the rest: symmetric and asymmetric cryptography.
- A cryptographic primitive is a type of cryptographic algorithm. For example, symmetric encryption is a cryptographic primitive, while AES is a specific symmetric encryption algorithm.
- Symmetric cryptography uses a single key, as we've seen with symmetric encryption.
- Asymmetric cryptography (or public-key cryptography) makes use of different keys (private and public) as we've seen with key exchanges, asymmetric encryption, and digital signatures.
- Cryptographic properties are hard to classify, but they often aim to provide one of these two properties: authentication or confidentiality.
- real-world cryptography matters because it is ubiquitous in technological applications, while theoretical cryptography is often less useful in practice.
- Most of the cryptographic primitives contained in this book were agreed on after long standardization processes.
- Cryptography is complicated and there are many dangers in implementing or using cryptographic primitives.

Hash functions



This chapter covers

- Hash functions and their security properties.
- The wildly adopted hash functions in use today.
- What other types of hashing there exist.

I have talked about cryptographic primitives in the previous chapter, they are constructions that achieve specific security properties and form the building blocks of cryptography. In the first part of this book you will learn about several important ones. The very first one I'll talk about is the **hash function**. Informally, it is a tool that when given an input will produce a unique identifier tied to that input. A hash function is rarely used on its own but can be found everywhere in cryptography. For example, we will rarely see digital signatures not making use of hash functions (and some signature schemes even go as far as being built solely using hash functions).

2.1 What Is a Hash Function?

In front of you a download button is taking a good chunk of the page. You can read the letters *DOWNLOAD*, and clicking on them seems to redirect you to a different website containing the file. Below it, lies a long string of unintelligible letters:

```
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

It is followed by what looks like an acronym of some sort: `sha256sum`. Sounds familiar? You've probably downloaded something in your past life that was also accompanied with such an odd string.



Figure 2.1 A webpage linking to an external website containing a file. The external website cannot modify the content of the file, because the first page provides a hash or digest of the file, which ensures the integrity over the downloaded file.

If you've ever wondered what was to be done with it:

1. Click on the button to download the file.
2. Use the SHA-256 algorithm to *hash* the downloaded file.
3. Compare the output (the digest) with the long string displayed on the webpage.

This allows you to verify that you've downloaded the right file.

NOTE

The **output of a hash function** is often named a **digest** or a **hash**. I will use the two words interchangeably throughout this book. Note that others might call it a checksum or a sum, which I will avoid as it is primarily used by non-cryptographic hash functions and could lead to more confusion. Just keep that in mind when different codebases or documents use different terms.

To try hashing something, you can use the popular **OpenSSL** library. It offers a multi-purpose Command Line Interface that comes by default in a number of systems, including MacOS. For example, this can be done by opening the terminal and writing the following line:

```
$ openssl dgst -sha256 downloaded_file  
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

We used SHA-256 (a hash function) to transform the input (the file) into a unique identifier (the digest). What do these extra steps provide? **Integrity** and **Authenticity**. It tells you that what you downloaded is indeed the file you were meant to download.

All of this works thanks to a security property of the hash function called **second pre-image resistance**. This math-inspired term means that from the long output of the hash function `f63e...`, you cannot find another file that will "hash" to the same output `f63e...`. In practice it means that this digest is closely tied to the file you're downloading, and that no attacker should be able to fool you by giving you a different file.

NOTE

By the way, the long output string `f63e...` represents binary data displayed in **hexadecimal** (a base 16 encoding using numbers from `0` to `9` and letters from `a` to `f` to represent several bits of data). We could have displayed the binary data with `0`s and `1`s (base 2) but it would have taken more space. Instead, the hexadecimal encoding allows us to write 2 alphanumeric characters for every 8 bits (1 byte) encountered. It is somewhat readable by humans and takes less space. There are other ways to encode binary data for **human consumption**, but the two most widely used encodings are hexadecimal and **base64**. The larger the base, the less space it takes to display a binary string, but at some point we run out of human-readable characters :)

Note that this long digest is controlled by the owner(s) of the webpage, and it could easily be replaced by anyone who can modify the webpage. (If you are not convinced about this take a moment to think about it.) This means that we need to trust the page that gave us the digest, its owners and the mechanism used to retrieve the page (while we don't need to trust the page that gave us the file we downloaded). In this sense, **the hash function alone does not provide integrity**. The integrity and authenticity of the downloaded file comes from the digest combined with the trusted mechanism which gave us the digest (**HTTPS** in this case, we will talk about **HTTPS** in chapter 9 but for now imagine that it magically allows you to communicate securely with a website).

Back to our hash function, which can be visualized as a black box in figure 2.2 which takes a single input, and gives out a single output.



Figure 2.2 A hash function takes an arbitrary-length input (a file, a message, a video, etc.) and produces a fixed-length output (for example, 256 bits for SHA-256). Hashing the same input will produce the same digest or hash.

The **input** of this function can be of any size. It can even be empty.

The **output** is always of the same length and **deterministic**: it always produces the same result if given the same input. In our example, SHA-256 always provides an output of 256 bits (32 bytes)

which is always encoded as 64 alphanumeric characters in hexadecimal. One major property of a hash function is that one cannot revert the algorithm, meaning that one shouldn't be able to find the input from just the output. We say that hash functions are **one-way**.

To illustrate how a hash function work in practice, we hash different inputs with the SHA-256 hash function using the same OpenSSL Command Line Interface:

```
$ echo -n "hello" | openssl dgst -sha256 ①  
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824  
$ echo -n "hello" | openssl dgst -sha256 ①  
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824  
$ echo -n "hella" | openssl dgst -sha256 ②  
70de66401b1399d79b843521ee726dcecle9a8cb5708ec1520f1f3bb4b1dd984  
$ echo -n "this is a very very very very very very very very very long sentence" | openssl dgst  
③  
009e286e0261a8d0eca95649cf795db3572c515fe9dc7e319ece5af8f133637a
```

- ① Hashing the same input produces the same result.
- ② A tiny change in the input completely changes the output.
- ③ The output is always of the same size, no matter the input size.

In the next section, we will see what are the exact security properties of hash function.

2.2 Security Properties of a Hash Function

Hash functions in applied cryptography are constructions that were commonly defined to provide three specific security properties. This definition has changed over time, as we will see in the next sections. But for now, let's define these three strong foundations that makes a hash function. This is important as you need to understand where hash functions can be useful, and where they will not work.

The first one is **pre-image resistance**. This property ensures that no one should be able to reverse the hash function in order to, given an output, recover the input. In figure 2.3 we illustrate this one-wayness by imagining that our hash function is like a blender making it impossible to recover the ingredients from the produced smoothie.

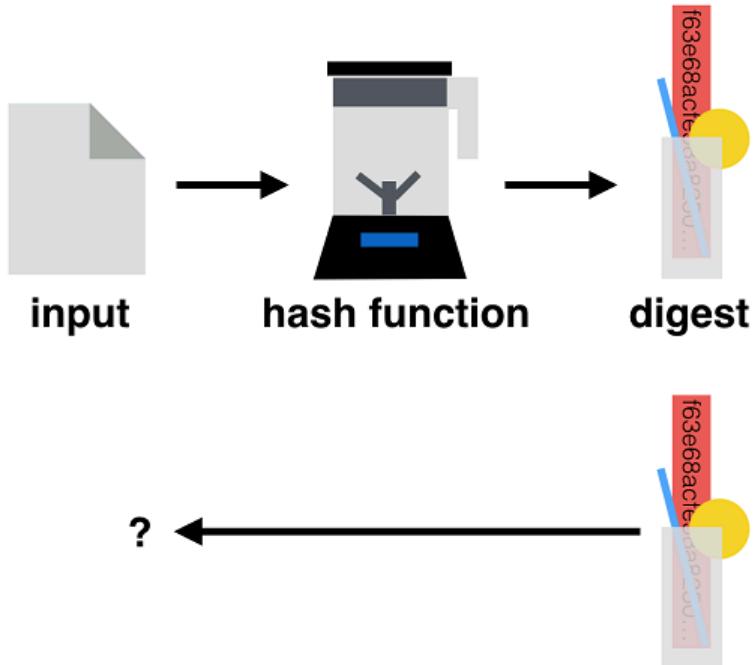


Figure 2.3 Given the digest produced by a hash function (represented as a blender here), it is impossible (or technically so hard we assume it should never happen) to reverse it and find the original input used. This security property is called pre-image resistance.

WARNING Is this true if your input is small? let's say that it's either "oui" or "non", then it is easy for someone to hash all the possible 3-letter words and find out what the input was. What if your input space is small? Meaning that you always hash variants of the following sentence "I will be home on Monday at 3am". Here one who can predict this, but does not know exactly the day of the week or the hour, can still hash all possible sentences until it produces the correct output. As such, this first pre-image security property has an obvious caveat: **you can't hide something that is too small or is predictable.**

The second one is **second pre-image resistance**. We've already seen this security property when we wanted to protect the integrity of a file. The property says the following: if I give you an input and the digest it hashes to, you should not be able to find a **different input** that hashes to the same digest. This is illustrated in figure 2.4.

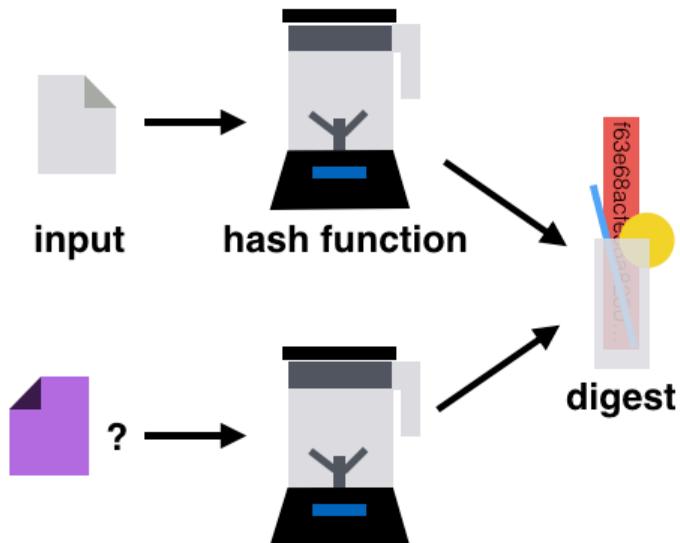


Figure 2.4 Considering an input and its associated digest, one should never be able to find a different input that hashes to the same output. This security property is called second pre-image resistance.

Note that we do not control the first input, this emphasis is important to understand the next security property.

Finally, the third property is **collision-resistance**. It guarantees that no one should be able to produce two different inputs that hash to the same output (as seen in figure 2.5). Here an attacker can choose the two inputs, unlike the previous property that fixes one of the inputs.

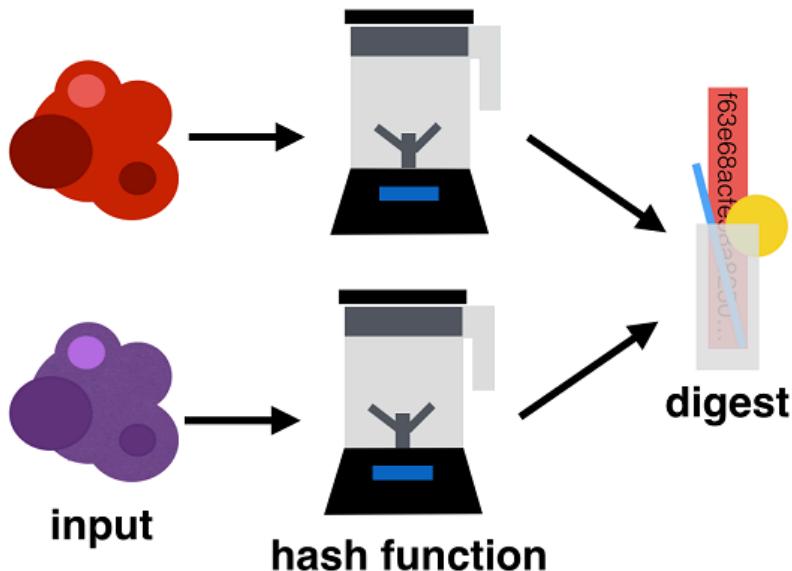


Figure 2.5 One should never be able to find two inputs (here represented on the left as two random blobs of data) that hash to the same output value (on the right). This security property is called collision resistance.

People often confuse collision resistance and second pre-image resistance. Take a moment to

understand the differences.

NOTE

In addition, hash functions are usually designed so that their digests are **unpredictable** and **random**. This is useful because one cannot always prove a protocol to be secure thanks to one of the security properties of a hash function we've talked about (like collision resistance for example). Many protocols are instead proven in the **random oracle model** where a fictive and ideal participant called a **random oracle** is used. In this type of protocol, one can send any inputs as requests to that random oracle which is said to return completely random outputs in response, and like a hash functions, giving it the same input twice will return the same output twice. Proofs in this model are sometimes controversial, as we don't know for sure if we can replace these **random oracles** with real hash functions in practice. Yet, many legitimate protocols are proven secure using this method where hash functions are seen as more ideal than they probably are.

2.3 Security Considerations for Hash Functions

We've seen three security properties of a hash function:

- pre-image resistance
- second pre-image resistance
- collision-resistance

These security properties are often meaningless on their own, and it all depends on how you make use of the hash function. Nonetheless, it is important that we understand some limitations here before we look at some of the real-world hash functions.

First, these security properties assume that you are reasonably using the hash function. Imagine that I either hash the word "yes" or the word "no" and I then publish the digest. If you have some idea of what I was doing, you can simply hash both of the words and compare the result with what I gave you. Since there are no secrets involved, and since the hashing algorithm we've used is public, you are free to do that. And indeed, one could think this would break the pre-image resistance of the hash function, but we'll argue that your input was not "random" enough. Furthermore, since a hash function accepts an arbitrary-length input and always produce an output of the same length, there are also an infinite number of inputs that hash to the same output.

Second, the **size of the parameters** do matter. This is not a peculiarity of hash functions by any mean, all cryptographic algorithms must care about the size of their parameters in practice. Let's imagine the following extreme example, we have a hash function that produces outputs of length 2 bits in a uniformly random fashion (meaning that it will output 00 25% of the time, 01 25% of

the time, etc.) You're not going to have to do too much work to produce a collision: after hashing a few random input strings you should be able to find two that hash to the same output. For this reason, there is a minimum output size that a hash function must produce in practice: 256 bits (or 32 bytes). With this large an output, collisions should be out of reach unless a breakthrough happens in computing.

How was this number obtained? In real-world cryptography, algorithms aim for a minimum of **128 bits of security**. It means that an attacker who wants to break an algorithm (providing 128-bit security) would have to perform around 2^{128} operations (for example, trying all the possible input strings of length 128-bit would take 2^{128} operations). For a hash function to provide all three security properties we mentioned earlier, it needs to provide at least 128 bits of security against all three attacks. The easiest attack is usually to find collisions, due to the **birthday bound**.

NOTE

The **birthday bound** takes its roots from probability theory, in which the birthday problem reveals some unintuitive results: how many people do you need in a room so that with at least 50% chance two people will share the same birthday (that's a collision). It turns out that 23 people taken at random are enough to reach these odds! Weird right? In practice, when we are randomly generating strings from a space of 2^N possibilities, you can expect someone to find a collision with 50% chance after having generated approximately $2^{N/2}$ strings.

If our hash function generates random outputs of 256 bits, the space of all outputs is of size 2^{256} . This mean that collisions can be found with good probability after generating 2^{128} digests (due to the birthday bound). This is the number we're aiming for, and this is why hash functions at a minimum must provide 256-bit outputs.

Certain constraints sometimes push developers to reduce the size of a digest by truncating it (removing some of its bytes). In theory this is possible, but can greatly reduce security. In order to achieve 128-bit security at a minimum, a digest must not be truncated under:

- 256-bit for collision-resistance.
- 128-bit for pre-image and second pre-image resistance.

This means that depending on what property one relies on, the output of a hash function could be truncated to obtain a shorter digest.

2.4 Hash Functions in practice

As we've said earlier, in practice hash functions are rarely used alone. They are most often combined with other elements to either create a cryptographic primitive, or a cryptographic protocol. We will have many examples of using hash functions to build more complex objects in this book, but here are a few different ways hash functions have been used in the real-world:

Commitments. Imagine that you know that a stock in the market will increase in value and reach \$50 in the coming month, but you really can't tell your friends about it (for some legal reason perhaps). You still want to be able to tell your friends that you knew about it, after the fact. Because you're smug (don't deny it). What you can do is to commit to a sentence like "*stock X will reach \$50 next month*". To do this, hash the sentence and give your friends the output. A month later, reveal the sentence. Your friends will be able to hash the sentence to observe that indeed, it is producing the same output. Can you tell which security property of the hash function prevents you from revealing a different sentence once you committed to a digest (by sending it to them)? What property prevents your friends from using the digest to recover the message?

Subresource Integrity. It happens (often) that webpages import external JavaScript files. For example, a lot of websites will use Content Delivery Networks (CDNs) to import javascript libraries or web framework related files in their pages. These CDNs are placed in strategic locations in order to quickly deliver these files to visitors of the pages. Yet, if the CDN goes rogue, and decides to serve malicious javascript files, this could be a real issue. To counter this, webpages can use a feature called **subresource integrity** that allows the inclusion of a digest in the import tag:

```
<script src="https://code.jquery.com/jquery-2.1.4.min.js"
       integrity="sha256-8WqyJLuWKRBVhxXIL1jBDD7SDxU936oZkCnxQbWwJVw=></script>
```

This is exactly the same scenario we talked about in the introduction of this chapter. Once the javascript file is retrieved, the browser will hash it (using SHA-256) and verify that it corresponds to the digest that was hardcoded in the page. If it checks out, the javascript file will get executed as its **integrity** has been verified.

Blockchains. Cryptocurrencies like Bitcoin keep track of all financial transactions (since the beginning of the currency) in a large digital ledger that is shared across all users. The pages of that ledger (containing the transactions) are called blocks, and each block authenticates the previous one by hashing it and carrying the digest. Hence, if one has the most current block, one can retrieve the previous block and verify if it hashes to the digest contained in the most current block. By doing this recursively, one can verify the entire chain of block (hence the name blockchain).

BitTorrent. The BitTorrent protocol is used by users (called peers) all around the world to share

files among each other directly (what we also call **peer-to-peer**). To distribute a file, it is cut into chunks and each chunk is individually hashed. These hashes are then shared as source of trust to represent the file to be downloaded. BitTorrent has several mechanisms to allow a peer to obtain the different chunks of a file from different peers. At the end, the integrity of the entire file is verified by hashing each downloaded chunks and matching the output to their respective known digests (before re-assembling the file from the chunks). For example, the following "magnet link" represents The Ubuntu Operating System version 19.04. It is a digest (represented in hexadecimal) obtained from hashing metadata about the file, as well as all the chunks' digests.

```
magnet:?xt=urn:btih:b7b0fbab74a85d4ac170662c645982a862826455
```

Tor. The Tor browser's goal is to give individuals the ability to browse the Internet anonymously. Another feature is that one can create hidden webpages, which physical locations are difficult to track. Connections to these pages are secured via a protocol that uses the webpage's public key (we will see more about how that works in chapter 9 on session encryption). For example, *Silk Road* which used to be the eBay of drugs until it got seized by the FBI, was accessible via `silkroad6ownowfk.onion` in the Tor browser. This base32 string actually represented the hash of Silk Road's public key. Thus by knowing the onion address, you can authenticate the public key of the hidden webpage you're visiting, and be sure that you're talking to the right page (and not an impersonator). If this is not clear, don't worry, I'll mention this again in chapter 9.

By the way, there is no way this string represents 256-bit (32-byte) right? How is this secure then according to what I said in section 2.3? Also, can you guess how *Dread Pirate Roberts* (the pseudonym of Silk Road's webmaster) managed to obtain a hash that contains the name of the website?

In all of these examples, a hash function was used to provide **content integrity** or **authenticity** in situations where:

- someone might tamper with the content being hashed
- the hash is **securely communicated** to you

We sometimes also say that we **authenticate** something or someone. It is important to understand that if the hash is not obtained securely, then anyone can replace it with the hash of something else, and thus it does not provide integrity by itself. The next chapter on message authentication code will fix this by introducing secrets. Let's now look at what actual hash function algorithms you can use.

2.5 Standardized Hash Functions

We've mentioned SHA-256 in our example above, which is only one of the hash functions one can use. Before we go ahead and list the recommended hash functions of our time, let's first mention other algorithms that people use in real-world applications that are not considered cryptographic hash functions.

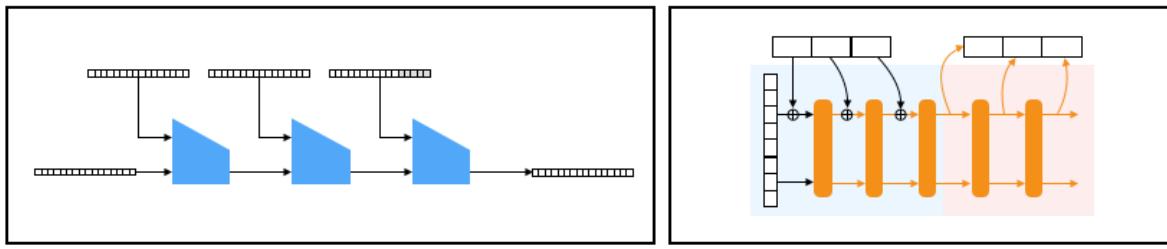
First, functions like *CRC32* are **not** cryptographic hash functions but error-detecting code functions. While they will helpfully detect some simple errors they provide none of the above mentioned security properties and are not to be confused with the hash functions we are talking about here (even though they might share the name sometimes). Their output is usually referred to as checksum.

Second, popular hash functions like *MD5* and *SHA-1* are considered **broken** nowadays. While they both have been the standardized and widely accepted hash functions of the 90's, MD5 and SHA-1 were shown to be broken in 2004 and 2016 respectively when collisions were published by different research teams. These attacks were successful partly because of advances in computing, but mostly because flaws were found in the way the hash functions were designed.

NOTE

Both MD5 and SHA-1 were good hash functions until researchers demonstrated their lack of resistance from collisions. It remains that their (second) pre-image resistance has not been affected by these attacks. This does not matter for us, as we want to only talk about secure algorithms in this book. Nonetheless, you will sometimes still see people using MD5 and SHA-1 in systems that only rely on the pre-image resistance of these algorithms and not on their collision resistance. They will often argue that they cannot upgrade these hash functions to more secure ones because of legacy reasons. As the book is meant to last in time and be a beam of bright light for the future of real-world cryptography, this will be the last time we will mention these hash functions.

The next two sections will introduce SHA-2 and SHA-3 which are the two most widely used hash functions.



SHA-2 (Merkle–Damgård)

SHA-3 (Sponge)

Figure 2.6 SHA-2 and SHA-3. The two most widely adopted hash functions. SHA-2 is based on the Merkle–Damgård construction, while SHA-3 is based on the sponge construction.

2.5.1 The SHA-2 Hash Function

Now that we have seen what hash functions are and had a glimpse at their potential use-cases, it remains to see which hash functions one can use in practice. In the next two sections we introduce two widely accepted hash functions and we also give a high-level explanations of how they work from the inside. The latter part should not provide deeper insights on how to use hash functions, as the black box descriptions we gave should be enough, but nevertheless it is interesting to see how these cryptographic primitives were designed by cryptographers in the first place.

The most widely adopted hash function is the **Secure Hash Algorithm 2 (SHA-2)**. SHA-2 was invented by the NSA and standardized by the NIST in 2001. It was meant to add itself to the aging Secure Hash Algorithm 1 (SHA-1) already standardized by the NIST. SHA-2 provides 4 different versions producing outputs of 224, 256, 384 or 512 bits. Their respective names omit the version of the algorithm: SHA-224, SHA-256, SHA-384, SHA-512. In addition, two other versions provide 224-bit and 256-bit by truncating the result of the larger versions SHA-512/224 and SHA-512/256 respectively.

Below we call each variant of SHA-2 with the OpenSSL CLI, observe that calling the different variants with the same input produce outputs of the specified lengths that are completely different.

```
$ echo -n "hello world" | openssl dgst -sha224
2f05477fc24bb4faef86517156dafdece45b8ad3cf2522a563582b
$ echo -n "hello world" | openssl dgst -sha256
b94d27b9934d3e08a52e52d7da7dabfac484efef37a5380ee9088f7ace2efcde9
$ echo -n "hello world" | openssl dgst -sha384
fdbd8e75a67f29f701a4e040385e2e23986303ea10239211af907fcbb83578b3e417cb71ce646efd0819dd8c088de1bd
$ echo -n "hello world" | openssl dgst -sha512
309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f989dd35bc5ff499670da34255b45b0cf830e81f
```

Nowadays, people mostly use SHA-256 which provides the minimum 128 bits of security needed for our three security properties, while more paranoid applications make use of SHA-512.

Now, let's have a simplified explanation of how SHA-2 works.

It all starts with a special function called a **compression function**. A compression function takes two inputs of some size, and produces one output of one of the input size. Put simply: it takes some data, and returns less data. This is illustrated in figure 2.7.

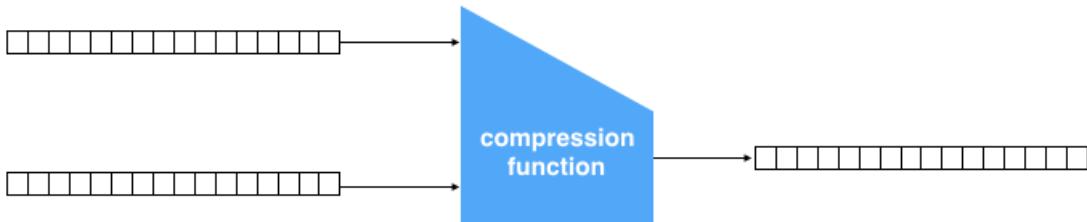


Figure 2.7 A compression function takes two different inputs of size X and Y (here both of 16 bytes) and returns an output of size either X or Y.

NOTE

While there are different ways of building a compression function, SHA-2 uses the **Davies–Meyer method** (see figure 2.8) which relies on a block cipher (a cipher that can encrypt a fixed-size block of data). I've mentioned the AES block cipher in chapter 1, but you haven't yet learned about them. For now accept the compression function as a blackbox until you read chapter 4 on authenticated encryption.

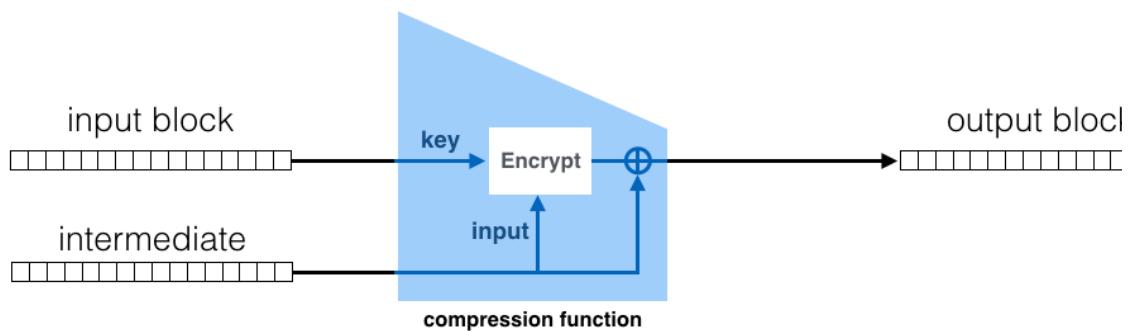


Figure 2.8 An illustration of a compression function built via the Davies–Meyer construction. The compression function's first input (the input block) is used as key block cipher. The second input (the intermediate value) is used as input to be encrypted by the block cipher, it is then used again by XORing () itself with the output of the block cipher.

SHA-2 is a **Merkle–Damgård** construction, which is an algorithm (invented by Ralph Merkle and Ivan Damgård independently) that hashes a message by iteratively calling such a compression function. Specifically, it works by going through the following two steps:

1. It applies a **padding** to the input we want to hash, then **cuts the input into blocks** that can fit into the compression function. Padding means to append specific bytes to the input in order to

make its length a multiple of some block size. Cutting the padded input into chunks of the same block size allows us to fit these in the first argument of the compression function. For example, SHA-256 has a block size of 512-bit. This step is illustrated in figure 2.9.

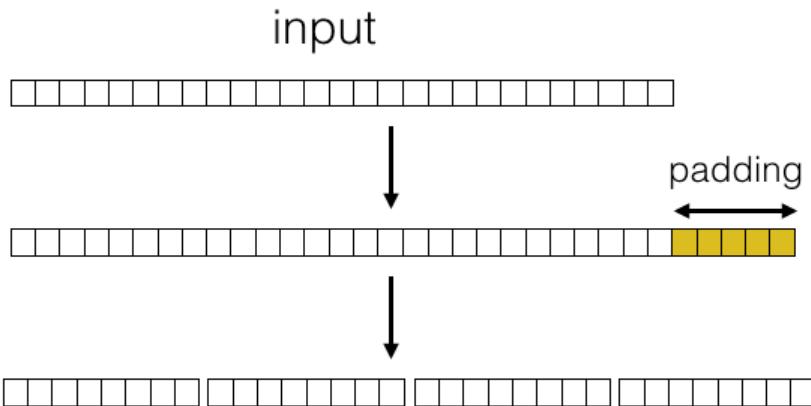


Figure 2.9 The first step of the Merkle–Damgård construction is to add some padding to the input message. After this step, the input should be of length a multiple of the input size of the compression function in use (for example 8 bytes). To do this we add 5 bytes of padding at the end to make it 32 bytes, we then cut the messages into 4 blocks of 8 bytes.

2. It iteratively applies the compression function to the message blocks, using the previous output of the compression function as second argument to the compression function. The final output is the digest. This is illustrated in figure 2.10.

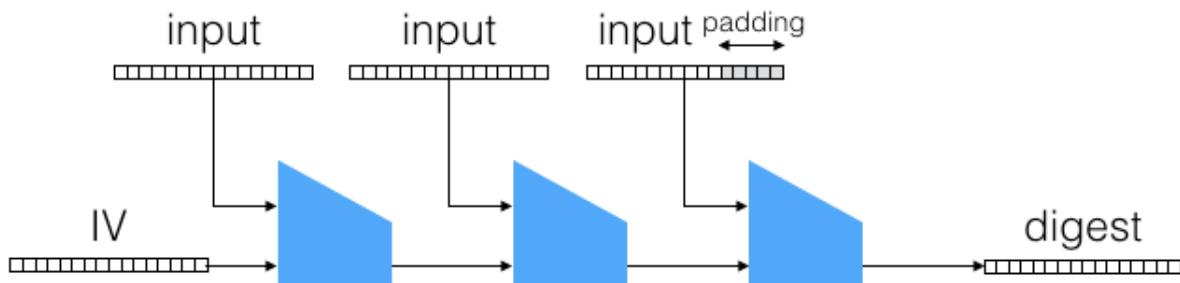


Figure 2.10 The Merkle–Damgård construction. It iteratively applies a compression function to each block of the input to be hashed, and the output of the previous compression function (except for the first step which takes an initial value IV). The final call to the compression function directly returns the digest.

And this is how SHA-2 works: by iteratively calling its compression function on fragments of the input until everything is processed into a final digest.

NOTE

The Merkle–Damgård construction has been proven to be collision resistant if the compression function itself is. Thus, the security of the *arbitrary-length input* hash function is reduced to the security of a *fixed-sized* compression function, which is easier to design and analyze. Therein lies the ingenuity of the Merkle–Damgård construction.

In the beginning, the second argument to the compression function is chosen to be a **nothing-up-my-sleeve** value. Specifically, SHA-256 uses the square roots of the first prime numbers to derive this value. A nothing-up-my-sleeve value is meant to convince the cryptographic community that it was not chosen to make the hash function weaker (for example, in order to create a backdoor). This is a popular concept in cryptography.

WARNING

Note that while SHA-2 is a perfectly fine hash function to use, it is not suitable for hashing secrets. This is because of a downside of the Merkle–Damgård construction which makes SHA-2 vulnerable to an attack called a **length-extension attack** if used to hash secrets. We will talk about this in more details in the next chapter.

2.5.2 The SHA-3 Hash Function

As I've mentioned earlier, both the MD5 and SHA-1 hash functions have been broken somewhat recently. These two functions made use of the same Merkle–Damgård construction I've described in the previous section. Because of this, and the fact that SHA-2 is vulnerable to length-extension attacks, the NIST decided in 2007 to organize an open competition for a new standard: **SHA-3**. This section will introduce the newer standard and will attempt to give a high-level explanation of its inner-workings.

In 2007, 64 different candidates from different international research teams entered the SHA-3 contest. 5 years later Keccak, one of the submissions, was nominated as the winner and took the name SHA-3. In 2015, SHA-3 was standardized in FIPS Publication 202.³

SHA-3 observes the three previous security properties we've talked about and provides as much security as the SHA-2 variants. In addition, it is not vulnerable to length-extension attacks and can be used to hash secrets. For this reason, it is now the recommended hash function to use. It offers the same variants as SHA-2, this time indicating the full name SHA-3 in their named variants: SHA-3-224, SHA-3-256, SHA-3-384, and SHA-3-512. Thus, similarly to SHA-2, SHA-3-256 provides 256 bits of output for example.

Let me now take a few pages to explain how SHA-3 works.

SHA-3 is a cryptographic algorithm built on top of a **permutation**. The easiest way to

understand a permutation is to imagine the following: you have a set of elements on the left, and the same set of elements on the right. Now trace arrows going from each elements on the left to the right. Each element can only have one arrow starting from and terminating to them. You now have one permutation. This is illustrated in figure 2.11. By definition, any permutation is also **reversible**, meaning that from the output we can find the input back.

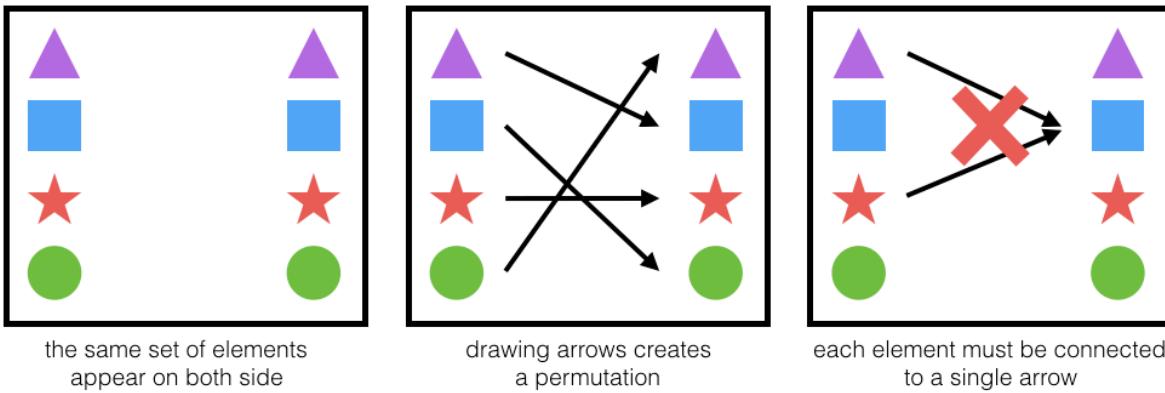


Figure 2.11 An example permutation acting on 4 different shapes. The permutation described by the arrows in the middle picture can be used to transform a given shape.

SHA-3 is built with a **sponge construction**, a very different construction from Merkle–Damgård that was invented as part of the SHA-3 competition. It is based on a particular permutation called **keccak-f** that takes an input and returns an output of the same size.

NOTE

We won't explain how keccak-f was designed, but you will get an idea in chapter 4 on authenticated encryption since it resembles the AES algorithm a lot (with the exception that it doesn't have a key). This is no accident, as one of the co-inventor of AES was also one of the inventor of SHA-3.

In the next few pages I will use an 8-bit permutation to illustrate how the sponge construction works. Since the permutation is set in stone, you can imagine that figure 2.12 is a good illustration of the mapping created by this permutation on all possible 8-bit inputs. Compared to our previous explanation of a permutation, you can imagine that each possible 8-bit string is what we represented as different shapes (000... is a triangle, 100... is a square, and so on).

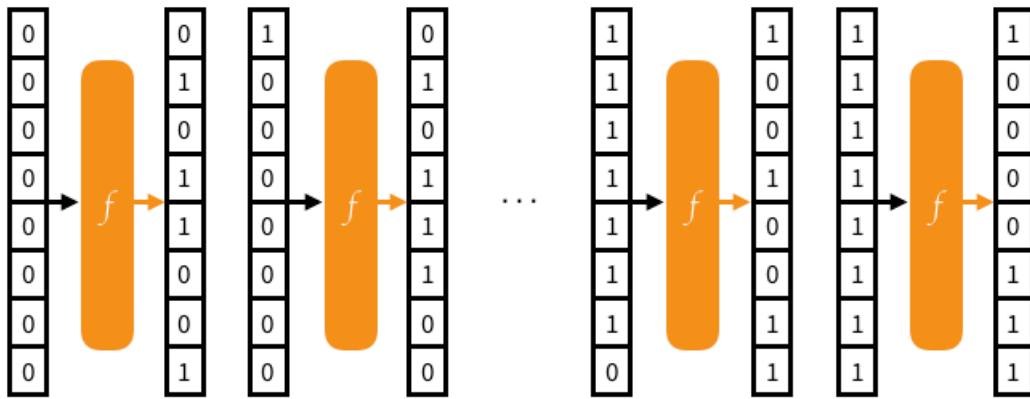


Figure 2.12 A sponge construction makes use of a specified permutation f . By operating on an input, our example permutation creates a mapping between all possible input of 8 bits and all possible output of 8 bits.

To use a permutation in our sponge construction, we also need to define an arbitrary division of the input and the output into a **rate** and a **capacity**. It's a bit weird but stick with it. I illustrate this in figure 2.13.

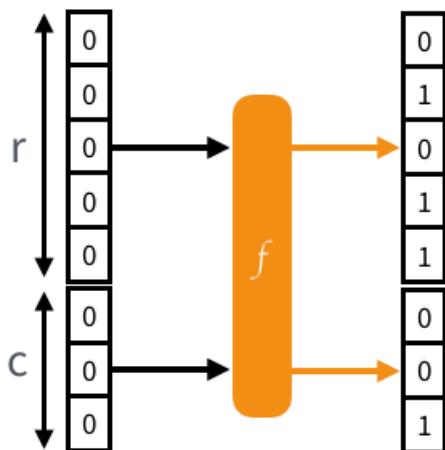


Figure 2.13 The permutation f randomizes an input of size 8 bits into an output of the same size. In a sponge construction, this permutation's input and output are divided into two parts: the **rate** (of size r) and the **capacity** (of size c).

Where we set the limit between the rate and the capacity is arbitrary. Different versions of SHA-3 will use different parameters. We informally point out that the capacity is to be treated like a secret, and the larger it is the more secure the sponge construction will be.

NOTE

To understand what follows, you need to understand the **XOR** (Exclusive OR) operation. XOR is a bitwise operation, meaning that it operates on bits. I illustrate how it works in figure 2.14. XOR is ubiquitous in cryptography, so make you sure you remember it.

1	\oplus	0	=	1
1	\oplus	1	=	0
0	\oplus	1	=	1
0	\oplus	0	=	0

Figure 2.14 Exclusive OR or XOR (often denoted \oplus) operates on two bits. It is similar to the OR operation except for the case where both operands are 1s.

Now, like all good hash functions, we need to be able to hash something right? Otherwise it's a bit useless. To do that, we simply XOR (\oplus) the input with the rate of the permutation's input. In the beginning, this is just a bunch of 0s. As we pointed out earlier, the capacity is to be treated like a secret so we won't XOR anything with it. I illustrate this in figure 2.15.

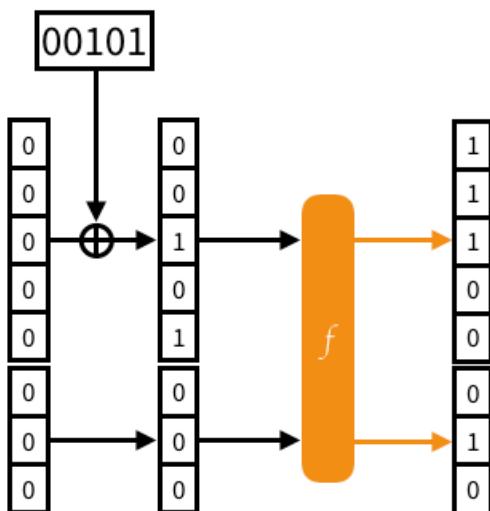


Figure 2.15 To absorb the 5 bits of input 00101, a sponge construction with a rate of 5 bits can simply XOR the 5 bits with the rate (which is initialized to 0s). The permutation then randomizes the state.

The output obtained should now look random (although we can trivially find what the input is since a permutation is reversible by definition).

What if we want to ingest a larger input? Well, similarly to what we did with SHA-2:

1. pad your input if necessary (we will omit explanations of the padding), then divide your input into blocks of the rate size
2. iteratively call the permutation while XORing each blocks with the input of a permutation while permuting the **state** (the intermediate value output by the last operation) after each block has been XORed

I illustrate this in figure 2.16.

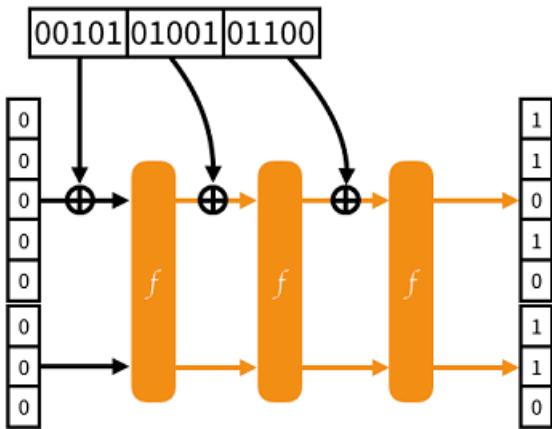


Figure 2.16 In order to absorb inputs larger than the rate size, a sponge construction will iteratively XOR input blocks with the rate and permute the result.

So far so good, but we still haven't produced a digest. To do this, we can simply use the rate of the last state of the sponge (again, we are not touching the capacity). To obtain a longer digest, we can continue to permute and read from the rate part of the state as illustrated in figure 2.17.

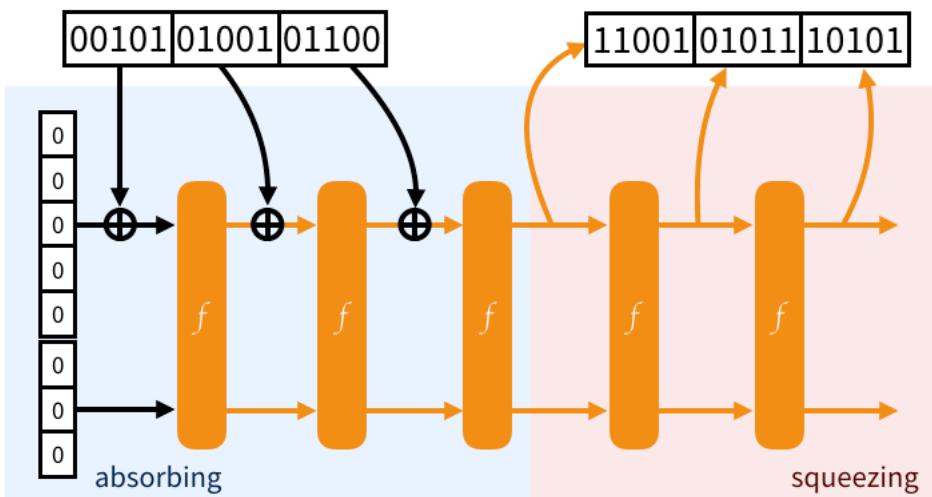


Figure 2.17 To obtain a digest with the sponge construction, one iteratively permutes the state and retrieves as much rate (the upper part of the state) as needed.

And this is how SHA-3 works. Since it is a **sponge construction**, ingesting the input is naturally called *absorbing* and creating the digest is called *squeezing*. The sponge is specified with a 1600-bit permutation using different values for r and c depending on the security advertised by the different versions of SHA-3.

NOTE

I've talked about random oracles earlier, an ideal and fictive construction that would return perfectly random response to queries and repeat itself if we query it with the same input. It turns out that the sponge construction has been proven to behave closely to a random oracle, as long as the permutation used by the construction looks random enough. How do we prove such security properties on the permutation? Our best approach is to try to break it, many times, until we get strong confidence in its design (which is what happened during the SHA-3 competition). The fact that SHA-3 can be modeled as a random oracle instantly gives it the security properties we would expect from a hash function.

2.5.3 **SHAKE and cSHAKE, Two eXtendable Output Functions (XOF)**

I've introduced the two major hash function standards: SHA-2 and SHA-3. These are well defined hash functions that take arbitrary-length inputs and produce random-looking and fixed-length outputs. As you will see in later chapters, cryptographic protocols often necessitate this type of primitives but do not want to be constrained by the fixed sizes of a hash function's digest. For this reason, a more versatile primitive called an **eXtendable Output Function** or **XoF** (pronounced "zoff") was introduced by the SHA-3 standard. This section introduces the two standardized XoFs: **SHAKE** and **cSHAKE**.

SHAKE, specified in FIPS 202 along with SHA-3, can be seen as the same black box as a hash function for the distinction that it returns an output of an arbitrary length. It is up to you to choose how long you want it to be. SHAKE is fundamentally the same construction as SHA-3, except that it is faster and permutes as much as you want it to permute in the squeezing phase. Producing outputs of different sizes is very useful, not only to create a digest, but also to create random numbers, derive keys, and so on. I will talk about the different applications of SHAKE again in this book, for now just imagine that SHAKE is like SHA-3 except that it provides an output of any length you might want.

This construction is so useful in cryptography, that one year after SHA-3 was standardized, NIST published Special Publication 800-185⁴ containing a **customizable SHAKE** called **cSHAKE**. cSHAKE is pretty much exactly like SHAKE, except that it also takes a **customization string**. This customization string can be empty, or can be any string you want. Let's first see an example of using cSHAKE in pseudo-code:

```
cSHAKE(input="hello world", output_length=256, custom_string="my_hash")
-> 72444fde79690f0cac19e866d7e6505c
cSHAKE(input="hello world", output_length=256, custom_string="your_hash")
-> 688a49e8c2ale1ab4e78f887c1c73957
```

As you can see, the two digests differ, even though cSHAKE is as deterministic as SHAKE and SHA-3. This is because a different customization string was used. A customization string thus

allows you to customize your XOF! This is very useful in some protocols where, for example, different hash functions must be used. We call this **domain separation**. As a golden rule in cryptography: if the same cryptographic primitive is used in different use-cases, do not use it with the same key (if it takes a key) or/and apply domain separation. You will see more example of domain separation as we survey cryptographic protocols in later chapters.

WARNING

The NIST tends to specify algorithms that take parameters in bits instead of bytes. For this reason, in the example above a length of 256 **bits** was requested. Imagine if you had requested a length of 16 bytes and got 2 bytes instead due to the program thinking you had requested 16 bits of output. This issue is sometimes called a bit attack.⁵

As with everything in cryptography the length of cryptographic strings like keys, parameters and outputs is strongly tied to the security of the system. For this reason, it is important that one does not request too-short outputs from SHAKE or cSHAKE. **One can never go wrong by using an output of 256 bits**, as it will provide 128-bit of security against collision attacks, but real-world cryptography sometimes operate in constrained environments that could use shorter cryptographic values. This can be done if the security of the system is carefully analyzed. For example, if collision resistance does not matter in the protocol making use of the value, pre-image resistance only needs 128-bit long outputs from SHAKE or cSHAKE.

2.5.4 Ambiguous Hashing and TupleHash

In this chapter, I have talked about different types of cryptographic primitives and cryptographic algorithms:

- The SHA-2 hash function, which is vulnerable to length-extension attacks but still widely used when no secrets are being hashed.
- The SHA-3 hash function, which is the recommended hash function nowadays.
- The SHAKE and cSHAKE XOFs, which are more versatile tools than hash functions as they offer a variable output length.

I will talk about one more handy function: **TupleHash**, that is based on cSHAKE and specified in the same standard as cSHAKE. TupleHash is an interesting function that allows one to hash a **tuple** (a list of something). To explain what TupleHash is and why it is useful, let me tell you a story.

A few years ago I was tasked to review a cryptocurrency as part of my work. It included basic features one would expect from a cryptocurrency: accounts, payments, and so on. Transactions between users would contain metadata about who is sending how much to who. It would also include a small fee to compensate the network for processing the transaction.

Alice, for example, can send transactions to the network, but to have them accepted she needs to

include a proof that the transaction came from her. For this, she can hash the transaction, and sign it (I gave a very similar example in the previous chapter). Anyone can hash the transaction and verify the signature on the hash to see that this is the transaction Alice meant to send. Figure 2.18 illustrates that a man-in-the-middle-attacker who would intercept the transaction before it reached the network would not be able to tamper with the transaction. This is because the hash would change, and the signature would then not verify the new transaction digest.

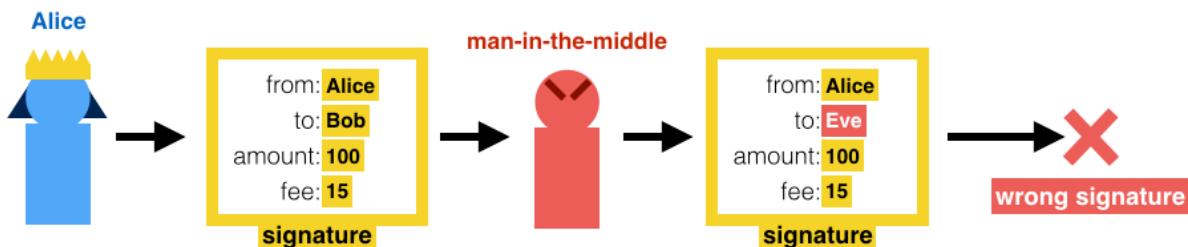


Figure 2.18 Alice sends a transaction, as well as a signature over the hash of the transaction. If a man-in-the-middle attacker attempts to tamper with the transaction, the hash will be different and thus the attached signature will be incorrect.

You will see in chapter 7 on signatures that such an attacker is, of course, unable to forge Alice's signature on a new digest. And thanks to the second pre-image resistance of the hash function used, the attacker cannot find a totally different transaction that would hash to the same digest either.

Is our man-in-the-middle attacker harmless? We're not out of the woods yet.

Unfortunately for the cryptocurrency I was auditing, the transaction was hashed by simply concatenating each field:

```
$ echo -n "Alice""Bob""100""15" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

What can appear as totally fine actually completely broke the cryptocurrency's payment system. Doing this trivially allows an attacker to **break the second pre-image resistance** of the hash function.

Take a few moments to think about how you could find a different transaction that hashes to the same digest 34d6....

What happens if we move one digit from the *tip* field to the *amount* field? One can see that the following transaction will hash to the same digest Alice signed:

```
$ echo -n "Alice""Bob""1001""5" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

And thus, a man-in-the-middle attacker who would want Bob to receive a bit more money would be able to modify the transaction without invalidating the signature. As you've probably guessed,

this is what **TupleHash** solves. It allows you to unambiguously hash a list of fields by using a non-ambiguous encoding. What happens in reality is something very close to the following (with the `||` string concatenation operation):

```
cSHAKE(input="5" || "Alice" || "3" || "Bob" || "3" || "100" || "2" || "10", output_length=256, custom_string="TupleHas
```

The input is this time constructed by preceding each field of the transaction with its length. Take a few moment to understand why this solves our issue.

In general, one can use any hash function safely by always making sure to **serialize** the input before hashing it. Serializing the input means that there always exist a way to deserialize it (meaning to recover the original input). If one can deserialize the data, then there can't be any ambiguity on field delimitation.

2.6 Hashing Passwords

You have seen several useful functions in this chapter that either are hash functions or extend hash functions. Before you can jump to the next chapter, I need to mention **password hashing**.

Imagine the following scenario: you have a website (which would make you a webmaster) and you want to have your users be able to register and login. So you create two web pages for these two respective features. Suddenly, you wonder: how are you going to store their passwords? Do you store them in clear in a database? There seems to be nothing wrong with this at first, you think. It is not perfect though, people tend to re-use the same password everywhere and if (or when) you get breached, and attackers manage to dump all of your users' passwords, it will be bad for them and it will be bad for the reputation of your platform. You think a little bit more, and you realize that an attacker who would be able to steal this database would then be able to log-in as any users. Storing the passwords in clear is now less than ideal and you would like to have a better way to deal with this.

One solution could be to hash your passwords and only store the digests. When someone logs in on your website, the flow would be the following:

- you receive the user's password
- you hash the password they give you and get rid of the password
- you compare the digest with what you had stored previously, if it matches the user is logged in

The flow allows you to handle their passwords for a limited amount time. Still, an attacker that gets into your servers can stealthily remain to log passwords from this flow until you detect his presence. We acknowledge that this is still not a perfect situation but that we still improved its security. In security, we also call this **defense-in-depth**, which is the act of layering imperfect defenses in hope that an attacker will not defeat them all. This is what real-world cryptography is also about.

Other problems exist with this solution:

- If an attacker retrieves hashed passwords he can brute-force or do an exhaustive search (try all possible passwords) and test each attempt against the whole database. Ideally we would want an attacker to only be able to attack one hashed password at a time.
- Hash functions are supposed to be as fast as can be. Attackers can leverage this to brute-force many many passwords per second. Ideally we would have a mechanism to slow down such attacks.

The first issue has been commonly solved by using **salts** which are random values that are public and different for each users. A salt is used along with the user's password when hashing it, which in some sense is like using a per-user customization string with cSHAKE: it effectively creates a different hash function for every user. Since each user uses a different hash function, an attacker cannot pre-compute large tables of passwords (called **rainbow tables**) in hope to test them against the whole database of stolen password hashes.

The second issue has been solved with **password hashes** which are designed to be slow. The current state of the art is **Argon2**, the winner of the Password Hashing Competition that run from 2013 to 2015.⁶ In practice, other algorithms are also used like PBKDF2, bcrypt and scrypt. The problem with these is that they can be used with insecure parameters and are thus not easy to configure in practice. In addition, only scrypt defends against heavy optimizations from attackers.⁷

2.7 Summary

- A hash function provides collision resistance, pre-image resistance and second pre-image resistance.
 - Pre-image resistance means that one shouldn't be able to find the input that produced a digest.
 - Second pre-image resistance means that from an input and its digest, one shouldn't be able to find a different input that hashes to the same digest.
 - Collision resistance means that one shouldn't be able to find two random inputs that hash to the same output.
- The most widely adopted hash function is SHA-2, while the recommended hash function is SHA-3 due to SHA-2's lack of resistance to length-extension attacks.
- SHAKE is an eXtendable Output Function (XOF) that acts like a hash function but provides an arbitrary-length digest.
- cSHAKE for customizable SHAKE allows one to easily create instances of SHAKE that behave like different XoFs. This is called domain separation.
- Objects should be serialized before being hashed, in order to avoid breaking the second pre-image resistance of the hash function. Algorithms like TupleHash automatically take care of this.
- Hashing passwords make use of slower hash functions designed specifically for that purpose. The state of the art being Argon2.

3

Message authentication codes

This chapter covers

- Message Authentication Codes (MAC), a cryptographic primitive to protect the integrity of data.
- The security properties and the pitfalls of MACs.
- The widely adopted standards for MACs.

In the previous chapter 2, you've learned about an interesting construction (a hash function) that on its own does not provide much, but if used in combination with a secure channel allows us to verify the authenticity and integrity of messages. In this chapter, we will see how one can provide integrity and authenticity over messages without the use of a secure channel.

For this chapter you'll need to have read:

- Chapter 2 on Hash Functions.

3.1 What is a Message Authentication Code?

Let's picture the following scenario: you are a webpage. You are bright, full of colors, and above all you are proud of serving a community of loyal users. To interact with your webpage, users must first log-in by sending you their credentials. Receiving these credentials, your job is to first validate them (they could be lying to you after all). If the credentials are matching those that were used when the user first signed up, we deem them correct and say that we **authenticated** the user. From then on, you want to avoid having them re-authenticate in every request to our webpage, so you send them a **session cookie**.



You might have heard of cookies, they are used all over the web to keep track of sessions with users. A cookie is most often just a **random string** that is generated right after a user has logged-in and stored in a database under that user's name. The user then sends you the cookie in every request to authenticate him or herself. When received, you simply check in your database who has been associated to the random string included in the cookie. This is illustrated in figure 3.1.

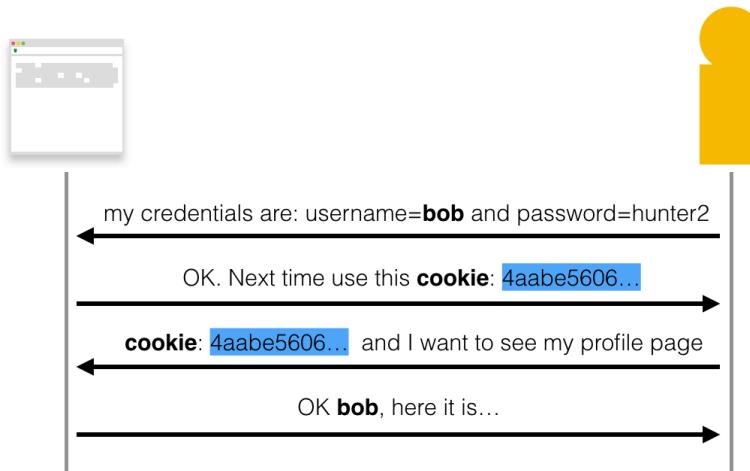


Figure 3.1 A webpage sends a random string as a cookie to Bob after receiving his credentials. Bob can then attach the cookie to every subsequent requests, which will be recognized by the webpage as a proof that this is coming from Bob.

There is nothing wrong with this approach, but you might want to do this in a **stateless** fashion: without having to store anything about the session on your side. This is particularly useful if you have multiple servers behind the webpage that all need to be able to handle the user's cookie, and you don't want them to share a database of cookies. To avoid storing state on your side, you could send a cookie containing the relevant user metadata instead of a random string. This should not leak any private information, so you can simply include the username of the user in there (if you want to do things right, usually you would also include an expiration date). The user can then use this cookie that simply contains his or her username in subsequent queries. This is still not enough though, as a malicious user could simply modify that metadata and impersonate other users.

Great. Our first attempt at a protocol has a **vulnerability**. How do we **fix this with cryptography?**

Let's think of the hash functions that you've learned in the previous chapter 2. Could you use SHA-3 to hash the cookie and provide the user with the obtained digest? I illustrate this in figure 3.2.

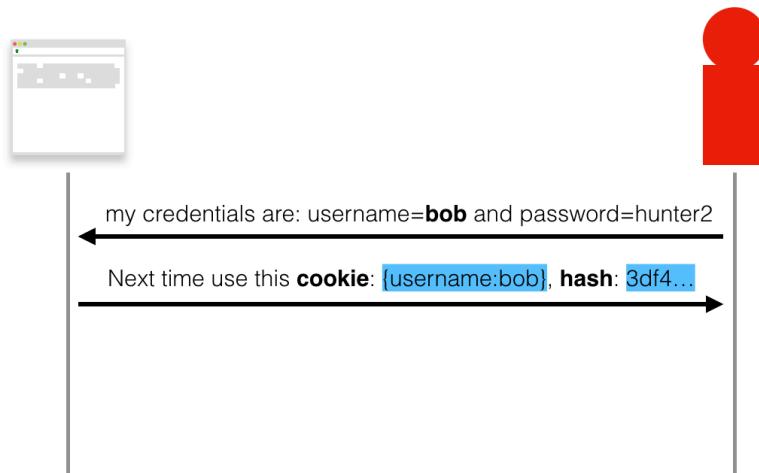


Figure 3.2 A webpage sends metadata information as a cookie, along with a hash of that data, to a user who just sent his credentials.

In subsequent requests, the user would then have to send this digest along with the cookie. When receiving such requests, you would hash the cookie with SHA-3 and compare the output with the digest the user has sent you. If it differs, you would know that the cookie has been modified since you've given it to the user.

Can you see anything wrong with this?

The problem here is that the hash function we are using is a public algorithm and can be re-computed on new data by a malicious user. Indeed, figure 3.3 shows that if a malicious user decides to change their username, they can also recompute the hash and bypass our integrity protection.

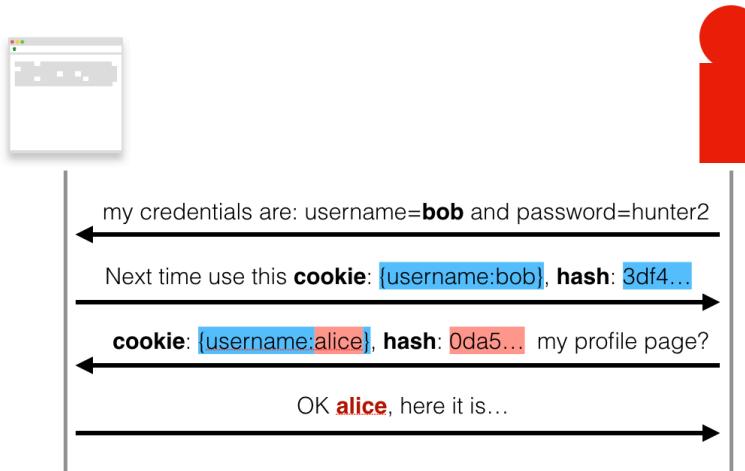


Figure 3.3 A malicious user can modify the information contained in his cookie, and re-compute the hash over that new cookie, to impersonate another user to a webpage.

To solve this, we need to use a different cryptographic primitive called a **message authentication code (MAC)**. Do not focus on the name too much, as it has poorly aged.

A MAC is a secret-key algorithm that has the following interface:



Figure 3.4 The interface of a Message Authentication Code. The algorithm takes a secret key and a message, and deterministically produces an unique authentication tag. Without the key, it should be impossible to reproduce that authentication tag.

A MAC takes a secret key and a message and produces a unique authentication tag. This process is **deterministic**: given the same secret key and the same message, the same authentication tag will be produced.

NOTE

The authentication tag is sometimes called a signature, and the act of using a MAC is sometimes called signing. This is ambiguous, and we won't use these terms, as signatures are very different cryptographic primitives than MACs (as we will see in chapter 7).

To secure our cookie's **integrity** and **authenticity** with a MAC, we can simply remember one secret key and use it to generate an authentication tag on the cookie as seen in figure 3.5.

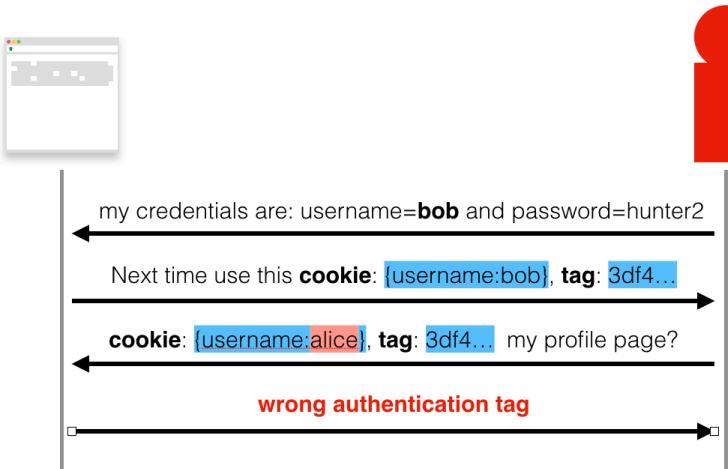


Figure 3.5 A malicious user tampers with his cookie, but cannot forge a valid authentication tag for the new cookie. Subsequently, the webpage cannot verify the authenticity and integrity of the cookie, and thus discard the request.

The malicious user is unable to **forge** a new authentication tag on a modified cookie, because he doesn't have the **secret key**.

This time, when you receive a cookie from the user, you re-compute the authentication tag with the secret key and the received cookie and compare it to the authentication tag sent by the user. If they don't match, you know that you've never created such a cookie.

And that's it! A MAC is like a private hash function that only you can compute (because you know the key). We will see in this chapter that MACs can actually be built using hash functions (you might already have an idea).

Now let's see a different example using real code.

So far we only had one peer use a MAC. Let's increase this number: imagine that you want to communicate with someone else, and you do not care about other people seeing the messages but you do really care about the messages not being modifiable by people in the middle. We can have both peers use the same secret key and a MAC to prevent this as in figure 3.6!

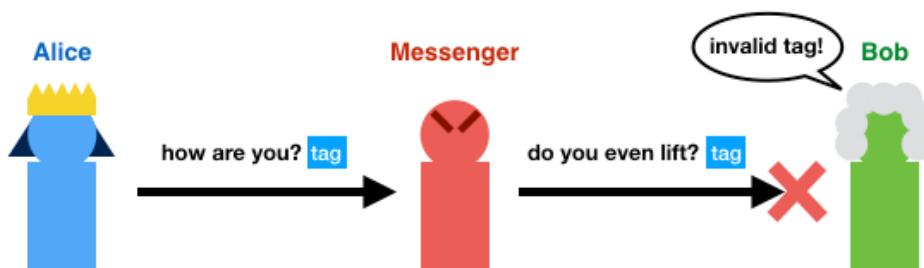


Figure 3.6 Alice sends a message in clear along with an authentication tag to Bob. A man-in-the-middle attacker who intercepts this message cannot tamper with it as the key used for the MAC is only known to Alice and Bob.

For this example, we'll use one of the most popular MAC function: **Hash-Based Message Authentication Code (HMAC)**. As it turns out, HMAC is based on hash functions (we will see that later in this chapter) and is widely supported across pretty much all programming languages. Here is an example in **Golang**, making use of its excellent standard library. (Golang has the best cryptographic standard library a programming language can have.)

Listing 3.1 sending.go

```
import (
    "crypto/hmac"      ①
    "crypto/sha256"     ①
)

func sendMessage(secret_key, counter, message []byte) {
    mac := hmac.New(sha256.New, secret_key) ②

    mac.Write(counter)   ③
    mac.Write(message)   ③
    authenticationTag := mac.Sum(nil)        ③

    to_send := append(message, authenticationTag...) ④
    send(message, authenticationTag)

    increment_counter(counter) ⑤
}
```

- ① Golang's standard library is extremely complete and features most of the widely accepted cryptographic algorithms.
- ② HMAC is instantiated with a secret key and the SHA-256 hash function. It is possible to use other hash functions but most often HMAC is paired with SHA-256.
- ③ HMAC can be used to produce an authentication tag over the message. This pattern is known as IUF for Initialize-Update-Finalize (or in this case, New-Write-Sum). Multiple `Write` calls could have been written as a single `write`, but sometimes the full bytestring to MAC is not accessible right away.
- ④ an authentication tag (or simply "tag") is often appended to the message it authenticates, before being sent.
- ⑤ A counter is used in front of every message in order to prevent replays. Don't worry too much about this as I will talk more about that in the next section.

On the other side, the process is very similar. After receiving both the message and the authentication tag, your friend can generate his or her own tag with the same secret key, and compare them. Of course, similarly to encryption, both of you need to share the same secret key in order to make this work.

Listing 3.2 receiving.go

```
func sendMessage(secret_key, counter, receivedMessage []byte) ([]byte, error) {
    mac := hmac.New(sha256.New, secret_key) ①

    tagOffset := len(receivedMessage) - 32 ②
    receivedMessage, receivedTag := receivedMessage[:tagOffset], receivedMessage[tagOffset:] ②

    mac.Write(counter) ③
    mac.Write(receivedMessage) ③
    authenticationTag := mac.Sum(nil) ③

    if !hmac.Equal(authenticationTag, receivedTag) { ④
        return nil, fmt.Errorf("invalid authentication tag")
    }

    increment_counter(counter)
    return receivedMessage, nil ⑤
}
```

- ① The receiver also has to initialize HMAC with the same secret key you used.
- ② Since the authentication tag was appended, the receiver must retrieve it from the message. In our example the authentication tag is 32 bytes because we use SHA-256 as the hash function (which has a 32-byte output length).
- ③ The receiver then computes the authentication tag over the received message, like you did before sending it.
- ④ The receiver compares the computed authentication tag with the received one. If the message has been modified the comparison will fail. Note that simply comparing these values is insecure (`bytes.Equal(authenticationTag, receivedTag)`) and the receiver must use the provided `hmac.Equal` function. More about this later in this chapter.
- ⑤ The message is returned and used only after its integrity has been checked with the authentication tag.

Now that you know what a MAC can be used for, I'll talk about some of the gotchas of MACs in the next section. It should answer your questions about the `counter` we used, and why we cannot directly compare authentication tags (which is a common mistake).

3.2 Security Properties of a Message Authentication Code

MACs, like all cryptographic primitives, have their oddities and pitfalls. Before going any further, I will provide a few explanations on what security properties MACs provide and how to use them correctly. You will learn in this order:

- MACs are resistant against forgery of authentication tags.
- An authentication tag needs to be of a minimum length to be secure.
- Messages can be replayed if authenticated naively.
- Verifying an authentication tag is prone to bugs.

Let's get started!

3.2.1 Forgery of Authentication Tag

The general security goal of a MAC is to prevent **authentication tag forgery** on a new message. This means that without knowledge of the secret key k , one cannot compute the authentication tag $t = \text{MAC}(k, m)$ on messages m of his or her choice.

This sounds fair right? How can we compute a function if we're missing an argument. But MACs provide much more than that! real-world applications often let attackers control parts of the input message m argument of a MAC. For example, this was the case in our previous introduction scenario (you can chose your nickname at registration). Thus, MACs work even in these types of situations: even if an attacker can ask you to produce the authentication tags for a large number of arbitrary messages, the attacker should still not be able to forge an authentication tag on a new message without asking you.

NOTE

One could wonder how proving such a property is useful. Indeed, if the attacker can directly request authentication tags on arbitrary messages then what is there left to protect? But this is how security proofs work in cryptography: they take the most powerful attacker and show that even then, the attacker is hopeless. In practice, the attacker is usually less powerful and thus we have confidence that, if a powerful attacker can't do something bad, a less powerful one has even less recourse. Let's take the example of a service that lets you create authentication tags for messages like `{username: bob}` with arbitrary usernames, except for the username "david". Then the security property above says that even by asking for a trillion such cookies with diverse usernames, we still can't forge a cookie with username "david".

As such, you should be protected against such forgeries, **as long as the secret key used with the MAC stays secret**. This implies that the secret key has to be random enough (more on that in chapter 8) and large enough (usually 16 bytes). Furthermore, a MAC is vulnerable to the same type of **ambiguous attack** we've seen in c 2. If you are trying to authenticate structures, make sure to serialize them before authenticating them with a MAC, otherwise forgery might be trivial.

3.2.2 Lengths of Authentication Tag

Another possible attack against usage of MACs are **collisions**.

Remember, finding a collision for a hash function means finding two different inputs x and y such that $\text{HASH}(x) = \text{HASH}(y)$. We can extend this definition to MACs by defining a collision when $\text{MAC}(k, x) = \text{MAC}(k, y)$ for inputs x and y . As we've learned in the previous chapter 2 with the **birthday bound**, collisions can be found with high probability if the output length of

our algorithm is too small. For example with MACs, an attacker who has access to a service producing 64-bit authentication tags can find a collision with high probability by requesting a much lower number (2^{32}) of tags. Such a collision is rarely exploitable in practice, but there exist some scenarios where collision resistance matters. For this reason, we would want an authentication tag size that would limit such attacks. In general, 128-bit authentication tags are used as they provide enough resistance.

[requesting 2^{64} authentication tags] would take 250,000 years in a continuous 1Gbps link, and without changing the secret key K during all this time

– RFC 2104: HMAC: Keyed-Hashing for Message Authentication

Using 128-bit authentication tag might appear counter-intuitive, since we wanted 256-bit outputs for hash functions. But hash functions are public algorithms that one can compute **offline**, which allows an attacker to optimize and parallelize an attack heavily. With a keyed function like a MAC, an attacker cannot efficiently optimize the attack offline and is forced to directly request authentication tags from you (which usually makes the attack much slower). A 128-bit authentication tag would require 2^{64} **online** queries from the attacker in order to have a 50% chance to find collisions, which is deemed large enough. Nonetheless, one might still want to increase an authentication tag to 256-bit, which is possible as well.

3.2.3 Replay Attacks

One thing I still haven't mentioned are **replay attacks**. Imagine that Alice and Bob communicate in the open (using an insecure connection) and append each of their messages with an authentication tag in order to protect them from tampering. To do that they both use two different secret keys, k_1 for protecting the integrity of messages coming from Bob and k_2 for protecting the integrity of messages coming from Alice (we've mentioned this concept called **domain separation** in the previous chapter 2. To avoid ambiguity you should always use different keys or parameters for different use cases). See figure 3.7.

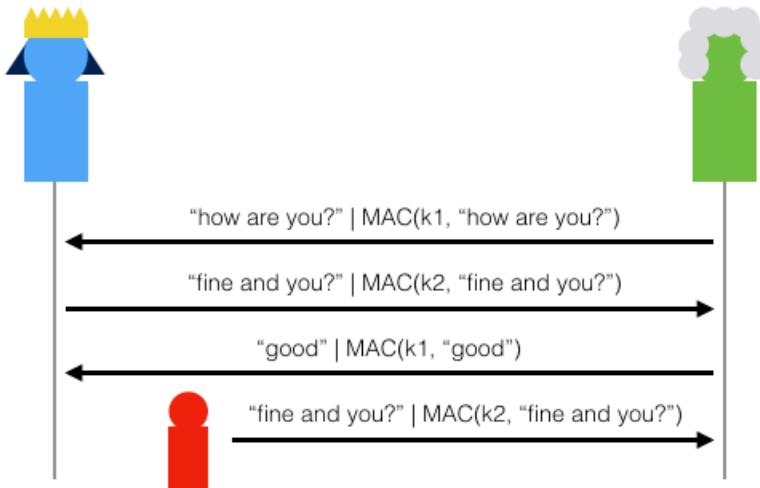


Figure 3.7 Two users sharing two keys k1 and k2 exchange messages, along with authentication tags. These tags are computed out of k1 or k2 depending on the direction of the messages. A malicious observer replays one of the messages to the user.

In this scenario, nothing prevents a malicious observer from replaying one of the messages to its recipient. For this reason, a protocol relying on MAC must be aware of this and build protections against this. One way is to add an incrementing counter to the input of the MAC as in figure 3.8.

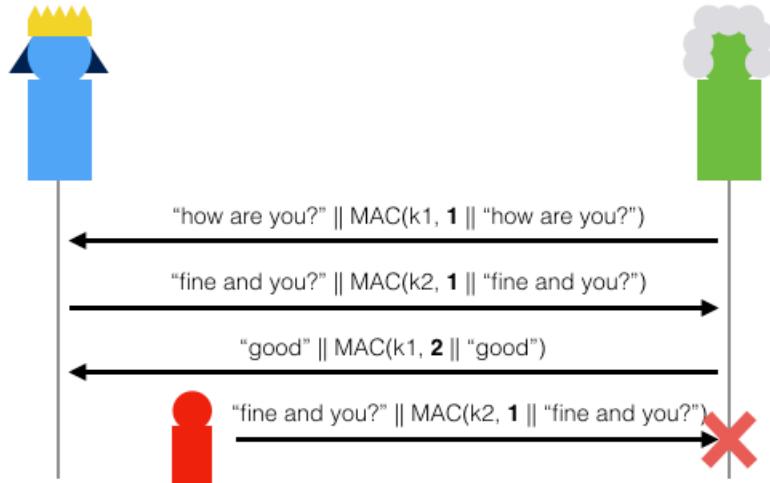


Figure 3.8 Two users sharing two keys k1 and k2 exchange messages, along with authentication tags. These tags are computed out of k1 or k2 depending on the direction of the messages. A malicious observer replays one of the messages to the user. Since the victim has incremented his counter, the tag will be computed over 2 || "fine and you?" and will not match the tag sent by the attacker. This will allow the victim to successfully reject that replayed message.

In practice, counters are often a fixed 64-bit length. This allows one to send 2^{64} messages before filling up the counter (and risking it to wrap around and repeat itself). Of course, if the shared secret is **rotated** frequently (meaning that after X messages, participants agree to use a new shared secret), then the size of the counter can be reduced and reset to 0 after a key rotation.

(You should convince yourself that re-using the same counter with two different keys is OK.) Counters are **never variable-length** because of ambiguous attacks (again). Can you figure out how a variable-length counter could possibly allow an attacker to forge an authentication tag? You can find an example in the answer appendix at the end of this book.

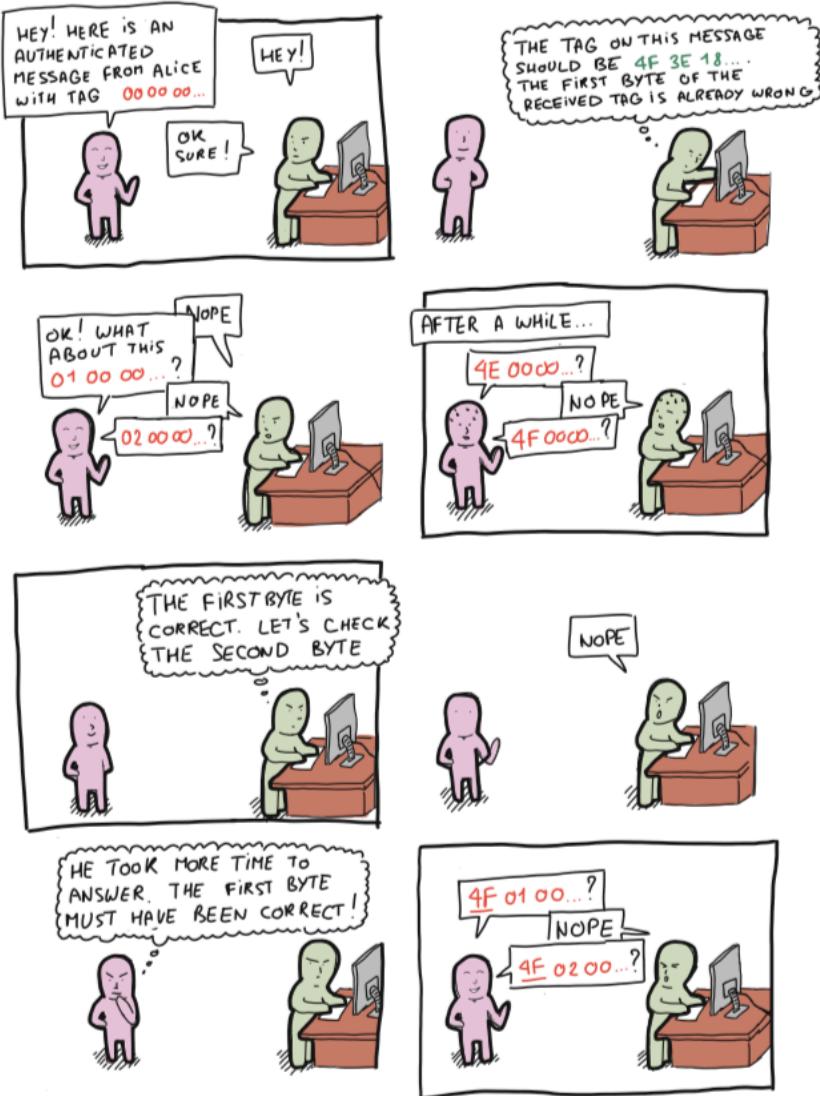
3.2.4 Verifying Authentication Tags in Constant-Time

This last one is dear to me, as I found this vulnerability many times in different applications I've audited as part of my work.

NOTE

As you will see, this vulnerability is quite **subtle**. It belongs to the larger class of attacks called **side-channel attacks**, as it does not exploit the algorithm directly, but indirectly by analyzing some information leaked by the algorithm being run. Side-channel attacks can be exploited by measuring the power consumption of a device during a cryptographic operation, or the electromagnetic radiations emitted, or even by analyzing the time an algorithm takes to run (which is the case here).

When verifying an authentication tag, the comparison between the received authentication tag and the one you computed must be done in **constant-time**. This means the comparison should always take the same time (assuming the received one is of the correct size). If the time it takes to compare the two authentication tags is not constant-time, it is probably because it returns the moment the two tags differ. This usually gives enough information to enable attacks that can re-create a valid authentication tag byte-by-byte by measuring how long it took for the verification to finish. I explain this in the following comicstrip. We call these types of attacks **timing attacks**.



Fortunately for us, cryptographic libraries implementing MACs also provide convenient functions to verify an authentication tag in constant-time. If you're wondering how this is done, here is how Golang implements `hmac.Equal` in constant-time code (taken from golang.org/src/crypto/subtle/constant_time.go?s=505:546#L2):

Listing 3.3 constant_time.go.go

```
for i := 0; i < len(x); i++ {
    v |= x[i] ^ y[i]
}
```

How this works is left as an exercise for the reader.

3.3 MAC in the real-world

Now that I have introduced what MACs are and what security properties they provide, let's take a look at how people use them in real settings.

Authentication of Messages. MACs are used in many places to ensure that the communication between two machines or two users was not tampered with. This is necessary in both cases where communications are in clear and where communications are encrypted. I have already explained how this happens when communications are transmitted in clear, and I will explain how this is done when communications are encrypted in the next chapter 4 on Authenticated Encryption.

Deriving keys. One particularity of MACs is that they are often designed to produce bytes that look random (like hash functions). This property can be utilized to use a single key to generate random numbers or to produce more keys. In chapter 8 on secrets and randomness, I will introduce the **HMAC-based Key Derivation Function (HKDF)** that does exactly this by using **HMAC**, one of the MAC algorithm we will talk about in this chapter.

NOTE

Imagine the set of all functions that take a variable-length input and produce a random output of a fixed-size. If we could pick a function at random from this set, and use it as a MAC (without a key), it would be swell. We would just have to agree on which function (kind of like agreeing on a key). Unfortunately, we can't have such a set (it is way too large) and thus we can emulate picking such a random function by designing something close enough: we call such constructions **pseudo-random function (PRF)**. HMAC and most practical MACs are such constructions: they are randomized by the key argument instead. Choosing a different key, is like picking a random function. Caution though, as not all MACs are PRFs.

Integrity of Cookies. To track your users' browser sessions you can send them a random string (associated to their metadata) or send them the metadata directly, attached with an authentication tag so that they cannot modify it. This is what I explained in the introduction example.

Hash Tables. Programming languages usually expose data structures called hash tables (also called hashmaps, dictionaries, associated arrays, etc.) that make use of non-cryptographic hash functions. If a service expose this data structure in such a way where the input of the non-cryptographic hash function can be controlled by attackers, this can lead to **denial of service** (DoS) attacks (meaning that an attacker can render the service unusable). To avoid this, the non-cryptographic hash function is usually randomized at the start of the program. Many major applications have decided to use a MAC with a random key in place of the non-cryptographic hash function. This is the case for many programming languages like Rust, Python, Ruby or major applications like the Linux kernel that all make use of the SipHash MAC with a random key generated at the start of the program.⁸

3.4 Message Authentication Codes in Practice

You've learned that MACs are cryptographic algorithms that can be used between one or more parties in order to protect the integrity and the authenticity of information. As widely-used MACs also exhibit good randomness, MACs are also often used to produce random numbers deterministically in different types of algorithms (like TOTP).

In this section, we will look at two standardized MAC algorithms that one can be use nowadays: HMAC and KMAC.

3.4.1 HMAC, a Hash-Based Message Authentication Code

The most widely used MAC is **HMAC** (for Hash-based MAC) invented in 1996 by M. Bellare, R. Canetti, and H. Krawczyk and specified in RFC 2104, FIPS PUB 198, and ANSI X9.71. HMAC, like its name indicates, is a way to use hash functions with a key. Using a hash function to build MACs is a popular concept as hash functions have widely available implementations, are fast in software, and also benefit from hardware support on most systems.

Remember that I have mentioned in chapter 2 that SHA-2 should not be used directly to hash secrets due to **length-extension attacks** (more on that at the end of this chapter). How does one figure out how to transform a hash function into a keyed function? This is what HMAC solves for us. Under the hood, HMAC is pretty basic, as illustrated in figure 3.9.

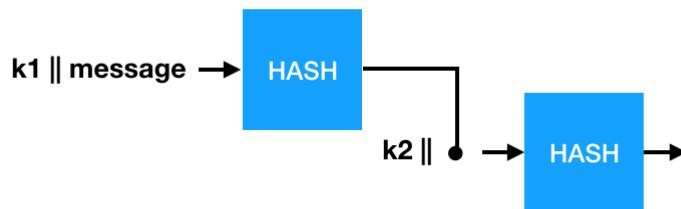


Figure 3.9 HMAC works by hashing the concatenation (\parallel) of a key k_1 and the input message, and by then hashing the concatenation of a key k_2 with the output of the first operation. k_1 and k_2 are both deterministically derived from a secret key k .

More formally, HMAC follows these steps:

1. It first creates two keys from the main key: $k_1 = k \text{ } \text{ipad}$ and $k_2 = k \text{ } \text{opad}$ where ipad (inner padding) and opad (outer padding) are constants and opad is the symbol for the XOR operation.
2. It then concatenates a key k_1 with the message and hashes it.
3. The result is concatenated with a key k_2 and hashed one more time.
4. This produces the final authentication tag.

Because of this, the authentication tag's size is dictated by the used hash function. HMAC-SHA256 makes use of SHA-256 and produces an authentication tag of 256 bits;

HMAC-SHA512 produces an authentication tag of 512 bits; and so on.

WARNING While one can truncate the output of HMAC to reduce its size, an authentication tag should be at minimum 128 bits as we've talked about earlier. This is not always respected and some applications will go as low as 64 bits due to explicitly handling a very limited amount of queries. There are trade-offs with this approach and once again it is important to read the fine print before doing something non-standard.

HMAC was constructed this way in order to facilitate proofs. In several papers, HMAC is proven to be secure against forgeries as long as the hash function underneath holds some good properties, which all cryptographically secure hash functions should hold. For this reason, HMAC can be used in combination with a large number of hash functions. Today, HMAC is mostly used with SHA-2.

3.4.2 KMAC, a hash based on cSHAKE

As SHA-3 is not vulnerable to length-extension attacks (this was actually a requirement for the SHA-3 competition), it makes little sense to use SHA-3 with HMAC instead of something like `SHA-3-256(key || message)` that would work well in practice.

This is exactly what **KMAC** is. It makes use of **cSHAKE**, the customizable version of the SHAKE eXtendable Output Function (XOF) we've seen in chapter 2 to build the following construction (keep in mind that this is a simplified explanation of cSHAKE, in practice the construction is a bit different):

```
cSHAKE(input=key || "hello world", output_length=128, custom_string="KMAC" || "my_hash")
```

As one can see, since KMAC is based on a XOF it also let the caller decide on the authentication tag size and a customization string. This makes KMAC a very versatile function in practice.

3.5 SHA-2 and Length-Extension Attacks

we have mentioned several times that one shouldn't hash secrets with SHA-2 as it is not resistant to **length-extension attacks**. In this section, we aim to provide a simple explanation of what this attack is.

Let's go back to our introduction scenario, to the step where we attempted to simply use SHA-2 in order to protect the integrity of the cookie. Remember that it was not good enough, as the user can tamper with the cookie (for example, by adding an `admin=true` field) and re-compute the hash over the cookie. Indeed, SHA-2 is a public function and nothing prevents the user from doing this. This is illustrated in figure 3.10.

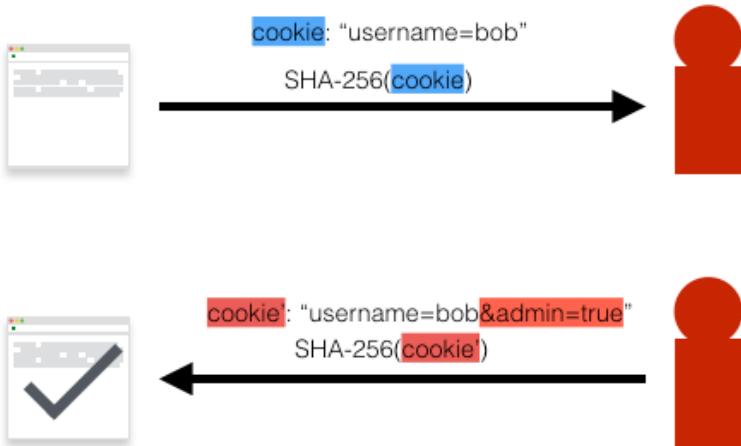


Figure 3.10 A webpage sends a cookie, followed by a hash of that cookie, to a user. The user is then required to send the cookie to authenticate him/herself in every subsequent request. Unfortunately, a malicious user can tamper with the cookie and re-compute the hash, breaking the integrity check. The cookie is then accepted as valid by the webpage.

The next best idea could have been to add a secret key to what we hash. This way the user cannot re-compute the digest as the secret key is required. Very much like a MAC. On receipt of the tampered cookie, the page will compute $\text{SHA-256}(\text{key} \parallel \text{tampered_cookie})$ (where \parallel represents the concatenation of the two values) and obtain something that won't match what the malicious user probably sent. We illustrate this in figure 3.11.

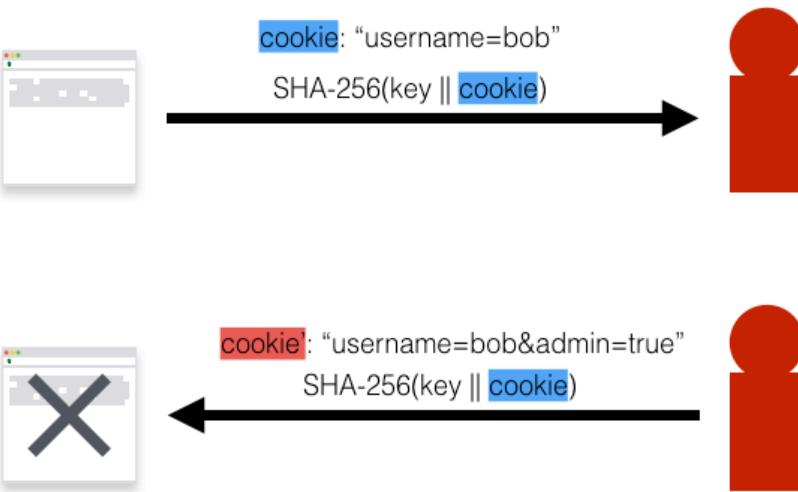


Figure 3.11 By using a key when computing the hash of the cookie, one could think that a malicious user (who wants to tamper with his cookie) wouldn't be able to compute the correct digest over the new cookie. We will see later that this is not true for SHA-256.

Unfortunately, SHA-2 has an annoying peculiarity: from a digest over an input, one can compute the digest of an input and more. What does this mean? Let's take a look at figure 3.12 where one uses SHA-256 as $\text{SHA-256}(\text{secret} \parallel \text{input1})$.

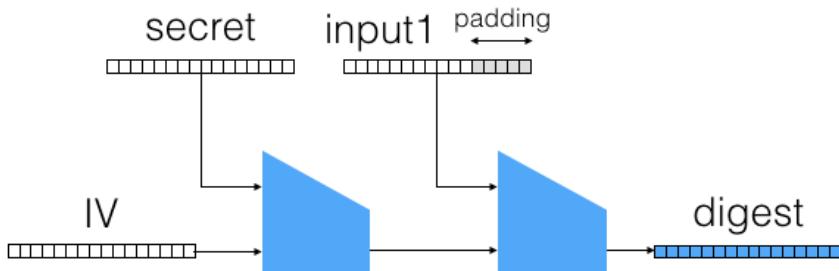


Figure 3.12 SHA-256 is used to hash a secret concatenated with a cookie (here named input1). Remember that SHA-256 works by having the Merkle–Damgård construction to iteratively call a compression function over blocks of the input, starting from an initialization vector (IV).

This figure is highly simplified, but imagine that `input1` is the string `user=bob`. Notice that the digest obtained is, effectively, the full intermediate state of the hash function at this point. Nothing prevents one from pretending that the padding section is part of the input and continuing the Merkle–Damgård dance. In diagram 3.13 we illustrate this attack, where one would take the digest and compute the hash of `input1 || padding || input2`. In our example, `input2` is `&admin=true`.

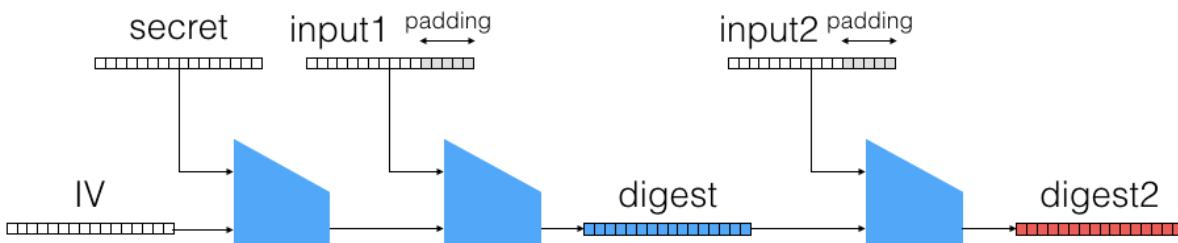


Figure 3.13 The output of the SHA-256 hash of a cookie (in blue) is used to extend the hash to more data, creating a hash (in red) of the secret concatenated with `input1`, the first padding bytes, and `input2`.

This vulnerability allows one to continue hashing, from a given digest, like the operation was not finished. Breaking our previous protocol as can be seen in figure 3.14.

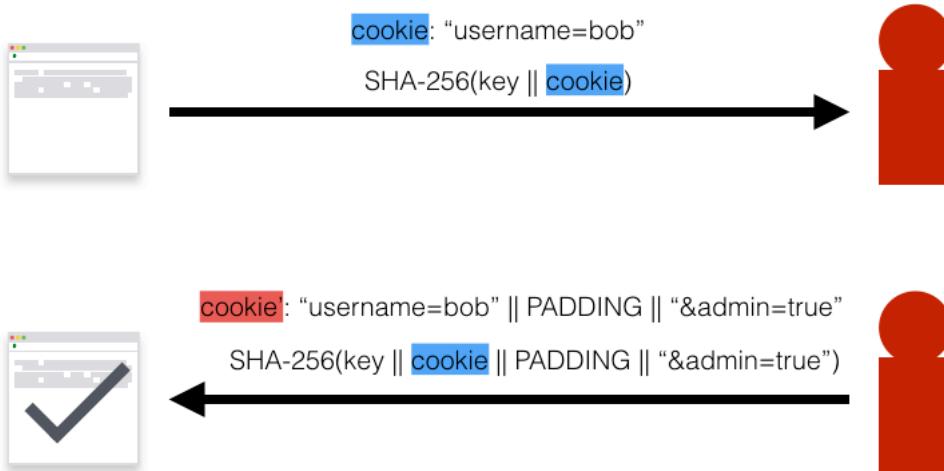


Figure 3.14 An attacker successfully uses a length-extension attack to tamper with his/her cookie and compute the correct hash using the previous hash.

The fact that the first padding now needs to be part of the input might prevent some protocols to be exploitable. Still, the smallest amount of change might re-introduce a vulnerability.

For this reason one should never, ever hash secrets with SHA-2. Of course there are several other ways to do it correctly (for example `SHA-256(k || message || k)` works) which is what HMAC provides. So use HMAC if you want to use SHA-2, use KMAC if you prefer SHA-3.

3.6 Summary

- Message Authentication Codes (MACs) are symmetric cryptographic algorithms that allow one or more parties who share the same key to verify the integrity and authenticity of messages.
 - To verify the authenticity of a message and its associated authentication tag, one can re-compute the authentication tag of the message (and a secret key) and match the two authentication tags. If they differ, the message has been tampered with.
 - Always compare a received authentication tag with a computed one in constant-time.
- While MACs protect the integrity of messages, by default they do not detect when messages are being replayed.
- Standardized and well accepted MACs are the HMAC and the KMAC standards.
- One can use HMAC with different hash functions, in practice HMAC is often used with the SHA-2 hash function.
- Authentication tags should be of a minimum length of 128 bits to prevent collisions and forgery of authentication tags.
- Never use SHA-256 directly to build a MAC as it can be done incorrectly. Always use a function like HMAC to do this.

Authenticated encryption



This chapter covers

- Symmetric Encryption, a cryptographic primitive to hide communication from observers
- Authenticated Encryption, the secure evolution of symmetric encryption.
- The popular authenticated encryption algorithms.
- Other types of symmetric encryption.

This is where it all began: the science of cryptography was first and foremost invented to fill our need to hide information. Encryption is what pre-occupied most of the early cryptographers: "How can we prevent observers from understanding our conversations?". While the science and its advances first bloomed behind closed door, benefiting the governments and their military only, it has now opened and spread throughout the world. Today, encryption is used everywhere to add a sense of privacy and security in the different aspects of our modern lives. In this chapter we'll find out what encryption really is, what types of problem it solves and how today's applications makes use of this cryptographic primitive.

For this chapter you'll need to have read:

- Chapter 3 on Message Authentication Codes.

4.1 What Is a Cipher?

So far we've talked about hash functions and message authentication codes (MACs). These two constructions have distinctive interfaces that we recapitulate in figure 4.1.

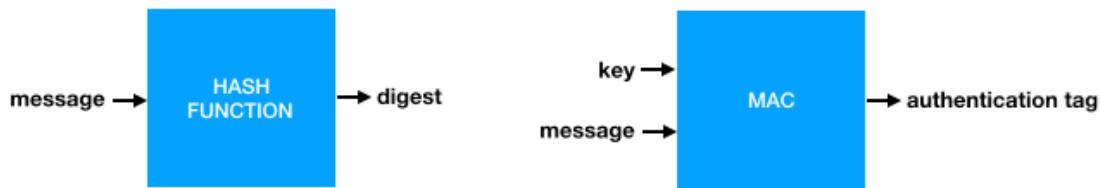


Figure 4.1 The interfaces of the hash function and the message authentication code.

We will now attempt to introduce one as equally important cryptographic primitive: **authenticated encryption**.

It's like when you use slang to talk to your siblings about what you'll do after school so your mom doesn't know what you're up to.

— Natanael L.

Let's imagine that our two characters Alice and Bob want to exchange some messages and they both want to keep these messages confidential. In practice they have many mediums as their disposition: the mail, phones, the Internet, etc. and each of these mediums are by default insecure: The mailman could open their letters, the telecommunication operators can spy on their calls and text messages, Internet service providers or any servers on the Internet network that are in between Alice and Bob have access to the content of the packets being exchanged.

Without further ado, let's introduce Alice and Bob's savior: the **encryption algorithm**, also called "**cipher**". For now, let's picture this new algorithm as a black box (see figure 4.2) that Alice can use to **encrypt** her messages to Bob. By "encrypting" a message, Alice transforms it into something that looks random.

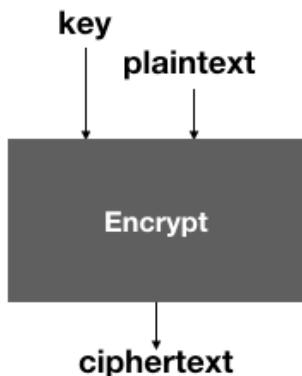


Figure 4.2 Encryption transforms a message (plaintext) into a ciphertext, via the help of a secret key. A ciphertext appears random to the outside world, and thus does not reveal any information about the content of the plaintext, until it is decrypted via a different algorithm and the same secret key.

This encryption algorithm takes:

- A **secret key**. It is crucial that this element is unpredictable (so random), and well protected, as the security of the encryption algorithm relies directly on the secrecy of the

key. We will talk more about this in chapter 8 on secrets and randomness.

- A **plaintext**. This is what you want to encrypt. It can be some text, an image, a video, or anything that can be translated into bits (your love for ramen might be a bit too abstract for encryption for example).

This encryption process produces a **ciphertext**, which is the encrypted content. Alice can safely use one of the mediums we listed previously to send that ciphertext to Bob. The ciphertext will look random to anyone who does not know the secret key, and no information about the content of the message (the plaintext) will be leaked. Once Bob receives this ciphertext, he can use a "decryption algorithm" in order to revert the ciphertext into the original plaintext. We illustrate this algorithm as a black box in figure 4.3:

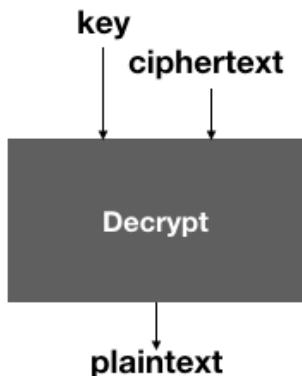


Figure 4.3 Decryption transforms a ciphertext back into its original plaintext, with the help of a secret key.

Decryption takes:

- A **secret key**. This is the same secret key that Alice has used to create the ciphertext. Because the same key is used for both algorithms, we sometimes call the key a **symmetric key**. This is also why we also sometimes specify that we are using **symmetric encryption** and not just **encryption**.
- A **ciphertext**. This is the encrypted message Bob received from Alice.

The process then reveals the original **plaintext**. The full flow is illustrated in figure 4.4.

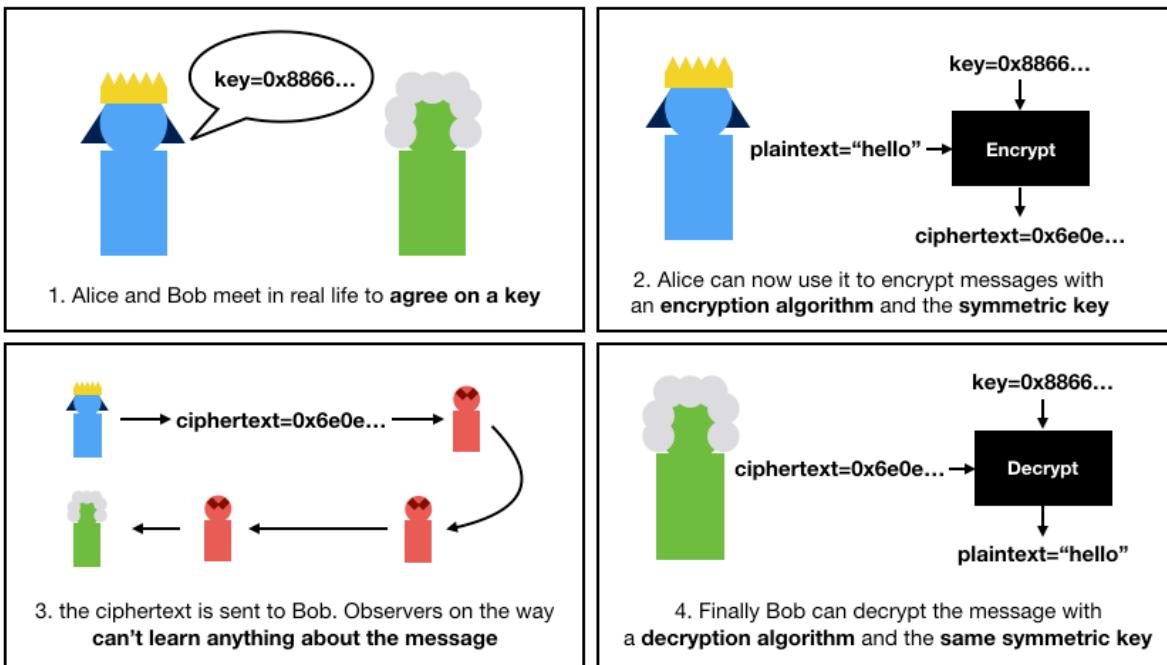


Figure 4.4 Alice (top right) can encrypt the plaintext "hello" with the key `0x8866...` (abbreviated hexadecimal). The ciphertext produced can be sent to Bob. Bob (bottom right) can decrypt the received ciphertext by using the same key and a decryption algorithm.

And this is it! Encryption allows Alice to transform her message into something that looks random and that can safely be transmitted to Bob, and decryption allows Bob to revert the encrypted message back to the original message. This new cryptographic primitive provides something called **confidentiality** (or **secrecy**, or **privacy**) to their messages.

NOTE

We haven't talked about how Alice and Bob agreed to use the same symmetric key. For now, we'll assume that one of them had access to an algorithm that generates unpredictable keys, and that they met in person to exchange the key. In practice, how to bootstrap such protocols with shared secrets is often one of the great challenges companies have to solve. We will see throughout this book many examples of solutions to this problem.

4.2 Symmetric Encryption in the real-world

Now that we have introduced the new concept that is encryption, let's see where it is used in practice and what kind of attackers it defends against. This should give you a peek at how one can leverage the security properties provided by this primitive.

Cookies. We have seen examples of how one can build stateless cookies with message authentication codes (MACs). In this solution a cookie contained some metadata information about the user in clear, and was attached to an authentication tag. In some cases, we do not want the user to have access to the content of the cookie. One solution is to encrypt it, which will

provide the same integrity protection as the MAC in addition to masking the content of the cookie to the user.

The Web. Browsing the web implies that you're communicating with servers that are located outside of your home network, sometimes even sitting on the other side of the world. Queries go through your home router (or the starbucks public WiFi), pass through the Internet Service Provider, and then the many servers on the way to your destination. Many of these servers could potentially be actively analyzing the traffic they send forward. While current solutions do not hide what websites you are visiting (for example, you cannot hide that you are visiting www.iana.org) from these **man-in-the-middle (MITM) attackers**, they can hide what specific pages of these websites you are browsing (for example, nobody would know that you are on www.iana.org/domains/reserved), the content of the pages, what you are submitting in forms, etc. The most widely deployed solution to secure these connections is called **HTTPS**, which relies on the **Transport Layer Security (TLS) protocol** (that we will see in chapter 9). TLS makes heavy use of encryption to provide confidentiality to your browsing.

Virtual Private Network. Companies can be distributed across several cities or countries, each offices (or data center) having their own private networks. In addition, employees could be working from different locations including their home. To allow communication and access to company tools and services, one solution is to make them accessible from the Internet. This can be unsettling for many companies that would rather keep these under their network. For these, solutions like the **Internet Protocol Security (IPSEC)** exist. IPSEC is very similar to TLS, in which it heavily relies on symmetric encryption to provide confidentiality to the packets from a private network transiting through the Internet to reach another private network.

End-to-End Encryption. While encryption can be used to secure communications from one server to the other, the concept of encrypting communications through two ends is called end-to-end encryption. One can see TLS as providing end-to-end encryption between a user and a web server. The term is also often used to describe encryption between two users, like email encryption or secure messaging. Interestingly, end-to-end encryption also provides cover to criminals and terrorists, which makes it a controversial topic that is often politicized.

Encryption at Rest. Data is precious, and you sometimes don't want it to end up in the hands of someone else. For this reason, you might want to encrypt it as long as you don't use it, only decrypting it when needed. This is called encryption at rest and is used everywhere nowadays. Most modern phones will encrypt their content when not in use; modern OS provide encryption for laptops and desktop computers; Servers have solutions to encrypt their files and databases. All of this is usually transparent to the user.

Cryptographic Primitives. Ciphers can also be used inside of another cryptographic primitive to provide other properties. For example, to generate random numbers, or inside compression functions if you remember chapter 2 on hash functions.

Part 2 of this book will introduce many more protocols that make use of encryption. Now, let's see what the standard algorithms are for encryption.

4.3 The AES-CBC-HMAC Encryption Algorithm

In this chapter we will talk about three main **authenticated encryption** algorithms. Notice that we have yet to introduce what authenticated encryption is. This section will begin by introducing the **Advanced Encryption Standard (AES)** and will then proceed to explain that AES cannot be used as-is. The section will then introduce the **AES-CBC-HMAC** algorithm which augments AES in order to make it secure.

4.3.1 The Advanced Encryption Standard (AES)

The **AES** algorithm was the product of the long AES competition organized by the National Institute of Standards and Technology of the United States (NIST). It lasted from 1997 to 2000, during which 15 different designs were submitted from different countries. At the end, only one ("Rijndael") submission by two Belgian cryptographers Vincent Rijmen and Joan Daemen, was nominated as the winner. AES is still used nowadays, and has survived the test of time: it has not shown any major signs of weaknesses along the years. We will see in later sections that AES is rarely used on its own, but for now, let's see how AES works.

AES offers three different versions: **AES-128** takes a key of 128 bits, **AES-192** takes a key of 192 bits, and **AES-256** takes a key of 256 bits. The length of the key, dictates the level of security: the bigger the stronger. Nonetheless, most applications make use of AES-128 as it provides enough security: **128-bit of security**.

The term **bit-security** is commonly used to indicate the security of cryptographic algorithms. For example with AES-128, it specifies that the best attack we know of would take around 2^{128} operations. This number is gigantic, and is the security level that most applications aim for. The fact that a 128-bit key with AES provides 128-bit of security (and AES-256 provides 256-bit security) is specific to the algorithm itself, the size of the key cannot be shorter but it can be bigger. Imagine using a 100-bit key instead, trying all the possible keys (exhaustive search) would take 2^{100} operations and would eventually work, reducing the security to 100-bit.

NOTE

How big is 2^{128} ? Observe that the amount between two powers of 2 is doubled. (For example 2^2 is half 2^3 .) Then imagine that to reach 2^{128} you have doubled your initial amount 128 times. The number we obtain is *340 undecillion 282 decillion 366 nonillion 920 octillion 938 septillion 463 sextillion 463 quintillion 374 quadrillion 607 trillion 431 billion 768 million 211 thousand 456*. It is quite hard to imagine how big that number is, but you can assume that we will never be able to reach such a number in practice. We also didn't account for the amount of space required for such attacks to work, which is equally as enormous in practice. That is to say, it is foreseeable that AES-128 will remain secure for as long as we know, unless advances in cryptanalysis find a yet undiscovered vulnerability that would reduce the number of operations needed to attack the algorithm.

The interface of AES is the following:

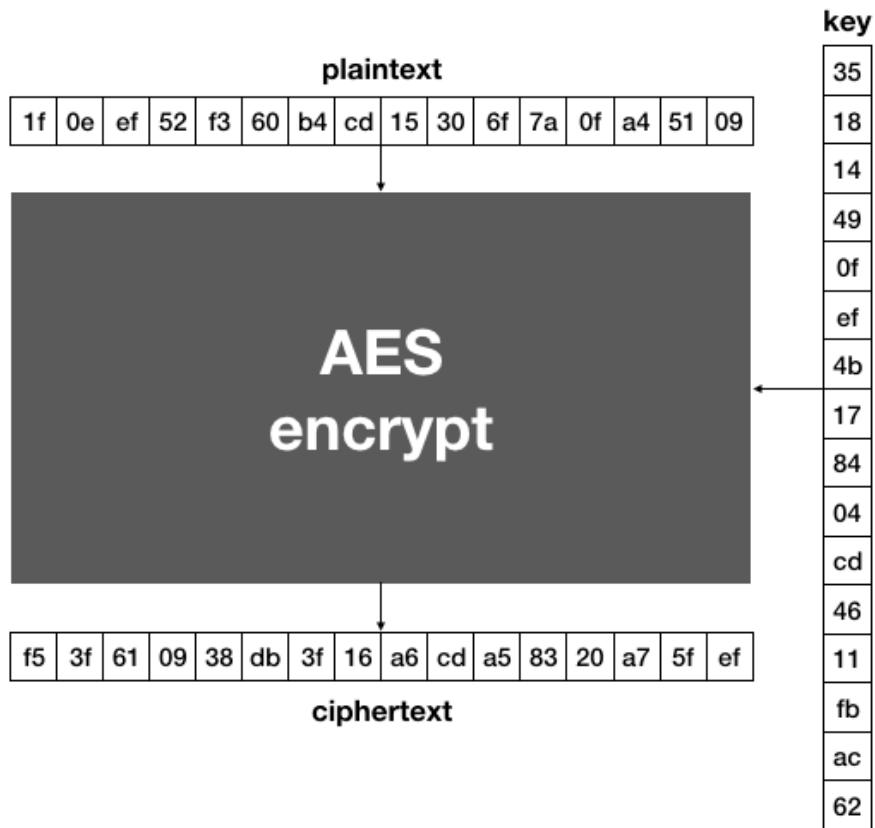


Figure 4.5 The AES encryption algorithm takes a variable-length key (16, 24 or 32 octets) and a 16-octet plaintext. It then produces a 16-octet ciphertext.

The algorithm takes:

- A **symmetric key** of size 128-bit (16-octet), 192-bit (24-octet), or 256-bit (32-octet).
- Encryption: a **plaintext** of 16 octets exactly, and produces a **ciphertext** of the same size.
- Decryption: a **ciphertext** of 16 octets exactly, and produces a **plaintext** of the same size.

The decryption operation is reverting the encryption operation, you only get one ciphertext when encrypting, and you cannot retrieve different plaintexts by decrypting. The encryption and decryption operations are **deterministic** as they will produce the same results as many times as you call them.

Before we start to explain how AES works, we briefly mention that AES sees the **state** of the plaintext, during the encryption process, as a 4 by 4 square of octets like so:

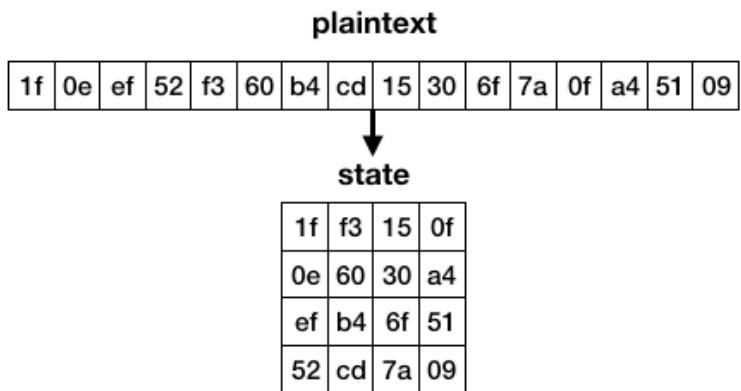


Figure 4.6 When entering the AES algorithm, a plaintext of 16 bytes gets transformed into a 4x4 square. This state is then encrypted, and finally transformed into a 16-byte ciphertext.

This doesn't really matter in practice, but this is how AES is defined. Under the hood, AES works like many similar symmetric cryptographic primitives called **block ciphers** (as they are ciphers that encrypt fixed-sized blocks).

NOTE

In technical terms, a block cipher with a key is a **permutation**: it maps all the possible plaintexts to all possible ciphertexts (see example in figure 2.13). Changing the key changes that mapping. A permutation is also reversible: from a ciphertext, you have a map back to its corresponding plaintext (otherwise decryption wouldn't be very useful).

```
00000000000000000000000000000000 → acc4822dc42346f92e1ec2695340c1b3  
00000000000000000000000000000001 → d488cf87bfb125021cdab6663b08ed19  
00000000000000000000000000000002 → 21e0b72510072a2990da839b046b66ad  
...  
c7c0ebb53f3d3d12acb947b4bcd1d8b5 → 79d38411a4a2c4e1f7bf27741d4507a8  
...  
ffffffffffffffffff → 77068a7ed1cce1371f75ad4dbc2484d3
```

Figure 4.7 A cipher with a key can be seen as a permutation: it maps all the possible plaintexts to all the possible ciphertexts.

Of course, we do not have the room to list all the possible plaintexts and their associated ciphertexts (that would be 2^{128} mappings for a 128-bit block cipher). Instead, we design constructions like AES which behave like permutations. We say that they are **pseudo-random permutations (PRPs)**.

AES has a round function that it iterates several time starting on the original input (the plaintext).

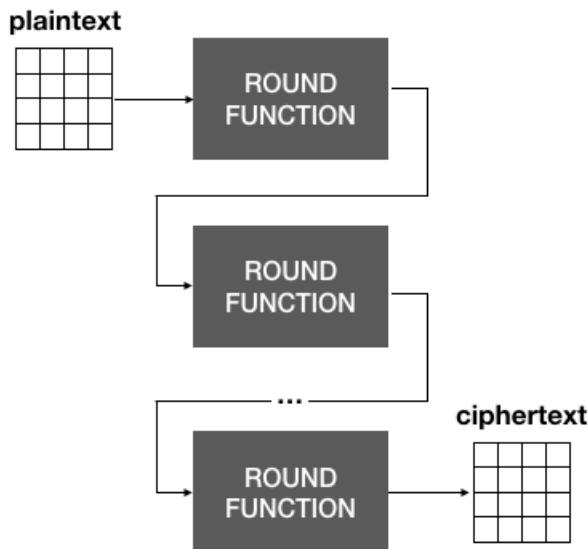


Figure 4.8 AES iterates a round function over a state in order to encrypt it. The round function takes several arguments including a secret key. This is missing from the diagram for simplicity.

Each call to the round function transforms the state further, eventually producing the ciphertext. To do that, the round function of AES takes:

- The **current state**: either the plaintext, or the output of the previous round function.
- The **round number**: there are 10 rounds in AES-128, 12 rounds for AES-192, and 14 rounds in AES-256.
- The relevant **round key**: each round uses a different round key, created from the main symmetric key.



Figure 4.9 The round function of AES at the end of round 2. It takes the previous state as input (the plaintext for round 1), the round number (3 in the example) and the associated round key. It outputs the new state, that will either become the input of a subsequent round, or converted back into a ciphertext if this was already the final round.

The round function is comprised of multiple operations that mix and transform the octets of the state. The round function of AES specifically makes use of 4 different subfunctions. While we will shy away from explaining exactly how they work (you can find this information in any book treating about AES), they are SubBytes, ShiftRows, MixColumns and AddRoundKey. The first three are easily reversible (you can find the input from the output of the operation) but the last one is not, as it exclusive or (XOR) the round key with the state (and thus needs the knowledge of the round key to be reversed). The succession of these subfunctions allows a round function to slowly transform a state into something else.

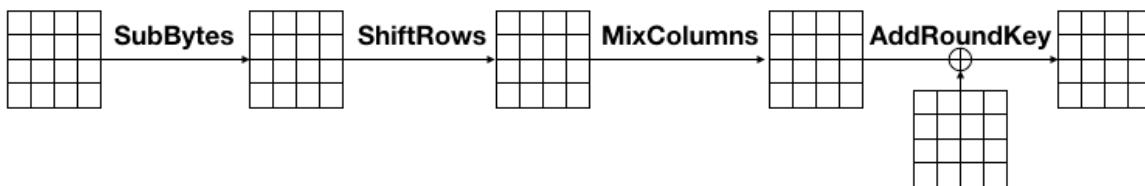


Figure 4.10 A typical round of AES. (The first and last rounds omit some operations.) 4 different functions transform the state. They are all reversible as it is required for decryption. The addition sign inside a circle is the symbol for the XOR operation.

The number of iteration of the round function in AES was chosen to thwart cryptanalysis, which are usually practical on a reduced number of rounds. For example on AES-128, very efficient total breaks (attacks that recover the key) exist on 3 rounds. By iterating many times, the cipher transforms a plaintext into something that looks nothing like the original plaintext. The slightest change in the plaintext also gives out a completely different ciphertext.

Another aspect of AES is that each round uses a different round key. For AES-128, with 10

rounds, we need 11 round keys which are derived from the main symmetric key via a **key schedule**. This allows any change in the bits of a key to give a completely different AES cipher. This key schedule can be performed at the very beginning, or step-by-step during the encryption process.

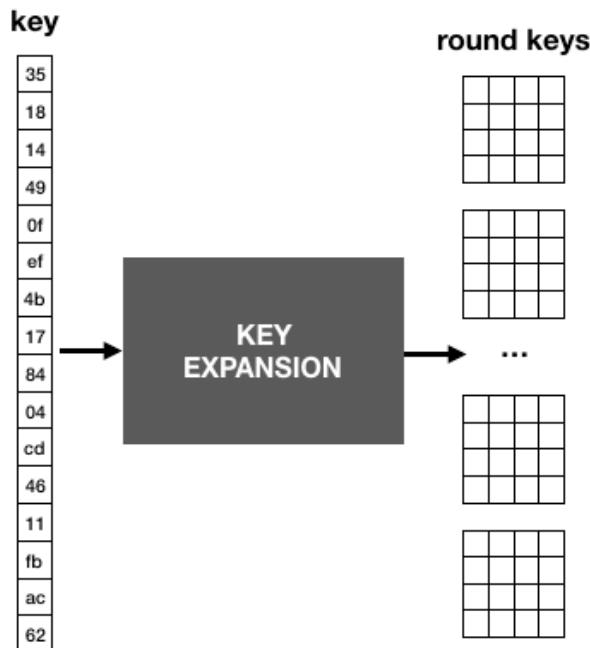


Figure 4.11 The key schedule of AES transforms a 16-byte or 32-byte key into 11 round keys for AES-128, 13 keys for AES-192, and 15 keys for AES-256.

For now, we have looked into the AES cipher, and we have noticed that it has quite a basic interface. Indeed, it only allows us to encrypt a block of 128-bit. How do we encrypt more? Or how do we encrypt less? The next section will answer this question.

NOTE Often, real-world cryptographic algorithms are compared with the security, size and speed they provide. We've already talked about the security of AES (depending on the key it can provide 128-bit, 192-bit or 256-bit of security), and the fact that it is an 128-bit block cipher. Speed-wise: AES has been implemented in hardware by many CPU vendors: **AES-New Instructions (AES-NI)** is a set of instruction implemented by Intel and AMD CPUs that can be used to encrypt and decrypt with AES. These special instructions make AES extremely fast in practice.

4.3.2 Mode of operation and integrity: How AES-CBC-HMAC works

Now that we have introduced the AES cipher and explained a bit of its internals, let's see how to use it in practice. In order to encrypt something that is not exactly 128 bits, a **padding** as well as a **mode of operation** are commonly used.

Padding. First, the plaintext must be a multiple of 128 bits (or 16 octets). To do this, we "pad" it

with octets until it turns into a multiple of 16 octets. Because most of the time we encrypt multiples of octets (for example 8 bits, or 16 bits), and not multiple of bits (for example 3 bits), paddings are usually defined in octets. There are several ways to specify a padding, but importantly it must be reversible: once a ciphertext is decrypted, one should be able to deterministically remove the padding in order to find out the original message. (For example, adding 0 octets wouldn't work, as you wouldn't be able to know if they were part of the original message or not.)

The most known padding mechanism is often referred to as "PKCS#7 padding", because it appeared in the famous **Public Key Cryptography Standard** document number 7⁹ (initially published by the RSA company). It works as follow: add as many octets necessary to make the plaintext a multiple of 16 octets, set the value of each of these octets to the length of the padding.

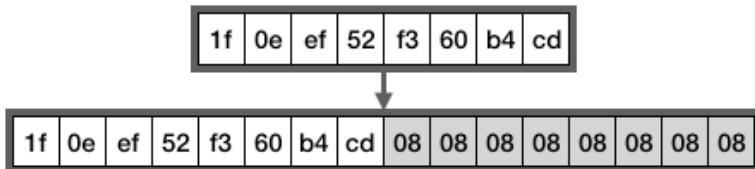


Figure 4.12 If a plaintext is not a multiple of the block size, it is padded with the length needed to reach a multiple of the block size. In the example, the plaintext is 8 bytes, so 8 more bytes (containing the value 8) are used to pad the plaintext up to the 16 bytes required for AES.

To remove the padding, one can easily check what is the value of the last octet of plaintext, and interpret it as a length to remove. If the plaintext is already a multiple of 16 octets, add a full block (16 octets in AES) of padding.

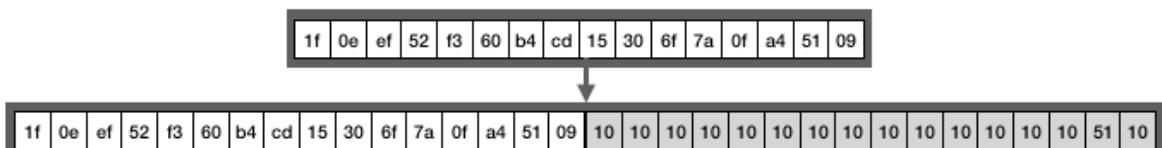


Figure 4.13 If a plaintext is already a multiple of the block size, it still needs to indicate to the decryption how much padding to remove. For this reason a whole new block of padding is inserted. (Remember, 0x10 in hexadecimal represents the number 16).

Mode of Operation. Second, a mode of operation is used to encrypt multiple blocks of 128-bits (if needed) as well as "randomizing" the encryption. The latter is useful because as we've previously mentioned, encryption with AES is **deterministic** and leaks information when the same plaintext is encrypted twice.

There exist many mode of operations, but the most popular one for AES is the **Cipher Block Chaining** (CBC). CBC works for any deterministic block cipher (not just AES) by taking an additional value called an Initialization Vector (IV) to randomize the encryption. Because of this, the IV is the length of the block size (16 bytes for AES) and must be random and unpredictable.

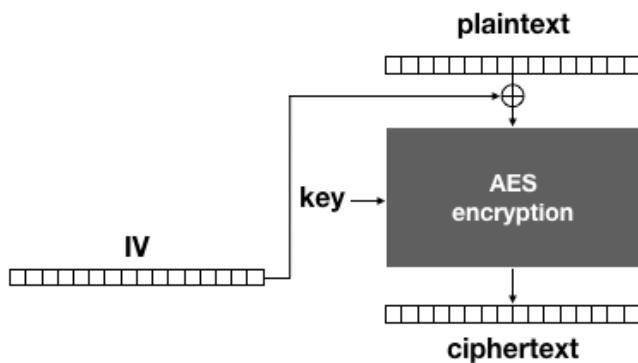


Figure 4.14 The CBC mode of operation with AES. To start encrypting the first block of plaintext, a random Initialization Vector (IV) is generated and XORed with the block prior to encrypting it.

To encrypt with the CBC mode of operation, start by generating a random IV of 16-byte (we will see in chapter 8 how to do this), then XOR the generated IV with the first 16-byte of plaintext before encrypting them. This effectively randomize the encryption. Indeed, if the same plaintexts is encrypted twice but with different IVs, the mode of operation will render two different ciphertexts.

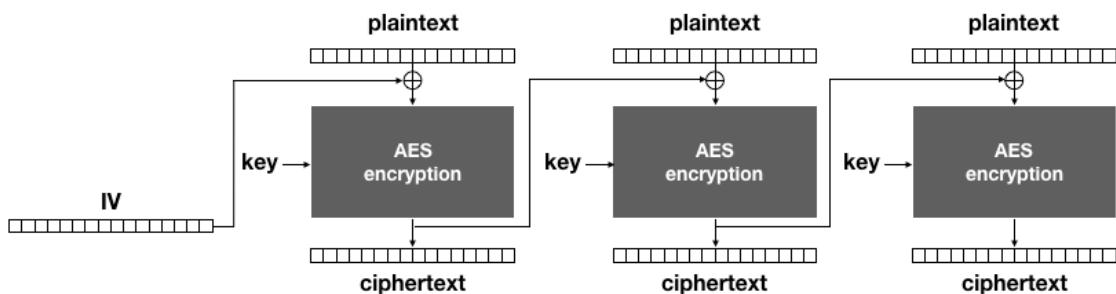


Figure 4.15 The CBC mode of operation with AES. To encrypt, a random Initialization Vector (IV) is used in addition to a padded plaintext (split in multiple blocks of 16 bytes).

If there is more plaintext to encrypt, use the previous ciphertext (like we used the IV previously) to XOR it with the next block of plaintext before encrypting it. This effectively randomizes the next block of encryption as well. Remember, the encryption of something is unpredictable and should be as good as the randomness we used to create our real IV.

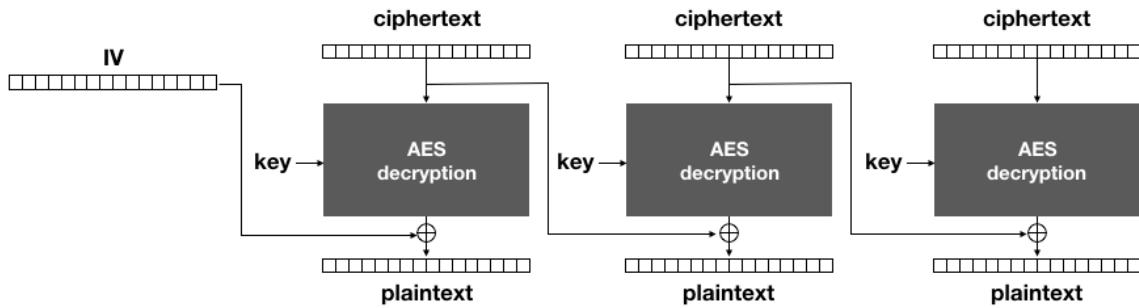


Figure 4.16 The CBC mode of operation with AES. To decrypt, the associated Initialization Vector (IV) is required.

To decrypt with the CBC mode of operation, reverse the operations. the IV is needed. For this reason, it must be transmitted in the clear along with the ciphertext. As it is supposed to be random, no information is leaked by this public value.

NOTE

additional parameters, like IVs, are prevalent in cryptography. Yet, they are often poorly understood and a great source of vulnerabilities. With the CBC mode of operation, an IV needs to be **unique** (it cannot repeat), as well as **unpredictable** (it really needs to be random). These requirements can fail for a number of reasons. Developers sometimes mistakenly set IVs to always be zero (which has led some cryptographic libraries to remove the possibility to specify an IV when encrypting with CBC) or to be predictable. When this happens, the encryption becomes deterministic again and a number of clever attacks become possible, this was the case with the famous BEAST attack on the TLS protocol.¹⁰. Note that **other algorithms might have different requirements for IVs**. this is why it is always important to read the manual. Dangerous details lie in the fine prints.

Nonetheless, we have failed to address one fundamental flaw so far: the ciphertext as well as the IV (in the case of CBC) can be modified by an attacker, changing the way that they are later decrypted. Many simple attacks exist against CBC and other modes of operations if left unprotected, where an attacker can targetedly switch bits of a message by switching bits in a ciphertext and its associated IV. This is illustrated in figure 4.16.

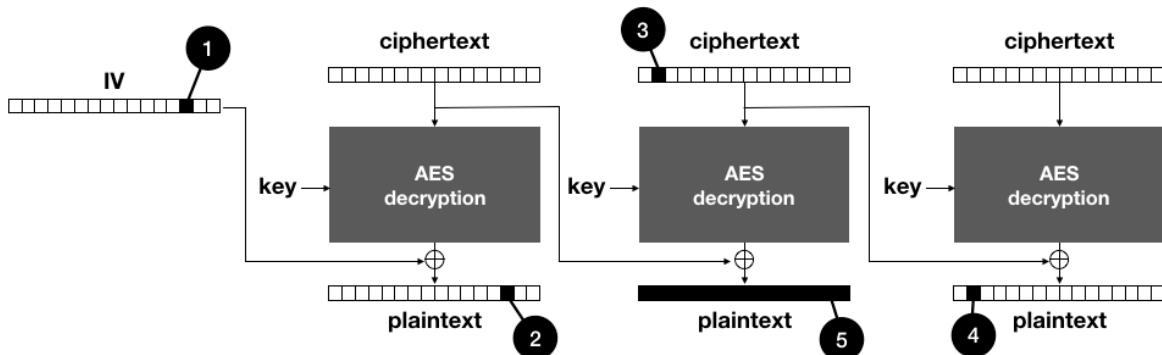


Figure 4.17 An attacker that can intercept an AES-CBC ciphertext can do the following: (1) Since the IV is public, flipping a bit (from 1 to 0 for example) of the IV also (2) flips a bit of the first block of plaintext. (3) Modifications of bits can happen on the ciphertext blocks as well (4) which are reflected on the next block of decrypted plaintext. (5) Note that tampering with the ciphertext blocks has the direct effect of scrambling the decryption of that block.

The missing element is **integrity**. AES-CBC cannot be used as-is because it lacks integrity protection. This property ensures that a ciphertext and its associated parameters (here the IV) cannot be modified without triggering some alarms. In order to add integrity protection, the **Message Authentication Codes (MACs)** that we've seen in chapter 3 can be used!

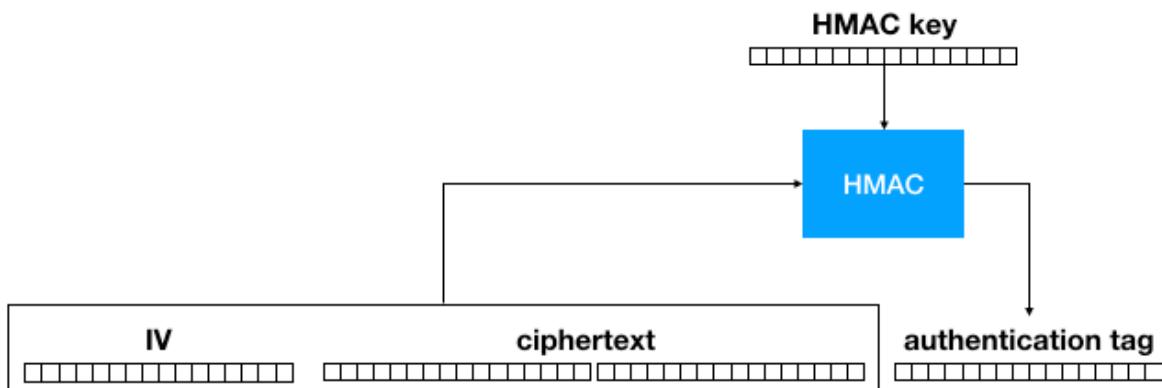


Figure 4.18 The AES-CBC-HMAC construction produces three arguments that are usually concatenated in the following order: the public IV, the ciphertext, the authentication tag.

For AES-CBC, **HMAC** (with the **SHA-256** hash function) is usually used to provide integrity. For security reasons, it is applied after padding the plaintext and encrypting it, and over both the ciphertext and the IV (otherwise an attacker can still modify the IV without being caught). The authentication tag created can be transmitted along with the IV and the ciphertext. Usually, all are concatenated together as illustrated in figure 4.17. In addition, it is best practice to use different keys for AES-CBC and HMAC in order to provide **domain separation**.

NOTE This construction is called **Encrypt-then-MAC**. The alternatives (like MAC-then-Encrypt) can sometimes lead to clever attacks (like Vaudenay's padding oracle attack)¹¹ and are thus not recommended in practice.

Prior to decryption, the tag needs to be verified in **constant-time** — meaning the time it takes to verify an authentication tag has to always be the same. It is a bit tricky to implement, but defends in situations where an attacker can measure the time it takes for you to figure out if a tag is invalid or not. This information is usually enough to compute the correct tag.¹²

The combination of all of these algorithms is referred to as **AES-CBC-HMAC** and has been one of the most widely used authenticated encryption mode, up until more recent all-in-one constructions were adopted.

WARNING AES-CBC-HMAC is not the most developer-friendly construction, it is often poorly implemented and have some dangerous pitfalls when not used correctly (for example the IV of each encryption must be unpredictable). I have spent a few pages introducing this algorithm as it is still widely used, and still works, but I recommend against using it in favor of the more recent constructions I will introduce next.

4.4 Authenticated Encryption with Associated Data (AEAD)

The history of encryption is not pretty. Not only has it been poorly understood that encryption without authentication is dangerous, but mis-applying authentication has also been a systemic mistake made by developers. For this reason a lot of research has emerged in trying to standardize all-in-one constructions that simplify the use of encryption for developers. In the rest of this section I introduce this new concept as well as two widely adopted standards: AES-GCM and ChaCha20-Poly1305.

4.4.1 What is an AEAD?

The most current way of encrypting data is to use an all-in-one construction called **Authenticated Encryption with Associated Data** (AEAD). The construction is extremely close to what AES-CBC-HMAC provides as it also offers confidentiality of your plaintexts while detecting any modifications that could have occurred on the ciphertexts. What's more, it provides a way to authenticate **Associated Data** (AD):

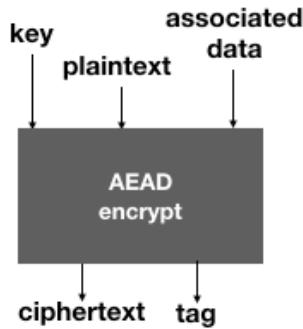


Figure 4.19 The interface of Authenticated Encryption with Associated Data (AEAD). To encrypt, an AEAD works in the same way as an encryption algorithm, with the addition of an Associated Data string. This new argument is not encrypted, but authenticated by the authentication tag appended to the ciphertext.

This **associated data** argument is optional and can be empty, or it can also contain metadata that is relevant to the encryption and decryption of the plaintext. This data will not be encrypted, and is either implied or transmitted along with the ciphertext.

In addition, the ciphertext's size is larger than the plaintext, as it now contains an additional authentication tag (usually appended at the end of the ciphertext).

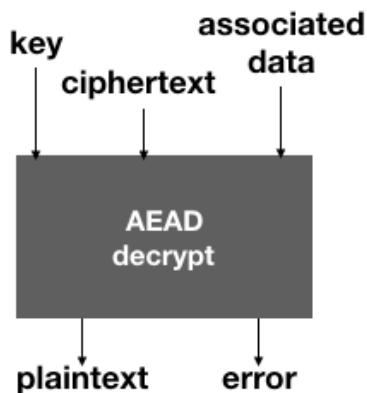


Figure 4.20 The interface of Authenticated Encryption with Associated Data (AEAD). To decrypt, an AEAD works in the same way as a decryption algorithm, with the additions of an Associated Data string and an authentication tag. These new arguments are not encrypted. If the ciphertext was modified, or the associated data different from what was used during encryption, the decryption returns an error.

Once we decrypt the ciphertext, we are required to use the same implied or transmitted **associated data**. The result is either an error, indicating that the ciphertext was modified in transit, or the original plaintext after decryption. This flow is illustrated in figure 4.20.

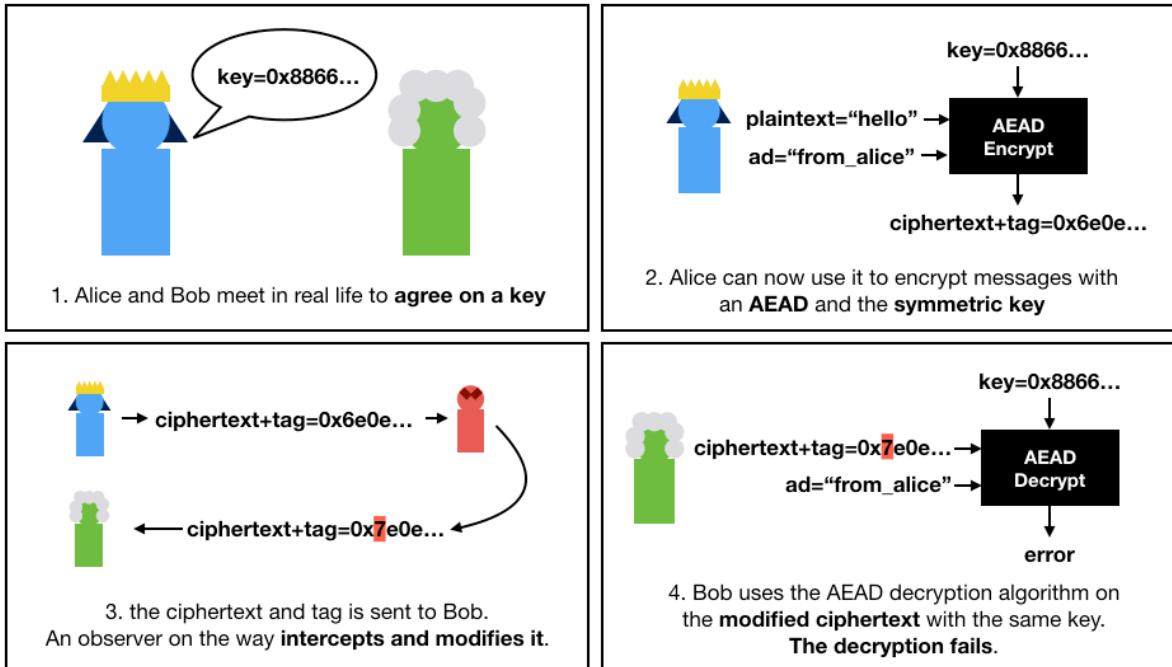


Figure 4.21 Both Alice and Bob meet in-person to agree on a shared key (top left). Alice (top right) can then use an AEAD encryption algorithm with the shared key to encrypt her messages to Bob. She can optionally authenticate some additional data (ad), here the sender of the message. After receiving the ciphertext and the authentication tag, Bob (bottom right) attempts to decrypt it using an AEAD decryption algorithm, the shared key, and the same additional data (in our example: who he thinks sent the message). If the additional data is incorrect, or the ciphertext was modified in transit, the decryption will fail.

4.4.2 The AES-GCM AEAD

The most widely used AEAD is AES with the Galois/Counter Mode also abbreviated **AES-GCM**. It was designed for high performance, by taking advantage of hardware support for AES and by using a MAC (GMAC) that can be implemented efficiently via specific CPU instructions like `PCLMULQDQ`.¹³

AES-GCM is included in NIST's Special Publication 800-38D since 2007,¹⁴ is FIPS approved,¹⁵ and was also part of NSA's suite B (a suite of algorithms that can be used by the US government to protect unclassified and classified information). It is the main cipher used in many protocols, including several versions of the TLS protocol which is used to secure connections to websites on the Internet. Effectively, we can say that AES-GCM encrypts the web.

AES-GCM combines the Counter Mode (CTR) mode of operation, with the GMAC message authentication code. Let's see how CTR with AES works first:

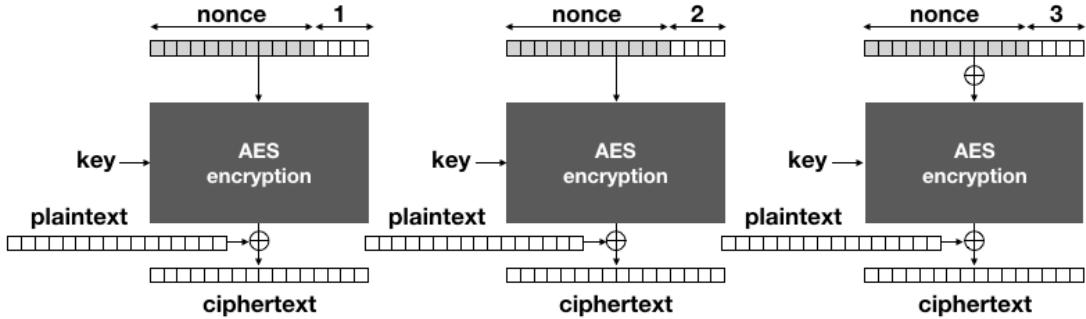


Figure 4.22 AES-CTR. A unique nonce is concatenated with a counter and encrypted to produce a "keystream". The keystream is then XORed with the actual bytes of the plaintext.

As you can see in the diagram, AES-CTR uses AES to encrypt a **nonce** concatenated with a number (starting at 1) instead of the plaintext. This additional argument, a **nonce** for "number once", serves the same purpose as an IV: it allows the mode of operation to randomize the AES encryption. The requirements are a bit different from the IV of CBC mode: a nonce needs to be **unique**, but not unpredictable. Once this 16-byte block is encrypted, the result is called a **keystream** and it is XORed with the actual plaintext to produce the encryption.

NOTE

Like IVs, **nonces** are a common term in cryptography, and they are found in different types of primitives. They can have very different requirements, although the name often indicates that it should not repeat. But as usual, what matters is what the manual said, not what the name of the argument implies. Indeed, the nonce of AES-GCM is sometimes referred to as an IV.

In the diagram, the nonce is 96-bit (12-byte) and takes most of the 16-byte to be encrypted. The 32 bits (4 bytes) left serve as a counter, starting from 1 and incremented for each block encryption until it reaches its maximum value at $2^{4*8}-1 = 4,294,967,295$. This means that at most 4,294,967,295 blocks of 128-bit can be encrypted with the same nonce (so less than 69 gigabytes).

If the same nonce is used twice, the same keystream is created and by XORing the two ciphertexts together one recovers the XOR of the two plaintext. This can in practice be devastating if you know some of one of the plaintext.

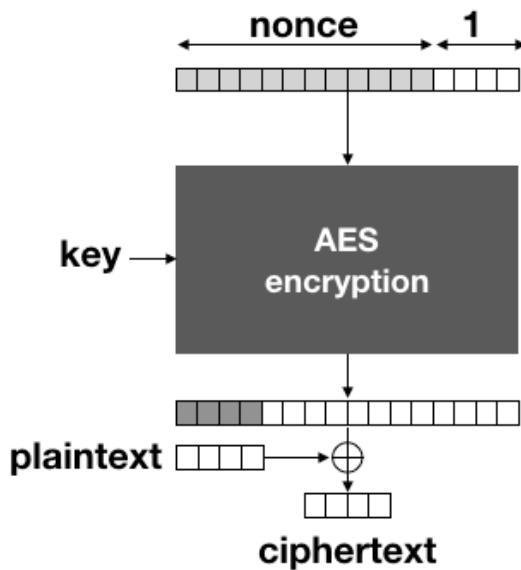


Figure 4.23 If the keystream of AES-CTR is longer than the plaintext, it is truncated to the length of the plaintext prior to XORing it with the plaintext. This permits AES-CTR to work without a padding.

An interesting aspect of Counter Mode is that no padding is required. We say that it turns a block cipher (AES) into a **stream cipher**: it encrypts the plaintext bytes by bytes.

NOTE

Stream ciphers are another category of ciphers. They are different than **block ciphers** because they can be used directly to encrypt a ciphertext by XORing it with a keystream. No need for a padding or a mode of operation, allowing the ciphertext to be of the same length as the plaintext. In practice, there isn't much difference between these two categories of ciphers since block ciphers can easily be transformed into stream ciphers via the CTR mode of operation. But in theory, block ciphers have the advantage that they can be useful to construct other categories of primitives, similar to what you've seen in chapter 2 with hash functions. This is also a good moment to note that by default encryption doesn't (or badly) hide the length of what you are encrypting. Because of this, the use of compression before encryption can lead to attacks if an attacker can influence parts of what is being encrypted.¹⁶

The second part of AES-GCM is **GMAC**. It is a MAC constructed from a keyed-hash **GHASH**.

In technical terms, GHASH is an almost-XOR universal hash (AXU) also called a difference unpredictable function (DUF). The requirement of such a function is weaker than a hash. For example, an AXU does not need to be collision resistant. Thanks to this, GHASH can be significantly faster.

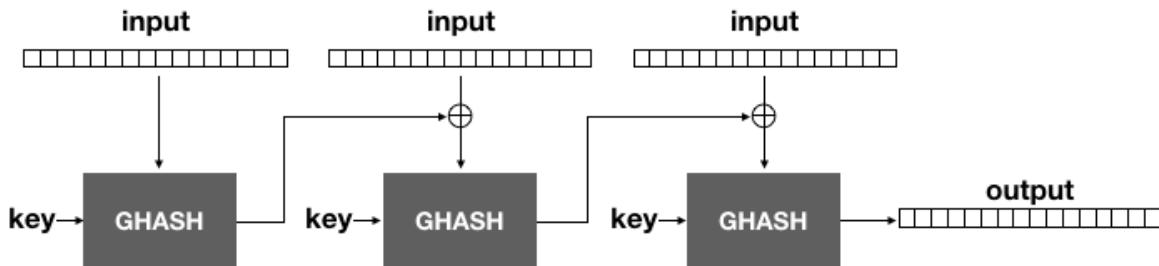


Figure 4.24 GHASH takes a key, and absorbs the input block by block in a manner resembling CBC mode. It produces a digest of 16 bytes.

To hash something with GHASH, the input is broken in blocks of 16-bytes and hashed in a similar way as CBC mode. As this hash takes a key as input, it can theoretically be used as a MAC, but only once — It's a one-time MAC.

As this is not ideal for us, we use a technique due to Wegman-Carter to transform GHASH into a many-time MAC.

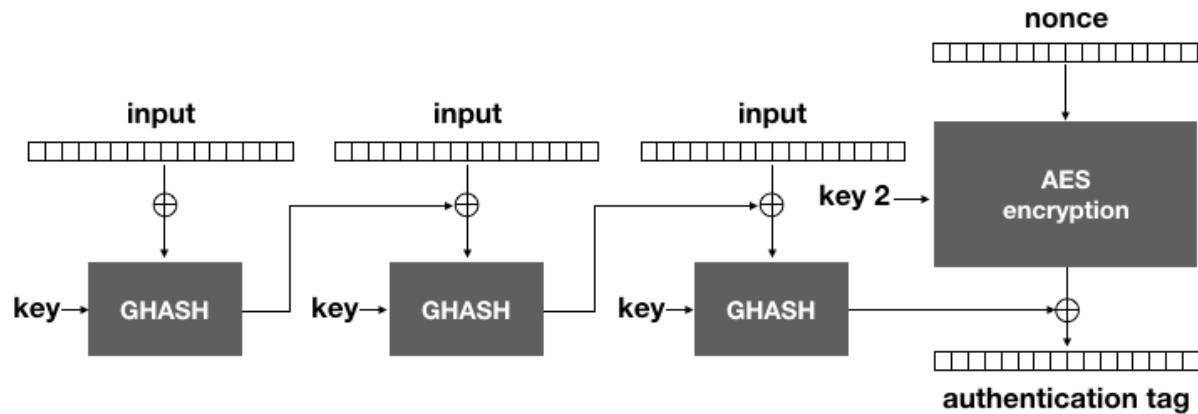


Figure 4.25 GMAC uses GHASH with a key to hash the input, then encrypts it with a different key and AES-CTR to produce an authentication tag.

GMAC is effectively the encryption of the GHASH output with AES-CTR (and a different key). The nonce must be **unique**, otherwise clever attacks can recover the authentication key used by GHASH (which would be catastrophic and would allow easy forgery of authentication tags).¹⁷

Finally, **AES-GCM** can be seen as an intertwined combination of CTR mode and GMAC, similar to the Encrypt-then-MAC construction we've previously discussed.

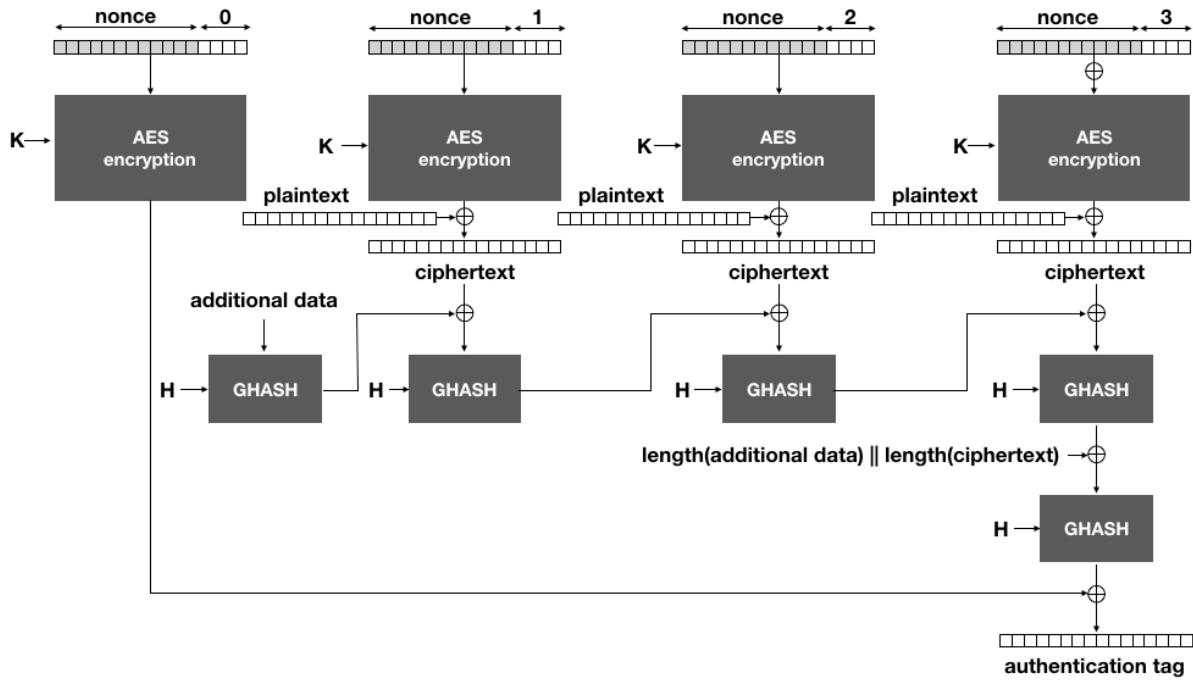


Figure 4.26 AES-GCM works by using AES-CTR with a symmetric key K to encrypt the plaintext, and using GMAC to authenticate the associated data and the ciphertext using an authentication key H .

The counter starts at 1 for encryption (unlike AES-CTR that starts its counter at 0), leaving the 0 counter for encrypting the authentication tag created by GHASH. GHASH in turns take an independent key H which is the encryption of the all-zero block with a key κ . This way one does not need to carry two different keys, as the key κ suffices to derive the other one.

As I've said previously, the 12-byte nonce of AES-GCM needs to be unique, and thus to never repeat. Notice that it doesn't need to be random, consequently some people like to use it as a **counter**, starting it at 1 and incrementing it for each encryption. In this case, one must use a cryptographic library that lets the user to choose the nonce. This allows one to encrypt $2^{12*8}-1$ messages before reaching the maximum value of the nonce. Suffice to say, this is an impossible number of messages to reach in practice.

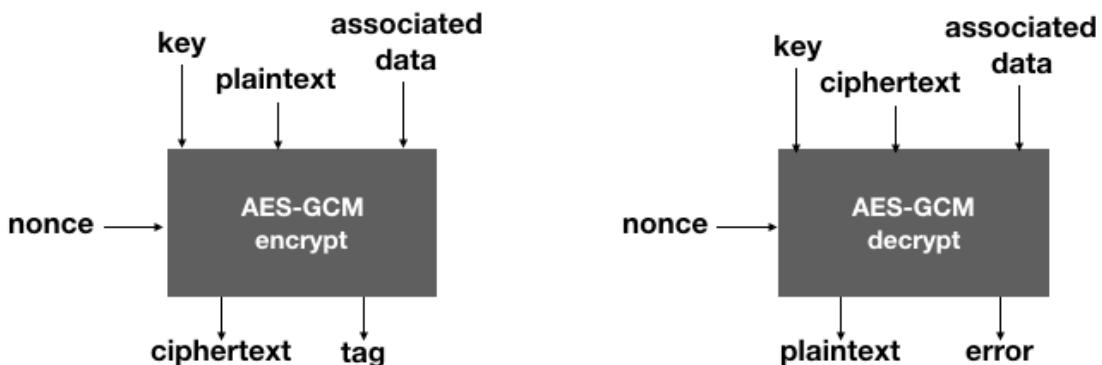


Figure 4.27 AES-GCM is a nonce-based AEAD. It provides the same interface as an AEAD but requires a nonce in order to work. The nonce must never repeat between encryptions.

On the other hand, having a counter means that one needs to keep state. If a machine crashes at the wrong time, it is possible that nonce-reuse could happen. For this reason, it is sometimes preferred to have a **random nonce**. Actually, some libraries will not let developers choose the nonce and will generate them at random. Doing this should avoid repetition with probabilities so high that they shouldn't happen in practice. Yet, the more messages are encrypted, the more nonces are used, and the higher the chances of getting a collision become. Because of the **birthday bound** we talked about in chapter 2, it is recommended not to encrypt more than $2^{92/3} \approx 2^{30}$ messages with the same key (when generating nonces randomly).

NOTE

2^{30} messages is quite a large number of messages. It might never be reached in many scenarios, but real-world cryptography often pushes the limit of what is considered reasonable. Some long-lived systems need to encrypt many many messages per seconds, eventually reaching these limits. Visa for example process 150 million transactions per day, if it needs to encrypt them with a unique key it would reach the limit of 2^{30} messages in only a week. In these extreme cases, re-keying (changing the key used to encrypt) can be a solution, other research exist to improve the maximum number of messages that can be encrypted with the same key.¹⁸

4.4.3 Chacha20-Poly1305

The second AEAD I will talk about is **Chacha20-Poly1305**. It is the combination of two algorithms, the Chacha20 stream cipher and the Poly1305 MAC, both designed separately by Daniel J. Bernstein. Both algorithms were designed to be fast in software, contrary to AES which is slow when hardware support is unavailable. In 2013 Google standardized the Chacha20-Poly1305 AEAD,¹⁹ in order to make use of it in Android mobile phones relying on low-end processors. Nowadays it has been widely adopted by Internet protocols like OpenSSH, TLS and Noise.

Chacha20 is a modification of the **Salsa20** stream cipher,²⁰ which was originally designed by Daniel J. Bernstein around 2005. It was one of the nominated algorithms in the ESTREAM competition.²¹

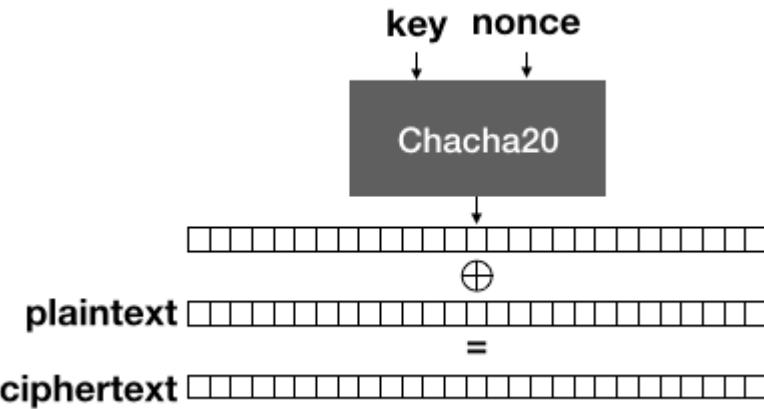


Figure 4.28 Chacha20 works by taking a symmetric key and a unique nonce. It then generates a keystream that is XORed with the plaintext to produce the ciphertext. The encryption is length-preserving as the ciphertext and the plaintext are the same length.

Like all stream cipher, the algorithm produces a **keystream** — a series of random bytes of the length of the plaintext. It is then XORed with the plaintext to create the ciphertext.

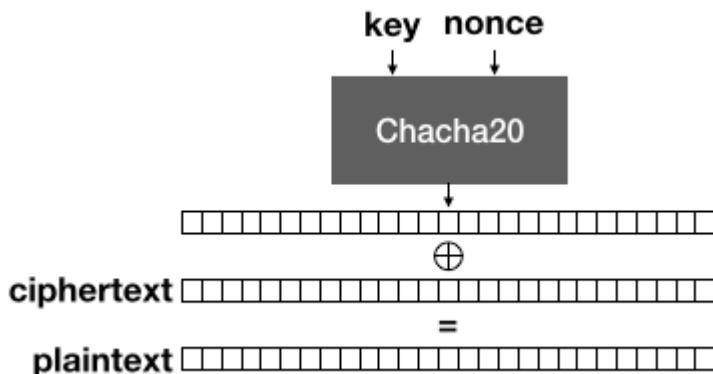


Figure 4.29 Chacha20 decryption looks very similar to the encryption process, thanks to how the XOR operation works.

To decrypt, the same algorithm is used to produce the same keystream, which is XORed with the ciphertext to give back the plaintext.

Under the hood, this is how Chacha20 generate a keystream:

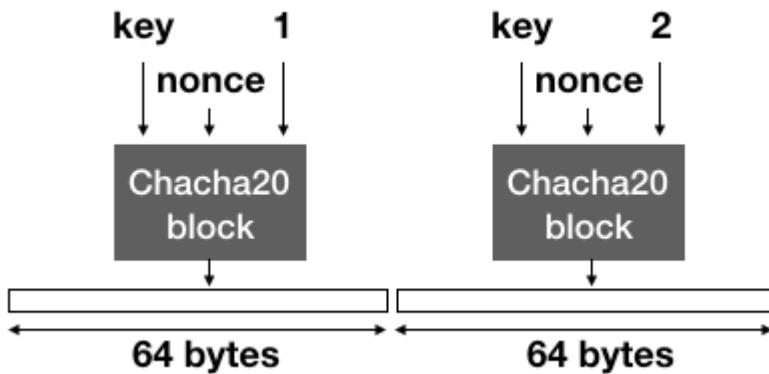


Figure 4.30 Chacha20’s keystream is created by calling an internal block function until enough bytes are produced. One block function call creates 64 bytes of random keystream.

Via its core **block function**, Chacha produces a keystream of **64 bytes** by taking:

- A 256-bit (32-byte) key, like AES-256.
- A 92-bit (12-byte) nonce, like AES-GCM.
- A 32-bit (4-byte) counter, like AES-GCM.

For this reason, we can infer that Chacha20 can be used to encrypt as many messages as AES-GCM before its nonce runs into the same issues. Since the output created by this block function is much larger, the size of the messages that can be encrypted is also impacted: you can encrypt $2^{32} \times 64$ bytes ≈ 274 gigabytes-long messages. If a nonce is used twice to encrypt, similar issues to AES-GCM arise: an observer can obtain the XOR of the two plaintexts by XORing the two ciphertexts, and the authentication key for that nonce can be recovered. These are serious issues that can lead to an attacker being able to forge messages!

NOTE

The size of the nonces and the counters are actually not always the same everywhere (both for AES-GCM and Chacha20-Poly1305), but these are the recommended values from the adopted standards. Still, some cryptographic libraries will accept different size of nonces, and some applications will increase the size of the counter (or the nonce) in order to allow encryption of larger messages (or more messages). Note although that increasing the size of one component, necessarily decrease the size of the other. To prevent this, while allowing a large number of messages to be encrypted under a single key, other standards like XSalsa20²² and XChacha20²³ are available. These standards increase the size of the nonce while keeping the rest intact, which is important in cases where the nonce needs to be generated randomly instead of being a counter tracked in the system.

The process to encrypt is the same as with AES-CTR:

1. Run the block function, incrementing the counter every time, until enough keystream is produced.
2. Truncate the keystream to the length of the plaintext.
3. XOR the keystream with the plaintext.

Inside the Chacha20 block function, a state is formed as follow:

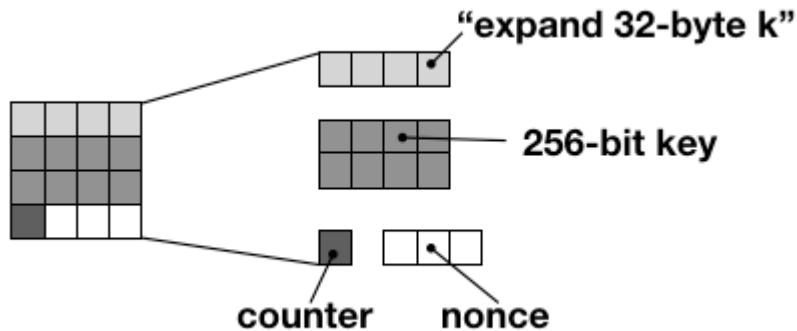


Figure 4.31 The state of Chacha20 block function. It is formed by 16 words (represented as squares) of 32 bytes each. The first line stores a constant, the second and third lines store the 32-byte symmetric key, the following word stores a 4-byte counter, and the last 3 words store the 12-byte nonce.

This state is then transformed into 64 bytes of keystream by iterating a round function 20 times (hence the 20 in the name of the algorithm). This is very similar to what was done with AES and its round function. The round function is itself calling a **Quarter Round** (QR) function 4 times per round, acting on different words of the internal state every time, depending if the round number is odd or even.

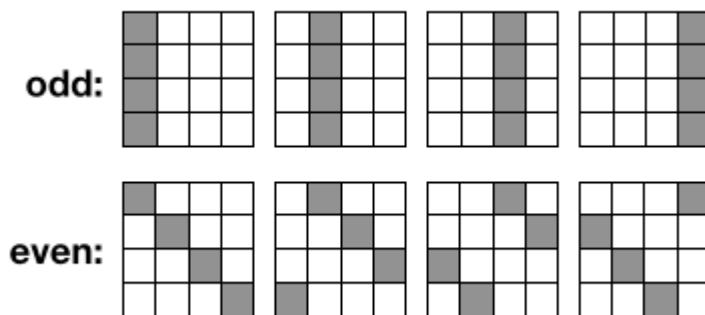


Figure 4.32 A round in Chacha20 affects all the words contained in a state. As the Quarter Round function only takes 4 arguments, it must be called at least 4 times on different words (greyed in the diagram) to modify all 16 words of the state.

The QR function takes 4 different argument, and updates them using only Add, Rotate and XOR

operations. We say that it is an **ARX** stream cipher. This makes Chacha20 extremely easy to implement and fast in software.

Poly1305 is a MAC created via the Wegman-Carter technique, very much like the GMAC we've previously talked about.

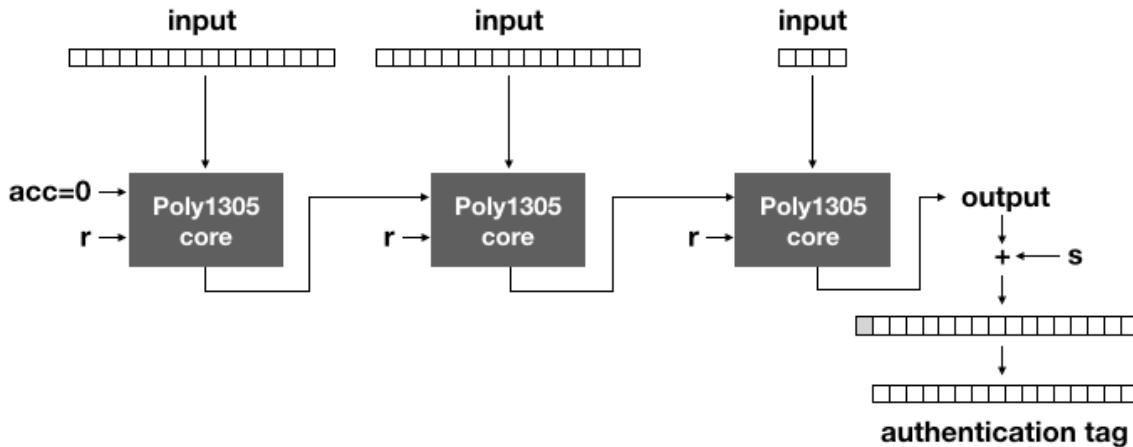


Figure 4.33 Poly1305's core function absorbs an input one block at-a-time by taking an additional accumulator set to 0 initially, and an authentication key r . The output is fed as accumulator to the next call of the core function. Eventually the output is added to a random value s to become the authentication tag.

Here, r can be seen as the authentication key of the scheme, like the authentication key \mathbb{H} of GMAC. And s makes the MAC secure for multiple uses by encrypting the result, thus it must be unique for each usage.

The **Poly1305 core function** mixes the key with the accumulator (set to 0 in the beginning) and the message to authenticate. The operations are simple multiplications modulo a constant \mathbb{P} .

NOTE

Obviously, a lot of details are missing from our description. I seldom mention how to encode data, or how some arguments should be padded before being acted on. These are all implementation specificities that do not matter for us as we are trying to get an intuition of how these things work.

Eventually, we can use Chacha20 and a counter set to 0 to generate a keystream and derive the 16-byte r and 16-byte s values we need for Poly1305. The resulting AEAD cipher looks like this:

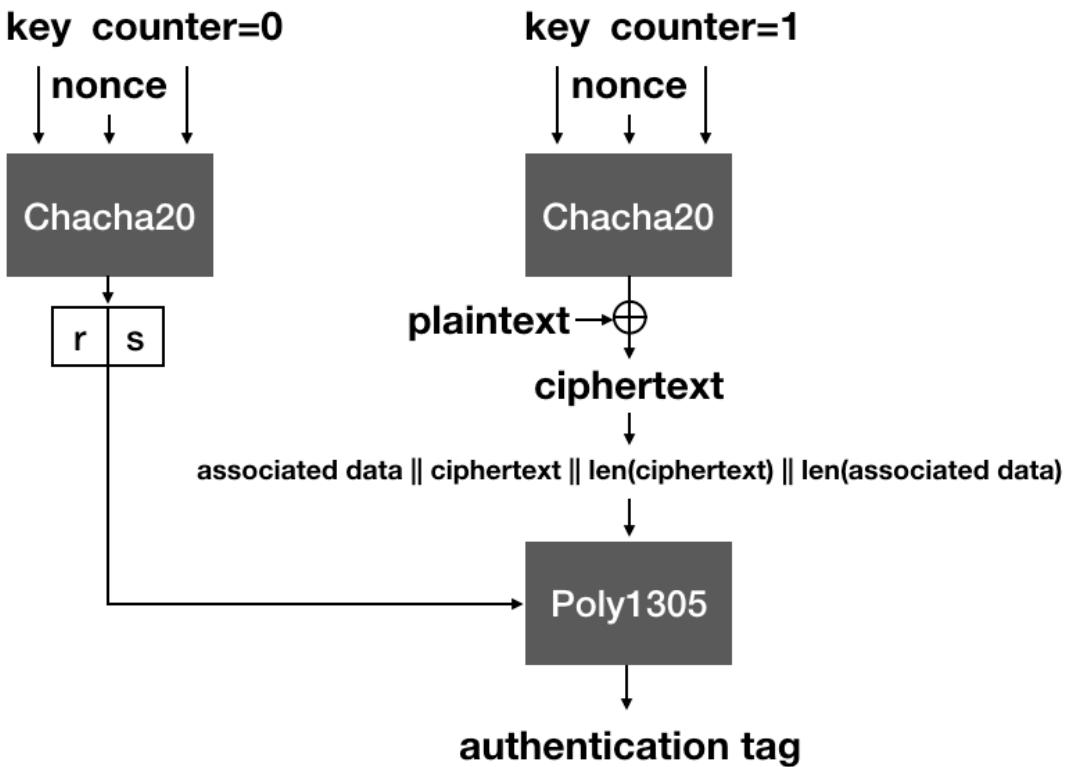


Figure 4.34 Chacha20-Poly1305 works by using Chacha20 to encrypt the plaintext, and to derive the keys required by the Poly1305 MAC. Poly1305 is then used to authenticate the ciphertext as well as the associated data.

The normal Chacha20 algorithm is first used to derive the authentication secrets r and s needed by Poly1305. The counter is then incremented, and Chacha20 is used to encrypt a plaintext. After that, the associated data and the ciphertext (and their respective lengths) are passed to Poly1305 to create an authentication tag.

To decrypt, the exact same process is applied. Chacha20 first verifies the authentication of the ciphertext and the associated data via the tag received. It then decrypts the ciphertext.

4.5 Key Wrapping and Nonce-Misuse Resistance

Now that you have learned about authenticated encryption, you could stop here, understand that you can now use AES-GCM or Chacha20-Poly1305 whenever you need to encrypt something, and move on to the next chapter. Unfortunately real-world Cryptography is not always about the agreed standard, it is also about constraints. Constraints in size, constraints in speed, constraints in format, etc. For this reason, we need to look at scenarios where these AEADs won't fit, and what solutions have been invented by the field of cryptography.

4.5.1 Wrapping Keys: How To Encrypt Secrets

One of the problems of the nonce-based AEADs we've discussed previously is that they all require a nonce, which takes additional space. In worse scenarios, the nonce to-be-used for encryption comes from an untrusted source and can thus lead to nonce repetition that would damage the security of the encryption. From these assumptions, it was noticed that encrypting a key might not necessarily need randomization, since what is encrypted is already random.

NOTE Encrypting keys is a useful paradigm in cryptography, and is used in a number of protocols as you will see in the second part of this book.

The most widely adopted standard is NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping.²⁴ It specifies two key wrapping algorithms based on AES: the AES Key Wrap (KW) mode and the AES Key Wrap With Padding (KWP) mode.

NOTE These two algorithms are often implemented in **Hardware Security Modules** (HSM). HSMs are devices that are capable of performing cryptographic operations while ensuring that keys they store cannot be extracted by physical attacks. That's at least if you're under a certain budget.

These key-wrapping algorithms do not take an additional nonce or IV, and randomize their encryption based on what they are encrypting. Consequently, they do not have to store an additional nonce or IV next to the ciphertexts.

4.5.2 AES-GCM-SIV and Nonce-Misuse Resistance Authenticated Encryption

In 2006, Rogaway published a new key-wrapping algorithm called Synthetic Initialization Vector (SIV).²⁵ In the white paper, Rogaway notes that the algorithm is not only useful to encrypt keys, but as a general AEAD algorithm as well that is resistant to nonce repetitions. As we've seen with AES-GCM and ChaCha20-Poly1305, repeating a nonce is catastrophic. It not only reveals the XOR of the two plaintexts, but it also allows an attacker to recover an authentication key and to forge more messages. In *Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS*,²⁶ a group of researchers found 184 HTTPS servers being guilty of re-using nonces, and demonstrated live attacks on these websites.

The point of a **nonce-misuse resistant** algorithm is that encrypting two plaintexts with the same nonce will only reveal if the two plaintexts are equal or not.

As you will see in chapter 8 on randomness, it is sometimes hard to obtain good randomness on

constrained devices or mistakes can be made. In this case, nonce-misuse resistant algorithms solve real problems.

In the rest of this section, I describe the scheme being standardized by Google — AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption.²⁷

The idea of AES-GCM-SIV is to generate the encryption and authentication keys separately via a main key, every time a message has to be encrypted (or decrypted). This is done by producing a keystream long enough with AES-CTR, the main key and a nonce:

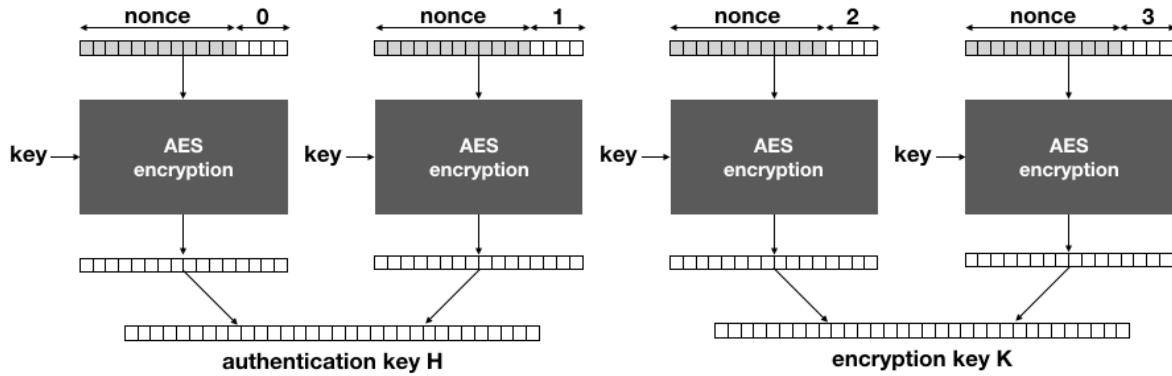


Figure 4.35 The main key of AES-GCM-SIV is used solely with AES-CTR to derive the encryption key K and the authentication key H .

Notice that if the same nonce is used to encrypt two different messages, the same keys will be derived here.

Next, AES-GCM-SIV authenticates the plaintext, instead of the ciphertexts as we have seen in the previous schemes. This creates an authentication tag over the associated data and the plaintext (and their respective lengths). Instead of GMAC, AES-GCM-SIV defines a new MAC called Polyval. It is quite similar and only attempts to optimize some of GMAC's operations.

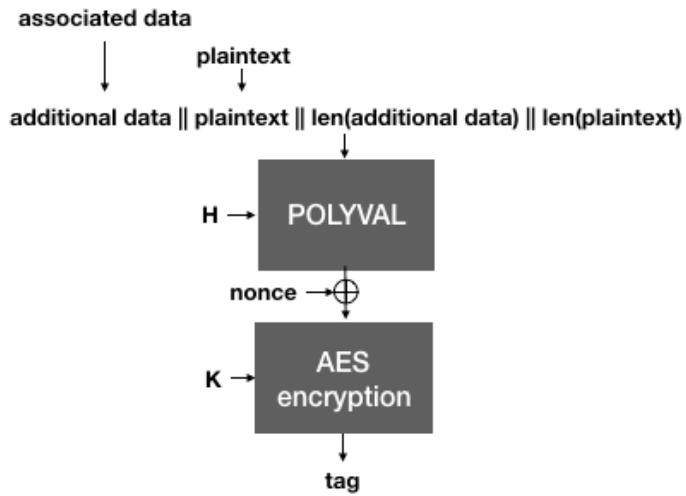


Figure 4.36 The Polyval function is used to hash the plaintext and the associated data. It is then encrypted with the encryption key K to produce an authentication tag.

Importantly, notice that if the same nonce is re-used, two different messages will of course produce two different tags. This is important because in AES-GCM-SIV, the tag is then used as a nonce to AES-CTR in order to encrypt the plaintext.

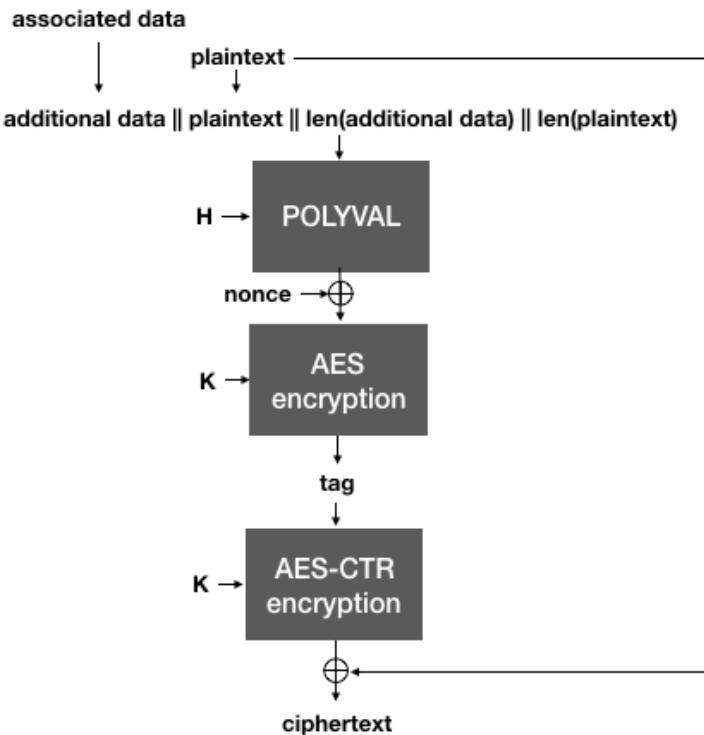


Figure 4.37 AES-GCM-SIV uses the authentication tag (created with Polyval over the plaintext and the associated data) as a nonce for AES-CTR to encrypt the plaintext.

This is the trick behind SIV: the nonce used to encrypt in the AEAD is generated from the plaintext itself, which makes it highly unlikely that two different plaintexts will end up being encrypted under the same nonce. To decrypt, the same process is done in reverse:

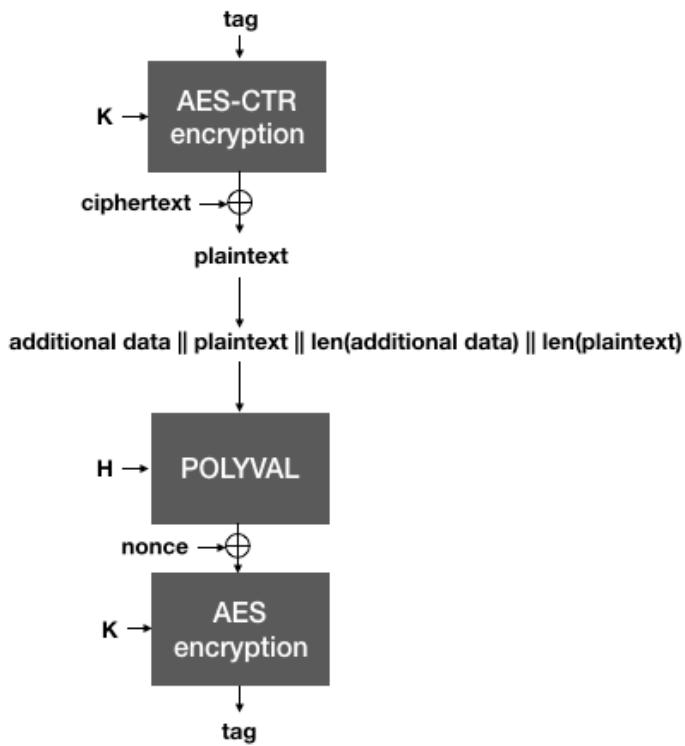


Figure 4.38 AES-GCM-SIV decrypts a ciphertext by using the authentication as a nonce for AES-CTR. The plaintext recovered is then used along with the associated data to validate the authentication tag. Both tags need to be compared (in constant-time) before releasing the plaintext to the application.

As the authentication tag is computed over the plaintext, the ciphertext must first be decrypted (using the tag as an effective nonce). Then, the plaintext must be validated against the authentication tag. Because of that, one must realize two things:

- The plaintext is released by the algorithm **before it can be authenticated**. For this reason it is extremely important that nothing is done with the plaintext, until it is actually deemed valid by the authentication tag verification.
- Since the algorithm works by decrypting the ciphertext (respectively encrypting the plaintext) as well as authenticating the plaintext, we say that it is *two-pass* — it must go over the data twice. Because of this, it usually is slower than its counterpart AES-GCM.

SIV is a powerful concept that can be implemented for other algorithms as well.²⁸

4.6 A Map of Authenticated Encryption

While we have covered the most popular and most widely adopted algorithms, there exist many others:

- **Broken Algorithms.** Obviously, we want to cover what works. Many algorithms like RC4 and DES used to be standards, but have been badly broken by new advances in cryptanalysis and were phased out in place of better researched and more modern alternatives (like AES).

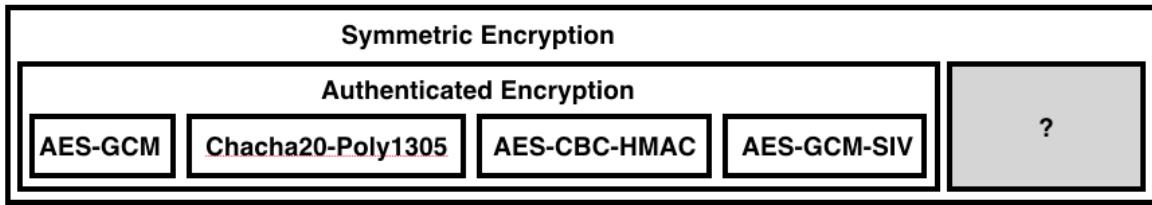
- **Legacy Algorithms.** We won't be talking about these, as this book is about good modern cryptography. Still, know that many deprecated algorithms are being in use today. For example, banking institutions still support and make use of 3DES (a symmetric encryption algorithm which was replaced by AES) which has seen many issues in the past.²⁹
- **Second-choice Algorithms.** Some algorithms are standardized and integrated in many protocols. For example, AES-CCM is in standardized in the Wi-Fi protocol WPA2, the TLS protocol, IPSec and Bluetooth Low Energy; yet, they are often second-choice algorithms (behind AEADs like AES-GCM) that most people do not use.
- **Forgotten Algorithms.** Many competitions have been organized in cryptography, as I write this the CAESAR competition (Competition for Authenticated Encryption: Security, Applicability, and Robustness)³⁰ just ended, nominating 6 candidates for its final portfolio: Ascon, ACORN, AEGIS-128, OCB, Deoxys-II, and COLM. No one knows what these algorithms will become, but past competitions like eSTREAM (2008)³¹ and NESSIE (2003)³² have been known to leave a trail of unused algorithms in their wake.
- **National ciphers.** AES and the other standardized algorithms we've talked about have been either designed, or standardized via the help of United States organizations. For this reason, some countries have advertised their reluctance to use them. National ciphers like GOST in Russia, or SM4 in China, do exist but are rarely seen in practice in the western side of the world. For the most part, and fortunately, most countries around the world agree on standards like AES-GCM.

Nonetheless, it is possible that in the future, different algorithms will be used. Recently a new category of devices have attracted the attention of cryptographers, little chips and embedded devices that have speed and size constraints. In 2018, the NIST organization launched a *call for submissions* for the **Lightweight Cryptography** project,³³ although it will be many years before we see any of the submissions becoming standards.

4.7 Other Kinds of Symmetric Encryption

Let's pause for a moment and recapitulate what we have seen so far:

- **Non-Authenticated Encryption.** It is AES with a mode of operation, but without a MAC. This is insecure in practice as ciphertexts can be tampered with.
- **Authenticated Encryption.** AES-GCM and Chacha20-Poly1305 being the two most widely adopted ciphers.
- **Key Wrapping Algorithms.** We have seen that several schemes provide authenticated encryption but without the use of nonces.
- **Nonce-Misuse Resistant Authenticated Encryption.** We have seen that AES-GCM-SIV does not fail catastrophically when a nonce is being re-used, at the cost of a slower cipher.



The last two are interesting, because they would not exist in a perfect world. Indeed, if we can generate good nonces and can afford to store a per-ciphertext nonce, then AES-GCM or Chacha20-Poly1305 are always good solutions. But real-world cryptography is often the cryptography of the imperfect. Harder constraints sometimes exist, and different areas of cryptography have been trying to provide solutions to these different settings:

Format Preserving Encryption (FPE). Historically, financial institutions and businesses have had a lot of credit card numbers to deal with, as well as large systems and applications in place to process, transport and store them. Once enough people were convinced that it was important to secure such numbers, regulations like PCI-DSS started enforcing encryption of personally identifiable information (PII) like credit card numbers. It was too late for many systems to include an additional IV and authentication tag, or to process different formats. In this imperfect situation, instead of leaving credit card numbers unencrypted, format preserving encryption (FPE) took over.³⁴

FPE schemes transform a plaintext into a ciphertext that looks like a plaintext. For example, a credit card of 16 digits would get encrypted to 16 other digits. In 2016, NIST released Special Publication 800-38G containing two format preserving encryption algorithms called FF1 and FF3.³⁵

Unfortunately these schemes do not provide integrity, as there is no authentication tag. Furthermore they were also the target of several attacks.³⁶ While the research is still recent, the previously discussed technical constraints are also becoming less and less important. For these reasons, it is possible that the field of FPE has a not-so bright future.

Disk Encryption. When you lock your computer and leave it unattended, it is possible that someone can physically access it for some period of time (or even steal the machine). If you do not have Full Disk Encryption (FDE) enabled, what is stored on the hard drive can simply be dumped and analyzed. If only you had enabled Bitlocker on Windows, Filevault on MacOS, maybe LUKS on Linux; then you wouldn't have had this problem. FDE effectively encrypts your hard drive, sector by sector, and decrypts it on-the-fly when a sector is requested during unlocked access to the computer.

Disk encryption is another one of these interesting constrained problems: in order to save space, (in-place) encryption of disk sectors cannot expand. Yet, authenticated encryption expands a

plaintext by 16 bytes (at least) because of the 16-byte authentication tag and the potential nonce that has to be kept around. In order to circumvent this issue one could use unauthenticated AES-CTR or AES-CBC or Chacha20. But without integrity protection, attacks that flip targeted bits of the plaintexts after decryption not only exist but are practical (for both AES-CTR and Chacha20, for example, flipping a bit in the ciphertext directly affects the same bit in the plaintext). An attacker could boot on a different OS, and modify specific bits of the encrypted hard drive, before running away. After that the user could boot the device normally and the exploit would run. So far it seems like the main research has been trying to solve the issue with **wide-block** ciphers. These ciphers have the particularity that they encrypt very large blocks at a time (one entire sector for example), and any attacks attempting to targetedly flip bits in the ciphertext would totally mess the plaintext. Such modifications would be so damaging, that it would highly likely crash the OS rather than allow an exploit to run. This has been called *poor-man's authentication*. Now, we could use some metadata like the sector number to encode a nonce. But an attacker who can rewrite a sector would be able recover the key stream for this sector by XORing the renewed sector's ciphertext with the plaintext. The solution is to let the plaintext influence what nonce will be used to encrypt it, much like AES-GCM-SIV, or by having it influencing the encryption algorithm itself (in which it would require a **tweakable cipher**).

In 2010 the NIST released Special Publication 800-38E,³⁷ which standardized the XTS-AES mode for usage in disk encryption solutions. To this day, most of the full disk encryption solutions make use of XTS-AES. The mode is length-preserving and protect against bit flippings to some degree: it only randomizes the 16-byte block where the bit was flipped.

In 2019 Google standardized Adiantum,³⁸ a wide-block cipher (or wide-block mode) making use of Chacha20, and deriving the nonce from the message and metadata about what sector is being encrypted. Adiantum is used in low-end android devices who cannot benefit from AES hardware support.

4.8 Summary

- Symmetric encryption allows two parties to protect the confidentiality of their communications via a shared secret.
- Symmetric encryption needs to be authenticated to be secure. The standard solution is to use an Authenticated Encryption with Associated Data (AEAD) which is an all-in-one authenticated encryption construction.
- AES-GCM and Chacha20-Poly1305 are the two most widely adopted AEADs, most applications nowadays use either one of them.
- Reusing nonces breaks the authentication of AES-GCM and Chacha20-Poly1305, schemes like AES-GCM-SIV are nonce-misuse resistant.
- real-world Cryptography is about constraints, and AEADs cannot always fit every scenarios. This is the case for disk encryption or database encryption for example, that require the development of new types of encryption.

5 Key exchanges

This chapter covers

- Key Exchanges and their security properties.
- The popular key exchange algorithms.
- How key exchanges are used in practice.

We are now entering the realm of public key cryptography with our very first asymmetric cryptographic primitive: the **key exchange**. Briefly, this primitive allows two peers to openly derive a shared secret. Keep on reading to learn more!

As I have hinted in the introduction of this book, there is much more math involved in asymmetric cryptography, and thus the next chapters are going to be a tad more difficult. Don't get discouraged, as what you will learn in this chapter will be helpful to understand the following chapters.

To understand exactly what real-world problems public key cryptography aims to solve, we need to take a step back and return to the symmetric cryptographic primitive you learned in chapter 4: **Authenticated Encryption**. Remember, it allows you to hide messages between participants, as long as they share a symmetric key.

Now imagine the following scenario: you are working for Google and need to setup two servers in different geo-locations. You want these two machines to be able to communicate with each other, although you don't really trust the network they will have to go through in order to do this. Using our simple authenticated encryption primitive, we can encrypt all communications between these two machines with something like AES-GCM (one of the authenticated encryption algorithm you've learned in chapter 4), and the problem seems to be solved. Of course you still need to **provision** (set up) these machines with a shared symmetric secret for this

protocol to work, but we'll assume it is not too hard for you to do this as there are only two machines after all. The problem arises when we increase the number of machines in this scenario, as can be seen in figure 5.1.

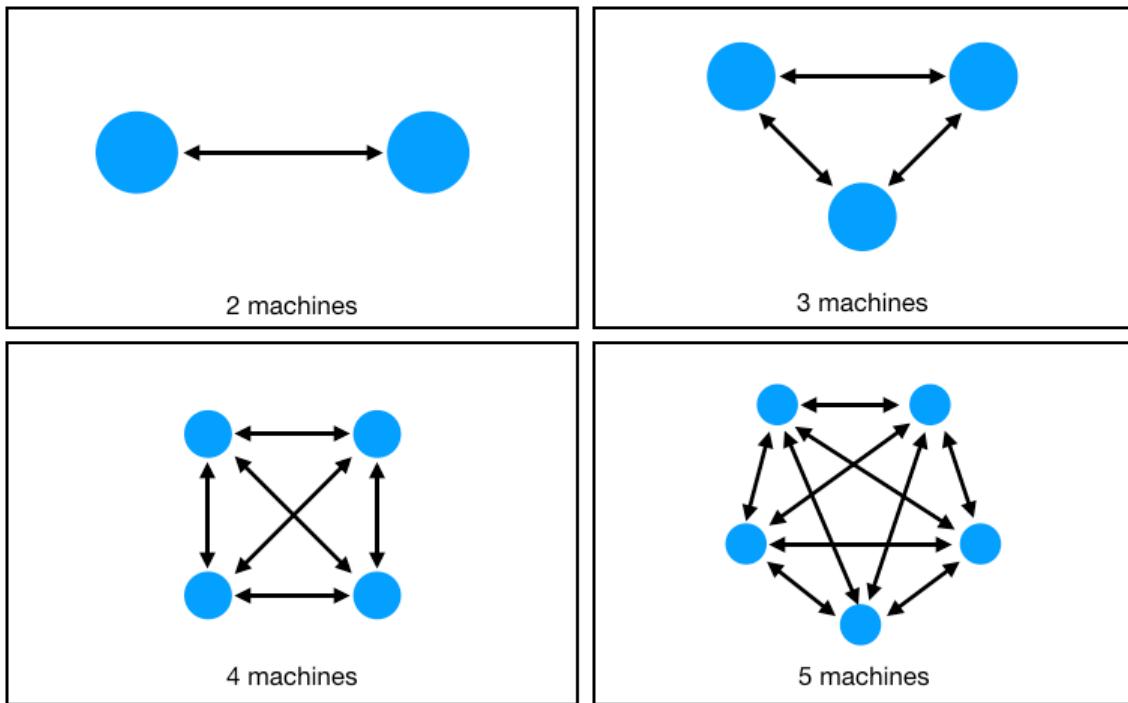


Figure 5.1 How to set up encrypted channels between different machines? Circles represent machines and arrows between them represent a common shared secret. With 2 and 3 machines we can see that we need respectively 2 and 3 different shared secrets, but this does not increase linearly. With 4 machines we need 6 different shared secrets and with 5 we need 10 different shared secrets.

Now imagine that you need to setup 100 different machines, and you always want them to have their own secret channel. It is important that you do not use the same shared secret between all machines! If a machine gets compromised, an attacker shouldn't be able to start using the compromised shared secrets to decrypt other machines' communications. Having a different shared secret between every machine means that you need to provision 4,950 different shared secrets (one for every possible channel).³⁹ That's a lot!

Wouldn't it be better if this would grow linearly instead? Meaning that for 100 machines we would only need 100 secrets?

This is what key exchanges allow us to do! In this chapter I will introduce this new cryptographic primitive and explain how it can help us bootstrap trust between components (here servers) of a system. You will learn what type of key exchanges exist, how they work and how you can make use of them in real-world applications.

For this chapter you'll need to have read:

- Chapter 4 on Authenticated Encryption.

5.1 What is a Key Exchange?

In this section we will answer the question: "what is a **key exchange**?" and we will show how we can use it to facilitate our previously discussed scenario.

So here we are again, with our friends Alice and Bob, who this time want to come up with a **shared secret** (usually a large number or bytestring). What they want to do with the shared secret afterwards does not matter for us too much in this chapter, but you can imagine that they might want to use it as a key for encrypting their communications for example. To do this, they can perform a **key exchange**. At a high level, a key exchange is a series of steps and exchange of messages that will end with both Alice and Bob sharing the same secret. Passive observers to that exchange will not be able to learn anything about the resulting shared secret.

Now, let's get into the meaty details.

A key exchange starts with both Alice and Bob generating some keys. To do this, they both use a **key generation algorithm** (usually provided by the key exchange cryptographic primitive) that generate a random **key pair**. Their key pairs include:

- a private key (also called secret key)
- a public key

This is how most public key primitives work by the way. Instead of a symmetric key, you get two types of keys (a public and a private one) that are related to one another by some magical link.

We will now assume (for the sake of a simple explanation) that Alice knows Bob's public key, and Bob knows Alice's public key as well. If this bothers you, you can imagine that they have met in real life to exchange their respective public keys.

To perform the key exchange, Alice can use Bob's public key with her own private key and compute the shared secret. Bob can, similarly, use his private key with Alice's public key and obtain the **same** shared secret. This is illustrated in figure 5.2.

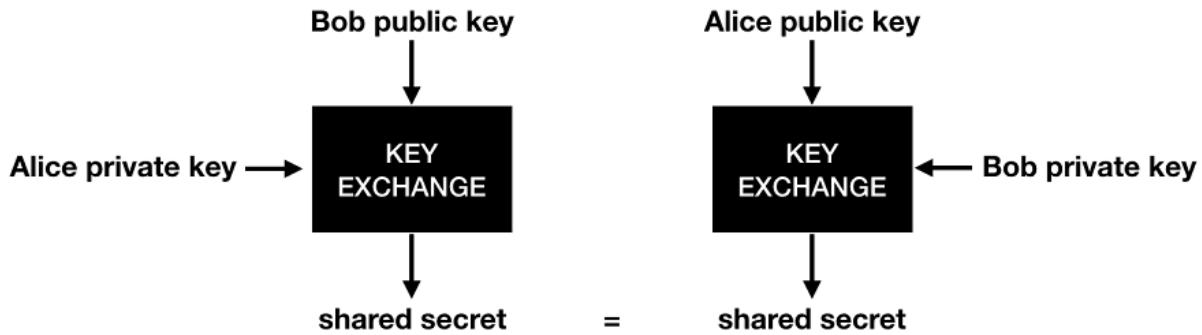


Figure 5.2 A key exchange usually provides the following interface: it takes a peer's public key and your private key to produce a shared secret between your peer and you.

Et voila! That's pretty much all there is to key exchanges. You can see this as a protocol that allow two peers to **agree** on a shared secret. Actually, key exchanges are sometimes called **key agreement** protocols.

Now let's look at our previous problem where we needed to setup 5 (or more) machines and we wanted to only have to deal with 5 secrets. In figure 5.3 we do exactly this: each machine now has an identity represented by their public keys.

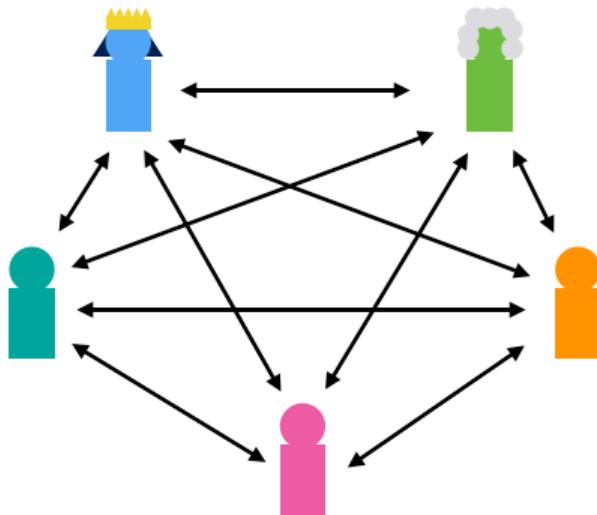


Figure 5.3 Five machines are trying to encrypt their communications via a unique per-channel shared secret. Here each arrow represent a unique shared secret. Each of these shared secrets can be calculated interactively from the peers' public keys.

Thus, for 100 machines, we need 100 public keys. To communicate between one another, two machines can start a key exchange with their respective public keys and end up with a shared secret without us having to do anything.

WARNING By the way, how do you think real-world applications manage to learn about each other's public keys? If you thought "well, they are transmitted in the clear" then you were right, many applications do transmit public keys in the clear to bootstrap a key exchange because they do not know the other public keys in advance. There's more to it though! What if someone malicious intercepts the public keys being exchanged? That someone has a name in cryptography, a **man-in-the-middle (MITM)**, and unfortunately a key exchange is vulnerable to a simple MITM attack illustrated in figure 5.4.

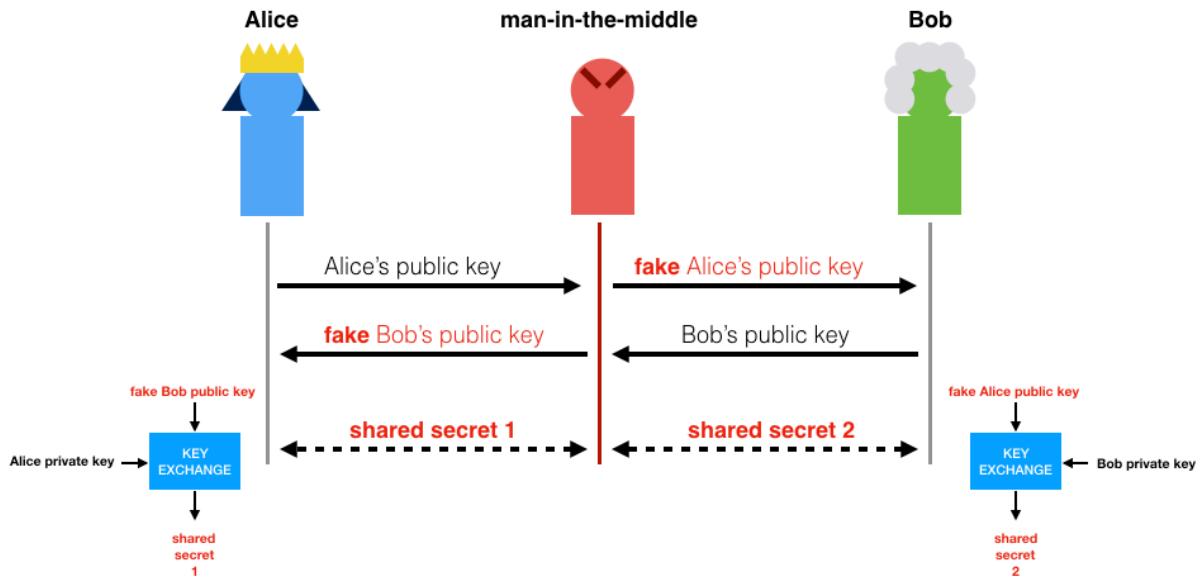


Figure 5.4 If Alice and Bob do not know their respective public keys in advance of a key exchange, and they need to exchange them to perform a key exchange, then it is trivial for a man-in-the-middle adversary to impersonate both end of the exchange. A key exchange that starts with an exchange of public keys is thus insecure by default. This book will discuss many of the real-world mechanisms that provide trust to exchanged public keys.

WARNING Because we did not know the other peer's public key in advance, anyone can **impersonate** them by sending us any public key. For now, the only way for Alice and Bob to **authenticate** each other's public keys is to know them in advance of the key exchange. This is not great, as it means that for 100 machines we need each of them to store the other 99 machines' public keys! It is still an improvement. For example, if we want to add new machines to our setup, and we will see in chapter 7 on signatures that we can further improve this scenario.

All right, now let's look at how you would use a key exchange cryptographic primitive in code. Libsodium⁴⁰ is one of the standard and widely used C/C++ library. Here is how you would use libsodium in order to perform a key exchange:

Listing 5.1 key_exchange.c

```
unsigned char client_pk[crypto_kx_PUBLICKEYBYTES];    ①
unsigned char client_sk[crypto_kx_SECRETKEYBYTES];    ①
crypto_kx_keypair(client_pk, client_sk);               ②

unsigned char server_pk[crypto_kx_PUBLICKEYBYTES];     ③
obtain(server_pk);                                     ③

unsigned char key_to_decrypt[crypto_kx_SESSIONKEYBYTES]; ③
unsigned char key_to_encrypt[crypto_kx_SESSIONKEYBYTES]; ④

if (crypto_kx_client_session_keys(key_to_decrypt, key_to_encrypt,
                                  client_pk, client_sk, server_pk) != 0) { ⑤
    abort_session(); ⑤
}
```

- ① Use this function to generate the client's keypair (secret key, public key).
- ② We assume at this point that we have some way to obtain the server's public keys.
- ③ Instead of generating one shared secret, libsodium goes further and per best practice derives two symmetric keys for us to use: one to decrypt messages from the server, and one to encrypt messages to the server.
- ④ We use this function with our secret key and the server's public key in order to perform the key exchange.
- ⑤ If the server's public key is malformed (or even maliciously formed) the function returns an error.

Libsodium hides a lot of unnecessary details from the developer, while also exposing safe-to-use interfaces (which is unfortunately not always the case for every library).

Now that we have introduced what key exchanges are, it is good to note that **key exchanges are rarely used on their own**. I'll go even further and say that you will probably not make use of them directly if you are writing applications involving cryptography. The reason is, they are used as the center piece of more complicated protocols that provide additional security properties via the use of other cryptographic primitives. You will learn more about this in chapter 9 on Session Encryption. For this reason, most people's interaction with key exchanges happen as part of an interaction with a larger protocol.

Still, you will need to understand what key exchanges are to understand most of the protocols I'll talk about in the second part of this book. In the next section, you will learn about how key exchanges work, and what instantiations of key exchanges are commonly used.

5.2 Key Exchange Standards

There are many key exchange algorithms, and all are based on mathematical problems believed to be hard to solve. This is unlike symmetric cryptography primitives that most often rely on statistical analysis and years of cryptanalysis.

All right, what are you going to learn in this section?

- The first key exchange algorithm I will talk about is actually the very first key exchange algorithm ever invented (and one of the first formalization of a public key cryptographic algorithm). It is called **Diffie-Hellman (DH)** after the last names of the two inventors Whitfield Diffie and Martin E. Hellman.
- The second key exchange algorithm I will talk about is actually the same Diffie-Hellman algorithm but built with elliptic curves, it is thus usually called **Elliptic Curve Diffie-Hellman (ECDH)**.

I will do my best to give good intuitions on how these algorithms work, but there's no way around it: there is going to be some math. Brace yourself!

5.2.1 Diffie-Hellman (DH)

In 1976, Whitfield Diffie and Martin E. Hellman wrote their seminal paper on the DH key exchange algorithm entitled "New Direction in Cryptography" (what a title right?).

The construction was based on a mathematical problem that was (and still is) believed to be hard: how to find a **discrete logarithm** in a **group** (a mathematical concept I will explain later).

In this section I will provide explanations on what is a group, what is this discrete logarithm problem, and what group DH chose to use. You will see later that ECDH is the same algorithm but using a different group. At the end of this section you will hopefully leave with a strong intuition on how a DH key exchange algorithm works, and how it can be used in practice.

As I've stated earlier, a key exchange is a few mathematical operations happening in a group. But what's a **group**?

It's two things:

1. A set of elements.
2. An operation defined on these elements that satisfy certain properties.

This is a bit too abstract for us. Here is what we use in practice for DH:

First. The **set of strictly positive integers 1, 2, 3, 4, ..., p-1 where p is a prime number**. Different standards will specify different numbers for p.

Remember, a **prime number** is just a number that can only be divided by 1 or itself. The first prime numbers are 2, 3, 5, 7, 11, and so on.

NOTE

Prime numbers are **everywhere** in asymmetric cryptography! And fortunately, we have efficient algorithms to find large ones, but to speed things up most cryptographic libraries will instead look for **pseudo-primes** (which are numbers that have a very high probability of being primes). Interestingly, such optimizations have been broken several times in the past, the most infamous occurrence was in 2017 when more than a million devices were found to have generated incorrect primes for their cryptographic applications.⁴¹

Second. A **modular multiplication** operation defined on this set of numbers.

If you've never heard about **modular arithmetic** and what "modulo" means, read on...

Let's take for example the number 7, and write its Euclidian division with 5 as

$$7 = 5 \times 1 + 2$$

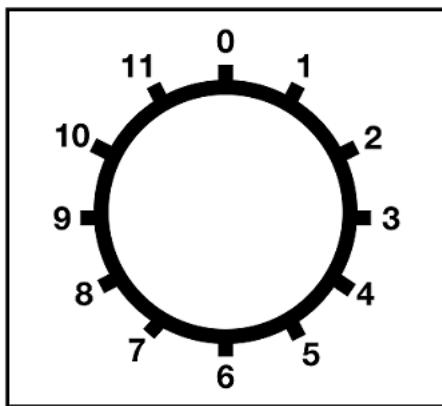
Notice that the remainder is 2. The equation above can be read as

"7 is congruent to 2 modulo 5" or **7 = 2 mod 5**

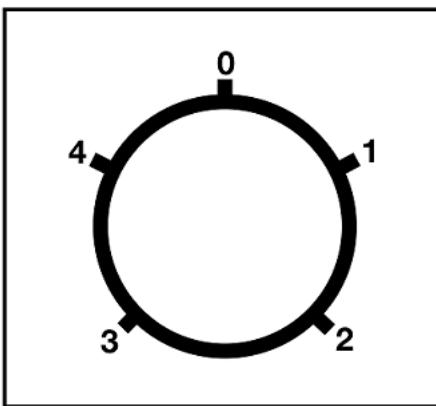
We call 5 the **modulus** in this equation. Similarly:

- $8 = 1 \text{ mod } 7$
- $54 = 2 \text{ mod } 13$
- $170 = 0 \text{ mod } 17$
- and so on...

The classical way of picturing such a concept is with a clock. Once we reach the last number (eleven), incrementing means going back to 0. This is illustrated in figure 5.5.



The normal clock **wraps around** at 12.
12 is 0, 13 is 1, 14 is 2, and so on.



Integers numbers **modulo 5**.
5 is 0, 6 is 1, and so on.

Figure 5.5 The group of integers modulo the prime number 5 can be pictured as a clock that resets to 0 after the number 4. Thus 5 is represented as 0, 6 as 1, 7 as 2, 8 as 3, 9 as 4, 10 as 0, and so on.

A modular multiplication is very natural to define over such a set of numbers. Let's take the following multiplication for example:

$$3 \times 2 = 6$$

With what you learned above, you know that 6 is congruent to 1 modulo 5, and thus the equation can be rewritten as

$$3 \times 2 = 1 \bmod 5$$

Quite straightforward isn't it?

NOTE

I said earlier that the operation satisfies a number of properties. One of them is that every element defined in our group must have an **inverse**. We say that an element a has an inverse b if

$$a \times b = 1 \bmod n$$

We usually write the inverse b of a as a^{-1} . For example, we know that 2 is the inverse of 3 mod 5 so we can write

$$3^{-1} = 2 \bmod 5$$

It happens that when we use the positive numbers modulo a **prime**, only the **zero** element lacks an inverse (indeed, can you find an element b such that $0 \times b = 1 \bmod 5$?). This is the reason why we do not include zero as one of our element in our group.

OK, we now have a group: the set of strictly positive integers $1, 2, \dots, p-1$ for p a prime number, along with the modular multiplication.

The group we formed also happens to be a **finite field**, a much broader mathematical structure

which I will ignore for now (briefly, a finite field defines two operations instead of one: a multiplication and an addition). For this reason, DH defined over this group is sometimes called **FFDH** (for **Finite Field Diffie-Hellman**). Since cryptographers like to talk about DH as the general construction that can be instantiated over any group, FFDH is sometimes used to avoid ambiguities with other instantiations of DH (like Elliptic Curve Diffie-Hellman).

One more thing I need to define for convenience, you can write:

- 3×3 as 3^2
- $3 \times 3 \times 3$ as 3^3
- and so on...

This allows me to introduce the notion of a **subgroup**. By choosing a number that we call a **generator** (or **base**), and by multiplying it over and over by itself, you can sometimes produce a **subgroup**. For example, the generator 4 defines a subgroup consisting of the numbers 1 and 4:

- $4^1 \bmod 5 = 4$
- $4^2 \bmod 5 = 1$
- $4^3 \bmod 5 = 4$ (we start again from the beginning)
- $4^4 \bmod 5 = 1$
- and so on ...

It happens that when our modulus is prime, every element of our group can **generate** a subgroup. Different subgroups have different sizes, which we call **orders**. I illustrate this in figure 5.6.

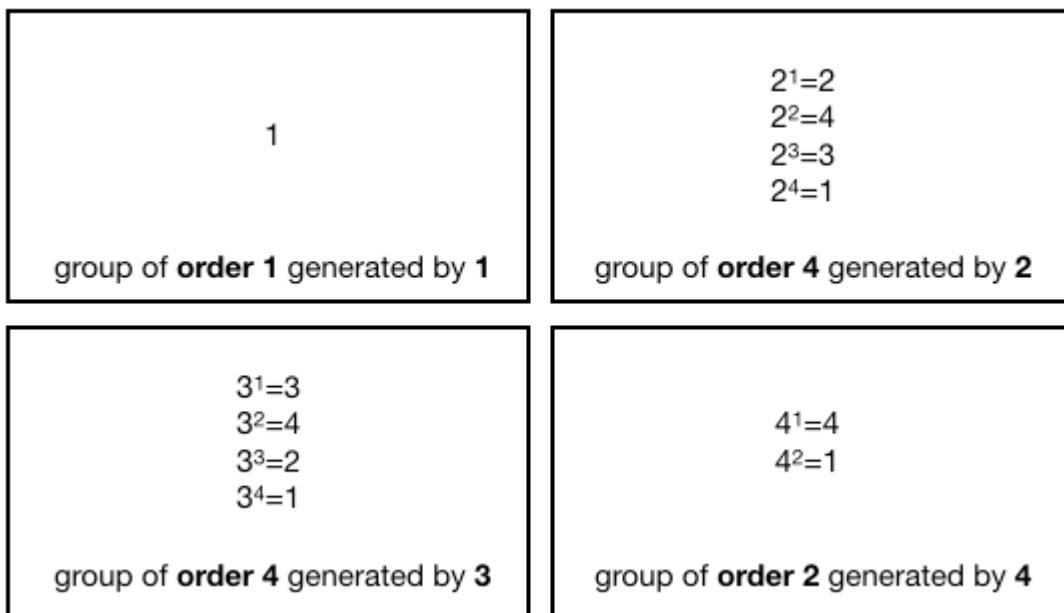


Figure 5.6 The different subgroups of the multiplicative group modulo 5. They all include the number 1 (called the identity element) and have different orders (number of elements).

All right, you now understand:

- What a **group** is.
- A generator can be used to generate a whole (sub)group.

Groups are the center of a huge amount of different cryptographic primitives. It is thus important to have good intuitions about them, if you want to understand how other cryptographic primitives work.

One more thing, what is the **discrete logarithm problem**?

Imagine that I take a generator, let's say 3, and give you a random element it generates, let's say $2 = 3^x \text{ mod } 5$. Asking you "*what is the X here?*" is the same as asking you "find the **discrete logarithm of 2 in base 3**". Thus, the discrete logarithm problem in our group is about finding out how many times we multiplied the generator with itself in order to produce a given group element.

This is a very important concept. Take a few minutes to understand it before continuing.

In our example group, you can quickly find out that 3 is the answer. Indeed $3^3 = 2 \text{ mod } 5$. But if we picked a way bigger prime number than 5, things get much more complicated: it becomes a **hard problem**.

This is the secret sauce behind Diffie-Hellman!

You now know enough to understand how to generate a keypair in DH:

1. All the participants must agree on a **large prime p** and a **generator g**.
2. Each participant generates a random number X, this becomes their **private key**.
3. Each participant derives their **public key** as $g^X \text{ mod } p$.

The fact that the discrete logarithm problem is hard means that no one should be able to recover the private key out of the public key. I illustrate this in figure 5.7.

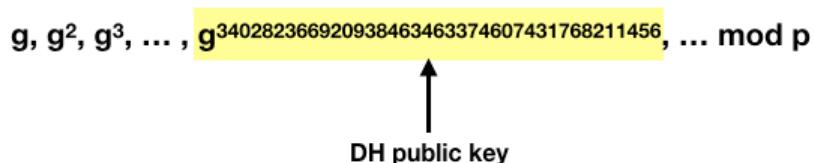


Figure 5.7 Choosing a private key in Diffie-Hellman is like choosing an index in a list of numbers produced by a generator g. The discrete logarithm problem is to find the index from the number alone.

While we do have algorithms to compute discrete logarithms, they are not very efficient. On the other hand, if I give you the solution to the problem (the X), we have very fast algorithm for you to check that indeed I provided you with the right solution ($g^X \text{ mod } p$ is fast to compute).

NOTE Like everything in cryptography, it is **not impossible** to find a solution by simply trying to guess. Yet, by choosing large enough parameters (here a large prime number), it is possible to reduce the efficacy of such a search for a solution down to negligible odds. Meaning that even after hundreds of years of random tries, your odds of finding a solution should still be statistically close to zero.

Great. **How do we use all of this math for our DH key exchange algorithm?**

Imagine that:

- Alice has a private key a and a public key $A = g^a \text{ mod } p$.
- Bob has a private key b and a public key $B = g^b \text{ mod } p$.

With the knowledge of Bob's public key, Alice can compute the shared secret as

$$B^a \text{ mod } p$$

Bob can do a similar computation with Alice's public key and his own private key

$$A^b \text{ mod } p$$

Very naturally, we can see that these two calculations end up computing the same number!

$$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \text{ mod } p$$

And that's the magic of DH. From an outsider, just observing the public keys A and B does not help in any way to compute the result of the key exchange $g^{ab} \text{ mod } p$.

NOTE By the way, in theoretical cryptography the idea that observing $g^a \text{ mod } p$ and $g^b \text{ mod } p$ does not help you to compute $g^{ab} \text{ mod } p$ is called the **Computational Diffie-Hellman Assumption (CDH)**. It is often confused with the stronger **Decisional Diffie-Hellman Assumption (DDH)** which intuitively states that given the tuple $(g^a \text{ mod } p, g^b \text{ mod } p, g^z \text{ mod } p)$, nobody should be able to confidently guess if the latter element is the result of a key exchange between the two public keys or just a random element of the group. Both are very useful theoretical assumptions that have been used to build many different algorithms in cryptography.

Next, you will learn about how real-world applications make use of this algorithm, and the different standards that exist.

5.2.2 Diffie-Hellman Standards

Now that you have seen how Diffie-Hellman works, you understand that participants need to agree on a set of parameters, specifically on a prime number p and a generator g .

In this section you'll learn about how real-world applications choose these parameters, and what are the different standards that exist.

First thing first, the prime number p . As I stated earlier, the bigger, the better. Since DH is based on a the discrete logarithm problem, its security is directly correlated to the best attacks known on the problem. Any advances in this area can weaken the algorithm. With time, we've managed to obtain a pretty good idea of how fast (or slow) these advances are, and how much is enough security. The current known best practices are to use a prime number of **2048 bits**.

NOTE

In general, www.keylength.com summarizes recommendations on parameter lengths for common cryptographic algorithms. The results are taken from authoritative documents produced by research groups or government bodies like the ANSSI (France), the NIST (US), and the BSI (Germany). While they do not always agree, they often converge towards similar orders of magnitude.

In the past, many libraries and software were generating and hardcoding their own parameters. Unfortunately, they were often found to be either weak, or worse, backdoored. While blindly following standards might seem like a good idea in general, DH is one of the unfortunate counter-example that exist, as it was found that some of them might have been backdoored, as was the case for RFC 5114. Due to all of these issues, newer protocols and libraries have converged towards either deprecating DH (in favor of ECDH) or using the groups defined in the better standard RFC 7919.⁴²

For this reason, best practice nowadays is to use RFC 7919 which defines several groups of different sizes and security. For example, **ffdhe2048** is the group defined by the 2048-bit prime modulus

$p =$
32317006071311007300153513477825163362488057133489075174588434139269806834136210002

and with generator $g = 2$

NOTE

It is common to choose the number 2 for the generator, as computers are quite efficient at multiplying with 2 (it's a simple left shift <<instruction).

The group size (or order) is also specified as $q = (p-1)/2$.

This implies that both private keys and public keys will be around 2048-bit of size.

In practice, these are quite large sizes for keys (compare that with symmetric keys for example, that are usually 128-bit). You will see in the next section that defining a group over the elliptic curves allow us to obtain much smaller keys for the same amount of security.

5.2.3 Elliptic Curve Diffie-Hellman (ECDH)

It turns out that the Diffie-Hellman algorithm, which we just discussed, can be implemented in more than just modulo a prime number. All we need is just a group! It also turns out that a group can be made from elliptic curves (a type of curves in mathematics). The idea was proposed in 1985 by Neal Koblitz and Victor S. Miller independently, and much later adopted when **Elliptic Curve Cryptography (ECC)** (cryptographic algorithms that are based on elliptic curves) started seeing standardization around 2000. The world of applied cryptography quickly adopted algorithms based on elliptic curves as they provided way smaller keys than the previous generation's algorithms. Compared to the recommended 2048-bit parameters in DH, parameters of 256-bit were possible with the elliptic curve variant of the algorithm.

Elliptic curve cryptography has remained at its full strength since it was first presented in 1985. [...] The United States, the UK, Canada and certain other NATO nations have all adopted some form of elliptic curve cryptography for future systems to protect classified information throughout and between their governments

– NSA *The Case for Elliptic Curve Cryptography* (2005)

Let's now explain how elliptic curves work. First and foremost, it is good to understand that elliptic curves are just curves! Meaning that they are defined by all the coordinates x and y that solves an equation. Specifically this equation

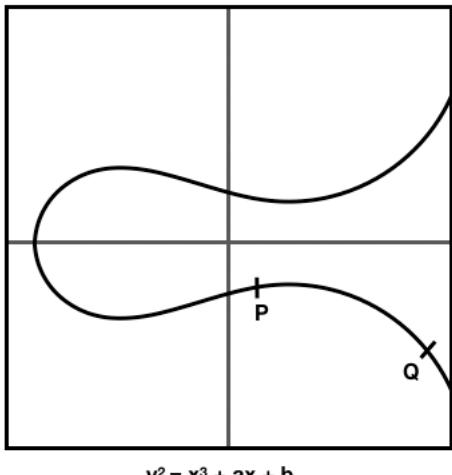
$$y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$$

for some a_1, a_2, a_3, a_4 , and a_6 .

NOTE

Note that for most practical curves today, this equation can be simplified to $y^2 = x^3 + ax + b$. While the simplification is not possible for two types of curves (called binary curves and curves of characteristic 3), they are used rarely enough that we will use the simplified form in the rest of this chapter.

Figure 5.8 shows an example of an elliptic curve, with two points taken at random.



$$y^2 = x^3 + ax + b$$

Figure 5.8 One example of an elliptic curve defined by an equation.

At some point in the history of elliptic curves, it was found that a **group** could be constructed over them. From there, implementing Diffie-Hellman on top of these groups was straightforward. I will use this section to explain the intuition behind elliptic curve cryptography.

Groups over elliptic curves are often defined as **additive groups**. Unlike multiplicative groups defined in the previous section, the $+$ sign is used instead.

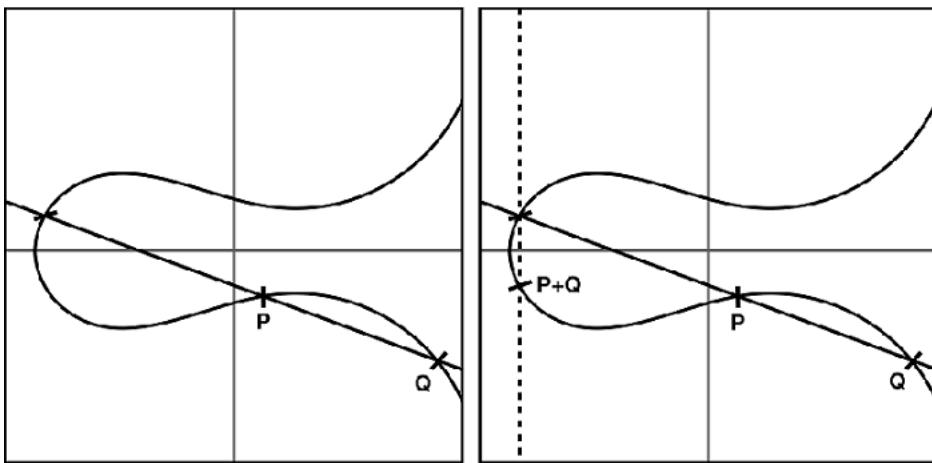
NOTE

Using an addition or a multiplication does not matter much in practice, it is just a matter of preference. While most of cryptography uses a multiplicative notation, the literature around elliptic curve has centered around a multiplicative notation, and thus this is what I will use when referring to elliptic curve groups in this book

This time, I will define the operation before defining the elements of the group. Our **addition operation** is defined in the following way:

1. Draw a line going through two points you want to add. The line hits the curve in another point.
2. Draw a vertical line from this newly found point. The vertical line hits the curve in yet another point.
3. This point is the result of adding the original two points together.

This process is illustrated in figure 5.9.



1. trace a line going through P and Q
it hits the curve at another point

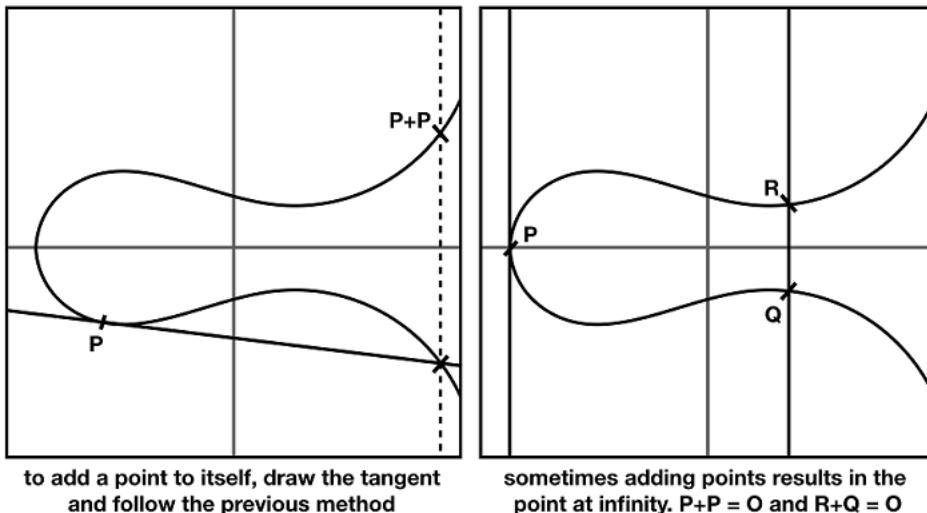
2. trace a vertical line through that point
it hits the curve at point $P+Q$

Figure 5.9 An addition operation can be defined over points of an elliptic curve by using geometry!

There are two special cases where this rule won't work. Let's define them as well:

- How do we add a point to itself? The answer is to draw the tangent to that point (instead of drawing a line between two points).
- What happens if the line we draw in step 1 (or step 2) does not hit the curve at any other point? Well, this is embarrassing and we need this special case to work and produce a result. The solution is to define the result as a fictive point (something we made up). This fictive point is called the **point at infinity** (that we usually write with a big letter O).

Figure 5.10 illustrates these special cases.



to add a point to itself, draw the tangent
and follow the previous method

sometimes adding points results in the
point at infinity. $P+P = O$ and $R+Q = O$

Figure 5.10 Building on figure , addition on an elliptic curve is also defined when adding a point to itself, or when two points cancel each other to result in the point at infinity.

I know this point at infinity is some next-level weirdness, but don't worry too much about it. It is really just something we came up with in order to make the addition operation work. Oh and by the way, it behaves like a zero, and it is our **identity element**:

$$O + O = O$$

and for any point P on the curve

$$P + O = P$$

All good. So far we've seen that to create a group on top of an elliptic curve, we need:

- An elliptic curve equation that defines a set of valid points.
- A definition of what an addition means in this set.
- An imaginary point called a point at infinity.

I know this is a lot of information to unpack, but we are missing one last thing. Elliptic curve cryptography is defined over a finite field. In practice, what this means is that our coordinates are the numbers 1, 2, ..., p-1 for some large prime number p. This should sound familiar!

For this reason, when thinking of elliptic curve cryptography, you should be thinking of a graph that looks much more like the one on the right in figure 5.11.

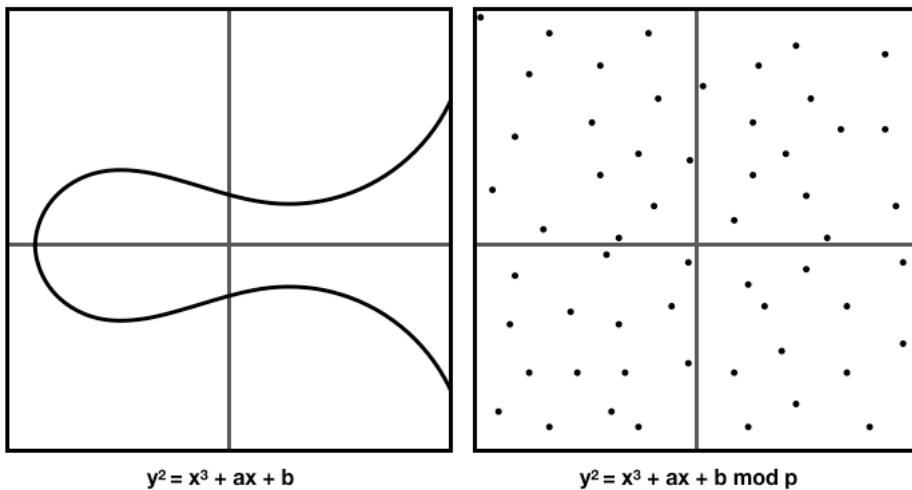


Figure 5.11 Elliptic Curve Cryptography (ECC) in practice is mostly specified with elliptic curves in coordinates modulo a large prime number p. This means that what we use in cryptography looks much more like the right graph than the left graph.

And that's it!

We now have a group we can do cryptography on. The same way we had a group made with the numbers (excluding 0) modulo a prime number and a multiplication operation for Diffie-Hellman.

How can we do Diffie-Hellman with this group defined on elliptic curves? Let's see how the **discrete logarithm** works now in this group.

Let's take a point G and add it to itself x times to produce another point P via the addition

operation we defined. We can write that as $P = G + \dots + G$ x times, or use some mathematical syntactic sugar to write that as $P = [x]G$ which reads " x times G ". The **elliptic curve discrete logarithm problem (ECDLP)** is to find the number x from knowing just P and G .

That's all we needed, you now know enough to understand how to generate a keypair in ECDH:

1. All the participants agree on an **elliptic curve equation**, a **finite field** (most likely a prime number), and a **generator G** (usually called **based point** in elliptic curve cryptography).
2. Each participant generates a random number X , this becomes their **private key**.
3. Each participant derives their **public key** as $[X]G$.

As the elliptic curve discrete logarithm problem is hard, you guessed it, no one should be able to recover your private key just by looking at your public key. I illustrate this in figure 5.12.

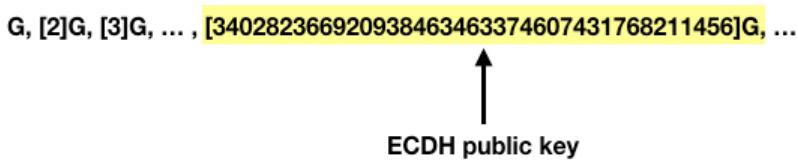


Figure 5.12 Choosing a private key in ECDH is like choosing an index in a list of numbers produced by a generator (or base point) G . The elliptic curve discrete logarithm problem (ECDLP) is to find the index from the number alone.

All of this might be a bit confusing, as the operation we had defined for our DH group was a multiplication, and for an elliptic curve we now use an addition. Again, these distinctions do not matter at all, as they are equivalent. You can see a comparison in figure 5.13.

DH	1, 2, ..., $p-1$	$1 \times 1 = 1 \text{ mod } p$ $1 \times a = a \text{ mod } p$	$a \times a^{-1} = 1 \text{ mod } p$	$a^3 = a \times a \times a \text{ mod } p$
	set of elements	identity element	multiplicative inverse	modular exponentiation
ECDH	(x, y) in $y^2 = x^3 + ax + b \text{ mod } p$	$O + O = O$ $O + A = A$	$A - A = O$	$[3]A = A + A + A$
	set of elements	point at infinity	additive inverse	scalar multiplication

Figure 5.13 Some comparisons between the group used in Diffie-Hellman and the group used in Elliptic Curve Diffie-Hellman.

You should now be convinced that the only thing that matters for cryptography is that we have a group defined with its operation, and that the discrete logarithm for this group is hard. See figure 5.14.

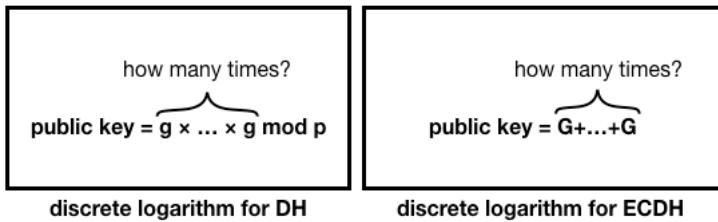


Figure 5.14 A comparison between the discrete logarithm problem modulo large primes, and the discrete logarithm problem in elliptic curve cryptography. They both relate to the Diffie-Hellman key exchange as the problem is to find the private key from a public key.

Last note on the theory, the group we formed on top of elliptic curves has some difference with the group we formed with the strictly positive integers modulo a prime number. Due to some of these differences, the strongest attacks known against DH (index calculus) do not work well on the elliptic curve groups. This is the main reason why parameters for ECDH can be much much lower than the parameters for DH, for the same level of security.

OK we are done with the theory. Let's go back to defining ECDH. Imagine that:

- Alice has a private key a and a public key $A = [a]G$.
- Bob has a private key b and a public key $B = [b]G$.

With the knowledge of Bob's public key, Alice can compute the shared secret as

$$[a]B$$

Bob can do a similar computation with Alice's public key and his own private key

$$[b]A$$

Very naturally, we can see that these two calculations end up computing the same number!

$$[a]B = [a][b]G = [ab]G = [b][a]G = [b]A$$

And no passive adversary should be able to derive the shared point just from observing the public keys.

Looks familiar right?

Next, let's talk about standards!

ELLIPTIC CURVE DIFFIE-HELLMAN STANDARDS

The standardization of ECDH has been even more chaotic than DH. Many standardization bodies have come out and specified a large number of different curves, which was then followed by many wars over which curve was more secure. A large amount of research, mostly led by Daniel J. Bernstein, pointed out the fact that a number of curves standardized by the NIST could potentially be part of a weaker class of curves only known to the NSA.

I no longer trust the constants. I believe the NSA has manipulated them through their relationships with industry.

— Bruce Schneier *The NSA Is Breaking Most Encryption on the Internet* (2013)

Nowadays, most of the curves in use come from two different standards:

- NIST FIPS 186-4, which was originally published in 2000.
- RFC 7748 which was published in 2016.

NIST FIPS 186-4 specifies 15 different curves, amongst which **P-256** is the most widely used curve on the internet. It is defined with the points on the equation

$$y^2 = x^3 - 3x + b \pmod{p}$$

where

$$\begin{aligned} b &= \\ 41058363725152142129326129780047268409114441015993725554835256314039467401291 & \end{aligned}$$

and

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

It defines a curve of order

$$\begin{aligned} n &= \\ 115792089210356248762697446949407573529996955224135760342422259061068512044369 & \end{aligned}$$

meaning that there are n points (including the point at infinity).

The base point is the point

$$\begin{aligned} G &= \\ (48439561293906451759052585252797914202762949526041747995844080717082404635286, \\ 36134250956749795798585127919587881956611106672985015071877198253568414405109) & \end{aligned}$$

The curve provides 128-bit of security. For applications that are using other cryptographic algorithms providing 256-bit security instead of 128-bit of security (for example AES with a

256-bit key), P-512 is available in the same standard to match the level of security.

NOTE Interestingly P-256, and other curves defined in FIPS 186-4, are said to be generated from a seed. For P-256, the seed is the bytestring 0xc49d360886e704936a6678e1139d26b7819f7e90. I've talked about this notion of "nothing-up-my-sleeve" numbers before, that are constants aimed at proving that there was no room for backdoor during the design of the algorithm. Unfortunately, there isn't much explanation behind this seed, other than the fact that it is specified along the curve's parameter.

RFC 7748 specifies Curve25519, the curve defined by the equation

$$y^2 = x^3 + 486662x^2 + x \pmod{p}$$

where

$$p = 2^{255} - 19$$

It defines a curve of order

$$n = 2^{252} + 27742317777372353535851937790883648493$$

The base point is

$$G = (9, 14781619447589544791020593568409986887264606134616475288964881837755586237401)$$

The curve also provides 128-bit of security. For 256-bit of security, Curve448 is available in the same standard.

5.3 Summary

- Key exchanges are useful to establish trust in settings with a lot of participants. They are rarely used on their own and most often live inside more complex protocols.
- Diffie-Hellman (DH) is the first key exchange algorithm invented, and is still widely used.
- Not knowing the other peer's public key in advance (non-authenticated DH) is insecure as one can easily impersonate the other peer if the public keys are exchanged in clear.
- The recommended standard to use for DH is RFC 7919 which includes several parameters to choose from. The smallest option is the recommended 2048-bit prime parameter.
- Elliptic Curve Diffie-Hellman (ECDH) has much smaller key sizes than DH. For 128-bit of security, DH needs 2048-bit parameters whereas ECDH needs 256-bit parameters.
- The most widely used curves for ECDH are P-256 and X25519. Both provide 128-bit of security. For 256-bit of security, P-521 and Curve448 are available in the same standards.

6

Asymmetric encryption and hybrid encryption

This chapter covers

- Asymmetric Encryption can be used to encrypt secrets to a public key.
- Hybrid Encryption can be used to encrypt large amounts of data to a public key.
- The standards for Asymmetric and Hybrid Encryption.

You've learned about authenticated encryption in chapter 4, which is a form of symmetric encryption. Authenticated encryption allowed you to encrypt data to someone else who shared the same symmetric key. This is an extremely useful cryptographic primitive, yet in the real-world, there exist many situations where different peers do not have a shared secret. chapter 5 introduced asymmetric cryptography and how key exchanges allow two participants who are aware of each other's public key to derive a shared secret in the open. This chapter bridges asymmetric cryptography with symmetric cryptography, showing you how you can encrypt to a person with whom you do not share a secret yet, as long as you know their public key.

For this chapter you'll need to have read:

- Chapter 4 on Authenticated Encryption.
- Chapter 5 on Key Exchanges.

Let's get started!

6.1 What is Asymmetric Encryption?

The first step to understanding how to encrypt a message to someone is **asymmetric encryption** (also called **public-key encryption**). In this section you will learn about this cryptographic primitive and its properties.

Let's take a look at the following real-world scenario: **encrypted emails**.

You probably know that all the emails you send are sent in the clear, for anyone (sitting in between your and your recipient's email providers) to read.

That's not great. How do you fix this? You could use a cryptographic primitive like AES-GCM, which you've learned about in chapter 4 on authenticated encryption. To do that, you would need to set up a different shared symmetric secret for each person that wants to message you (using the same shared secret with everyone would be very bad, can you see why?). In practice, how can you do this? You can't expect to know in advance who is going to want to message you.

One solution is **asymmetric encryption**! With such a primitive, anyone can encrypt messages to you using your **public key**, and you are the only one who can decrypt such messages via your associated **private key**. See figure 6.1.

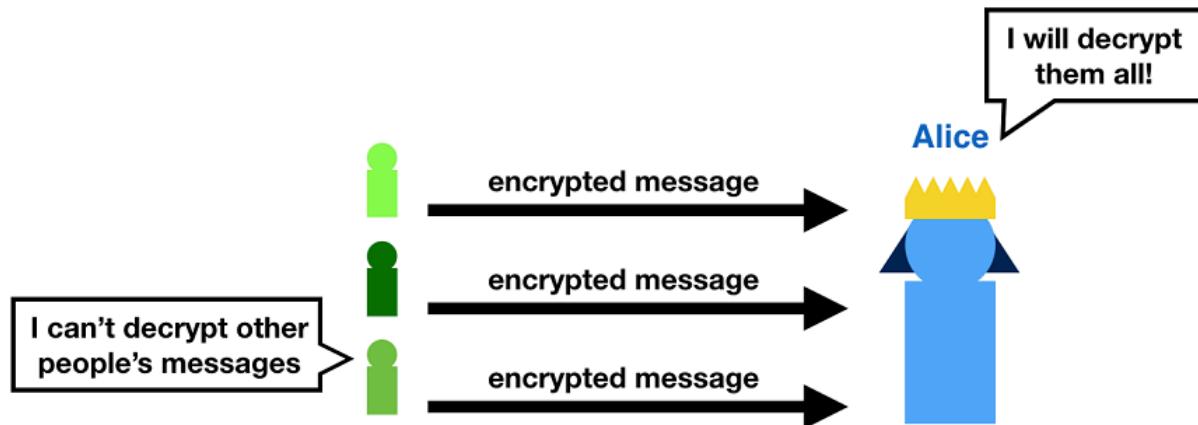


Figure 6.1 With asymmetric encryption, anyone can use Alice's public key to send her encrypted messages. Only Alice, who owns the associated private key, can decrypt these messages.

To set up such an algorithm, you first need to **generate a key pair** via some algorithm as illustrated in figure 6.2.

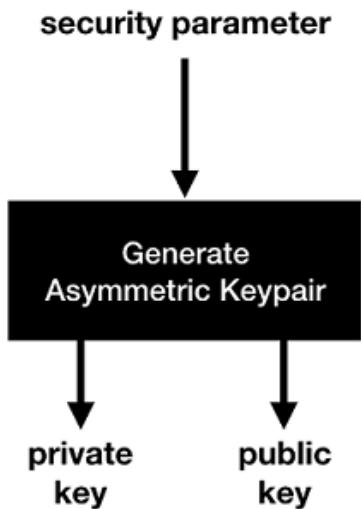


Figure 6.2 To use asymmetric encryption, you first need to generate a keypair. Depending on the security parameters you provide, you will generate keys of different security strength.

The key generation algorithms generates a keypair which comprises two different parts: the **public key** part (as the name indicates) can be published and shared without much concerns, while the **private key** must remain secret. Similar to the key generation algorithms of other cryptographic primitives, a security parameter is required in order to decide on the bit-security of the algorithm.

Anyone can then use the public key part to encrypt messages, and you can use the private key part to decrypt them as illustrated in figure 3.4.

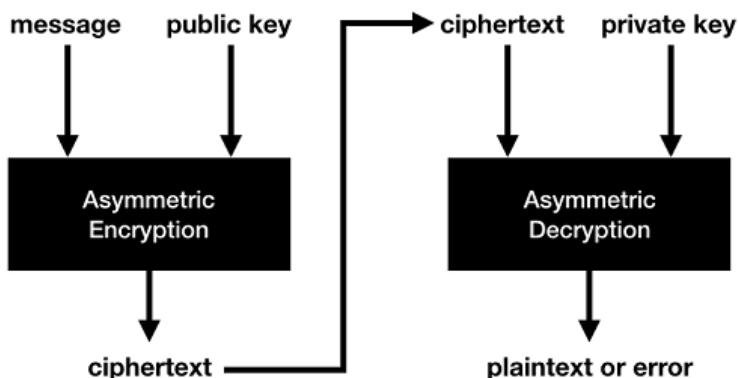


Figure 6.3 Asymmetric encryption allows one to encrypt a message (plaintext) using a recipient's public key. The recipient can then use a different algorithm to decrypt the encrypted message (ciphertext) using a private key related to the public key used previously.

Similar to authenticated encryption, decryption can still fail if presented with an incoherent ciphertext.

WARNING Note that **no one is authenticated** here by default: you know you are encrypting to **a public key** (which you think is owned by Alice) and Alice does not know for sure who sent this message.

Like in chapter 5, we still need to understand where the **trust** comes from. For now, we will imagine that we've obtained Alice's public key in real life, or from a friend, or from another secure mechanism we trust. In chapter 7, which covers digital signatures, I teach how real-world protocols solve this trust problem at scale.

Now assuming that **you know for sure that the public key you're using is Alice's**, the protocol is still not **mutually** authenticating the participants. Only **you** know you are encrypting to Alice, but Alice still has no way to verify your identity.

Let's now go to the next section, where you'll learn about how asymmetric encryption is used in practice (and spoiler alert, why it's rarely used in practice).

6.2 Asymmetric Encryption in Practice and Hybrid Encryption

You've just learned about asymmetric encryption, and you might be thinking that it is probably enough to start encrypting your emails, but in reality asymmetric encryption is quite limited due to the limited length of messages it can encrypt. In this section you will learn about these limitations, what asymmetric encryption is actually used for in practice, and finally how cryptography overcomes these limitations. The section is divided in two parts for the two main use-cases of asymmetric encryption:

- Key exchanges. You will see that it is quite natural to perform a key exchange (or key agreement) with an asymmetric encryption primitive.
- Hybrid encryption. The use-cases for asymmetric encryption are quite limited due to the maximum size of what you can encrypt with it. To encrypt larger messages, you will learn about a more useful primitive called hybrid encryption.

Key Exchanges/Key Encapsulation. It turns out that asymmetric encryption can be used to perform a key exchange! The same kind as the ones we've seen in chapter 5. In order to do this, you can start by generating a symmetric key and encrypt it with Alice's public key — what we also call **encapsulating a key** — as seen in figure 6.4.

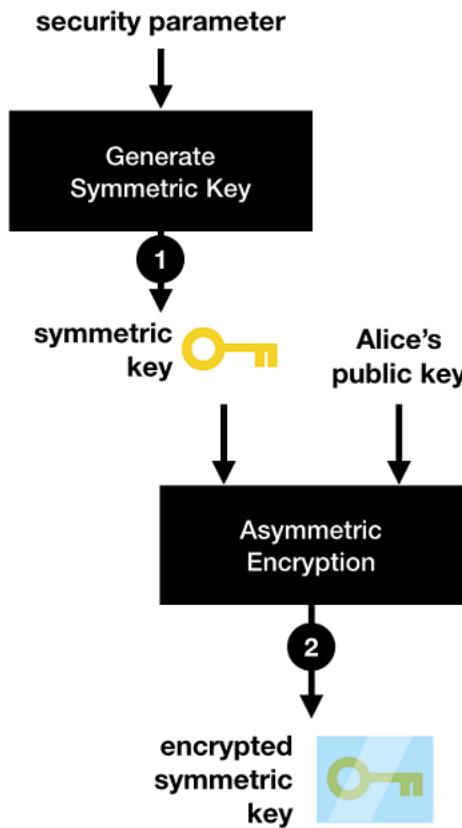


Figure 6.4 To use asymmetric encryption as a key exchange primitive, you first (1) generate a symmetric key and (2) encrypt it with Alice's public key.

You can then send the ciphertext to Alice who will be able to decrypt it and learn the symmetric key. Subsequently, you will eventually both have a **shared secret!** The complete flow is illustrated in figure 6.5.

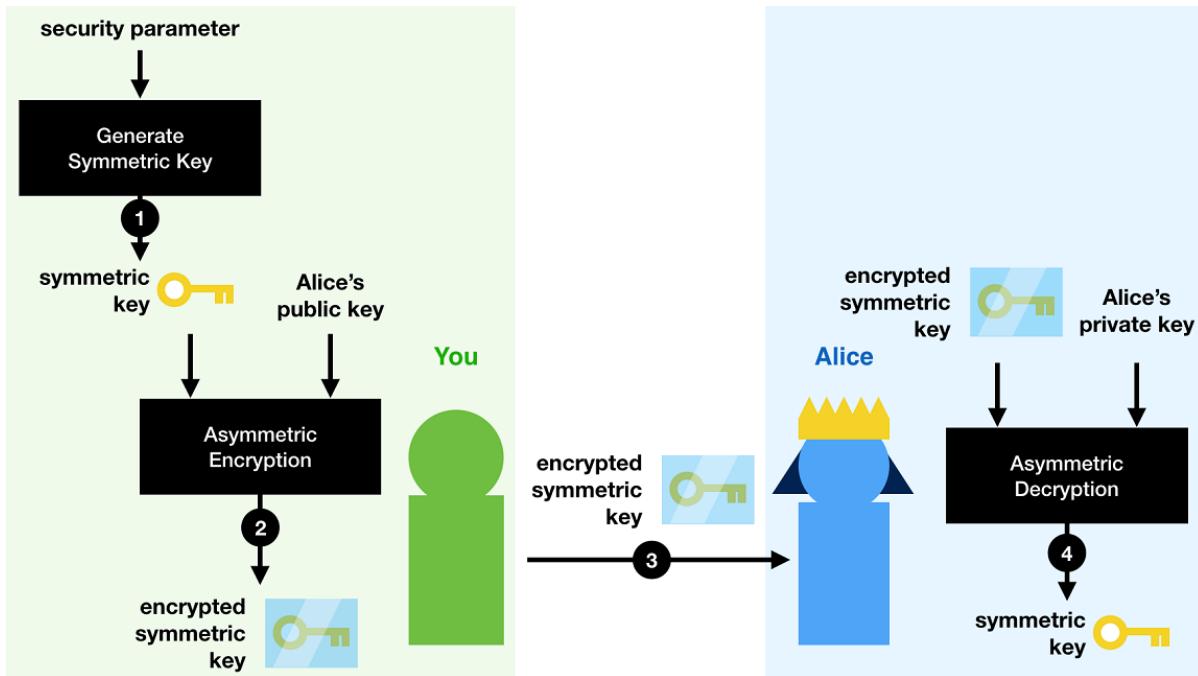


Figure 6.5 To use asymmetric encryption as a key exchange primitive, you can (1) generate a symmetric key and (2) encrypt it with Alice's public key. After (3) sending it to Alice, she can (4) decrypt it with her associated private key. At the end of the protocol they both have the shared secret, while no one else was able to derive it from observing the encrypted symmetric key alone.

Using asymmetric encryption to perform a key exchange is usually done with an algorithm called **RSA** (following the names of its inventors **Rivest, Shamir, and Adleman**), and used in many internet protocols.

NOTE

RSA is not the only asymmetric encryption algorithm, but it is the most widely used one in practice. Note that RSA is also a signature scheme (which we will talk about in chapter 7 on digital signatures). Mentions of RSA in this chapter are to be understood as "RSA the asymmetric encryption algorithm".

Today, RSA is often not the preferred way of doing a key exchange, and it is being used less and less in protocols in favor of Elliptic Curve Diffie-Hellman (ECDH). This is mostly due to historical reasons (many vulnerabilities have been discovered with RSA implementations and standards) and the attractiveness of the smaller parameter sizes offered by ECDH.

Hybrid Encryption. In practice, asymmetric encryption can only encrypt messages up to a certain length. For example with RSA, the size of plaintext messages that can be encrypted by the algorithm are limited by the security parameter used. Nowadays, with the security parameters used, the limit is approximately 500-byte characters. Pretty small. Therefore most applications make use of **hybrid encryption** instead which has much larger limitations.

In practice, hybrid encryption has the same interface as asymmetric encryption (see figure 6.6). People can encrypt messages with a public key and the one who owns the associated private key can decrypt the encrypted messages. The real difference is in the size limitations of the message that can be encrypted.

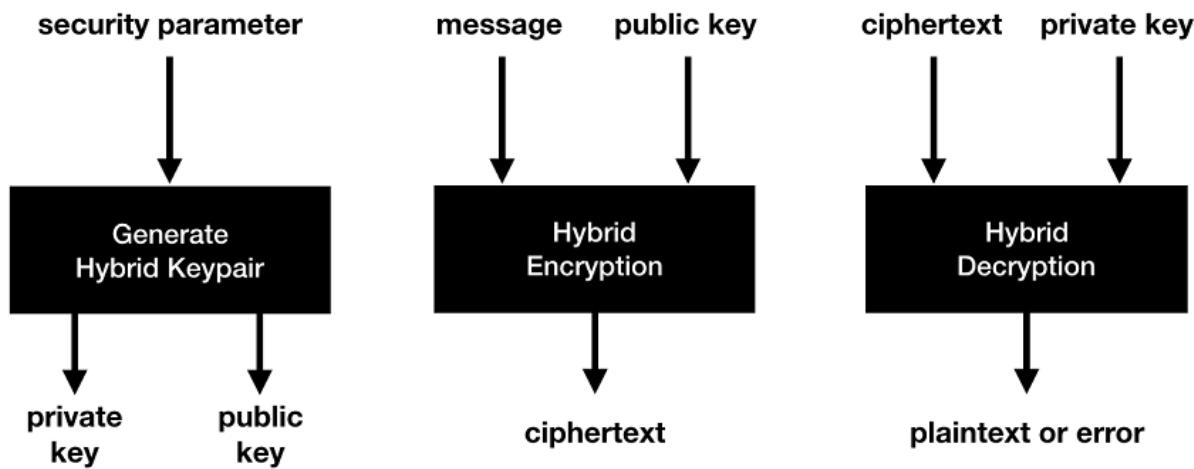


Figure 6.6 Hybrid Encryption has the same interface as asymmetric encryption, except that messages that can be encrypted are much larger in size.

Under the cover, hybrid encryption is simply the combination of an **asymmetric** cryptographic primitive with a **symmetric** cryptographic primitive (hence the name). Specifically, it is a key exchange with the recipient followed by the encryption of a message with an authenticated encryption algorithm.

NOTE

Remember chapter 4 on authenticated encryption. You could have used a simple symmetric encryption primitive instead of an authenticated encryption primitive, but symmetric encryption does not protect against someone tampering your encrypted messages by default. This is why we never use symmetric encryption alone in practice.

Let's learn about how hybrid encryption works!

If you want to encrypt a message to Alice, you first generate a symmetric key and encrypt your message with it and an authenticated encryption algorithm as seen in figure 6.7.

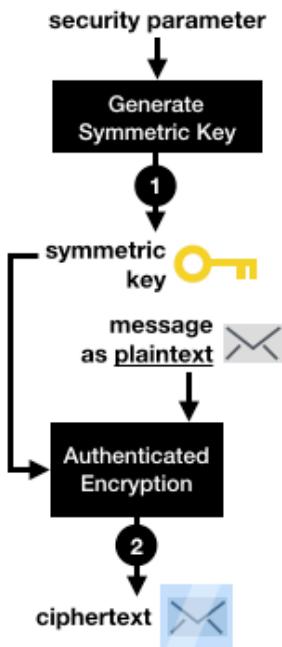


Figure 6.7 To encrypt a message to Alice using Hybrid Encryption with asymmetric encryption, you first (1) generate a symmetric key for an authenticated encryption algorithm. (2) You use the symmetric key to encrypt your message to Alice.

Once you have encrypted your message, Alice still cannot decrypt it without the knowledge of the symmetric key. How do we provide Alice with that symmetric key? Asymmetrically encrypt the symmetric key with Alice's public key as in figure 6.8.

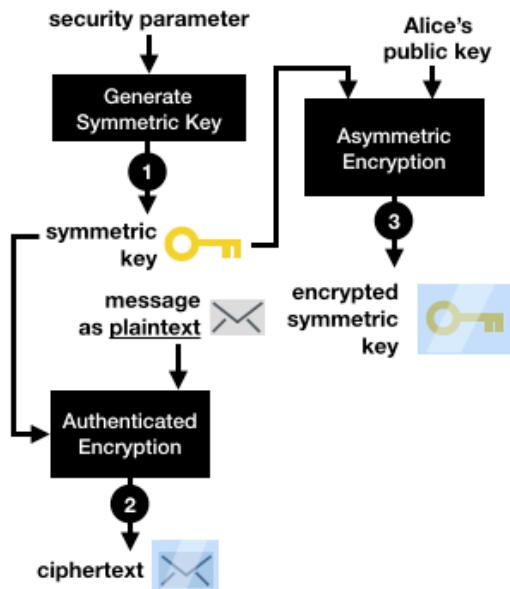


Figure 6.8 Building on figure . (3) You can encrypt the symmetric key itself by using Alice's public key and an asymmetric encryption algorithm.

Finally, you can send both of the results to Alice:

- the asymmetrically encrypted symmetric key.
- the symmetrically encrypted message.

which is enough information for Alice to decrypt the message. I illustrate the full flow in figure 6.9.

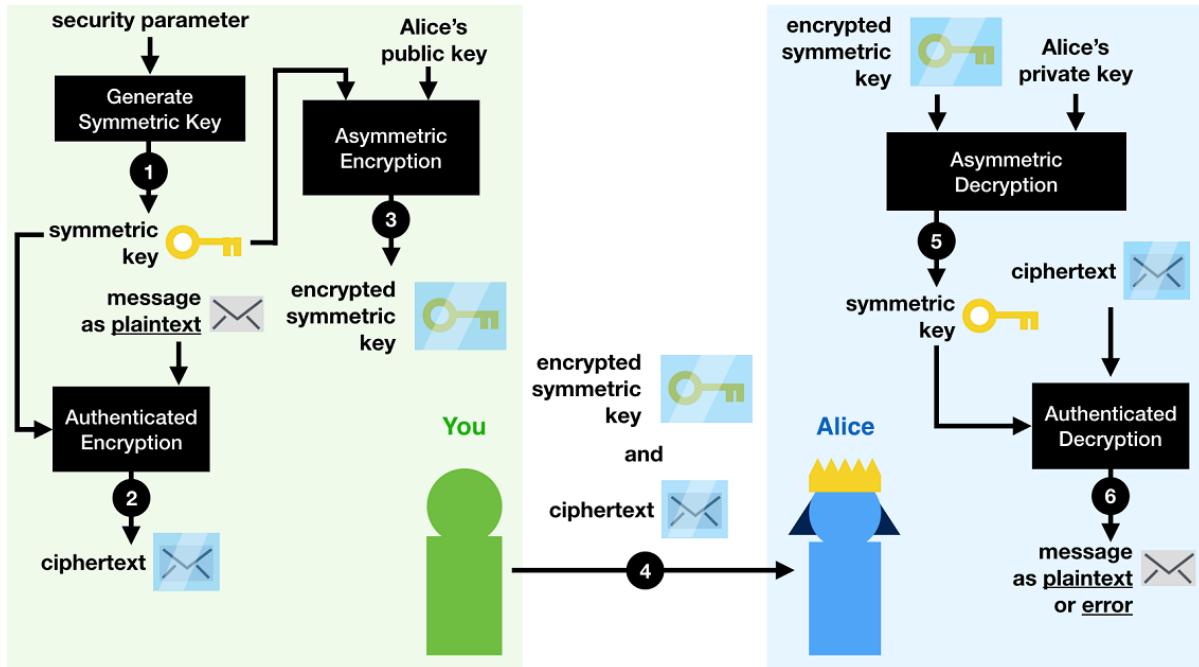


Figure 6.9 Building on figure . (4) After you send both the encrypted symmetric key and the encrypted message to Alice, (5) Alice can decrypt the symmetric key using her private key. (6) She can then use the symmetric key to decrypt the encrypted message. Step 5 and Step 6 can both fail and return errors if the communications were tampered with by a man-in-the-middle attacker at step 4.

And this is how we can use the best of both worlds, mixing asymmetric encryption and symmetric encryption to encrypt large amounts of data to a public key.

NOTE

In more theoretical terms, we often call the first asymmetric part of the algorithm a **Key Encapsulation Mechanism (KEM)** and the second symmetric part a **Data Encapsulation Mechanism (DEM)**.

Before we move to the next section and learn about the different algorithms and standards that exist for both asymmetric encryption and hybrid encryption, let's see in practice how you can use a cryptographic library to perform hybrid encryption.

To do this, I chose the **Tink** cryptographic library. Tink was developed by a team of cryptographers at Google to support large teams inside and outside of the company. Because of the scale of the project, conscious design choices were made and sane functions were exposed in order to prevent developers from mis-using cryptographic primitives. In addition, Tink is available in several programming languages (Java, C++, Obj-C, and Golang).

Listing 6.1 hybrid_encryption.java

```
import com.google.crypto.tink.HybridDecrypt;
import com.google.crypto.tink.HybridEncrypt;
import com.google.crypto.tink.hybrid.HybridKeyTemplates;
import com.google.crypto.tink.KeysetHandle;

KeysetHandle privateKeysetHandle = KeysetHandle.generateNew(
    HybridKeyTemplates.ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM); ①

KeysetHandle publicKeysetHandle =
    privateKeysetHandle.getPublicKeysetHandle(); ①

HybridEncrypt hybridEncrypt =
    publicKeysetHandle.getPrimitive(HybridEncrypt.class); ②
byte[] ciphertext = hybridEncrypt.encrypt(plaintext, associatedData); ③

HybridDecrypt hybridDecrypt = privateKeysetHandle.getPrimitive(
    HybridDecrypt.class); ③
byte[] plaintext = hybridDecrypt.decrypt(ciphertext, associatedData); ③
```

- ① We generate keys for a specific Hybrid Encryption scheme.
- ② We obtain the public key part that we can publish or share.
- ③ Anyone who knows this public key can use it to encrypt a plaintext as well as authenticate some associated data.
- ④ We can decrypt such an encrypted message using the same associated data. If the decryption fails, it will throw an exception that you have to catch. (This is specific to Java.)

One note to help you understand the ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM string:

- **ECIES (for Elliptic Curve Integrated Encryption Scheme)** is the hybrid encryption standard to use, you'll learn about this later in this chapter. The rest of the string lists the algorithms used to instantiate ECIES.
- **P256** is the NIST standardized elliptic curve you've learned about in chapter 5 on key exchanges.
- **HKDF** is a key derivation function you will learn about in chapter 8 on randomness and secrets.
- **HMAC** is the message authentication code you've learned about in chapter 3.
- **SHA-256** is the hash function you've learned about in chapter 2.
- **AES-128-GCM** is the AES-GCM authenticated encryption algorithm using a 128-bit key you've learned about in chapter 4.

See how everything is starting to fit together?

And that's it. In the next section you will learn about RSA and ECIES, the two widely adopted standards for asymmetric encryption and hybrid encryption.

6.3 Standards for Asymmetric Encryption and Hybrid Encryption

It is time for us to look at the standards that define asymmetric encryption and hybrid encryption in practice. Historically, both of these primitives have not been spared by cryptanalysts, and many vulnerabilities and weakenesses have been found in both standards and implementations. This is why I will start this section with an introduction of the RSA asymmetric encryption algorithm, and how not to use it. The rest of the section will go through the actual standards you can follow to use asymmetric and hybrid encryption:

- **RSA-OAEP.** The main standard to perform asymmetric encryption with RSA.
- **ECIES.** The main standard to perform hybrid encryption with Elliptic Curve Diffie-Hellman.

6.3.1 Textbook RSA

In this section you will learn about the RSA public-key encryption algorithm, and how it has been standardized throughout the years. This will be useful in order to understand other secure schemes based on RSA.

Unfortunately, RSA has caught quite some bad rap since it was first published in 1977. One of the popular theories is that RSA is too easy to understand and implement, and thus many people do it themselves which leads to a lot of vulnerable implementations. But this is not the whole story! While the concept of RSA (often called "textbook RSA") is insecure if implemented naively, even some standards have been found to be insecure!

To understand these issues with RSA, you first need to learn how RSA works.

Remember the multiplicative group of numbers modulo a prime p (we've talked about it in chapter 5). It is the set of strictly positive integers:

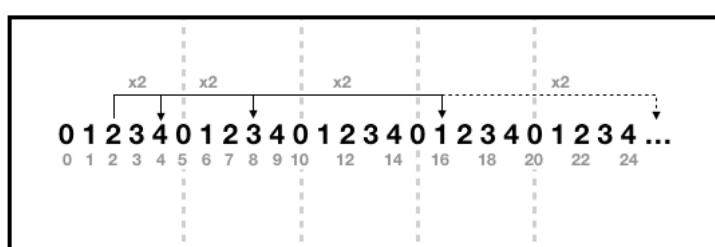
$1, 2, 3, 4, \dots, p-1$

Let's imagine that **one of these numbers is our message**. For p large enough, let's say 4096-bit, our message can contain around 500 characters.

NOTE For computers, a message is just a series of bytes, which can also be interpreted as a number.

We have seen that by exponentiating a number (let's say our message), we can **generate** other numbers which forms a **subgroup**. I illustrate this in figure 6.10.

0 1 2 3 4 integers mod 5
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 ... 0 1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22 24



let's take 2 as a generator, it produces the subgroup $\{2, 4, 3, 1\}$

Figure 6.10 Integers modulo a prime (here 5) are divided in different subgroups. By picking an element as a generator (let's say the number 2) and exponentiating it, we can generate a subgroup. For RSA, the generator is the message.

This is useful for us to define how to encrypt with RSA. To do this, we publish a **public exponent e** (for *encryption*) and a prime number p . To encrypt a message m , one compute

$$\text{ciphertext} = m^e \bmod p$$

For example, to encrypt the message $m=2$ with $e=2$ and $p=5$, we do

$$\text{ciphertext} = 2^2 \bmod 5 = 4$$

And **this is the idea behind encryption with RSA!**

NOTE Usually, a small number is chosen as **public exponent e** so that encryption is fast. Historically, standards and implementations seem to have settled on the prime number 65537 for the public exponent.

This is great, you now have a way for people to encrypt messages to you. But **how do you decrypt?**

Remember, if you continue to exponentiate a generator, you actually go back to the original number (see figure 6.11).

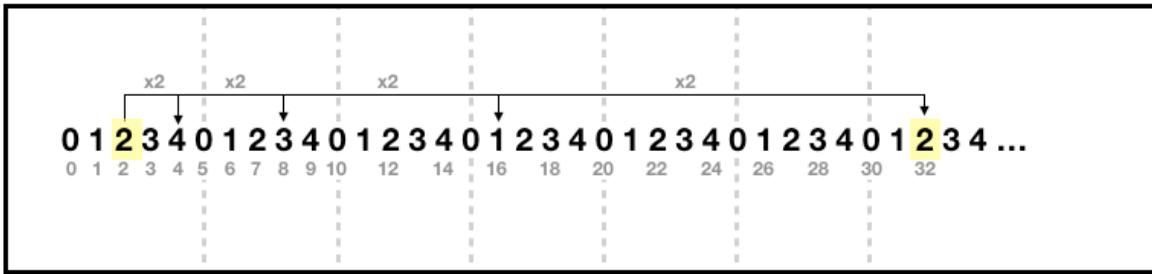


Figure 6.11 Let's say that our message is the number 2. By exponentiating it we can obtain other numbers in our group. If we exponentiate it enough, we go back to our original message 2. We say that the group is cyclic. This property can be used to recover a message after it has been raised to some power.

This should give you an idea of how to implement decryption: find out how much you need to exponentiate a ciphertext in order to recover the original generator (which is the message).

Let's say that you know such a number that we'll call the **private exponent d** (for *decryption*). If you receive ciphertext = message^e mod p, you should be able to raise it to the power d to recover the message:

$$\text{ciphertext}^d = (\text{message}^e)^d = \text{message}^{e \times d} = \text{message} \bmod p$$

The actual mathematic behind finding this private exponent d is a bit tricky, but simply put you compute the inverse of the public exponent modulo the order (number of elements) of the group:

$$d = e^{-1} \bmod \text{order}$$

We have efficient algorithm to compute modular inverses,⁴³ and so this is not a problem.

We do have a problem though!

For a prime p, the order is simply p-1, and thus **the private exponent is easy to calculate for anyone**. This is because every element in this equation, besides d, is public.

NOTE

How did we obtain the previous equation to compute the private exponent d ? **Euler's theorem** states that for m co-prime with p (meaning that they have no common factors)

$$m^{\text{order}} \equiv 1 \pmod{p}$$

For order the number of elements in the multiplicative group created by the integers modulo p . This implies in turn that for any integer multiple

$$m^{1+\text{multiple} \times \text{order}} = m \times (m^{\text{order}})^{\text{multiple}} \pmod{p} = m \pmod{p}$$

This tells us that the equation we are trying to solve

$$m^{e \times d} \equiv m \pmod{p}$$

can be reduced to

$$e \times d = 1 + \text{multiple} \times \text{order}$$

which can be rewritten as

$$e \times d = 1 \pmod{\text{order}}$$

which by definition means that d is the inverse of e modulo order

One way we could prevent others from computing the private exponent from the public exponent is to **hide the order of your group**, and this is the brilliant idea behind RSA: if our modulus is not a prime anymore, but a **product of prime** $N=p \times q$ (with p and q large primes that are known only to you), then **the order of our multiplicative group is not easy to compute anymore as long as you don't know p and q!**

NOTE

The order of the multiplicative group modulo a number N can be calculated with Euler's totient function ($\phi(N)$) which returns the count of numbers that are **coprime** with N . (5 and 6 are coprime for example, because the only positive integer that divides both of them is 1. On the other hand, 10 and 15 are not, because 1 and 5 divides them both.)

The order of a multiplicative group modulo an RSA modulus $N=p \times q$ is $\phi(N) = (p-1) \times (q-1)$ which is too hard to calculate unless you know the factorization of N .

We're all good! To recapitulate, this is how RSA works:

- **Key Generation.**
 - Generate two large prime numbers p and q .
 - Choose a random public exponent e , or a fixed one like $e=65537$.
 - Your public key is the public exponent e and the public modulus $N = p \times q$.
 - Derive your private exponent $d = e^{-1} \pmod{(p-1)(q-1)}$.
 - Your private key is the private exponent d .
- **Encryption.** To encrypt a message compute $\text{message}^e \pmod{N}$.
- **Decryption.** To decrypt a ciphertext compute $\text{ciphertext}^d \pmod{N}$.

Figure 6.12 recapitulates how RSA finally works in practice.



Figure 6.12 RSA encryption works by exponentiating a number (our message) with the public exponent e modulo the public modulus N=pq. RSA decryption works by exponentiating the encrypted number with the private exponent d modulo the public modulus N.

We say that RSA relies on the **factoring problem**. Without the knowledge of p and q, no-one can compute the order, thus no-one but you can compute the private exponent from the public exponent. This is very similar to how Diffie-Hellman was based on the discrete logarithm problem, see figure 6.13.

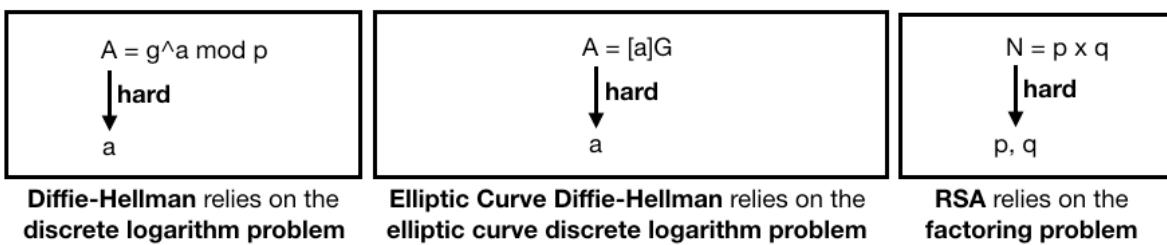


Figure 6.13 Diffie-Hellman, Elliptic Curve Diffie-Hellman and RSA are asymmetric algorithms that rely on three distinct problems in mathematics that we believe to be hard. "Hard" meaning that we do not know efficient algorithms to solve them when instantiated with large numbers.

Thus, textbook RSA works modulo a composite number $N=p\times q$ where p and q are two large primes that need to remain secret.

Now that you understand how RSA works, let's see how insecure it is in practice and what standards do to make it secure.

6.4 Why Not To Use RSA PKCS#1 v1.5

You've learned about "textbook RSA", which is insecure by default for many reasons. Before you can learn about the secure version of RSA, let's see what you need to avoid.

There are many reasons why you cannot use textbook RSA directly, one example is that if you encrypt small messages to Alice (for example $m=2$), then I can encrypt all the small numbers between 0 and 100 (for example) and observe very quickly that the encryption of the number 2 matches your ciphertext. Standards fix this issue by maximizing the size of your message with a **non-deterministic padding** before encrypting it. For example, the **RSA PKCS#1 v1.5** standard

defines a padding that adds a number of random bytes to the message before using RSA to encrypt it. I illustrated this in figure 6.14.

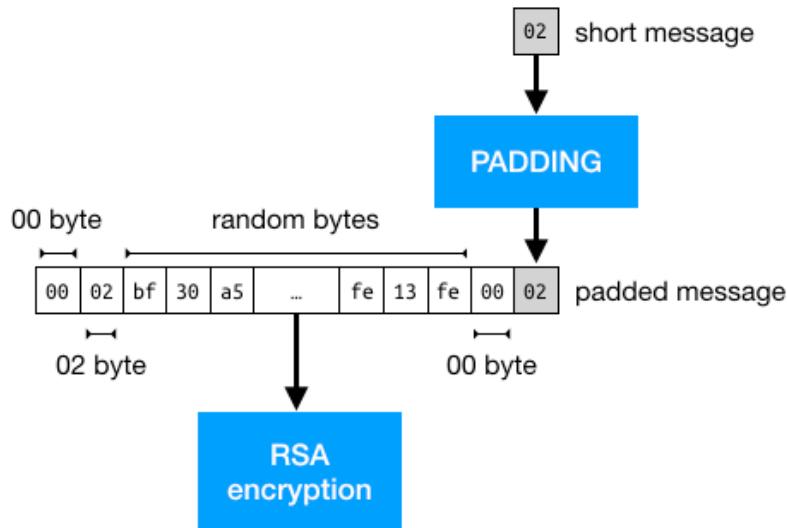


Figure 6.14 The RSA PKCS#1 v1.5 standard specifies a padding to apply to a message prior to encryption. The padding must be reversible (so that decryption can get rid of it) and must add enough random bytes to the message in order to avoid brute-force attacks.

The **PKCS#1** standard is actually the first standard on RSA, published as part of a series of **Public Key Cryptography Standard** written by the RSA company in the early 90's. While the PKCS#1 standard fixes some known issues, in 1998 **Bleichenbacher** found a practical attack on **PKCS#1 version 1.5** that allowed an attacker to decrypt messages encrypted with the padding specified by the standard.⁴⁴ As it required a million messages it is infamously called the **Million Message Attack**. Mitigations were later found, but interestingly along the years the attack has been re-discovered again and again as researchers found that the mitigations were too hard to implement securely (if at all).

NOTE

Bleichenbacher's attack is a type of attack called an **adaptive chosen-ciphertext attack (CCA2)** in theoretical cryptography. CCA2 means that to perform this attack, an attacker is in a context that allows him or her to submit arbitrary RSA encrypted messages (*chosen-ciphertext*), observe how it influences the decryption, and continue the attack based on previous observation (*adaptive*). CCA2 is often used to model attackers in cryptographic security proofs.

To understand why the attack was possible, you need to understand that RSA ciphertexts are **malleable**: you can tamper with an RSA ciphertext without invalidating its decryption.

If I observe the ciphertext $c = m^e \text{ mod } N$, then I can submit the following ciphertext:

$$3^e \times m^e = (3m)^e \text{ mod } N$$

which will decrypt as:

$$((3m)^e)^d = (3m)^{e \times d} = 3m \bmod N$$

I used the number 3 as an example, but I can multiply the original message with whatever number I want. In practice, a message must be well-formed (due to the padding) and thus, tampering with a ciphertext should break the decryption. Nevertheless, it happens that sometimes, even after that transformation, the padding is accepted after decryption.

Bleichenbacher made use of this property in his Million Message Attack on RSA PKCS#1 v1.5. His attack works by intercepting an encrypted message, modifying it and sending it to the person in charge of decrypting it. By observing if that person can decrypt it (the padding remained valid), we obtain some information about the range in which the message is (since the first two bytes are `0x0002`, we know that the decryption is smaller than some value). By doing this iteratively, we can narrow that range down to the original message itself.

NOTE

Interestingly, encryption schemes exhibiting such properties are part of a field called **homomorphic encryption** and sometimes desired for their privacy aspect. For example, encrypting your data with an homomorphic encryption scheme before uploading it to a cloud service could allow you to store your data without the cloud being able to understand any of it. In addition, the cloud could be able to compute functions of interest over your encrypted data and return the encrypted results for you to decrypt.

Eventhough the Bleichenbacher attack is well-known, there are still many systems in use today that implement RSA PKCS#1 v1.5 for encryption. As part of my work as a security consultant, I found many applications that were vulnerable to this attack, so be careful!

6.5 Asymmetric Encryption with RSA-OAEP

In 1998, version 2.0 of the same PKCS#1 standard was released with a new padding scheme for RSA called **Optimal Asymmetric Encryption Padding (OAEP)**. Unlike its predecessor — PKCS#1 v1.5 — OAEP is not vulnerable to Bleichenbacher's attack and is thus a strong standard to use for RSA encryption nowadays. Let's see how OAEP works and prevents the previously discussed attacks.

First, let's mention that like most cryptographic algorithms OAEP comes with a key generation algorithm that takes a security parameter (as illustrated in figure 6.15).

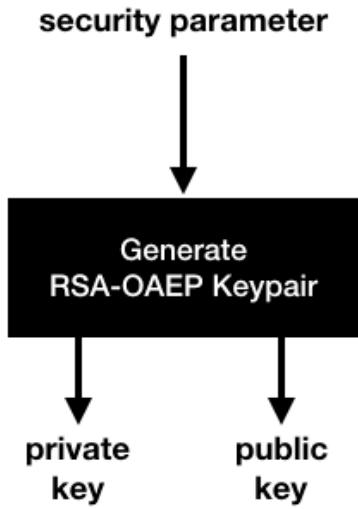


Figure 6.15 RSA-OAEP, like many public-key algorithms, first need to generate a keypair that can be used later in the other algorithms provided by the cryptographic primitive.

This algorithm takes a security parameter, which is a number of bits. As with Diffie-Hellman, operations happen in the set of numbers modulo a large number. When we talk about the security of an instantiation of RSA, we usually refer to the size of that large modulus. This is very similar to Diffie-Hellman if you remember. Currently, most guidances (www.keylength.com) estimate a modulus between 2048 and 4096 bits to provide 128-bit security. As these estimations are quite different, most applications seem to conservatively settle on 4096-bit parameters.

NOTE

We've seen that RSA's large modulus is not a prime, but a product $N=p \times q$ of two large prime numbers p and q . For a 4096-bit modulus, the key generation algorithm typically splits things in the middle and generates both p and q of size approximately 2048-bit.

To encrypt, the algorithm first pads the message and mixes it with a random number generated per-encryption. The result is then encrypted with RSA. To decrypt the ciphertext, the process is reversed as can be seen in figure 6.16.

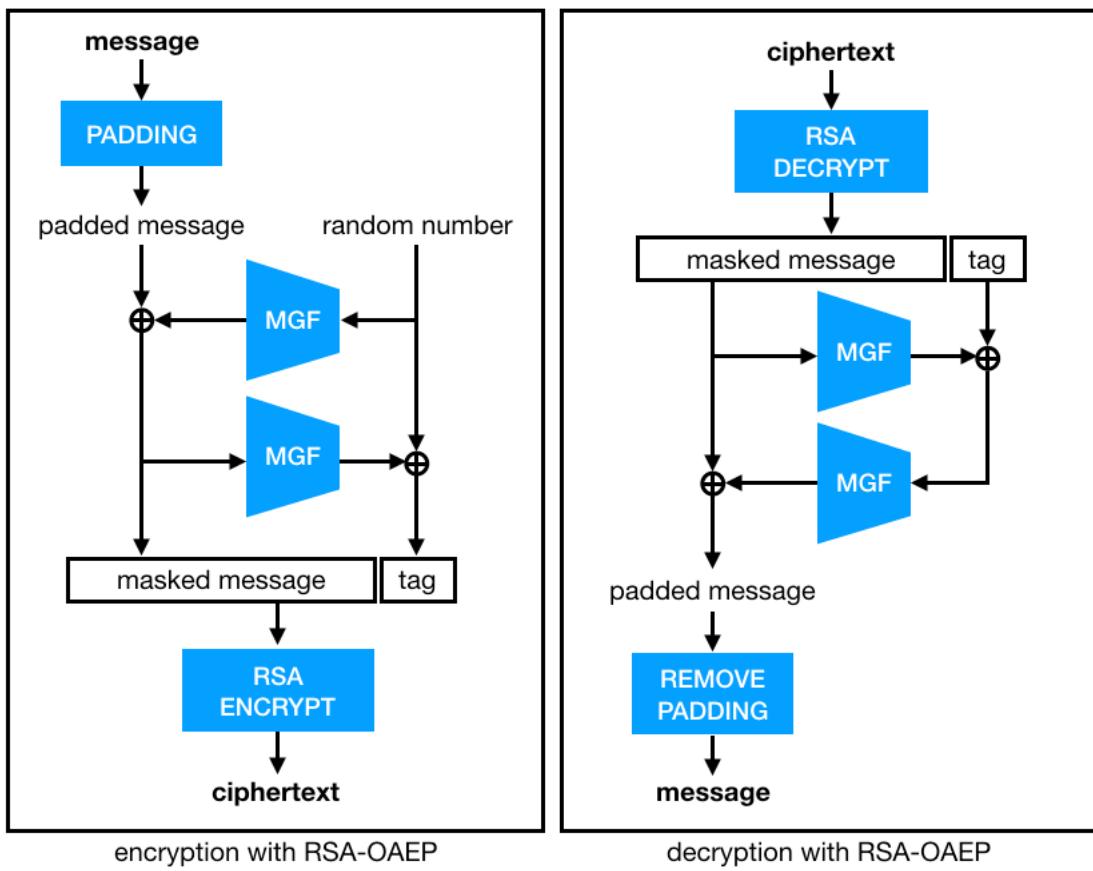


Figure 6.16 RSA-OAEP works by mixing the message with a random number prior to encryption. The mixing can be reverted after decryption. At the center of the algorithm, a Mask Generation Function (MGF) is used to randomize and enlarge or reduce an input.

RSA-OAEP uses this mixing in order to make sure that, if a few bits of what is encrypted with RSA leak, no information on the plaintext can be obtained. Indeed, to reverse the OAEP padding, you need to obtain (close to) all the bytes of the OAEP padded plaintext.

In addition, Bleichenbacher's attack should not work anymore as the scheme makes it impossible to obtain a well-formed plaintext by modifying a ciphertext.

NOTE

The property that it is too difficult for an attacker to create a ciphertext that will successfully decrypt (of course without the help of encryption) is called **plaintext-awareness**. Due to the plaintext-awareness provided by OAEP, Bleichenbacher's attack do not work on the scheme.

Inside of OAEP, **MGF** stands for **Mask Generation Function**. In practice, a MGF is an **extendable output function (XOF)** (you've learned about them in chapter 2). As MGFs were invented before the sponge construction, they simply make use of a hash function that absorbs the input repeatedly with a counter (see figure 6.17).

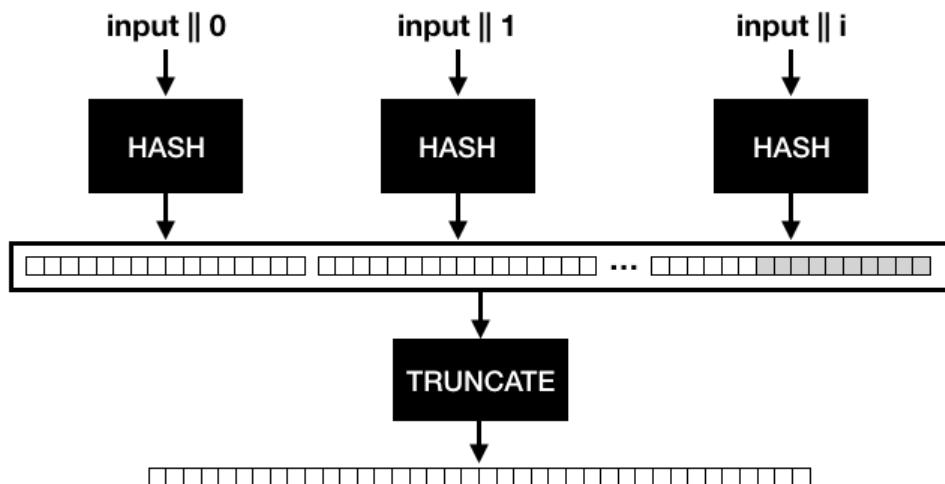


Figure 6.17 A Mask Generation Function (MGF) is simply a function that takes an arbitrary-length input, and produces a random-looking arbitrary-length output. It works by hashing an input and a counter, concatenating the digests together, and truncating the result to obtain the length desired.

And this is how OAEP works.

NOTE

Only 3 years after the release of the OAEP standard, James Manger found a timing attack similar to Bleichenbacher's Million Message attack (but much more practical) on OAEP if not implemented correctly. Fortunately, it is much simpler to securely implement OAEP compared to PKCS#1 v1.5 and vulnerabilities in this scheme's implementation are much more rare.

Furthermore the design of OAEP is not perfect, better constructions have been proposed and standardized over the years. One example is **RSA-KEM** which has stronger proofs of security and is much simpler to implement securely. You can observe how much more elegant the design is in figure 6.18.

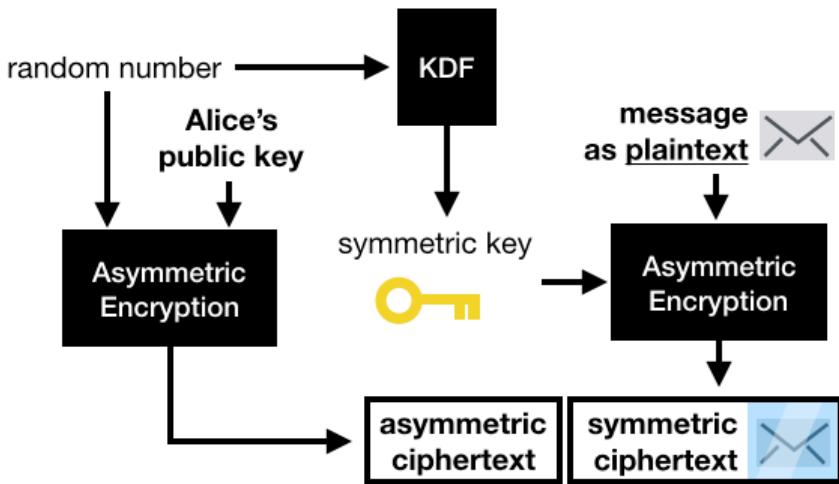


Figure 6.18 RSA-KEM is an encryption scheme that works by simply encrypting a random number under RSA. No padding is needed. The random number can be used as a symmetric key after being passed through a Key Derivation Function (KDF). The symmetric key is then used to encrypt a message via an authenticated encryption algorithm.

Note the **Key Derivation Function (KDF)** in-use here. It is another cryptographic primitive that can be replaced with a MGF or a XOF. I'll talk more about what KDFs are in chapter 8 on randomness and secrets.

Nowadays most protocols and applications that use RSA either still implement the insecure PKCS#1 v1.5 or OAEP. On the other hand, more and more protocols are moving away from RSA encryption in favor of Elliptic Curve Diffie-Hellman (ECDH) for both key exchanges and hybrid encryption. This is understandable as ECDH provides shorter public keys and benefits in general from much better standards and much safer implementations.

6.6 Hybrid Encryption with ECIES

While there exist many hybrid encryption schemes, the most widely adopted standard is **Elliptic Curve Integrated Encryption Scheme (ECIES)**. The scheme has been specified to be used with Elliptic Curve Diffie-Hellman, and has been included in many standards like ANSI X9.63, ISO/IEC 18033-2, IEEE 1363a, and SECG SEC 1. Unfortunately, every standard seems to implement a different variant, and different cryptographic libraries will implement hybrid encryption differently in part due to this.

For this reason, I've rarely seen two similar implementations of hybrid encryption in the wild. It is important to understand that while this is annoying, if all the participants of the protocol use the same implementation or document the details of the hybrid encryption scheme they have implemented, then there are no issues.

ECIES works very similarly to the hybrid encryption scheme I've explained in section 6.2. The difference is that we implement the Key Encapsulation Mechanism (KEM) part with an ECDH

key exchange instead of with an asymmetric encryption primitive.

Let's explain this step by step. First, if you want to encrypt a message to Alice, you use an (EC)DH-based key exchange with Alice's public key and a keypair that you generate for the occasion (this is called an **ephemeral keypair**). You can then use the obtained shared secret with an authenticated symmetric encryption algorithm like AES-GCM to encrypt a longer message to her. This is illustrated in figure 6.19.

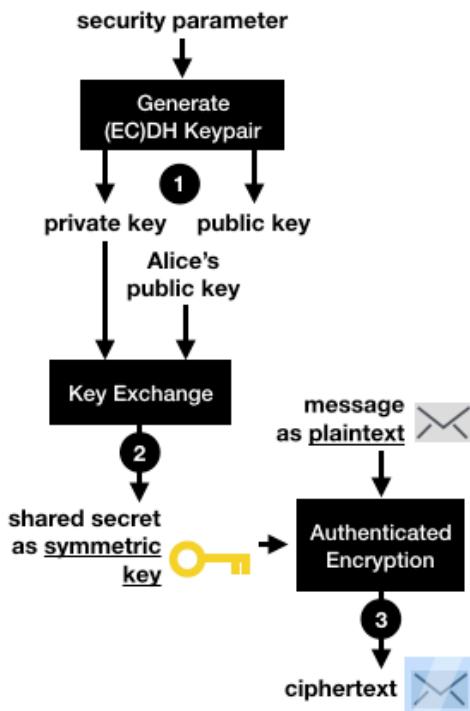


Figure 6.19 To encrypt a message to Alice using Hybrid Encryption with (EC)DH, you first (1) generate an ephemeral (Elliptic Curve) Diffie-Hellman key pair. (2) Perform a key exchange with your ephemeral private key and Alice's public key. (3) Use the resulting shared secret as a symmetric key to an authenticated encryption algorithm to encrypt your message.

After this, you can send the ephemeral public key and the ciphertext to Alice. Alice can use your ephemeral public key to perform a key exchange with her own keypair. She can then use the result to decrypt the ciphertext and retrieve the original message. The result is either the original message, or an error if the public key or the encrypted message were tampered in transit. The full flow is illustrated in figure 6.20.

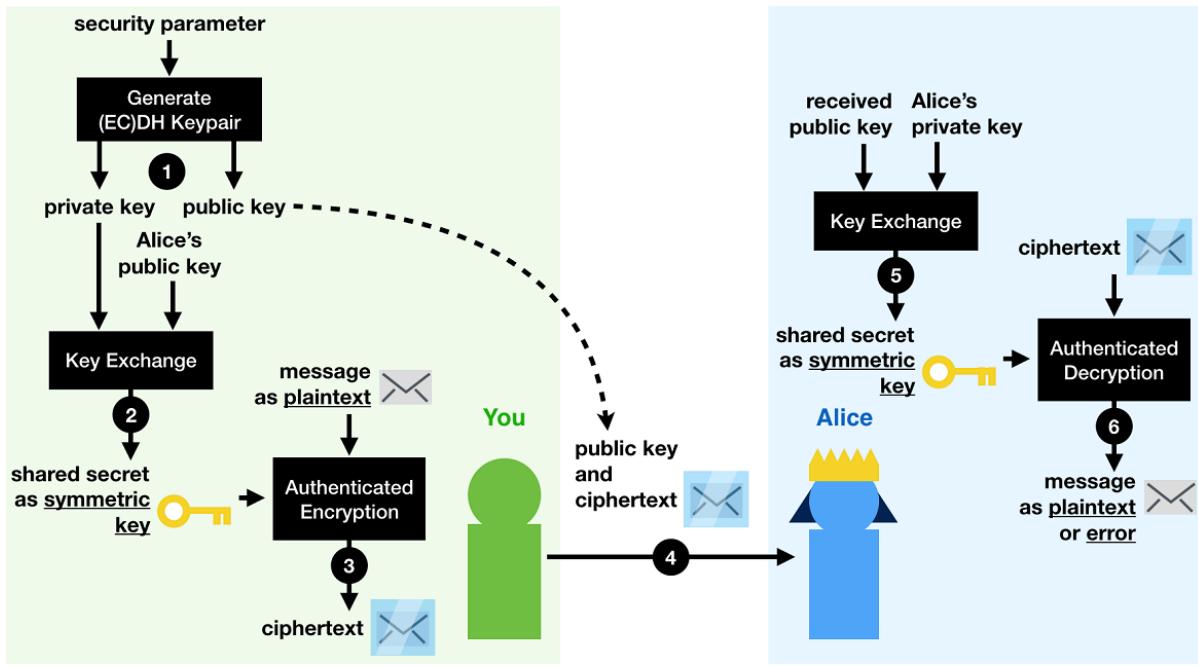


Figure 6.20 Building on figure , After (4) you send your ephemeral public key and your encrypted message to Alice. (5) Alice can perform a key exchange with her private key and your ephemeral public key. (6) She finally use the resulting shared secret as a symmetric key to decrypt the encrypted message with the same authenticated encryption algorithm.

And this is pretty much how ECIES work. There also exist a variant of ECIES using Diffie-Hellman called IES that works pretty much the same way, but not many people seem to use it.

NOTE

Note that I simplified the diagram above. Most authenticated encryption primitives expect an **uniformly-random** symmetric key. Since the output of a key exchange is generally not uniformly-random, we need to pass the shared secret through a Key Derivation Function (KDF) or a XOF beforehand. You will learn more about this in chapter 8. Not uniformly random here means that the probability that each bit of the key exchange result is 0 or 1 is not balanced. The first bits might always be set to 0 for example. This is an inherent consequence of using mathematics! **Can you see why?**

And that's it for the different standards you can use.

6.7 Summary

- Asymmetric encryption is rarely used to encrypt messages directly. This is due to the relatively low size limitations of the data that asymmetric encryption can encrypt.
- Hybrid Encryption can encrypt much larger messages by combining asymmetric encryption, or a key exchange, with a symmetric authenticated encryption algorithm.
- The RSA PKCS#1 v1.5 standard for asymmetric encryption is broken in most settings. Prefer the RSA-OAEP algorithm standardized in RSA PKCS#1 v2.2.
- ECIES is the most widely used hybrid encryption scheme. It is preferred over RSA-based schemes due to its parameter sizes and its reliance on solid standards.
- Different cryptographic libraries might implement hybrid encryption differently. This is not a problem in practice if interoperable applications use the same implementations.

7

Signatures and zero-knowledge proofs

This chapter covers

- a new cryptographic primitive called a signature, which is non-interactive zero-knowledge proof used to simulate pen-and-paper signatures.
- The existing standards for signatures and how they are implemented in real-world applications.
- The subtle behaviors of signatures and how you can avoid their pitfalls.

So far, you've learned that symmetric cryptographic primitives like **authenticated encryption** algorithms are useful as well as efficient to provide **confidentiality and integrity** to your messages. You've also learned that **they don't scale well**, as they require you to share a symmetric secret with every person you're talking to. For example, how can your browser manage to share symmetric secrets with every website on the internet? It sounds like an impossible task. Chapters 5 and 6 have introduced a couple of **asymmetric cryptographic primitives** that provide ways for two participants to **agree on a secret**, a secret that can then be used as a symmetric key by our authenticated encryption algorithms. Yet, these cryptographic primitives still don't solve the real-world problem of scalability. We are now reaching a very interesting point in this book. The cryptographic primitive that you are going to learn in this chapter — **digital signature** — is one of the best answers we have found to that scalability problem.

NOTE

Signatures in cryptography are often referred to as **digital signatures**, or **signature schemes**. In this book, I interchangeably use these terms.

Nonetheless, this is not the only use for digital signatures, and we will first focus on what they are, and what you can do with them in this chapter. In the second part of this book you will cross

paths with digital signatures over and over as cryptographic protocols make heavy use of them. So pay attention, you're going to learn about one of the most useful and ubiquitous building blocks of cryptography.

For this chapter you'll need to have read:

- Chapter 2 on Hash Functions.
- Chapter 5 on Key Exchanges.
- Chapter 6 on Asymmetric Encryption.

7.1 What Is a Signature?

I explained in chapter 1 that cryptographic signatures are pretty much like real life signatures (the one you write on checks or contracts). For this reason, they are usually one of the most intuitive cryptographic primitives to understand:

- only you can sign arbitrary messages (or other words, nobody should be able to forge your signature)
- anybody can verify a signature on messages that you've signed

As we're in the realm of asymmetric cryptography, you can probably guess how this asymmetry is going to take place. Three algorithms are defined in a signature algorithm or signature scheme:

1. A **keypair generation** algorithm that a signer can use to create a new private and public key (the public key can then be shared with anyone).
2. A **signing** algorithm that takes a private key and a message to produce a signature.
3. A **verifying** algorithm that takes a public key, a message, and a signature and returns a success or error message.

Sometimes the private key is also called the **signing key**, and the public key is called the **verifying key**. Makes sense right? I recapitulate this in figure 3.4.

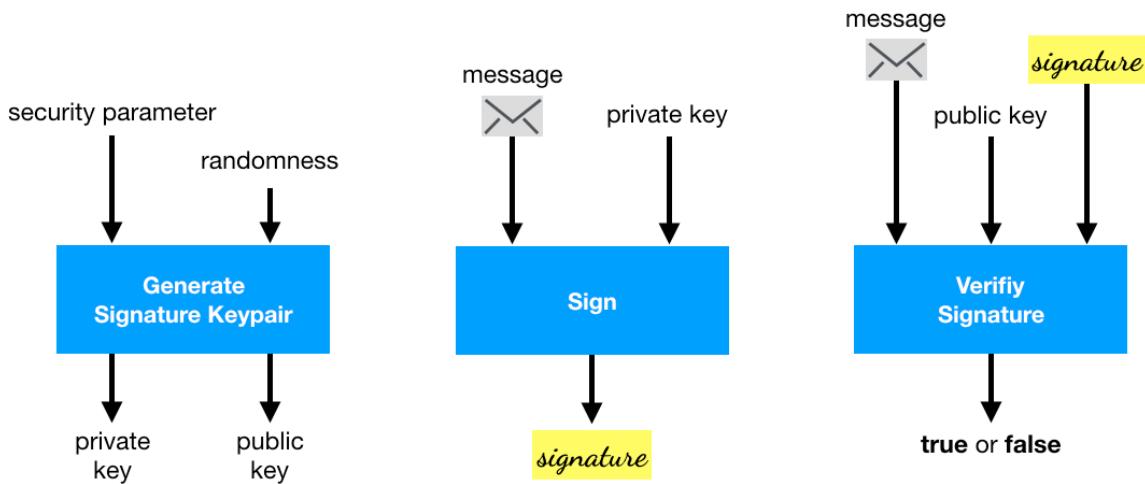


Figure 7.1 The interface of a digital signature. Like other public-key cryptographic algorithms you first need to generate a keypair via a key generation algorithm that takes a security parameter and some randomness. A signing algorithm can then be used with the private key to sign a message, and a verifying algorithm can be used with the public key to verify a signature over a message. Nobody should be able to forge a message under someone else's public key.

Knowing this, what are signatures good for? Well, they are good for authentication!

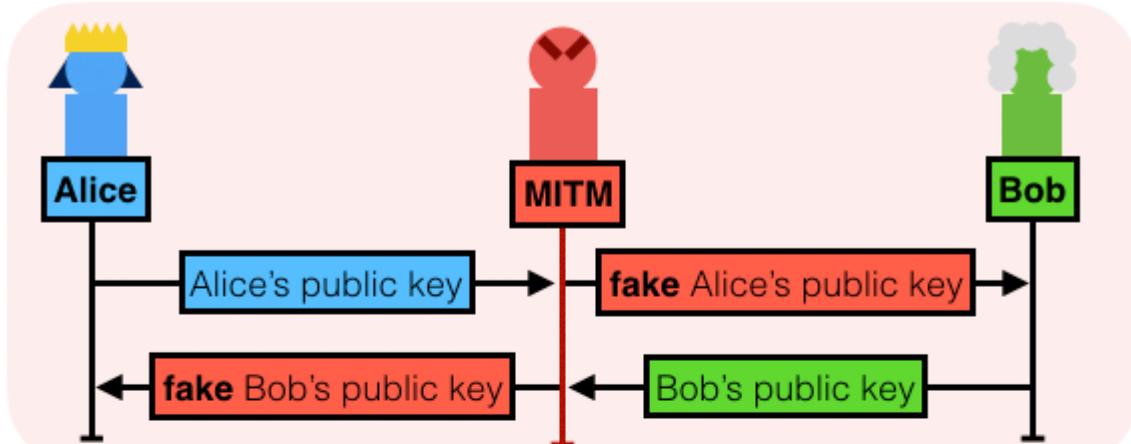
In this sense, signatures are somewhat similar to the message authentication codes (MACs) you've learned about in chapter 3. But unlike MACs, they allow one to **authenticate messages asymmetrically**: a participant can verify that a message hasn't been tampered, without knowledge of the private or signing key.

NOTE As you've seen in chapter 3 on Message Authentication Codes, authentication tags produced by MAC functions must be verified in **constant-time** to avoid timing attacks. This idea of implementing cryptographic algorithms in constant-time is prevalent in cryptography in order to prevent against side-channel attacks. Do you think we need to do the same for verifying signatures?

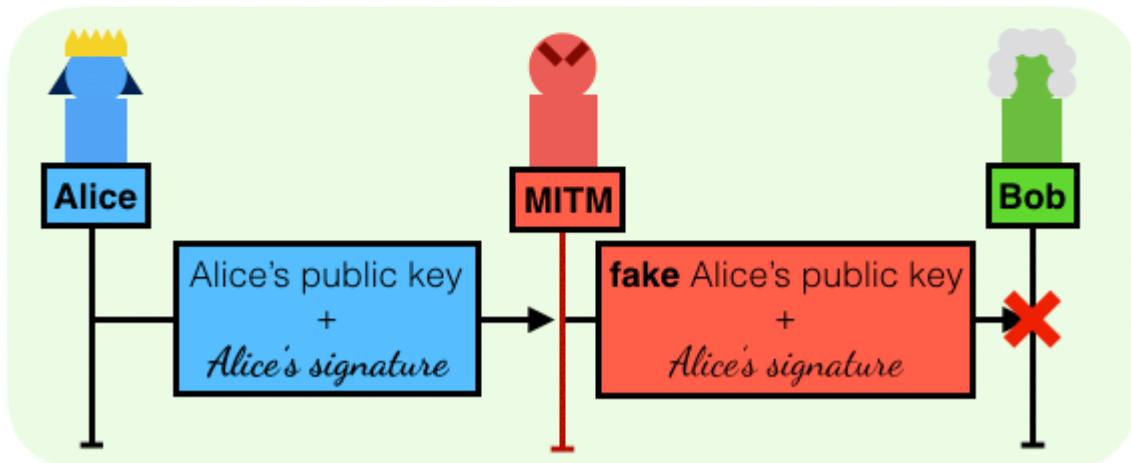
Signatures are also very good tools to simply **prove who someone is** in a protocol. You'll learn more about this, and how protocols make heavy use of signatures in part 2 of this book, but for now let's focus on some intuition.

Remember how an active man-in-the-middle attacker could trivially impersonate both sides of a key exchange in chapter 5? Signatures are a good way to prevent this. For example, if Bob is aware of Alice's verifying/public key, she can use this to authenticate her side of the key exchange. To do this, she can sign the public key she sends with her signing/private key, and

Bob can verify that the signature is valid using the associated verifying key. If the signature is invalid, Bob can tell someone is actively man-in-the-middleing the key exchange (as illustrated in figure 7.2).



unauthenticated key exchange



authenticated key exchange

Figure 7.2 The first picture represents an unauthenticated key exchange, which is insecure to an active man-in-the-middle attacker who can trivially impersonate both sides of the exchange by swapping their public keys with its own. The second picture represents the first part of the key exchange authenticated by Alice's signature over her public key. As Bob (who knows Alice's verifying key) is unable to verify the signature after the message was tampered by the man-in-the-middle attacker, he aborts the key exchange.

This is of course not super useful (yet). If Alice has shared her verifying key with Bob prior to the key exchange, she could also have shared her key exchange public key.

A more useful use of signatures is to assume trust is **transitive**. This means that if you trust some authority, and the authority trusts Alice, then you can trust Alice. With this, you can potentially trust the public key of not just Alice, but of Charles, David, Eve, and so on, as long as they were signed by this authority you trust. I illustrate this in figure 7.3.

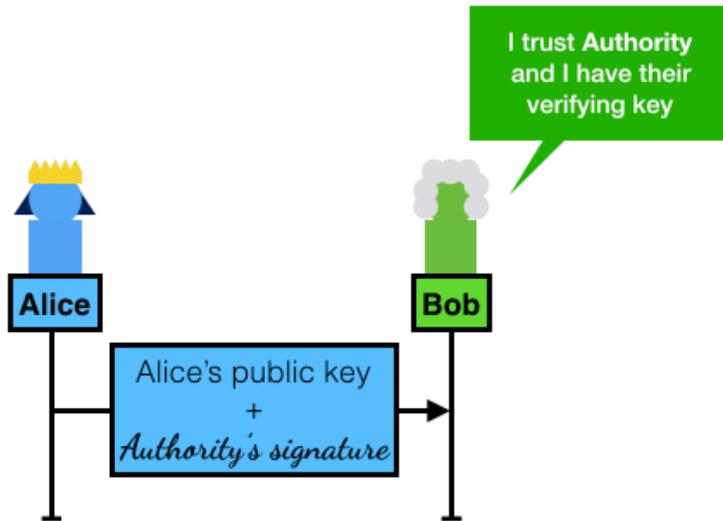


Figure 7.3 The first part of a key exchange with Alice where her public key and her identity have been signed by an authority you trust. Even though you did not talk to Alice prior to the key exchange, you can choose to trust that this is Alice due to the Authority's signature. Note that if the signature of the authority does not cover some identity (like the string "Alice") and only covers a public key, then you only know that you're talking to a public key that the authority trust. You can make no claim about who the authority think it is associated to.

This is a very important concept that you'll see more of in later parts of this book. For example, in chapter 9 you will learn that signatures are used to scale trust on the web, or in other words to allow browsers to authenticate key exchanges people perform with the millions of websites they visit.

To give you a preview, imagine that your browser trusts some authority (when you download a browser, it comes with some authority verifying key baked into the program). This authority's responsibility is to sign thousands of websites' keys, so that you can trust them without knowing about them, as long as they provide a signature from this authority over their public keys and domain (like example.com). Figure 7.4 represents this model.

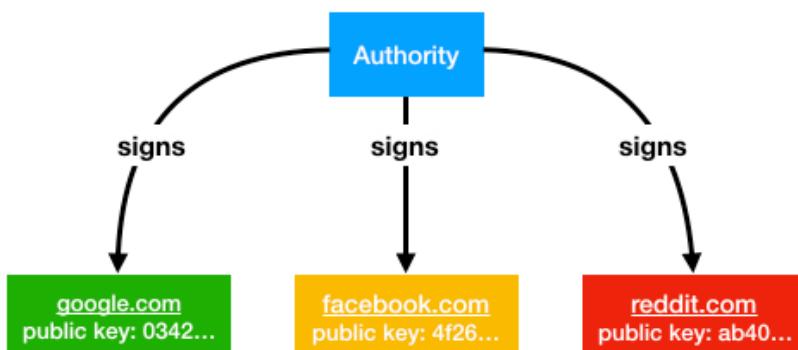


Figure 7.4 An authority signs the public keys of several websites it trusts. If you trust the authority's public key, you can verify the signatures and choose to trust the websites and their public keys as well.

This is a simplified version of what is really happening, you will learn more about this type of systems (called public-key infrastructures) in chapter 9 on transport security.

Let's now see a practical example on how you can use a signature scheme in practice. I chose to use the pyca/cryptography cryptographic library for the Python programming language.⁴⁵ It is a pretty complete and multi-purpose cryptographic library that is used in many real-world applications. The following example simply generate a keypair, signs a message using the private key part, then verifies the signature using the public key part.

Listing 7.1 signature.py contains code to sign and verify signatures

```
from cryptography.hazmat.primitives.asymmetric.ed25519 import Ed25519PrivateKey ①

private_key = Ed25519PrivateKey.generate() ②
public_key = private_key.public_key() ②

message = b"example.com has the public key 0xab70..." ③
signature = private_key.sign(message) ③

try: ③
    public_key.verify(signature, message) ④
    print("valid signature") ⑤
catch InvalidSignature: ⑤
    print("invalid signature") ④
```

- ① We use the Ed25519 signing algorithm which is one of the popular signature scheme.
- ② With pyca/cryptography you first generate the private key, then generate the public key.
- ③ Using the private key we can sign a message and obtain a signature.
- ④ We verify the signature over the message using the public key part.

In this section you have learned about signatures from a high-level point of view. Let's dig deeper into how signatures really work, but for this we need to make a detour and take a look at zero-knowledge proofs first.

7.2 What Are Zero-Knowledge Proofs? And What Does This Have To Do With (Schnorr) Signatures?

The best way to understand how signatures work in cryptography, is to understand where they come from. For this reason, let's take a moment to briefly introduce **zero-knowledge proofs (ZKPs)** and then I'll get back to signatures.

Imagine that Alice (the **prover**) wants to prove something to Bob (the **verifier**). For example, imagine that she wants to prove that she knows the discrete logartihm of a group element, or more precisely she wants to prove that she knows x if $Y = g^x$ and g is a generator of a group (as in figure 7.5).

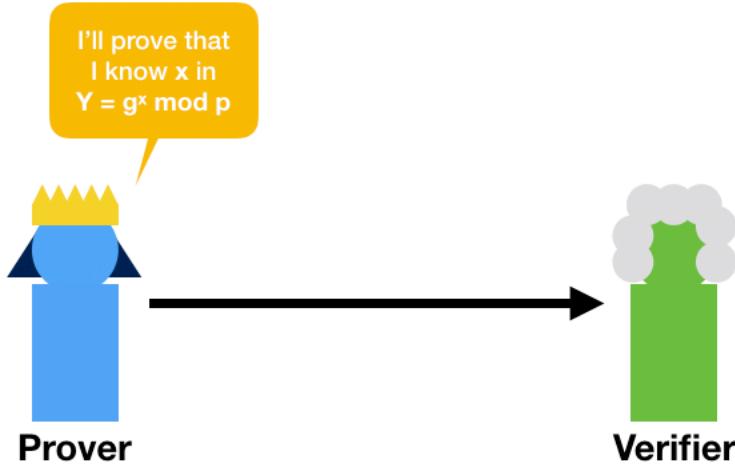


Figure 7.5 A proof of knowledge has two participants. Alice (the prover) wants to prove that she knows the discrete logarithm of a group element to Bob (the verifier).

Of course, the simplest approach to this problem is for Alice to simply send the value x (called the **witness**) as in figure 7.6. This would be a simple proof of knowledge, and this would be OK unless Alice does not want Bob to learn it.

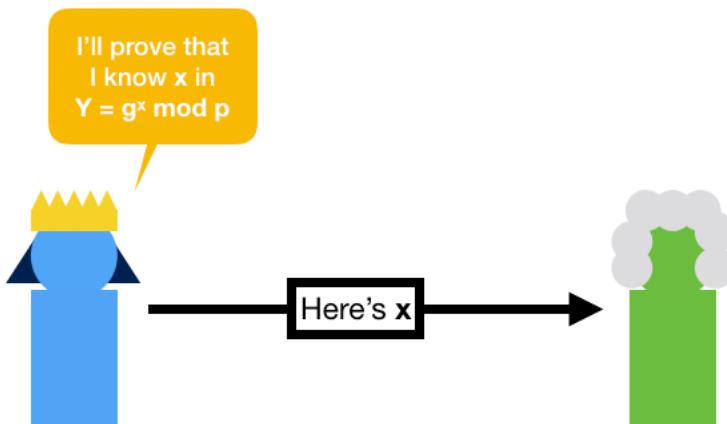


Figure 7.6 To prove that she knows the discrete logarithm x to a group element, Alice can simply send x .

And this works!

NOTE

In theoretical terms, we say that the scheme is **complete** if Alice can use it to prove to Bob that she knows the witness. If she can't use it to prove what she knows, then the scheme is useless right?

In cryptography, we're mostly interested in proofs of knowledge that are **zero-knowledge** (hence the name zero-knowledge proofs). This means that Alice wants to prove to Bob that she knows x without having to reveal x .

One way to approach this problem in cryptography is to "hide" the value with some randomness.

We call this approach **blinding**, which is quite a common technique in cryptography.

To do this, Alice can try to simply add the witness to a value k that she generates uniformly at random called a **blinding factor**. The result $s = k + x$ along with the blinding factor k can both be sent to Bob instead of x , and Bob can then use some arithmetic to indeed verify that Alice knows x (as illustrated in figure 7.7). Indeed if $g^s (=g^{k+x} = g^k \times g^x)$ is equal to $Y \times g^k (=g^x \times g^k)$ then she must know x .

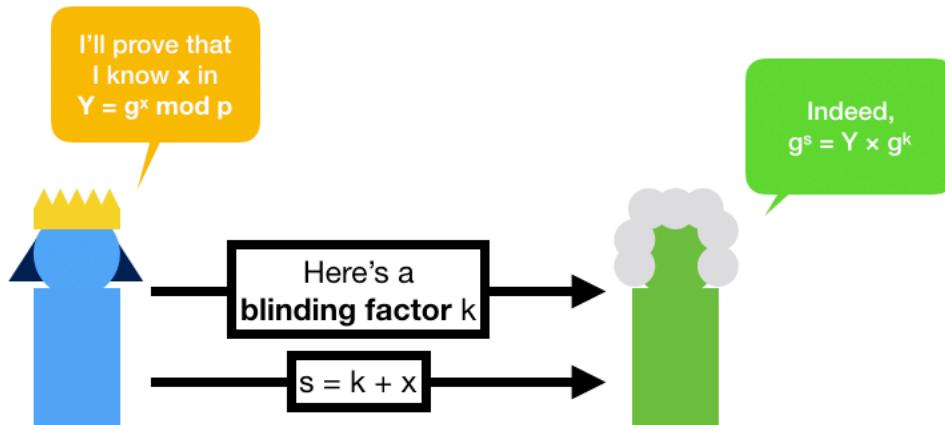


Figure 7.7 In order to hide the witness (what she wants to prove knowledge of), Alice attempts to hide it with a random value called a blinding factor.

The problem with this scheme, is that it's not secure. Indeed since the equation hiding the witness x only has one unknown (x itself), Bob can simply reverse the equation to retrieve x (as illustrated in figure 7.8). Indeed x simply is $s - k^{-1}$ (often written $s - k$).

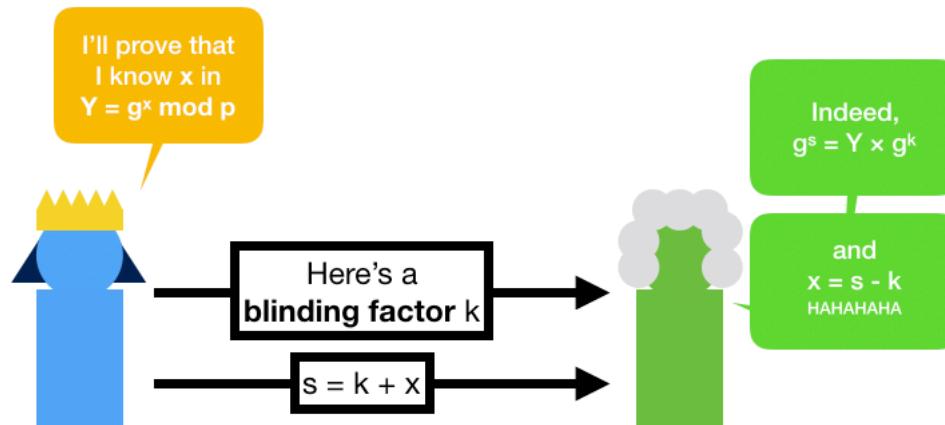


Figure 7.8 In a zero-knowledge proof, simply adding a random value to the witness does not add the zero-knowledge property to the scheme, as the verifier can easily reverse the equation to recover the witness.

The way ZKPs for discrete logarithms fix this is by hiding the blinding factor itself! To do this, Alice can use another hiding construction (you've learned about in chapter 2): a **commitment scheme**. By sending a commitment to the blinding factor $R = g^k$, Bob does not learn the value k

that would allow him to recover the witness x , but still has enough information to verify that Alice knows x (as illustrated in figure 7.9). By verifying that g^k ($= g^{k+x} = g^k \times g^x$) is equal to $Y \times R$ ($= g^x \times g^k$), Bob can see that only someone with the knowledge of x could have computed such values R and s .

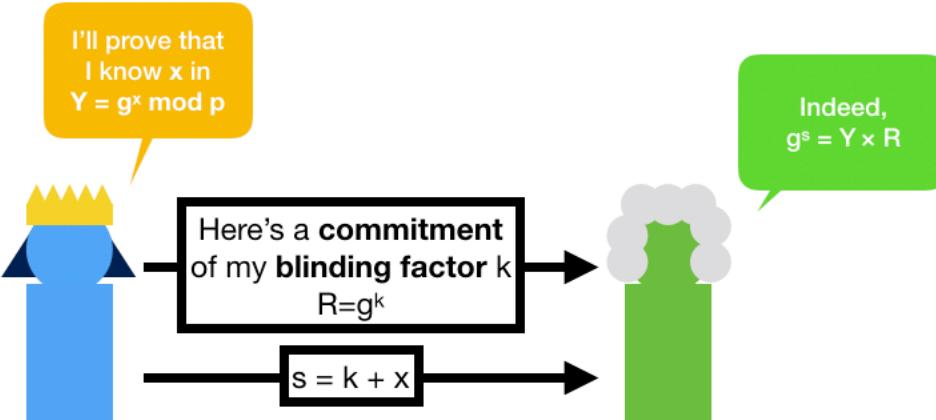


Figure 7.9 To make a knowledge proof "zero-knowledge", the prover can hide the witness with a blinding factor that is itself committed.

There is one last issue with our scheme: Alice can cheat! This is pretty bad, but how can she do that? Easy, she can just choose a random value s , and then calculate the value r based on s as $g^k \times Y^{-1}$ (sometimes written as g^k / Y) as illustrated in figure 7.10. Bob computes $Y \times R = Y \times g^k \times Y^{-1}$ which indeed matches g^k .

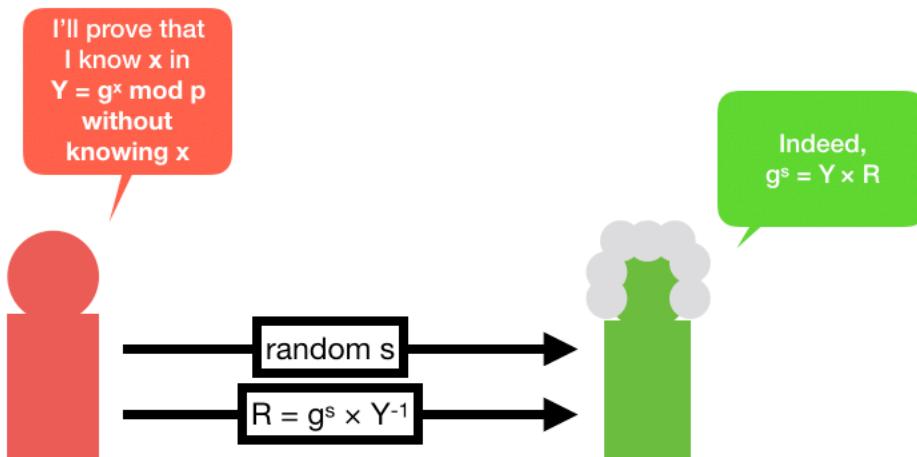


Figure 7.10 The zero-knowledge proof scheme previously discussed is broken, as a prover can easily cheat and fool the verifier. This is possible because the prover has too much freedom in choosing the parameters in the protocol.

To fix this last issue, the protocol must be made **interactive**: Bob sends a random challenge to Alice after she commits to her blinding factor. Alice must use the random challenge in the computation of the proof s , canceling the previous attack.

NOTE

In theoretical terms, we say that the scheme is **sound** if Alice cannot cheat, meaning that if she doesn't know x then she can't fool Bob.

The obtained protocol (illustrated in figure 7.11) is often referred to as **Schnorr identification protocol**. These types of protocols following a three-move pattern (commitment, challenge, proof) are called **Sigma protocols** (sometimes written Σ -protocol) in the literature.

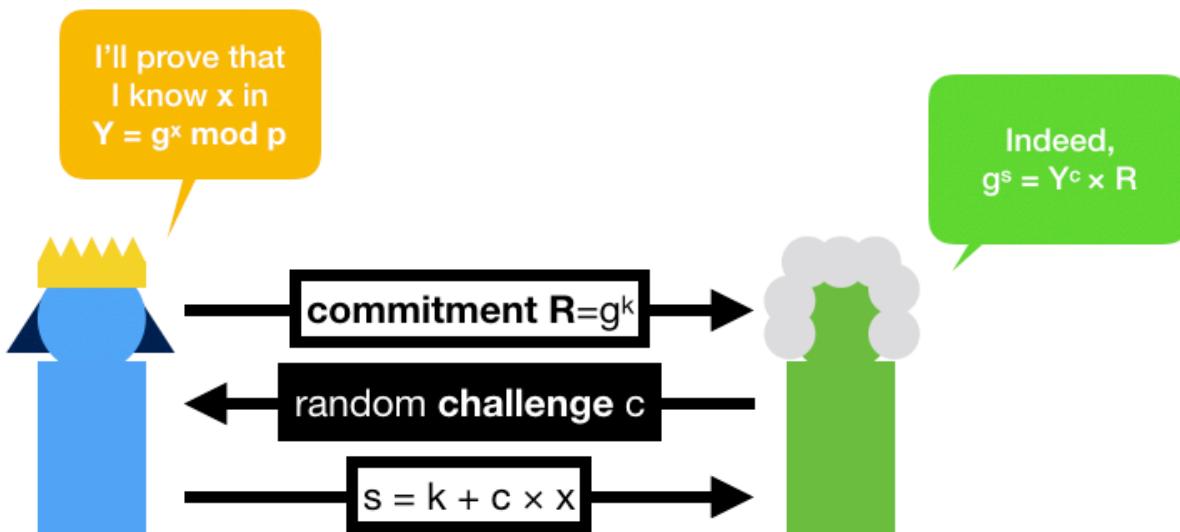


Figure 7.11 The Schnorr identification protocol is an interactive zero-knowledge proof that is complete (Alice can prove she knows some witness), sound (Alice cannot prove anything if she doesn't know the witness), and zero-knowledge (Bob learns nothing).

And this is what **interactive zero-knowledge proofs** are about.

NOTE

Schnorr identification protocol works in the honest verifier zero-knowledge (HVZK) model. In this model, we expect the verifier (Bob) to randomly pick a challenge, instead of maliciously choosing it. Some stronger zero-knowledge proof schemes are zero-knowledge even when the verifier is malicious, so this is an important distinction.

The problem with these is that they are **interactive**. This means some constraints that real-world protocols have to comply to:

- both participants must be present for the scheme to work (in other word the protocol must be **synchronous**)
- the scheme requires one **round-trip** and a half (where Alice sending the first message and then receiving the challenge from Bob represents one round trip)

Most real-world protocols are obsessed with working in **asynchronous** settings (both participants don't have to be online for the protocol to work). For this reason, interactive

zero-knowledge proofs have been mostly absent from the world of applied cryptography.

All of this is not for nothing though!

In 1986 Amos Fiat and Adi Shamir published a technique that allowed one to **convert an interactive ZKP into a non-interactive ZKP**. The trick they introduced (referred to as the **Fiat-Shamir heuristic or transform**) is to compute the challenge in a way we can't control. The theoretical way to achieve this is to use random oracles, and the practical way to emulate such random oracles is to use hash functions (you've learned this in chapter 2).

Here's the trick: **compute the challenge as a hash of the commitment**. And that's it!

If we assume that the hash function gives outputs that are random enough, then it can successfully simulate a verifier.

Schnorr went a step further, he noticed that:

- anything can be included in the digest
- hash functions can be used as commitment schemes as well

For example, what if we included a message in there? (Such that the challenge is now computed as a hash of the commitment and a message.) What we obtain is not only a proof that we know some witness x , but a commitment to a message that is cryptographically linked to the proof. In other words, if the proof is correct then only someone with the knowledge of the witness x could have committed that message.

That's a signature!

Digital signatures are just non-interactive ZKPs.

Applying the fiat-shamir transform to the Schnorr identification protocol, we obtain the **Schnorr signature** scheme. I illustrate this in figure 7.12.



Figure 7.12 The first picture is the schnorr identification protocol previously discussed, which is an interactive protocol. Below it is a non-interactive version of it, also called a Schnorr signature, where the verifier message has been replaced by a call to a hash function.

To recapitulate, A Schnorr signature is essentially two values:

- R. A group element representing the commitment to some secret random value. This is often called a nonce has it needs to be unique per-signature.
- s. A proof computed over the commitment R, a message, and a private key (the witness x).

The Schnorr signature scheme has not been used much in real-world applications due to patents, but the scheme has re-gained some popularity after the patents expired in 2008.

Let's look next at other standards and popular signature algorithms.

7.3 The Signature Algorithms You Should Use (Or Not)

Like other fields in cryptography, digital signatures have many standards and it is sometimes hard to understand which one to use. This is why I'm here! Fortunately, the types of algorithms for signatures is very similar to the ones for key exchanges: there are algorithms based on arithmetic modulo a large number (like Diffie-Hellman and RSA), and there are algorithms based on elliptic curves (like Elliptic Curve Diffie-Hellman). So be sure you understand the algorithms in chapter 5 and chapter 6 well enough, as we're going to build upon them.

Interestingly, the paper introducing the Diffie-Hellman key exchange in 1976 also introduces the concept of digital signatures without a solution:

In order to develop a system capable of replacing the current written contract with some purely electronic form of communication, we must discover a digital phenomenon with the same properties as a written signature. It must be easy for anyone to recognize the signature as authentic, but impossible for anyone other than the legitimate signer to produce it. We will call any such technique one-way authentication. Since any digital signal can be copied precisely, a true digital signature must be recognizable without being known.

– Diffie and Hellman *New Directions in Cryptography* (1976)

A year after in 1977, the first signature algorithm called **RSA** is introduced, along with the RSA asymmetric encryption algorithm (which you learned about in chapter 6). RSA for signing is the first algorithm we'll talk about in this section.

In 1991, the **Digital Signature Algorithm (DSA)** is proposed by the NIST as an attempt to avoid the patents on Schnorr signatures. For this reason DSA is a weird variant of Schnorr signatures, published without a proof of security (although no attacks have been found so far). The algorithm is adopted by many, but quickly replaced with an elliptic curve version called **ECDSA** (for Elliptic Curve DSA), the same way Elliptic Curve Diffie-Hellman (ECDH) replaced Diffie-Hellman (DH) thanks to its smaller keys (see chapter 5). ECDSA is the second signature algorithm I will talk about is in this section.

After the patents on Schnorr signatures expired in 2008 Daniel J. Bernstein, the inventor of Chacha20-Poly1305 (covered in chapter 4) and X25519 (covered in chapter 5), introduces a new signature scheme called **EdDSA** (for Edwards-curve DSA) based on Schnorr signatures. Since its invention, EdDSA has quickly gained adoption and is nowadays considered the state of the art signature for real-world applications. EdDSA is the third and last signature algorithm I will talk about in this section.

7.3.1 RSA Signatures, What Standard To Use? PKCS#1 v1.5 Or RSA-PSS?

RSA signatures are currently used everywhere, even though they shouldn't (as you will see in this section they present many issues). This is due to the algorithm being the first signature scheme to be standardized, as well as real-world applications being really slow to move to newer and better algorithms. Because of this, you will most likely encounter RSA signatures in your journey, and I cannot avoid explaining how they work and which standards are the adopted ones. But let me say that if you understood how RSA encryption works in chapter 6, then this section should be straightforward as signing with RSA is the opposite of encrypting with RSA:

- To **sign**, you encrypt the message with the private key (instead of the public key) which produces a signature (a random element in the group).
- To **verify a signature**, you decrypt the signature with the public key (instead of the private key). If it gives you back the original message, then the signature is valid.

So if your private key is the private exponent d , and your public key is the public exponent and

public modulus (e, N) you:

- sign a message by computing $signature = message^d \bmod N$
- verify a signature by computing $signature^e \bmod N$ and verifying that it is equal to the message.

I illustrated this visually in figure 7.13.

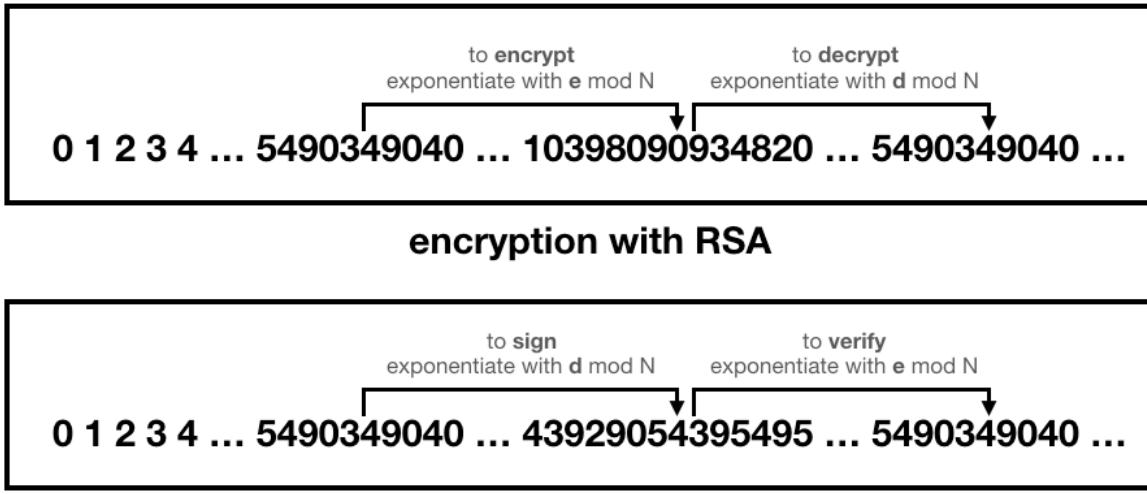


Figure 7.13 To sign with RSA, we simply do the inverse of the RSA encryption algorithm: we exponentiate the message with the private exponent, and to verify we exponentiate the signature with the public exponent (which returns back to the message).

This works because only the one knowing about the private exponent d can produce a signature over a message. And as with RSA encryption, the security relies on the hardness of the factorization problem.

What about the standards to use RSA for signatures? Luckily, they follow the same pattern the standards for RSA encryption did. Remember what you've learned in chapter 6:

- RSA for encryption was loosely standardized in the PKCS#1 v1.5 document.
- RSA was then standardized again in the in PKCS#1 v2 document with a better construction called RSA-OAEP.

The standards for RSA signatures follow the same pattern:

- **RSA PKCS#1 v1.5** standardized a signature scheme without a security proof.
- **RSA-PSS** (for Probabilistic Signature Scheme), introduced in PKCS#1 v2, comes with a security proof.

NOTE

By the way, don't get tricked by the different terms surrounding RSA! Let's recapitulate to avoid confusion. There is **RSA the asymmetric encryption primitive** and **RSA the signature primitive**. On top of that there is also RSA the company (founded by the inventors of RSA)... When mentioning encryption with RSA, most people refer to the schemes RSA PKCS#1 v1.5 and RSA-OAEP. When mentioning signatures with RSA, most people refer to the schemes RSA PKCS#1 v1.5 and RSA-PSS. I know, this can be confusing.

I talked about RSA PKCS#1 v1.5 in chapter 6 on Asymmetric Encryption. The signature scheme standardized in that document is pretty much the same as the encryption scheme. To sign, first hash the message with a hash function of your choice, then pad it according to PKCS#1 v1.5's padding for signature (which is similar to the padding for encryption with the same standard) and then "encrypt" the padded and hashed message with your private exponent. I illustrated this in figure 6.14.

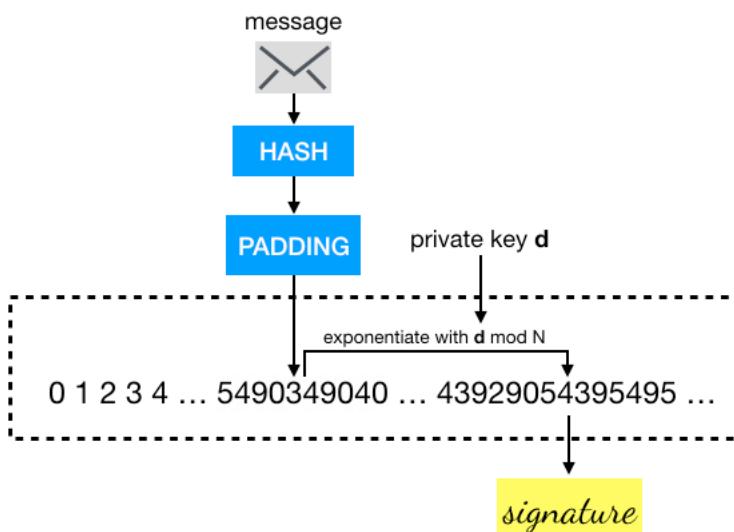


Figure 7.14 RSA PKCS#1 v1.5 for signatures. To sign, hash then pad the message with the PKCS#1 v1.5 padding scheme. The final step exponentiates the padded hashed message with the private key d modulo N . To verify, simply exponentiate the signature with the public exponent e modulo N , and verify that it matches the padded and hashed message.

Unfortunately, attacks on the RSA signature standards also followed the same patterns as for attacks on the standards RSA for encryption. If you remember chapter 6 on Asymmetric Encryption, I said that Bleichenbacher published an attack on RSA PKCS#1 v1.5 for **encryption** in 1998. Bleichenbacher didn't stop there and went on to show a **signature forgery** attack (an attacker can forge a signature without knowledge of the private key) on RSA PKCS#1 v1.5 for

signatures in 2006. Unlike the first attack that broke the encryption algorithm completely, the second attack is an implementation attack. This means that if the signature scheme is implemented correctly (according to the specification), the attack does not work.

Yet, it was shown in 2019 that many open source implementations of RSA PKCS#1 v1.5 for signatures actually fell for that trap and mis-implemented the standard, which enabled different variants of Bleichenbacher's forgery attack to work!⁴⁶

Unfortunately, RSA PKCS#1 v1.5 for signatures is still widely used. So be aware of these issues if you really **have to** use this algorithm for backward compatibility reason.

Having said that, this does not mean that RSA for signatures is insecure. The story does not end here. **RSA-PSS** was standardized in the updated PKCS#1 v2.1 standard, and included a proof of security (unlike the signature scheme standardized in the previous PKCS#1 v1.5). Unfortunately, RSA-PSS has seen little adoption to this day.

The newer specification works like this:

- Encode the message using the PSS encoding algorithm.
- Sign the encoded message using RSA (as was done in the PKCS#1 v1.5 standard).

The PSS encoding is a bit more involved, and similar to OAEP. I illustrate it in figure 7.15.

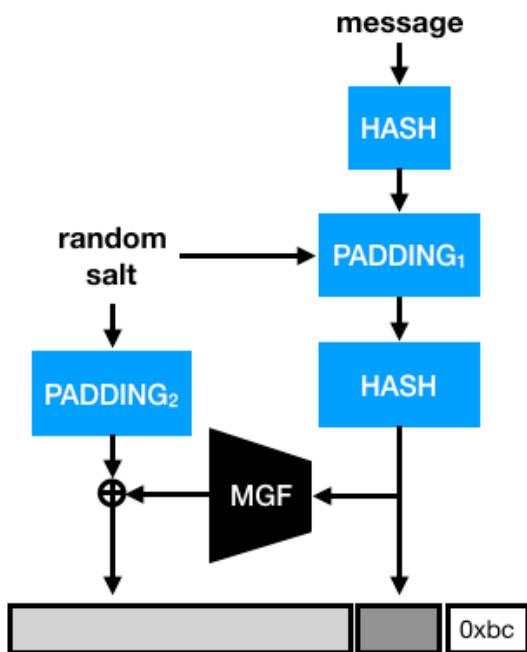


Figure 7.15 The RSA-PSS signature scheme encodes a message (using a Mask Generator Function like the RSA-OAEP algorithm you've learned about in chapter 6) before signing it in the usual RSA way.

Verifying a signature produced by RSA-PSS is just a matter of inverting the encoding once the

signature has been raised to the public exponent modulo the public modulus.

NOTE

PSS is proven secure (meaning that no one should be able to forge a signature without knowledge of the private key) via the usual way of proving security in asymmetric cryptography: instead of proving that if RSA (RSA-PSS without the encoding part) is secure then RSA-PSS is secure, we prove the inverse (which is equivalent): if someone can break RSA-PSS, then that someone can also break RSA. Of course this only works if RSA is indeed secure, which we assume here.

If you remember, I also talked about a third algorithm in chapter 6 for RSA encryption (called RSA-KEM). A simpler algorithm that is not used by anyone and yet is proven to be secure as well. Interestingly RSA for signatures also mirror this part of the RSA encryption history, and has a much simpler algorithm that pretty much nobody uses called **Full Domain Hash (FDH)**. FDH works by simply hashing a message and then signing it (by interpreting the digest as a number) using RSA.

Eventhough both RSA-PSS and FDH come with proofs of security, and are much easier to implement correctly, today most protocols still make use of RSA PKCS#1 v1.5 for signatures. This is just another example of the slowness that typically takes place around deprecating cryptographic algorithms. As older implementations still have to work with newer implementations, it is difficult to remove or replace algorithms. Think of users that do not update applications, vendors that do not provide new versions of their softwares, hardware devices that cannot be updated, and so on.

Next, let's take a look at a more modern algorithm.

7.3.2 The Elliptic Curve Digital Signature Algorithm (ECDSA)

In this section let's look at the Elliptic Curve Digital Signature Algorithm (ECDSA), an elliptic curve variant of DSA which was itself invented only to circumvent patents in Schnorr signatures.

The signature scheme is specified in many standards including ISO 14888-3, ANSI X9.62, NIST's FIPS 186-2, IEEE P1363, and so on. Not all standards are compatible, and applications that want to interoperate have to make sure that they use the same standard.

Unfortunately ECDSA, like DSA, does not come with a proof of security while Schnorr signatures did. Since then, the patents on Schnorr signatures have expired and it is making a coming back especially with cryptocurrencies. Nonetheless, ECDSA has been widely adopted and is one of the most widely used signature scheme. In this section I will explain how it works and how it can be used.

Let's get to the details. As with all such schemes, the public key is pretty much always generated

according to the same formula:

- $\text{private_key} = x$
- $\text{public_key} = [x]G$

Where x is a random number obtained via a random number generator, and G is our group generator (usually called a **based point** in elliptic curve cryptography).

NOTE

Notice that I use the additive notation (with the elliptic curve syntax of placing brackets around the scalar), but that I could have written $\text{public_key} = G^x$ if I had wanted to use the multiplicative notation. These differences do not matter in practice, but most of the time protocols that do not care about the underlying nature of the group are written using the multiplicative notation whereas protocols that are defined specifically in elliptic curve-based groups tend to be written using the additive notation.

To compute an ECDSA signature, you need the same inputs required by a Schnorr signature:

- a hash of the message you're signing $H(m)$
- your private key x
- a random number k unique per-signature.

An ECDSA signature is two integers r and s computed as follows:

- $r = \text{x-coordinate of } [k]G$
- $s = k^{-1} (H(m) + x r) \bmod p$

You can certainly recognize that this is extremely similar to a Schnorr signature.

The random number k is sometimes called a **nonce**, because it is a number that must only be used once, and is sometimes called an **ephemeral key** because it must remain secret.

I'll re-iterate this: **k must never be repeated nor be predictable!**

In general, cryptographic libraries will hide the generation of this nonce, but sometimes they either don't and let the caller provide it. Worse, some libraries do not even generate the nonce correctly. Infamously, Sony's Playstation 3 signature keys were found to repeat nonces when used with ECDSA.⁴⁷ Observing two signatures using the same nonce k turns up being enough to retrieve the ECDSA private key, so this was pretty devastating for Sony.

NOTE

Even more subtle, if the nonce k is not totally picked uniformly and at random (specifically, if you can predict the first few bits), there still exist very powerful attacks that can recover the private key in no time (so-called lattice attacks). In theory, we call these kinds of key retrieval attacks **total breaks** (because they break everything)! Such total breaks are quite rare in practice, which makes ECDSA an algorithm that can fail in spectacular ways.

Attempts at avoiding issues with nonces exist. For example, RFC 6979 specifies a **deterministic ECDSA** scheme which generates a nonce based on the message and the private key.⁴⁸ This means that signing the same message twice involves the same nonce twice, and as such produces the same signature twice (which is obviously not a problem).

To verify an ECDSA signature, a verifier needs to use the same hashed message $H(m)$, the signer's public key and the message signature (r, s) :

- Compute $[H(m) s^{-1}]G + [r s^{-1}]\text{public_key}$.
- The signature is valid if the x-coordinate of the point obtained is the same as the value r of the signature.

The elliptic curves that tend to be used with ECDSA are pretty much the same curves that are popular with the Elliptic Curve Diffie-Hellman algorithm (see chapter 5), with one notable exception: **Secp256k1**.

Secp256k1 is defined in the Standards for Efficient Cryptography Group (SECG) document.⁴⁹ This curve has gained a lot of traction after Bitcoin decided to use it instead of the more popular NIST curves (due to the lack of trust I've talked about in chapter 5).

Secp256k1 is a type of elliptic curve called a Koblitz curve. A Koblitz curve is just an elliptic curve with some constraints in how it can be chosen, and that consequently obtains some extra possible optimizations. The elliptic curve has the following equation:

$$y^2 = x^3 + ax + b$$

With $a=0$ and $b=7$ constants, and with x and y defined over the numbers modulo the prime p :

$$p = 2^{192} - 2^{32} - 2^{12} - 2^8 - 2^7 - 2^6 - 2^3 - 1$$

This defines a group of prime order (like the NIST curves). In other words, the number of points on our elliptic curve is the following prime number:

115792089237316195423570985008687907852837564279074904382605163141518161494337

And we use as generator (or base point) the fixed point G of coordinates:

- x:
55066263022277343669578718895168534326250603453777594175500187360389116729240
- y:
32670510020758816978083085130507043184471273380659243275938904335757337482424

Nonetheless, today ECDSA is mostly used with the NIST curve P-256.

Next let's look at another widely popular algorithm: EdDSA

7.3.3 The Edwards-curve Digital Signature Algorithm (EdDSA)

Let me introduce the last signature algorithm of the chapter, The **Edwards-curve Digital Signature Algorithm (EdDSA)**, published in 2011 by Daniel J. Bernstein in response to the lack of trust in NIST and other curves created by government agencies.

The name EdDSA seems to indicate that it is based on the DSA algorithm, like ECDSA is, but this is deceiving. EdDSA is actually based on Schnorr signatures, which has been possible due to the Schnorr signatures expiring earlier in 2008.

One particularity of EdDSA is that the scheme does not require new randomness for every signing operation. EdDSA produces signatures **deterministically**. This has made the algorithm quite attractive, and it has since been adopted by many protocols and standards.

Currently EdDSA is on track to be included in NIST's FIPS 186-5 document (still a draft at the moment of this writing). The current official standard is RFC 8032,⁵⁰ which defines two curves of different security levels to be used with EdDSA. Both of the curves are twisted Edwards curves, a type of elliptic curve enabling more optimizations:

- **Edwards25519** is based on Daniel J. Bernstein's Curve25519. It is faster than Curve25519 but was invented later, thus not used in the X25519 algorithm you've learned in chapter 5. As with Curve25519, Edwards25519 provides 128-bit of security.
- **Edwards448** is based on Mike Hamburg's Ed448-Goldilocks curve, providing 224-bit of security.

In practice, EdDSA is mostly instantiated with the Edwards25519, and the combo is called **Ed25519** (whereas EdDSA with Edwards448 is shortened as Ed448).

Key generation with EdDSA is a bit different from other existing schemes. Instead of generating a signing key directly, EdDSA generates a secret key that can be used to derive the actual signing key and another key. The other key is important, it is the one used to deterministically generate the required per-signature nonce. I illustrate this in figure 7.16.

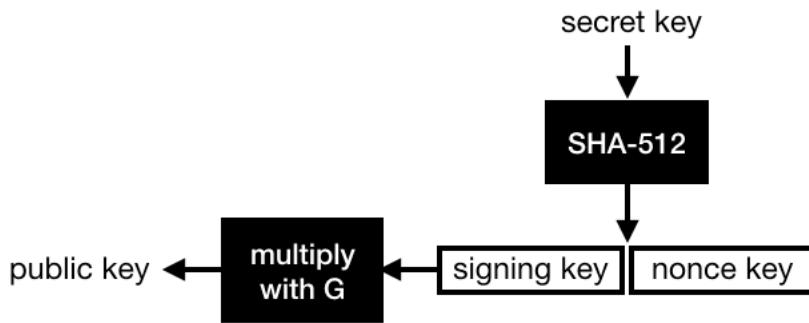


Figure 7.16 EdDSA's secret key is used to derive, with the SHA-512 hash functions, two other keys. The first key is the actual signing key and can thus be used to derive the public key. The other key is used to deterministically derive the nonce during signing operations.

Depending on the cryptographic library you're using, you might be storing the secret key, or the derived private key and "nonce key". Not that this matter, but if you don't know this you might get confused if you run into Ed25519 secret keys being stored as 32 bytes and 64 bytes depending on the implementation used.

To sign, EdDSA first deterministically generates the nonce by hashing the nonce key with the message to sign. Thus, if you attempt to sign the same message twice, the same nonce will be generated.

After that, a process similar to Schnorr signatures happen:

1. compute the nonce as $\text{HASH}(\text{nonce key} // \text{message})$
2. compute the commitment R as $[\text{nonce}]G$ with G the base point of the group.
3. compute the challenge as $\text{HASH}(\text{commitment} // \text{public key} // \text{message})$
4. compute the proof S as $\text{nonce} + \text{challenge} \times \text{signing_key}$
5. the signature is (R, S)

I've illustrated the differences with Schnorr signatures in figure 7.17.

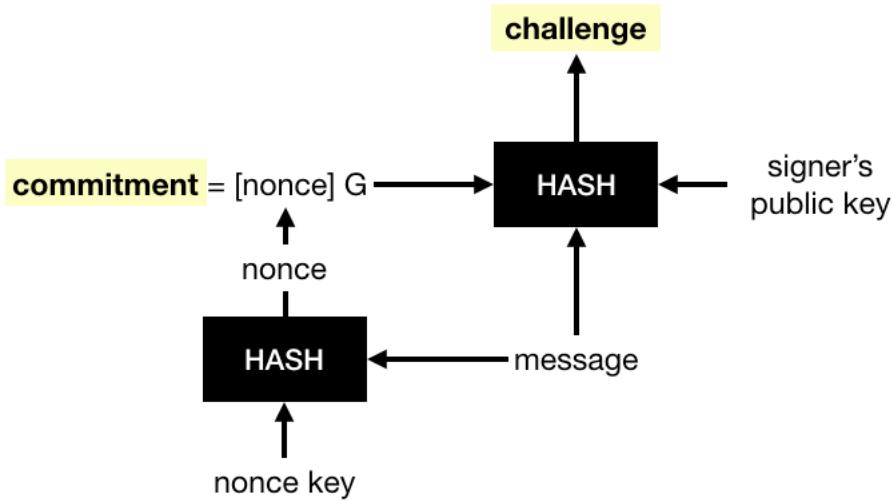


Figure 7.17 EdDSA signatures are computed like Schnorr signatures, except for (1) the nonce that is generated deterministically from a nonce key and the message and (2) the public key of the signer is included as part of the challenge.

Notice how the nonce (or ephemeral key) is derived deterministically, and not probabilistically, from the message and the nonce key (derived from the main secret key). This means that signing two different messages should involve two different nonces, ingeniously preventing the signer from re-using nonce and leaking out the key (as you've seen can happen with ECDSA). This also means that signing the same message twice will produce the same nonce twice, which will produce the same signature twice (which is obviously not a problem).

A signature can be verified by computing the following two equations:

- $[S]G$
- $R + [\text{HASH}(R \parallel \text{public key} \parallel \text{message})] \text{ public key}$

The signature is valid if the two values match.

Indeed, as equation 7.1 shows the two values should be equal if correctly computed.

SIDE BAR **Equation 7.1**

$$\begin{aligned}
 [S]G &= [\text{nonce} + \text{challenge} \times \text{signing_key}]G \\
 &= G + [\text{challenge} \times \text{signing_key}]G \\
 &= [\text{nonce}]G + [\text{challenge}] \text{ public key} \\
 &= R + [\text{challenge}] \text{ public key} \\
 &= R + [\text{HASH}(R \parallel \text{public key} \parallel \text{message})] \text{ public key}
 \end{aligned}$$

The most-used instantiation of EdDSA, **Ed25519**, is defined with the Edwards25519 curve and the SHA-512 as hash function. The Edwards25519 curve is defined with all the points satisfying this equation

$$-x^2 + y^2 = 1 + d \times x^2 \times y^2 \bmod p$$

where

$$\begin{aligned} d &= \\ 37095705934669439343138083508754565189542113879843219016388785533085940283555 \end{aligned}$$

and x, y are taken modulo $p = 2^{255} - 19$

(the same prime used for Curve25519)

The base point is

$$\begin{aligned} G &= \\ (15112221349535400772501151409588531511454012693041857206046113283949847762202, \\ 46316835694926478169428394003475163141307993866256225615783033603165251855960) \end{aligned}$$

RFC 8032 actually defines 3 variants of EdDSA using this curve Edwards25519. All 3 variants can work with the same keys (in other words, only the signing and verification algorithms are different):

- **Ed25519 (or pureEd25519).** That's the algorithm that I've explained above
- **Ed25519ctx.** An algorithm without name introducing a mandatory customization string. This algorithm is rarely implemented, if even used, in practice. The only difference is that some prefix is added to every call to the HASH function.
- **Ed25519ph (or HashEd25519).** This allows applications to pre-hash (hence the "ph" in the name) the message before signing it. It also builds upon Ed25519ctx, allowing the caller to include an optional custom string.

The addition of a **customization string** is something quite common in cryptography, as you have seen with some hash functions in chapter 2, or key derivation functions in chapter 8. It is a useful addition when the same protocol has a signer sign messages, using the same key, in different contexts.

For example, you can imagine an application that would allow you to sign transactions with your private key, but also to sign private messages to people you talk to. If you mistakenly sign a message that looks like a transaction, this could be used against you.

Pre-hashing in Ed25519ph was introduced solely to please callers that need to sign large messages. Indeed, as you've seen in chapter 2, hash functions often provide an "init-update-finalize" interface that allow you to continuously hash a stream of data. This allows you to hash a large input without having to keep the whole input in memory.

You are now done with your tour of the signature schemes used in real-world applications.

To recap:

- **RSA PKCS#1 v1.5 for signatures** is still widely in use, but is hard to implement correctly and many implementations have been found to be broken.
- **RSA-PSS** has a proof of security, is easier to implement, but has seen poor adoption due to newer schemes based on elliptic curves.
- **ECDSA** is the main competition to RSA PKCS#1 v1.5, it is mostly used with NIST's curve P-256, except in the cryptocurrency world where Secp256k1 seems to dominate.
- **Ed25519** is a newer contender based on Schnorr signatures that has received wide adoption as well. It is easier to implement and does not require new randomness for every signing operation. This is the algorithm you should use if you can.

Next, let's look at how you can possibly shoot yourself in the foot when using these signature algorithms.

7.4 Subtle Behaviors in Signatures

The last section of this book goes through the dangers of mis-using signatures. Brace yourself!

On August 11th, 2015, Andrew Ayer sent an email to the IETF mailing list starting with the following words:⁵¹

I recently reviewed draft-barnes-acme-04 and found vulnerabilities in the DNS, DVSNI, and Simple HTTP challenges that would allow an attacker to fraudulently complete these challenges.

The **draft-barnes-acme-04** mentioned by Andrew Ayer is a document specifying **ACME**, one of the protocols behind the **Let's Encrypt** certificate authority.⁵² A **certificate authority** is the thing that your browser trust and that signs the public keys of websites you visit (I talked about this in the introduction of this chapter). It is called a "certificate" authority due to the fact that it does not sign public keys, but **certificates**. A certificate is just a blob of data bundling a website's public key, its domain name, and some other relevant metadata.

The attack was found merely 6 weeks before major browsers were supposed to start trusting Let's Encrypt's public key. The draft has since become RFC 8555: Automatic Certificate Management Environment (ACME),⁵³ mitigating the issues. Since then no cryptographic attacks are known on the protocol.

This section will go over the accident, and explain why it happened, why it was a surprising bug, and what you should watch for when using signatures in cryptography.

7.4.1 How Let's Encrypt Used Signatures

Let's Encrypt is a pretty big deal. Created in 2014, it is a certificate authority ran as a non-profit, and currently providing trust to ~200 million of websites.

The key to Let's Encrypt's success are two folds:

- It is **free**. Before Let's Encrypt most certificate authorities charged fees from webmasters who wanted to obtain certificates.
- It is **automated**. If you follow their standardized protocol, you can request, renew and even revoke certificates via a web interface. Contrast that to other certificate authorities who did most processing manually, and took time to issue certificates.

If a webmaster wants her website `example.com` to provide a secure connection to her users (via HTTPS), she can request a certificate from Let's Encrypt (essentially a signature over its domain name and public key), and after proving that she owns the domain `example.com` and getting her certificate issued, she will be able to use it to negotiate a secure connection with any browser trusting Let's Encrypt.

That's the theory.

In practice the flow goes like this:

1. Alice registers on Let's Encrypt with an RSA public key.
2. Alice asks Let's Encrypt for a certificate for `example.com`.
3. Let's Encrypt asks Alice to prove that she owns `example.com`, for this she has to sign some data and upload it to `example.com/.well-known/acme-challenge/some_file`.
4. Once Alice has signed and uploaded the signature, she asks Let's Encrypt to go check it.
5. Let's Encrypt checks if it can access the file on `example.com`, if it successfully downloaded the signature and the signature is valid then Let's Encrypt issues a certificate to Alice.

I recapitulate some of this flow in figure 7.18.

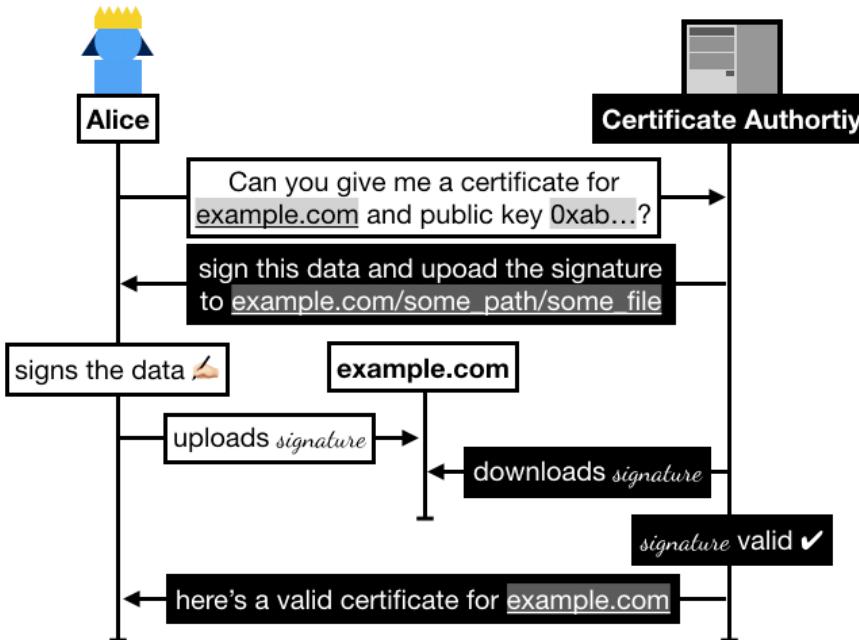


Figure 7.18 In 2015, Alice could request a signed certificate from Let's Encrypt by uploading a signature (from the key she registered with) on her domain. The certificate authority verifies that Alice owns the domain by downloading the signature from the domain and verifying it. If it is valid, the authority signs a certificate (which contains the domain's public key, the domain name example.com, and some other metadata) and sends it to Alice who can then use it to secure her website in a protocol called TLS that you'll learn about in chapter 9.

Let's see next how the attack worked.

7.4.2 How Did The Let's Encrypt Attack Worked

In the attack that Andrew Ayer found in 2015, Andrew proposes a way to gain control of a Let's Encrypt account that has already validated a domain (let's pick example.com as an example)

The attack goes something like this (keep in mind that I'm simplifying):

1. Alice registers and goes through the process of verifying her domain example.com by uploading some signature over some data on example.com/.well-known/acme-challenge/some_file. She then successfully manages to obtain a certificate from Let's Encrypt.
2. Later, Eve signs up to Let's Encrypt with a new account and an RSA public key, and request to recover the example.com domain
3. Let's Encrypt asks Eve to sign some new data, and upload it to example.com/.well-known/acme-challenge/some_file (note that the file is still lingering there from Alice's previous domain validation)
4. Eve crafts a new malicious keypair, and updates her public key on Let's Encrypt. She then asks Let's Encrypt to check the signature
5. Let's Encrypt obtains the signature file from example.com, the signature matches, Eve is granted ownership of the domain example.com. She can then ask Let's Encrypt to issue valid certificates for this domain and any public key.

I recapitulate the attack in figure 7.19:

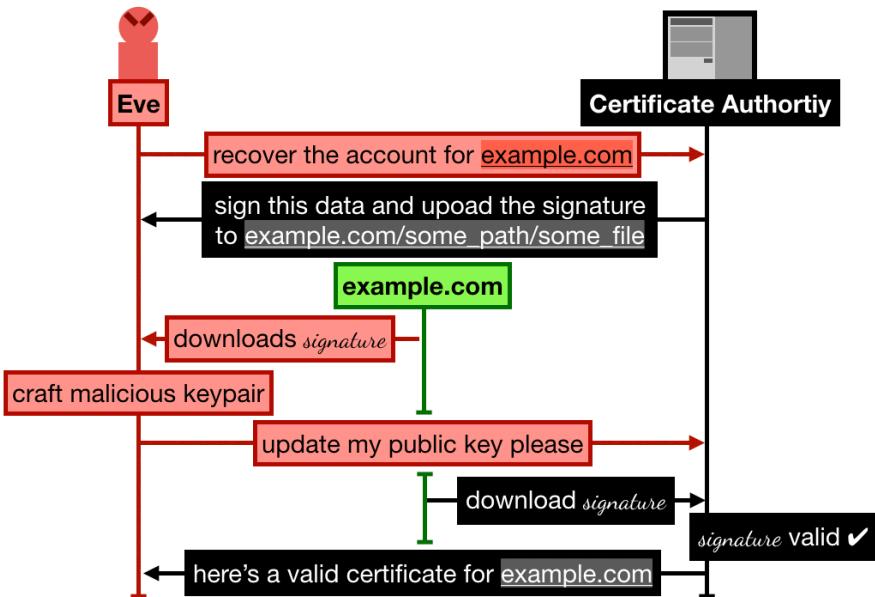


Figure 7.19 The 2015 Let's Encrypt attack allowed an attacker (here Eve) to successfully recover an already approved account on the certificate authority. To do this, she simply forges a new keypair that can validate the already existing signature and data from the previous valid flow.

Take a few minute to understand the attack. It should be quite surprising to you. Next, let's see how Eve could craft a new keypair that worked like the original one did.

7.4.3 Key Substitution Attacks On RSA

In the previously discussed attack, Eve managed to create a valid public key that validates a given signature and message. This is quite a surprising property of RSA, so let's see how this works.

A digital signature does not uniquely identify a key or a message

– Andrew Ayer Duplicate Signature Key Selection Attack in Let's Encrypt

Here is the problem given to the attacker: for a fixed signature and (PKCS#1 v1.5 padded) message, a public key (e , N) must satisfy the following equation to validate the signature:

$$\text{signature} = \text{message}^e \bmod N$$

One can easily craft a key pair that will (most of the time) satisfy the equation:

- a public exponent $e = 1$
- a private exponent $d = 1$
- a public modulus $N = \text{signature} - \text{message}$

You can easily verify that the validation works with this keypair:

SIDE BAR **Equation 7.2**

$$\text{signature} = \text{message}^e \bmod N$$

$$\text{signature} = \text{message} \bmod \text{signature message}$$

$$\text{signature message} = 0 \bmod \text{signature message}$$

Is this issue surprising?

It should be.

This property called "key substitution" comes from the fact that there exist a gap between the theoretical cryptography world and the applied cryptography world, between the security proofs and the implemented protocols.

Signatures in cryptography are usually analyzed with the **EUF-CMA model**, which stands for **Existential Unforgeability under Adaptive Chosen Message Attack**.

In this model YOU generate a key pair, and then I request YOU to sign a number of arbitrary messages. While I observe the signatures you produce, I win if I can at some point in time produce a valid signature over a message I hadn't requested.

Unfortunately, even though our modern signature schemes seem to pass the EUF-CMA test fine, they tend to exhibit some **surprising properties** like the key substitution one.

In the next section, let's look at the key substitution property, and some other ones that can surprise you in the wrong way.

7.4.4 Subtle Behaviors of Signature Schemes

There are a number of subtle properties that signature schemes might exhibit. While they might not matter in most protocols, not being aware of these gotchas can end up biting you when working on more complex and non-conventional protocols.

This section will go over a number of known issues with digital signatures.

Substitution attacks. Also referred to as **Duplicate Signature Key Selection (DSKS)**,⁵⁴ these attacks are possible on both RSA PKCS#1 v1.5 and RSA-PSS. There are two variants that exists:

1. **key substitution** attacks, where a different keypair or public key is used to validate a given signature over a given message.
2. **message key substitution** attacks, where a different keypair or public key is used to validate given signature over a new message.

One more time: the first attack fixes both the message and the signature, the second one only fixes the signature. I recapitulate this in figure 7.20.

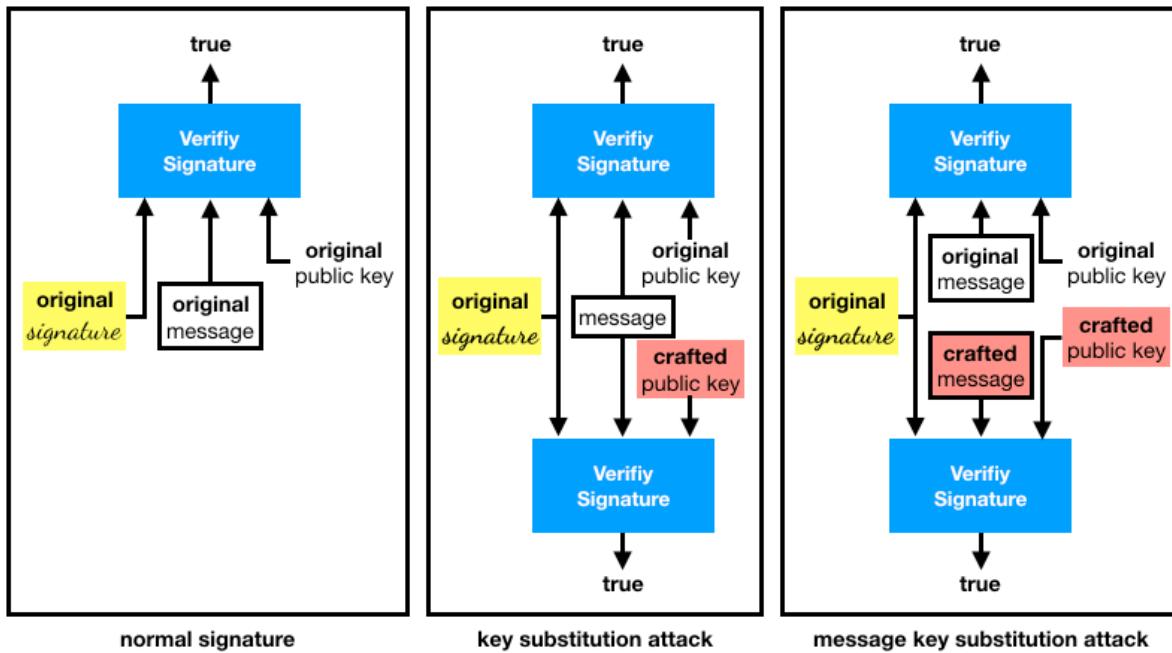


Figure 7.20 Signature algorithms like RSA are vulnerable to key substitution attacks, which are a surprising and unexpected behavior for most users of cryptography. A key substitution attack allows one to take a signature over a message, and to craft a new key pair that will validate the original signature. A variant called message key substitution allows an attacker to create a new key pair and a new message that validates under the original signature.

Malleability. Most signature schemes are malleable, meaning that if you give me a valid signature I can tamper with it so that it becomes a different but still valid signature. Note that if I'm the signer I can usually create different signatures for the same message, but here malleability refers to the fact that someone who has zero knowledge of the private key can also create a new valid signature for the same signed message. It is not clear if this has any impact on any real-world protocol, even though the bitcoin MtGox exchange blamed their loss of funds on this one. From the paper Bitcoin Transaction Malleability and MtGox:⁵⁵

In February 2014 MtGox, once the largest Bitcoin exchange, closed and filed for bankruptcy claiming that attackers used malleability attacks to drain its accounts.

Note that a newer security model called SUF-CMA (for strong EUF-CMA) attempts to include non-malleability (or resistance to malleability) in the security definition of signature schemes. Some recent standards (like RFC 8032 that specifies Ed25519)⁵⁶ update existing algorithms to mitigate against malleability attacks. Since these mitigations are not always present, nor common, you should not rely on signatures being non-malleable.

Re-signability. This one is simple to explain. To validate a signature over a message you often

don't need the message itself but its digest. This would allow anyone to re-sign the message with their own keys without knowing the message itself. How is this impactful in real-world protocols? Not sure, but we never know.

Collidability. This is another not-so-clear if it'll bite you one day: some schemes allow you to craft signatures that will validate under several messages (not just one). For example, Ed25519 as designed allows a signer to craft a public key and a signature that would validate any messages with high probability.

What to do with all of this information?

Well, for one signature schemes are definitely not broken, and you probably shouldn't worry if your use of them are not too out-of-the-box.

But if you're designing cryptographic protocols, or if you're implementing something that's more complicated than the every day use of cryptography, you might want to keep these subtle properties in the back of your mind.

7.5 Summary

- Digital signatures are similar to pen-and-paper signatures but are backed with cryptography, making them unforgeable by anyone who does not control the signing (private) key.
- Digital signatures can be useful to authenticate origins (for example the side of a key exchange) as well as providing transitive trust (if I trust Alice and she trust Bob, I can trust Bob).
- Zero-knowledge proofs (ZKPs) allow a prover to prove the knowledge of something (called a witness), without revealing the something. Signatures can be seen as non-interactive zero-knowledge proofs as they do not require the verifier to be online during the signing operation.
- As for asymmetric encryption, there exist many standards that you can use to sign:
 - RSA PKCS#1 v1.5 for signing is widely used today, but not recommended as it is hard to implement correctly.
 - RSA-PSS is a better algorithm, as it is easier to implement and has a proof of security. Unfortunately it is not very popular nowadays due to elliptic curve variants that support shorter keys.
 - The most popular signature schemes right now are based on elliptic curves: ECDSA and EdDSA. ECDSA is often used with NIST curve P-256 and EdDSA is often used with the Edwards25519 curve and the combination is referred to as Ed25519.
- Some subtle properties can be dangerous if signatures are used in a non-conventional way:
 - always avoid ambiguity as to who signed a message as some signature schemes are vulnerable to key substitution attacks: external actors can create a new keypair that would validate an already existing signature over a message, or create a new keypair and a new message that would validate a given signature.
 - do not rely on uniqueness of signatures. First, in most signature schemes the signer can create an arbitrary amount of signatures for the same message. Second, most signature schemes are "malleable", meaning that external actors can take a signature and create another valid signature for the same message.

Randomness and secrets



This chapter covers

- What randomness is and why it's important.
- Obtaining strong randomness and producing secrets for cryptography.
- What the pitfalls of randomness are.

This is the last chapter of the first part of this book, and I have one last thing to tell you before we move on to the second part of this book and learn about actual protocols used in the real-world. It is something I've grossly overlooked at so far: **randomness**.

You must have noticed that in every cryptographic algorithm you've learned, with the exception of hash functions, you had to use randomness at some point. Secret keys, nonces, IVs, prime numbers, and so on. As I was going through these different concepts, randomness always came from some magic black box as illustrated in figure 8.1.

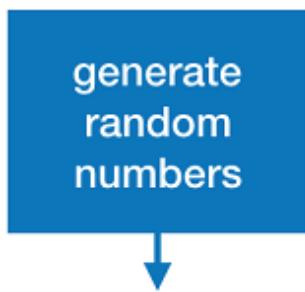


Figure 8.1 Randomness in practice is often ignored and abstracted by cryptography as a black box that magically provides random numbers for us. This chapter will teach you where this randomness comes from in practice and how you can obtain it.

In this chapter, I will provide you with explanations as to what cryptography means when it mentions randomness, and what are the practical ways that exist to obtain randomness for

real-world cryptographic applications.

For this chapter you'll need to have read:

- Chapter 2 on hash functions.
- Chapter 3 on message authentication codes.

8.1 What is Randomness?

Everyone understands the concept of randomness to some degree. Whether playing with dice or buying some lottery tickets, we've all been exposed to it. My first encounter with randomness was at a very young age, when I realized that a "RAND" button on my calculator would produce a different number every time I would press it. This troubled me to no end. I had very little knowledge about electronics, but I thought I could understand some of its limitations. When I added 4 and 5 together, surely some circuits would do the math and give me the result. But a random button? Where were the random numbers coming from? I couldn't wrap my head around it. It took me some time to ask the right questions and understand that calculators actually cheated! They would hardcode large lists of "random" numbers and go through them one by one. These lists would exhibit good "randomness" property meaning that if you counted, you would realize that they would contain as many 1s as 9s and so on, what we call **a uniform distribution**.

Not only that, but pressing the "RAND" button X times would also often return as many 1s and 9s and 8s and so on... The odds had been respected, **every number had equal chance of being selected**. What the calculator was doing, was simulating picking numbers **uniformly at random** from a set (the set of the numbers 0 to 9).

Yet, the sequence was known in advance and likely to be identical between calculators, and so this was probably not secure.

When random numbers are needed for security and cryptography purposes, then randomness must be **unpredictable**. For this, we **extract** randomness from observing **hard to predict physical phenomena**. For example, lava lamps have shape-changing blobs of something that are extremely hard to predict. Knowing the state of a lava lamp at some point in time (in addition to the temperature of the room, the amount of vibration induced by trucks passing near the building, etc.) usually does not help you much figuring out the future state of the lamp (see the LavaRand random number generator in figure 8.2). This concept is often dubbed the "butterfly effect" in the popular culture, or more scientifically "sensitive dependence on initial conditions". It is also the reason why things like the weather are extremely hard to predict accurately passed a certain number of days.

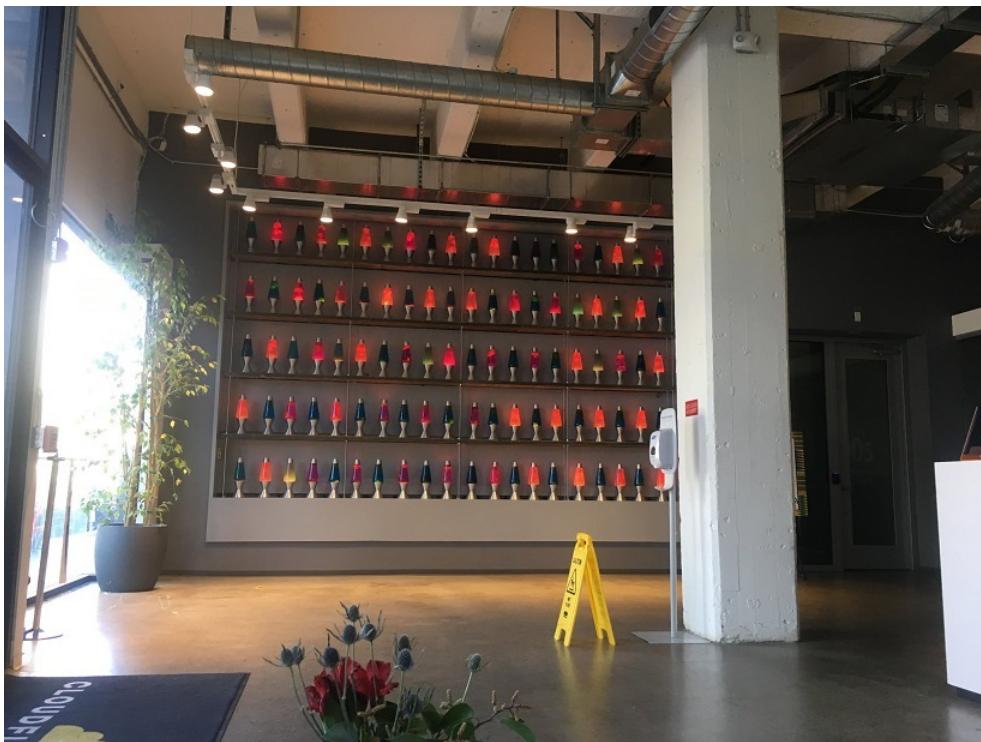


Figure 8.2 A picture I snapped during one of my visit to the headquarters of Cloudflare in San Francisco. LavaRand is a wall of lava lamp, which are lamps that produce a hard to predict flow of "lava". A camera is set in front of the wall to extract and convert this noise to random bytes.

To obtain usable randomness your laptop, cellphone, and most other normal devices will of course not do what LavaRand does. Instead, they observe **noise** that are available to them, and extract randomness from it. (Since cryptography is omnipresent in the operating systems (OS's) and applications running on these devices, most OS's provide handy interfaces to obtain random numbers.) OS's will gather randomness using different tricks. Common **sources of randomness** (or noise) can be the timing of hardware interrupts (e.g. your mouse movements), software interrupts, hard disk seek time, and so on.

NOTE

These sources of randomness are also sometimes called **entropy sources**. In information theory the word **entropy** is used to judge how much randomness a string contains. The term was coined by **Claude Shannon** who devised an entropy formula that would output larger and larger numbers as a string would exhibit more and more unpredictability (starting at 0 for completely predictable). The formula or the number itself is usually not that interesting for us, but in cryptography you will often hear "this string has low entropy" (meaning that it is predictable) or "this string has high entropy" (meaning that it is less predictable). So be aware of this.

Some device also have access to additional sensors and hardware aides that can provide more sources of entropy. For examples, most recent CPUs have hardware random number generators often called **True Random Number Generators (TRNGs)** which make use of external

unpredictable physical phenomena like thermal noise to extract randomness.

Note that noise obtained via all these different types of input is usually not "clean" (for example the first bit obtained from some entropy source could be 0 more often than not) and can sometimes not provide enough entropy (or even none!). For this reason, randomness extractors must de-skew (applying a hash function to the noise for example) and gather several sources of noise together before it can be used for cryptographic applications.

Is this all there is to randomness?

Unfortunately not. Extracting randomness from noise is a process that is not very fast, and in some cryptographic applications (that might need lots of random numbers very quickly) it can become the bottleneck.

For this reason, the next section will go over how real-world applications and OS's boost the generation of random numbers: they use **pseudo-random** numbers instead.

8.2 What is a Pseudo-Random Number Generator (PRNG)?

Randomness is used everywhere. At this point you should be convinced that this is true at least for cryptography, but surprisingly cryptography is not the only place making heavy use of random numbers. For example, simple unix programs like `ls` require randomness too! As a bug in a program can be devastating if exploited, binaries attempt to defend against low-level attacks using a multitude of tricks, one of them is ASLR which randomizes the memory layout of a process every time it is ran (and thus requires random numbers). Another example is TCP, which makes use of random numbers every time it creates a connection to produce unpredictable sequence numbers and thwart attacks attempting to hijack TCP connections. While all of this is out-of-scope for this book, it is good to have an idea of how much randomness is useful in security in general.

Unfortunately, I hinted in the last section that obtaining unpredictable randomness is somewhat of a slow process (this is sometimes due to a source of entropy being slow to produce noise). For this reason, OS's often implement an additional step to generate random numbers faster via a cryptographic construction called a **pseudo-random number generator (PRNG)**.

NOTE

PRNGs are sometimes shortened as PRGs, or sometimes called CSPRNG for "cryptographically secure" PRNGs in order to contrast with PRNGs not designed to be secure (useful in video games for example). The NIST wanting to do things differently than everybody else (as usual) often call theirs DRBGs for Deterministic Random Bit Generators. Just be aware of these differences in names as you might encounter them in different literatures.

A PRNG needs an initial secret usually called a **seed** (which we can obtain from mixing different entropy sources together) and can then produce lots of random numbers very quickly. I illustrated a PRNG in figure 8.3.

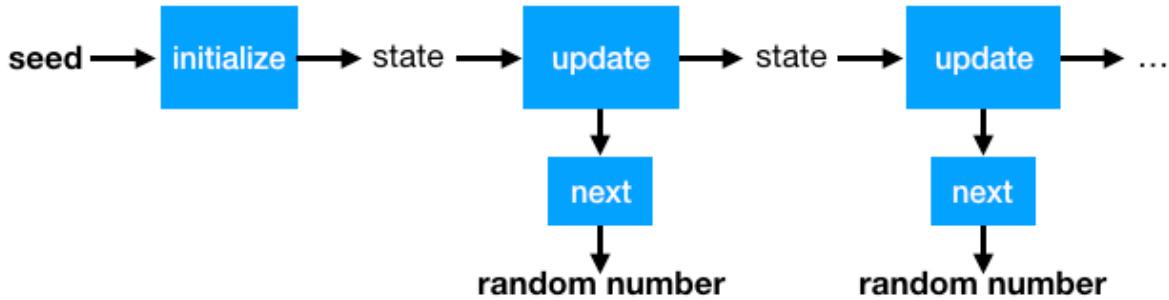


Figure 8.3 A Pseudo-Random Number Generator (PRNG) generates a sequence of random numbers based on a seed. Using the same seed makes the PRNG produce the same sequence of random numbers. It should be impossible to recover the state using knowledge of the random outputs. It follows that it should be impossible, from observing the produced random numbers alone, to predict future random numbers or to recover previously generated random numbers.

A cryptographically secure PRNG exhibits some properties:

- **deterministic.** Using the same seed twice will produce the same sequence of random numbers. This is unlike the unpredictable randomness extraction I talked about previously: if you know a seed used by a PRNG, it should be completely predictable. This is why the construction is called **pseudo-random**, and this is what allows a PRNG to be extremely fast.
- **indistinguishability from randomness.** Theoretically, this means that you should not be able to distinguish between a PRNG outputting a random number from a set of possible numbers, and a fairy impartially choosing a random number from the same set (assuming the fairy knows a magical way to pick a number such that every possible number can be picked with equal probability). Consequently, observing the random numbers generated alone shouldn't allow anyone to recover the internal state of the PRNG (the function `next` in the diagram above is **one-way**).

The last point is very important, as a PRNG simulates picking a number **uniformly at random**, meaning that each number from the set has an equal chance of being picked. For example, if your PRNG produces random numbers of 8 bytes, the set is all the possible strings of 8 bytes, and each 8-byte value should have equal probability of being the next value that can be obtained from your PRNG.

In addition, many PRNGs exhibit additional security properties.

A PRNG has **forward secrecy** if the `update` function is one-way, meaning that if an attacker learns the state (by getting in your computer at some point in time for example) the attacker shouldn't then be able to go back and retrieve previous states and all the random numbers they generated. I illustrated this in figure 8.4.

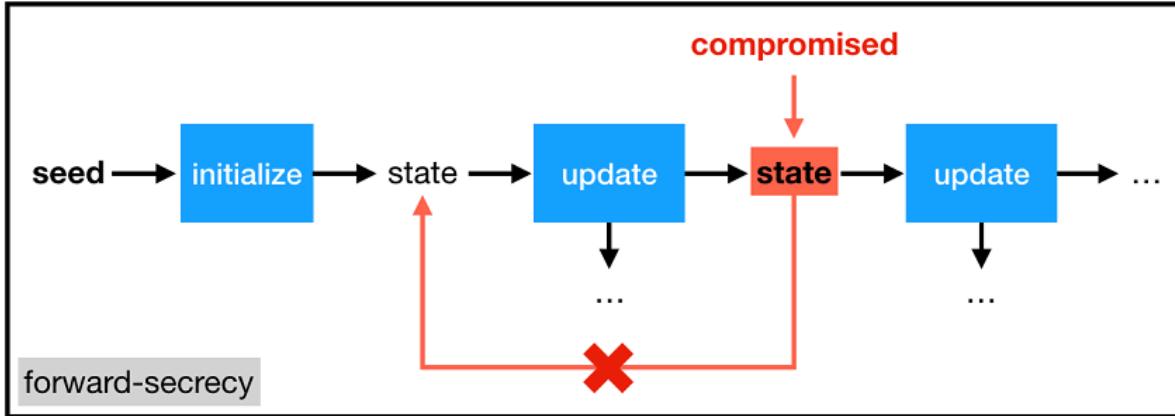


Figure 8.4 A Pseudo-Random Number Generator (PRNG) has **forward secrecy** if compromise of a state does not allow recovering previously generated random numbers.

NOTE Forward secrecy is also sometimes called *perfect forward secrecy*, which over the years has been used interchangeably with forward secrecy to mean pretty much the same thing.

In general, obtaining the state of a PRNG means that you can determine all future pseudo-random numbers it will generate. To prevent this, many PRNGs have mechanisms to "heal" themselves if compromised. We say that a PRNG has **backward secrecy**. This can be achieved by re-injecting (**re-seeding**) new entropy after a PRNG was already seeded, for example by having the `update` function accept an additional `seed` parameter as illustrated in figure 8.5.

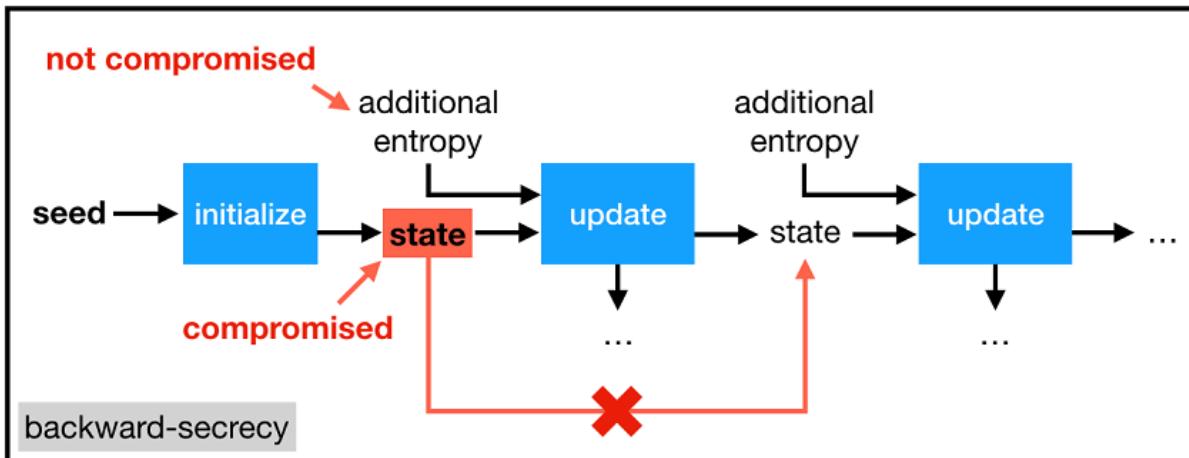


Figure 8.5 A Pseudo-Random Number Generator (PRNG) has **backward secrecy** if compromise of a state does not allow predicting future random numbers that will be generated by the PRNG. This is of course only true once new entropy has been produced and injected in the update function after the compromise.

NOTE

The terms forward and backward secrecy are often source of confusion. If you read this section thinking "shouldn't forward secrecy be backward secrecy, and backward secrecy be forward secrecy instead?" then you are not crazy. For this reason, backward secrecy is sometimes called future-secrecy, or even post-compromise security.

PRNGs can be extremely fast, and are considered secure ways to generate large (sometimes infinite) numbers of random values for cryptographic purposes (if properly seeded with an unpredictable random number). This effectively means that we have secure cryptographic ways to stretching a secret (of appropriate size) to billion of other secret keys.

Pretty cool right?

This is why most (if not all) cryptographic applications do not use random numbers directly extracted from noise, but instead use them to seed a PRNG in an initial step, and then switch to generating random numbers from the PRNG when needed.

In practice, it is quite hard to understand how "random" an entropy source is. And since most PRNGs are not based on mathematical properties (due to speed limitations of math-based constructions), it is also hard to evaluate how good a PRNG is at simulating picking a number uniformly at random. There exist some statistical tests (some example are given in figure 8.6), but no amount of checks will create 100% confidence in the quality of a source or the solidity of a PRNG's design. Similarly to other cryptographic primitives not based on mathematical problems, confidence in a PRNG is obtained by having experts analyzing and attacking a construction for years before using it in any real-world applications.

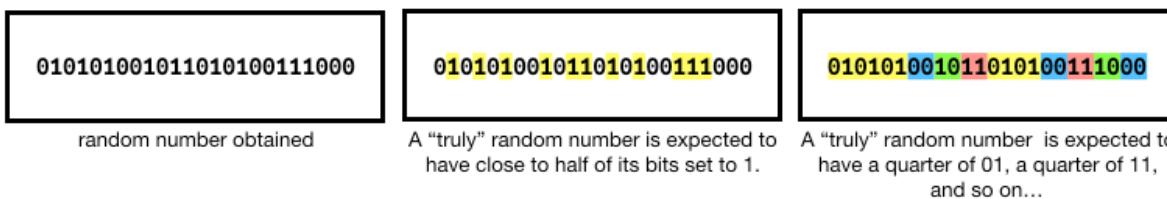


Figure 8.6 Two simple examples of statistical tests. One can expect a good algorithm to produce random numbers that contain a balanced amount of bits set to 1s and 0s. Looking at pairs, a good random number is expected to contain a balanced amount of 01, 00, 10, and 11. If a statistical test notices that on average a PRNG produces random numbers that do not follow this expected "balanced" distribution of bits, then the PRNG has failed the test.

To be secure, a PRNG must be seeded with an unpredictable secret. More accurately, we say that the PRNG takes a key of n-byte sampled **uniformly at random**, meaning that we should pick the key randomly from the set of all possible n-byte strings where each bytestring has the same chance of being picked.

If you remember what you've read in this book, you know that I've talked about many cryptographic algorithms that produce outputs indistinguishable from outputs chosen uniformly at random. Intuitively, you should be thinking "can we use these algorithms to generate random numbers then?", and you would be right! Hash functions, XOFs, block ciphers, stream ciphers, and MACs can be used to produce random numbers.⁵⁷ (MACs and signatures do not "theoretically" provide outputs indistinguishable from random, but instead other properties like unforgeability, see chapters 3 and 7.)

Actually, since AES is hardware supported on most machines, it is customary to see AES-CTR being used to produce random numbers (see figure 8.7).

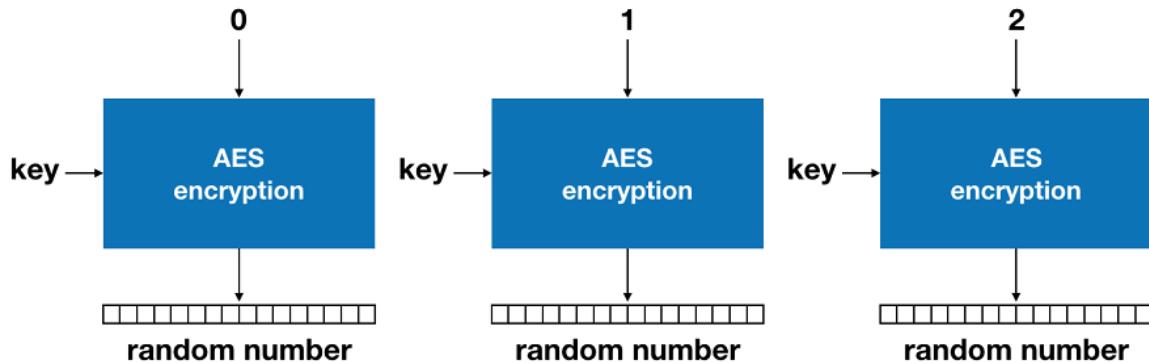


Figure 8.7 AES-CTR can be used as a PRNG where a seed is used as a key to the cipher. One can use the ciphertexts as random numbers, due to the block cipher's output being indistinguishable from random. This construction is an insufficient and naive approach, as it does not have forward secrecy or backward secrecy.

The process to create random numbers by using a hash function is also not too hard to picture. So I illustrate this as well in figure 8.8.



Figure 8.8 The SHA-256 hash function can be used to generate random numbers. This is possible because a hash function is well-accepted to behave like a random oracle: its output is indistinguishable from random. Like the CTR construction you've seen previously, this naive construction does not provide forward or backward secrecy.

Of course, you can see that both of these PRNG constructions do not provide any forward or backward secrecy. In practice, there is a bit more complexity added to these constructions in order to provide either or both of the security properties.

Finally, let me mention that there do exist math-based PRNGs, but they are often too slow to be practical. One notorious example is **Dual EC**, which was invented by the NSA and relied on elliptic curves. The PRNG was pushed to various standards (including NIST ones) around 2006, and not too long after several researchers independently discovered a potential backdoor in the algorithm. This was later confirmed by the Snowden revelations in 2013, and a year later the algorithm was withdrawn from multiple standards.

You now understand enough to go to the next section, which will provide an overview of obtaining randomness for real.

8.3 Obtaining Randomness in Practice

You've learned about the three ingredients that an OS needs to provide cryptographically secure random numbers to its programs:

1. **Noise Sources.** Ways for the OS to obtain raw randomness from unpredictable physical phenomena like the temperature of the device or your mouse movements.
2. **Cleaning and Mixing.** Since raw randomness can be of poor quality (some bits might be biased), OS's clean up and mix a number of sources together in order to produce a good random number.
3. **Pseudo-Random Number Generators.** Since the first two steps are slow, a single uniformly random value can be used to seed a PRNG which will produce a (practically) infinite number of random numbers very quickly.

In this section I will explain how systems bundle these three concepts together to provide simplified interfaces to developers. These functions exposed by OS's usually allow you to generate a random number in a straightforward function call like the following:

```
secret_key = create_random_number(number_of_bytes=16)
```

Behind such a call, is indeed a system bundling up noise sources, a mixing algorithm and a PRNG (summarized in figure 8.9).

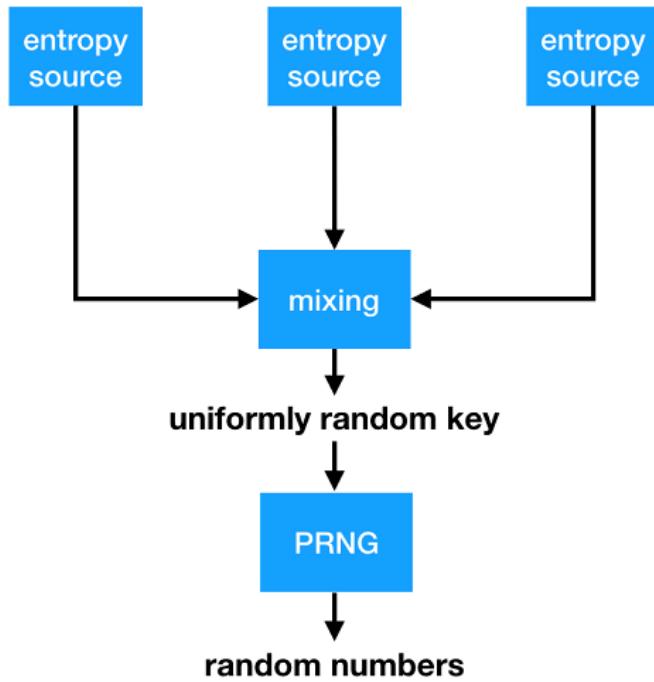


Figure 8.9 Generating random numbers on a system usually means that entropy was mixed together from different noise sources and used to seed a long-term PRNG.

Depending on the OS, and on the hardware available, these three concepts might be implemented differently. For example, at the moment of this writing Linux uses a PRNG based on the Chacha20 stream cipher while macOS uses a PRNG based on the SHA-1 hash function. In addition, the random number generator interface exposed to developers will be different depending on the OS:

- On **Windows**, The `BCryptGenRandom` system call can be used to produce random numbers.
- On **other platforms**, special files usually called `/dev/random` and `/dev/urandom` are exposed and can be read to provide randomness. For example, on MacOS, one can read 16 bytes from `/dev/urandom` directly from the terminal using the `dd` command line tool:

```
$ cat /dev/urandom | xxd -p # xxd display the result in hexadecimal
8856a8744c697411f8b2c77b853add9d77a4dd4cc46277a0be83fb9e10a6
b34e5eb2de4e66b7b930842d293d251d3b5465991df68f0b73174abddcd8
2f60ce2fe7f73f9735e2dbba29cdff37b90495f3b25bdf27c46e879a9167
d317229f35335d08cd4854b500b8d16ba78a57f17b4b5832da74ab895bab
# ...
```

The distinction between `/dev/random` and `/dev/urandom` is that:

- `/dev/random` **blocks** if it does not have enough entropy to properly seed (or re-seed) its internal PRNG.
- `/dev/urandom` **never blocks**.

NOTE

A function that **blocks** is a programming concept you might already be familiar with. Briefly, if you call a function that blocks it means that your program will literally wait for the function to respond. On the other hand, a function that does not block can for example return a value indicating that you need to call it again.

In practice, cryptographic applications should use `/dev/urandom`, as a well-seeded PRNG should be able to produce an infinite number of random values without having to be re-seeded. That being said, `/dev/urandom` still needs to be seeded with proper entropy, as not enough entropy could mean that attackers could find out the seed by trying multiple values until one allows the PRNG to generate the same random numbers. On **Linux** and **FreeBSD**, a `getrandom` system call was released in 2014 with the goal of providing an answer to `/dev/urandom` sometimes not being properly seeded, and to `/dev/random` blocking unnecessarily once properly seeded. (Another advantage is that the `getrandom` system call does not rely on having permissions (or an available file descriptor) to read the `/dev/(u)random` special files.) `getrandom` did this by mixing the behaviors of the two interfaces: like `/dev/urandom`, it never blocks once seeded, but in the rare case where you would ask for random numbers too early, it will block until properly seeded.

The following example shows how one can securely use `getrandom` in C:

Listing 8.1 random_numbers.c

```
#include <sys/random.h>

uint8_t secret[16];
int len_read = getrandom(secret, sizeof(secret), 0); ①

if (len_read != sizeof(secret)) {
    abort(); ②
}
```

- ① `getrandom` fills a buffer with random bytes (up to 256 bytes per call). The last argument of `getrandom` is a flag, if set to 0 it defaults to not blocking unless it has not been properly seeded yet.
- ② It is possible that the function fails or return less than the desired amount of random bytes. If this is the case, it is possible that the system is corrupt and aborting might be the only good thing to do.

With that example in mind, it is also good to point out that many programming languages have standard libraries and cryptographic libraries that provide better abstractions. It might be easy to forget that `getrandom` will only return up to 256 bytes per call for example. For this reason, you should always attempt to use cryptography-related functions (like `getrandom`) through the standard library of the programming language you're using, or through a well-respected cryptographic library.

WARNING Note that many programming languages expose functions and libraries that can produce random numbers which are predictable. These are not suited for cryptographic use! Make sure that you're using random libraries that are generating cryptographically strong random numbers. Usually the name of the library helps (for example you can probably guess which you should use between the `math/rand` and `crypto/rand` package in Golang), but nothing replaces reading the manual!

The following example shows how to generate some random bytes using PHP 7. These random bytes can be used by any cryptographic algorithm, for example as a secret key to encrypt with an authenticated-encryption algorithm. Every programming language does things differently, so make sure to consult your programming language's documentation in order to find out the standard way to obtain random numbers for cryptography purposes.

Listing 8.2 random_numbers.php

```
<?php
$bad_random_number = rand(0, 10));    ①
$secret_key = random_bytes(16);      ②
?>
```

- ① This will produce a random number between 0 and 10. While fast, `rand` does not produce cryptographically secure random numbers! It is thus not suitable for cryptographic algorithms and protocols.
- ② `random_bytes` creates and fills a buffer with 16 random bytes. The result is suitable to be used for cryptographic algorithms and protocols.

Now that you've learned how you can obtain cryptographically secure randomness in your programs, let's think about the security considerations you need to keep in mind when you generate randomness.

8.4 Randomness Generation and Security Considerations

It is good to remember at this point that any useful protocol based on cryptography requires good randomness, and that a broken PRNG could lead to the entire cryptographic protocol or algorithm to be broken. It should be clear to you that a MAC is only as secure as the key used with it, or that the slightest ounce of predictability usually destroys signature schemes like ECDSA, and so on...

So far, this chapter makes it sound like generating randomness should be a simple part of overall applied cryptography, but in practice it is not. Randomness has actually been the source of many many bugs in real-world cryptography, due to a multitude of issues. Using a non-cryptographic PRNG is one common cause of vulnerabilities, badly seeding a PRNG is another (for example

with the current time, which is generally predictable). Another example are programs using custom-PRNGs (called **userland PRNGs** as opposed to the kernel PRNGs which are behind system calls). Userland PRNGs usually add friction to a place where they are not needed, and can in the worst of cases cause devastating bugs if misused. This was notably the case with the PRNG offered by the OpenSSL library that was patched in some OS's in 2006, inadvertently causing all SSL and SSH keys generated using the vulnerable PRNG to be affected.

Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.

– H D Moore *Debian OpenSSL Predictable PRNG Toys*

For this reason, and others I will mention later in this chapter, it is wise to avoid userland PRNG and to stick to randomness provided by the OS when available. In most situations, sticking to what the programming language's standard library, or what a good cryptography library provides, should be enough.

We cannot keep on adding 'best practice' after 'best practice' to what developers need to keep in the back of their heads when writing everyday code.

– Martin Boßlet *The Android SecureRandom incident*

Unfortunately, no list of advice can really prepare you to the many pitfalls of getting randomness. Since randomness is at the center of every cryptography algorithms, making tiny mistakes can lead to devastating outcomes. It is good to keep in mind the following edge cases in case you run into them.

Forks. When using a userland PRNG (some applications with extremely high performance requirements might have no other choice), it is important to keep in mind that a program which forks will produce a new child process that will have the same PRNG state as its parent. Consequently, both PRNGs will produce the same sequence of random numbers from there on. For this reason, if you really want to use a userland PRNG you have to be careful to make forks use different seeds for their PRNGs.

Virtual machines (VMs). This cloning of PRNG state can also become a problem when using the OS PRNG! Think about VMs. If the entire state of a VM is saved and then started several times from this point on, every instance might produce the exact same sequence of random numbers. This is sometimes fixed by hypervisors and OS's, but it is good to look into what the hypervisor you're using is doing before running applications that request random numbers in VMs.

Highly adversarial environments. Due to advances in technology, some cryptographic applications are sometimes in the hand of the adversary. For example, smart cards (think credit

cards) sometimes need to generate random numbers for cryptographic operations (like signatures) in an environment where the adversary might be in control of the available sources of entropy (perhaps being able to always make them produce 0s for example). A common solution to this **entropy starvation** issue is the use of hardware random number generators (so-called TRNGs).

Early boot entropy. While OS's should have no trouble gathering entropy in user-operated devices due to the noise produced by their interactions with the device, embedded devices and headless systems have more challenges to overcome in order to produce good entropy at boot time. History has shown that some devices will tend to boot in a similar fashion and end up amassing the same initial "noise" from the system, leading to the same seed being used for their internal PRNG and the same series of random numbers being generated.

There is a window of vulnerability—a boot-time entropy hole—during which Linux's urandom may be entirely predictable, at least for single-core systems. [...] When we disabled entropy sources that might be unavailable on a headless or embedded device, the Linux RNG produced the same predictable stream on every boot.

— Heninger et al. *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices*

In these rare cases where you really (really) need to obtain random numbers early during boot, one can help the system by providing some initial entropy generated from another machine's well-seeded `/dev/urandom`. Different OS's might provide this feature and you should consult the manual (as usual) if you find yourself in this situation. If available, a TRNG provides an easy solution to the problem. For example modern intel CPUs embed a special hardware chip that extracts randomness from thermal noise. This randomness is available through an instruction called `RDRAND`.

NOTE

Interestingly, Intels' `RDRAND` has been quite controversial due to fear of backdoors. Most OS's that have integrated `RDRAND` as a source of entropy today, mix it with other sources of entropy in a way that is **contributory**. Contributory here means that one source of entropy cannot force the outcome of the randomness generation. Imagine for a minute that mixing different sources of entropy was done by simply XORing them together, can you see how this might fail to be contributory?

Finally, let me mention that one solution of avoiding the randomness pitfalls is to use algorithms that **rely less on randomness**.

For example, you've seen in chapter 7 on signatures that ECDSA requires you to generate a random nonce every time you sign, whereas EdDSA does not. Another example you've seen in

chapter 4 on authenticated encryption is AES-GCM-SIV, which does not catastrophically breaks down if you happen to re-use the same nonce twice, as opposed to AES-GCM which will leak the authentication key and will then lose integrity of the ciphertexts.

8.5 Public Randomness

There are other ways to obtain randomness. In this section I briefly survey these, as they are applicable only to a few specific settings. Note that these are sources of "public" randomness, and are thus not suitable to use for generating secret keys (indeed, anybody observing the public randomness would then be able to derive your secret keys as well).

Verifiable Random Function. Imagine that you have a signature scheme which provides unique signatures based on a keypair and a message. This is true for some signature schemes (like the BLS signature scheme) but not true for ECDSA and EdDSA. (Can you see why?) You now have a very simple way to obtain random numbers from someone holding a keypair, and a way to verify that the random number was generated properly. The owner of the private key can compute the random_number as such (for some public value seed):

```
signature = sign(private_key, seed); random_number = hash(signature)
```

And the signature can be released as well, serving as a proof that the random number was generated correctly.

You can verify that the random number was computed correctly by verifying the signature on the seed. Since the signature is unique, and the seed is fixed, there is no way for the signer to generate a different random number.

Note that it is up to the protocol using VRFs to enforce what the seed is, as long as it is a public value that every participant knows.

Randomness Beacons. In some scenarios, for example a lottery game, you might want to randomly decide on a winner using a third-party trusted source. There do exist services (called randomness beacons) that you can use to obtain random numbers from, but not all are secure. One amusing example is the book "A Million Random Digits with 100,000 Normal Deviates", which received mixed reviews:

I was duped by the title of this book. It is supposed to be about random digits. And at first glance you do see randomness. But after reading the book a while I started seeing a pattern. I did extensive research to prove my theory. After hours of mathematical modeling I conclusively proved that there is a set of numbers in this book that is not only a pattern, but is outright sequential! The top corner of each page (left corner on the left side pages, right corner of the right side pages) was a list of sequential numbers from 1 to 628, all in a row. No numbers are skipped. Even the prime numbers are included! At first you don't notice this because there is

only 1 number on each page. But as you advance through the book you notice that the numbers keep advancing by 1 every time you turn the page.

– Obi Wan Random? It lists almost 600 integers in numerical order!

One of the available randomness beacon (at the time of this writing) is called **drand**, and is a distributed service generating verifiable (like VRFs) random numbers. It is available at www.leagueofentropy.com

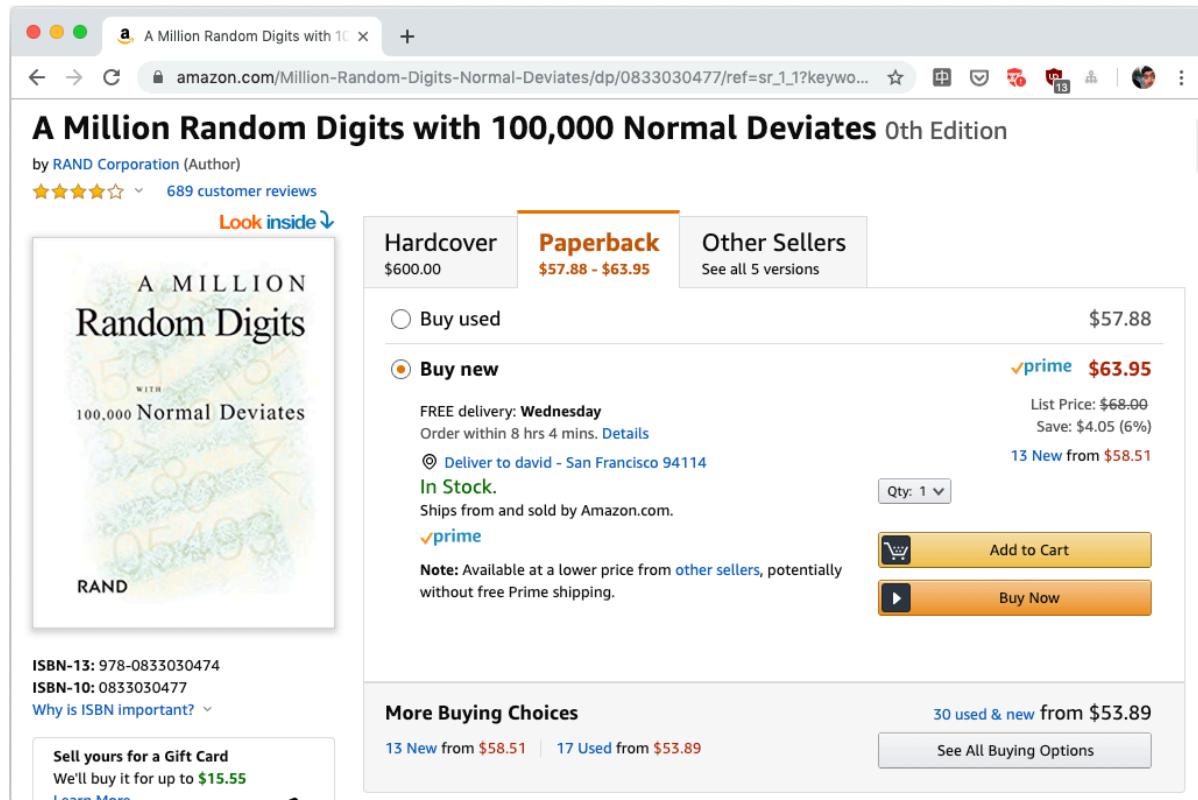


Figure 8.10 A book you can purchase on Amazon.com that contains a list of "random" numbers.

Now that I've talked extensively about randomness and how programs obtain it nowadays, let's move the discussion towards the role of secrets in cryptography and how one can manage them.

8.6 Key Derivation With HKDF

PRNGs are not the only constructions one can use to derive more secrets from one secret (in other words, to "stretch" a key). Deriving several secrets from one secret is actually such a frequent pattern in cryptography that this concept was given its own name: **Key Derivation**.

A **Key Derivation Function (KDF)** is like a PRNG in many ways, except for a number of subtleties:

- A KDF does not necessarily expect a uniformly random secret (as long as it has enough entropy). This makes a KDF very useful for deriving secrets from key exchanges output

which produced high-entropy but biased results (see chapter 5 on key exchanges). The resulting secrets are in turn uniformly random and can be used in constructions that expect uniformly random keys.

- A KDF is usually used in protocols that require participants to re-derive the same keys several times. In this sense, a KDF is expected to be deterministic, while PRNGs sometimes provide backward secrecy by frequently re-seeding themselves with more entropy.
- A KDF is usually not designed to produce a LOT of random numbers, instead it is usually used to derive a limited number of keys.

These differences are summarized in figure 8.10.

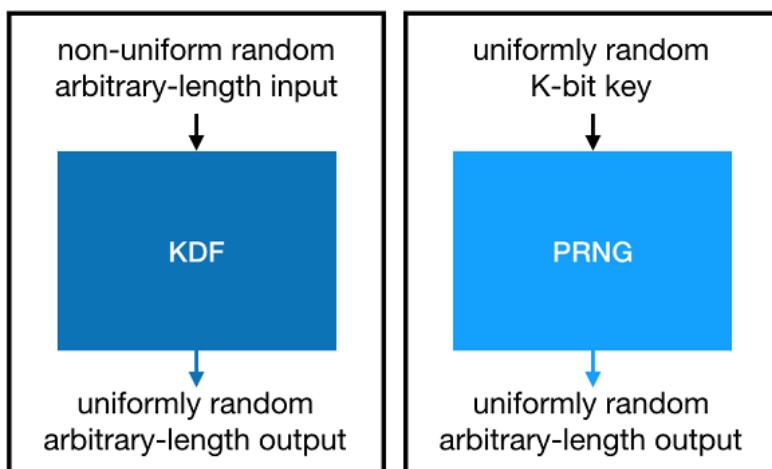


Figure 8.11 A Key Derivation Function (KDF) and a PRNG are two similar constructions. The main differences are that a KDF does not expect a fully uniformly random secret as input (as long as it has enough entropy), and is usually not used to generate too much output.

The most popular KDF is **Hash-based Key Derivation Function (HKDF)**. You've learned about HMAC in chapter 3, a MAC based on hash functions. HKDF is a light KDF built on top of HMAC and defined in RFC 5869.⁵⁸ For this reason, one can use HKDF with different hash functions (although it is most commonly used with SHA-2).

HKDF is specified as two different functions:

- **HKDF-Extract.** This function is used to remove biases from the input secret.
- **HKDF-Expand.** This function is more similar to a PRNG, it expects a uniformly random secret (and is thus run with the output of HKDF-Extract) and produces an arbitrary-length and uniformly random output.

Let's see HKDF-Extract first (which I illustrated in figure 8.11 below). Technically a hash function is enough to uniformize the randomness of an input bytestring (remember, the output of a hash function is supposed to be indistinguishable from randomness), but HKDF goes further and accepts one additional input: a `salt`. As for password hashing, a `salt` is here to differentiate different usages of HKDF-Extract in the same protocol. While this `salt` is optional (and set to an

all-zero bytestring if not used), it is recommended to use it. Furthermore, HKDF does not expect the `salt` to be a secret, it can be known to everyone including adversaries.

Instead of a hash function HKDF-Extract uses a MAC (specifically **HMAC**) which coincidentally has an interface that accepts two arguments.

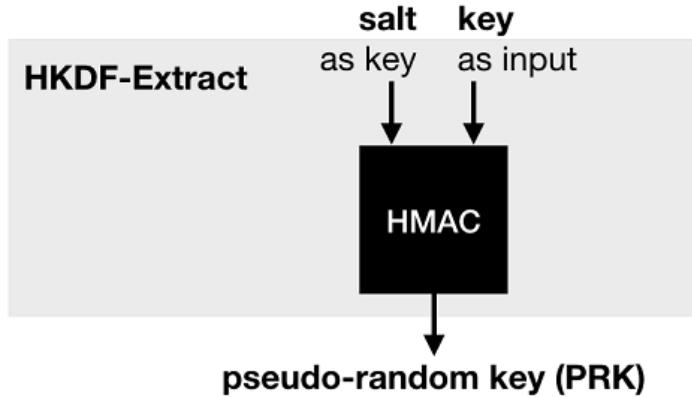


Figure 8.12 HKDF-Extract is the first function specified by HKDF. It takes an optional salt (which will be used as the key in HMAC) and the input secret that might be non-uniformly random. Using different `salt`s with the same input secret will produce a different output.

Let's now look at HKDF-Expand (which I illustrated in figure 8.12 below). If your input secret is already uniformly random, you can skip HKDF-Extract and directly use this one. Similar to HKDF-Extract, it also accepts an additional and optional customization argument called `info`. While `salt` is meant to provide some domain separation between calls of HKDF (or HKDF-Extract) within the same protocol, `info` is meant to be used to differentiate your version of HKDF (or HKDF-Expand) from other protocols. You can also specify how much output you want, but keep in mind that HKDF is not a PRNG and is not designed to derive a large number of secrets. HKDF is limited by the size of the hash function used, more precisely if you use SHA-512 (which produces outputs of 512-bits) with HKDF, you will be limited to $512 * 255$ bits = 16,320 bytes of output for a given key and `info` bytestring.

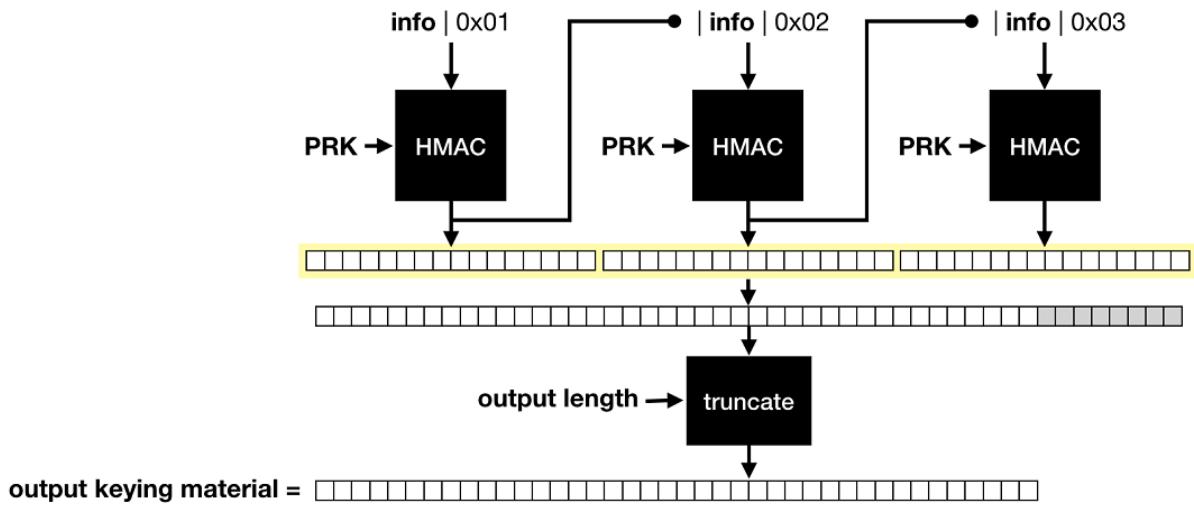


Figure 8.13 HKDF-Expand is the second function specified by HKDF. It takes an optional info bytestring and an input secret that needs to be uniformly random. Using different info bytestrings with the same input secret will produce different outputs. The length of the output is controlled by a length argument.

Calling HKDF or HKDF-Expand several times with the same arguments except for the output length, will produce the same output truncated to the different length requested (see figure 8.13). This property is called **related outputs** and can, in rare scenarios, surprise protocol designers. It is good to keep this in mind.

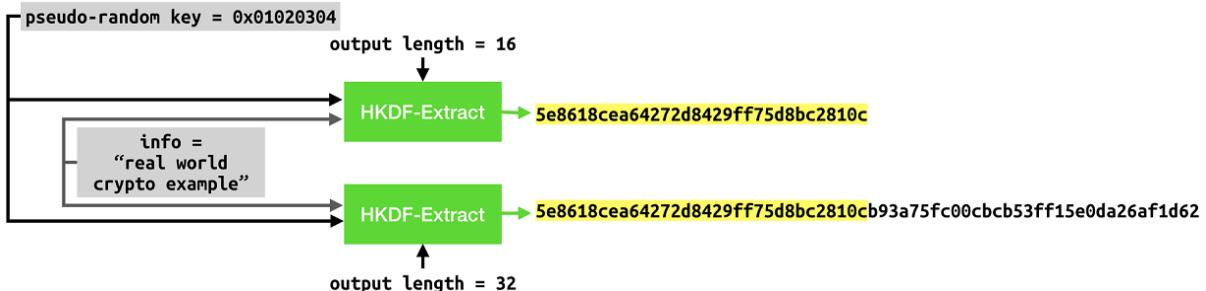


Figure 8.14 HKDF and HKDF-Expands provide related outputs, meaning that calling the function with different output lengths will just truncate the same result to the requested length.

Most cryptographic libraries will combine the HKDF-Extract and HKDF-Expand into a single call, as illustrated in figure 8.14. As usual, make sure to read the manual (in this case RFC 5869) before using HKDF.

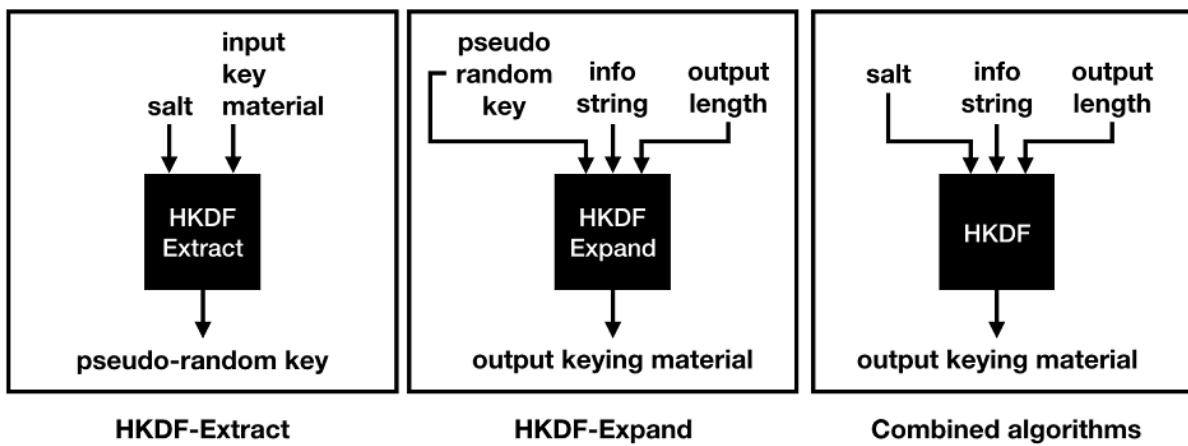


Figure 8.15 HKDF is usually implemented as a single call, that combines both HKDF-Extract (to extract uniform randomness from an input key) and HKDF-Expand (to generate an arbitrary-length output).

HKDF is not the only way to derive multiple secrets from one secret. A more naive approach is to use hash functions. As hash functions do not expect a uniformly random input, and produce uniformly random outputs, they are fit for the task. Hash functions are not perfect though, as their interface does not take into account domain separation (no customization string argument) and their output length is fixed. For this reason, best practice is to avoid hash functions when you can use a KDF instead. Nonetheless, some well-accepted algorithms do use hash functions for this purpose. For example, you've learned in the previous chapter 7 about the Ed25519 signature scheme that hashes a 256-bit key with SHA-512 to produce two 256-bit keys.

NOTE

In theory, a hash function's properties do not say anything about the output being uniformly random, the properties only dictates that a hash function should be collision resistant, pre-image resistant and second pre-image resistant. In the real-world though, we use hash functions all over the place to implement random oracles (as you've learned in chapter 2) and thus we assume that their outputs are uniformly random. This is the same with MACs which are in theory not expected to produce uniformly random outputs (unlike PRFs, as seen in chapter 3), but in practice do for the most part, and this is why HMAC is used in HKDF. In the rest of this book, I will assume that popular hash functions (like SHA-2 and SHA-3) and popular MACs (like HMAC and KMAC) produce random outputs.

The eXtended Output Functions (XOFs) we've seen in chapter 2 (SHAKE and cSHAKE) can be used as a KDF as well! Remember, a XOF:

- does not expect a uniformly random input
- can produce a practically infinitely large uniformly random output

In addition KMAC (that you've learned about in chapter 3) does not have the related output issue I've mentioned earlier. Indeed, KMAC's length argument randomizes the output of the algorithm

(effectively acting like an additional customization string).

Finally, there exist an edge-case for inputs that have low entropy. Think about passwords for example, that can be relatively guessable compared to a 128-bit key. The same constructions used to "hash" passwords that you've learned in chapter 2 can be used to derive keys as well. These so-called **Password-Based Key Derivation Functions** can be useful in these rare and non-ideal cases where a high-entropy secret cannot be used.

8.7 Managing Keys and Secrets

Alright, all good, we know how to generate cryptographic random numbers and we know how to derive secrets in different types of situations. We're not out of the woods yet. Now that we're using all of these cryptographic algorithms, we end up having to maintain a lot of secret keys. How do we store these keys? And how do we prevent these extremely sensitive secrets from being compromised? And what do we do if a secret is compromised? This problem is commonly known as **key management**.

Crypto is a tool for turning a whole swathe of problems into key management problems.
– Lea Kissner

While many applications choose to leave keys close to the application that makes use of them, this does not necessarily mean that they have no recourse when bad things happen.

To prepare against an eventual breach, or a bug that would leak a key, most serious applications employ two defense-in-depth techniques:

- **Key rotation.** By associating an expiration date to a key (usually a public key), and by replacing your key with a new key periodically, you can heal from an eventual compromise. The shorter the expiration date and rotation frequency, the faster you will replace a key that might be known to an attacker.
- **Key revocation.** Key rotation is not always enough, and you might want to be able to "cancel" a key as soon as you hear it has been compromised. For this reason, some systems allow you to ask if a key has been revoked before making use of it. You will learn more about this in the next chapter 9 on secure transport.

Automation is often key to successfully using these techniques, as a well-oiled machinery is much more akin to work correctly in times of crisis.

Furthermore, you can also associate a particular role to a key in order to limit the amount of compromise. For example, you could differentiate two public keys in some fictive application as:

- public key 0x5678... is only for signing transactions
- public key 0x8530... is only for doing key exchanges

This would allow a compromise of public key 0x8530 not to impact signing of transactions.

If one does not want to leave keys lying around, hardware solutions exist that attempt at preventing keys from being extracted.

Hardware Security Modules (HSMs) are one of the prime example of hardware key management, as most banks use them to store important secrets. HSMs are hardware devices that can manage keys, and that you most often find directly plugged into servers that make use of them. Once a key is generated by an HSM, it can perform cryptographic operations on request without having to retrieve the key from it. Different vendors implement different techniques to ensure that it is hard (and expensive) to recover a private key store on an HSM. Some HSMs will even wipe out their keys if they detect any physical attacks.

Note that while the HSM's goal is to prevent extraction of key, one can still make any request to the HSM or even steal the HSM to use it themselves (but this is often pretty obvious) and are such not panacea for all types of attacks.

There exist other types of hardware devices that can hold keys, you will learn more about them in chapter 12 on hardware cryptography.

Finally, there exist a lot of ways for applications to delegate key management. This is often the case on mobile OS's that provide "key stores" or "key chains" which will keep keys for you and even perform cryptographic operations. Applications living in the cloud can sometimes have access to cloud key management services, or even cloud HSMs. These services allow an application to delegate creation of secret keys and cryptographic operations and avoid thinking about the many ways to attack them.

None-the-less, as with HSMs, if an application is compromised it will still be able to do any types of request to the delegated service. So again, these are not silver bullets and you should still wonder what you can do if a secret is compromised.

Key management is a hard problem that is often not too much about cryptography, so I will not dwell on this topic too much.

In the next section, I will go over cryptographic techniques that attempts at avoiding the key management problem!

8.8 Avoiding Key Management, Or How To Split Trust

Key management is a vast field of study, which can be quite annoying to invest in as users do not always have the resources to implement best practices or tools available in the space.

Fortunately, cryptography has something to offer for those who wish to avoid key management issues, or at least lessen their burden.

The first one I'll talk about is **secret sharing** (or **secret splitting**). Secret splitting allows you to

break a secret in multiple parts that can be shared among a set of participants.

Here a secret can be anything you want, a symmetric key, a signing private key, etc.

Typically, a person called a dealer will generate the secret, then split it and share the different parts among all participants before deleting the secret. I illustrated this process in figure 8.15>.

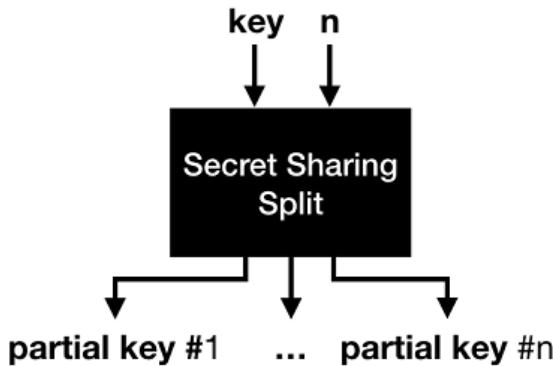


Figure 8.16 Given a key and a number of shares n , the Shamir Secret Sharing scheme creates n partial keys of the same size as the original key.

When time comes, and the secret is needed to perform some cryptographic operation (encrypting, signing, etc.), all share owners need to provide their private shares back to the dealer who will be in charge of reconstructing the original secret.

Such a scheme prevents attackers from targeting a single user, as each share is useless by itself, and instead force attackers to compromise all the participants before they can exploit a key. I illustrated this in figure 8.16.

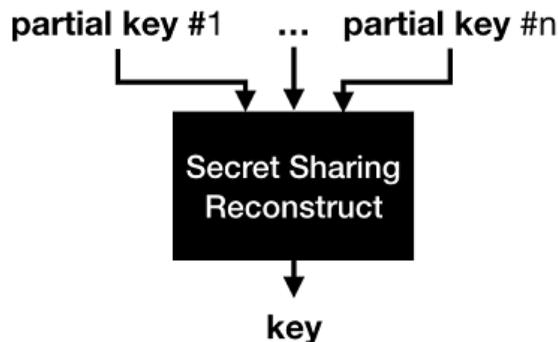


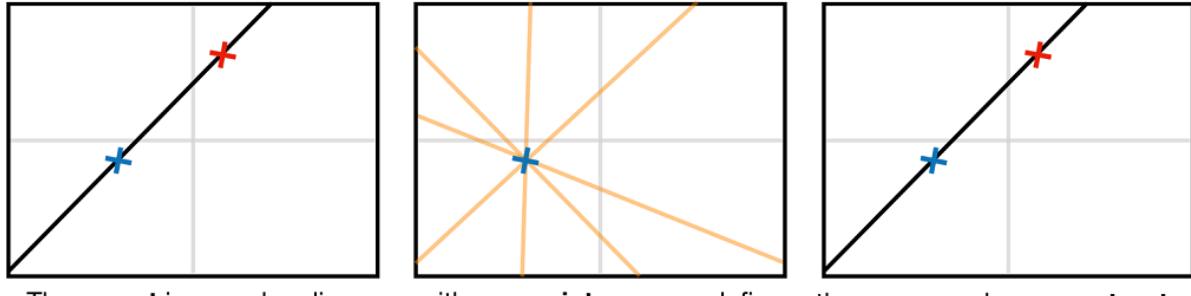
Figure 8.17 The Shamir Secret Sharing scheme used to split a secret in n partial keys, will require all of the n partial keys to reconstruct the original key.

The most famous secret splitting scheme was invented by Adi Shamir (one of the co-inventor of RSA) and is called **Shamir Secret Sharing (SSS)**.

The mathematics behind the algorithm are actually not too hard to understand! So let me spare a few paragraphs here to give you a simplified idea of the scheme.

Imagine a random straight line on a 2-dimensional space, and let's say that its equation $y = ax + b$ is the secret.

By having two participants hold two random points on the line, they can collaborate to recover the line equation. This is illustrated in figure 8.17.



The **secret** is a random line.
pick two **random points** on
the line as **partial keys**

with **one point**, you can define
an **infinite number of**
lines passing through it

the curve can be **reconstructed**
only with the knowledge
of the **two points**

Figure 8.18 The idea behind the Shamir Secret Sharing scheme is to see a polynomial defining a curve as the secret, and random points on the curve as partial keys. To recover a polynomial of degree n defining a curve, one needs to know $n+1$ points on the curve. For example, $f(x) = 3x + 5$ is of degree 1 so you need any two points $(x, f(x))$ to recover the polynomial; $f(x) = 5x^2 + 2x + 3$ is of degree 2 so you need any three points to recover the polynomial.

The scheme generalizes to polynomials of any degree, and thus can be used to divide a secret into an arbitrary number of shares.

Secret splitting is a technique often adopted due to its simplicity. Yet, in order to be useful key shares must be gathered into one place in order to recreate the key, every time the key has to be used in a cryptographic operation. This creates a window of opportunity in which the secret becomes vulnerable to robberies or accidental leaks, effectively getting us back to a **single point of failure** model.

To avoid this single point of failure issue, there exist several cryptographic techniques that can be useful in different scenarios.

For example, imagine a protocol that accepts a financial transaction only if it has been signed by Alice. This places a large burden on Alice, who might be scared of getting targeted by attackers. In order to reduce the impact of an attack on Alice, we can change the protocol to accept a number n of signatures instead (on the same transaction) from n different public keys including Alice's. An attacker would thus have to compromise all n signatures in order to forge a valid transaction. Such systems are called **multi-signature** (often shortened as multi-sig) and have seen a lot of adoption in the cryptocurrency space.

Naive multi-signature schemes can add some annoying overhead. Indeed, the size of a transaction in our example grows linearly with the number of signatures required. To solve this,

some signatures schemes (like the **BLS signature scheme**) can have their signatures compressed down to a single signature. This is called **signature aggregation**.

Some multi-signature schemes go even further in the compression by allowing the n public keys to be aggregated into a single public key. This technique is called **distributed key generation (DKG)**.

This might be a bit confusing to understand. In other words: n participants can collaboratively compute a public key without ever having the associated private key in clear during the process (unlike SSS, there is no dealer).

If they want to sign a message, they can then collaboratively create a signature (using each participant's private shares) that can be verified using the public key they previously created. Again, the private key never exists in clear, preventing the single point of failure problem SSS had. I recapitulate all these examples in figure 8.18.

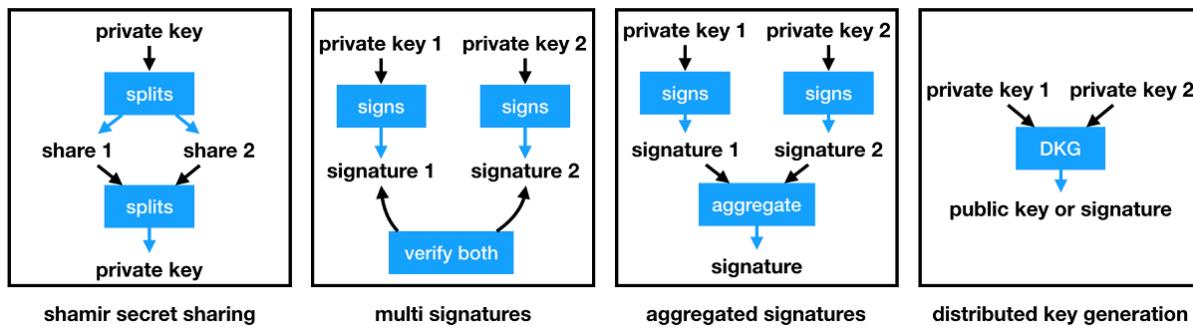


Figure 8.19 A recap of existing techniques to split trust we have in one participant into several participants.

Finally, note that:

- These types of schemes can work with other asymmetric cryptographic algorithms like encryption, where participants must all collaborate to asymmetrically decrypt a message for example.
- Each schemes I've mentioned can be made to work even when only a threshold m out of the n participants helps. Although the term **threshold signatures** often refer to the threshold version of DKG.

If you're interested, check chapter 12 on cryptocurrencies where I'll go into more details about these schemes.

8.9 Summary

- A number is taken uniformly and at random from a set, if it was picked with equal probability compared to all other numbers from that set.
- Entropy is a metric to indicate how much randomness a bytestring has. High-entropy refers to bytestrings that have are uniformly random, while low-entropy refers to bytestrings that are easy to guess or predict.
- Pseudo-Random Number Generators (PRNGs) are algorithms that take a uniformly random seed, and generate (in practice an infinite) amount of randomness that can be used for cryptographic purposes (as cryptographic keys for example). That is if the seed is large enough.
- To obtain random numbers, one should rely on a programming language's standard library or on its well-known cryptographic libraries.
- If these are not available, OS's generally provide interfaces to obtain random numbers:
 - Windows offers the `BCryptGenRandom` system call.
 - Linux and FreeBSD offer the `getrandom` system call.
 - Other OS's usually have a special file `/dev/urandom` exhibiting randomness.
- Key Derivation Functions (KDFs) are useful in scenarios where one wants to derive secrets from a biased but high-entropy secret.
- HKDF is the most widely used KDF, and is based on HMAC.
- Key management is the field of keeping secrets, well, secret. It mostly consists of:
 - Finding where to store secrets.
 - Proactively expiring and rotating secrets.
 - Figuring out what to do when secrets are compromised.
- To lessen the burden of key management, one can split the trust of one participant into multiple participants.

9

Secure transport

This chapter covers

- Transport Security Protocols, protocols used to encrypt communications between machines.
- The Transport Layer Security (TLS) protocol, the most widely used transport security protocol.
- Other transport security protocols in-use.

You now enter the second part of this book, which is going to make use of most of what you've learned in the first part of this book. Most, if not all chapters in this part will make heavy use of the first part. For this reason, if you have any doubt, go back to the basics.

The most widely used protocol to secure communications is SSL/TLS. This is what you use every day to browse the web! For this reason, most of this chapter is about SSL/TLS and how it works for the web. I will also lightly cover other secure transport protocols and how they differ from SSL/TLS.

This chapter is quite long, as there is a lot to say on the topic of securing communications. After all, cryptography was invented for this purpose.

Brace yourself!

9.1 What is SSL/TLS?

In order to understand why we came up with SSL/TLS as a secure transport protocol, let's walk through the following scenario.

When you enter www.example.com in your web browser, your browser uses a number of

protocols to connect to a web server and retrieve the page you requested (see figure 9.1).

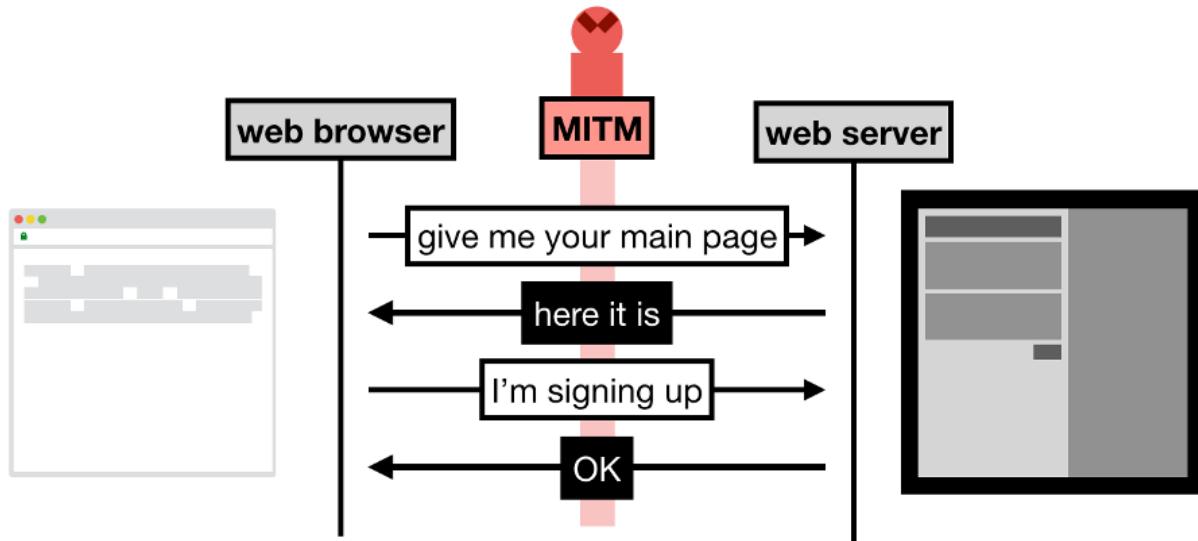


Figure 9.1 An insecure connection between a web browser and a web server using the HTTP protocol. A man-in-the-middle (MITM) attacker on the network can passively observe requests and responses, and actively tamper, replay and reorder messages.

Simplified, the browser requests pages and parses responses using the **Hypertext Transfer Protocol (HTTP)**. HTTP uses a human-readable format (you can look at the HTTP messages and read them without the help of any tool) to wrap the actual content of the requests and responses. HTTP messages are encapsulated using the Transmission Control Protocol (TCP), which is a binary protocol and thus not human-readable (you need a tool to understand what each byte of the request means). TCP messages are also encapsulated using the Internet Protocol (IP), and IP messages are further encapsulated... If you want to learn more about all these networking layers, many books exist.⁵⁹

NOTE Cryptography can mostly be found (although not always) in the application layer (the very last layer used by the application itself).

In our scenario, anyone sitting on the wire in between your browser and the web server can passively observe and read your requests and the server's responses. Worse, the MITM attacker can also actively tamper and reorder messages.

This is not great, imagine your credit card information leaking every time you buy something on the internet, your passwords being stolen every time you log into a website, your pictures and private messages snooped on as your communicate with your friends and loved ones.

Thus, the **Secure Sockets Layer (SSL)** protocol was invented in the 90s by a team over at Netscape (the most popular browser at the time). After that, the **Hypertext Transfer Protocol Secure (HTTPS)** protocol allowed browsers to secure their communications to websites by combining HTTP with the newly born SSL. I illustrate this change in figure 9.2.

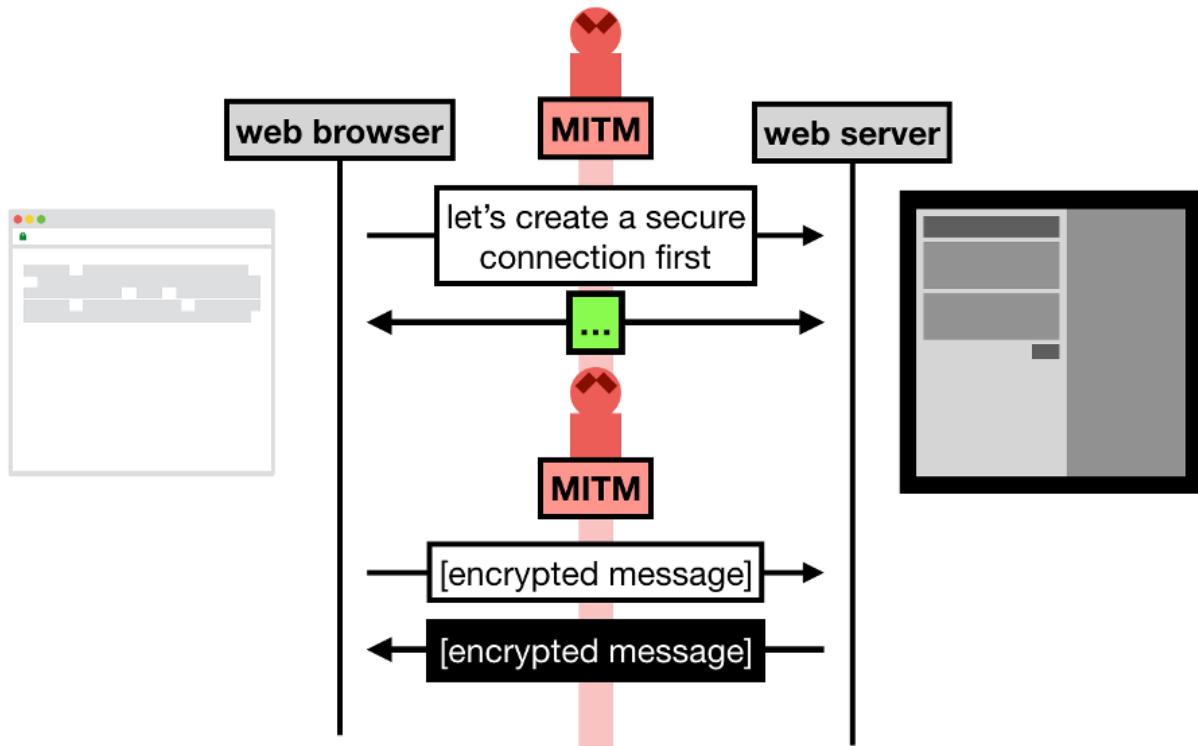


Figure 9.2 A secure connection between a web browser and a web server using the HTTPS protocol. The MITM attacker only sees encrypted messages and is unable to passively decrypt them or actively tamper, replay or reorder them.

While SSL was not the only protocol that attempted to secure some of the web, it did attract most of the attention and with time has become the de-facto standard.

But this is not the whole story, between the first version of SSL and what we currently use today, a lot has happened! All versions of SSL (the last being SSL version 3.0) were broken due to a combination of bad design and bad cryptographic algorithms.⁶⁰ After SSL 3.0, the protocol was officially transferred to the Internet Engineering Task Force (IETF), the organization in charge of publishing **Request For Comments (RFCs)** standards. The name SSL was dropped in favor of **Transport Layer Security (TLS)**, and TLS 1.0 was released in 1999 as RFC 2246.

The most recent version of TLS is **TLS version 1.3**, specified in RFC 8446 and published in 2018. In this chapter, I will explain how TLS 1.3 version works.

Today, the internet is divided between many different versions of SSL and TLS (many vulnerable to attacks), as servers have been slow to update. As the two different names (SSL and TLS) bring a lot of confusion, many tools use the name "SSL" and articles might often use the name SSL in place of TLS.

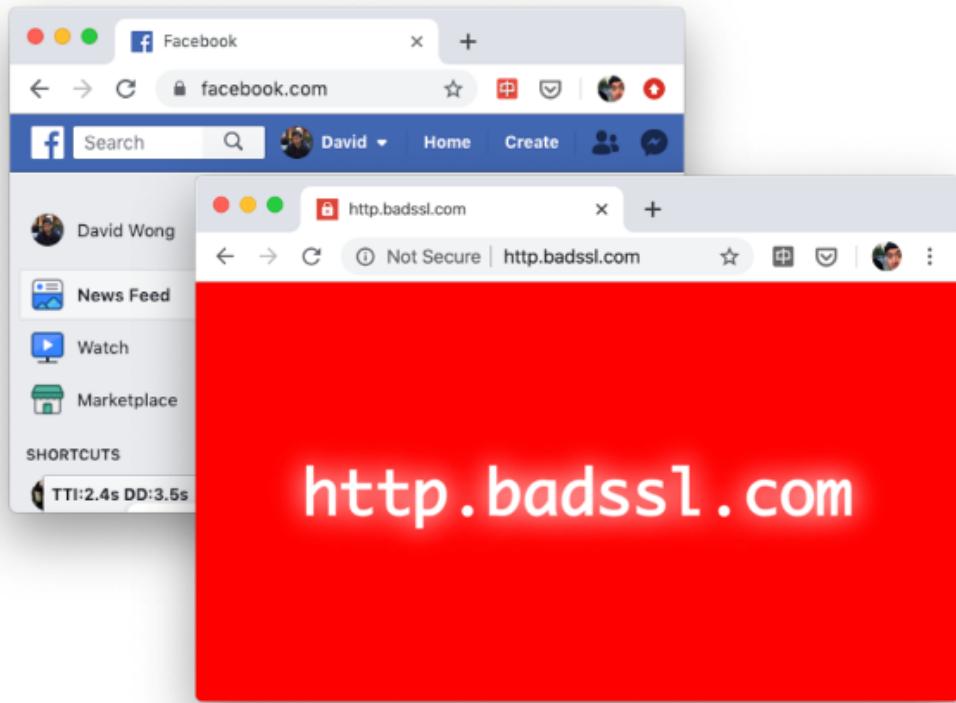


Figure 9.3 Historically, addresses would start with either `http://` or `https://` indicating which protocol they were using. Nowadays browsers like Chrome simply display a lock next to the address when using HTTPS, and a "not secure" warning when using HTTP without TLS. (www.badssl.com is a useful page to test browsers in different settings.)

Furthermore, TLS has become more than just the protocol securing the web, it is now being used in many different scenarios and between many different types of applications and devices as a protocol to secure communications. Thus what you will learn about TLS is not just useful for the web, but for any scenarios where communications between two applications need to be secured.

In TLS terms, the two participants that want to secure their communications are called a **client** and a **server**. The client always being the one that initiates the connection.

The interface for a TLS client, as illustrated in figure 9.4, typically takes:

- Some **configuration** containing information like the versions of SSL/TLS that the client supports, algorithms that a client is willing to use to secure the connection, ways the client can authenticate servers, and so on.
- Some **information about the server it wants to connect to**. It includes at least an IP address and a port, and for the web it most often includes a fully qualified domain name (like www.google.com).

Once given these two arguments, a client interface produces a secure **session** (a stateful

connection that expires, unless kept alive) that the application can use to securely communicate with the server. In some cases a secure session cannot successfully be created and fails midway. For example, if an attacker attempts to tamper with the connection, or if the server configuration is not compatible with the client's (more on that later).

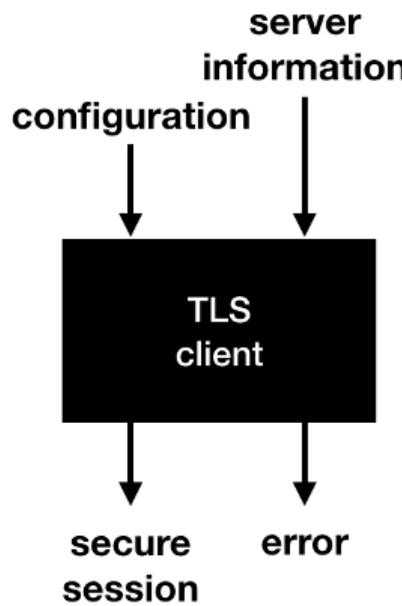


Figure 9.4 A client in TLS is the one who initiates the connection. For example in HTTPS, the web browser is the client. A client must be configured, and must be provided with enough information about the server it wants to connect to (typically an ip address and port) to create a secure connection with it.

The interface for a **TLS server** is even simpler (illustrated in figure 9.5). It simply takes a **configuration** (which is very similar to the configuration of the client) and waits for a client to connect to it in order to produce a secure connection.

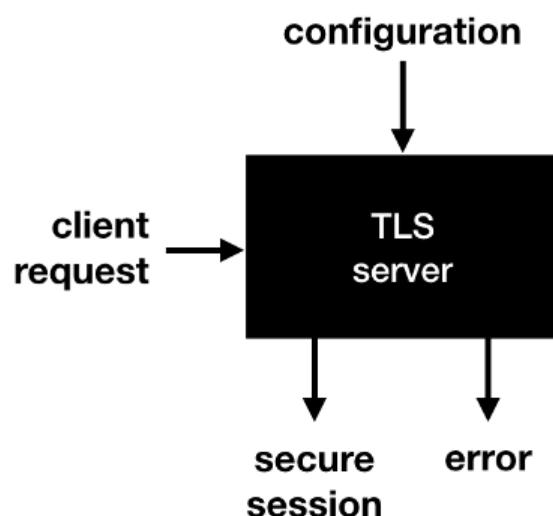


Figure 9.5 A server in TLS is the one who waits for clients to connect to it. For example in HTTPS, the website is the server. Once configured, it awaits for clients' connection requests. TLS implementations typically provide interfaces for servers to accept multiple clients simultaneously.

In practice, using TLS on the client side can be as easy as the following snippet of code using the standard library of Golang:

Listing 9.1 tls_server.go

```
import "crypto/tls"

func main() {
    destination := "www.google.com:443"      ①
    TLSconfig := &tls.Config{}                ②
    conn, err := tls.Dial("tcp", destination, TLSconfig)
    if err != nil {
        panic("failed to connect: " + err.Error())
    }
    conn.Close()
}
```

- ① The fully qualified domain name and the port (443 is the default port for HTTPS).
- ② An empty config serves as default configuration.

Notice that Golang does not require the IP address of the server, this is because Golang's standard TLS library resolves it for you automatically.

You might wonder, how does this code knows that it is really talking to www.google.com? By default Golang's TLS implementation uses your operating system configuration to figure out how to authenticate websites it wants to connect to (more on that later).

Using TLS on the server side is pretty easy as well using Golang's standard library, as the following code snippet shows:

Listing 9.2 tls_server.go

```
import "crypto/tls"

func main() {
    config := &tls.Config{          ①
        MinVersion: tls.VersionTLS13,  ②
    }  ②

    http.HandleFunc("/", func(rw http.ResponseWriter, req *http.Request) { ③
        rw.Write([]byte("Hello, world\n"))
    })  ③

    server := &http.Server{          ④
        Addr:      ":8080",           ⑤
        TLSConfig: config,           ③
    }

    err := server.ListenAndServeTLS("cert.pem", "key.pem")  ④
    if err != nil {
        log.Fatal(err)
    }
}
```

- ① A solid minimal configuration for a TLS 1.3 server.

- ② Serves a simple page displaying "Hello, world".
- ③ An HTTPS server is started on port 8080.
- ④ Some .pem files containing a certificate and a secret key (more on this later).

Golang does a lot for us here. Unfortunately not all languages' standard libraries provide easy-to-use TLS implementations (if they provide a TLS implementation at all), and not all TLS libraries provide secure-by-default implementations. For this reason, configuring a TLS server is not always straightforward depending on the library.

NOTE

TLS is a protocol that works on top of **TCP**. To secure **UDP** connections, **DTLS** (D for Datagram) can be used and is fairly similar to TLS, for this reason I will ignore DTLS in this chapter.

In the next section, you will learn about the inner workings of TLS and the different subtleties that can provide more or less security to your applications.

9.2 How Does TLS Work?

As I said earlier, today TLS is the de-facto standard to secure communications between applications. In this section you will learn more about how TLS works underneath the surface, and how it is used in practice. You will find this section useful to understand how to use TLS properly, but also to understand how most (if not all) secure transport protocols work. (You will also find out why it is hard and strongly discouraged to re-design or re-implement such protocols.)

At a high level, TLS is split into two phases:

- A **handshake** phase where a secure communication is negotiated and created between two participants.
- A **post-handshake** phase where communications are encrypted between the two participants.

This idea is shown in illustration 9.6

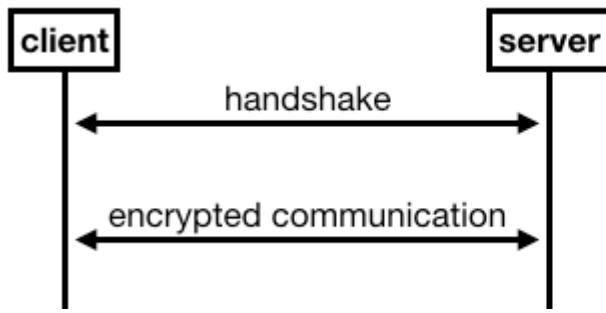


Figure 9.6 At a high level, secure transport protocols first create a secure connection during a handshake phase. After that applications on both sides of the secure connection can communicate securely.

At this point, since you have learned about hybrid encryption in chapter 6, you should have the following (correct) intuition about how these two steps works:

- The handshake is, at its core, simply a key exchange. The handshake ends up with the two participants agreeing on a set of symmetric keys.
- The post-handshake phase is purely about encrypting messages between participants, using an authenticated encryption algorithm and the set of keys produced at the end of the handshake.

Most transport security protocols work this way, and the interesting parts of these protocols always lie in the handshake phase.

Next, let's take a look at the handshake phase.

9.2.1 The TLS Handshake

As you've seen, TLS is (and most transport security protocols are) divided into two parts: a handshake and a post-handshake phase. In this section you'll learn about the handshake first. The handshake itself has 4 aspects that I want to tell you about:

1. Negotiation. TLS is highly configurable. Both a client and a server can be configured to negotiate a range of SSL and TLS versions, as well as a menu of acceptable cryptographic algorithms. The negotiation phase of the handshake aims at finding common ground between the client's and the server's configurations, in order to securely connect the two peers.
2. Key exchange. The whole point of the handshake is to perform a key exchange between the two participants. What key exchange algorithm to use? This is one of the things decided as part of the negotiation process.
3. Authentication. As you've learned in chapter 5 on key exchanges, it is trivial for a MITM attacker to impersonate any side of a key exchange. For this reason, key exchanges must be authenticated. (Your browser must have a way to make sure that it is talking to google.com and not your Internet service provider for example.)
4. Session Resumption. As browsers often connect to the same websites again and again, key exchanges can be costly and slow down a user's experience. For this reason, mechanisms to fast-track secure sessions without re-doing a key exchange are integrated into TLS.

This is a long list. As fast as greased lightning, let's start with the first item.

NEGOTIATION IN TLS: WHAT VERSION AND WHAT ALGORITHMS?

Most of the complexity in TLS comes from the negotiation of the different moving parts of the protocol. Infamously, this negotiation has also been the source of many issues in the history of TLS. Attacks like FREAK, LOGJAM, DROWN, and others... took advantage of weaknesses present in older versions to break more recent versions of the protocol (as long as a server offered to negotiate these versions). While not all protocols have versioning, or allow for different algorithms to be negotiated, SSL/TLS was designed for the web. As such, SSL/TLS needed a way to maintain backward compatibility with older clients and servers that could be slow to update.

This is what happens on the web today: your browser might be recent and up-to-date, made to support TLS 1.3, but visiting some old web page chances are that the server behind it only supports versions of TLS up to 1.2 or 1.1 (or worse). Vice-versa, many websites must support older browsers (and thus older versions of TLS) as some users are stuck in the past.

NOTE

Most versions of SSL/TLS have security issues (except for TLS 1.2 and TLS 1.3), which should make you think that it is safer to only support TLS 1.3 and maybe TLS 1.2 but nothing more. Yet, some large companies must support a large amount of older TLS clients due to their business. It is not uncommon to find TLS libraries that support these older and broken versions by implementing mitigations to known attacks. Mitigations that are often too complex to implement. For example, the Lucky13 and Bleichenbacher98 attacks which broke some versions of the protocol (and thus every implementations) have been mitigated in many "hardened" TLS implementations, yet researchers have been re-discovering them in different TLS implementations pretty much once every year since they were found.

Negotiation starts with the client sending a first request called a **Client Hello** to the server. The Client Hello contains a range of supported SSL and TLS versions, a suite of cryptographic algorithms that the client is willing to use, and some more information that can be relevant for the rest of the handshake or for the application. The suite of cryptographic algorithms include:

- One or more **key exchange** algorithms. TLS 1.3 defines the following algorithms allowed for negotiations: ECDH with P-256, P-384, P-521, X25519, X448; and FFDH with the groups defined in RFC 7919. I've talked about all of these in chapter 5. Previous versions of TLS also offered RSA key exchanges (covered in chapter 6) but they were removed in the last version.
- Two (for different parts of the handshake) or more **digital signature** algorithms. TLS 1.3 specifies RSA PKCS#1 v1.5 and the newer RSA-PSS, as well as more recent elliptic curve algorithms like ECDSA and EdDSA. I've talked about these in chapter 7. Note that

digital signatures are specified with a hash function, allowing you to negotiate for example RSA-PSS with SHA-256 or/and SHA-512.

- One or more **hash functions** to be used with HMAC (the message authentication code you've learned in chapter 3) and HKDF (the key derivation function covered in chapter 8). TLS 1.3 specifies SHA-256 and SHA-384, two instances of the SHA-2 hash function. You've learned about SHA-2 in chapter 2. This choice of hash function is unrelated to the one used by the digital signature algorithm.
- One or more **authenticated encryption** algorithms. These can include AES-GCM (with keys of 128 or 256 bits), ChaCha20-Poly1305, and AES-CCM. I've talked about all of these in chapter 4.

The server then responds with a **Server Hello** message that contains one of each type of cryptographic algorithms, cherry-picked from the client's selection.



If the server is unable to find an algorithm it supports, it has to abort the connection. Although in some cases, the server does not have to abort the connection and can ask the client to provide more information instead. To do this, the server replies with a message called a Hello Retry Request asking for specific piece of information. The client can then re-send its Client Hello, this time with the added requested information.

TLS AND FORWARD SECURE KEY EXCHANGES

The key exchange is the most important part of the TLS handshake! Without it, there's obviously no symmetric key being negotiated. But for a key exchange to happen, the client and the server must first trade their respective public keys.

In TLS 1.2 and previous versions, the key exchange is done once both participants know which key exchange algorithm to use. This means that they first negotiate on which algorithm to use, and then exchange their public keys.

In TLS 1.3, to avoid that first negotiating round trip (one client message and one server

message), the client speculatively sends a public key in the very first message (the Client Hello). If the client fails to predict the server's choice of key exchange algorithm then the client will have to send a new Client Hello containing the correct public key. For example:

- The client sends a TLS 1.3 Client Hello announcing that it can do either an X25519 or an X448 key exchange. It also sends an X25519 public key.
- The server does not support X25519, but does support X448. It sends a Hello Retry Request to the client announcing that it only supports X448.
- The client sends the same Client Hello but with an X448 public key instead.
- The handshake goes on.

I illustrate this difference in figure 9.7.

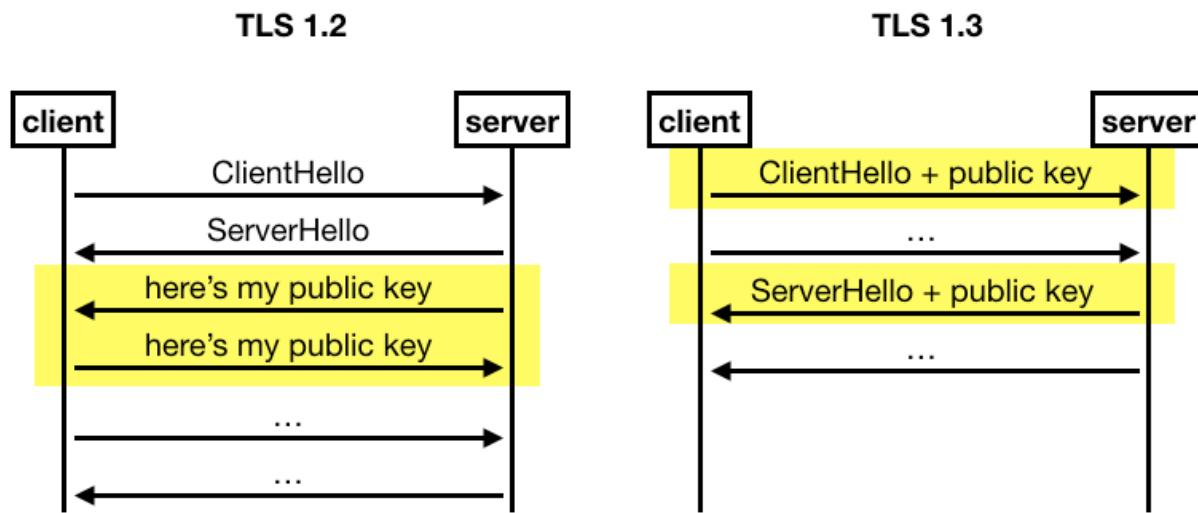


Figure 9.7 In TLS 1.2, the client waits for the server to choose which key exchange algorithm to use before sending a public key. In TLS 1.3, the client speculates on which key exchange algorithm(s) the server will settle on, and preemptively sends a public key (or several) in the first message, potentially avoiding an extra round trip.

TLS 1.3 is full of such optimizations, which are important for the web. Indeed many people in the world have unstable or slow connections, and it is important to keep non-application communication to the bare minimum required.

Furthermore, in TLS 1.3 and unlike previous versions of TLS, all key exchanges are **ephemeral**. This means that for each new session, the client and the server both generate new key pairs, then get rid of them as soon as the key exchange is done. Why? The key word is **forward secrecy**! In chapter 8 you learned about forward secrecy with pseudo-random number generators. The same concept applies here.

Imagine what would happen if instead, a TLS server used a single private key for every key exchange it performed with its clients.

A compromise of the server's private key at some point in time would be devastating, as a MITM attacker would then be able to decrypt all previously recorded conversations. (Do you

understand how?)

Instead, by performing ephemeral key exchanges and getting rid of private keys as soon as a handshake ends, the server protects against such attackers. I illustrate this in figure 9.8.

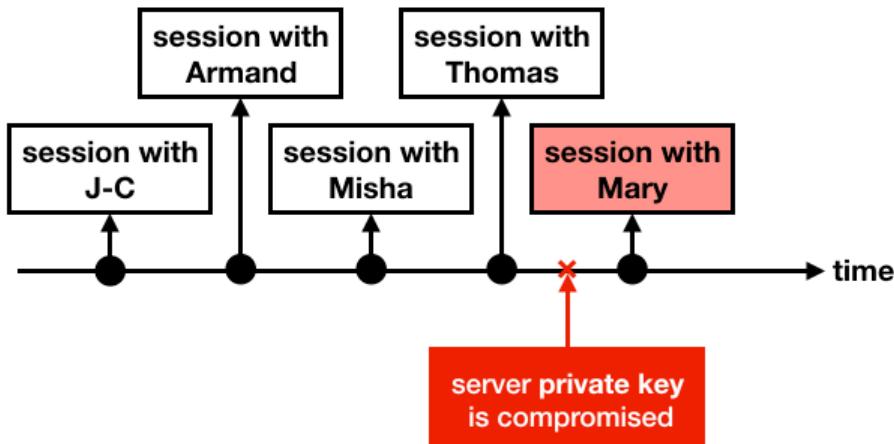


Figure 9.8 In TLS 1.3 each session starts with an ephemeral key exchange. If a server is compromised at some point in time, no previous sessions will be impacted.

Once ephemeral public keys are traded, a key exchange is performed, and keys can be derived. TLS 1.3 uses derive different keys at different points in time, to encrypt different phases with independent keys.

The first two messages, the Client Hello and the Server Hello, cannot be encrypted as no public keys were traded at this point. But after that, as soon as the key exchange happens, TLS 1.3 encrypts the rest of the handshake. (This is unlike previous versions of TLS that did not encrypt any of the handshake messages.)

To derive the different keys, TLS 1.3 uses HKDF with the hash function negotiated. HKDF-Extract is used on the output of the key exchange (to remove any biases) while HKDF-Expand is used with different `info` parameters to derive the encryption keys. For example, "c hs traffic" (for client handshake traffic) is used to derive symmetric keys for the client to encrypt to the server during the handshake, "s ap traffic" (for "server application traffic") is used to derive symmetric keys for the server to encrypt to the client after the handshake.

Don't forget, unauthenticated key exchanges are insecure, next you'll see how TLS addresses this.

TLS AUTHENTICATION AND THE WEB PUBLIC KEY INFRASTRUCTURE

After some negotiations, and after the key exchange has taken place, the handshake must go on. What happens next is the other most important part of TLS: **authentication**. You've seen in chapter 5 on key exchanges that it is trivial to intercept a key exchange and impersonate one (or both) sides of the key exchange. In this section I'll explain how your browser cryptographically validates that it is talking to the right website, and not to an impersonator.

But let's take a step back. There is something I haven't told you so far. A TLS 1.3 handshake is actually split in three different stages (as illustrated in figure 9.9):

1. **Key Exchange.** This phase contains the ClientHello and ServerHello messages which provide some negotiation and perform the key exchange. All messages (including handshake messages) after this phase are encrypted.
2. **Server Parameters.** Messages in this phase contain additional negotiation data from the server. This is negotiation data that does not have to be contained in the first message of the server, and that could benefit from being encrypted.
3. **Authentication.** This phase includes authentication information from both the server and the client.

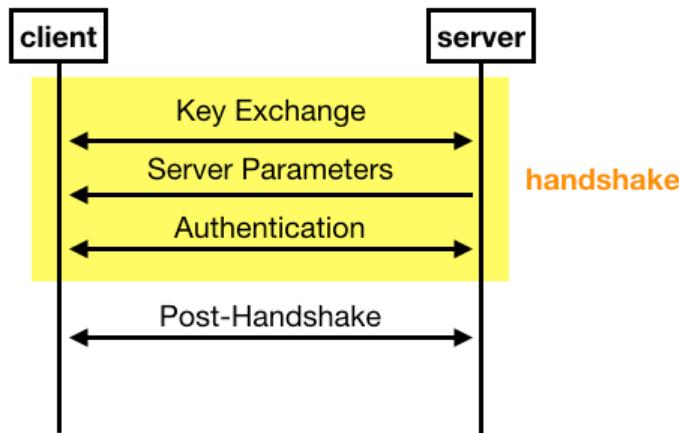


Figure 9.9 A TLS 1.3 handshake is divided into 3 phases: the key exchange phase, the server parameters phase, and finally the authentication phase.

On the web, **authentication in TLS is usually one-sided**. Only the browser verifies that google.com is indeed google.com, but google.com does not verify who you are (or at least not as part of TLS).

NOTE

Client authentication is often delegated to the application layer for the web (via a form asking you for your credentials most often). That being said, client authentication can also happen in TLS if requested by the server (during the *server parameters* phase). When both sides of the connection are authenticated, we talk about **mutually-authenticated TLS** (sometimes abbreviated as mTLS). Client authentication is done the exact same way as server authentication, and can happen at any point in time after the authentication of the server (during the handshake or in the post-handshake phase).

Let's now answer the question: when connecting to google.com, how does your browser verify that you are indeed handshaking with google.com?

Using the **web public key infrastructure (web PKI)**!

You've learned about the concept of public key infrastructure in chapter 7 on digital signatures, but let me briefly re-introduce this concept as it is quite important to understand how the web works.

There are two sides to the web PKI:

First, **Browsers must trust a set of root public keys** that we call **Certificate Authorities (CAs)**. Usually, browsers will either use a hardcoded set of trusted public keys or rely on the operating system to provide them.

NOTE

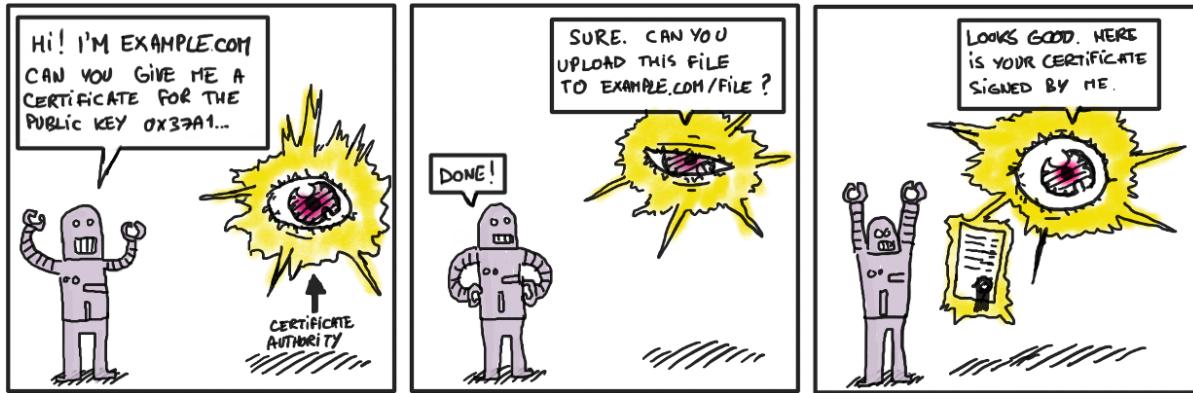
For the web, there exist hundreds of these CAs which are independently run by different companies and organizations across the world. It is quite a complex system to analyze and these CAs can sometimes also sign the public keys of intermediate CAs that then also have the authority to sign the public keys of websites. For this reason, organizations like the Certification Authority Browser Forum (CA/Browser forum) enforce rules and decide when new organizations can join the set of trusted public keys, and when a CA can no longer be trusted and must be removed from that set.

Second, **websites who want to use HTTPS must have a way to obtain a certification from these CAs** (a signature of their signing public key). In order to do this, a website owner (or a webmaster as we used to say) must prove to a CA that they own a specific domain.

NOTE

Obtaining a certificate for your own website used to involve a fee, this is no longer the case nowadays as Certificate Authorities like **Let's Encrypt** provide these for free.

For example, to prove that you own www.example.com a CA might ask you to host a file at www.example.com/some_path/file.txt that contains some random numbers generated for your request.



After this, a CA can provide a signature over the website's public key. As the CA's signature is usually valid for a period of years, we say that it is over a **long-term** signing public key (as opposed to an ephemeral public key). More specifically, CAs do not actually sign public keys, but instead they sign **certificates** (more on this later). A certificate contains the long-term public key, along with some additional important metadata like the name of your domain if you are a web page.

To prove to your browser that the server it is talking to is indeed google.com, the server sends a **certificate chain** as part of the TLS handshake which comprises of:

1. Its own (leaf) certificate containing among others the name google.com, google's long-term signing public key, as well as a signature from a CA.
2. A chain of intermediate CA certificates, from the one that signed google's certificate to the root CA that signed the last intermediate CA.

This is a bit wordy so I illustrated this in figure 9.10.

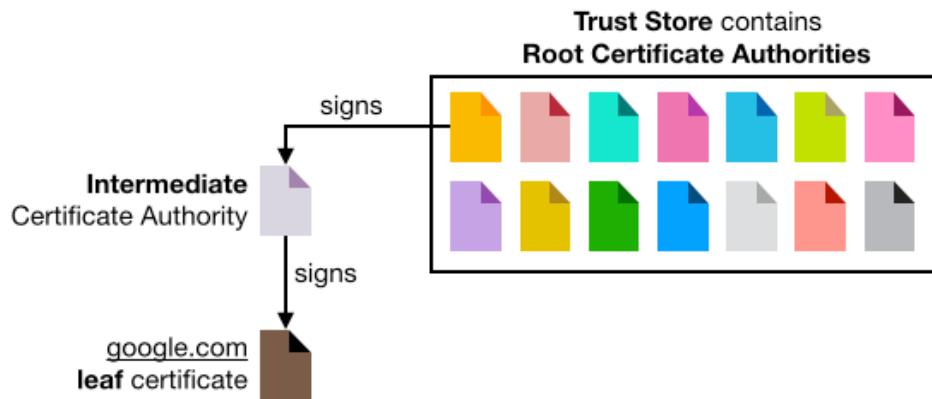


Figure 9.10 Web browsers only have to trust a relatively small set of root Certificate Authorities (CAs) in order to trust the whole web. These CAs are stored in what is called a trust store. In order for a website to be trusted by a browser, the website must have its (leaf) certificate be signed by one of these CAs. Sometimes root CAs only sign intermediate CAs, who in turn sign other intermediate CAs or leaf certificates. This is called the web public key infrastructure (web PKI).

This certificate chain is sent in a **Certificate** TLS message by the server, and by the client as well if the client has been asked to authenticate.

Following this, the browser can use its certified long-term key pair to sign all handshake messages that have been received and sent since then (in what is called a **Certificate Verify** message).

I've recapitulated this flow (where only the server authenticates itself) in figure 9.11.

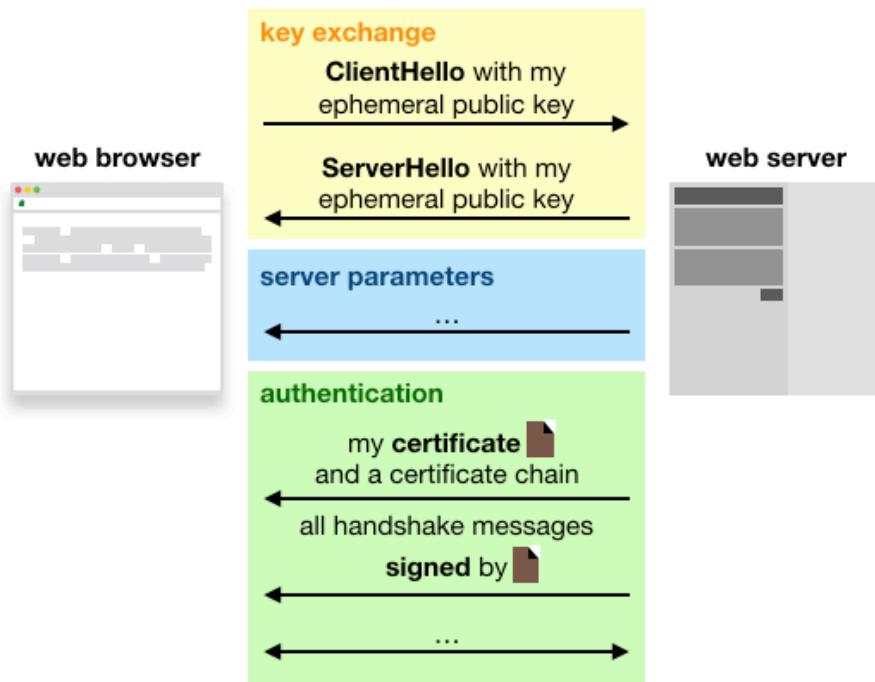


Figure 9.11 The authentication part of a handshake starts with the server sending a certificate chain to the client. The certificate chain starts with the leaf certificate (the certificate containing the website's public key and additional metadata like the domain name) and ends with a root certificate that is trusted by the browser. Each certificate contains a signature from the certificate above it in the chain.

The signature in the Certificate Verify message proves to the client what the server has seen so far. Without this signature, a MITM attacker could intercept the server's handshake messages and replace the ephemeral public key of the server contained in the Server Hello message, allowing the attacker to successfully impersonate the server.

Take a few moments to understand why an attacker cannot replace the server's ephemeral public key in the presence of the Certificate Verify signature.

SIDE BAR Story time.

A few years ago I was hired to review a custom-TLS protocol made by a large company. It turned out that their protocol had the server provide a signature that did not cover the ephemeral key. When I told them about the issue, the whole room went silent for a full minute.

It was of course a substantial mistake: an attacker who could have intercepted the custom handshake and replaced the ephemeral key with its own, would have successfully impersonated the server. The lesson here is that it is important not to reinvent the wheel. Secure transport protocols are hard to get right and if history has shown anything, they can fail in many unexpected ways. Instead, you should rely on mature protocols like TLS and make sure you use a popular implementation that has received a substantial amount of public attention.

Finally, in order to officially end the handshake, both sides of the connection must send a **Finished** message as part of the Authentication phase. A Finished message contains an authentication tag produced by HMAC (instantiated with the negotiated hash function for the session). This allows both the client and the server to tell the other side: "these are all the messages I have sent and received, in order, during this handshake". If the handshake was intercepted and tampered with by a MITM attacker, this integrity check can allow the participants to detect and abort the connection. (This is especially useful as some handshakes modes are not signed, more on this later).

Before heading to a different aspect of the handshake, let's double click on X.509 certificates, as they are an important detail of many cryptographic protocols.

AUTHENTICATION VIA X.509 CERTIFICATES

While certificates are optional in TLS 1.3 (you can always use plain keys), many applications and protocols (not just the web) make heavy use of them in order to certify additional metadata. Specifically, the **X.509 certificate standard version 3** is used.

X.509 is a pretty old standard that was meant to be flexible enough to be used in a multitude of scenarios, from email to web pages. The X.509 standards uses a description language called **Abstract Syntax Notation One (ASN.1)** to specify information contained in a certificate. An ASN.1 looks like this:

```
Certificate ::= SEQUENCE {
    tbsCertificate      TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING }
```

You can literally read this as a structure that contains three fields:

- `tbsCertificate`. The "to-be-signed" certificate. This contains all the information that one wants to certify. For the web, this can contain a domain name (www.google.com), a public key, an expiration date, and so on.
- `signatureAlgorithm`. The algorithm used to sign the certificate.
- `signatureValue`. The signature from a Certificate Authority.

By the way, the last two values are not contained in the actual certificate (`tbsCertificate`), do you know why?

You can easily check what's in an X.509 certificate by connecting to any website using HTTPS, and using your browser functionalities to observe the certificate chain sent by the server. See figure 9.12 for an example.

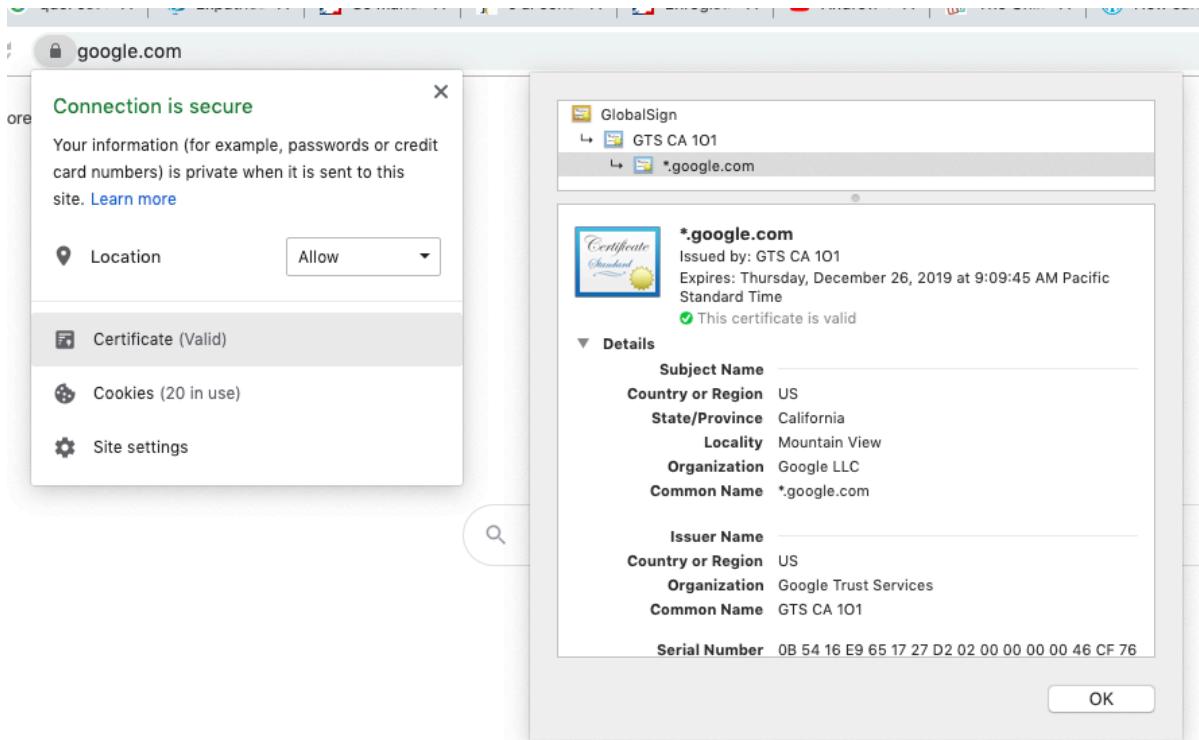


Figure 9.12 Using Chrome's Certificate viewer we can observe the certificate chain sent by a Google's server. The root Certificate Authority is "Global Sign" which is trusted by your browser. Down the chain an intermediate CA called "GTS CA 101" is trusted due to its certificate containing a signature from "Global Sign". In turn, Google's leaf certificate valid for *.google.com (www.google.com, mail.google.com, etc.) contains a signature from "GTS CA 101".

You might encounter X.509 certificates as .pem files, which is some base64 encoded content surrounded by some human-readable hint of what the base64 encoded data contains (here a certificate). The following snippet represents the content of a certificate in a .pem format:

```
-----BEGIN CERTIFICATE-----
MIJJQzCCCCugAwIBAgIQC1QW6WUXJ9ICAAAAAEbPdjANBgkqhkiG9w0BAQsFADBC
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2x1IFRydxN0IFNlcnZpY2VzMRMw
EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTE5MTAwMzE3MDk0NVoxDTE5MTIyNjE3MDk0
NVowZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWEfjAUBgNVBAcT
[...]
vaoUqelfNJJvQjBmQbSQEp9y8EiI4BnWGZjU6Q+q/3VZ7ybR3cOzhnaLGmqiwFv
4PNBdnVVfVbQ9CxRip1KVzzSnUvypgBLryYn16kquh1AJSSgnJhzogrz98IiXCQZ
c7mkvTKgCNIR9fedIus+LPHCSD7zUQTgRoOmcB+kwY7jrFqKn6tjhPnfB5aVNK
d10nq4fcF8PN+ppgNFbwC2JxX08L1wEFk2LvDOQgKqHR1TRJ0U3A2gkuMtf6Q6au
3KBzGW61/vt3coyyDkQKDmT61tjwy5k=
-----END CERTIFICATE-----
```

If you decode the base64 content surrounded by the BEGIN CERTIFICATE and END CERTIFICATE, you end up with a **Distinguished Encoding Rules (DER)** encoded certificate. DER is a deterministic (only one way to encode) binary encoding used to translate X.509 certificates into bytes. All these encodings are often quite confusing to new-comers! I recap all of this in figure 9.13.



Figure 9.13 On the top left corner, an X.509 certificate is written using the ASN.1 notation. It is then transformed into bytes that can be signed via the DER encoding. As this is not text that can easily be copied around and be recognized by humans, it is base64 encoded. The last touch wraps the base64 data with some handy contextual information using the PEM format.

DER only encodes information as "here is an integer" or "this is a bytearray". Fields' names described in ASN.1 like `tbsCertificate` are thus lost after encoding. Decoding DER without the knowledge of the original ASN.1 description of what each field truly means is thus pointless. Handy command line tools like OpenSSL allow you to decode and translate in human terms the content of a DER-encoded certificate. For example, if you download google.com's certificate, you can use the following command to display its content in your terminal:

```
$ openssl x509 -in google.pem -text
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      0b:54:16:e9:65:17:27:d2:02:00:00:00:00:46:cf:76
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = US, O = Google Trust Services, CN = GTS CA 101
    Validity
      Not Before: Oct 3 17:09:45 2019 GMT
      Not After : Dec 26 17:09:45 2019 GMT
    Subject: C = US, ST = California, L = Mountain View, O = Google LLC, CN = *.google.com
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
        Public-Key: (256 bit)
          pub:
            04:74:25:79:7d:6f:77:e4:7e:af:fb:1a:eb:4d:41:
            b5:27:10:4a:9e:b8:a2:8c:83:ee:d2:0f:12:7f:d1:
            77:a7:0f:79:fe:4b:cb:b7:ed:c6:94:4a:b2:6d:40:
            5c:31:68:18:b6:df:ba:35:e7:f3:7e:af:39:2d:5b:
            43:2d:48:0a:54
        ASN1 OID: prime256v1
        NIST CURVE: P-256
[...]
```

Having said all of this, X.509 certificates are quite controversial.

Validating X.509 certificates has been comically dubbed "The Most Dangerous Code in the World" by a team of researchers in 2012.⁶¹ This is because DER encoding is a difficult protocol to parse correctly, and the complexity of X.509 certificates makes for many mistakes to be potentially devastating. For this reason I don't recommend any modern application to use X.509 certificates, unless they have to.

PRE-SHARED KEYS AND SESSION RESUMPTION IN TLS, OR HOW TO AVOID KEY EXCHANGES

Key exchanges can be costly, and are sometimes not needed. For example, you might have two machines that only connect to each other, and you might not want to have to deal with a public-key infrastructure in order to secure their communications. TLS 1.3 offers a way to avoid this overhead with **pre-shared keys (PSK)**.

A pre-shared key is simply a secret that both the client and the server know, and that can be used to derive symmetric keys for the session.

In TLS 1.3, a PSK handshake works by having the client advertise in its Client Hello message that it supports a list of PSK identifiers. If the server recognizes one of them, it can say so in its response (the Server Hello message) and both can avoid doing a key exchange (if they want to). By doing this, the authentication phase is skipped, making the Finished message at the end of the handshake important to prevent MITM attacks.

NOTE

Note that this does not mean that the same set of symmetric keys is derived for every session using the same PSK. Using different keys for different sessions is extremely important, as you do not want these sessions to be linked. Worse, since encrypted messages might be different between sessions, this could lead to nonce reuses and their catastrophic implications (see chapter 4). To mitigate this, both the Client Hello and Server Hello messages have a `random` field which is randomly generated for every new session. These random fields are used in the derivation of symmetric keys in TLS, effectively creating never-seen-before encryption keys every time you create a new connection.

Another use-case for PSKs is **session resumption**. Session resumption is about reusing secrets created from a previous session or connection. If you have already connected to google.com and have already verified their certificate chain and agreed on a shared secret, why do this dance again a few minutes or hours later? If you know how browsers work, you probably know that they also create several TCP connections to a web pages they visit in order to quickly load a page, do we really want to do a full TLS handshake for all of these separate TCP connections as well?

TLS 1.3 offers a way to generate a PSK after a handshake was successfully performed, which

can be used in subsequent connections to avoid having to re-do a full handshake.

If the server wants to offer this feature, it can send **New Session Ticket** message(s) at any time during the post-handshake phase. There are multiple ways for the server to create so-called "session tickets". For example, the server can send an identifier, associated to the relevant information in a database (forcing the server to keep a state), or the server can send the authenticated encryption of the required information to perform session resumption with the client (allowing the server's session resumption mechanism to be stateless). These are not the only ways, but as this mechanism is quite complex and most of the time not necessary I won't touch more of it in this chapter.

Next let's see the easiest part of TLS: how application data is encrypted.

9.2.2 How TLS 1.3 Encrypts Application Data

Once a handshake has taken place, and symmetric keys have been derived, both the client and the server can send each other encrypted application data. But this is not all, TLS ensures that such messages cannot be replayed nor reordered.

To do this, the nonce used by the authenticated encryption algorithm starts at a fixed value and is incremented for each new message. If a message is re-played, or reordered, the nonce will be different from what is expected and decryption will fail. When this happens the connection is killed.

NOTE

As you've learned in chapter 4, encryption does not always hide the length of what is being encrypted. TLS 1.3 comes with **record padding**, which can be configured to pad application data with a random number of zero bytes before encrypting it, effectively hiding the true length of the message. In spite of this, statistical attacks that remove the added noise can exist, and it is not straightforward to mitigate them. If you really require this security property, you should refer to the TLS 1.3 specification.

Starting in TLS 1.3, clients have the possibility to send encrypted data as part of their first series of messages (right after the Client Hello message), that is if the server agrees. This means that browsers do not necessarily have to wait until the end of the handshake to start sending application data to the server. This mechanism is called **early data** or **0-RTT** (for zero round-trip-time) and can only be used with the combination of a PSK (as it allows derivation of symmetric keys during the Client Hello message).

This feature was quite controversial during the development of the standard, as a passive attacker can replay an observed Client Hello followed by the encrypted 0-RTT data. This is why 0-RTT must be used only with application data that can be replayed safely.

For the web, browsers treat every GET queries as *idempotent*, meaning that they should not change any state on the server side and are only meant to retrieve data (unlike POST queries for example). This is of course not always the case, and applications have been known to do whatever they want to do. For this reason, if you are confronted with the decision of using 0-RTT or not, it is simpler just not to use it.

9.3 The State of the Encrypted Web Today

Today, standards are pushing for the deprecation of all versions of SSL and TLS that are not TLS 1.2 and TLS 1.3. Yet, due to legacy clients and servers, many libraries and applications continue to support older versions of the protocol (up to SSL version 3 sometimes!). This is not straightforward and due to the number of vulnerabilities you need to defend against, many hard-to-implement mitigations must be maintained.

WARNING Using TLS 1.3 (and TLS 1.2) is considered secure and best practice. Using anything lower means that you will need to consult experts and will have to figure out how to avoid known vulnerabilities.

By default, browsers still connect to web servers using HTTP and websites still have to manually ask a CA to obtain a certificate. This means that with the current protocols, the web will never be fully encrypted (although some estimates global web traffic to be 90% encrypted in 2019).⁶²

The fact that by default, your browser always uses an insecure connection is also an issue. Web servers nowadays usually redirect users accessing their page using HTTP towards HTTPS. Web servers can also (and often do) tell browsers to use HTTPS for subsequent connections. This is done via an HTTPS response header called **HTTP Strict Transport Security (HSTS)**. Yet, the very first connection to a website is still unprotected (unless the user thinks about typing "https" in the address bar) and can be intercepted to remove the redirection to HTTPS.

In addition, other web protocols like NTP (to get the current time) and DNS (to obtain the IP behind a domain name) are currently largely unencrypted and subject to man-in-the-middle attack. While there are research efforts to improve the status quo,⁶³⁶⁴ these are limitations one need to be aware of today.

There's another threat to TLS users, and this is misbehaving CAs. What if today, a CA decides to sign a certificate for your domain and a public key that they control? If they can obtain a man-in-the-middle position, they could start impersonating your website to your users.

The obvious solution if you control the client-side of the connection is to either not use the web PKI (and rely on your own PKI), or to **pin** a specific certificate or public key. **Certificate or public key pinning** is simply about hardcoding in the client code the hash of the leaf certificate, or the public key that the leaf certificate contains, and to only accept a server that uses these for

the authentication phase of the handshake. This practice is often used in mobile applications, as they know exactly what the server's public key or certificate should look like (unlike browsers that have to connect to an infinite number of servers).

This is not always possible though, and two other mechanisms exist:

- **Certificate revocation.** Like the name indicates, this allows a CA to revoke a certificate and warn browsers about it.
- **Certificate monitoring.** This is a relatively new system that forces CAs to publicly log every certificate they sign.

The story of certificate revocation has historically been bumpy. The first solution proposed was **Certificate Revocation Lists (CRLs)**, which allowed CAs to maintain a list of certificates that they had revoked (were no longer considered valid). The problem with CRLs is that they could grow quite large, and that one needs to constantly check them. They were deprecated in favor of **Online Certificate Status Protocol (OCSP)**, which are simple web interfaces that one can query to know if a certificate is revoked or not. OCSP has its own share of problems: they require CAs to have a highly-available service that can answer to OCSP requests, they leak web traffic information to the CAs, and browsers often decide to ignore OCSP requests that time out (to not disrupt the user's experience).

The current solution is to augment OCSP with **OCSP Stapling**: the website is in charge of querying the CA for a signed status of its certificate, and attach (staple) the response to its certificate during the TLS handshake.

I recapitulate the three solutions in figure 9.14.

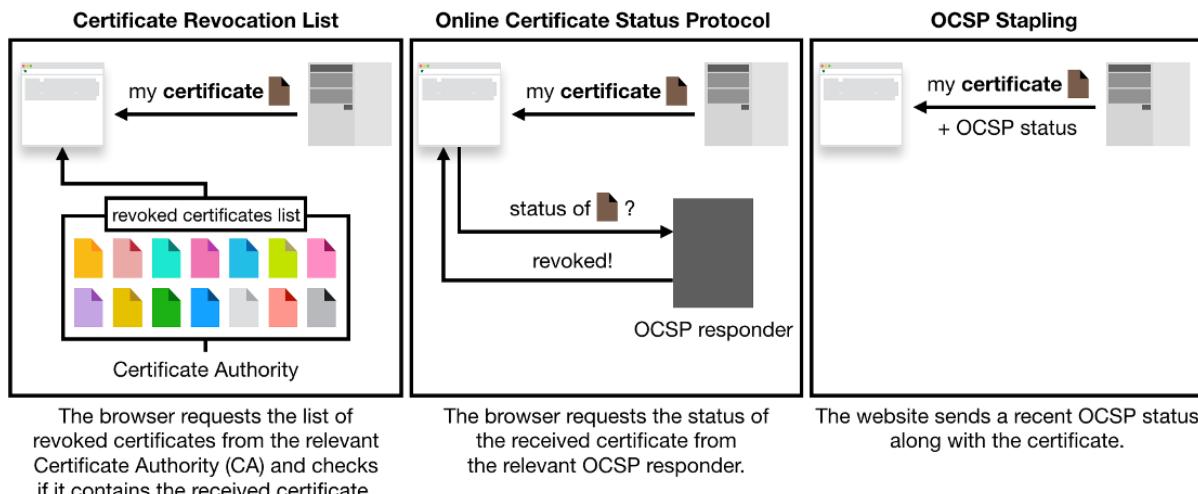


Figure 9.14 Certificate revocation on the web has had three popular solutions: Certificate Revocation Lists (CRL), Online Certificate Status Protocol (OCSP), and OCSP Stapling.

Certificate revocation might not seem to be a prime feature to support (especially for smaller systems compared to the web), that is until a certificate gets compromised. Like a car seatbelt,

certificate revocation is a security feature that is useless most of the time, but can be a lifesaver in rare cases (what we call **defense-in-depth** in security).

NOTE For the web, certificate revocation has largely proven to be a good decision. In 2014, the **Heartbleed** bug was and turned out to be one of the most devastating bugs in the history of SSL and TLS. The most widely used SSL/TLS implementation (OpenSSL) was found to have a buffer overread bug, allowing anyone to send a specially crafted message to any OpenSSL server and receive a dump of its memory, often revealing its long-term private keys.

Yet, if a CA truly misbehaves, it can decide not to revoke malicious certificates, or not to report it. The problem is that we are blindly trusting a non-negligible number of actors (the CAs) to do the right thing. To solve this issue at scale, **Certificate Transparency** was proposed in 2012 by Google.

The idea behind certificate transparency is to force CAs to add each certificate they issue to a giant log of certificates for everyone to see. To do this, browsers like Chrome now reject certificates if they do not include proofs of inclusion in a public log. This transparency allows you to check if a certificate was wrongly issued for a domain you own for example (there should be no other certificates than the ones you requested).

Note that Certificate Transparency relies on people monitoring logs to catch bad actors **after the fact** (and that is, if you're looking). It also still requires CAs to react and revoke mis-issued certificates once detected, or browsers to remove misbehaving CAs from their trust stores. It is thus not as powerful as certificate and public key pinning which mitigate all CA misbehaviors.

9.4 Other Secure Transport Protocols

You've now learned about TLS, which is the most popular protocol to encrypt communications. Congratulations! You're not done yet though. TLS is not the only one in the class. Many other protocols exist, and you might most likely be using them already.

Yet, most of them are TLS-like protocols customized to support their specific use-case. This is the case for example with:

- **Secure Shell (SSH)**, the most widely used protocol and application to securely connect to a remote shell on a different machine.
- **Wi-Fi Protected Access (WPA)**, the most popular protocol to connect devices to private network access points or the internet.
- **IPSec**, one of the most popular virtual network protocols (VPNs) used to connect different private networks together. It is mostly used by companies to link different office

networks. As its name indicates, it acts at the IP layer and is often found in routers, firewalls, and other network appliances. Another popular VPN is **OpenVPN** which makes direct use of TLS.

All of these protocols typically re-implement the handshake/post-handshake paradigm, and sparkle some of their own flavors to it. This is not without issues, as many of the Wi-Fi protocols have been broken time and time again.

In this section, we will focus on two interesting and modern protocols that deserve a place in this book:

- The **Noise Protocol Framework**. Noise is framework, as opposed to a protocol that one can use out-of-the-box. It is meant to be used to derive different TLS-like protocols depending on what is needed. It is the responsibility of the application designer to decide what instance of the Noise protocol will be used.
- **Wireguard**. A virtual private network (VPN) protocol and tool that aims at becoming the standard solution on the market. It directly competes with other VPNs like IPSec and OpenVPN and relies on the Noise Protocol Framework.

Without further ado, let's take a look at Noise.

9.4.1 The Noise Protocol Framework: A Modern Alternative To TLS

TLS is now quite mature, and considered a solid solutions in most cases due to the attention it gets, yet it adds quite a lot of overhead to applications that makes use of it due to historical reasons, backward compatibility constraints, and overall complexity.

Indeed, in many scenarios where you are in control of all endpoints, you might not need all of the features that TLS has to offer.

The next best solution is called the **Noise Protocol Framework**.

The Noise Protocol Framework removes the runtime complexity of TLS by avoiding all negotiation in the handshake. A client and a server running noise follow a linear protocol that does not branch. Contrast this to TLS which can take many different paths depending on information contained in the different handshake messages. What Noise does is that it pushes all the complexity to the design phase. Developers who want to use the Noise protocol framework must decide what ad-hoc instantiation of the framework they want their application to use. (This is why it is called a framework and not a protocol.)

In such, they must first decide what cryptographic algorithms will be used, what side of the connection is authenticated, if any pre-shared key is used, etc. After that, the protocol is implemented and turns into a rigid series of messages (which can be a problem if one needs to update the protocol later on, while maintaining backward compatibility with devices that cannot be updated).

Briefly, the Noise Protocol Framework offers different **handshake patterns** you can choose from. Handshake patterns typically come with a name that indicates what is going on. For example, the **NN** handshake pattern indicates that the client does not authenticate itself (the first N), and neither does the server (the second N).

NOTE

Other handshake patterns use letters like `x` to mean "this participant sends its long-term public key as part of the handshake" and `k` to mean "this participant long-term public key is known to the other participant". For example, the **XK** pattern means that the client sends its long-term public key as part of the handshake while the server long-term public key is already known to the client.

Once a handshake pattern has been chosen, applications making use of it will never attempt to perform any of the other possible handshake patterns.

I will use the **NN** handshake pattern to explain how Noise works, as it is quite simple (but insecure as no one is authenticated).

In Noise's lingo the pattern is written like this:

```
NN:  
-> e  
<- ee, ee
```

Each line represents a message pattern, and the arrow indicates the direction of the message. Each message pattern is a succession of tokens (here there are only two: `e` and `ee`) that dictates what both sides of the connection have to do.

`e` means that the client must generate an ephemeral key pair and send the public key to the server. The server interprets this message differently: it must receive an ephemeral public key and store it.

`e, ee` means that the server must generate an ephemeral key pair and send the public key to the client, then it must do a Diffie-Hellman key exchange with the client's ephemeral (the first `e`) and its own ephemeral (the second `e`). On the other hand, the client must receive an ephemeral public key from the server, and use it to do a Diffie-Hellman key exchange as well.

NOTE

Noise uses a combination of defined tokens in order to specify different types of handshake. For example, the `s` token means a static key (another word for long-term key) as opposed to an ephemeral key, and the token `es` means that both participants must perform a Diffie-Hellman key exchange using the client's ephemeral key and the server's static key.

There's more to it: at the end of each message pattern (`e` and `ee`, `ee`), the sender also gets to transmit a payload. If a Diffie-Hellman key exchange has happened previously (not the case in the first message pattern `e`), the payload is encrypted and authenticated.

At the end of the handshake both participants derive a set of symmetric keys and start encrypting communications, similarly to TLS.

One particularity of Noise is that it continuously authenticates the handshake. To do this, both sides maintain two variables: a hash `h` and a chaining key `ck`. Each handshake message sent or received is hashed with the previous `h` value. I illustrate this in figure 9.15.

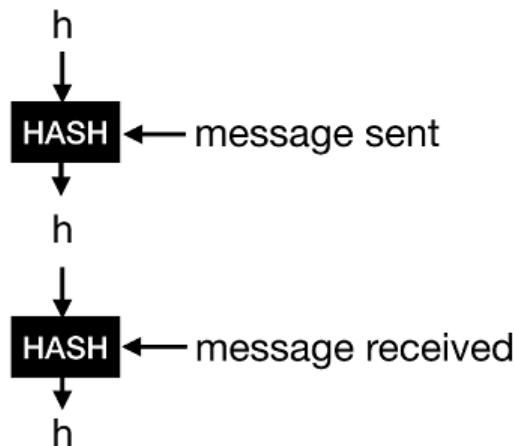


Figure 9.15 In the Noise Protocol Framework, each side of the connection keeps track of a digest `h` of all messages that have been sent and received during the handshake.

Whenever a message is sent encrypted with an authenticated encryption with associated data algorithm, the current `h` value is used as associated data in order to authenticate the handshake up to this point.

At the end of each message pattern, if the payload is sent encrypted, it also authenticates the `h` value using the associated data field of the AEAD algorithm used (I covered this in chapter 4).

This allows Noise to continuously verify that both sides of the connection are seeing the exact same series of messages and in the same order.

In addition, every time a Diffie-Hellman key exchange happens (several can happen during a handshake), its output is fed along with the previous chaining key `ck` to HKDF in order to derive a new chaining key and a new set of symmetric keys to use for authenticating and encrypting subsequent messages. I illustrate this in figure 9.16.

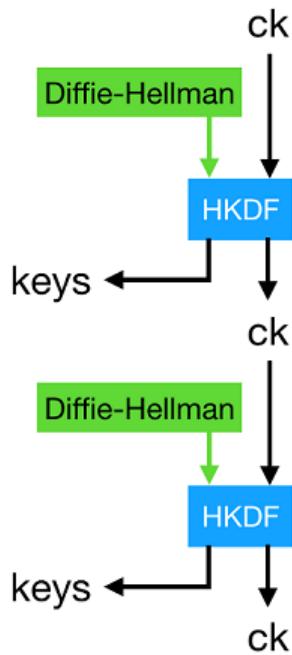


Figure 9.16 In the Noise Protocol Framework, each side of the connection keeps track of a chaining key ck . This value is used to derive a new chaining key and encryption keys to be used in the protocol every time a Diffie-Hellman key exchange is performed.

This makes Noise a very simple protocol at run-time: there is no branching and both sides of the connection simply do what they need to do. Libraries implementing Noise are also extremely simple, and end up being a few hundred lines compared to hundreds of thousands of lines for TLS libraries.

While Noise is more complex to use, and will require developers who understand how Noise works to integrate it into an application, it is a very good alternative to TLS.

9.4.2 Wireguard

Wireguard is a modern VPN solution whose goal is to become the standard solution on the market. It is open source software and available for free on mobile platforms and popular OS's (and is even integrated by default in the last version of the Linux kernel).

Unlike other popular VPN solutions like IPSec and OpenVPN, it is extremely simple, fast, and relies on the Noise Protocol Framework. For this reason it is quickly becoming the standard to connect to a remote private network.

At its core, it makes use of the IK handshake pattern of the Noise Protocol Framework:

```

IK:
<- s
...
-> e, es, s, ss
<- e, ee, se

```

Let's deconstruct the handshake pattern:

- The first `i` indicates that the client send its long-term public key as part of the first handshake message (the `s` in `ie`, `es`, `s`, `ss`). This is because a VPN gateway might have several clients (associated to different public keys) that can connect to it.
- The second `k` indicates that the server's long-term public key is known to the client (the `s` before the ...). This is because a VPN client always know who (and what public key) it is connecting to.

There is more to the design of Wireguard, but this chapter is already long enough. You can learn about it on www.wireguard.com.

9.5 Summary

- Transport Layer Security (TLS) is a secure transport protocol to encrypt communications between machines. It was previously called Secure Sockets Layer (SSL) and is sometimes still referred to as SSL.
- TLS works on top of TCP and is used every day to protect connections between browsers, web servers, mobile applications, and so on.
- To protect sessions on top of UDP, DTLS can be used instead of TLS.
- TLS, and most other transport security protocols, have a handshake phase in which the secure negotiation is created, and a post-handshake phase in which communications are encrypted using keys derived from the first phase.
- To avoid delegating too much trust to the web public key infrastructure, applications making use of TLS can use certificate and public key pinning to only allow secure communications with specific certificates or public keys.
- As a defense-in-depth measure, systems can implement certificate revocation (to remove compromised certificates) and monitoring (to detect compromised certificates or CAs).
- In order to avoid TLS' complexity and size, and if you control both sides of the connection, you can use the Noise Protocol framework.
- Wireguard is the most secure VPN solution, it is built on top of the Noise Protocol Framework.

End-to-end encryption

This chapter covers

- The importance of end-to-end encryption for companies and users.
- The different attempts at solving email encryption.
- How end-to-end encryption is changing the landscape of messaging.

Chapter 9 explained transport security via protocols like TLS and Wireguard. At the same time, I spent quite some time explaining where trust was rooted on the web: hundreds of certificate authorities trusted by your browser and operating system. While not perfect, this system has worked so far for the web, which is a complex network of participants who know nothing of each other.

This problem of finding ways to trust others (and their public keys), and making it scale, is at the center of real-world cryptography. A famous cryptographer was once heard saying "symmetric crypto is solved" to describe a field of research that had overstayed its welcome. And for the most part the statement was true. We seldom have issues encrypting communications, and we have strong confidence in the current encryption algorithms we use. Most engineering challenges when it comes to encryption are not about the algorithms themselves anymore, but about who Alice and Bob are, and how to prove it.

Cryptography does not provide one solution to trust, but many different ones that are more or less practical depending on the context. In this chapter, I will survey some of the different techniques that people and applications have used to create trust between users.

10.1 Why end-to-end encryption?

This chapter starts with a "why" instead of a "what". This is because end-to-end encryption (often abbreviated as **e2e encryption**) is a concept more than a cryptographic protocol, a concept of securing communications between two (or more) participants across an adversarial path.

I started this book with a simple example: Alice wanted to send a message to Bob without anyone in the middle being able to see it. Nowadays, many applications like email and messaging exist to connect users, and most of them seldom encrypt messages from soup to nuts.

You might ask: isn't TLS enough? In theory it could. You've learned in chapter 9 that TLS is used in many places to secure communications. But end-to-end encryption is a concept that involves **actual human beings**. In contrast, TLS is most often used by systems that are by design men-in-the-middle (see figure 10.1). In these, TLS is only used to protect the communications between a central server and its users, allowing the server to see everything. Effectively these man-in-the-middle servers that sit in-between users, and that are necessary for the application to function, are **trusted third parties** of the protocol. That is to say, we have to trust these parts of the system in order for the protocol to be considered secure (spoiler alert: that's not a great protocol).

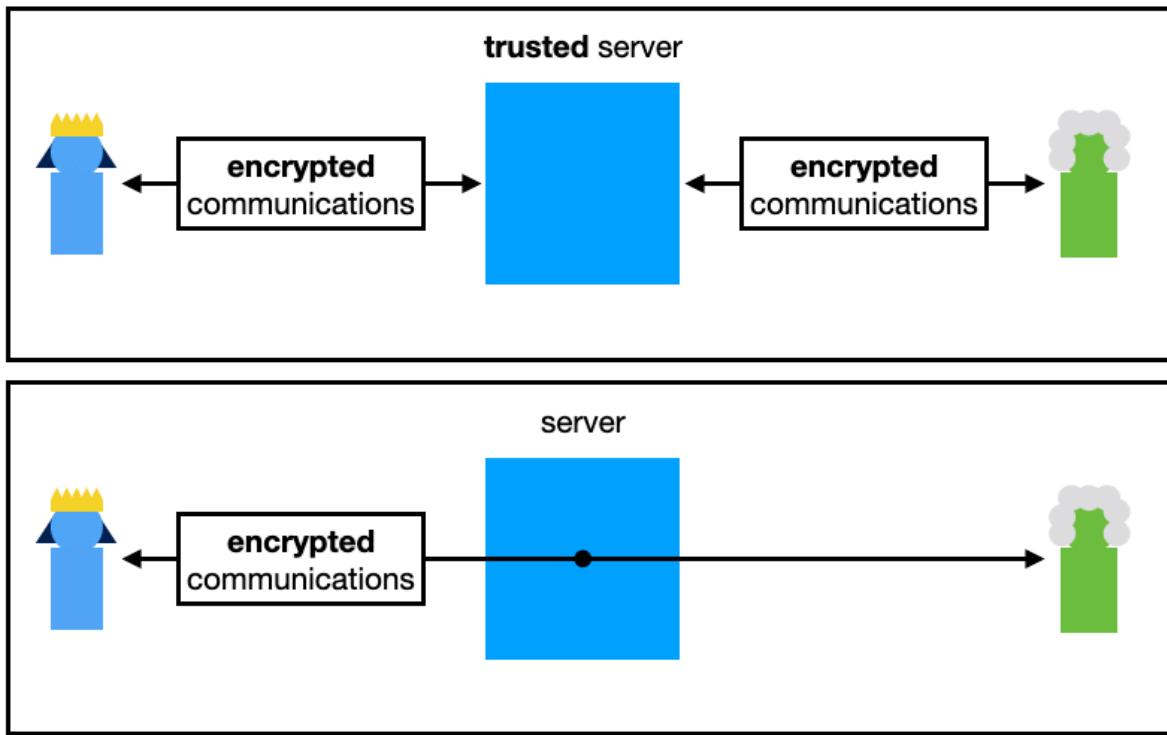


Figure 10.1 In most systems (top diagram), a central server is used to relay messages between users. A secure connection is usually established between a user and the central server (which can thus see all user messages). A protocol providing end-to-end encryption (bottom diagram) encrypts communications from one user up to its intended recipient, preventing any server in the middle from observing messages in clear.

Sometimes worse topologies exist that increase the attack surface exist. Communications between a user and a server can go through several hoops, machines often referred to as middleboxes that end the TLS connection earlier (we say that they terminate TLS) and either forward the traffic in clear to, or start another TLS connection with, the next hoop. Perhaps this is done in order to better filter traffic, or balance connections geographically for optimization purposes, or to intercept, record, and spy on traffic.

Many cases exist where such interception (deliberate or not) happened, and of course many more that we don't know about. In 2015 Lenovo was caught selling laptops with pre-installed custom certificate authorities (covered in chapter 9) and software. The software was man-in-the-middle'ing HTTPS connections (using Lenovo's certificate authorities) and injecting ads into web pages. More seriously, large countries like China and Russia have been caught redirecting traffic on the internet (making it pass through their network) in order to intercept and observe connections.⁶⁵ In 2013, Edward Snowden leaked a massive number of documents from the NSA showing the abuses of many governments (not just the US) in spying on their people's communications by intercepting the internet cables that link the world together.⁶⁶

Owning and seeing user data is also a liability for companies. As I've mentioned many times in this book, breaches and hacks happen way too often and can be devastating for the credibility of a company. From a legal standpoint, laws like the General Data Protection Regulation (GDPR) can end up costing organizations a lot of money. British Airways was fine £183.39 million after a data breach that happened in 2019. Government requests like the infamous National Security Letters (NSLs) that sometimes prevent companies and people involved from sharing them (so-called gag orders) can be seen as additional cost and stress to an organization too, that is unless you have nothing much to share.

Bottom line, if you're using a popular online application, chances are that one or more governments already have access to everything you wrote or uploaded there. Depending on an application's **threat model** (what the application wants to protect against), or the threat model of an application's most vulnerable users, end-to-end encryption plays a major role in ensuring confidentiality and privacy of end-users.

This chapter will go over different techniques and protocols that have been created in order to create trust between people. In particular, you will learn about how email encryption works today, and how secure messaging is changing the landscape of end-to-end encrypted communications.

10.2 A root of trust nowhere to be found

The simplest scenario for end-to-end encryption is the following:

Alice wants to send an encrypted file to Bob over the internet.

With all the cryptographic algorithms you've learned about in the first chapter of this book, you can probably think of a way to do this if you've read chapter 6. For example:

1. Bob sends his public key to Alice.
2. Alice encrypts the file with Bob's public key, and sends it to Bob.

Perhaps Alice and Bob can meet in real life, or use another secure channel they already share, to exchange the public key in the first message. If this is possible, we say that they have an **out-of-band** way of creating trust.

This is not always the case though. You can imagine me including my own public key in this book, and asking you to use it to send me an encrypted message at some email address. Who says my editor did not replace the public key with theirs?

Same for Alice, how does she figure out if the public key she received truly is Bob's public key? It's possible that someone in the middle could have tampered with the first message as illustrated in figure 10.2.

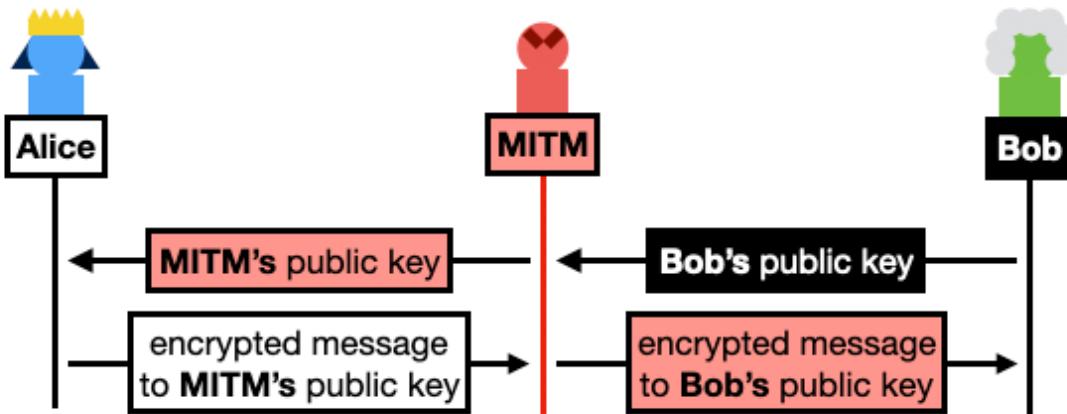


Figure 10.2 Bob sends his public key to Alice so that she can use it to encrypt her messages to him. As nobody is authenticated, a man-in-the-middle attacker can simply replace Bob's public key with their own.

A painful realization is that **there is no real solution to this trust issue**. As you will see in this chapter cryptography has no real answer to the trust issue, instead it provides different solutions to help in different scenarios.

This whole business of protecting public keys from tampering is the single most difficult problem in practical public key applications. It is the 'Achilles heel' of public key cryptography, and a lot of software complexity is tied up in solving this one problem.

– PGP User's Guide Volume I: Essential Topics

The reason why there is no true solution is that we are trying to bridge reality (real human

beings) to a theoretical cryptographic protocol.

Going back to our simple setup where Alice wants to send a file to Bob, and assuming that their untrusted connection is all they have, they have somewhat of an impossible trust issue at hand. Alice has no good way of knowing for sure what truly is Bob's public key. It's a chicken and egg type of problem.

Yet, let me point out that if no malicious **active** man-in-the-middle attacker is here to replace Bob's public key in the first message, then the protocol is safe. Even if the messages are being **passively** recorded, it is too late for an attacker to come after the fact to decrypt the second message.

Of course, relying on the fact that your chances of being actively man-in-the-middled are "not too high" is not the best way to do cryptography, but we unfortunately often do not have a way to avoid this. For example, Google Chrome ships with a set of certificate authorities that you choose to trust, but how do you obtain Chrome in the first place? Perhaps you used the default browser of your operating system, which relies on its own set of certificate authorities. But where did that come from? From the laptop you bought. But where did this laptop come from?

As you can quickly see, it's **turtles all the way down**. At some point, you will have to trust that something was done right. A threat model typically chooses to stop addressing issues after a specific turtle, and considers that any turtle further down is out-of-scope.

This is why the rest of the chapter will assume that you have a secure way to obtain some **root of trust**. All systems based on cryptography work by relying on a root of trust, something that a protocol can build security on top of. A root of trust can be a secret or a public value that we start the protocol with, or an out-of-band channel that we can use to obtain these.

10.3 The failure of encrypted email

Email was created as, and is still today, an unencrypted protocol. We can only blame a time where security was second thought. Email encryption started to become more than just an idea after the release of a tool called **Pretty Good Privacy (PGP)** in 1991. At the time, the creator of PGP Phil Zimmermann decided to release PGP in reaction to a bill that almost came to law earlier in the same year. The bill would have allowed the US government to obtain all voice and text communications from any electronic communication company and manufacturer. The protocol was finally standardized in RFC 2440 as **OpenPGP** in 1998, and caught traction with the release of the open source implementation GNU Privacy Guard (GPG) around the same time.⁶⁷ Today, GPG is still the main implementation and people interchangeably use the terms GPG and PGP to pretty much mean the same thing.

10.3.1 PGP or GPG? And how does it work?

PGP, or OpenPGP, works by simply making use of hybrid encryption (covered in chapter 6). The details are in RFC 4880, the last version of OpenPGP, and can be simplified to these steps:

1. The sender creates an email. At this point the email's content is usually compressed before it is encrypted (do you know why it is not compressed after encryption?)
2. The OpenPGP implementation generates a random symmetric key and symmetrically encrypts the email using the symmetric key.
3. The symmetric key is asymmetrically encrypted to all the recipients' public keys (using techniques you've learned in chapter 6).
4. The email body is replaced with the encrypted version of the message and sent to all recipients along with their respective encrypted symmetric key.
5. To decrypt an email, a recipient uses their private key to decrypt the symmetric key, then decrypts the content of the email using the decrypted symmetric key.

Note that OpenPGP also defines how an email can be signed (in order to authenticate the sender). To do this, the plaintext email's body is hashed and then signed using the sender's private key. The signature is then added to the message before being encrypted in step 2. Finally, so that the recipient can figure out what public key to use to verify the signature, the sender's public key is sent along the encrypted email in step 4.

I illustrate the PGP flow in figure 10.3.

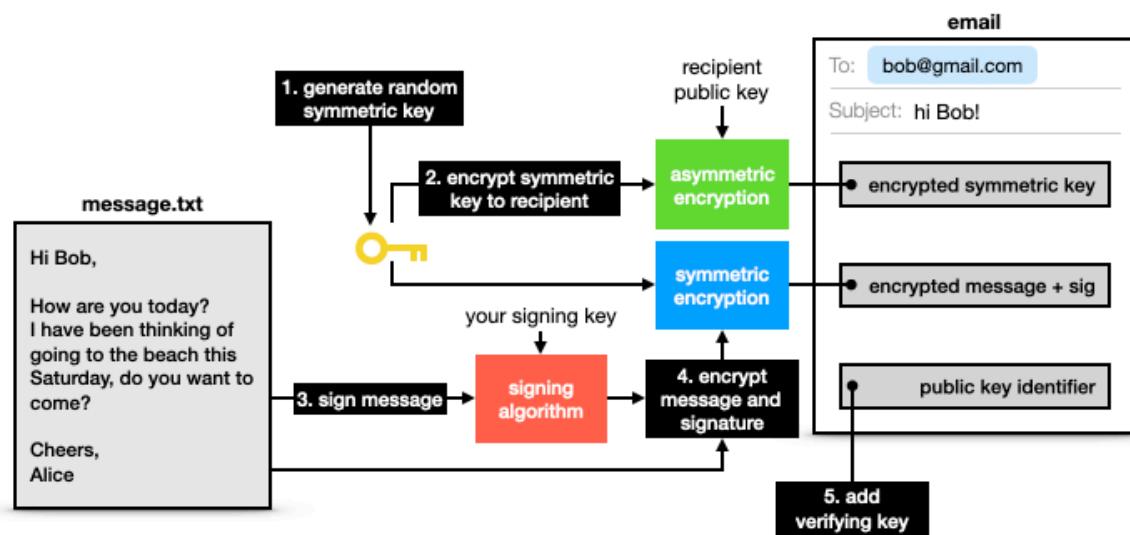


Figure 10.3 PGP's goal is to encrypt and sign messages. When integrated to email clients it does not care about hiding the subject or other metadata.

There's nothing inherently wrong with this design at first sight. It seems to prevent man-in-the-middle attackers from seeing your emails's content (although the subject and other email headers are not encrypted).

NOTE

It is important to note that cryptography can only go so far to hide metadata. In privacy-conscious applications, metadata is a big problem and can in the worst cases de-anonymize you! For example, in end-to-end encrypted protocols you might not be able to decrypt messages between users, but you can probably tell what their IP addresses are, and what is the length of the messages they send and receive, and who they commonly talk to (their social graphs), and so on. A lot of engineering is put into hiding this type of metadata.

Yet, in the details PGP is actually quite bad.

The OpenPGP standard and its main implementation GPG make use of old algorithms and ways. The most critical issue is that encryption is not authenticated, which means that anyone intercepting an email that hasn't been signed might be able to tamper with the encrypted content to some degree depending on the exact encryption algorithm used. For this reason alone I would not recommend anyone to use PGP today.

A surprising flaw of PGP comes from the fact that the signing and encryption operations are composed without care. In 2001 Don Davis pointed out that because of this naive composition of cryptographic algorithms, one can re-encrypt a signed email they received to another recipient. Effectively allowing Bob to send you the email Alice sent him, like you were the intended recipient.

If you're wondering, signing the ciphertext instead of the plaintext is still flawed, as one could then simply remove the signature that comes with the ciphertext and add their own signature instead. In effect, Bob could pretend that they sent you an email that was actually coming from Alice. I recapitulate these two signing issues in figure 10.4. By the way, can you think of a unambiguous way of signing a message?

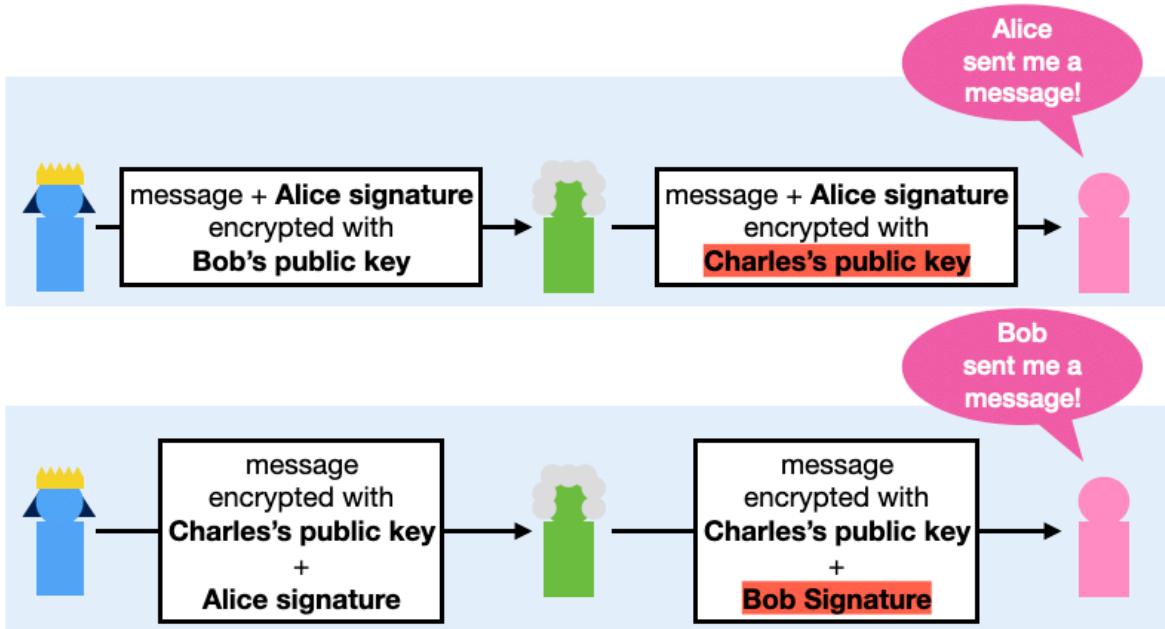


Figure 10.4 In the top diagram, Alice encrypts a message and signature (over the message) with Bob's public key. Bob can re-encrypt this message to Charlie who might believe that it was intended for him. This is the PGP flow. In the bottom diagram, this time Alice encrypts a message to Charles. She also signs the encrypted message instead of the plaintext content. Bob who intercepts the encrypted message can replace the signature with its own, fooling Charlie into thinking that he wrote the content of the message.

The cherry on the cake is that the algorithm does not provide forward secrecy by default. As a reminder: without forward secrecy, a compromise of your private key implies that all previous emails sent to you encrypted under that key can now be decrypted. You can still force forward secrecy by changing your PGP key, but this process is not straightforward (you can for example sign your new key with your older key) and most users just don't bother.

So to recap:

- PGP uses old cryptographic algorithms.
- PGP does not have authenticated encryption, and is thus not secure if used without signatures.
- Due to bad design, receiving a signed message doesn't necessarily mean we were the intended recipient.
- There is no forward secrecy by default.

10.3.2 Scaling trust between users with the web of trust

So why am I really talking about PGP here? Well, there is something interesting about PGP that I haven't talked about yet: how do you obtain other people's public keys?

The answer is that in PGP, **you build trust yourself**.

OK what does this mean? Imagine that you install GPG today and decide that you want to encrypt some messages to your friends. To start, you must first find a secure way to obtain your friends' PGP public keys. Meeting them in real life is one sure way to do it. So you meet, you copy their public keys on a piece of paper, and then you type those keys back into your laptop at home. Now you can send your friends signed and encrypted messages with OpenPGP.

But this is tedious. Do you have to do this for every person you want to email? Of course not.

Let's take the following scenario:

- You have obtained Bob's public key in real life and thus you trust it.
- You do not have Mark's public key, but Bob does and trusts it.

Take a moment here to think about what you could be doing to trust Mark's public key.

Bob can simply sign Mark's key, showing you that he trusts the association between the public key and Mark's email. If you trust Bob, you can now trust Mark's public key and add it to your repertoire.

And this is all there is to the concept of decentralized trust behind PGP. It is called **the web of trust (WoT)**.

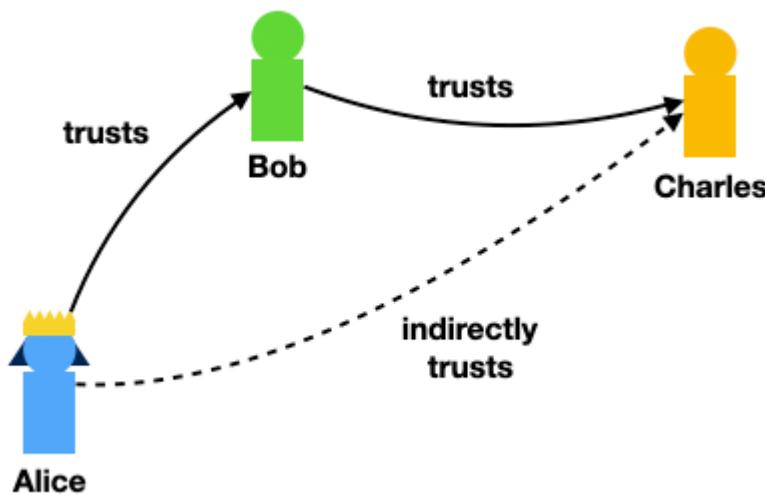


Figure 10.5 The web of trust is the concept that users can transitively trust other users by relying on signatures. In this diagram we can see that Alice trusts Bob who trusts Charles. Alice can use Bob's signature over Charles identity and public key to trust Charles as well.

You will sometimes see "key parties" at conferences, where people meet in real life and sign their respective public keys. But most of that is role-playing, and in practice very few people rely on the web of trust to enlarge their PGP circle.

10.3.3 Key discovery is a real issue

PGP did try other ways to solve the issue of discovering public keys: key registries. The concept is pretty simple, publish your PGP public key (and associated signatures from others that attest your identity) on some public list so that people can find it. In practice this doesn't work as anyone can publish a signature under your email.

In some settings, we can relax our threat model and allow for a trusted authority to attest identities and public keys. Think of a company managing their employees' emails for example.

In 1995, the RSA company proposed **Secure/Multipurpose Internet Mail Extensions (S/MIME)** as an extension to the MIME format (which itself is an extension to the email standard), and as an alternative to PGP. S/MIME, standardized in RFC 5751, took an interesting departure from the web of trust by using a public key infrastructure to build trust. That is pretty much the only conceptual difference that S/MIME has with PGP.

As companies have processes in place to onboard and offboard employees, it makes sense for them to start using protocols like S/MIME in order to bootstrap trust in their internal email ecosystem.

It is important to note that both PGP and S/MIME are usually used over the **Simple Mail Transfer Protocol (SMTP)** which is the protocol used today for emails. These protocols were also invented later, and for this reason their integration with SMTP and email clients is far from perfect. For example, only the body of an email is encrypted, not the subject or any of the other email headers.

S/MIME, like PGP, is also quite an old protocol that uses outdated cryptography and practices. Like PGP, it does not offer authenticated encryption.

Recent research on integration of both protocols in email clients showed that most of them were vulnerable to exfiltration attacks where an attacker who can observe encrypted emails can retrieve their content by sending tampered versions to the recipients.⁶⁸

Furthermore, this does not even matter as today most email is still sent unencrypted. PGP has proven to be quite hard to use for normal as well as advanced users who need to understand the many subtleties and processes of PGP's key management. And I'm not even talking about poor integration with email clients, where users can still respond to an encrypted email without encryption, potentially quoting the whole thread in cleartext.

In the 1990s, I was excited about the future, and I dreamed of a world where everyone would install GPG. Now I'm still excited about the future, but I dream of a world where I can uninstall it.

— Moxie Marlinspike *GPG And Me* (2015)
<https://moxie.org/blog/gpg-and-me/>

For these reasons, PGP has slowly been losing support (for example Golang removed support for PGP from its standard library in 2019), while more and more real-world cryptography applications are aiming at replacing PGP and solving its usability problems.

Today, it is hard to argue that email encryption will ever be a thing.

If messages can be sent in plaintext, they will be sent in plaintext.

Email is end-to-end unencrypted by default. The foundations of electronic mail are plaintext. All mainstream email software expects plaintext. In meaningful ways, the Internet email system is simply designed not to be encrypted.

– Thomas Ptakcek - *Stop Using Encrypted Email* (2020)
<https://latacora.micro.blog/2020/02/19/stop-using-encrypted.html>

10.3.4 If not PGP, then what?

I spent some pages to talk about how a simple design like PGP can fail in a lot of different and surprising ways in practice.

I would recommend against using PGP.

While email encryption is still an unsolved problem, alternatives are being developed to replace different usecases of PGP.

Saltpack is a similar protocol and message format to PGP. It attempts to fix some of the PGP flaws I've talked about.⁶⁹ Its main implementations, at the moment of this writing, are keybase (keybase.io/) and keys.pub (keys.pub/). (See figure 10.6 for an illustration of the keys.pub tool.)

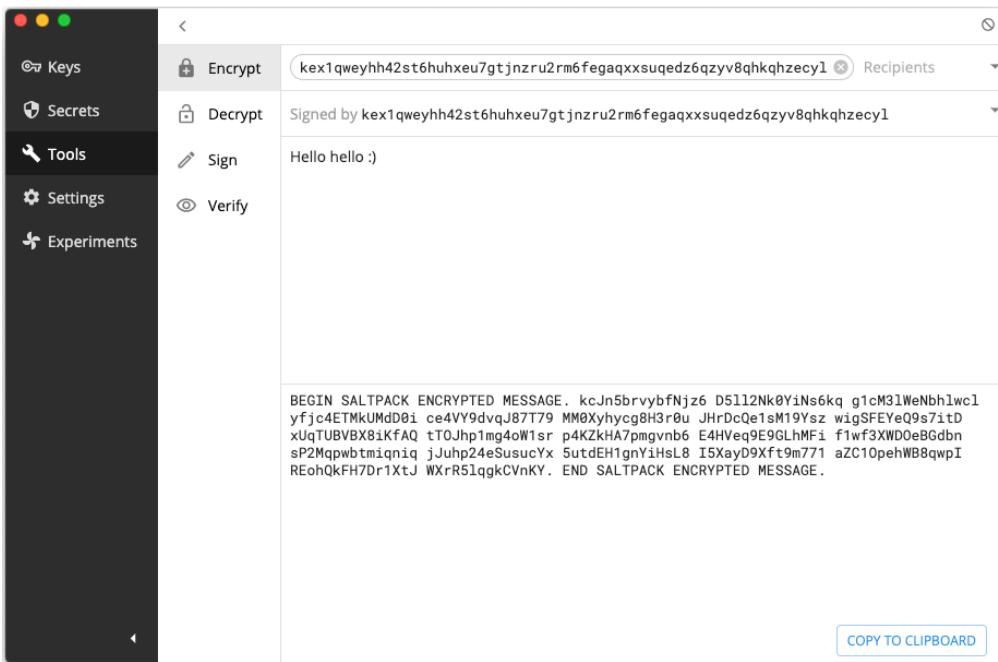


Figure 10.6 `keys.pub` is a native desktop application implementing the saltpack protocol. You can use it to import other people's public keys, and encrypt and sign messages to them.

These implementations have all moved away from the web of trust, and allow users to broadcast their public keys on different social networks in order to instill their identity into their public keys (as illustrated in figure 10.7). PGP could obviously not have thought about this key discovery mechanism, as it predates the boom of social networks.

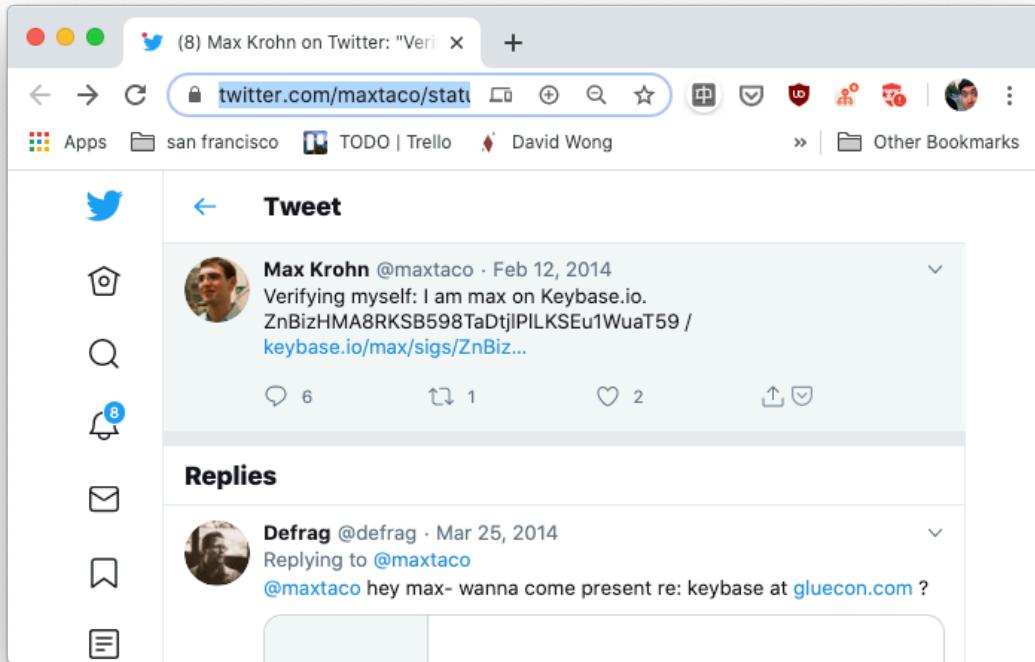


Figure 10.7 A keybase user broadcasting their public key on the Twitter social network. Allowing other users to obtain additional proof that his identity is linked to a specific public key.

On the other hand, most secure communication nowadays is far from a one-time message, and the use of these tools is less and less relevant. In the next section I talk about **secure messaging**, one of the field that aims at replacing the communication aspect of PGP.

10.4 Secure messaging, a modern look at end-to-end encryption with Signal

In 2004, **Off-The-Record (OTR)** was introduced in white paper entitled "Off-the-Record Communication, or, Why Not To Use PGP". Unlike PGP or S/MIME, OTR is not used to encrypt emails but instead chat messages, specifically it extends a chat protocol called the Extensible Messaging and Presence Protocol (XMPP).

One of the particularity of OTR was **deniability**. A claim that recipients of your messages (and passive observers) cannot use messages you sent them in a court of justice. Since messages you send are authenticated and encrypted symmetrically with a key your recipient shares with you, they could have easily forged these messages themselves. By contrast, in PGP messages are signed and are thus the inverse of deniable: they are **non-repudiable**. To my knowledge, none of these properties have actually been tested in court.

In 2010 the Signal mobile phone application (then called TextSecure) was released, making use

of a newly created protocol called the **Signal protocol**. The Signal mobile application was a large departure from PGP, S/MIME, OTR and other protocols based on **federation** (no central entity is required for the network to run) by taking a centralized approach like most commercial products do. Yet Signal is a nonprofit organization that runs solely on grants and donations.

While Signal prevents interoperability with other servers, the Signal protocol is an open standard and has been adopted by many other messaging applications, including Google Allo (now defunct), WhatsApp, Facebook Messenger, Skype, and many others. The Signal protocol is truly a success story, transparently being used by billions of people including journalists, targets of government surveillance, and your average joe.

It is interesting to look at how Signal works, because it attempts to fix many of the flaws that I've previously mentioned with PGP. In this section, I will go over each one of these interesting features of Signal:

1. How can we do better than the web of trust? Is there a way to upgrade the existing social graphs with end-to-end encryption? The answer of Signal is to use a **trust on first use (TOFU)** approach. TOFU allows users to blindly trust other users the first time they communicate, relying on this first insecure exchange to establish a long-lasting secure communication channel. Users are then free to check if the first exchange was man-in-the-middle'ed by matching their session secret out-of-band.
2. How can we upgrade PGP to obtain forward secrecy every time we start a conversation with someone? The first part of the Signal protocol is like most secure transport protocols: it's a key exchange, but a particular one called **Extended Triple Diffie-Hellman (X3DH)**. More on that later.
3. How can we upgrade PGP to obtain forward secrecy for every single message. This is important as conversations between users can span years and a compromise at some point in time should not reveal years of communication. Signal addresses this with something called a **symmetric ratchet**.
4. What if two users' session secrets are compromised at some point in time? Is that game over? Can we also recover from that? Signal introduces a new security property called **post-compromise security** and addresses it with what they call a **Diffie-Hellman ratchet**.

Let's get started!

10.4.1 Trust but verify

One of email encryption's biggest failure was its reliance on PGP and the web of trust model to transform social graphs into secure social graphs. It is really rare today to see people signing each other PGP keys.

The way most people use applications like PGP, OTR, Signal, etc. is to blindly trust a key the first time they see it, and to reject any future changes (as illustrated in figure 10.8). This way only the very first connection can be attacked, and this only by an active man-in-the-middle attacker.

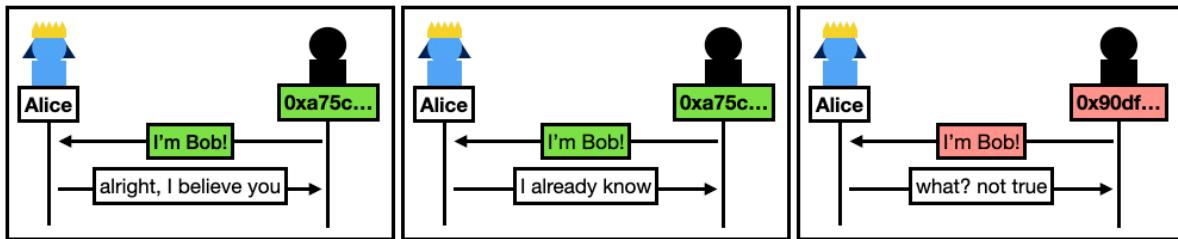


Figure 10.8 Trust On First Use (TOFU) allows Alice to trust her very first connection, but not subsequent connections if they don't exhibit the same public key. TOFU is an easy mechanism to build trust when the chances that the very first connection is actively man-in-the-middle are low. The association between a public key and the identity (here Bob) can also be verified after the fact out-of-band.

While TOFU is not the best security model, it is often the best we have, and has proven extremely useful. The Secure Shell (SSH) protocol, for example, is often used by trusting the server's public key during the first connection (see figure 10.9) and by rejecting any future change.

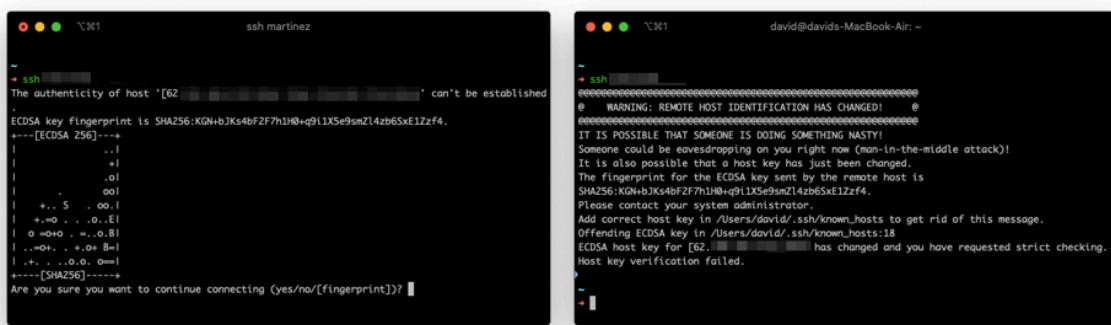


Figure 10.9 SSH clients use trust on first use. the first time you connect to an SSH server (see left picture) you have the option to blindly trust the association between the SSH server and the public key displayed. If the public key of the SSH server later changes (see right picture) your SSH client will prevent you from connecting to it.

While TOFU systems trust the first key they see, they still allow the user to later verify that the key is indeed the right one and catch any impersonation attempts. In real-world applications users typically compare **fingerprints**, which are most often hexadecimal representations of public keys. Sometimes hashes of the public keys are used to prevent disclosing what the public key is. This verification is of course done out-of-band. (If the SSH connection was compromised, then the verification would be compromised as well.)

NOTE

Of course, if users do not verify fingerprints then they can be man-in-the-middle'ed without knowing it. But that is the kind of trade-off that real-world applications have to deal with if they want to bring end-to-end encryption at scale. Indeed, the failure of the web of trust shows that security-focused applications must keep usability in mind to get widely adopted.

In the Signal mobile application, a fingerprint between Alice and Bob is computed by:

- hashing Alice's identity key prepended to her username (a phone number in Signal)
- hashing Bob's identity key prepended to his username
- truncating both results and interpret them as a series of 6 numbers of 5 digits
- displaying the concatenation of the two series of 6 numbers to the user

I recapitulate this flow in figure 10.10.

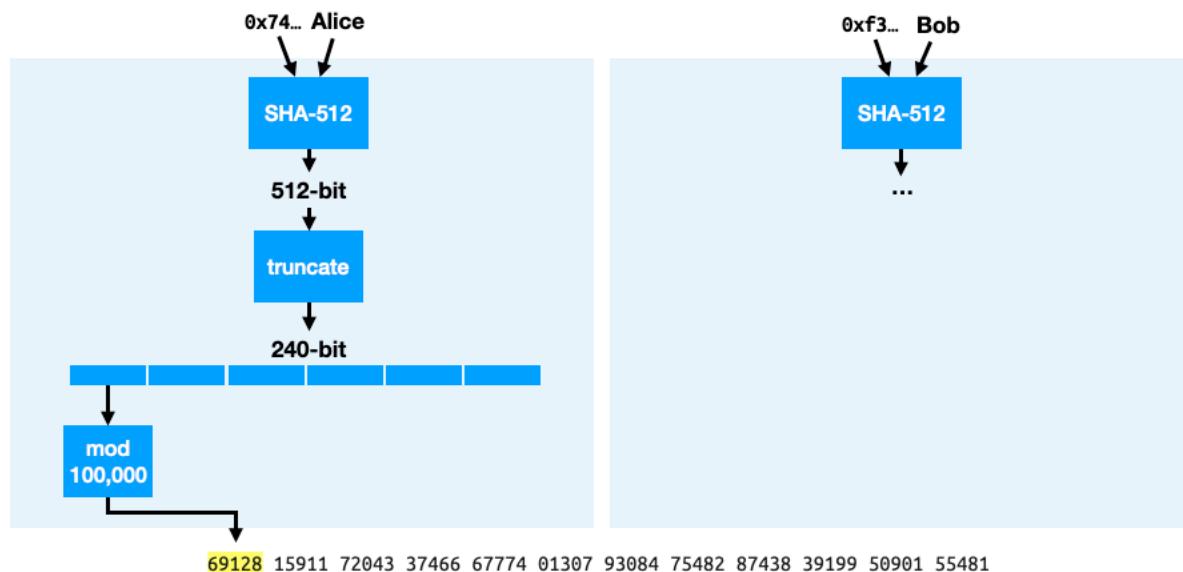


Figure 10.10 To compute a relationship's fingerprint, Signal uses SHA-512 to hash the identity key followed by the username of a peer, then truncate the result and interpret it as six 5-byte numbers taken modulo 100,000.

Applications like Signal make use of QR codes (see figure 10.11) to let users easily verify fingerprints more easily (as they can be pretty long).

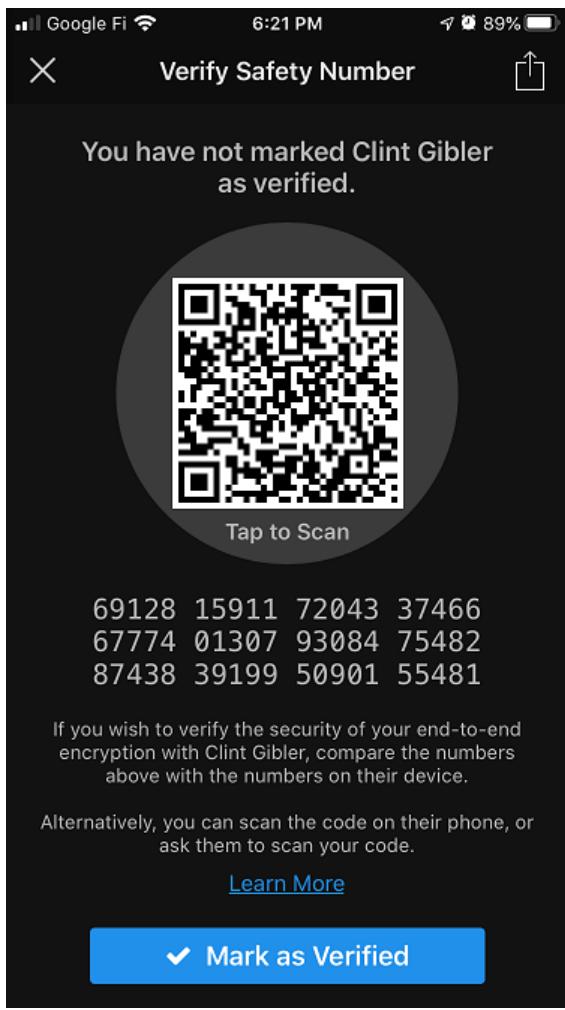


Figure 10.11 In Signal, you can verify the authenticity and confidentiality of the connection you have with a friend by using a different channel (like the real life) to make sure the two fingerprints (called "safety numbers" in Signal) you and your friend have match. This can be done more easily via the use of QR code that encodes this information in a scannable format. Signal also hashes the session secret instead of the two users' public keys, allowing them to verify one large string instead of two.

Next, let's see how the Signal protocol works under the hood.

10.4.2 X3DH, the Signal protocol's handshake

Most secure messaging apps before Signal were **synchronous**. This meant that e.g. Alice wasn't able to start (or continue) an end-to-end encrypted conversation with Bob if Bob was not online. The Signal protocol on the other hand is **asynchronous**, like email, meaning that Alice can start (and continue) a conversation with people that are offline. Remember that **forward secrecy** (covered in chapter 9) means that a compromise of keys does not compromise previous sessions, and that forward secrecy usually means that the key exchanges are interactive (as both sides have to generate ephemeral Diffie-Hellman keypairs). In this section you will see how Signal uses non-interactive key exchanges (i.e. where one side is potentially offline) that are still forward secure.

OK let's get going.

To start a conversation with Bob, Alice initiates a key exchange with him. Signal's key exchange is named **Extended Triple Diffie-Hellman (X3DH)** as it combines three (or more) Diffie-Hellman key exchanges into one. But before you learn how it works, you need to understand the three different types of Diffie-Hellman keys Signal uses:

- **Identity keys.** These are the long-term keys that represent the users. You can imagine that if Signal only used identity keys, then the scheme would be pretty similar to PGP and there would be no forward secrecy.
- **One-time prekeys.** In order to add forward secrecy to the key exchange, even when the recipient of a new conversation is not online, Signal has users upload multiple single-use public keys. These are simply ephemeral keys that are uploaded in advance, and are deleted after being used.
- **Signed prekeys.** We could stop here, but there's one edge case missing. Since the one-time prekeys that a user has uploaded can at some point be depleted, users also have to upload a medium-term public key that they sign (hence the name) and that they rotate periodically (e.g. every week). This way, if no more ephemeral public keys (one-time prekeys) are available on the server under your username, one can still use your medium-term public key (signed prekey) to add forward secrecy up to the last time you changed your signed prekey.

This is enough to preview what the flow of a conversation creation in Signal looks like in figure 10.12.

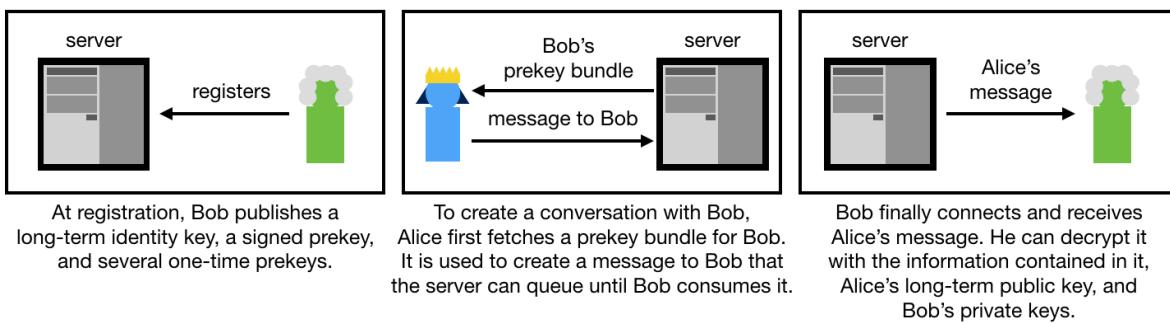


Figure 10.12 Signal's flow starts with a user registering with a number of public keys. If Alice wants to talk to user Bob, she first retrieves some public key information about the user (called a bundle of keys in the diagram), then performs an X3DH key exchange and creates an initial message using the output of the key exchange. Bob can perform the same on his side, after receipt of the message, to initialize the conversation.

Let's go over each of these steps in more depth.

First, a user registers by sending:

- one identity key
- one signed prekey and its signature
- a defined number of one-time prekeys

At this point, it is the responsibility of the user to periodically rotate the signed prekey and upload new one-time prekeys.

NOTE

Signal makes use of the identity key to perform signatures (over signed prekeys) and key exchanges (during the X3DH key exchange). While I've warned against using the same key for different purposes, Signal has deliberately analyzed that in their case there should be no issue. This does not mean that this would work in your case, and with your key exchange algorithm, and so I would still advise against doing this.

I recapitulate this flow in figure 10.13.

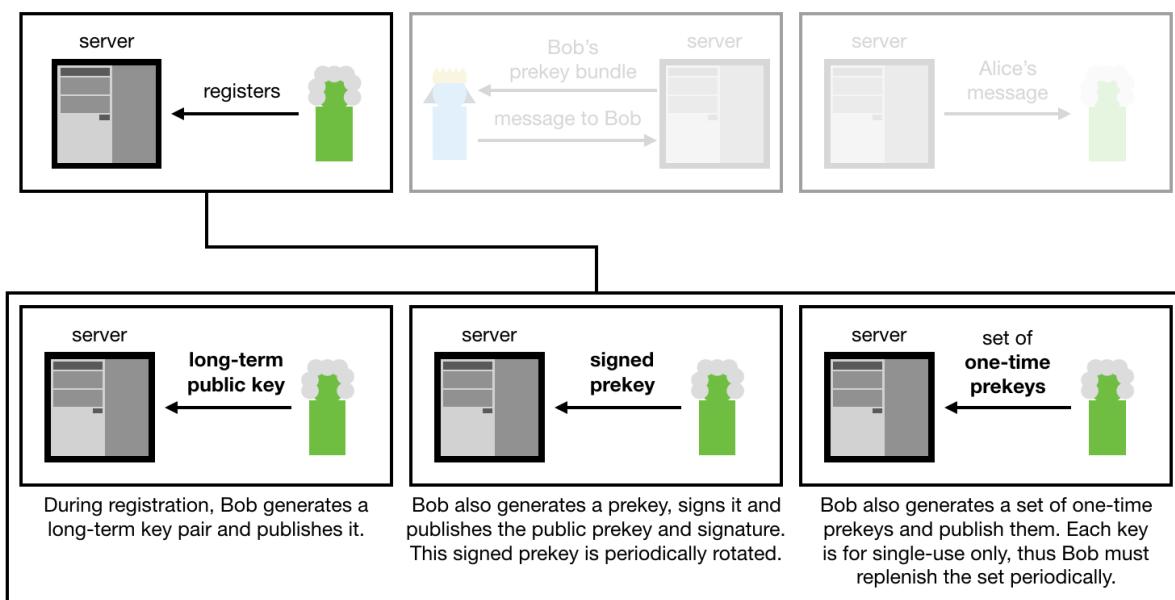


Figure 10.13 Building on figure . The first step is for a user to register by generating a number of Diffie-Hellman key pairs and sending the public parts to a central server.

Second, Alice starts a conversation with Bob by retrieving:

- Bob's identity key
- Bob's current signed prekey and its associated signature
- if there are still some: one of Bob's one-time prekeys (the server then deletes the one-time prekey sent to Alice)

Alice can then verify that the signature over the signed prekey is correct, and perform the X3DH handshake with:

- all of these public keys from Bob
- an ephemeral keypair that she generates for the occasion, in order to add forward secrecy
- her own identity key

The output of X3DH is then used in a post-X3DH protocol to encrypt her messages to Bob (more

on that in the next section).

X3DH is composed of three (optionally four) Diffie-Hellman key exchanges grouped in one. They are Diffie-Hellman key exchanges between:

1. the identity key of Alice and the signed prekey of Bob
2. the ephemeral key of Alice and the identity key of Bob
3. the ephemeral key of Alice and the signed prekey of Bob
4. if Bob still has a one-time prekey available, his one-time prekey and the ephemeral key of Alice

The output of X3DH is the concatenation of all of these Diffie-Hellman key exchanges, passed to a key derivation function (covered in chapter 8).

Different key exchanges provide different properties. 1 and 2 are here for mutual authentication, while 3 and 4 are here for forward secrecy. All of this is analyzed in more depth in the X3DH specification,⁷⁰ which I encourage you to read if you want to know more (as it is very well written).

I recapitulate this flow in figure 10.14.

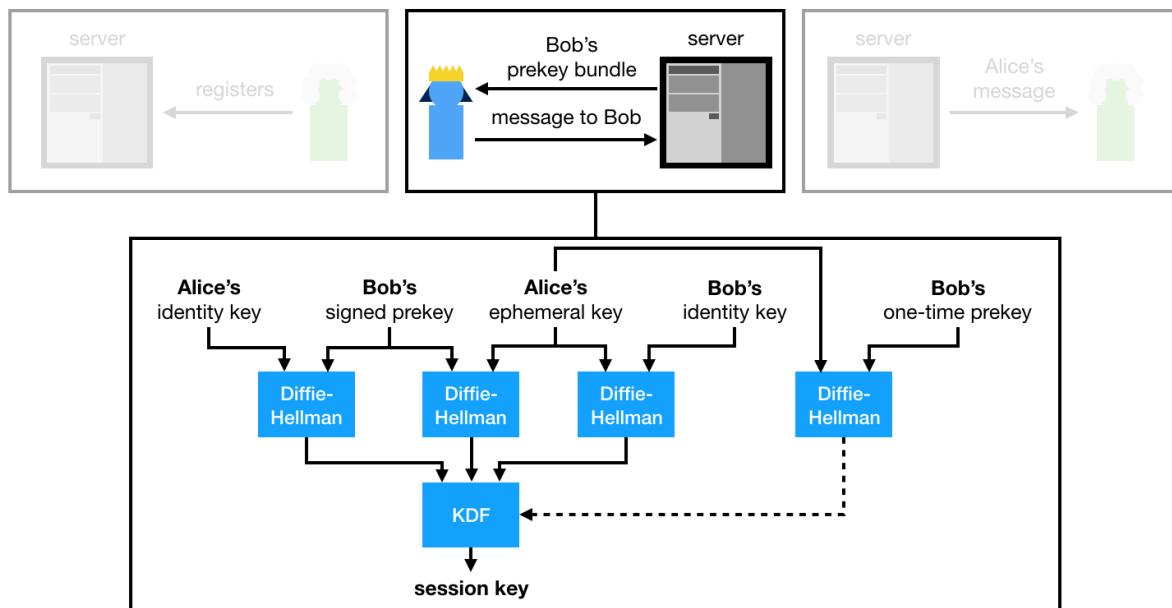


Figure 10.14 Building on figure . To send a message to Bob, Alice fetches a prekey bundle containing Bob's long-term key, Bob's signed prekey and optionally one of Bob's one-time prekey. After performing different key exchanges with the different keys, all outputs are concatenated and passed into a KDF to produce an output used in a subsequent post-X3DH protocol to encrypt messages to Bob.

Alice can then send to Bob her identity public key, the ephemeral public key she generated to start the conversation, and other relevant information (like which one-time prekey from Bob she used).

Bob receives the message and can perform the same X3DH key exchange with the public keys contained in it. Bob also gets rid of the one-time prekey of his that Alice decided to use in X3DH (if there was one).

I recapitulate this flow in figure 10.15.

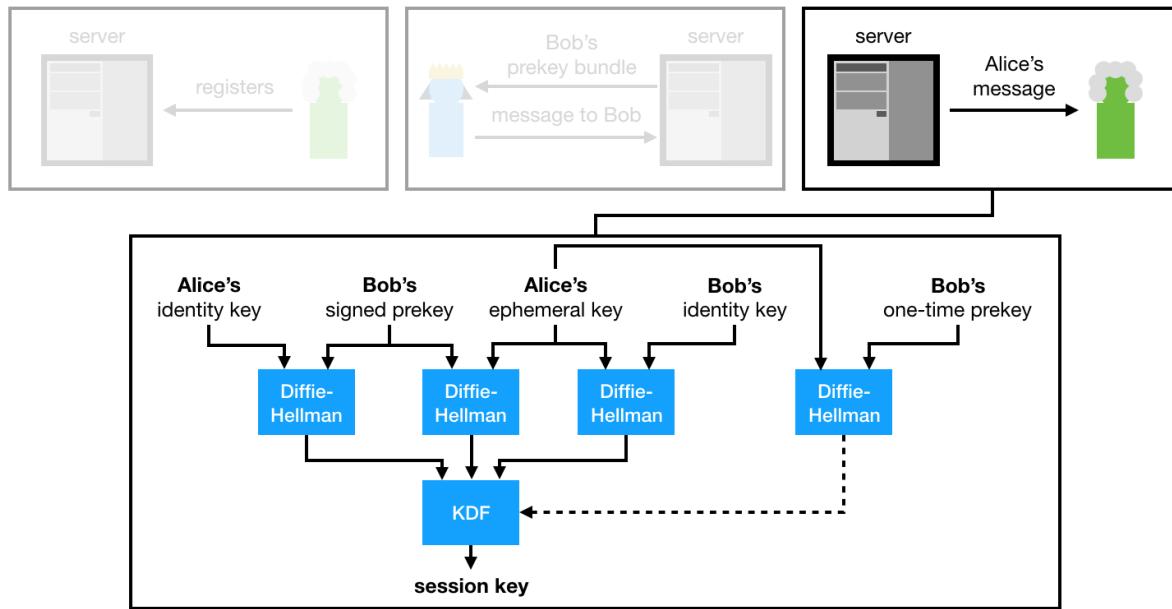


Figure 10.15 Building on figure . After receiving the initial message, Bob has all he needs to perform the same X3DH key exchange Alice had performed, deriving the same output used in the post-X3DH protocol.

What happens after X3DH is done? Let's see that next.

10.4.3 Double ratchet: Signal's post-handshake protocol

The post-X3DH phase lives as long as the two users do not delete their conversations or lose any of their keys. For this reason, and because Signal was designed with SMS conversations in mind (where the time between two messages might be counted in months), Signal introduces forward secrecy at the message level. In this section you will learn how this post-handshake protocol called the **Double Ratchet** works.

But imagine a simple post-X3DH protocol first. Alice and Bob could have taken the output of X3DH as a session key and use it to encrypt messages between them (as illustrated in figure 10.16).



Figure 10.16 Naively, a post-X3DH protocol could simply use the output of X3DH as a session key to encrypt messages between Alice and Bob.

We usually want to separate the keys used for different purposes though. So what we can do, is to use the output of X3DH as a seed (or root key according to the Double Ratchet specification) to a KDF in order to derive two other keys. One key can be used for Alice to encrypt messages to Bob, and the other for Bob to encrypt messages to Alice. I illustrate this in figure 10.17.

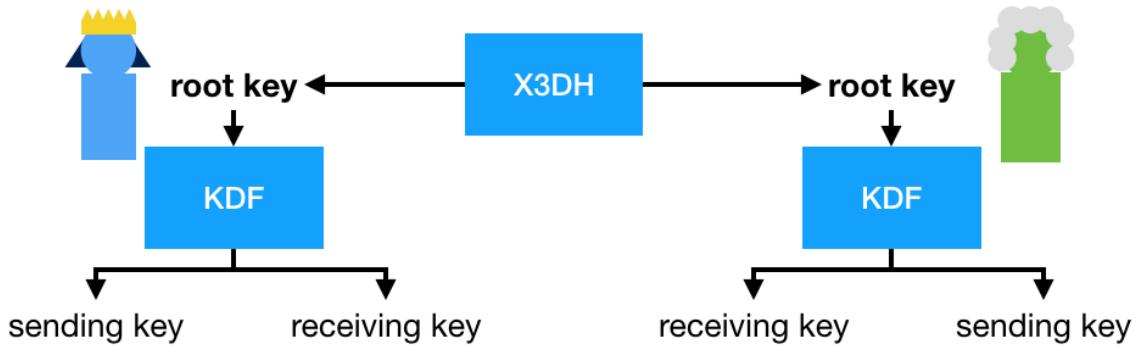


Figure 10.17 Building on figure , a better post-X3DH protocol would make use of a KDF with the output of the key exchanges to differentiate keys used to encrypt Bob’s messages and to encrypt Alice’s messages. Here Alice’s sending key is the same as Bob’s receiving key, and Bob’s seending key is the same as Alice’s receiving key.

This naive approach obviously does not provide any forward secrecy: if at one point in time this session key is stolen all previously recorded messages can be decrypted.

This post-X3DH protocol is better, but it still doesn’t have forward secrecy. To fix this, Signal introduced what is called a **symmetric ratchet** as illustrated in figure 10.18. The sending key is now renamed a sending chain key, and not used directly to encrypt messages. When sending a message, Alice continuously passes that sending chain key into a one-way function that will produce the next sending chain key as well as the actual sending keys to encrypt her messages. (Bob on the other hand will have to do the same but with the receiving chain key.) Thus, by compromising one sending key or sending chain key, an attacker cannot recover previous keys. (And the same is true when receiving messages.)

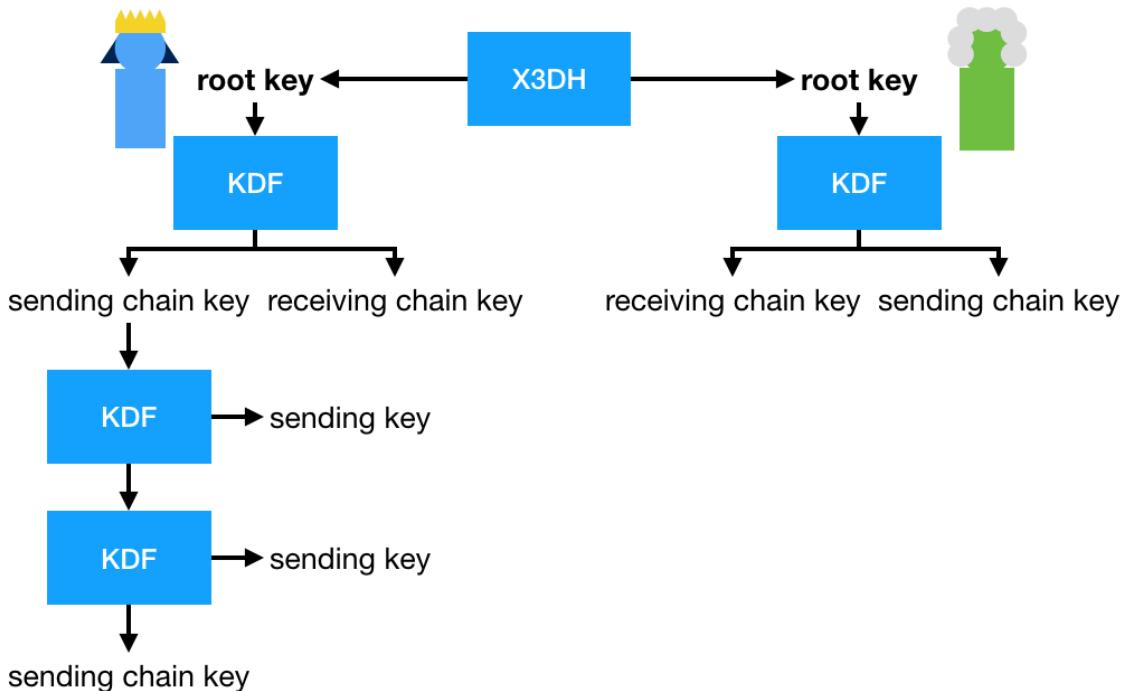


Figure 10.18 Building on figure , forward secrecy can be introduced in the post-X3DH protocol by "ratcheting" (passing into a KDF) a chain key every time one needs to send a message, and ratcheting another chain key every time one receives a message. Thus, the compromise of a sending or receiving chain key does not allow an attacker to recover previous ones.

Good. We now have forward security baked into our protocol and at the message level. Every message sent and received protects all previously sent and received messages.

Note that this is somewhat debatable as an attacker who compromises a key probably does this by compromising a user's phone, which will likely contain all previous messages in clear next to the key. Nevertheless, if both users in a conversation decide to delete previous messages (for example by using Signal's "disappearing messages" feature) the forward secrecy property is achieved.

The Signal protocol has one last interesting thing I want to talk about: **post-compromise security** (also called backward secrecy as you've learned in chapter 8). Post-compromise security is the idea that if your keys get compromised at some point, you can still manage to recover from this as the protocol goes on and heals itself. Of course if the attacker still has access to your device after a compromise, then this is for nothing. Post-compromised security can only work by re-introducing new entropy that a non-persistent compromise wouldn't have access to. The new entropy has to be the same for both peers. Signal's way of finding such entropy is by doing an ephemeral key exchange.

To do this, the Signal protocol continuously performs key exchanges in what is called a **Diffie-Hellman ratchet**. Every message sent by the protocol comes with the current "ratchet

public key" as illustrated in figure 10.19.

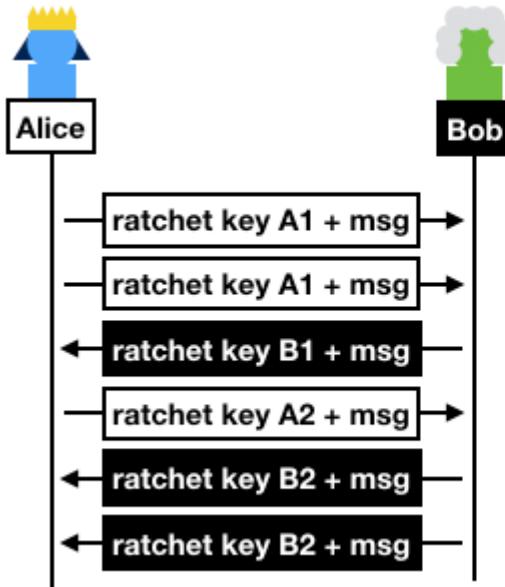


Figure 10.19 The Diffie-Hellman ratchet works by advertising a "ratchet public key" in every message sent. This ratchet public key can be the same as the previous one, or can advertise a new ratchet public key if a participant decides to refresh his or hers.

When Bob notices a new ratchet key from Alice, he must perform a new Diffie-Hellman key exchange with Alice's new ratchet key and Bob's own ratchet key. The output can then be used with the symmetric ratchet to decrypt the messages received. I illustrate this in figure 10.20.

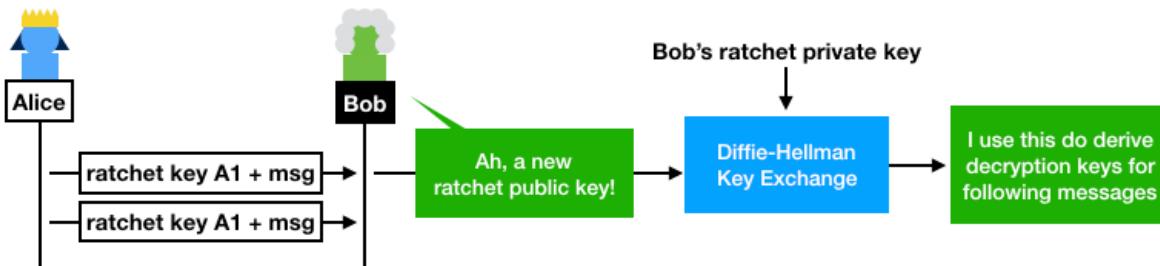


Figure 10.20 When receiving a new ratchet public key from Alice, Bob must do a key exchange with it and his own ratchet key to derive decryption keys (this is done with the symmetric ratchet). Alice's messages can then be decrypted.

Another thing that Bob must do when receiving a new ratchet key: generate a new random ratchet key for himself. With his new ratchet key, he can perform another key exchange with Alice's new ratchet key, which he will use to encrypt messages to her. This should look like figure 10.21.

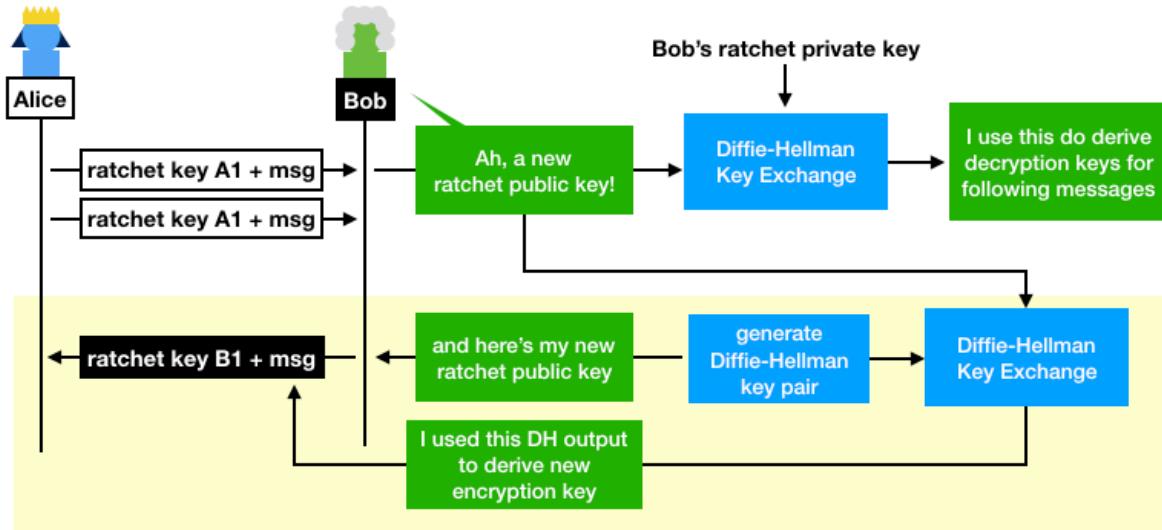


Figure 10.21 Building on figure , after receiving a new ratchet key Bob must also generate a new ratchet key for himself. This new ratchet key is used to derive encryption keys, and is advertised to Alice in his next series of messages (up until he receives a new ratchet key from Alice).

This back and forth of key exchanges is mentioned as a "ping pong" in the Double Ratchet specification:

This results in a “ping-pong” behavior as the parties take turns replacing ratchet key pairs. An eavesdropper who briefly compromises one of the parties might learn the value of a current ratchet private key, but that private key will eventually be replaced with an uncompromised one. At that point, the Diffie-Hellman calculation between ratchet key pairs will define a DH output unknown to the attacker.

– *The Double Ratchet Algorithm*
<https://signal.org/docs/specifications/doubleratchet/>

Finally, the combination of the Diffie-Hellman ratchet and the symmetric ratchet is called the **double ratchet**. It's a bit dense to visualize as one diagram, but figure 10.22 attempts to do it.

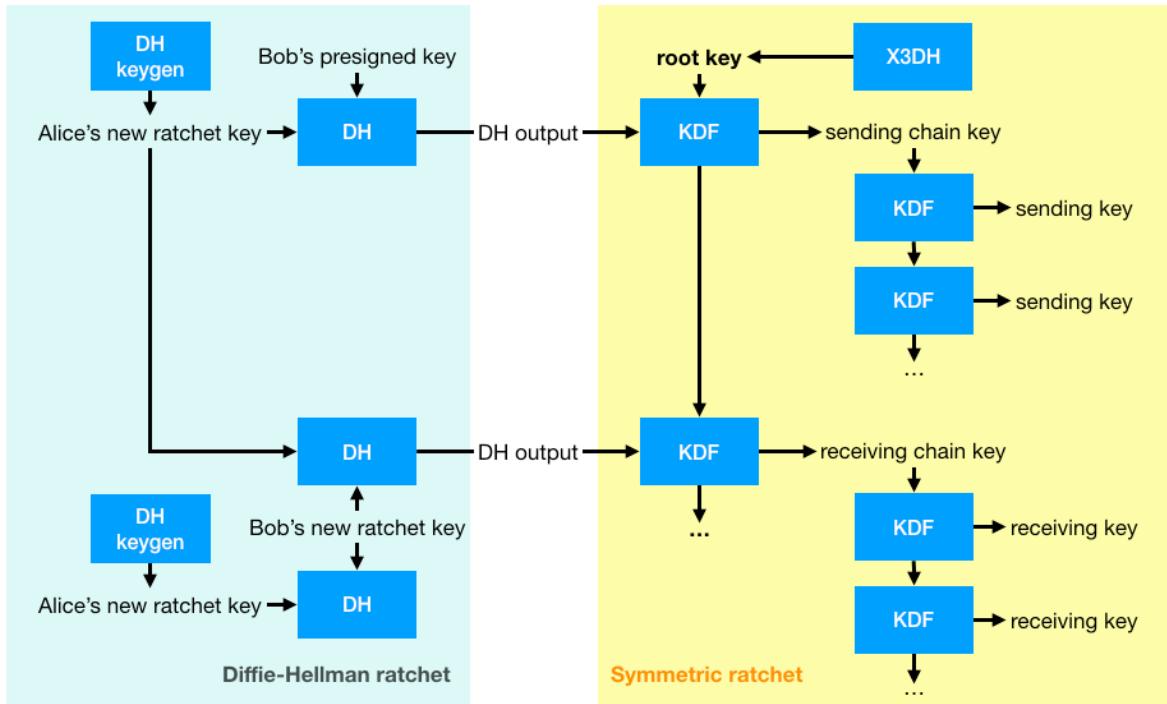


Figure 10.22 The double ratchet (from Alice’s point of view) combines the Diffie-Hellman ratchet (on the left) with the symmetric ratchet (on the right), which provides post-compromise security as well as forward secrecy to the post-X3DH protocol. In the first message, Alice does not yet know Bob’s ratchet key so she uses his presigned key instead.

I know this last diagram is quite dense, so I encourage you to take a look at Signal’s specifications, which are published on signal.org/docs/ and which provide another well-written explanation of the protocol.

10.5 The state of end-to-end encryption

Today, most secure communications between users happen through secure messaging applications instead of encrypted emails. The Signal protocol has been the clear winner in its category, being adopted by many proprietary applications but also by open source and federated protocols like XMPP (via the OMEMO extension) and Matrix (a modern alternative to IRC). On the other hand, PGP and S/MIME are being dropped as published attacks have led to a loss of trust.

What if you want to write your own? Unfortunately a lot of what’s being used in this field is ad-hoc, and you would have to fill many of the details yourself in order to obtain a full-featured and secure system.

Signal has open sourced a lot of its code, but it lacks documentation and can be hard to use correctly.⁷¹

On the other hand, you might have better luck using a decentralised and open source solution

like Matrix, which might prove easier to integrate with (and this is what the French government has done).⁷² (See figure 10.23 for an illustration of a Matrix client.)

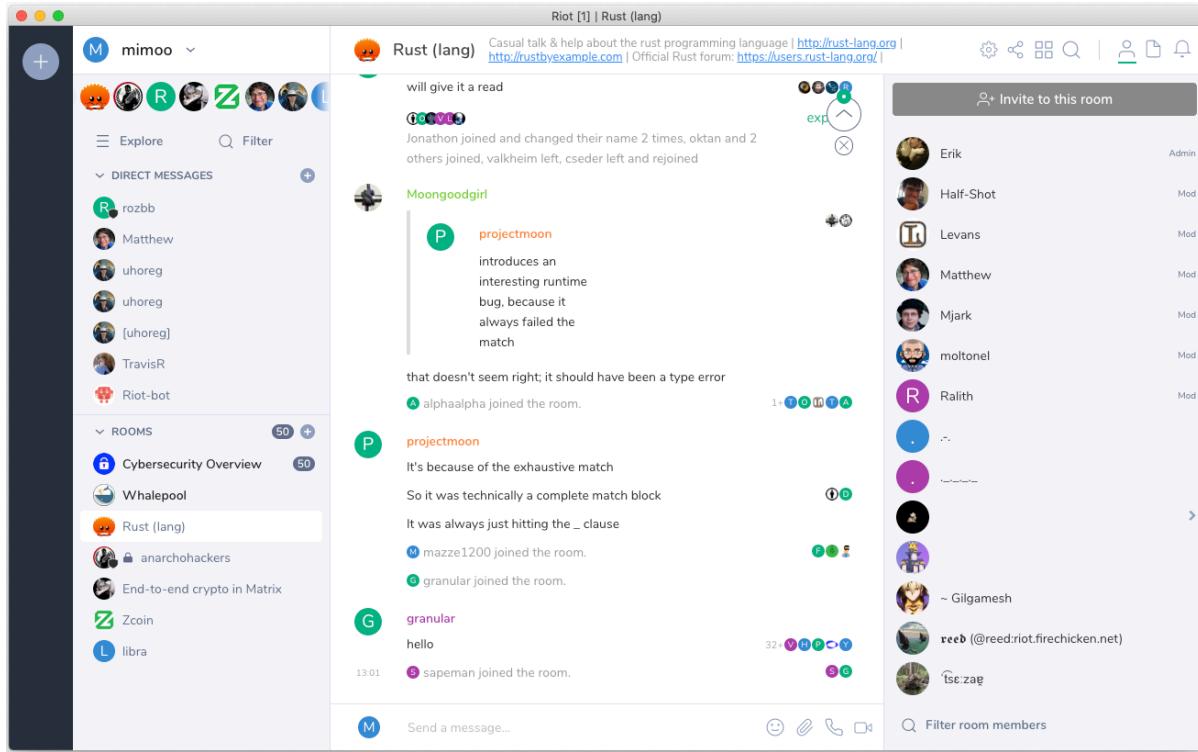


Figure 10.23 Riot, an end-to-end encrypted chat application using the Matrix network.

There are also a number of open questions and active research problems that I want to talk about:

- Group messaging.
- Support for multiple devices.
- Better security assurances than TOFU.

Let's start with the first item: **group messaging**. At this point, while implemented in different ways by different applications, group messaging is still being actively researched.

For example, the Signal application have clients make sense of group chats. Servers only see pairs of users talking, never less, never more. This means that clients have to encrypt a group chat message to all of the group chat participant, and send these individually. This is called **client-side fanout** and does not scale super well. It is also not too hard for the server to figure out what are the group members when it sees Alice sending several messages of the same length to Bob and Charles.

WhatsApp on the other hand use a variant of the Signal protocol where the server is aware of group chat membership.⁷³ This change allows a participant to send a single encrypted message to the server, which in turn will have the responsibility to forward it to the group members. This is called **server-side fanout**.

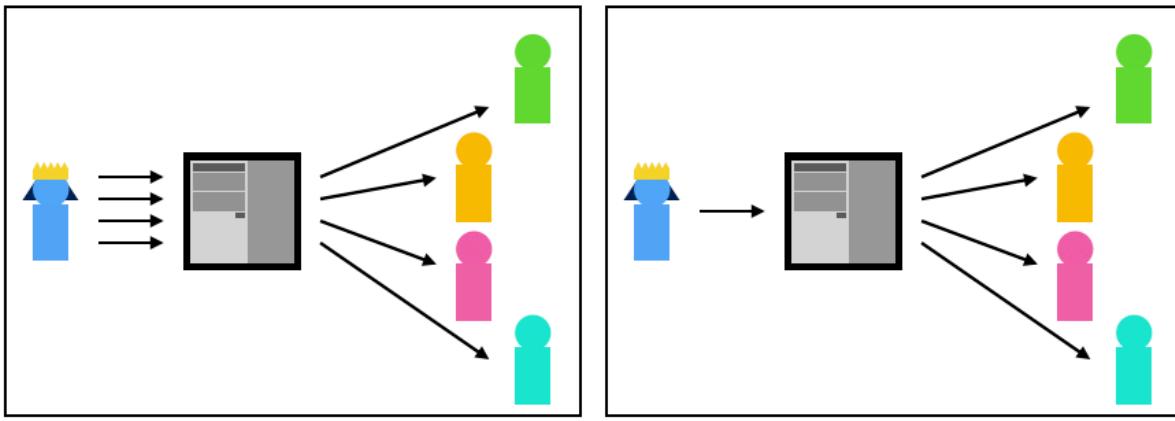


Figure 10.24 There are two ways to approach end-to-end encryption for group chats. A client-side fanout approach means that the client has to individually message each recipient using their already existing encrypted channel. This is a good approach to hide group membership from the server. A server-side fanout approach let the server forward a message to each group chat participant. This is a good approach to reduce the number of messages sent from the client perspective.

Another problem of group chat is scaling to groups of large memberset. For this, many players in the industry have recently gathered around a **Messaging Layer Security (MLS)** standard to tackle secure group messaging at scale,⁷⁴ but there seems to be a lot of work to be done (and one can wonder, is there really any confidentiality in a group chat with more than a hundred participants?).

Note that this is still an area of active research, and different approaches come with different trade-offs in security and usability. For example, no group chat protocol at the moment seems to provide transcript consistency, which ensures that all participants see the same messages in the same order.

Support for multiple devices is either not a thing, or implemented in various way, most often by pretending that your different devices are different participants of a group chat. The TOFU model can make handling multiple devices quite complicated, as having different identity keys per device can become a real key management problem. Imagine having to verify fingerprints for each of your devices, and each of your friends' devices. For example, Matrix has a user signs their own devices. Other users then can trust all your devices as one entity by verifying their associated signatures.

Finally, I've mentioned that the TOFU model is also not the greatest, as it is based on trusting a public key the first time we see it, and most users do not verify later that fingerprints match. Can something be done about this? What if the server decides to impersonate Bob to Alice only? This is a problem that **Key Transparency** is trying to tackle.⁷⁵ Key Transparency is a protocol similar to the Certificate Transparency protocol that I've talked about in chapter 9. There is also some research making use of the blockchain technology that I'll talk about in chapter 12 on cryptocurrencies.

10.6 Summary

- End-to-end encryption is about securing communications between real human beings. A protocol implementing end-to-end encryption is more resilient to vulnerabilities that can happen in servers sitting in-between users, and can greatly simplify legal requirements for companies.
- End-to-end encryption systems need a way to bootstrap trust between users. This trust can come from a public key that we already know, or an out-of-band channel that we trust.
- PGP and S/MIME are the main protocols that are used to encrypt emails today, yet none of them are considered safe to use as they make use of old cryptographic algorithms and practices. They also have poor integration with email clients that have been shown to be vulnerable to different attacks in practice.
 - PGP uses a web of trust model where users sign each other's public keys in order to allow others to trust them.
 - S/MIME uses a public key infrastructure to build trust between participants. It is most commonly used in companies and universities.
- An alternative to PGP today is saltstack, which fixes a number of issues while relying on social networks to discover other people's public keys.
- Emails will always have issues with encryption, as the protocol was built without encryption in mind. On the other hand, modern messaging protocols and applications are considered better alternatives to encrypted emails, as they are built with end-to-end encryption in mind.
 - The Signal protocol is used by most messaging applications to secure end-to-end communications between users. For example Signal messenger, WhatsApp, Facebook Messenger, and Skype all advertise their use of the Signal protocol to secure messages.
 - Other protocols, like Matrix, attempt to standardize federated protocols for end-to-end encrypted messaging. Federated protocols are open protocols that anyone can interoperate with, as opposed to centralized protocols that are limited to a single application.

User authentication

This chapter covers

- User authentication based on passwords.
- User authentication based on symmetric and asymmetric keys.
- User-aided authentication and how humans can help secure connections between devices.

In the introduction of this book, I boiled cryptography down to two concepts: confidentiality and authentication. In real-world applications, confidentiality is (usually) the least of your problems, and authentication is where most of the complexity arises. I know I've already talked a lot about authentication throughout this book, but it can be a confusing concept as it is used with different meanings in cryptography. For this reason, this chapter starts with an introduction of what authentication really is about.

As usual with cryptography, no protocol is a panacea. For this reason, the rest of the chapter will teach you about a number of authentication protocols that are being used in a multitude of real-world applications. So let's get started!

11.1 A recap on authentication

By now, you have heard of authentication many times, so let's recap. You've seen:

1. authentication in cryptographic primitives like message authentication codes (covered in chapter 3) and authenticated encryption (covered in chapter 4)
2. authentication in cryptographic protocols like TLS (covered in chapter 9) and Signal (covered in chapter 10) where one or both sides of the connection can be authenticated.

In the former case, authentication refers to the **authenticity** (or **integrity**) of messages. In the

latter case, authentication refers to **proving who you are to someone else**.

These are different concepts covered by the same word, which can be quite confusing! But both usages are correct, as the Oxford dictionary says:

Authentication. The process or action of proving or showing something to be true, genuine, or valid.

– Oxford dictionary

For this reason, you should think of authentication as a term used in cryptography to convey two different concepts depending on the context:

1. **Message/payload authentication.** You're proving that a message is genuine and hasn't been modified since its creation (e.g. "are these messages authenticated or can someone tamper with them?")
2. **Origin/entity/identity authentication.** You're proving that an entity really is who they say they are (e.g. "am I actually communicating with google.com?")

Bottom line: authentication is about proving that something is what it is supposed to be. And that thing can be a person, a message, or something else.

In this chapter, I will use the term authentication only to refer to identifying people or machines. In other words, identity authentication.

By the way, you've already seen a lot about this type of authentication:

- In chapter 9 on secure transport, you've learned that machines can authenticate other machines at scale by using public key infrastructures (so-called PKIs).
- In chapter 10 on end-to-end encryption, you've learned about ways humans can authenticate one another at scale by using trust on first use (TOFU) and fingerprints.

In this chapter, you will learn the two other cases not previously mentioned:

- **user authentication**, or how machines authenticate humans. Beep boop.
- **user-aided authentication**, or how humans can authenticate machines, or more accurately how humans can help their devices authenticate machines.

I recapitulate this in figure 11.1.

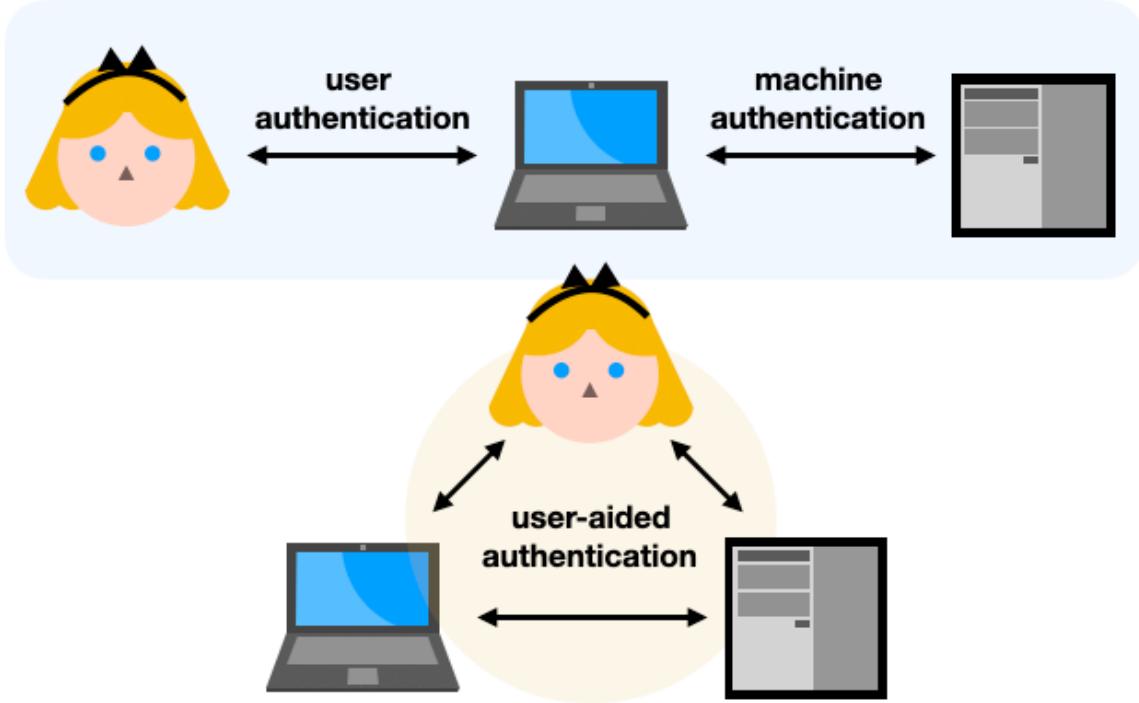


Figure 11.1 In this book I talk about origin authentication in three types of scenarios. **User authentication** happens when a device authenticates a human being. **Machine authentication** happens when a machine authenticates another machine. **User-aided authentication** happens when a human is involved in a machine authenticating another machine.

Another aspect of identity authentication is the identity part. Indeed, how do we define someone like Alice in a cryptographic protocol? How can a machine authenticate you and I? There is, unfortunately (or fortunately), an inherent gap between flesh and bits. To bridge reality and the digital world we always assume that, for example, Alice is the only one who knows some secret data. To prove her identity, Alice then has to demonstrate knowledge of that secret data. For example, she could be sending her password, or she could be signing a random challenge with the private key associated to her public key.

Alright, that's enough intro. If this section didn't make too much sense, the multitude of examples that are to follow will. Let's now first have a look at the many ways machines have found to authenticate us humans!

11.2 User authentication, or the quest to get rid of passwords

The first part of this chapter is about how machines authenticate humans, or in other words **user authentication**. There are many ways to do this, and no solution is a panacea. But in most user authentication scenarios, we assume that:

- the server is already authenticated
- the user shares a secure connection with it

For example, you can imagine that the server is authenticated to the user via the web public key infrastructure, and that the connection is secured via TLS (both covered in chapter 9). In a sense, most of this section is about **upgrading a one-way authenticated connection to a mutually-authenticated connection**, as illustrated in figure 11.2.

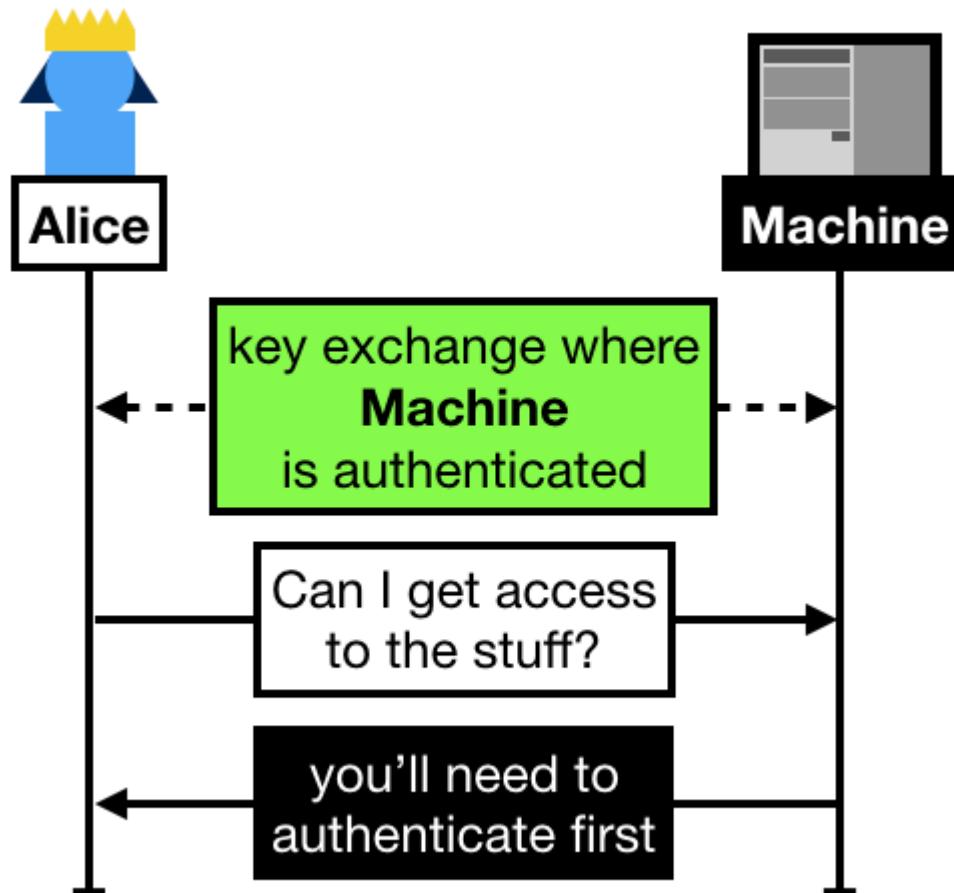


Figure 11.2 User authentication typically happens over a channel that is already secured, but where only the server is authenticated. A typical example is when you browse the web using HTTPS and log into a webpage using your credentials.

I have to warn you, user authentication is a vast land of broken promises. You must have used **passwords** many times to authenticate to different webpages, and your own experience probably resembles something like this:

- You register with a username and password on a website.
- You log into the website using your new credentials.
- You change your password after recovering your account or because the website forces you to.
- If you're out of luck, your password (or a hash of it) is leaked in a series of database breaches.

Sounds familiar?

NOTE

I will ignore password/account recovery in this chapter, as they have little to do with cryptography. Just know that they are often tied to the way you first registered (e.g. if you registered with the IT department of your workplace, then you'll probably have to go see them if you lose your password) and they can be the weakest link in your system if you are not careful. Indeed, if I can recover your account by calling a number and giving someone your birthdate, then no amount of cool cryptography at login time will help.

A naive way to implement the previous user authentication flow is to store the user password at registration, and then ask the user for it at login time (as illustrated in figure 11.3). As covered in chapter 3, once successfully authenticated a user is typically given a cookie that she can send in every subsequent requests instead of her username and password.

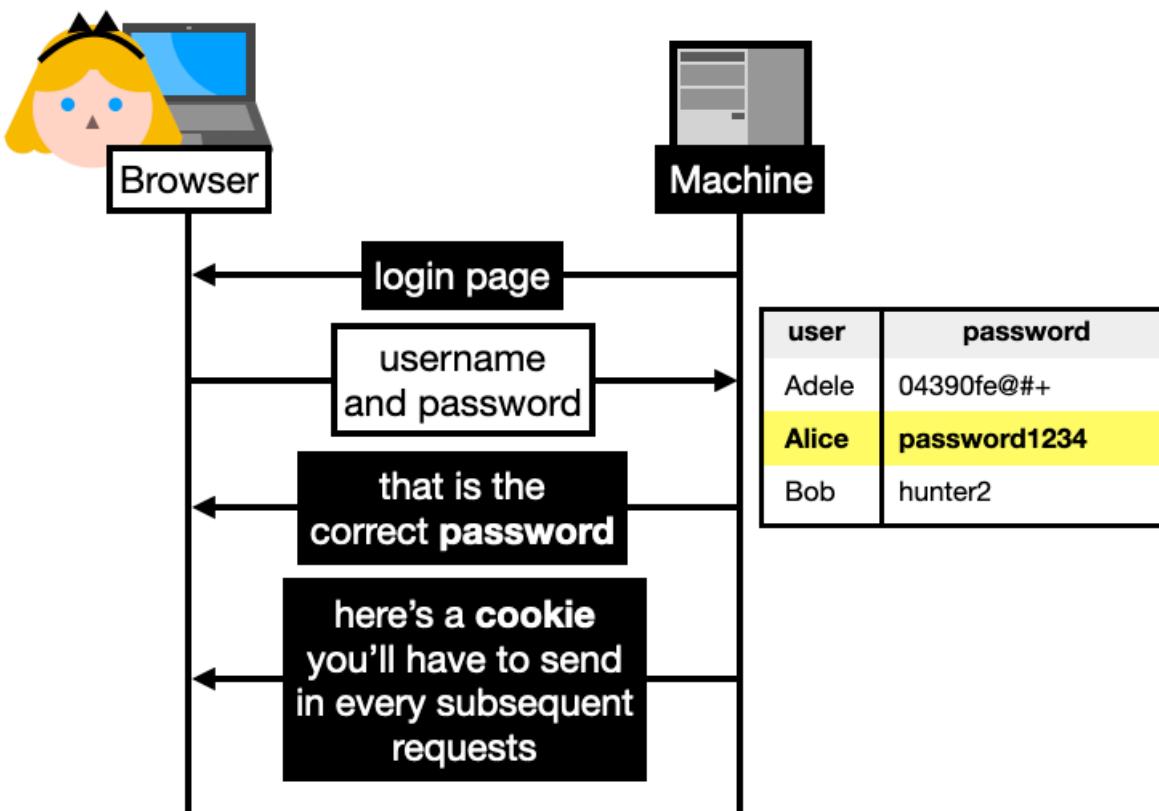


Figure 11.3 A password-based authentication flow. Alice sends her username and password in clear to the server (the server sees the password), and the server checks if it matches what it has in its database. If it does, Alice successfully authenticates.

But wait, if the server stores your password in clear, then any breach of their databases will reveal your password to the attackers. These attackers will then be able to use it to log into any websites where you used the same password to register. A better way to store passwords would be to use a **password hashing** algorithm like the standardized **Argon2** you've learned about in

chapter 2. This would effectively prevent the server to see your password every time you log in (although they would still see it at registration time). Yet, a lot of websites and companies still store passwords in clear.

NOTE Sometimes, login applications will attempt to fix the issue of the server learning about their passwords at registration time by having the client hash (or password hash) the password before sending it to the server. Can you tell if this really works?

Moreover, humans are naturally bad at passwords. We are usually most comfortable with small and easy-to-remember passwords. And if possible, we would want to just reuse the same password everywhere.

*81% of all hacking-related breaches leverage stolen or weak passwords.
– Verizon Data Breach Report 2017*

The problem of weak passwords and password reuse has led to many silly and annoying design patterns that attempt to force users to take passwords more seriously. For example, some websites will require you to use special characters in your passwords, or will force you to change password every 6 months, and so on.

Furthermore, many protocols attempt to "fix" passwords or to get rid of them altogether. Every year, new security experts seem to think that the concept of password is dead. Yet, it is still the most widely used user authentication mechanism.



So here you have it, passwords are probably here to stay. Yet, there exist many protocols that improve or replace passwords. Let's take a look at them!

11.2.1 One password to rule them all, single sign-on (SSO) and password managers

OK, password reuse is bad, what can we do about it? Naively, users could use different passwords for different websites, but there are two problems with this approach:

- Users are bad at creating many different passwords.
- The mental load required to remember multiple passwords is impractical.

To alleviate these concerns, two solutions have been widely adopted:

- **Single-sign on (SSO)**. The idea of SSO is to allow users to connect to many different services by proving that they own the account of a single service. This way the user only has to remember the password associated with that one service in order to be able to connect to many services. Think "connect with Facebook" type of buttons, as illustrated in figure 11.4.
- **Password Managers**. The previous SSO approach is convenient if the different services you use all support it, but this is obviously not scalable for scenarios like the web. A better approach in these extreme cases is to improve the client as opposed to attempting to fix the issue on the (too many) servers' side. Nowadays, modern browsers have built-in password managers that can suggest complex passwords when you register on new websites, and can remember all of these passwords as long as you remember one master password.

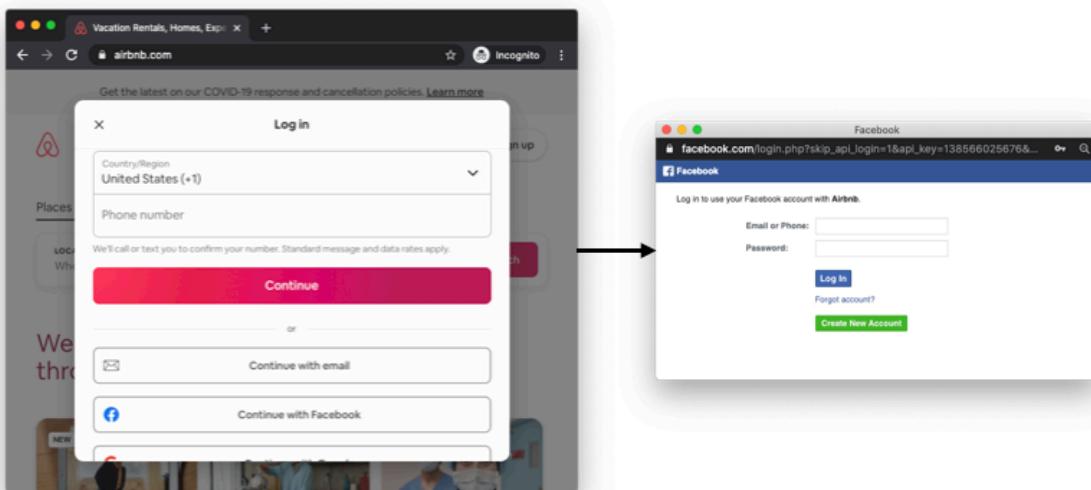


Figure 11.4 An example of single-sign on (SSO) on the web. By having an account on Facebook or Google, a user can connect to new services (in this example Airbnb) without having to think about a new password.

The concept of SSO is not new in the enterprise world, but its success with normal end-users is relatively recent. Today, two protocols are the main competitors when it comes to setting up SSO:

- **Security Assertion Markup Language 2.0 (SAML)**. A protocol using the Extensible Markup Language (XML) encoding.
- **OpenID Connect (OIDC)**. An extension to the **OAuth 2.0** (RFC 6749) authorization protocol using the JavaScript Object Notation (JSON) encoding.

SAML is still widely used, mostly in an enterprise setting, but it is at this point a legacy protocol.

OpenID Connect, on the other hand, can be seen everywhere on web and mobile applications. You most likely already used it!

While OpenID Connect allows for different types of flows, let's see the most common use case for user authentication on the web via the **authorization code flow**:

1. Alice wants to log into some application, let's say `example.com`, via an account she owns on `cryptologie.net` (that's just my blog, but let's pretend that you can register an account on it).
2. `example.com` redirects her browser to `cryptologie.net` where she asks the site to give her an "authorization code". If she is not logged-in in `cryptologie.net`, the website will first ask her to log in. If she is already logged-in, the website will still confirm with the user that they want to connect to `example.com` using their identity on `cryptologie.net` (it is important to confirm user intent).
3. `cryptologie.net` redirects Alice back to `example.com` which then learns the authorization code.
4. `example.com` can then query `cryptologie.net` with this authorization code to confirm Alice's claim that she owns an account on `cryptologie.net`, and potentially retrieve some additional profile information about that user.

I recapitulate this flow in figure 11.5.

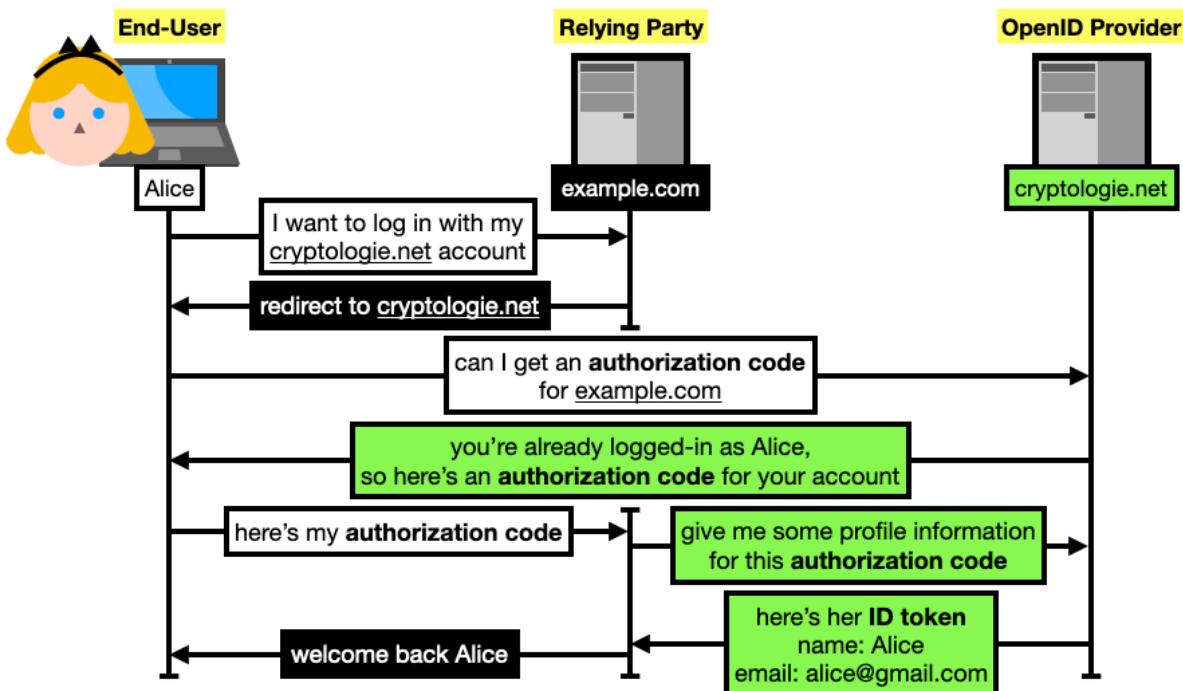


Figure 11.5 In OpenID Connect (OIDC), Alice (the end-user in OIDC terms) can authenticate to a service `example.com` (the relying party) using her already existing account on `cryptologie.net` (the OpenID provider). For the web, the authorization code flow of OIDC is usually used. It starts by having Alice request an "authorization code" from `cryptologie.net` (and that can only be used by `example.com`). `example.com` can then use it to query `cryptologie.net` for Alice's profile information (encoded as an ID token), and then associate her `cryptologie.net` identity with an account on their own platform.

There are many important details that I am omitting here. For example, the authorization token that Alice receives in step 2 must be kept secret, as it can be used to log in as her on `example.com`. Another example: so that `example.com` cannot re-use the authorization token to connect as Alice on a different website, the OpenID provider `cryptologie.net` retains an association between this authorization token and the intended audience (`example.com`). This can be done by simply storing this association in a database, or by having the authorization code contain this information authenticated (I explained a similar technique with cookies in chapter 3).

By the way, the proof given to `example.com` in step 4 is called an **ID token** in OpenID Connect, and is represented as a **JSON Web Token (JWT)** which is just a list of JSON-encoded data. I illustrate JWTs in figure 11.6.

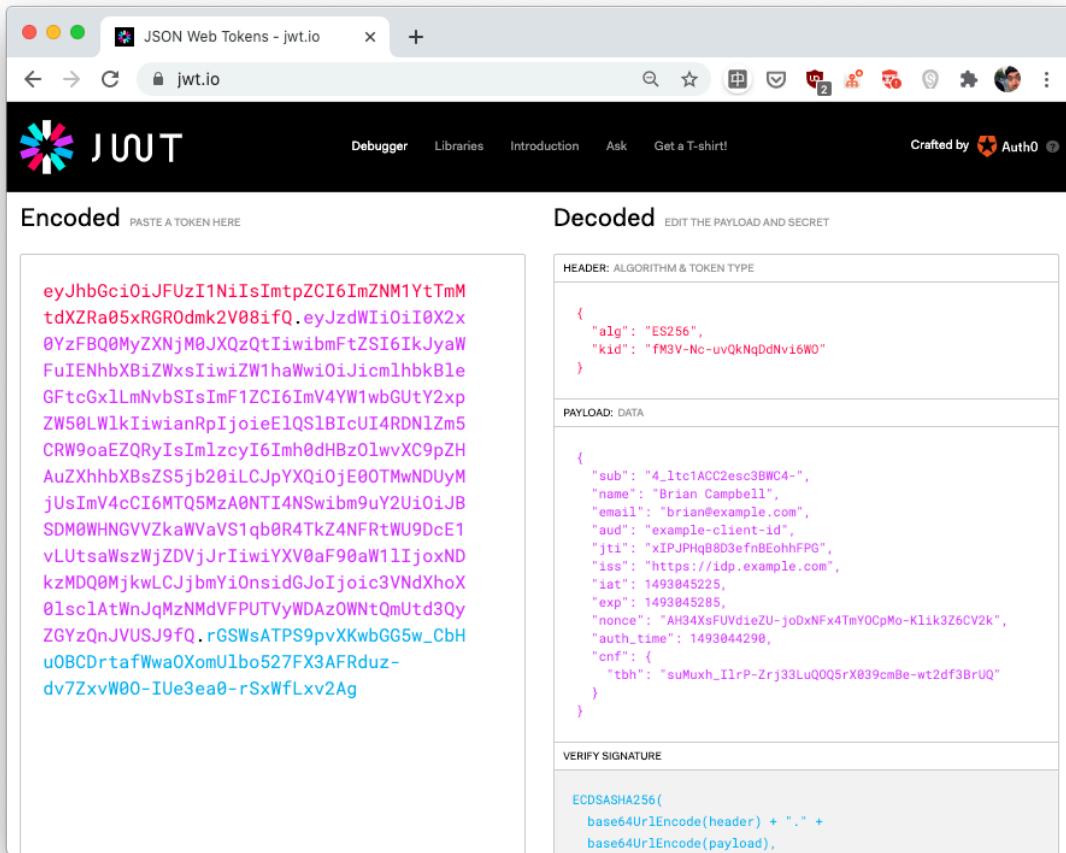


Figure 11.6 In OpenID Connect, if Alice wants to log in on example.com via her account on another website (an OpenID provider), example.com eventually needs to obtain what is called an "ID token". An ID token is some user information encoded via the JSON Web Token (JWT) standard. a JWT is simply the contatenation of three JSON-encoded objects. In the picture, the website jwt.io lets you decode a JWT and learn what every field stands for. In our browser-based example, example.com uses TLS to communicate with (and authenticate) the OpenID provider. Thus, the ID token it receives can be trusted. In other OpenID Connect flows (used for example by mobile applications) the ID token can be provided directly by the user, and can thus be tampered with. In this case, an ID token contains a signature which can be verified using the OpenID provider's public key.

Authentication protocols are often considered hard to get right. OAuth2, the protocol OpenID Connect relies on, is notorious for being easy to mis-use.⁷⁶ On the other hand, OpenID Connect is very well specified.⁷⁷ Make sure that you follow the standards and that you look at best practices, this can save you from a lot of trouble.

NOTE

Here's another example of a pretty large company deciding not to follow this advice. In May 2020, the "Sign-in with Apple" SSO flow that took a departure from OpenID Connect was found to be vulnerable. Anyone could have obtained a valid ID token for any Apple account, just by querying Apple's servers.⁷⁸

SSO is great for users, as it reduces the number of passwords they have to manage, but it does not remove passwords altogether. The user still has to use passwords to connect to OpenID providers. So next, let's see how cryptography can help hide passwords!

11.2.2 Don't want to see their passwords? Use an asymmetric password-authenticated key exchange

The previous section surveyed solutions that attempt at simplifying identity management for users, allowing them to authenticate to multiple services using only one account linked to a single service. While protocols like OpenID Connect are great, as they effectively decrease the number of passwords users have to manage, they don't change the fact that some service still needs to see the password of the user in clear. (Even if the password is stored after password hashing it, it is still sent in clear every time the user registers, changes their password, or logs in.)

Cryptographic protocols called **asymmetric (or augmented) password-authenticated key exchanges (PAKEs)** attempt at providing user authentication without having the user ever communicate their passwords directly to the server. They contrast with **symmetric or balanced PAKEs** protocols where both sides know the password.

The most popular asymmetric PAKE at the moment is the **Secure Remote Password (SRP)** protocol, which was standardized for the first time in 2000,⁷⁹ and later integrated into TLS.⁸⁰

It is quite an old protocol, and has a number of flaws.⁸¹ For example, if the registration flow is intercepted by a MITM attacker, the attacker would then be able to impersonate and log in as the victim. It also does not play well with modern protocols, it cannot be instantiated on elliptic curves and worse, it is incompatible with TLS 1.3.

Since then, many asymmetric PAKEs have been proposed and standardized. In the summer of 2019, the Crypto Forum Research Group (CFRG) of the IETF started a PAKE selection process,⁸² with the goal to pick one algorithm to standardize for each category of PAKEs (symmetric/balanced and asymmetric/augmented).

In march 2020, the CFRG announced the end of the PAKE selection process, selecting:

- **CPace** as the recommended symmetric/balanced PAKE (invented by from Björn Haase and Benoît Labrique)
- **OPAQUE** as the recommended asymmetric/augmented PAKE (invented by Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu)

In this section I will talk about OPAQUE, which is at the moment of this writing still in the process of being standardized.⁸³ In the second section of this chapter, you will learn more about symmetric PAKEs and CPace.

OPAQUE takes its name from the homonym O-PAKE, where O refers to the term "oblivious". This is because OPAQUE relies on a cryptographic primitive that I have not yet mentioned in this book: an **Oblivious Pseudo-Random Function (OPRF)**.

OBLIVIOUS PSEUDO-RANDOM FUNCTIONS (OPRFS)

OPRFs are a two-participant protocol that mimics the PRFs that you've learned about in chapter 3. As a reminder, a PRF is somewhat equivalent to what one would expect of a MAC: it takes a key and an input and gives you a totally random output of a fixed length.

NOTE

The term *oblivious* in cryptography generally refers to protocols where one party computes a cryptographic operation without knowing the input provided by another party.

Here is how an OPRF works at a high level:

1. Alice wants to compute a PRF over an input, but wants the input to remain secret. She "blinds" her input with a random value called a "blinding factor" and sends this to Bob.
2. Bob runs the OPRF on this blinded value with his secret key, the output is still blinded so useless for Bob. Bob then sends this back to Alice.
3. Alice finally "unblinds" the result, using the same blinding factor she had previously used, to obtain the real output.

It is important to note that every time Alice wants to go over this protocol, she has to create a different blinding factor. But no matter what blinding factor she uses, as long as she uses the same input she will always obtain the same result. I illustrate this in figure 11.7.

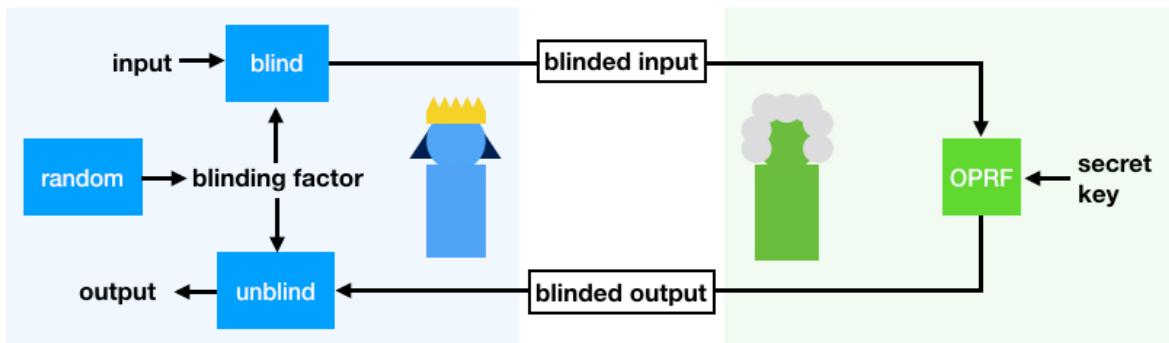


Figure 11.7 An oblivious PRF (OPRF) is a construction that allows one party to compute a PRF over the input of another party without learning that input. To do this, Alice first generates a random blinding factor, then blinds her input with it before sending it to Bob. Bob uses his secret key to compute the PRF over the blinded value, then send the blinded output to Alice who can then unblind it. The result does not depend on the value of the blinding factor.

Here's an example of an OPRF protocol implemented in a group where the discrete logarithm problem is hard:

1. Alice converts her input to a group element x .
2. Alice generates a random blinding factor r .
3. Alice blinds her input by computing $\text{blinded_input} = x^r$.
4. Alice sends the blinded_input to Bob.
5. Bob computes $\text{blinded_output} = \text{blinded_input}^k$ where k is the secret key.
6. Bob sends the result back to Alice.
7. Alice can then unblind the produced result by computing $\text{output} = \text{blinded_output}^{1/r} = x^k$ where $1/r$ is the inverse of r .

How OPAQUE uses this interesting construction is the whole trick behind the asymmetric PAKE!

THE OPAQUE ASYMMETRIC PAKE, HOW DOES IT WORK?

The idea is that we want a client, let's say Alice, to be able to do an authenticated key exchange with some server. We also assume that Alice already knows the server's public key, or already has a way to authenticate it (the server could be an HTTPS website and thus Alice can use the web PKI).

Let's see how we could build this, to progressively understand how OPAQUE works.

First idea: use public-key cryptography to authenticate Alice's side of the connection. If Alice owns a **long-term keypair** (and the server knows the public key) she can simply use it to perform a mutually-authenticated key exchange, or sign a challenge given by the server.

Unfortunately, an asymmetric private key is just too long and Alice can only remember her password... She could store a keypair on her current device, but she also want to be able to log in from another device later.

Second idea: Alice can derive the asymmetric private key from her password, using the password-based key derivation functions (like Argon2) that you've learned about in chapter 2 and chapter 8. Alice's public key could then be stored on the server. If we want to avoid someone testing a password against the whole database (in the case of a database breach) we can have the server supply each users with a different salt that they have to use with the password-based key derivation function.

This is pretty good already, but there's one attack that OPAQUE wants to discard: **pre-computation attacks**. I can try to log in as you, receive your salt, and then pre-compute a huge number of asymmetric private keys (and their associated public keys) **offline**. The day the database is compromised, I can quickly see if I can find your public key (and the associated password) in my huge list of pre-computed asymmetric public keys.

Third idea: That's where the **main trick of OPAQUE** comes in! We can use the **OPRF protocol with Alice's password** in order to derive the asymmetric private key. If the server uses

a different key per-user, that's as good as having salts (attacks can only target one user at a time). This way, an attacker that wants to pre-compute asymmetric private keys, based on guesses of our password, has to perform **online** queries. Online queries are good, because they can be rate-limited in order to prevent these kind of online brute-force attacks.

NOTE

OPAQUE goes a bit further. Instead of having the user derive an asymmetric private key, it has the user derive a symmetric key. The symmetric key is then used to encrypt a backup of your asymmetric keypair and some additional data (like the server's public key). In this case, it prevents one more attack: **offline brute-force attacks**. Indeed, since you have to retrieve a backup of your keypair as part of the login flow, an attacker can use the backup to test password guesses **offline** (by trying to decrypt the encrypted backup with each symmetric key derived from password guesses). Due to the use of an OPRF, OPAQUE forces every password guess to be performed **online**.

Pretty clever no? I illustrate this technique in figure 11.8.

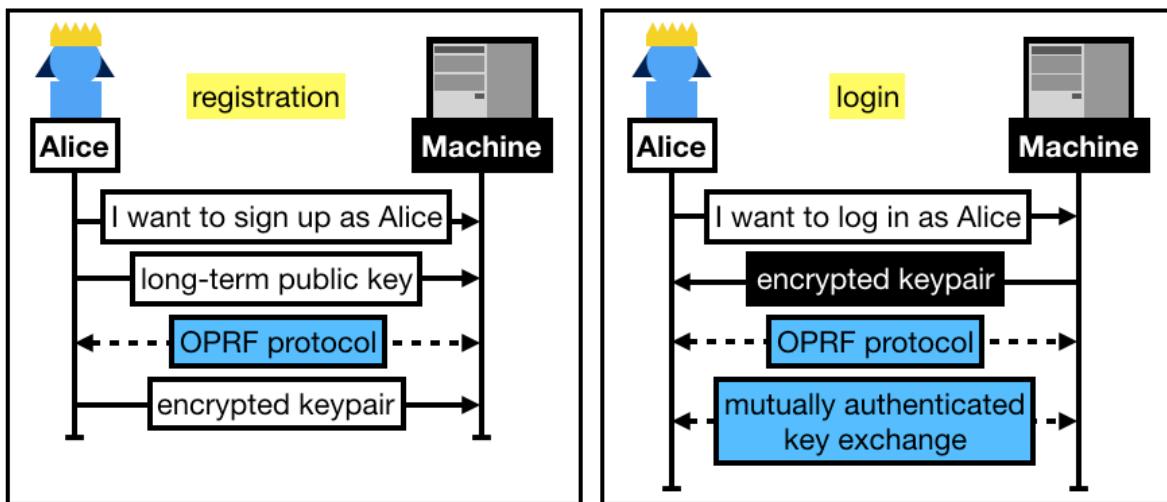


Figure 11.8 To register to a server using OPAQUE, Alice generates a long-term keypair and send her public key to the server who stores it and associates it with Alice's identity. She then uses the OPRF protocol to obtain a strong symmetric key from her password, and send an encrypted backup of her keypair to the server. To log in, she obtains her encrypted keypair from the server, then performs the OPRF protocol with her password to obtain a symmetric key capable of decrypting her keypair. All that's left is to perform a mutually-authenticated key exchange (or possibly sign a challenge) with this key.

Before going to the next section, let's recapitulate what you've learned here in figure 11.9.

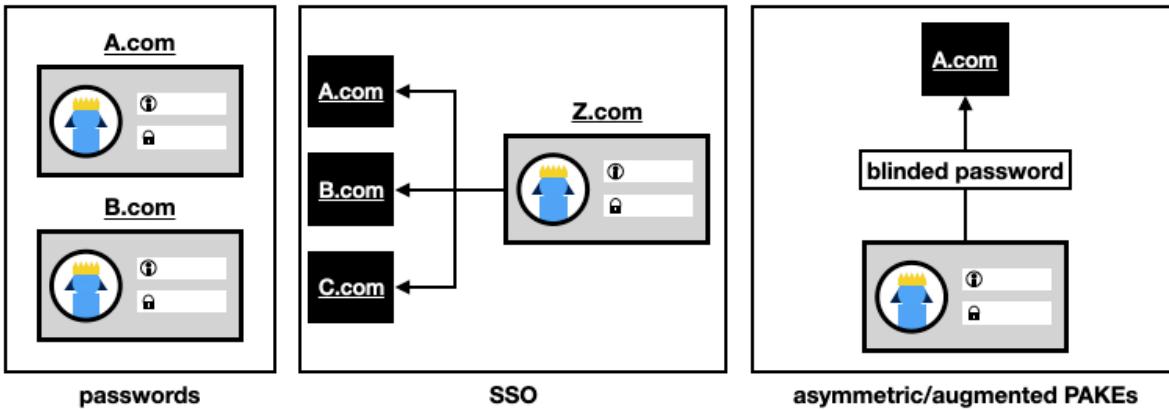


Figure 11.9 Passwords are a very handy way to authenticate users, as they live in someone's head and can be used on any devices. But on the other hand, users have trouble creating many strong passwords, and password breaches can be damaging as users tend to re-use passwords across websites. Single-sign on (SSO) allows you to connect to many services using one (or a few) service(s), while asymmetric (or augmented) password-authenticated key exchanges allow you to authenticate without the server ever learning your real password.

11.2.3 One-time passwords aren't really passwords, going passwordless with symmetric keys

Alright, so far so good. You've learned about different protocols that applications can leverage to authenticate users with passwords. But as you've heard, passwords are also not that great, they are vulnerable to brute force attacks, tend to be reused, stolen, and so on. So what is available to us if we can afford to avoid passwords?

Keys!

And as you know there are two types of keys in cryptography, and both types can be useful:

- Symmetric keys.
- Asymmetric keys.

This section goes over solutions that are based on symmetric keys, while the next section will go over solutions based on asymmetric keys.

Let's imagine that Alice registers with a service using a symmetric key (often generated by the server and communicated to you via a QR code). A naive way to authenticate Alice later would be to simply ask her to send the symmetric key. This is of course not great, as a compromise of her secret would give an attacker unlimited access to her account. Instead, Alice can derive what are called **one-time passwords (OTPs)** from the symmetric key and send that in place of the longer-term symmetric key. Even though an OTP is not a password, the naming indicates that an OTP can be used in place of a password, and warns that it should never be reused.

The idea behind OTP-based user authentication is straightforward: your security comes from the

knowledge of a (usually 16 to 32-byte uniformly random) symmetric key instead of a low-entropy password. This symmetric key allows you to generate one-time passwords on demand, as illustrated by figure 11.10.



Figure 11.10 A one-time password (OTP) algorithm allows you to create as many one-time passwords as you want from a symmetric key and some additional data. The additional data is different depending on the OTP algorithm.

There are two main schemes that one can use to produce OTPs:

- The **HMAC-based One-time Password algorithm (HOTP)** standardized in RFC 4226.
An OTP algorithm where the additional data is a counter.
- The **Time-based One-Time Password algorithm (TOTP)** standardized in RFC 6238.
An OTP algorithm where the additional data is the time.

Most applications nowadays use TOTP, as HOTP requires you to store a state (a counter) on both sides which might lead to issues if states fall out of synchrony.

OTP-based authentication is most often implemented in mobile applications (see figure 11.11 for a popular example) or in security keys (a small device that you can plug in the USB port of your computer).

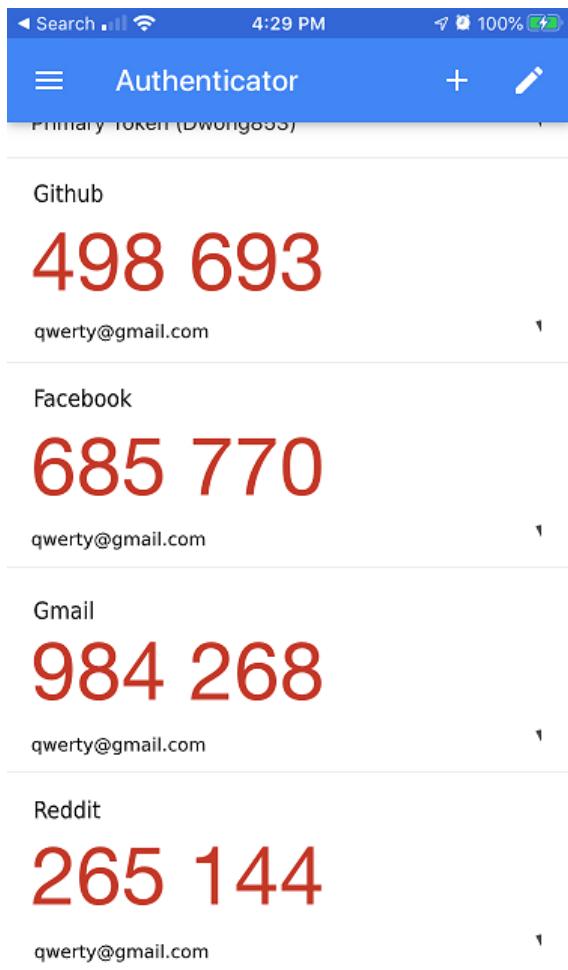


Figure 11.11 A screenshot of the Google Authenticator mobile app. The application allows you to store unique per-application symmetric keys, which can then be used with the TOTP scheme to generate 6-digit one-time passwords (OTPs) valid only for 30 seconds.

In most cases, this is how TOTP works:

At registration time, the service communicates a symmetric key to the user (perhaps using a QR code). The user then adds this key to a TOTP application.

At login time, The user can use the TOTP application to compute a one-time password. This is done by computing $\text{HMAC}(\text{symmetric_key}, \text{time})$ where time represents the current time (rounded to the minute in order to make a one-time password valid for 60 seconds). The TOTP application then displays the derived one-time password truncated and in a human-readable base to the user (for example reduced to 6 characters in base 10 to make it all digits). The user then either copies or types the one-time password into the relevant application. The application retrieves the user's associated symmetric key, and computes the one-time password in the same way as the user did. If the result matches the received one-time password, the user is successfully authenticated.

I recapitulate this flow in figure 11.12.

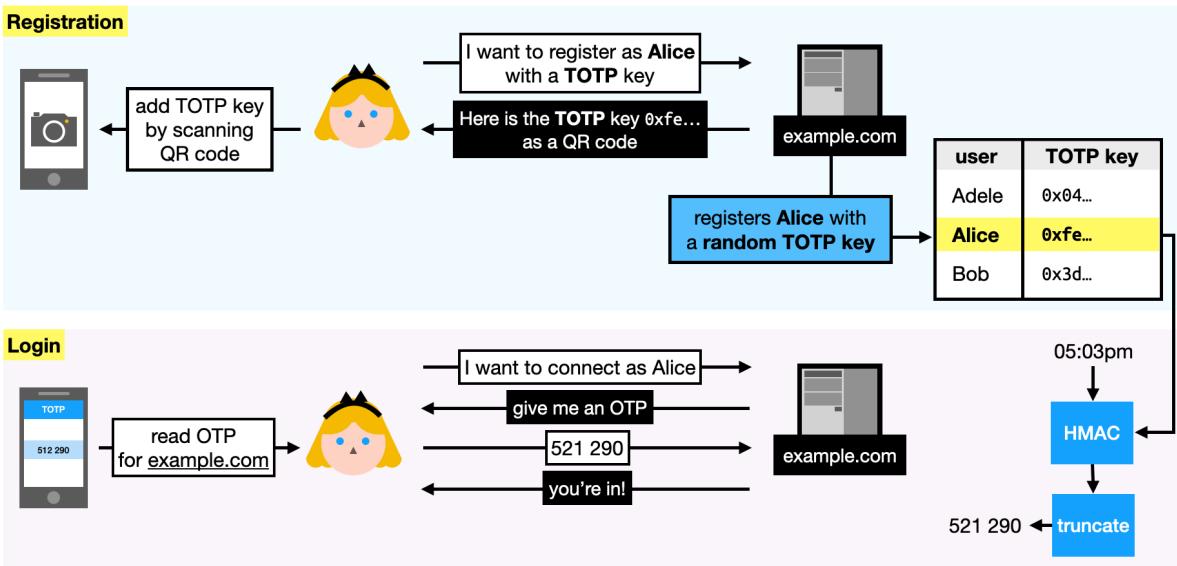


Figure 11.12 Alice registers with example.com using TOTP as authentication, by importing a symmetric key from the website into her TOTP application. Later, she can ask the application to compute a one-time password for example.com, and use it to authenticate with the website. Example.com just has to fetch the symmetric key associated to Alice, and compute the one-time password as well (using HMAC and the current time), then compare it in constant-time with what she sent.

Of course, similar to MAC authentication tag checks, the comparison between the user's OTP and the one computed on the server must be done in constant-time.

This authentication flow is not ideal though. There are a number of things that could be improved, for example:

- The authentication can be faked by the server, as they also own the symmetric key.
- You can be social engineered out of your one-time password.

NOTE

Phishing (or social engineering) is an attack that does not target vulnerabilities in the software, but rather vulnerabilities in human beings. Imagine that an application requires you to enter a one-time password to authenticate. What an attacker could do in this case, is to attempt to log in the application as you, and when prompted with a one-time password request, give you a call to ask you for a valid one (pretending that they work for the application). You're telling me you wouldn't fall for it, but good social engineers are very good at spinning believable stories and fabricating a sense of urgency that would make the best of us spill the beans. If you think about it, all the protocols that we've talked about previously are vulnerable to these types of attack.

For this reason, symmetric keys are yet another not-perfect replacement for passwords.

Next, let's see how using asymmetric keys can address these downsides.

11.2.4 Replacing Passwords With Asymmetric Keys

Now that we're dealing with public-key cryptography, there's more than one way we can use asymmetric keys to authenticate to a server:

1. by using our asymmetric key inside a key exchange to authenticate our side of the connection.
2. by using our asymmetric key in an already-secured connection with an authenticated server.

Let's see each methods

MUTUAL AUTHENTICATION IN KEY EXCHANGES

You've already heard about the first method. In chapter 9, I mentioned that a TLS server can request the client to use a **certificate** as part of the handshake. Often, companies will provision their employees' devices with a unique per-employee certificate that will allow them to authenticate to internal services. See figure 11.13 for an idea of what it looks like from a user perspective.

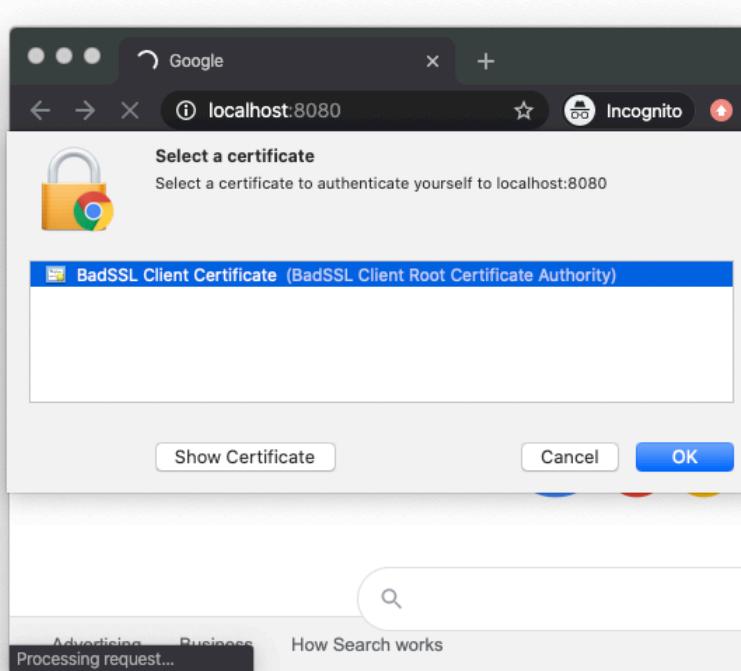


Figure 11.13 A page prompting the user's browser for a client certificate. The user can then select which certificate to use from a list of locally installed certificates. In the TLS handshake, the client certificate's key is then used to sign the handshake transcript, including the client's ephemeral public key used as part of the handshake.

Client-side certificates are pretty straightforward to understand. For example, in TLS 1.3, a server can request the client to authenticate during the handshake by sending a `CertificateRequest` message. The client then responds by sending its certificate (in a `Certificate` message) followed by a signature of all messages sent and received (in a `CertificateVerify` message). The signature of course includes the ephemeral public key used in the handshake's Diffie-Hellman key exchange. The client is authenticated if the server can recognize the certificate and successfully verify the client's signature.

Another example is the Secure Shell (SSH) protocol, which also have the client sign parts of the handshake with a public key known to the server.

Note that signing is not the only way to authenticate with public-key cryptography during the handshake phase. The Noise protocol framework (covered in chapter 9 as well) has several handshake patterns that enable client-side authentication using just Diffie-Hellman key exchanges.⁸⁴

POST-HANDSHAKE USER AUTHENTICATION WITH FIDO2

The second type of authentication with asymmetric keys use an already secure connection where only the server is authenticated.

To do this, a server can simply ask the client to sign a random challenge (this way replay attacks are prevented).

One interesting standard in this space is the **Fast IDentity Online 2 (FIDO2)**. FIDO2 is an open standard that defines how to use asymmetric keys to authenticate users. The standard specifically targets phishing attacks, and for this reason FIDO2 is made to work only with **hardware authenticators**. Hardware authenticators are simply physical components that can generate and store signing keys, and sign arbitrary challenges.

FIDO2 is split into two specifications:

- **Client-to-Authenticator Protocol (CTAP)**. CTAP specifies a protocol that **roaming authenticators** and **clients** can use to communicate with one another. Roaming authenticators are hardware authenticators that are external to your main device (think security keys, like the one pictured in figure 11.14). A client in the CTAP specification is defined as the software that wants to query these authenticators as part of an authentication protocol. Thus a client can be an operating systems, a native application like a browser, and so on.
- **Web Authentication (WebAuthn)**. WebAuthn is the protocol that web browsers and web applications can use to authenticate users with hardware authenticators. It thus must be implemented by browsers to support authenticators. If you are building a web application and want to support user authentication via hardware authenticators, WebAuthn is what you need to use. The standard allows websites to use not only roaming authenticators, but also "platform" authenticators. Platform authenticators are

built-in authenticators provided by a device. They are usually implemented differently by different platforms, and often protected by biometrics (e.g. a fingerprint reader, facial recognition, and so on).



Figure 11.14 A Yubikey is a security key, one type of "roaming" authenticator that can be used with FIDO2. Its role is to store private keys associated with different applications. It is inserted in the USB port of a computer (roaming authenticators can communicate with a platform via USB, Near Field Communication (NFC), or Bluetooth) and will not perform any cryptographic operations unless triggered by the touch of a finger on its gold metal button.

From a cryptography perspective, what is interesting to us is how applications can use these authenticators, or the asymmetric keys carried by them, to authenticate users. At a high level, applications can register a user's authenticator by asking it to create a new unique-to-the-application private key, and advertise the associated public key back to the application.

Applications are free to design their own authentication protocol on top of these authenticators, and implement FIDO2's CTAP to communicate with them. Let's see how WebAuthn deals with web app authentication:

1. Alice wishes to register a security key as a way to authenticate to `example.com` (this is called a "registration ceremony"). She clicks on some button on the website that lets her do just that.
2. `example.com` generates a challenge (a random number) along with an application id (maybe `example.com` has different applications).
3. Alice's browser sends a registration request to the security key using CTAP. The request contains binding information (the origin `example.com`), and the given challenge.
4. The security key generates a new keypair for this origin and application id, then replies with a signature over both the challenge and the origin, and the public key newly created.
5. The browser relays the public key and signature to `example.com` who can verify the signature and store Alice's public key.

And the login flow:

1. Alice wishes to log into `example.com` with a security key that she previously registered with the website (this is called an "authentication ceremony").
2. `example.com` (called the "relaying party") generates a challenge and send it to the browser.
3. Alice's browser sends an authentication request to the security key using CTAP, the request contains binding information (the origin `example.com`), and the given challenge.
4. The security key replies with a signature over both the challenge and the origin.
5. The browser relays the signature to `example.com`, who can verify the signature with Alice's public key (that it had stored at registration).

I illustrate this flow in figure 11.15.

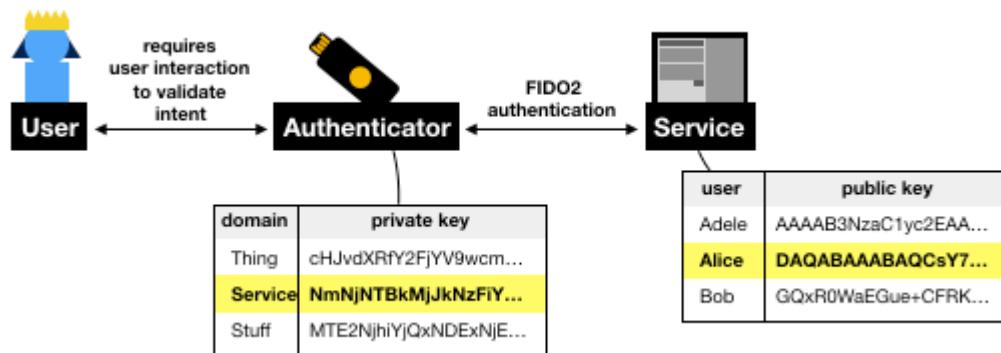


Figure 11.15 FIDO2 standardizes protocols for services to talk to authenticators, in order to provide user authentication only if the user consented to it it.

We are now ending the first part of this chapter. I recapitulate the non-password-based authentication protocols I've talked about in figure 11.16.

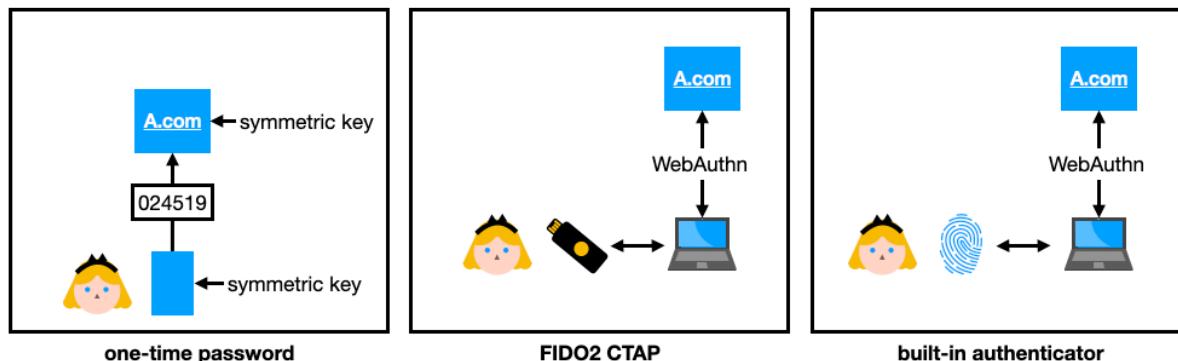


Figure 11.16 To authenticate without using a password, applications can allow users to either use symmetric keys via one-time password (OTP) based protocols, or use asymmetric keys via roaming authenticators or built-in authenticators.

Now that you have learned about many different techniques and protocols that exist to either improve passwords, or replace them with stronger cryptographic solutions, you might be wondering which one you should use. Especially, each of these solutions have their own caveats, and no single solution might do it. If so, combine multiple ones! This idea is called **multi-factor**

authentication (MFA). Actually, chances are that you might have already used OTPs or FIDO2 as a second authentication factor, in addition to (and not in place of) passwords.

Next, let's take a look at how humans can help devices to authenticate each other.

11.3 User-aided authentication, pairing devices using some human help

Humans help machines to authenticate one another every day. EVERY DAY!

You've done it by **pairing** your wireless headphones with your phone, or by pairing your phone with your car, or by connecting some device to your home WiFi, and so on. And as with any pairing stuff, what's underneath is most probably a key exchange.

The authentication protocols in the last section took place in already-secured channels (perhaps with TLS) where the server was authenticated.

Most of this section, in contrast, attempt at providing a secure channel to two devices that do not know how to authenticate each other. In that sense, what you'll be learning in this section is how humans can help to **upgrade an insecure connection into a mutually-authenticated connection**. For this reason, the techniques you will learn about next should be reminiscent of some of the trust establishment techniques in the end-to-end protocols of chapter 10 (except that there, two humans were trying to authenticate themselves to each other).

Nowadays the most common insecure connections that you will run into, that do not go through the internet, are protocols based on short-range radio frequencies like Bluetooth, WiFi, and Near Field Communication (NFC). Devices that use these communication protocols tend to range from low-power electronics to full-featured computers.

This already sets some constraints for us:

- The device you are trying to connect to might not offer a screen to display a key, or a way to manually enter a key (we call this **provisioning** the device). For example, most wireless audio headsets today only have a few buttons and that's it.
- As a human is part of the validation process, having to type or compare long strings is often deemed impractical and not very user-friendly. For this reason many protocols attempt to shorten security-related strings to 4 or 6 digits PINs. (Imagine a protocol where you have to enter the correct 4-digit PIN to securely connect to a device, what are the chances to pick a correct PIN by just guessing?)

You're probably thinking back at some of your device pairing experiences, and realizing now that a lot of them **just worked™**:

1. You pushed some button on some device.
2. It entered pairing mode.

3. You then tried to find the device in the bluetooth device list on your phone.
4. After clicking on it, it successfully paired the device with your phone.

If you've read chapter 10, this should remind you of **trust on first use (TOFU)**, except that this time we also have a few more cards in our hand:

- **Proximity**. Both devices have to be close to each other. If using the NFC protocol (see figure 11.17 for a usecase example), the devices have to be really close to each other.
- **Time**. Device pairing is often time-constrained. It is common that if, for example, in a 30 second window the pairing is not successful, the process must be manually restarted.



Figure 11.17 Near Field Communication (NFC) is a set of wireless protocols that allow devices to communicate when brought within 10cm of each other. The protocol is most commonly used by "contactless" payment systems (mobile payment applications like in the picture, contactless credit cards, and so on).

Unlike TOFU though, these real life scenarios usually do not allow you to validate after the fact that you've connected to the right device. This is of course not ideal, and one should strive for better security if possible.

NOTE

By the way this is how the Bluetooth core specification actually calls the TOFU-like protocol: "Just Works". I should mention that all built-in Bluetooth security protocols are currently broken, due to many attacks including the latest KNOB attack released in 2019.⁸⁵ The techniques surveyed in this chapter are nonetheless secure if designed and implemented correctly.

What's next in our toolkit? This is what we will see in this section: ways for a human to help devices to authenticate themselves.

Spoiler alert:

1. You'll see that cryptographic keys are always the most secure approach, but not necessarily the most user-friendly one.
2. You'll learn about symmetric PAKEs and how you can input the same password on two devices to connect them securely.
3. Finally you'll learn about protocols based on short authenticated strings (SAS) which authenticate a key exchange after the fact, by having you compare and match two short strings displayed by the two devices.

Let's get started!

11.3.1 Pre-Shared Keys

Naively, the first approach to connect a user to a device would be to reuse protocols that you've learned about in chapter 9 or chapter 10 (for example TLS or Noise), and to provision both devices with a symmetric shared secret, or better with long term public keys (in order to provide forward secrecy to future sessions).

This means that you need two things for each device to learn the other device's public key:

- You need a way to **export** a public key from its device.
- You need a way for a device to **import** public keys.

As we will see, this is not always straightforward or user-friendly.

But remember, we have a human in the mix, who can observe and maybe play with the two devices. This is unlike other scenarios that we've seen before. We can use this to our advantage! All the protocols that are to follow will be based on the fact that you have an additional **out-of-band** channel (you, the human in charge) that allows you to securely communicate some information.

The Authentication Problem - One of the main issues in cryptography is the establishment of a secure peer-to-peer (or group) communication over an insecure channel. With no assumption,

such as availability of an extra secure channel, this task is impossible. However, given some assumption(s), there exists many ways to setup a secure communication.

– Sylvain Pasini - *Secure Communication Using Authenticated Channels* 2009

The addition of this out-of-band channel can be modeled as the two devices having access to two types of channels:

- An insecure channel. Think about a bluetooth or a WiFi connection with a device. By default, the user has no way of authenticating the device and can thus be man-in-the-middle.
- An authenticated channel. Think about a screen on a device. The channel provides integrity/authenticity of the information communicated, but poor confidentiality (someone could be looking over your shoulder).

I illustrate this in figure 11.18.

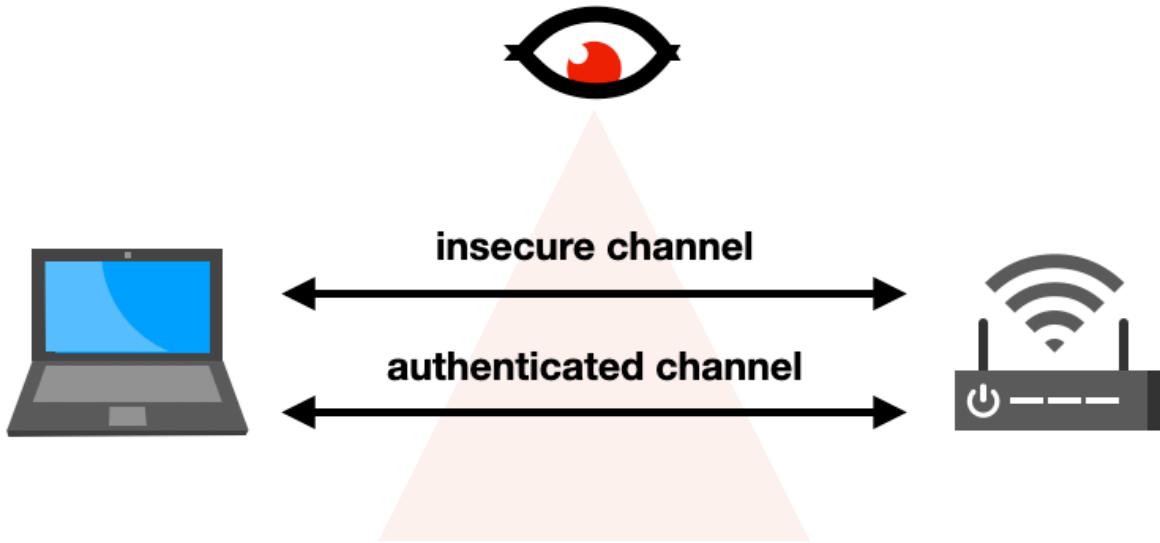


Figure 11.18 User-aided authentication protocols, that allow a human user to pair two devices, are modeled with two types of channels between the devices: an insecure channel (e.g. NFC, Bluetooth, WiFi, etc.) that we assume is adversary controlled and an authenticated channel (e.g. real life) that does not provide confidentiality but can be used to exchange relatively small amounts of information.

As this out-of-band channel provides poor confidentiality, we usually do not want to use it to export secrets, but rather public data. For example, a public key or some digest can be displayed by the device's screen.

Once you have exported a public key, you still need the other device to import it. For example, if the key was a QR code then the other device might be able to scan it, or if the key was encoded in a human-readable format then the user could manually type it in the other device using a keyboard.

Once both devices are provisioned with each others' public keys, you can use any protocols I've

mentioned in chapter 9 to perform a mutually-authenticated key exchange with the two devices.

What I want you to get from this section is that using cryptographic keys in your protocol is always the most secure way to achieve something, but that it is not always the most user-friendly way. Yet, real-world cryptography is full of compromise and trade-offs, and this is why the next two schemes not only exist but are the most popular ways to authenticate devices.

So let's see how **passwords** can be used to bootstrap a mutually-authenticated key exchange in cases where you cannot export and import long public keys, and then let's see how **short authenticated strings** can help when importing data is just not possible.

11.3.2 Symmetric Password-Authenticated Key Exchanges with CPace

The previous solution is what you should be doing if possible, as it relies on strong asymmetric keys as a root of trust. Yet, it turns out that in practice, typing a long string representing a key with some cumbersome keypad manually is not very user-friendly. What about these dear passwords? They are so much shorter, and thus easier to deal with. We love passwords right? Perhaps we don't, but users do, and real-world cryptography is full of compromises. So be it.

In the section on asymmetric password-authenticated key exchanges, I mentioned that a symmetric (or balanced) version exists where two peers who know a common password can perform a mutually-authenticated key exchange. This is exactly what we need.

Composable Password Authenticated Connection Establishment (CPace) was proposed in 2008 by Björn Haase and Benoît Labrique, and was chosen in early 2020 as the official recommendation of the CFRG.

The algorithm is currently being standardized as an RFC.⁸⁶ The protocol, simplified, looks like this:

1. The two devices derive a generator (for some pre-determined cyclic group) based on the common password.
2. The two devices use this generator to perform an ephemeral Diffie-Hellman key exchange on top of it.

I illustrate the algorithm in figure 11.19.

$h = g^x$
 with x derived from
 the common **password** and **metadata**

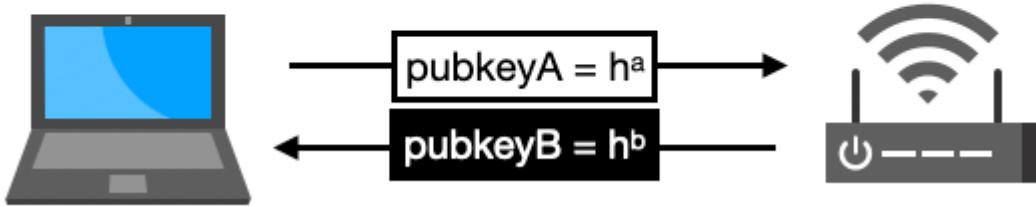


Figure 11.19 The CPace PAKE works by having the two devices generate a generator based on the password, and use it to perform as base for the usual ephemeral Diffie-Hellman key exchange.

The devil is in the details of course, and as a modern specification CPace targets elliptic curve gotchas and defines when one must verify that a received point is in the right group (due to the trendy Curve25519 that unfortunately does not span a prime group). It also specifies how one derives a generator based on a password when in an elliptic curve group (using so-called hash-to-curve algorithms), and how to do it using not only the common password but a unique session id and some additional contextual metadata like IP addresses of the peers and so on. Finally the session key is derived from the Diffie-Hellman key exchange output, the transcript (the ephemeral public keys) and the unique session id.

Intuitively, you can see that impersonating one of the peer, and sending a group element as part of the handshake, means that you're sending a public key which associated private key you cannot know. This means essentially that you can never perform the Diffie-Hellman key exchange if you don't know the password. The transcript just looks like a normal Diffie-Hellman key exchange and so no luck there as well as long as Diffie-Hellman is secure.

11.3.3 Was my key exchange man-in-the-middled? Just check a short authenticated string (SAS)

In the second part of this chapter, you've seen different protocols that allow two devices to be paired with the help of a human. Yet, I've mentioned that some devices are so constrained that they cannot make use of them. Let's take a look at a scheme that is used when the two devices cannot import keys, but can display some limited amount of data to the user. Perhaps via a screen, or by turning on some LEDs, or by emitting some sounds, etc.

First, remember that in chapter 10 you learned about authenticating a session post-handshake (after the key exchange) using fingerprints (i.e. hashes of the transcript). We could use something like this, as we have our out-of-band channel to communicate these fingerprints. If the user can successfully compare and match the fingerprints obtained from both devices, then the

user knows that the key exchange was not man-in-the-middle. The problem with fingerprints is that they are long bytestrings (typically 32-byte long) which might be hard to display to the user, and are also cumbersome to compare. But for device pairing, we can use much shorter bytestrings as we are doing the comparison in real time!

We call these **short authenticated strings (SAS)**. SAS are used a lot, notably by Bluetooth, due to them being quite user-friendly (see figure 11.20 for an example).

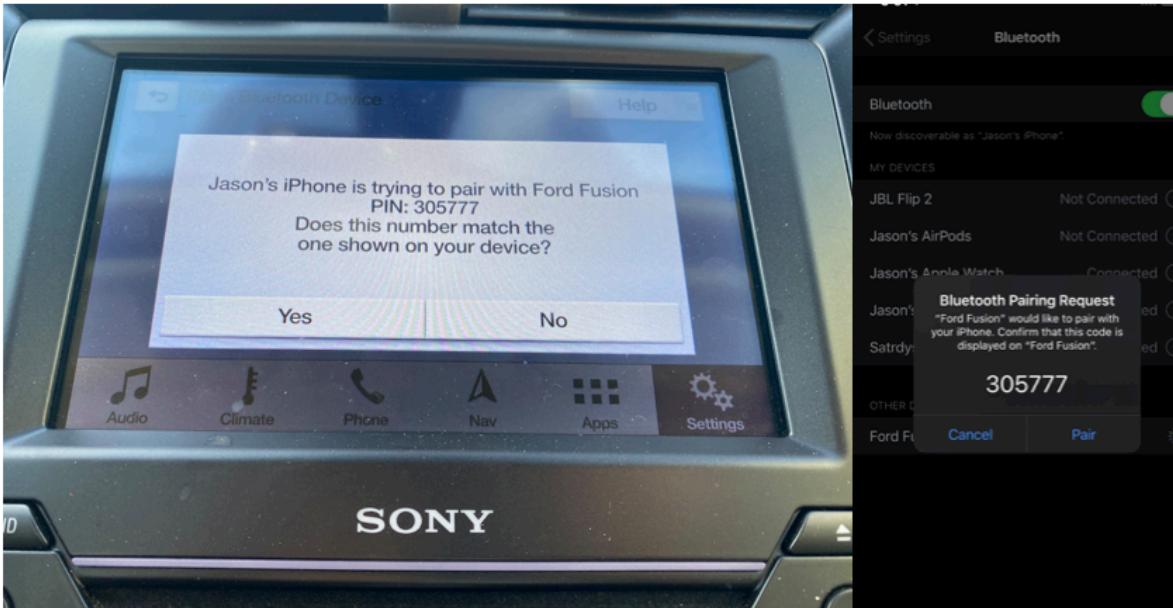


Figure 11.20 To pair a phone with a car via Bluetooth, the “numeric comparison” mode can be used to generate a short authenticated string (SAS) of the secure connection negotiated between the two devices. Unfortunately, as I stated earlier in this chapter, due to the KNOB attack Bluetooth’s security protocols are currently broken (as of 2020). If you control both devices, you need to implement your own SAS protocol.

There aren’t any standards for SAS-based schemes, but most protocols (including Bluetooth’s numeric comparison) implement a variant of the **Manually Authenticated Diffie-Hellman (MA-DH)**.⁸⁷ MA-DH is a simple key exchange with an additional trick that makes it hard for an attacker to actively man-in-the-middle the protocol.

But you might ask, why not just create a SAS from truncating a fingerprint? Why the need for a trick?

A SAS is typically a 6-digit number, which can be obtained by truncating a hash of the transcript to less than 20 bits and converting that to numbers in base 10. A SAS is thus dangerously small, which makes it much easier for an attacker to obtain a **second pre-image** on the truncated hash. In figure 11.21, we take the example of two devices (although we use Alice and Bob) performing an unauthenticated key exchange. An active man-in-the-middle attacker can substitute Alice’s public key with their own public key in the first message. Once the attacker receives Bob’s public key, they know what SAS Bob will compute (a truncated hash based on the attacker’s

public key and on Bob's public key). So the attacker just has to generate many public_key_{E2} (before sending it to Alice) in order to find one that will make the SAS of Alice match with Bob's.

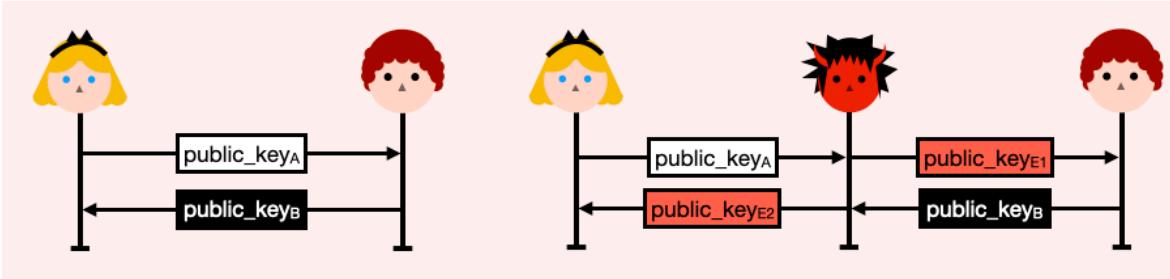


Figure 11.21 A typical unauthenticated key exchange (diagram on the left) can be intercepted by an active man-in-the-middle attacker (diagram on the right) who can then substitute the public keys of both Alice and Bob. A man-in-the-middle attack is successful if both Alice and Bob generate the same short authenticated string. That is, if $\text{hash}(\text{public_key}_A \parallel \text{public_key}_{E2})$ and $\text{hash}(\text{public_key}_{E2} \parallel \text{public_key}_B)$ match.

Generating a public key to make both SAS match is actually pretty easy. Imagine that the SAS is 20 bits, then after only 2^{20} computations you should find a second pre-image that will have both Alice and Bob generate the same SAS. This should be pretty instant to compute, even on a cheap phone.

The trick behind SAS-based key exchanges is to prevent the attacker from being able to choose their second public key to force the two SAS to match. To do this, Alice simply sends a **commitment** of her public key before seeing Bob's public key (as in figure 11.22).

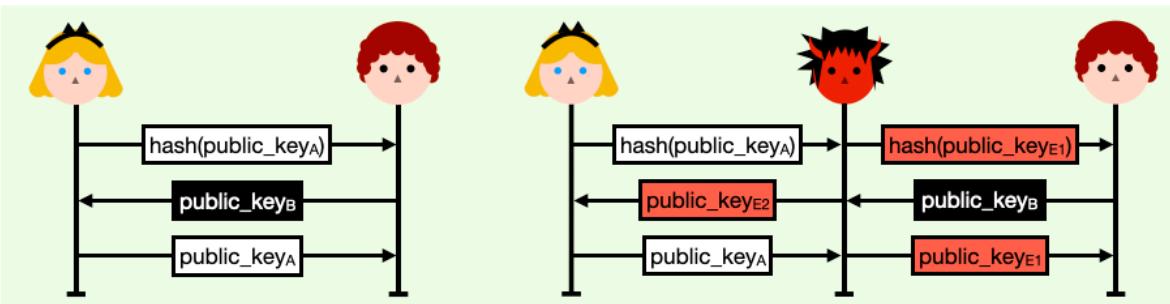


Figure 11.22 The diagram on the left pictures a secure SAS-based protocol in which Alice first sends a commitment of her public key. She then only reveals her public key after receiving Bob's public key. As she committed to it, she cannot freely choose her keypair based on Bob's key. If the exchange is actively man-in-the-middle'd (diagram on the right), the attacker cannot choose either keypairs to force Alice and Bob SAS to match.

As with the previous insecure scheme, the attacker's choice of public_key_{E1} does not give them any advantage. But now, they also cannot choose a public_key_{E2} that helps, as they do not know Bob's SAS at this point in the protocol. They are thus forced to shoot in the dark, and hope that Alice's and Bob's SAS will match. If a SAS is 20 bits, that's a probability of 1 out of 1,048,576.

An attacker can have more chances by running the protocol multiple times, but keep in mind that every instance of the protocol must have the user manually match a SAS. Effectively, this friction naturally prevents an attacker from having too many lottery tickets.

This is it! Figure 11.23 recapitulate the different techniques you learned in the second part of this chapter. I'll see you in chapter 12.

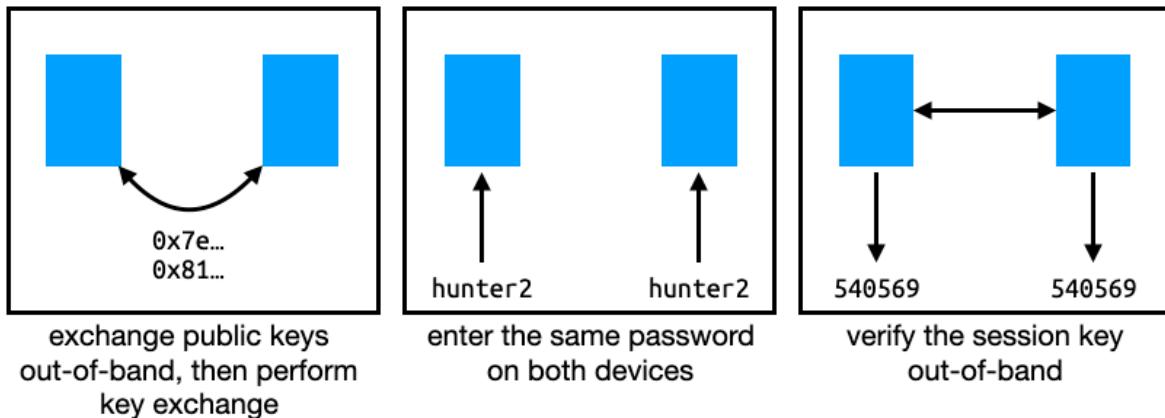


Figure 11.23 You've learned about three techniques to pair two devices: 1. a user can either help the devices obtain each other's public keys so that they can perform a key exchange 2. a user can enter the same password on two devices so that they can perform a symmetric password-authenticated key exchange 3. a user can verify a fingerprint of the key exchange after the fact, to confirm that no man-in-the-middle attacker intercepted the pairing.

11.4 Summary

- User authentication protocols (Protocols for machines to authenticate humans) often take place over secure connections, where only the machine (server) has been authenticated. In this sense, it upgrades a one-way authenticated connection to a mutually-authenticated connection.
- User authentication protocols make heavy use of passwords. Passwords have proven to be a somewhat practical solution that has been widely accepted by users, but has also led to many issues due to poor password hygiene, low entropy, and password database breaches.
- There are two ways to avoid having users carry multiple passwords (and possibly reuse passwords):
 - Password Managers, tools that users can use to generate (and manage) strong passwords for every applications they use.
 - Single-sign on (SSO), a federated protocol that allows a user to use one account to register and log into other services.
- A solution for servers to avoid learning about and storing their users' passwords, is to use an asymmetric password-authenticated key exchange (asymmetric PAKE). An asymmetric PAKE like OPAQUE allows users to authenticate to a known server using passwords, but without having to actually reveal their passwords to the server.
- Solutions to avoid passwords altogether are for users to use symmetric keys via one-time passwords (OTP) algorithms or use asymmetric keys via standards like FIDO2.
- User-aided authentication protocols often take place over insecure connections (WiFi, Bluetooth, NFC) and help two device to authenticate each other. To secure connections in these scenarios, user-aided protocols assume that the two participants possess an additional authenticated (but not confidential) channel that they can use (for example a screen on the device).
- Exporting a device public key to another device could allow strongly mutually-authenticated key exchanges to happen. These flows are unfortunately not very user-friendly, and sometimes not possible due to device constraints (no way to export or import keys).
- Symmetric password-authenticated key exchanges (symmetric PAKEs) like CPace can decrease the burden for the user to import a long public key, by only having to manually input a password in a device. Symmetric PAKEs are already used by most people to connect to their home WiFi for example.
- Protocols based on short authenticated strings (SAS) can provide security for devices that cannot import keys or passwords, but are able to display a short string after a key exchange has taken place. This short string has to be the same on both devices in order to ensure that the unauthenticated key exchange was not actively man-in-the-middled.

Crypto As In Cryptocurrency?

12

This chapter covers

- What consensus algorithms are.
- A tour of cryptocurrencies.
- How the Bitcoin and Libra cryptocurrencies work in practice.

The word "crypto" has been used to refer to cryptography as far as I can remember. Recently, I have seen its meaning quickly changing and being used by many people to refer to **cryptocurrencies**. Many cryptocurrency enthusiasts, in turn, seem to get more and more interested to learn about cryptography. Which makes sense, as cryptography is at the core of cryptocurrencies.

What's a cryptocurrency? It is two things:

- It's a **digital currency**. Simply: it allows people to transact some currency electronically. Sometimes a real currency is used (like the US dollar), sometimes a made up one is used (like Bitcoin). You already use digital currencies whenever you send money to someone on the internet. Indeed, you don't need to send cash by mail anymore.
- It's a currency that relies heavily on cryptography to avoid having to use a trusted third-party. In a cryptocurrency there is no central authority that one has to trust. We often talk about this property as **decentralization**, as in "we are decentralizing trust". Thus, as you will see in this chapter, cryptocurrencies are designed to tolerate a certain number of malicious actors, and allow people to verify their well-functioning.

Cryptocurrencies are relatively new, as the first successful experiment started in 2008 with the birth of **Bitcoin**. This was during the same year of a banking crisis that had impacted the world, and had eroded global trust in the financial system. At that time, many people started to realize

that the status quo was inefficient, expensive to maintain, and opaque to its people. The rest is history, and I believe this book is now the first cryptography book to include a chapter on cryptocurrencies.

In this chapter, I will shy away from debating on if we do need cryptocurrencies or not. Instead, I will focus on the technical aspect of cryptocurrencies. You will learn about what these cryptocurrencies are, and how they work. Let's get started!

12.1 A gentle introduction to byzantine fault tolerant consensus algorithms

Let's start from scratch. Imagine that you want to create a new digital currency. It's actually not too hard to get something that works, you could just setup a database on some dedicated server and provide an interface for people to query in order to let them send payments or check their balances. And to bootstrap it you could create some number, let's one million, of some fictive token and distribute it to your beta testers.

12.1.1 A problem of resilience - distributed protocols to the rescue

But such a simple system has obvious flaws. First, it's a **central point of failure**. If your server suddenly burst into flames everybody will lose their accounts. For this, there exist well-known protocols that can help replicate a database in real time on several servers (that can be distributed in various geographical locations). These algorithms are called **consensus algorithms** (and also referred to as "log replication", "state machine replication", or "atomic broadcasts") as they bring a number of machines in agreement as to what the mutating queries to the database are, and in what order they must be processed. Well-known consensus algorithms are PAXOS (published by Lamport in 1989) and its simplification RAFT (published by Ongaro and Ousterhout in 2013).¹

12.1.2 A problem of trust - decentralization helps

But we're left with another problem. Your system is a **trusted third party**. Others have to completely trust that your servers are doing the correct accounting. What it does is completely opaque to the users. Sure you could publish all of the queries (starting from the very first one) that you've applied to your databases. But what tells me you didn't show me a different history? For example, imagine the following scenario: I have 5 tokens left in my account. And let's say that after reading my book, you feel really good about helping me save some money. So I tell you to send 5 tokens to Alice, and 5 tokens to Bob. These are of course mutually exclusive transactions, but you could show Alice and Bob a different history making them both think that they have received the money. In the parlance this is called a **fork**.

This is a problem that should sound somewhat familiar with the problems of the web public key infrastructure that I've talked about in chapter 9. There, I mentioned Certificate Transparency, a gossip protocol that can **detect** such forks. The problem with money is that detection is not enough. To **prevent** forks, better consensus algorithms called **byzantine fault tolerant (BFT)**

consensus algorithms exist (introduced by Lamport et al. in 1982). BFT consensus algorithms resemble non-BFT consensus algorithms like PAXOS and RAFT, for the exception that the replicated databases (the participants of the protocol) do not blindly trust one another.

We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement.

– Lamport et al. *The Byzantine Generals Problem* (1982)

So here we have a solution to both our resilience and trust problems: as long as different participants run these BFT algorithms then they can continuously update the state with users payments, while policing each other to agree on new states. We say that the trust is now **decentralized**.

The first practical BFT consensus algorithm invented was, take a guess, **Practical BFT (PBFT)**, published in 1999. PBFT is a **leader-based algorithm**, similar to PAXOS and RAFT, where one leader is in charge of making proposals while the rest attempts to agree on these proposals.

Unfortunately PBFT is complex, slow, and doesn't scale well (passed a dozen participants). Today, Most modern cryptocurrencies use more efficient variants of PBFT. For example Libra, the cryptocurrency introduced by Facebook in 2019, is based on HotStuff which is a PBFT-inspired protocol.

12.1.3 A problem of censorship - permissionless networks

One limitation of these PBFT-based consensus algorithms is that they all require a known and fixed set of participants. Actually more than that, passed a certain number of participants, they start breaking apart.

So how does a cryptocurrency decide who the consensus participants are? There are several ways but the two most common ways are:

- **Proof of authority (PoA).** You just decide who are the ones that will be in charge of participating in consensus. Pretty much by listing the public keys of the participants.
- **Proof of stake (PoS).** Since we have a currency, we can also rely on the assumptions that the richest participants will have no incentive to act against the network (as it could reduce trust in it and thus lower the value of the tokens they hold).

But what if you want to avoid having a clear list of participants? To avoid them being targeted, by users but also by some countries' regulations. All of this changes in 2008, when the **Bitcoin** paper is published (under the anonymous pseudonym Satoshi Nakamoto) and introduces the **Nakamoto consensus**, the first practical solution to make a cryptocurrency censorship resistant.

Bitcoin did this by allowing anyone to become a leader (someone who can propose new transactions) by finding pre-images of specific (SHA-256) hashes. This is known as **proof of work (PoW)**. (Note that Bitcoin is not considered a BFT algorithm as it allows forks.)

This leaderless consensus allowed Bitcoin to create what we call a **permissionless** network, in which anyone can participate in its consensus protocol. This is in contrast to **permissioned** networks that have a fixed set of participants.

Bitcoin is not the only such censorship resistant cryptocurrency. BFT consensus algorithms were pushed to their limits when **Algorand**, in 2017, published a way to dynamically obtain a set of participants (with a leader) based on the stake (the amount of Algo token) they have in the cryptocurrency. (Which led them to call themselves a pure proof of stake protocol.)

Recap in figure 12.1.

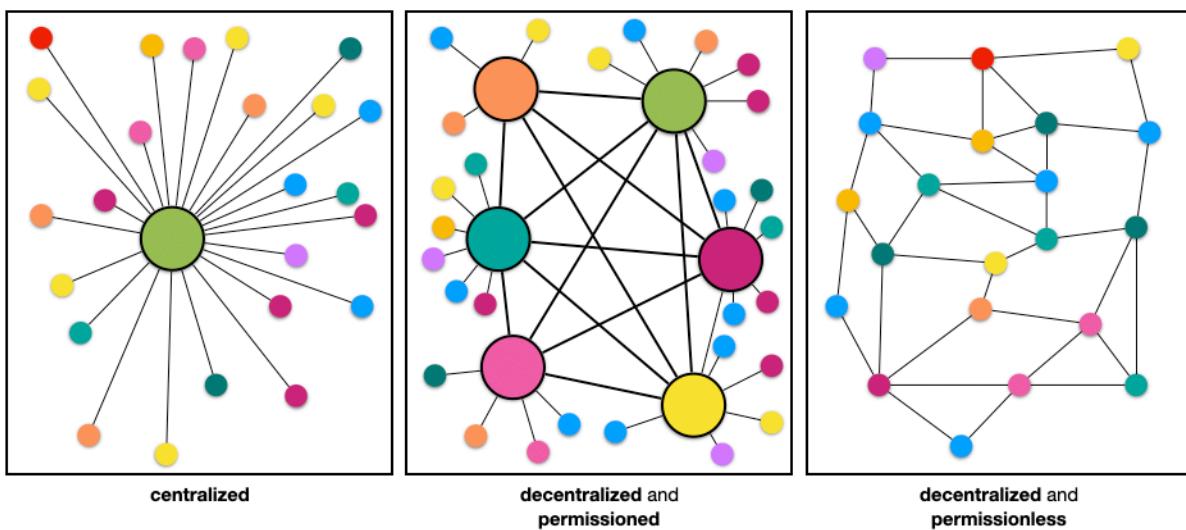


Figure 12.1 A centralized network relies on a single point of failure, whereas a decentralized network is resilient to a number of servers shutting down. A permissioned network has a known and fixed set of main actors, while in a permissionless network anyone can participate.

12.2 How does Bitcoin work?

On October 31st, 2008, an anonymous researcher(s) published "Bitcoin: A Peer-to-Peer Electronic Cash System" under the pseudonym Satoshi Nakamoto. Not long after, they released the bitcoin core client, a software that anyone can run to join and participate in the Bitcoin network. That's the only thing that Bitcoin needed: enough users that run the same software (or at least the same algorithm). The first ever cryptocurrency was born. The Bitcoin (or BTC). To this day, it remains unknown who Satoshi Nakamoto is.

Bitcoin is a true success story, the cryptocurrency has been running stably for more than a decade now, and has allowed users from all around the world to transact with the digital

currency. In 2010, Laszlo Hanyecz a developer bought two pizzas for 10,000 BTC. 7 years later, the price of one BTC had reached \$20,000. At the moment of this writing (June 2020) a bitcoin is worth a bit less than \$10,000. Thus, one can already take away that cryptocurrencies can be highly volatile.

12.2.1 Accounts and transactions

Let's first look at how Bitcoin handles user accounts and balances. The user experience makes direct use of cryptography, by having users create a "wallet" by generating an ECDSA keypair. In chapter 7 I mentioned that Bitcoin uses the secp256k1 curve with ECDSA (not to mix with NIST's P-256 curve).

Thus, accounts in Bitcoin are just public keys:

- To spend your bitcoins, you sign a transaction with your private key.
- To receive bitcoins, you can share your public key to others.

Thus, the protection of one's holdings is directly linked to their private key. And as you know, key management is hard, which has led to loss of keys that have cost millions of dollars.

Note that there exist different types of transactions in Bitcoin, and most of the transactions seen on the network actually hide the recipient public keys by hashing them. In these cases, the hash of public keys are referred to as "addresses" and hide the public key of an account until the account decides to spend its bitcoins. Indeed, you must reveal the public key behind an address so that others can verify your signed transaction.

The fact that different types of transactions exist is an interesting detail of Bitcoin: transactions are actually scripts written in a made-up and quite limited language. When a transaction is processed, the script is actually run and the output determines if the transaction is valid or not. Cryptocurrencies like Ethereum have pushed this scripting idea to the limits by allowing much more complex programs to run when a transaction is executed.

There are two things here that I didn't touch on:

1. What's in a transaction?
2. What does it mean for a transaction to be executed?

I will explain the second item in the next section, so let's look at what's in a transaction for now.

A transaction consists of several inputs and several outputs. This is a particularity of Bitcoin: there is no real database of account balances. Instead, a user has pockets of bitcoins spread around that are available to the user, and which are called **Unspent Transaction Outputs (UTXOs)**.

Figure 12.2 gives an example of how UTXOs are used in transactions.

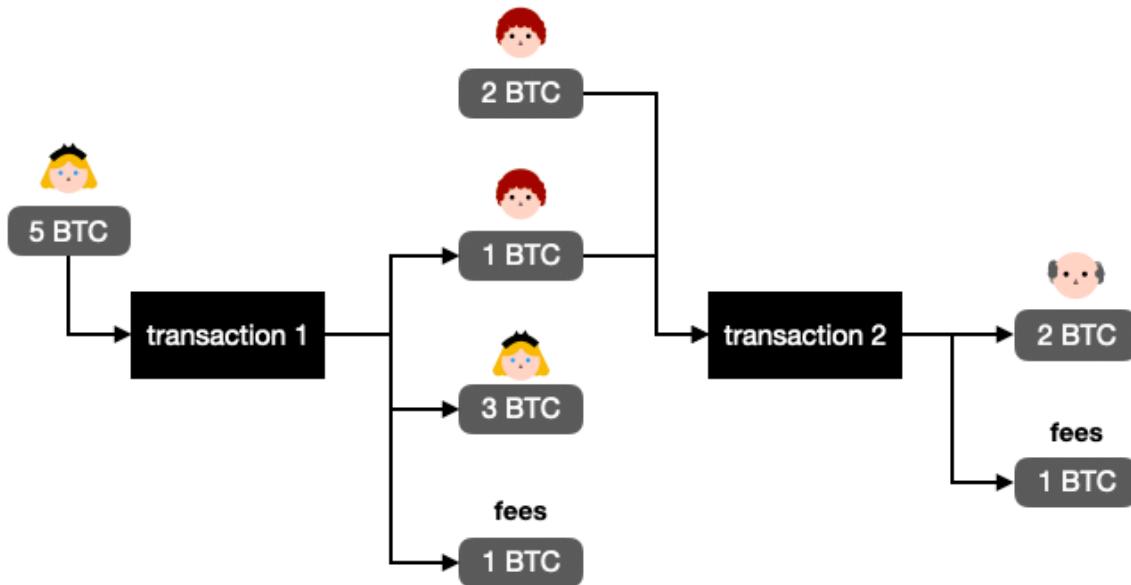


Figure 12.2 Transaction 1 is signed by Alice and transfers 1 BTC to Bob. Since it uses a UTXO of 5 BTC, it needs to also send back the change to Alice, as well as reserve some of that change as fees. Transaction 2 is signed by Bob and combines two UTXOs to transfer 2 Bitcoin to Felix. Note that in reality fees are much much lower.

12.2.2 Mining and Proof of Work and Forks

OK. You now understand what's in a transaction, and how you can manage your account or figure out someone's balance. But who keeps track of all these transactions? Everyone! Indeed, using Bitcoin means that every transaction must be publically shared and recorded in history. Bitcoin is an **append-only** ledger, a book of transactions where each page is connected to the previous one. Since every transaction is public, the only notion of anonymity is that it might be hard to figure out what public key is linked to who in real life.

One can easily inspect any transaction that has happened since the inception of Bitcoin by downloading a Bitcoin client and downloading the whole history. You can more easily inspect transactions by using a "blockchain explorer" (for as long as you trust an online service). See figure 12.3.

Inputs

Index	Address	Pkscript	Details	Output
0	34wdNyp6JfWo3qpva8QsJKpEghCinjDeqh	OP_HASH160 23abcaa80bf33e2f40c7c01b3b92140ea36aff56 OP_EQUAL	Value	1.97647532 BTC
Sigscript	00200cc2437e008d5d64cfe710856936a19b10743af577621701715c0f5c16c49ca			
Witness	304502210096d4ef173813a382cab60712c2270f71a2124cc10544e9cb14c162479686ee0702201fb15c6d6695983d7c44c1a5c062797a9d280f9e49b4879a45 beee9e0f4a4f59401 3045022100858dab609b8aa4c65bfc0d2fd5e39c7eff135894098140c40c9bed69d12d622802201c3e6f33a048b71c7a025cb76c21ac46beab0ef3a1cc575a93 7f0b032df1b87b01 522103d3a1c7abd8b15a9e91f2f540443611081d0c5419d2b968f9915497945343c30210265b6af9b40d2ae3769471d6cea0f2d84f9a03ba741fa3271f3c75e7 8ee2eba752ae			

Outputs

Index	Address	Pkscript	Details	Unspent
0	13W7jmZWZvZVGQ1oPy7kHVbN63izqkpRIE	OP_DUP OP_HASH160 1b7f104474dd62dd2716512f856f39769a5ef77 OP_EQUALVERIFY OP_CHECKSIG	Value	0.00901295 BTC
1	34wdNyp6JfWo3qpva8QsJKpEghCinjDeqh	OP_HASH160 23abcaa80bf33e2f40c7c01b3b92140ea36aff56 OP_EQUAL	Value	1.96741869 BTC

Figure 12.3 A random transaction I chose to analyze on blockchain.com/btc/tx/0f604d2d290798f0b7b1270875bdbc4e1fd00ae755cb3b57c15adb0014c58d90. The transaction uses one input (of around 1.976BTC) and splits it in two outputs (of around 0.009BTC and 1.967BTC). The other fields are scripts written using Bitcoin's scripting language in order to make the transaction work.

If you want to know how much bitcoins a user has, you need to download the whole history of transactions (the ledger) and add all the money that was received, and subtract all the money that was sent out. What is left are a number of UTXOs which represent the total amount of bitcoin a user possesses. In practice, a UTXO is some amount of bitcoin linked to a bitcoin script and addressable by a tuple (transaction id, output id).

Again, Bitcoin is really just a list of all the transactions that have been processed since the very beginning (we call that the genesis) up until now.

This should make you wonder:

1. Who is in charge of ordering all these transactions?
2. How can you and others verify that this ordering is valid, and not another one?

Let me answer both questions here.

In order to agree on an ordering of transactions, we let anyone make a proposal for what are going to be the next transactions to be processed (the next page in the ledger if you will). This proposal of transactions is called a **block** in Bitcoin's term. But letting anyone propose a block is a recipe for disaster, there are a lot of participants in Bitcoin. Instead, we want just one person to make a proposal for what is the next block of transaction. To do this, one major idea behind Bitcoin is to make everybody work on some probabilistic puzzle, and only let the person who solves it propose a block. This is called **proof of work (PoW)** and the puzzle is to find a block that hashes to a digest smaller than some value (in other words, a digest which binary representation starts with some known number of zeros).

In addition, this block must contain the hash of the previous block. Hence the Bitcoin ledger is really a succession of blocks, where each block refers to the previous one, down to the very first block: the genesis block. This is what Bitcoin calls a **blockchain**. See figure 12.4 for an illustration.

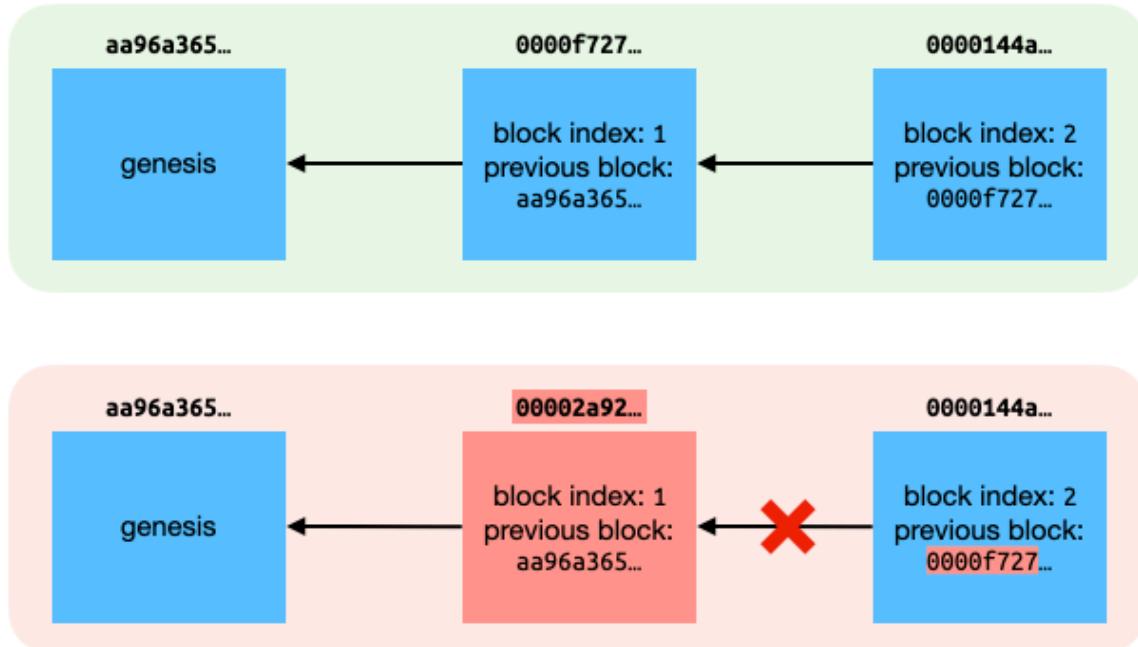


Figure 12.4 A blockchain is a series of blocks where each block refers to the previous one (top diagram). Changing the content of one of the blocks in the blockchain breaks the chain as the next block in the series does not point to the new hash.

Of course, you wouldn't want to really change the content of your block in order to change its hash. Instead, you fix all fields (the hash of the previous block, the list of transactions, etc.) blocks in Bitcoin contain a nonce, which you can randomly generate until you get the desired digest. I illustrate this in figure 12.5.

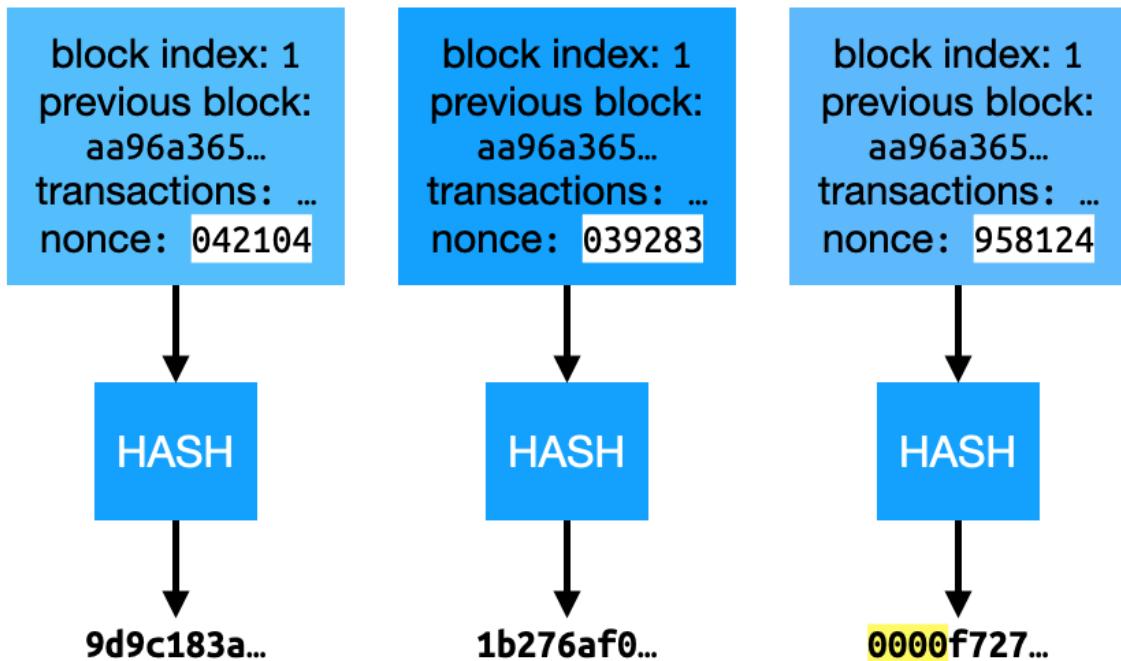


Figure 12.5 To mine, a miner can fix every field of a block (like the block number, the hash of the previous block, transactions to include) and randomly generate a nonce until the hash of the block is a value smaller than the current difficulty target.

All of this works because everyone is running the same protocol using the same rule. When you synchronize with the blockchain, you will download every block from other peers and verify that:

1. hashing a block indeed gives a digest that is smaller than some expected value.
2. the next block refers back to this block.

Not everyone has to propose blocks, but you can if you want. If you do so, you are called a **miner**. And if you find a block, you collect:

- A **reward**. That is a fixed number of bitcoins that will get created and sent to your address. In the beginning, miners would get 50 BTC per block mined. But the reward value halves every 210,000 blocks, and will eventually be reduced to 0, capping the total amount of bitcoin that can be created to 21 millions.
- All the **transaction fees** in it. This is why increasing the fees in your transactions allows you to get them accepted faster, as miners will tend to include transactions with higher fees in the block they mine.

This is how users of Bitcoin are incentivized in making the protocol advance. A block contains what is called a **coinbase**, which is the address that will collect the fees. All bitcoins in history have at some point been created as part of one of these mining events.

12.2.3 Forking hell!

And so, bitcoin distributes the task of choosing the next set of transactions to be processed via this proof-of-work-based system.

Your voting power is directly correlated to the amount of computation you can deliver, thus a lot of computation power nowadays is directed at mining blocks in Bitcoin (or other proof-of-work-based cryptocurrencies).

NOTE

Proof of work can be seen as Bitcoin's way of addressing **sybil attacks**, which are attacks that take advantage of the fact that you can create as many accounts as you want in a protocol, giving you an asymmetric edge to honest participants. In bitcoin, your only way to obtain more voting power is really to buy more hardware to compute hashes.

There is one problem though, the difficulty of finding a hash that is lower than some value cannot be too easy. If it is too easy, then the network will have too many participants mining a valid block at the same time. This is essentially what we call a **fork**, and this will create problems: how do you decide which block is the legitimate next block to be added to the blockchain?

To solve this forking issue, Bitcoin has two mechanism:

1. **Maintaining the hardness of proof of work.** If blocks get mined too quickly (or too slowly), the algorithm (that everyone is running) will adapt and increase (or decrease) the **difficulty** of the proof of work. In other words, the value you need your block's hash to be below to, will decrease (or increase) over time. Imagine that the block's hash needs to start with a 0 byte, you are expected to try 2^8 different nonces until you can find a valid block. Raise this to 2 bytes, and you are now expected to try 2^{16} different nonces. The algorithm aims for a block to be mined every 10 minutes.
2. **Relying on the chain with the most amount of work.** The 2008 Bitcoin paper stated "the longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power", dictating that participants should honor what they see as the longest chain. The protocol was later updated to follow the chain with the highest cumulative amount of work.² I illustrate this in figure 12.6.

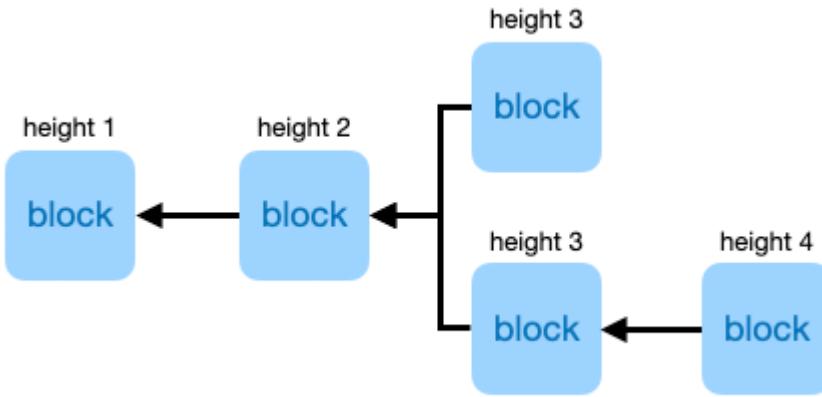


Figure 12.6 A fork in the blockchain: two miners have published a valid block at height 3. Later, another miner mines a block at height 4 that points to the second block at height 3. As the second fork is now longer, it is the valid fork that miners should continue to extend.

I've said previously that the consensus algorithm of Bitcoin is not a byzantine fault tolerant protocol, this is because it allows such **forks**. Thus, if you are waiting for your transaction to be processed, you should absolutely not rely on observing your transaction get included in a valid block! Because the observed block could actually be a fork, and a losing one at that (to a longer fork), you need more assurance. For this reason most wallets and exchange platforms will wait for a number of "confirmation" blocks to be mined on top of your block. This number is typically six blocks, which makes confirmation time around an hour.

But worse, Bitcoin does not provide much assurance that a fork passed 6 blocks would never happen. If the difficulty is well-adjusted, then it should be fine, and we have reason to believe that this is true for Bitcoin. Bitcoin's proof of work difficulty has increased gradually over time, as the cryptocurrency was getting more and more popular. So much, that most people cannot mine on their own and get together in what are called "mining pools" to distribute the work needed to mine a block (they then share the reward).

With block 632874 [...] the expected cumulative work in the Bitcoin blockchain surpassed 2^{92} double-SHA256 hashes.

— Pieter Wuille on June 4th 2020 on twitter
twitter.com/pwuille/status/1268467586422853641

For new cryptocurrencies based on proof of work, starting with a difficulty too low is a real issue. Imagine the following scenario: you wait for Alice to send you the 5 bitcoins she has in her account. Finally, you observe a block including her transaction. You decide to wait for 6 more blocks to be added on top of it, and call it a day, giving her the last painting you created. But 12 blocks later, a longer fork appears from nowhere, and is thus accepted by everyone as the legitimate branch over the one you previously saw. Except that this new branch doesn't include Alice's transaction. Instead it includes a transaction moving all of her funds to another address,

preventing you from re-publishing her transaction to get the bitcoins. She effectively **double spent** her money. We call this a **51% attack**.³ The name comes from the amount of computation power Alice needed to perform the attack: she needed more than everyone else.

This is not just a theoretical attack, 51% attacks have happened in the real world! For example, in 2018 an attacker managed to double spend a number of funds in a 51% attack on the Vertcoin currency.

The attacker essentially rewrote part of the ledger's history and then, using their dominant hashing power to produce the longest chain, convinced the rest of the miners to validate this new version of the blockchain. With that, he or she could commit the ultimate crypto crime: a double-spend of prior transactions, leaving earlier payees holding invalidated coins.

— Michael J. Casey *Vertcoin's Struggle Is Real: Why the Latest Crypto 51% Attack Matters* (Dec 2018)

In 2019, the same thing happened to Ethereum classic (a variant of Ethereum), causing losses of more than \$1 million at the time, with several **reorganizations** (dropping a branch for another one) of more than 100 blocks of depth. In 2020, Bitcoin Gold (a variant of Bitcoin) also suffered from a 51% attack, removing 29 blocks from the cryptocurrency's history and double spending more than \$70,000 in less than two days.

12.2.4 Reducing a block's size by using merkle trees

One last interesting aspect of Bitcoin that I want to talk about is how it compresses some of the information available. Blocks in Bitcoin actually do not contain any transactions, but rather a digest that authenticates all the transactions that are included in the block. Transactions are actually shared separately, and the digest is the root hash of a **merkle tree**. What's a merkle tree? It's a tree structure where internal nodes are hashes of the concatenation of their children. By publishing the root of the tree, one can authenticate all of the leaves. I illustrate this in figure 12.7.

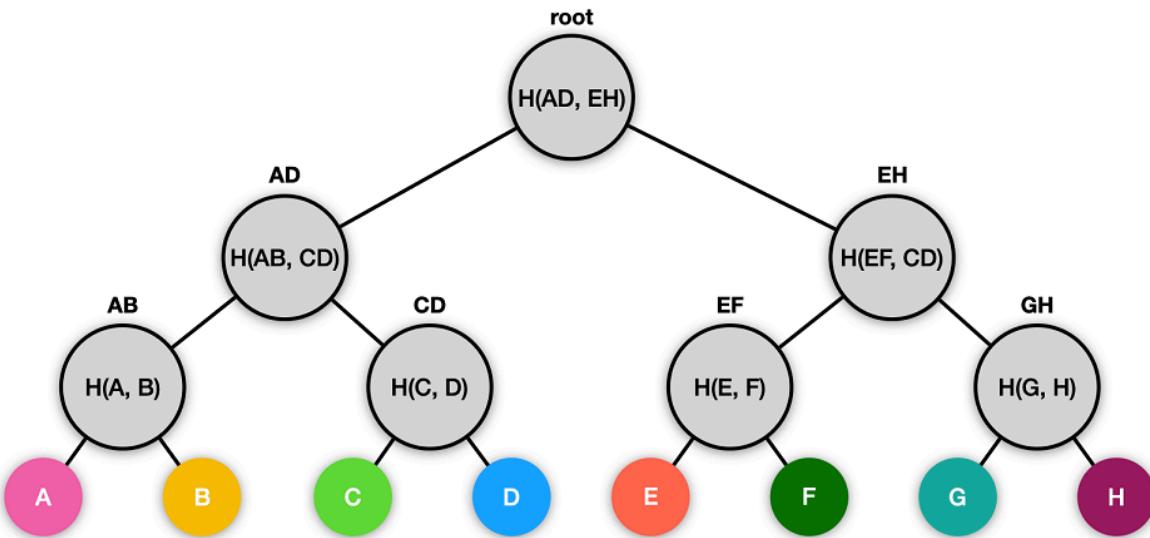


Figure 12.7 A merkle tree, a data structure that authenticates a number $2n$ of leaves for n the depth of the tree. In a merkle tree, internal nodes are the hashes of their children.

Merkle trees are very useful structures and you will find them in all types of real world protocols. They can be thought of as cryptographic accumulators, as they compress an unlimited amount of data into a fixed size value (the root of the tree). A merkle tree can provide **proofs of membership** that are logarithmic in the depth of the tree in size: if you know the root of the merkle tree, and want to know if a leaf is in the tree, I can share with you the neighbor nodes in the path up to the root as a membership proof. What's left for you is to compute the internal nodes up to the root of the tree by hashing each pair in the path. It's a bit complicated to explain this in writing so I illustrate this in figure 12.8.

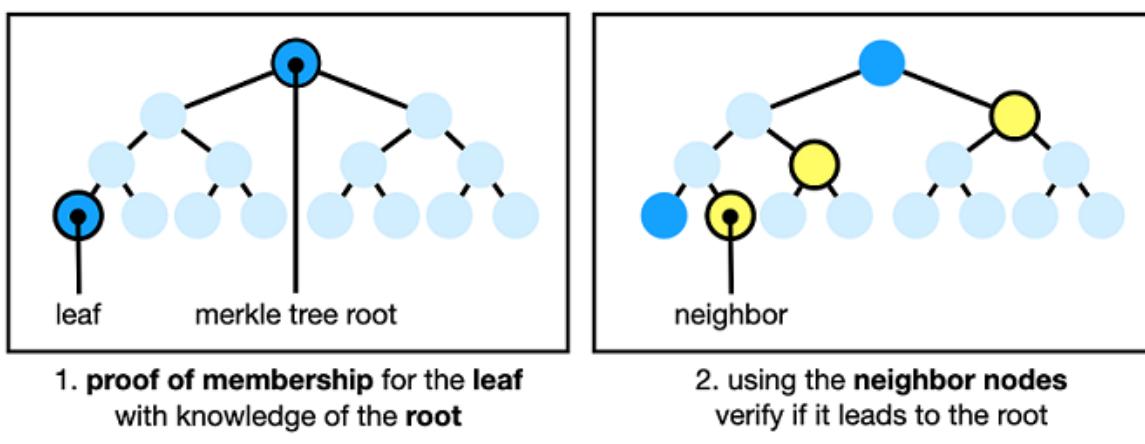


Figure 12.8 To prove that a leaf belongs in a merkle tree, one can simply publish the neighbor nodes in the path from the leaf to the root. A verifier can then use these neighbor nodes to compute the hash of all the nodes in the path to the root, and verify that the last node is indeed the known root hash.

The reason for using merkle trees in a block is to lighten the information that needs to be downloaded in order to perform simple queries on the blockchain. For example, imagine that you

want to check that your recent transaction has been included in a block. You could download all the recent blocks with all of their transactions, but you don't need to. Instead, you can download the block headers (which are lighter) and ask a peer to tell you what block includes your transaction, and to give you a proof of that.

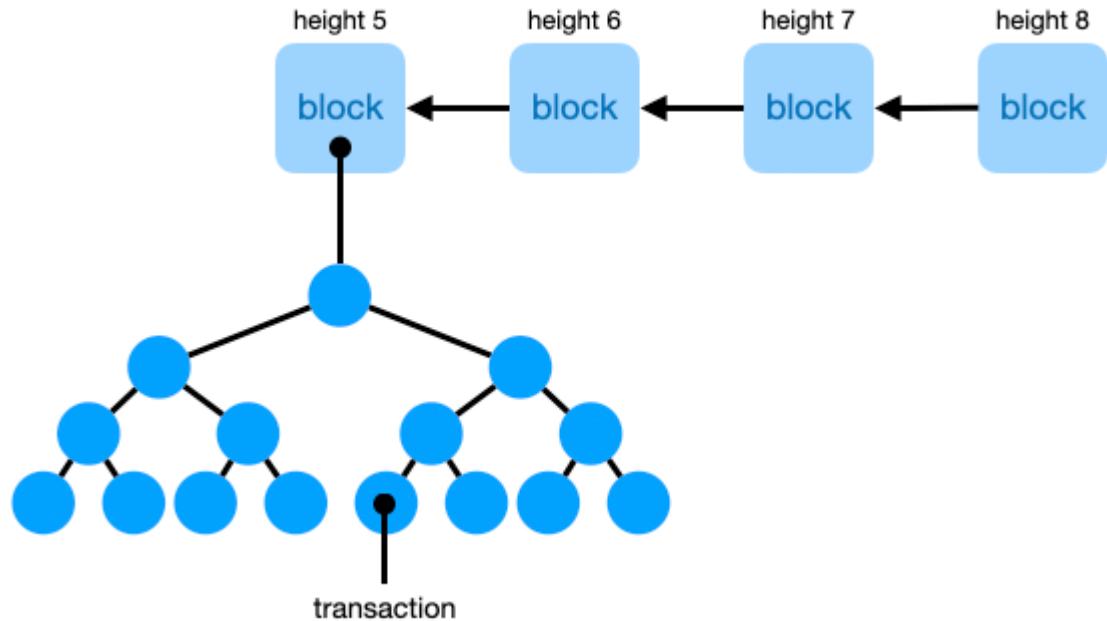


Figure 12.9 To verify that a transaction is part of a block, one only needs to download the transaction merkle tree root hash from the relevant block, as well as a proof.

12.3 A tour of cryptocurrencies

Bitcoin is the first successful cryptocurrency, and has remained the cryptocurrency with the largest market share and value in spite of hundreds of other cryptocurrencies being created. What's interesting is that Bitcoin had, and still has, many issues that other cryptocurrencies have attempted to tackle (and some with success). Even more interesting, the cryptocurrency field has made use of many cryptographic primitives that until now did not have many practical applications, or did not even exist.

So without further ado, here is a list of issues that have been researched since the advent of Bitcoin.

Volatility. Most people currently use cryptocurrencies as speculation vehicles. The price of Bitcoin obviously helps that story, as it has shown that it can easily move thousands of dollars up or down in a day only. Some people claim that the stability will come over time, but it remains that Bitcoin is not usable as a currency nowadays. Other cryptocurrencies have experimented with the concept of **stablecoin**, by pegging the price of their token to a real currency (for example the US dollar).

Latency. Throughput (transactions per second) in Bitcoin is quite low and finality (the time it takes for your transaction to get processed) is quite high. I mentioned that the protocol does not guarantee that forks (and thus double spending) won't happen, and that the current solution is to wait for a number of confirmation blocks before crying victory over a processed transaction. In practice, this takes an hour for Bitcoin. The fact that the miner who will successfully propose the next block is not known in advance is also a big challenge to the efficiency of the protocol. The solution to these speed issues can be solved by BFT protocols which usually provide finality of mere seconds with an insurance that no forks are possible. Yet, this is sometimes not enough as well, and different technologies are being explored. So-called **layer 2 protocols** attempt to provide additional solutions that can enact faster payments off-chain, while saving progress periodically on the main blockchain (referred to as the layer 1 in comparison).

Blockchain size. Another common problem with Bitcoin and other cryptocurrencies is that the size of the blockchain can quickly grow to impractical sizes. This creates usability issues when participants are expected to download the entire chain in order to interact with the network (for example to query their account's balance). In March 2020 the Bitcoin blockchain was almost 300GB. BFT-based cryptocurrencies that can process much more transactions per second are expected to easily reach Terabytes of data within months or even weeks. There exist several attempts at solving this, notable ones include cryptocurrencies like **Coda**, that make use of **recursive zero-knowledge proofs**. It is enough to download the latest state of the chain, along with a fixed-size zero-knowledge proof of 22KB, to join the network. The zero-knowledge proof proves that the current state correctly came from applying a set of transactions to the previous block, and that the previous block's zero-knowledge proof was correct as well. In turn the previous block's zero-knowledge proof proved that the transition from the grandparent block was correct and that the previous zero-knowledge proof was correct. This all the way up to the genesis.

Confidentiality. Bitcoin provides pseudo-anonymity, in that accounts are only tied to public keys, and as long as nobody can tie a public key to a person the account is anonymous. Note that all the transactions from and to that account are public though, and social graphs can be created to understand who tends to trade more often, and who owns how much of the currency. There are many cryptocurrencies that attempt at solving these issues using zero-knowledge proofs and other techniques. **Zcash** is the most well-known confidential cryptocurrency, as a zcash transaction can encrypt its sender, receiver, and amount fields. It prevents double spending and the creation of money from nowhere by using zero-knowledge proofs as well.

12.4 How does Libra work?

Libra is a digital currency initially created by Facebook, and now under the roof of the Libra association, an organization grouping a number of companies, universities, and nonprofits looking to push for an open and global payment network. One particularity of Libra is that it is backed by real money, using a reserve of different currencies like the US dollar or the British pound. This allows the digital currency to be stable, unlike its older cousin Bitcoin.

To run the payment network in a secure and open manner, a BFT consensus protocol called LibraBFT is used, which is a variant of HotStuff.⁴

In this section, let's see how LibraBFT works.

12.4.1 LibraBFT: a **byzantine fault tolerant consensus protocol**

A byzantine fault tolerant consensus protocol is meant to achieve two properties in the presence of malicious participants:

- **Safety.** This property states that no contradicting states can be agreed on, essentially this means that forks are not supposed to happen (or with a negligible probability).
- **Liveness.** This property states that whenever people submit transactions, the state will end up including them. In other words, nobody can stop the protocol from doing its thing.

Note that a participant is malicious (also called **byzantine**) if they do not behave according to the protocol.

Safety is usually a property that is quite straightforward to achieve, but liveness is known to be more difficult. Indeed, there's a well-known impossibility result linked to BFT protocols dating from 1985,⁵ which states that no **deterministic** consensus protocol can tolerate failures in an **asynchronous** network. Most BFT protocols avoid this impossibility result by considering the network somewhat **synchronous** (and indeed, no protocol is very useful if your network goes down for a long period of time) or by introducing randomness in the algorithm.

For this reason LibraBFT never forks, even under extreme network conditions. In addition it always makes progress as long as the network stabilizes.

Libra runs in a permissioned setting where participants—called **validators**—are known in advance. The protocol advances in strictly increasing **rounds**, during which validators take turns to propose **blocks** of transactions. In each round:

1. The validator that is chosen to lead collects a number of transactions, groups them into a block extending the blockchain, then signs it and sends it to all other validators.
2. Other validators upon receiving the block can vote for it by signing it and sending the signature to the next leader.

If enough votes have been gathered (more on that later), the next leader can then create what is called a **quorum certificate (QC)** that will contain all these signatures. This is how a block extends the blockchain: by including a QC that gathers a number of votes on a previous one.

Note that if validators do not see a proposal during a round, they can timeout and warn other validators that nothing happened. In this case the next round is triggered and the proposer can use the highest QC that they have.

I illustrate this in figure 12.10.

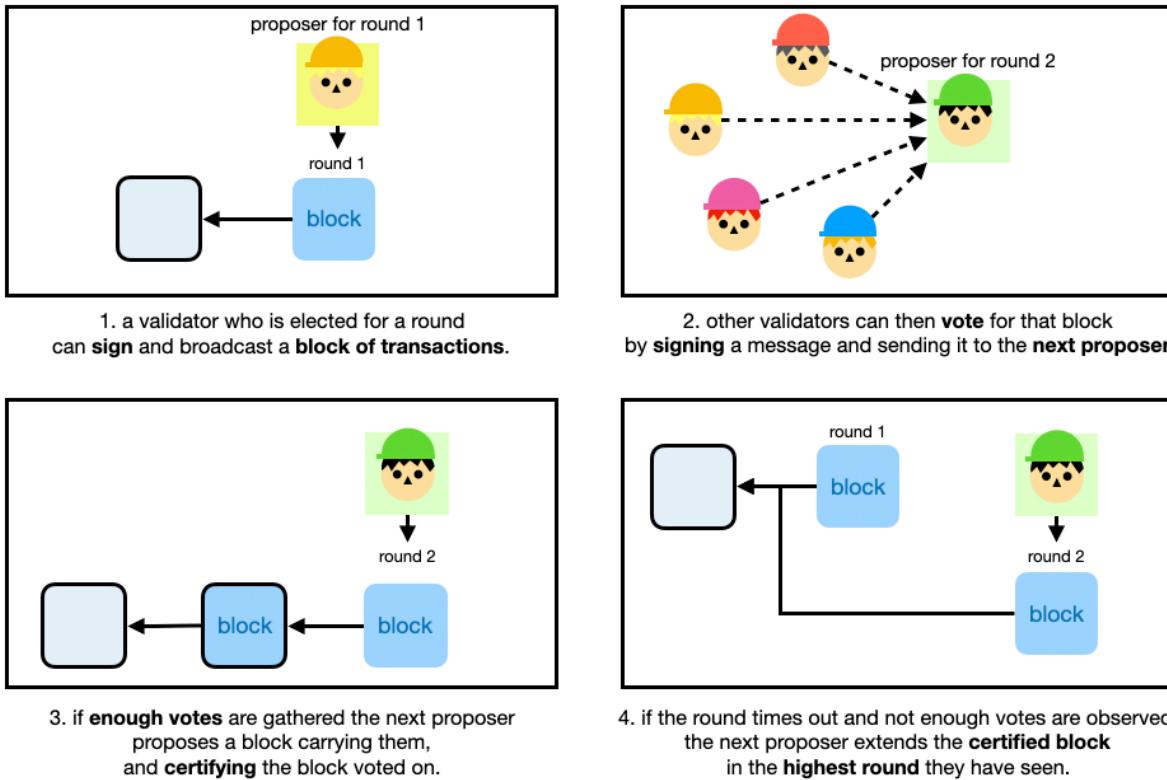
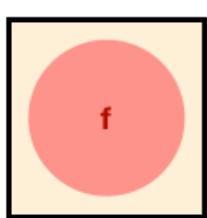
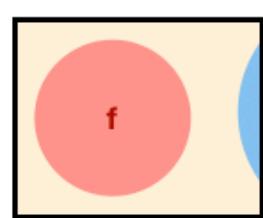


Figure 12.10 Each round of LibraBFT starts with the designated leader proposing a block that extends the last one they've seen. Other validators can then vote on this block by sending their vote to the next round's leader. If the next round's leader gathers enough votes to form a quorum certificate (QC), they can propose a new block containing the QC, effectively extending the previously seen block.

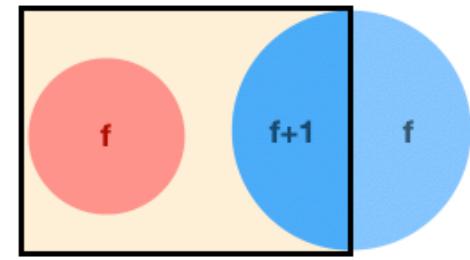
If $3f+1$ is the number of validators, then LibraBFT tolerates up to f malicious validators (even if they collude). As long as this assumption is true, the protocol provides safety and liveness.⁶ With that in mind, QCs can only be formed with a majority of honest validators' votes, which is $2f+1$ signatures (as explained in figure 12.11).



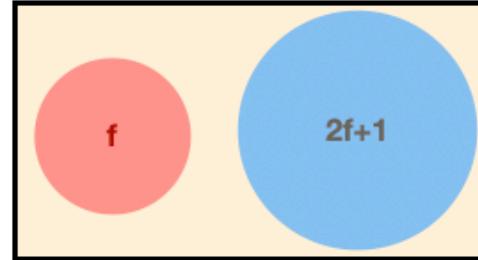
if f have voted for a block,
then they could be **all malicious votes**



if $f+1$ have voted for a block,
then **at least one honest node voted**



if $2f+1$ (a **quorum**) have voted for a block,
then **at least $f+1$ honest nodes voted**



if $3f+1$ have voted for a block,
then **everybody has voted**

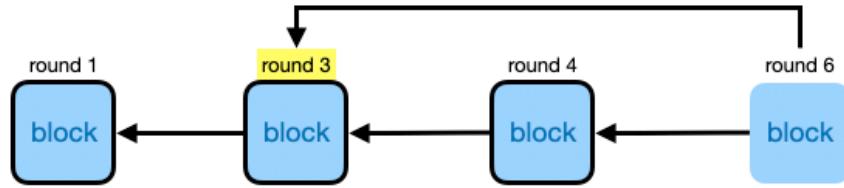
Figure 12.11 The LibraBFT protocol provides safety and liveness as long as the protocol does not exceed f malicious validators, if $3f+1$ is the number of validators. In other words, two thirds of the validators must be honest for the protocol not to fork. A quorum is a set of $2f+1$ votes, as it is the lowest number of votes that can represent a majority of honest validators.

There's two voting rules that must be followed at all times by validators:

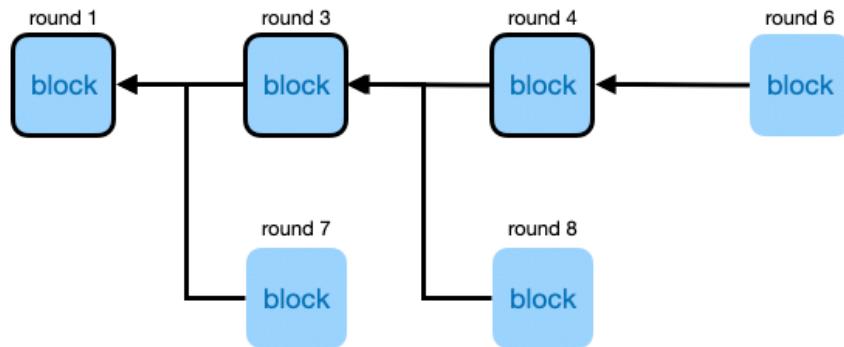
1. Never vote in the past. For example, if you vote for round 3, you can then only vote for round 4 and more.
2. Always vote for a block extending a block at your **preferred round** or higher.

What's a preferred round? By default it is 0, but if you vote for a block that extends a block that extends a block, and by that I mean you voted for a block that has a grandparent block, then that grandparent block's round becomes your preferred round (unless your previous preferred round was higher).

Complicated? Check figure 12.12.



if I vote for the block at round 6, my **preferred round** becomes round 3



then, I **cannot vote** for the block at round 7, but I **can vote** for the block at round 8

Figure 12.12 After voting for a block, a validator sets their preferred round to the grandparent block's round if it is higher than their current preferred round. Since block F's grand parent block A is at round 1, voting for block F at round 8 does not replace your preferred round (set to 3).

There's only two voting rules!

One last thing, votes that are certified are not finalized yet, or as we say **committed**. That means that nobody should assume that the transactions they contained have been processed.

The commit rule works as follows: whenever **three blocks** follow each other at **contiguous rounds**, the oldest block (and all of the blocks before it) get committed as soon as the most recent block gets certified.

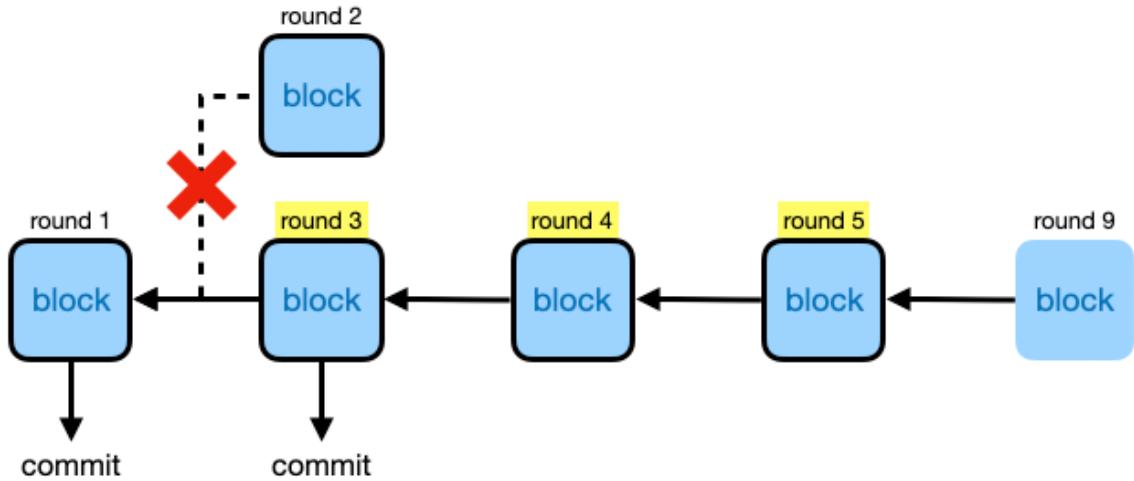


Figure 12.13 Three contiguous rounds (3, 4, 5) happen to have a chain of certified blocks. Any validator observing the certification of the last block in round 5 (by the QC of round 9) can commit the first block of the chain, as well as all of its ancestors (here the block of round 1). Any contradicting branches (for example the block of round 2) also get dropped.

And this is all there is to the protocol at a high level. But of course, once again, the devil hides in the details.

While I encourage you to go read the one-page safety proof on the LibraBFT paper,⁷ I want to use a couple pages here to give you an intuition on why it works.

First, we notice that two different blocks cannot be certified during the same round (see figure 12.14). From this we can simplify how we talk about blocks: block 3 is at round 3, block 6 is at round 6, and so on.

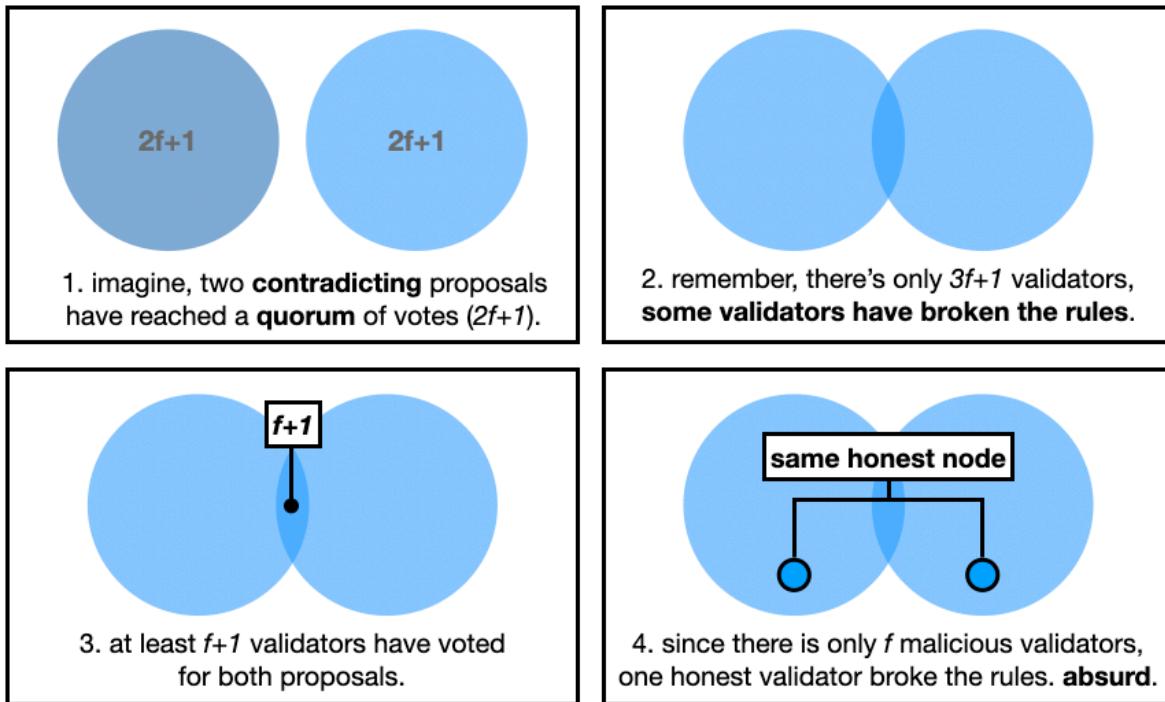


Figure 12.14 Assuming that there can only be up to f malicious validators in a protocol of $3f+1$ validators, there can only be one certified block per round.

Now, take a look at figure 12.15 and take a moment to figure out why a certified block, or two certified blocks, or three certified blocks at non-contiguous rounds, do not lead to a commit.

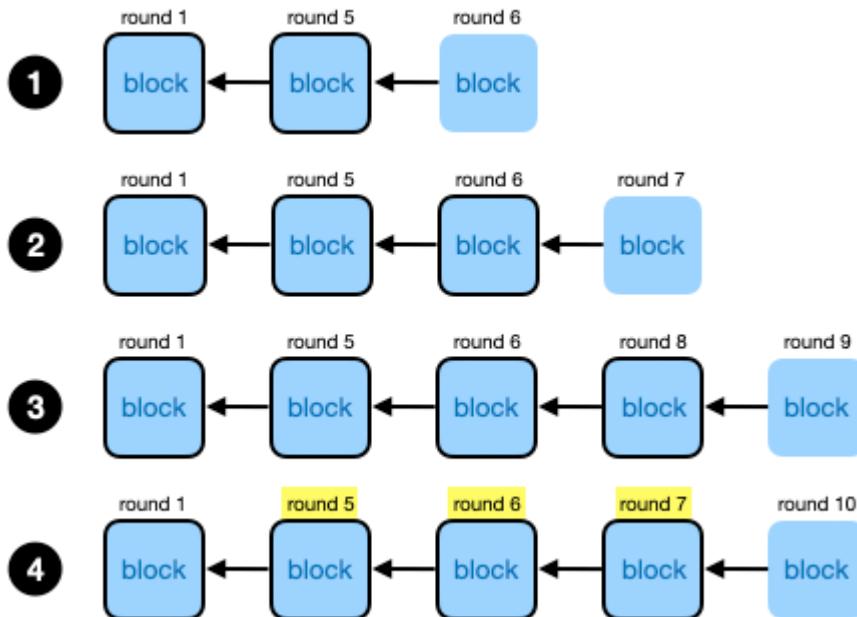


Figure 12.15 In all these scenarios, only scenario number 4 ends up committing block 5. Can you tell what can lead to validators not committing block 5 in the other scenarios?

Did you manage to find out answers for all the scenarios?

The short answer is that all scenarios, for the exception of the last one, leave room for a block to extend round 1. This late block effectively branches out and can be further extended according to the rules of the consensus protocol. If this happens, block 5 and other blocks extending it, will get dropped.

For scenario 1 and 2, this can be due to the proposer not seeing the previous blocks. In scenario 3 an earlier round's block appears later than expected, perhaps due to network issues or worse due to a validator withholding it up to the right moment.

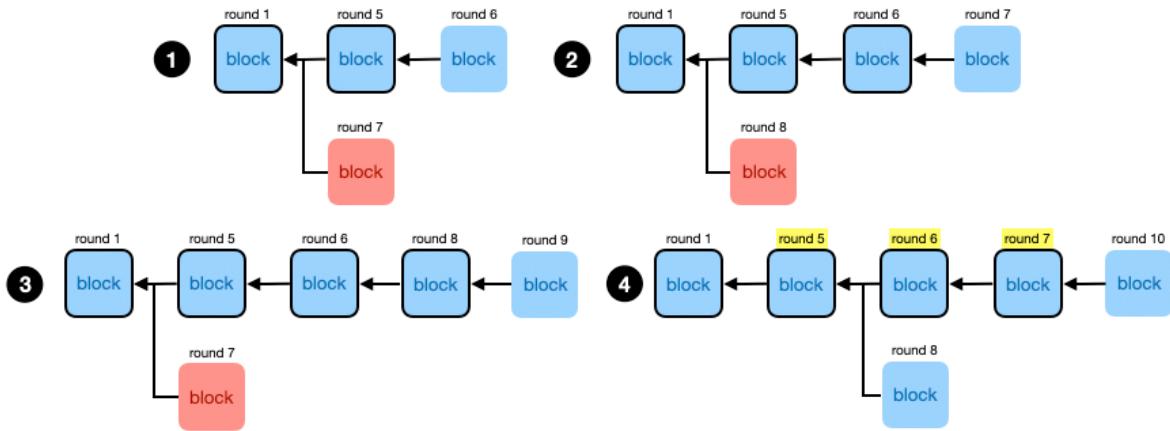


Figure 12.16 Building on figure , all scenarios except for the last one allow for a parallel chain that can eventually win and discard the branch of block 5. The last scenario has a chain of three certified blocks in contiguous rounds. This means that block 7 has had a majority of honest voters, who in turn updated their preferred round to round 5. This means that after that, no block can branch out before block 5 and obtain a QC at the same time. The worst that can happen is that a block extends block 5 or block 6, which will eventually lead to the same outcome: block 5 being committed.

12.5 Summary

- Cryptocurrencies are about decentralizing a payment network to avoid a single point of failure.
- To have everyone agree on the state of a cryptocurrency, consensus algorithms are used.
- Byzantine fault tolerant (BFT) consensus protocols were invented in 1982 and have evolved to become faster and simpler to understand.
- BFT consensus protocols need a known and fixed set of participants to work (permissioned network). Such protocols can just decide who is part of this participant set (proof of authority) or dynamically elect the participant set based on the amount of currency they hold (proof of stake).
- Bitcoin's consensus algorithm, the Nakamoto consensus, uses proof-of-work to validate the correct chain and to allow anyone to participate (permissionless network).
- Bitcoin's proof-of-work has participants (called "miners") compute a lot of hashes in order to find some with specific prefixes. Successfully finding a valid digest allows a miner to decide on the next block of transaction, collect a reward, as well as transaction fees.
- Accounts in Bitcoin are simply ECDSA keypairs using the secp256k1 curve. An account knows how much bitcoins they hold by looking at all transaction outputs that have not yet been spent (UTXOs). A transaction is thus a signed message authorizing the movement of a number of older transaction outputs to new outputs, spendable to different public keys.
- Bitcoin uses merkle trees to compress the size of a block and allow verification of transaction inclusion to be small in size.
- Stablecoins are cryptocurrencies that attempt to stabilize their values, most often by pegging their token to the value of a real currency like the US dollar.
- Cryptocurrencies use so-called layer 2 protocols in order to increase their latency by processing transactions off-chain, and saving progress on-chain periodically.
- Zero-knowledge proofs are used in many different blockchain applications, for example in Zcash to provide confidentiality, and in Coda to compress the whole blockchain to a short proof of validity.
- Libra uses a BFT consensus protocol called LibraBFT. It remains both safe (no forks) and live (progress is always made) as long as no more than f malicious participants exist out of $3f+1$ participants.
- LibraBFT works by having rounds, in which a participant proposes a block of transactions extending a previous block. Other participants can then vote for the block, potentially creating a quorum certificate if enough votes are gathered ($2f+1$).
- If a commit rule is triggered (a chain of 3 certified blocks at contiguous rounds), blocks can be committed and transactions they contain are deemed finalized.

Hardware cryptography

13

This chapter covers

- The issues that cryptography faces in highly-adversarial environments.
- The solutions that hardware offers to improve the attacker's cost in such environments.
- How software mitigations can also help cryptography against side-channel attacks.

At some point, writing cryptographic applications, you end up realizing that you have a number of short-term and long-term keys, and you have to make sure nobody can steal them. It means you're standing in the world of key management. Makes sense right? You've seen some of that in previous chapters, but in this chapter we'll do things a bit differently: we'll look at how key management and cryptography can be done in **highly-adversarial environments**. Environments where the attacker is much more powerful than the typical scenarios we've looked at so far.

Let's first introduce this concept in the next section. The rest of this chapter will then survey the different techniques that allow us to continue to do interesting things in spite of these constraints. Spoiler alert: it involves using specialized hardware. Finally, we'll see how cryptographic primitives have adapted to these highly-adversarial environments.

13.1 Modern cryptography attacker model

Present-day computer and network security starts with the assumption that there is a domain that we can trust. For example: if we encrypt data for transport over the internet, we generally assume the computer that's doing the encrypting is not compromised and that there's some other "endpoint" at which it can be safely decrypted.

– Joanna Rutkowska Intel x86 considered harmful (2015)

Cryptography used to be about "*Alice wants to encrypt a message to Bob, without Eve being able*

to intercept it". Today, a lot of it has moved to something more like "Alice wants to encrypt a message to Bob, but Alice has been compromised". It's a totally different attacker model, which is often not anticipated for in theoretical cryptography.

What do I mean by this? Let me give you some examples:

- Using your credit card on an automated teller machine (ATM) which might be augmented with a skimmer, a device that a thief can place on top of the card reader in order to copy the content of your bank card (see figure 13.1).
- Downloading an application on your mobile phone that compromises the operating system.
- Hosting a web application in a shared web hosting service, where the other customers sharing the same machine as you are malicious.
- Managing highly-sensitive secrets in a data center that gets visited by spies from a different country.

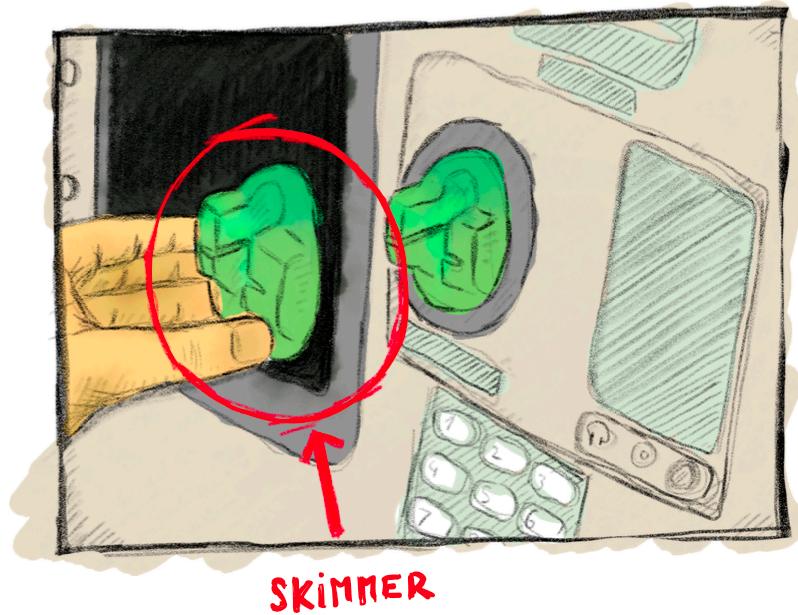


Figure 13.1 A skimmer: a malicious device that can be placed in front of an ATM card reader or a payment terminal card reader to copy data contained in the card magnetic stripe. The magnetic stripe usually contains the account number, the expiration date, and other metadata that can be used to pay online or in a number of payment terminals. Skimmers are sometimes seen accompanied with a hidden camera to obtain the user's PIN as well, potentially enabling the thief to use ATM withdrawals and payment terminals enforcing PIN entry.

All of these examples are modern use of cryptography in a threat model that many cryptographers ignore, or are totally unaware of.

Indeed, most of the cryptographic primitives that you can read about in the literature will just assume that Alice has total control of her execution environment, and only when a ciphertext (or a signature or a public key or ...) leaves her computer to go over the network, will a

man-in-the-middle attacker be able to perform their tricks. But in reality, and in modern days, we're often using cryptography in much more adversarial models.

Security is, after all, a product of your assumptions and what you expect of a potential attacker. If your assumptions are wrong, you're in for a bad time.

So how do real-world applications reconcile theoretical cryptography with these more powerful attackers? It makes **compromises**. In other words, they try to make the attackers' lives more difficult. The security of such systems is often calculated in cost (how much does the attacker have to spend to break the system) rather than computational complexity.

A lot of what you'll learn in this chapter will be imperfect cryptography, which in the real-world we call **defense-in-depth**. There's a lot to learn, and this chapter comes with a lot of new acronyms and different solutions that different vendors and their marketing teams and sales people have come up with.

So let's get started and learn about **trusted systems in untrusted environments!**

13.2 Untrusted environments: hardware to the rescue

If you thought that real-world cryptography was messy and that there were too many standards or ways to do the same thing, well wait until you read what's going on in the hardware world.

There are three interesting attacker models here that we'll talk about in this chapter:

- **Software Attacks.** Your phone, your laptop, and many of your devices are not as simple as they used to be: they can now run full-featured operating systems that allow users to install applications. These systems can also communicate with the outside world via a multitude of ways (WiFi, Bluetooth, etc.). Getting a virus on your device could mean a permanent intrusion. An intrusion that could help the attacker steal your keys, or observe your cryptographic operations, or hell even replace them! Specialized hardware can help thwart such software attacks.
- **Hardware Attacks.** By default, cryptography on hardware is helpless against the threat of physical attacks. Imagine the following scenario: you leave your phone or laptop unattended in your hotel room; a malicious maid comes, opens the device, uses a low-budget off-the-shelf tool to modify the system, and then leaves the device appearing untouched where it was found before you get back to your room. Maybe he copied or changed the content of your memory (perhaps even rolling it back to a previous state), or he replaced some of the hardware, or he installed components capable of man-in-the-middle'ing some of the hardware, etc. This is known as an **evil maid attack** in the literature and can be generalized to many situations (carrying devices in check-in luggages while flying, storing sensitive keys in an insecure data center, and so forth). More complex and more expensive hardware attacks exist. For example hardware certification labs will often use the same tools that chip manufacturers themselves use to debug their chips. Lasers can be shot on targetted places to force a computation to produce an erroneous value (so-called **fault attacks**), chips can be opened up to reveal their parts, focused ion beam (FIB) microscopes can be used to reverse-engineer

components. The sky's the limit and it is very hard to protect against such motivated attackers.

In the end, if one wants to protect against physical attacks, one typically just adds as many layers of defences as one can, in an attempt to make the attacker's life more and more difficult. **It is all about raising the costs**. In addition, it is important to design systems that do not collapse when a single instance gets broken. For cryptography this usually means that each device should use a unique key, preventing the compromise of one key to affect other devices.

NOTE

Unfortunately, the electronic world is full of bad cryptography. In 2017, the DUHK attack showed that tens of thousands of devices were using the same hardcoded seed with a random number generator, effectively breaking the security of a number of VPN appliances.⁸⁸

In this section we will survey different hardware solutions that attempt to protect against these different threats. As a preview you will learn about:

- **Whitebox cryptography.** A software-only mitigation that scrambles the implementation of a cryptographic primitive (like AES) with the key used, in order to prevent reverse-engineers from extracting the key.
- **Smart cards.** Small hardware chips that you can find in credit cards. They carry secret keys and perform cryptographic operations with them.
- **Secure elements.** A generalization of smart cards. The most commonly found secure elements are SIM cards in mobile phones.
- **Hardware Security Modules (HSMs).** Larger devices that are commonly found attached to enterprise servers in data centers. Like smart cards and secure elements, HSMs also hold secret keys and perform cryptographic operations.
- **Trusted Platform Modules (TPMs).** Re-packaged secure elements or secure-element-like microcontrollers that are commonly found directly plugged into laptops and server motherboards. As you will see, many vendors invent their own TPM-like chips.
- **Hardware Security Tokens.** Keys that can be typically plugged via the USB port of a laptop and that you've already seen in chapter 11 (then called roaming authenticators).
- **Trusted Execution Environments (TEEs).** A way to securely execute programs in a parallel environment isolated from the main operating system. In practice it is achieved via special features integrated directly into a normal processor, allowing execution of entire programs securely by the main CPU itself.

All of these solutions, for the exception of TEEs (that most often focuses on software attacks), attempt to protect against complex physical attacks. I expect that these short descriptions made little sense at this point, so let's take a look at each of them one by one!

13.2.1 Whitebox cryptography, a bad idea

Before getting into hardware solutions for untrusted environments, why not use software solutions? Can cryptography provide primitives that do not leak their own keys?

Whitebox cryptography is exactly this: a field of cryptography that attempts to scramble a cryptographical implementation with the key it uses, in order to prevent extraction of that key. That's right, the attacker can be given the source code of some whitebox AES-based encryption algorithm with a fixed key, and it will encrypt and decrypt fine, but the key is mixed so well with the implementation that it will be too hard for anyone to extract it from the algorithm. That's the theory at least. In practice, no published whitebox crypto algorithm has been found to be secure, and most commercial solutions are closed-source due to this fact (security through obscurity kinda works in the real-world, but should generally not be relied upon).⁸⁹ Again, it's all about raising the cost and making it harder for attackers.

This type of technique (scrambling an implementation to make it hard to understand) is called **obfuscation** in security, and it is rarely seen as a good way to provide strong security. Nonetheless it is not a useless technique! Remember, defense-in-depth is a thing in the real-world, and slowing down attackers is sometimes a desirable property of a system.

All in all, whitebox cryptography is a big industry that sells dubious products to businesses in need of Digital Rights Management (DRM) solutions.

NOTE

Briefly, DRM is a tool that controls how much access a customer can get to a product they bought. For example, you can find DRM in hardware that can play movies you bought in a store, or in software that can play movies you are watching on a streaming service. In reality DRM does not strongly prevent these "attacks", it just makes the life of their customers more difficult.

On the more serious side, there is a branch of cryptography called **Indistinguishability obfuscation (iO)** that attempts to do this cryptographically (so for realz). iO is a very theoretical, impractical, and so far not-a-really-proven field of research. We'll see how that one goes, but I wouldn't hold my breath.

13.2.2 You probably have one in your wallet: smart cards

OK, whitebox cryptography is not great, and that's pretty much the best software can do to defend against powerful adversaries. So let's turn to the hardware side for solutions, spoiler alert: things are about to get much more complicated and confusing. Different terms have been made up and used in different ways, and standards have unfortunately proliferated as much as (if not more than) cryptographic standards.

To understand what all of these hardware solutions are, and how they differ from one another, let's start with some necessary history.

Smart cards are small chips usually seen packaged inside plastic cards like bank cards, that is if your bank follows the EMV standard,⁹⁰ and were invented in the early 70's following advances in microelectronics. Smart cards started as a practical way to get everyone a **pocket computer!** Indeed, a modern smart card embeds its own CPU, different types of programmable or non-programmable memory (ROM, RAM, and EEPROM), inputs and outputs, a hardware random number generator (also called TRNG as you've learned in chapter 8), and so on. They're "smart" in the sense that they can run programs, unlike the not-so-smart cards that could only store data via a magnetic stripe (which could be easily copied via the skimmers I talked about previously). Most smart cards allow developers to write small and contained applications that can run on the card. The most popular standard supported by smart cards is JavaCard,⁹¹ which allow developers to write Java-like applications.

Today, it seems like most people have a much more powerful computer in their pockets, so smart cards are probably going to die.

Smart cards need to be activated by inserting it into a card reader. More recently cards have been augmented with the near-field communication (NFC) protocol to achieve the same result via radio frequencies (so by getting close to, as opposed to touching, another device).

NOTE

By the way, banks make use of smart cards to store a unique per-card secret capable of saying "I am indeed the card that you gave to this customer". Intuitively, you might think that this is implemented via public-key cryptography, but the banking industry is still stuck in the past and uses symmetric cryptography (due to the vast amount of legacy software and hardware still in use)! More specifically, every bank card stores a triple-DES (also called 3DES) symmetric key, an old 64-bit block cipher that improves on the Data Standard Encryption (DES) which was deprecated in favor of AES (covered in chapter 4). The algorithm is used not to encrypt, but to produce a MAC over some challenge. The MAC can be verified by the bank who holds every customer's current 3DES symmetric key. This is an excellent example of what real-world cryptography is often about: legacy algorithms used all over the place in a risky way. And this is also why key rotation is such an important concept (and why you have to change your bank cards so often).

Smart cards mix a number of physical and logical techniques to prevent observation, extraction, and modification of its execution environment and some of its memory (where secrets are stored). But as I said earlier, it's all about how much money an attacker is willing to spend. There exist many attacks that attempt to break these cards, which can be classified in three different categories:

- **Non-invasive attacks** such as differential power analysis (DPA) which analyze the power consumption of a smart card while it is doing cryptographic operations in order to extract its associated keys. Power consumption is not the only way cryptographic operations can leak critical information, as you will see later in this chapter in the section on side-channel attacks. (Although you've already seen how time can negatively affect cryptographic algorithms in chapter 3.)
- **Semi-invasive attacks** can use access to the chip's surface to mount attacks such as differential fault analysis (DFA) which use heat, lasers, and other techniques to modify the execution of a program running on the smart card in order to leak keys via cryptographic attacks.
- **Invasive attacks** can modify the circuitry in the silicon itself to alter its function and reveal secrets. These attacks are noticeable (because the device is damaged afterwards) and have higher chances of rendering the device unusable.

The fact that these hardware chips are extremely small and tightly packaged can make attacks very difficult, but specialized hardware usually go much further by using different layers of materials to prevent depackaging and physical observation, and by using hardware techniques to increase the innaccuracy of known attacks.

For example, the ATECC508A chip claims the following hardware protections:

Physical and cryptographic countermeasures make it impossible for an attacker to sniff operations to learn the keys, or probe the device to obtain the keys. The entire device is shielded with a serpentine metal pattern that prevents internal signal emissions from being detectable outside and provides a visual barrier against someone opening the package to observe and probe operations. The shield is electrically connected to the rest of the circuit. If it is compromised, the device will no longer operate, preventing a determined attacker from probing circuit nodes to learn the secrets. Regulators and counters are used to confound power and signal signatures. There are no extra internal pads for test and debug, so opening the package provides no additional access points.

– Atmel: Security for Intelligent Connected IoT Edge Nodes

Software can also help thwart some physical attacks, for example by adding jitter to the power consumption, or the clock. But more interestingly, cryptographic implementations can be implemented in ways that can render some of these attacks impossible. In the last section of this chapter, I will spend some time going over these implementation techniques.

13.2.3 Secure elements: a generalization of smart cards

Smart cards got really popular really fast, and it became obvious that having such a secure blackbox in other devices could be useful. The concept of a **secure element** was born: a tamper-resistant microcontroller that can be found in a plugxgble form factor like UUICs (also known as SIM cards, which are required by carriers for your phone to access their network) or directly bonded on chips and motherboards like the embedded SE (eSE) attached to an iPhone's NFC chip. A secure element is really just a small **separate** piece of hardware meant to protect your secrets and their usage in cryptographic operations.

Secure elements are an important concept to protect cryptographic operations in the **Internet of Things (IoTs)**, a colloquial (and overloaded) term to refer to devices that can communicate with other devices (think smart cards in credit cards, SIM cards in phones, biometric data in passports, garage keys, smart home sensors, and so on).

Thus, you can see all of the solutions that will follow in this section as secure elements implemented in different form factors, using different techniques to achieve the same thing, but providing different levels of speed as well as defenses against software and hardware attackers.

The main definitions and standards around secure elements have been produced by Global Platform,⁹² a nonprofit association created from the need of the different players in the industry to facilitate interoperability between different vendors and systems. There exist more standards that focus on the security claims of secure elements like Common Criteria (CC), NIST's FIPS, EMV (for Europay, Mastercard, and Visa), and so on. If you're in the market for buying secure microcontrollers, you will often see claims like "FIPS 140-2 certified" and "certified CC EAL 5+" in the product's description. These are claims that can be obtained after spending some quality time, and a lot of money, with licensed certification labs.

As secure elements are highly secretive recipes, integrating them in your product means that you will have to sign non-disclosure agreements and use closed-source hardware and firmware. For many projects, this is seen as a serious limitation in transparency, but can be understood as part of the security in these chips come from the obscurity of their design.⁹³

Next, let's look at hardware security tokens!

13.2.4 Enforcing user intent with hardware security tokens

Hardware security tokens are keys that you can usually plug into your machine and that can do some cryptographic operations. I've mentioned hardware tokens and how they work in chapter 11. If you remember, YubiKeys are small dongles that you can plug in the USB port of a laptop, and that will perform some cryptographic operations if you touch their exposed yellow rings.⁹⁴

It's not clear how much protection against hardware attacks your typical hardware security token

has to implement, since they are usually used in addition to a second authentication factor like a password. Thus, the compromise of a security key is not enough to successfully authenticate as a user (unless of course it is used as single factor authentication). Yet, certifications like FIDO's "Authenticator Certification Levels" exist,⁹⁵ and higher-levels mandate the use of specialized hardware; thus YubiKeys typically have a secure element inside. Still, this doesn't exclude software attacks if badly programmed.⁹⁶

Cryptocurrencies have similar dongles called hardware wallets that can store account keys, and sign transactions for a user. For these cryptocurrency hardware wallets, the threat model is different; all of the digital funds' security is reduced to the security of the key stored in these devices. Thus, such hardware dongles will usually implement another layer of authentication (for example by having the user manually enter a PIN on the device).

Let's look at TPMs next.

13.2.5 Trusted Platform Modules (TPMs): a useful standardization of secure elements

Secure elements are useful, but they are either quite limited to some specific usecases, or one has to write custom applications in order to create new usecases. For this reason the **Trusted Computing Group (TCG)**, another nonprofit organization formed from different industry players, published the first **Trusted Platform Module** standard in 2009. The latest version of the standard is TPM 2.0, published as the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) 11889.

A TPM complying with the TPM 2.0 standard is a secure microcontroller that carries a hardware random number generator, secure memory for storing secrets, can perform cryptographic operations, and the whole thing is tamper resistant. If this description reminds you of smart cards and secure elements, I told you that everything you were going to learn about in this chapter was going to be secure elements of some form or another. And actually, it's common to see TPMs implemented as a repackaging of secure elements.

You usually find a TPM directly soldered or plugged to the motherboard of enterprise servers, laptops, and desktop computers (see figure 13.2).

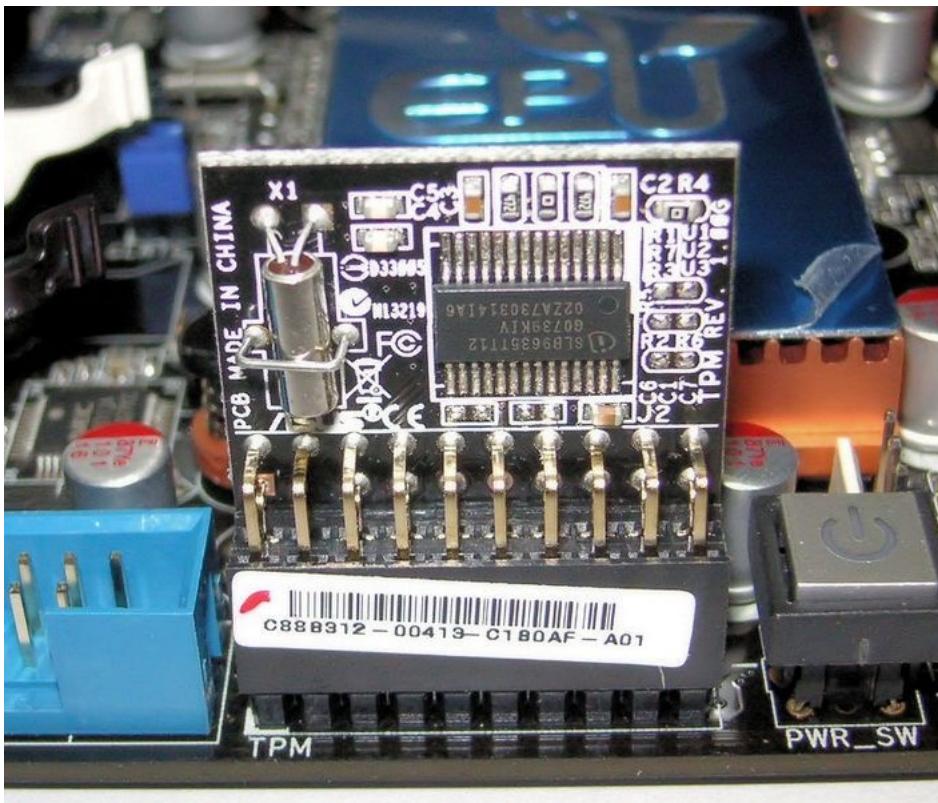


Figure 13.2 A chip implementing the TPM 2.0 standard, plugged into a PC's motherboard. This chip can be called by motherboard components of the system as well as user-applications running on the PC's operating system.

Unlike smart cards and secure elements, a TPM does not run arbitrary code, but offers a well-defined interface that a greater system can take advantage of. TPMs are usually pretty cheap, allowing more and more IoTs to ship with one.

Here is a non-exhaustive list of interesting applications that a TPM can enable:

- **User authentication.** TPMs can be used to verify a user PIN or password that is stored in the TPM itself. After that, the TPM can release secrets or perform cryptographic operations with its unique per-TPM key (called an endorsement key) to authenticate the user to a remote service (for example by signing a random challenge). In order to prevent low entropy credentials to be easily brute forced, a TPM can also rate limit or even count the number of failed attempts. More clever approaches limit brute force attacks by using a password-based key derivation function in order to verify the credentials (see chapter 2).
- **Measured boot.** Measured boot is about measuring what is being launched at boot. For this, the TPM offers special "registers" (called Platform Configuration Registers) that reset only when power is shut down. These registers are "rolling" hashes, meaning that you can only extend them and not replace them (they can only hash the previous hash with new input). During booting, the first piece of code ran will hash the image of the next piece of code that will be booted and send it to the TPM. The second piece of code will do the same with every piece of code that it will boot next. And so on.
- **Full disk encryption (FDE).** FDE encrypts the entire content of a disk with a key stored in the TPM in case your device becomes lost or stolen. A TPM can enforce user

authentication and ensures that the system has reached a known good state (via measured boot) before releasing the decryption key. When the device is locked or shut down, the key vanishes from memory and has to be released by the TPM again. Note that the TPM has to release the key because it is too slow to decrypt disk data on the fly.

- **Remote attestation.** This allows a device to prove to a remote service that it is running specific software. For example, during employee onboarding a company can add a new employee's laptop's TPM endorsement key to a whitelist of approved devices. Later, if the employee wants to access one of the company's services, the service can request the employee's TPM to sign a random challenge (to prevent replays) along with the result of the measured boot to authenticate the employee and prove the well-being of their device.
- **Secure boot.** Secure boot is stronger than measured boot: it is about enforcing that the system starts in a good state in order to avoid tampering of the OS by malware (or light physical intrusions). During boot, the startup code initializes the measuring registers in the TPM, then hashes the code of the next image to load and asks the TPM to compare these with the expected hashes before actually running the code. The process will fail to boot if the TPM cannot successfully match the hash of that next piece of code. If you hold a public key you can also verify that a piece of code has been signed before running it. Note that the very first piece of code booted (usually called the Core Root of Trust for Measurement), or the very first public key used to verify that code, is the **root of trust**. That root of trust is thus usually stored in read-only memory, fused during manufacturing, and cannot be modified.

NOTE Interestingly, the Nintendo switch was found to have a bug in its bootrom, which is the first piece of code that boots and as its name indicates is stored in read-only memory. Because of that, the bug is unpatchable.⁹⁷

These are only a few functionalities that a TPM can enable, and there are many more of them. There's after all hundreds of commands in the TPM standards.

Now the bad: the communication channel between a TPM and a processor can be man-in-the-middle'd easily.⁹⁸ While many TPMs provide a high level of resistance against physical attacks, the fact that their communication channel is somewhat open does reduce their usecases to mostly defend against software attacks. To solve these issues, there's been a move to TPM-like chips that are integrated directly into the main processor. For example Apple has the Secure Enclave, Microsoft has Pluton, and Google has Titan.

13.2.6 Banks love them: hardware security modules (HSMs)

If you understood what a secure element was, well a **hardware security module (HSM)** is pretty much a **bigger and faster** secure element. It's not always true, some are small,⁹⁹ and the term hardware security module can be used to mean different things by different people. Many would argue that all of the hardware solutions discussed so far are HSMs of different form factors, and that secure elements are just HSMs that are specified by GlobalPlatform while TPMs are HSMs that are specified by the Trusted Computing Group. But most of the time, when people talk about HSMs, they mean the big stuff. Like some secure elements, some HSMs can run arbitrary code as well. HSMs typically contain specialized hardware designed to perform certain cryptographic operations faster than a standard general-purpose computer, thus providing some degree of crypto acceleration.

HSMs are often classified according to FIPS 140-2: Security Requirements for Cryptographic Modules.¹⁰⁰ The document is quite old, published in 2001, and does not take into account a number of attacks discovered after its publication.¹⁰¹ Fortunately, it is superseded by the more modern version FIPS 140-3, and will be phased out by the end of 2021. FIPS 140 defines security levels for hardware security modules between 1 and 4; level 1 HSMs do not provide any protection against physical attacks, while level 4 HSMs will wipe their secrets if they detect any intrusion!

NOTE Wiping secret is a practice called **zeroization**. Some HSMs overwrite the data multiple times, even if unpowered thanks to backup internal batteries. And that's why there's very few level 4 HSMs in the market.

The US, Canada, and some other countries mandate certain industries like banks to use devices that have been certified according to the FIPS 140 levels. Many companies in the world have decided to follow these same recommendations.

Typically, you find an HSM as an external device with its own shelf on a rack (see figure 13.3) plugged to an enterprise server in a data center (), as a PCIe card plugged into a server's motherboard, or even as small dongles that resemble a hardware security token and that you can plug via USB (if you don't care about performance).



IBM 4767 cryptographic adapter (HSM) - picture from Eigenes Werk

Figure 13.3 An IBM 4767 HSM as a PCI card.

To go full circle, some of these HSMs can be administered using smart cards (to install applications, backup keys, and so on).

HSMs are highly used in some industries. Every time you enter your PIN in an ATM or a payment terminal, the PIN ends up being verified by an HSM somewhere. Whenever you connect to a website via HTTPS, the root of trust comes from a Certificate Authority (CA) that stores its private key in an HSM,¹⁰² and the TLS connection is possibly terminated by an HSM. You have an Android or iPhone? Chances are Google,¹⁰³ or Apple,¹⁰⁴ are keeping a backup of your phone safe with a fleet of HSMs. This last case is interesting because the threat model is reversed: the user does not trust the cloud with its data, and thus the cloud service provider claims that its service can't see the user's encrypted backup nor can access the keys used to encrypt it.

HSMs don't really have a standard, but most of them will at least implement the **Public-Key Cryptography Standard 11 (PKCS#11)**, one of these old standards that were started by the RSA company and that were progressively moved to the OASIS organization (2012) in order to facilitate adoption of the standards.

While the last version (2.40) of PKCS#11 was released in 2015, it is merely an update of a standard that originally started in 1994. For this reason it specifies a number of old cryptographic algorithms, or old ways of doing things, which can of course lead to vulnerabilities.¹⁰⁵ Nevertheless, it is good enough for many uses, and specifies an interface that allows different systems to easily interoperate with each other. The good news is that PKCS#11 v3.0 might be released at the time this writing gets printed in ink, which will include a lot of modern cryptographic algorithms like Curve25519, EdDSA, and SHAKE to name a few.

While HSMs' real goal is to make sure nobody can extract key material from them, their security is not always shining. A lot about the security of these hardware solutions really relies on their high price, the protection techniques used not being disclosed, and the certifications (like FIPS

and Common Criteria) mostly focusing on the hardware side of things. In practice, devastating software bugs have been found and it is not always straightforward to know if the HSM you use is vulnerable to any of these vulnerabilities.¹⁰⁶

NOTE

Not only the price of one HSM is high (it can easily be tens of thousands of dollars depending on the security level), in addition to an HSM you often have at least another HSM you use for testing, and at least another HSM you use for backup (in case your first HSM dies with its keys in it). It can add up!

Furthermore, I still haven't touched on the elephant in the room with all of these solutions: while you might prevent most attackers from reaching your secret keys, you can't prevent attackers from compromising the system and making their own calls to the secure hardware module (be it a secure element or an HSM). Again, these hardware solutions are not a panacea and depending on the scenario they provide more or less defense-in-depth.

13.2.7 Modern integrated solutions: Trusted Execution Environment (TEE)

So far, all of the hardware solutions we've talked about have been **standalone** secure hardware solutions (with the exceptions of smart cards which can be seen as tiny computers). Secure elements, HSMs, and TPMs can be seen as an additional computer.

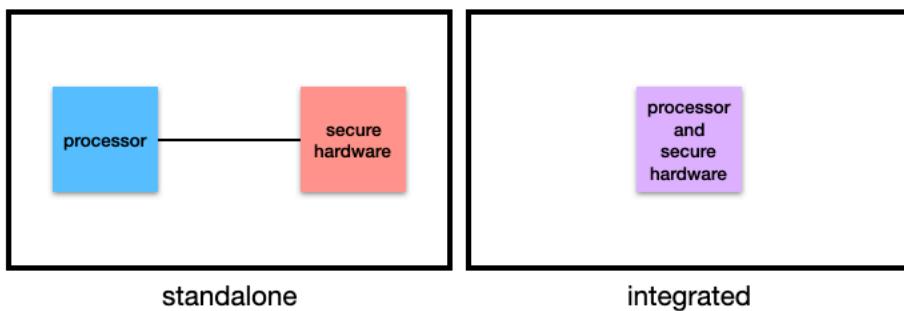


Figure 13.4 Standalone security modules typically carry their own CPU, memory, true random number generator, etc. and talk to the main processor via system bus or other wires. Integrated security modules on the other hand are implemented directly in the main processor and thus benefits from a lot of its features and speed.

Let's now talk about **integrated** secure hardware!

Trusted Execution Environment (TEE) is a concept that extends the instruction set of a processor to allow for programs to run in a separate secure environment. The separation between this secure environment and the ones we are used to dealing with already (often called "rich" execution environment) is done via hardware. So what ends up happening is that modern CPUs run both a normal OS as well as a secure OS simultaneously. Both have their own set of registers but share most of the rest of the CPU architecture. By using CPU-enforced logic, data from the

secure world cannot be accessed from the normal world. For example a CPU will usually split its memory, giving one part for the exclusive use of the TEE. The TEE also has access to a read-only key (usually fused manually at manufacturing) that it can use to wrap other keys, or sign payloads from the TEE (attestations). Due to TEE being implemented directly on the main processor, not only does it mean a TEE is a faster and cheaper product than a TPM or secure element, it also comes for free in a lot of modern CPUs.

TEE like all other hardware solutions has been a concept developed independently by different vendors, and then a standard trying to play catch up (by Global Platform). The most known TEEs are Intel's **Software Guard Extensions (SGX)** and ARM's **TrustZone**. But there are many more like AMD PSP, RISC-V MultiZone and IBM Secure Service Container.

By design, since a TEE runs on the main CPU and can run any code given to it (in a separate environment called an "enclave" or "secure world"), it offers more functionality than secure elements, HSMs, TPMs (and TPM-like chips). For this reason TEEs are used in a wider range of applications. TEEs being used in clouds when clients don't trust servers with their own data,¹⁰⁷, to emulate multi-party computation.¹⁰⁸, to run smart contracts.¹⁰⁹.

TEE's goal is to first and foremost thwart **software attacks**. While the claimed software security seems to be really attractive, it is in practice hard to segregate execution while on the same chip due to the extreme complexity of modern CPUs and their dynamic states, as can attest to the many software attacks against SGX and TrustZone.¹¹⁰¹¹¹¹¹²¹¹³¹¹⁴¹¹⁵¹¹⁶¹¹⁷

In theory, a TPM can be re-implemented in software only via a TEE, but one must be careful as again, TEE as a concept provides no resistance against hardware attacks besides the fact that things at this microscopic level are way too tiny and tightly packaged together to analyze without expensive equipment. But by default a TEE does not mean you'll have a secure internal storage (you need to have an additional fused key that can't be read to encrypt what you want to store), or a hardware random number generator, and other wished hardware features. But every manufacturer sure has different offers with different levels of physical security and tamper resistance when it comes to chips that support TEE.

13.3 What solution is good for me?

You have learned about many hardware products in this chapter. As a recap, this is the list:

- **Smart cards** are microcomputers that need to be turned on by an external device like a payment terminal. They can run small Java-like applications. Bank cards are an example of a widely used smart card.
- **Secure elements** are a generalization of smart cards, which rely on a set of Global Platform standards. SIM Cards are secure elements for example.
- **TPMs** are re-packaged secure elements plugged on personal and enterprise computers' motherboards. They follow a standardized API (by the Trusted Computing Group) that

are used in a multitude of ways from measured/secure boot with FDE to remote attestation.

- **Hardware Security Tokens** are dongles like YubiKeys that often repackage secure elements to provide a 2nd factor by implementing some authentication protocol (usually TOTP or FIDO2).
- **HSMs** can be seen as external and big secure elements for servers. They're faster and more flexible. Seen mostly in data centers to store keys.
- **TEEs** like TrustZone and SGX can be thought of as secure elements implemented within the CPU. They are faster and cheaper but mostly provide resistance against software attacks unless augmented to be tamper-resistant. Most modern CPUs ship with TEEs and various levels of defense against hardware attacks.

I illustrate all these in figure 13.5.

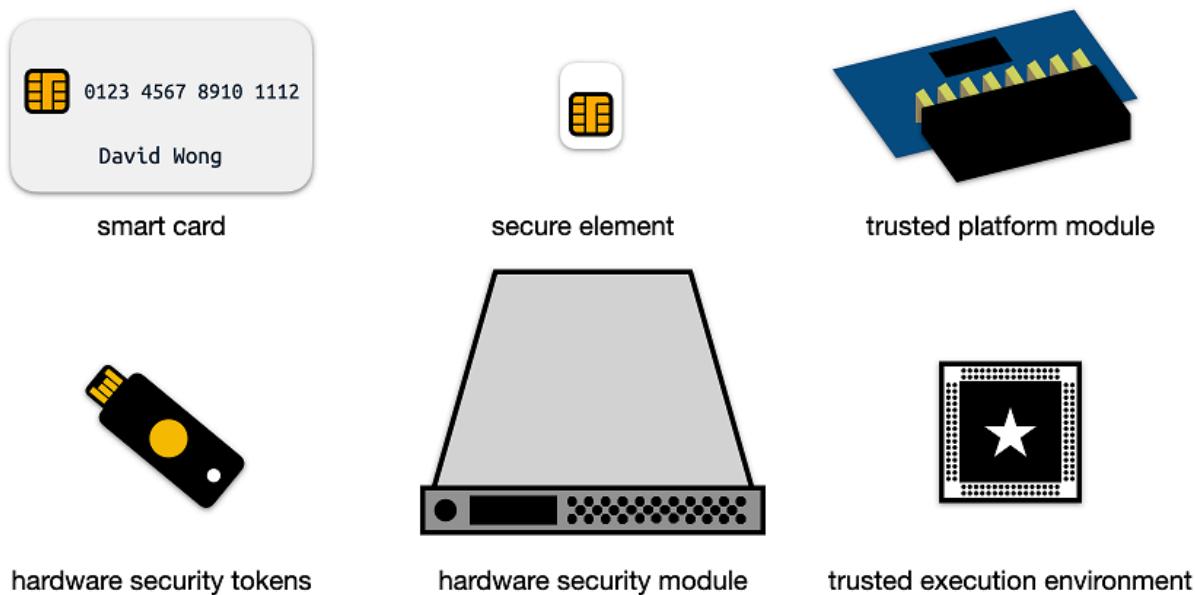


Figure 13.5 The different hardware solutions you've learned in this chapter and an idea of what they look like.

What is the best solution for you? Try to narrow your choice by asking yourself some questions:

- In what form factor? For example, the need for a secure element in a small device will dictate what solutions you won't be able to use.
- How much speed do you need? Applications that need to perform a high number of cryptographic operations per second will be highly constrained in the solutions they can use: probably limited to HSMs and TEEs.
- How much security do you need? Certifications and claims by vendors will correspond to different levels of software or hardware security. The sky's the limit.

Keep in mind that no hardware solution is the panacea, you're only increasing the attack's cost. Against a sophisticated attacker all of that is pretty much useless. For this reason design your system so that one device compromised doesn't imply all devices are compromised. Even against normal adversaries, compromising the main operating system often means that you can make arbitrary calls to the secure element. Design your protocol to make sure that the secure element

doesn't have to trust the caller by either verifying queries, or relying on an external trusted part, or by relying on a trusted remote party, or by being self-contained, etc. And after all of that, you still have to worry about side-channel attacks :)

After all of that, you still have more problems: how do you know all of this was done correctly? How do you know that no backdoor was inserted in the hardware you're relying on at some point during manufacturing or transport (so-called **supply chain attacks**)?

It's turtles all the way down. So this is where I'll stop.

13.4 Leakage-resilient cryptography - or how to mitigate side-channel attacks in software

We've seen how hardware attempts to prevent direct observation and extraction of secret keys, but there's only so much the hardware can do. At the end of the day, it is possible for the software to not care and give out the key despite all of this hardware hardening. The software can do so somewhat directly (like a backdoor) or it can do it indirectly by leaking enough information for someone to reconstruct the key. This latter option is called a side-channel, and side-channel vulnerabilities are most of the time non-intended bugs (at least one would hope).

I've mentioned **timing attacks** in chapter 3, where you learned that MAC authentication tags had to be compared in constant-time otherwise attackers could infer the correct tag after sending you many incorrect ones and measuring how long they waited for you to respond. Timing attacks are usually taken seriously in all areas of real-world cryptography due to the fact that they can potentially be remotely exploited over the network unlike physical side-channels.

The most important and known side-channel is **power consumption** which I've mentioned earlier in this chapter. It was discovered as an attack called **Differential Power Analysis (DPA)** by Kocher, Jaffe, and Jun in 1998; when they realized that they could hook an oscilloscope to a device, and observe variance in the electricity consumed by the device over time when performing encryption of known plaintexts. This variance clearly dependend on the bits of the key used, and the fact that operations like XORing would consume more or less power depending if the operand bits were set or not. This observation led to a key-extraction attack (so-called "total breaks").

This concept can be illustrated with **Simple Power Analysis (SPA)** attacks. In some ideal situations, and when no hardware or software mitigations are implemented against power analysis attacks, it suffices to measure and analyze the power consumption of a single cryptographic operation involving a secret key. I illustrate this in figure 13.6.

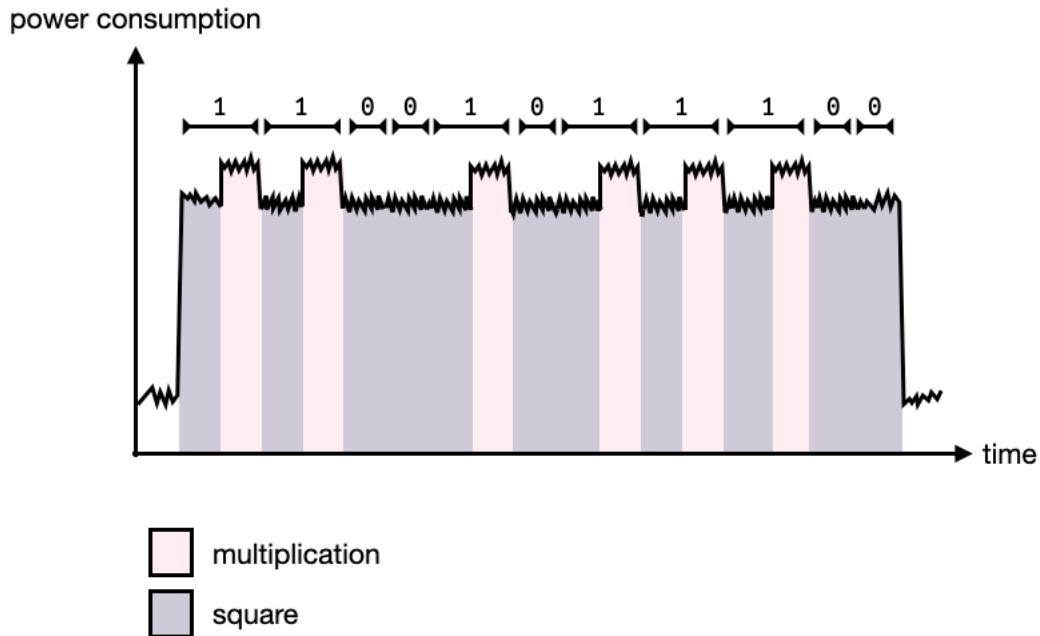


Figure 13.6 Some cryptographic algorithms leak so much information via their power consumption that a simple power analysis of a single power trace (a measure of the power consumed in time) can leak the private key of the algorithm. For example, this figure represents a trace of an RSA exponentiation (the message being exponentiated to the private exponent, see chapter 6) done with the square-and-multiply algorithm which looks at the bits of the private exponent one by one and works by applying a square and multiply operation if a bit is 1, and a square operation if a bit is 0. The multiplication is obviously consuming more power, hence the clarity of the power trace.

Power is not the only physical side-channel, some attacks rely on electromagnetic radiations, vibrations, and even the sound emitted by the hardware.

Let me still mention two other non-physical side-channels. I know we are in a hardware-focused chapter, but these non-physical side-channels attacks are important as they need to be mitigated in many real-world cryptographic applications:

Errors. Errors returned can sometimes leak critical information. For example, in 2018 the ROBOT attack figured out a way to exploit the Bleichenbacher attack (mentioned in chapter 6) on a number of servers that implemented RSA PKCS#1 v1.5 decryption in the TLS protocol (covered in chapter 9).¹¹⁸ Bleichenbacher's attack only works if you can distinguish when an RSA encrypted message sent has a valid padding or not. To protect against that attack, safe implementations will perform the padding validation in constant-time and avoid returning early if it detects that the padding is invalid early on. Yet, if at the very end of the padding validation the implementation returns a different error to the client based on the validity of the padding, then this was all for nothing.

Memory Access. Accessing memory can take more or less time depending if the data was previously accessed or not, this is due to the numerous layers of caching that exist in a computer.

For example if the CPU needs something it will first check if it has been cached in its internal memory, if not it will then reach into caches that are further and further away from it. The further away the cache, the more time it'll take. Not only that, but some caches are specific to a core (L1 cache for example) while some caches are shared amongst cores in a multicore machine (L3 cache, RAM, disk). **Cache attacks** exploit the fact that it is possible for a malicious program to run on the same machine and to use the same cryptographic library as a sensitive cryptographic program. For example, many cloud services host different clients' virtual servers on the same machine, and many servers use the OpenSSL library for cryptographic operations or for serving TLS pages. These malicious programs find ways to evict parts of the library that have been loaded in a cache shared with the victim's process, and then periodically measure the time it takes to re-read some parts of that library. If it took a long time, then the victim did not execute this part of the program, if it didn't take a long time then the victim did access this part of the program and re-populated the cache (to avoid having to re-fetch this to a far away cache, or worse from disk). What you obtain is a trace that resembles a power trace! And it is indeed exploitable in similar ways.

OK that's enough for side-channel attacks. If you're interested in attacking cryptography via these side-channels there are better resources than this book. In this section, I will talk only about software mitigations that cryptographic implementations can and should implement to protect against side-channel attacks in general. This whole field of study is called **leakage-resilient cryptography** as the cryptographer's goal here is to not leak anything.

Note that defending against physical attackers is an endless battle, which explains why many of these mitigations are proprietary and akin to obfuscation. This section is obviously not exhaustive, but should give you an idea of the type of things applied cryptographers are working on to address side-channel attacks.

13.4.1 Constant-Time Programming

The first line of defense for any cryptographic implementation is to implement its cryptographic sensitive parts (think any computation that involves a secret) in constant-time. It is obvious that implementing something in constant-time cancels timing attacks, but this also gets rid of many classes of attacks like cache attacks and simple power analysis attacks.

How to implement something in constant-time? Never **branch**. In other words: no matter what the input is, always do the same thing.

For example, this is how the Golang language implements a **constant-time comparison of authentication tags for the HMAC algorithm**. Intuitively, if two bytes are equal, then their XOR will be 0. If this property is verified for every pair of bytes we are comparing, then ORing them will also lead to a 0 value (and a non-zero value otherwise). Warning: it can be quite disconcerting to read this code if this is the first time you're looking at constant-time tricks :)

Listing 13.1 How Golang implements a constant-time comparison between two bytearrays.

```
func ConstantTimeCompare(x, y []byte) byte {
    if len(x) != len(y) { ❶
        return 0 ❷
    } ❸

    var v byte ❹
    for i := 0; i < len(x); i++ { ❺
        v |= x[i] ^ y[i] ❻
    } ❼

    return v ❼
}
```

- ❶ There is no point comparing two strings in constant-time if they are of different lengths.
- ❷ Here is where the magic happens, the loop OR accumulates the XOR of every byte into a value v.
- ❸ Returns 0 only if v is equal to 0, and returns a non-zero value otherwise.

For a MAC authentication tag comparison, it is enough to stop here to check if the result is 0 or not by branching (using a conditional expression like `if`).¹¹⁹

Another interesting example are **scalar multiplications** in elliptic curve cryptography, which as you've learned in chapter 5 consists of adding a point to itself x number of times (where x is what we call a scalar). This process can be somewhat slow, and thus clever algorithms exist to speed this part up. One of the popular one is called **Montgomery's ladder** and is pretty much the equivalent to the RSA's square-and-multiply algorithm I've mentioned earlier (but in a different group). Montgomery ladder's algorithm alternates between addition of two points and doubling of a point (adding the point to itself). Both algorithms have a simple way to mitigate timing attack: by not branching and always doing both operations. (And this is why the RSA exponentiation algorithm in constant-time is usually referred to as **square and multiply always**.)

NOTE

I have mentioned that signature schemes can go wrong in multiple ways in chapter 7, and that key recovery attacks exist against implementations that leak a few bytes of the nonces they use in signature schemes like ECDSA. This is what happened for real in the Minerva¹²⁰ and TPM Fail¹²¹ attacks that happened around the same time. Both attacks found out that a number of devices were vulnerable to these kinds of attacks, due to amount of timing variation the signing operation would take.

In practice timing attacks are not always straightforward to mitigate, as it is not always clear if CPU instructions for multiplications or conditional moves are always constant time, and it is not always clear how the compiler will compile high-level code when used with different compilation flags. For this reason, a manual review of the assembly generated is sometimes

performed in order to obtain more confidence in the constant-time code written. Different tools also exist (with different degrees of result) to analyze constant-time code.¹²²

13.4.2 Don't use the secret! Masking and blinding

Another common way of thwarting, or at least confusing attackers, is to add layers of indirection to any operations involving secrets. One of these techniques is called **blinding**, and is often possible thanks to the arithmetic structure of public-key cryptography algorithms.

You've seen blinding being used in oblivious algorithms like password-authenticated key exchange (PAKE) algorithms in chapter 11, and we can use blinding in the same way where we want the oblivious party to be the attacker observing leaks from our computations.

Let's talk about RSA as an example.

Remember, RSA decrypts by taking a ciphertext c and raising it to the private exponent d , where the private exponent d cancels the public exponent e which was used to compute the ciphertext as $m^e \bmod N$. If you don't remember the details, make sure to consult chapter 6.

One way to add indirection is to perform the decryption operation on a value that is not the ciphertext known to the attacker. This method is called **base blinding** and goes like this:

1. Generate a random blinding factor r .
2. Compute $\text{message}' = (ciphertext \times r^e)^d \bmod N$.
3. Finally, unblind the result by computing $\text{message} = m' \times r^{-1} \bmod N$ where r^{-1} is the inverse of r .

This method blinds the value being used with the secret, but we can also blind the secret itself. For example, elliptic curve scalar multiplication is usually used with a secret scalar. But as computations take place in a cyclic group, adding a multiple of order to that secret does not change the computation result. This technique is called **scalar blinding** and goes like this:

1. Generate a random value k .
2. Compute a scalar $d' = d + k \times \text{order}$ where d is the original secret scalar and order is its order.
3. To compute $Q = [d]P$, instead compute $Q = [d']P$ which will result in the same point.

Usually all of these techniques have been proven to be more or less efficient, and are often used in combinations with other software and hardware mitigations.

In symmetric cryptography, another somewhat similar technique called **masking** is used.

The concept of masking is to transform the input (the plaintext or ciphertext in the case of a cipher) before passing it to the algorithm. For example, by XORing it with a random value. The output is then unmasked in order to obtain the final correct output.

As any intermediate state is thus masked, this provides the cryptographic computation some amount of decorrelation from the input data, and makes side-channel attacks much more difficult.

Note that the algorithm has to be aware of this masking to correctly perform internal operations while keeping the correct behavior of the original algorithm.

13.4.3 What about fault attacks?

I've previously talked about **fault attacks**, a more intrusive type of side-channel attacks that modify the execution of the algorithm by inducing faults.

Injecting faults can be done in many creative ways physically, by increasing the heat of the system for example, or even by shooting lasers at calculated points in the targeted chip.

Surprisingly, faults can also be induced via software. An example was found independently in the Plundervolt and VOLTpwn attacks, which managed to change the voltage of a CPU to introduce natural faults. This also happened in the infamous rowhammer attack, which found out that repeatedly accessing memory of some DRAM devices could flip nearby bits.¹²³

These types of attacks can be difficult to achieve, but are extremely powerful. In cryptography, computing a bad result can sometimes leak the key, this is for example the case with RSA signatures that are implemented with some specific optimizations.¹²⁴

While it is impossible to fully mitigate these attacks, there exist some techniques that can increase the complexity of a successful attack. For example, by computing the same operation several times and comparing the results to make sure they match before releasing it, or by verifying the result before releasing it (for signatures, one can verify the signature via the public key before returning it).

Fault attacks can also have dramatic consequences against random number generators. One easy solution there is to use algorithms that do not use new randomness every time they run. For example, in chapter 7 you learned about EdDSA, a signature algorithm that requires no new randomness to sign as opposed to the ECDSA signature algorithm.

13.5 Summary

- The threat today is not just an attacker intercepting messages over the wire, but an attacker stealing or tampering with the device that runs your cryptography. So-called Internet of Things (IoT) devices often run into this type of threats and are by default unprotected against sophisticated attackers. More recently cloud services are also considered in the threat model of their users.
- **Hardware can help protect cryptography applications and their secrets in a highly-adversarial environment.** One of the ideas is to provide a device with a tamper-resistant chip to store and perform crypto operations. That is, if the device falls in the hands of an attacker, extracting keys or modifying the behavior of the chip will be hard.
- It is generally accepted that one has to combine different software and hardware techniques to harden cryptography in adversarial environments. But hardware-protected cryptography is not a panacea, it is merely **defense-in-depth**, effectively slowing down and **increasing the cost of an attack**. Adversaries with unlimited time and money will always break your hardware.
- Decreasing the impact of an attack can also help deter attackers. This must be done by designing a system well, for example by making sure that the compromise of one device does not imply a compromise of all devices.
- While there are many hardware solutions, the most popular ones are:
 - **Smart cards** were one of the first such secure microcontrollers that could be used as a micro computer to store secrets and perform cryptographic operations with them. These are supposed to use a number of techniques to discourage physical attackers.
 - The concept of a smart card was generalized as a **secure element**, which is a term employed differently in different domains, but boils down to a smart card that can be used as a coprocessor in a greater system that already has a main processor.
 - Trusted Platform Modules (TPMs) are chips that follow the TPM standard, more specifically they are a type of secure element with a specified interface. A TPM is usually a secure chip directly linked to the motherboard and perhaps implemented using a secure element. While it does not allow running arbitrary programs like some secure elements, smart cards, and HSMs do, it enables a number of interesting applications for devices as well as user applications.
 - Hardware security tokens are user-oriented devices that can be useful to store keys and enforce user intent at the same time. They are used for multi-factor authentication, cryptocurrency wallets, and other use-cases, and are often built with a secure element.
 - Hardware Security Module (HSM) is a term often used to refer to external, bigger and faster secure elements. They do not follow any standard interface, but usually implement the PKCS#11 standard for cryptographic operations. HSMs can be certified with different levels of security via some NIST standard (FIPS 140-2 and soon 140-3).
 - Trusted Execution Environment (TEE) is a way to segregate an execution environment between a secure one and an insecure one. This is usually implemented via special hardware, and is found integrated as part of many modern CPUs (extending their instruction set).
- Hardware is not enough to protect cryptographic operations in highly-adversarial environments, as software and hardware side-channel attacks can still exploit leakage that

occurs in different ways (timing, power consumption, electromagnetic radiations, etc.) In order to defend against these side-channel attacks cryptographic algorithms implement software mitigations:

- Serious cryptographic implementations are based on constant-time algorithms and avoid all branching, as well as memory accesses that depend on secret data.
- Mitigation techniques based on blinding and masking will decorrelate sensitive operations from either the secret or the data known to be operated on.
- Fault attacks are harder to protect against. Mitigations include computing an operation several times and comparing, and verifying the result of an operation (for example verifying a signature with the public key), before releasing the result.
- Hardening cryptography in adversarial settings is a never-ending battle. One should use a combination of software and hardware mitigations to increase the cost and the time for a successful attack up to a desired accepted risk. One should also decrease the impact of an attack by using unique keys per device, and potentially unique keys per cryptographic operation.

Notes

Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen - Dual EC: A Standardized Back Door

1. projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf
2. Partitions in the S-Box of Streebog and Kuznyechik tosc.iacr.org/index.php/ToSC/article/view/7405
3. nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf
4. csrc.nist.gov/publications/detail/sp/800-185/final
5. cr.yp.to/talks/2009.02.28-3/slides.pdf
6. password-hashing.net/
7. eprint.iacr.org/2016/989
8. 131002.net/siphash/
9. tools.ietf.org/html/rfc5652#section-6.3
10. cryptologie.net/article/413/beast-an-explanation-of-the-cbc-attack-on-tls/
11. robertheaton.com/2013/07/29/padding-oracle-attack/
12. www.isg.rhul.ac.uk/tls/Lucky13.html
13. www.intel.com/content/www/us/en/processors/carry-less-multiplication-instruction-in-gcm-mode-paper.html
14. csrc.nist.gov/publications/detail/sp/800-38d/final
15. csrc.nist.gov/publications/detail/fips/140/2/final
16. tools.ietf.org/html/rfc7457#section-2.6
17. Authentication Failures in NIST version of GCM
csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/800-38-series-drafts/gcm/joux_c
- Better Bounds for Block Cipher Modes of Operation via Nonce-Based Key Derivation
18. eprint.iacr.org/2017/702.pdf

19. tools.ietf.org/html/rfc8439
20. ChaCha, a variant of Salsa20 cr.yp.to/chacha/chacha-20080128.pdf
21. www.ecrypt.eu.org/stream/
22. Extending the Salsa20 nonce cr.yp.to/snuffle/xsalsa-20081128.pdf
23. XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305
tools.ietf.org/html/draft-arciszewski-xchacha-03
24. csrc.nist.gov/publications/detail/sp/800-38f/final

Deterministic Authenticated-Encryption: A Provable-Security Treatment of the Key-Wrap Problem
25. web.cs.ucdavis.edu/~rogaway/papers/siv.pdf

Hanno Böck and Aaron Zauner and Sean Devlin and Juraj Somorovsky and Philipp Jovanovic -
26. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS eprint.iacr.org/2016/475
27. datatracker.ietf.org/doc/draft-irtf-cfrg-gcmsiv/
28. Synthetic IV (SIV) for non-AES ciphers and MACs tools.ietf.org/html/draft-madden-generalised-siv-00
29. On the Practical (In-)Security of 64-bit Block Ciphers sweet32.info/SWEET32_CCS16.pdf
30. competitions.cr.yp.to/caesar.html
31. www.ecrypt.eu.org/stream/
32. www.cosic.esat.kuleuven.be/nessie/
33. csrc.nist.gov/projects/lightweight-cryptography
34. Ciphers with Arbitrary Finite Domains web.cs.ucdavis.edu/~rogaway/papers/subset.pdf
35. SP 800-38G Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38G.pdf
36. The Curse of Small Domains: New Attacks on Format-Preserving Encryption eprint.iacr.org/2018/556 and
Breaking the FF3 Format-Preserving Encryption Standard Over Small Domains eprint.iacr.org/2017/521

Recommendation for Block Cipher Modes of Operation: the XTS-AES Mode for Confidentiality on Storage
37. Devices csrc.nist.gov/publications/detail/sp/800-38e/final

38. security.googleblog.com/2019/02/introducing-adiantum-encryption-for.html

this number is obtained by doing 100 (the number of edges coming out of the first node) + 99 (the number of
39. edges coming out of the second node) + ... + 1

40. github.com/jedisct1/libsodium

41. ROCA: Vulnerable RSA generation (CVE-2017-15361) (crocs.fi.muni.cz/public/papers/rsa_ccs17)

42. tools.ietf.org/html/rfc7919

43. like the Extended Euclidean algorithm

1998 - Daniel Bleichenbacher - Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption
44. Standard PKCS #1 (archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf)

45. www.cryptography.io

2019 - Sze Yiu Chau, Moosa Yahyazadeh†, Omar Chowdhury†, Aniket Kate, Ninghui Li - Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification (46. www.cs.purdue.edu/homes/schau/files/pkcs1v1_5-ndss19.pdf)

Console Hacking 2010 - PS3 Epic Fail - Chaos Communication Congress 2010 (47. youtu.be/btDiX319P4w?t=588)

RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital
48. Signature Algorithm (ECDSA) (tools.ietf.org/html/rfc6979)

49. SEC 2: Recommended Elliptic Curve Domain Parameters (www.secg.org/sec2-v2.pdf)

50. 2017 - Edwards-Curve Digital Signature Algorithm (EdDSA) (tools.ietf.org/html/rfc8032)

51. mailarchive.ietf.org/arch/msg/acme/F71iz6qq1o_QPVhJCV4dqWf-4Yc/

52. www.letsencrypt.org

53. tools.ietf.org/html/rfc8555

54. eprint.iacr.org/2011/343

55. arxiv.org/abs/1403.6676

56. tools.ietf.org/html/rfc8032

Hash functions and MACs are theoretically not defined as providing outputs that are indistinguishable from random, but in practice they often are. Signatures on the other hand are almost all the time not
57. indistinguishable from random.

58. RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF) tools.ietf.org/html/rfc5869

59. "TCP/IP Illustrated" is often seen as a reference

many of the attacks have been summarized in RFC 7457 - Summarizing Known Attacks on Transport Layer
60. Security (TLS) and Datagram TLS (DTLS)

2012 - Georgiev et al. - The Most Dangerous Code in the World: Validating SSL Certificates in
61. Non-Browser Software

62. number from netmarketshare.com

63. The most popular way to hide DNS resolution is DNS over HTTPS (DoH).

64. Protocols like Roughtime exist to provide authenticated time to cryptographic applications.

Hijacking the Internet Is Far Too Easy
65. slate.com/technology/2018/11/bgp-hijacking-russia-china-protocols-redirect-internet-traffic.html

66. RAMPART-A en.wikipedia.org/wiki/RAMPART-A

in his 1991 essay "Why Do You Need PGP?", Philip Zimmermann ends with "PGP empowers people to take
67. their privacy into their own hands. There's a growing social need for it. That's why I wrote it."

68. see the white paper "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels"

69. www.saltstack.org

70. signal.org/docs/specifications/x3dh/

71. github.com/signalapp

72. France enters the Matrix (February 2019) lwn.net/Articles/779331/

73. WhatsApp Encryption Overview whitepaper www.whatsapp.com/security
74. datatracker.ietf.org/wg/mls/about/
75. security.googleblog.com/2017/01/security-through-transparency.html
76. RFC 6819: OAuth 2.0 Threat Model and Security Considerations
77. openid.net
78. bhavukjain.com/blog/2020/05/30/zeroday-signin-with-apple/
79. RFC 2944 - Telnet Authentication: SRP
80. RFC 5054 - Using the Secure Remote Password (SRP) Protocol for TLS Authentication
81. User authentication with passwords, What's SRP? www.cryptologie.net/article/503/
82. github.com/cfrg/pake-selection
83. datatracker.ietf.org/doc/draft-krawczyk-cfrg-opaque/
84. this is the case for the IK handshake pattern for example
85. knobattack.com/
86. tools.ietf.org/html/draft-haase-cspace-01
- Sven Laur and Kaisa Nyberg - Efficient Mutual Data Authentication Using Manually Authenticated Strings
87. (2008)
- The DUHK Attack - Don't Use Hard-coded Keys (duhkattack.com)

In 2017 and 2019, Crypto Experts organized two whitebox cryptography competitions
89. (whibox-contest.github.io/2019). Not a single construction managed to withstand the participants' attacks.
- Europay, Mastercard, and Visa gave their name to the standard, which has been widely adopted in places like
90. Europe while the US has been slow to catch up.
- javacardforum.com

92. globalplatform.org

Interestingly, in early 2020 Tropic Square (tropicsquare.com) was founded with the goal of creating an open-sourced secure element.

while YubiKeys are the most popular hardware security tokens, they are not the only ones. SoloKeys, for example, is an interesting open-source alternative (solokeys.com).

95. fidoalliance.org/certification/authenticator-certification-levels

96. Oswald et al. - Side-Channel Attacks on the YubiKey 2 One-Time Password Generator (2013)

97. fail0verflow.com/blog/2018/shofel2/

TPM Genie: Interposer Attacks Against the Trusted Platform Module Serial Bus
98. (nccgroup.com/us/our-research/tpm-genie-interposer-attacks-against-the-trusted-platform-module-serial-bus)

99. The YubiHSM is an HSM that resembles a YubiKey

100. csrc.nist.gov/publications/detail/fips/140/2/final

101. see section 4.11 Mitigation of Other Attacks of FIPS 140-2 (csrc.nist.gov/publications/detail/fips/140/2/final)

102. cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.6.9.pdf#page=48

103. security.googleblog.com/2018/10/google-and-android-have-your-back-by.html

104. youtube.com/watch?v=BLGFriOKz6U

The Untold Story of PKCS#11 HSM Vulnerabilities, 2015
105. (cryptosense.com/blog/the-untold-story-of-pkcs11-hsm-vulnerabilities)

106. Everybody be Cool, This is a Robbery! (2019) (donjon.ledger.com/BlackHat2019-presentation)

107. signal.org/blog/private-contact-discovery

108. github.com/microsoft/CCF

109. research.nccgroup.com/2020/03/24/smart-contracts-inside-sgx-enclaves-common-security-bug-patterns

110. Software Grand Exposure: SGX Cache Attacks Are Practical

Foreshadow - Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution
111. (foreshadowattack.eu)

112. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution

113. RIDL: Rogue In-Flight Data Load (mdsattacks.com)

114. Pludervolt (plundervolt.com)

115. VOLTpwn: Attacking x86 Processor Integrity from Software

116. LVI - Hijacking Transient Execution with Load Value Injection (lviattack.eu)

Introduction to Trusted Execution Environment: ARM's TrustZone
117. (blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html)

118. The Return Of Bleichenbacher's Oracle Threat (ROBOT) Attack (robotattack.org)

Some algorithms do care about checking if a value is 0 or not in constant time, and there are ways to do this as well. If you're curious to know more about constant-time code, check the Constant-Time Toolkit

119. (github.com/pornin/CTTK).

120. Minerva: The curse of ECDSA nonces (minerva.crocs.fi.muni.cz)

121. TPM—Fail TPM meets Timing and Lattice Attacks (tpm.fail)

122. Oscar Reparaz, Josep Balasch and Ingrid Verbauwhede - Dude, is my code constant time? (2016)

123. googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

124. cryptologie.net/article/371/fault-attacks-on-rsas-signatures/

Notes for chapter 12

1. for a good explanation on RAFT check thesecretlivesofdata.com/raft
2. cryptoservices.github.io/blockchain/consensus/2019/05/21/bitcoin-length-weight-confusion.html
crypto51.app has a table that lists the cost of performing a 51% attack on different cryptocurrencies based on
3. proof of work.
4. arxiv.org/abs/1803.05069
5. Fischer, Lynch, and Paterson - Impossibility of distributed consensus with one faulty process
this bound has been shown to be somewhat flexible, see "Flexible Byzantine Fault Tolerance" from Malkhi et al.
6. et al.
7. developers.libra.org/docs/state-machine-replication-paper