Cryptography CS 161 Computer Security Textbook

An online version is available at https://textbook.cs161.org.

Textbook by David Wagner, Nicholas Weaver, Peyrin Kao,
Fuzail Shakir, Andrew Law, and Nicholas Ngai

Additional contributions by Noura Alomar, Sheqi Zhang, and Shomil Jain

Last update: August 24, 2021

Contact for corrections: cs161-staff@berkeley.edu

In this unit, we'll be studying *cryptography*, techniques for securing information and communication in the presence of an attacker. In particular, we will see how we can prevent adversaries from reading or altering our private data. In a nutshell, cryptography is about communicating securely over insecure communication channels.

The ideas we'll examine have significant grounding in mathematics, and in general constitute the most systematic and formal set of approaches to security that we'll cover.

5 Introduction to Cryptography

5.1 Disclaimer: Don't try this at home!

In this class, we will teach you the basic building blocks of cryptography, and in particular, just enough to get a feeling for how they work at a conceptual level. Understanding cryptography at a conceptual level will give you good intuition for how industrial systems use cryptography in practice.

However, cryptography in practice is very tricky to get right. Actual real-world cryptographic implementations require great attention to detail and have hundreds of possible pitfalls. For example, private information might leak out through various side-channels, random number generators might go wrong, and cryptographic primitives might lose all security if you use them the wrong way. We won't have time to teach all of those details and pitfalls to you in CS 161, so you should never implement your own cryptography using the algorithms we teach you in this class.

Instead, the cryptography we show you in this class is as much about educating you as a consumer as educating you as an engineer. If you find yourself needing an encrypted connection between two computers, or if you need to send an encrypted message to another person, you should use existing well-vetted cryptographic tools. However, you will often be faced with the problem of understanding how something is supposed to work. You might also be asked to evaluate the difference between alternatives. For that, you will

need to understand the underlying cryptographic engineering involved. Similarly, there are sometimes applications that take advantage of cryptographic primitives in non-cryptographic ways, so it is useful to know the primitives. You never know when you might need a hash, an HMAC, or a block cipher for a non-security task that takes advantage of their randomness properties.

In summary, know that we're going to teach you just enough cryptography to be dangerous, but not enough to implement industrial-strength cryptography in practice.

5.2 Brief History of Cryptography

The word "cryptography" comes from the Latin roots *crypt*, meaning secret, and *graphia*, meaning writing. So cryptography is quite literally the study of how to write secret messages.

Schemes for sending secret messages go back to antiquity. 2,000 years ago, Julius Caesar employed what's today referred to as the "Caesar cypher," which consists of permuting the alphabet by shifting each letter forward by a fixed amount. For example, if Caesar used a shift by 3 then the message "cryptography" would be encoded as "fubswrjudskb". With the development of the telegraph (electronic communication) during the 1800s, the need for encryption in military and diplomatic communications became particularly important. The codes used during this "pen and ink" period were relatively simple since messages had to be decoded by hand. The codes were also not very secure, by modern standards.

The second phase of cryptography, the "mechanical era," was the result of a German project to create a mechanical device for encrypting messages in an unbreakable code. The resulting *Enigma* machine was a remarkable feat of engineering. Even more remarkable was the massive British effort during World War II to break the code. The British success in breaking the Enigma code helped influence the course of the war, shortening it by about a year, according to most experts. There were three important factors in the breaking of the Enigma code. First, the British managed to obtain a replica of a working Enigma machine from Poland, which had cracked a simpler version of the code. Second, the Allies drew upon a great deal of brainpower, first with the Poles, who employed a large contingent of mathematicians to crack the structure, and then from the British, whose project included Alan Turing, one of the founding fathers of computer science. The third factor was the sheer scale of the code-breaking effort. The Germans figured that the Enigma was well-nigh uncrackable, but what they didn't figure on was the unprecedented level of commitment the British poured into breaking it, once codebreakers made enough initial progress to show the potential for success. At its peak, the British codebreaking organization employed over 10,000 people, a level of effort that vastly exceeded anything the Germans had anticipated. They also developed electromechanical systems that could, in parallel, search an incredible number of possible keys until the right one was found.

Modern cryptography is distinguished by its reliance on mathematics and electronic computers. It has its early roots in the work of Claude Shannon following World War II. The analysis of the *one-time pad* (discussed in the next chapter) is due to Shannon. The early 1970s saw the introduction of a standardized cryptosystem, *DES*, by the National Institute for Standards in Technology (NIST). DES answered the growing need for digital encryption

standards in banking and other businesses. The decade starting in the late 1970s then saw an explosion of work on a computational theory of cryptography.

5.3 Definitions

Intuitively, we can see that the Caesar cypher is not secure (try all 26 possible shifts and you'll get the original message back), but how can we prove that it is, in fact, insecure? To formally study cryptography, we will have to define a mathematically rigorous framework that lets us analyze the security of various cryptographic schemes.

5.3.1 Alice, Bob, Eve, and Mallory

The most basic problem in cryptography is one of ensuring the security of communications across an insecure medium. Two recurring members of the cast of characters in cryptography are *Alice* and *Bob*, who wish to communicate securely as though they were in the same room or were provided with a dedicated, untappable line. However, they only have available a telephone line or an Internet connection subject to tapping by an eavesdropping adversary, *Eve*. In some settings, Eve may be replaced by an active adversary, *Mallory*, who can tamper with communications in addition to eavesdropping on them.

The goal is to design a scheme for scrambling the messages between Alice and Bob in such a way that Eve has no clue about the contents of their exchange, and Mallory is unable to tamper with the contents of their exchange without being detected. In other words, we wish to simulate the ideal communication channel using only the available insecure channel.

5.3.2 Keys

The most basic building block of any cryptographic system (or *cryptosystem*) is the *key*. The key is a secret value that helps us secure messages. Many cryptographic algorithms and functions require a key as input to lock or unlock some secret value.

There are two main key models in modern cryptography. In the *symmetric key* model, Alice and Bob both know the value of a secret key, and must secure their communications using this shared secret value. In the *asymmetric key* model, each person has a secret key and a corresponding *public key*. You might remember RSA encryption from CS 70, which is an asymmetric-key encryption scheme.

5.3.3 Confidentiality, Integrity, Authenticity

In cryptography, there are three main security properties that we want to achieve.

Confidentiality is the property that prevents adversaries from reading our private data. If a message is confidential, then an attacker does not know its contents. You can think about confidentiality like locking and unlocking a message in a lockbox. Alice uses a key to lock the message in a box and then sends the message (in the locked box) over the insecure channel to Bob. Eve can see the locked box, but cannot access the message inside since she does not

have a key to open the box. When Bob receives the box, he is able to unlock it using the key and retrieve the message.

Most cryptographic algorithms that guarantee confidentiality work as follows: Alice uses a key to *encrypt* a message by changing it into a scrambled form that the attacker cannot read. She then sends this encrypted message over the insecure channel to Bob. When Bob receives the encrypted message, he uses the key to *decrypt* the message by changing it back into its original form. We sometimes call the message *plaintext* when it is unencrypted and *ciphertext* when it is encrypted. Even if the attacker can see the encrypted ciphertext, they should not be able to decrypt it back into the corresponding plaintext–only the intended recipient, Bob, should be able to decrypt the message.

Integrity is the property that prevents adversaries from tampering with our private data. If a message has integrity, then an attacker cannot change its contents without being detected.

Authenticity is the property that lets us determine who created a given message. If a message has authenticity, then we can be sure that the message was written by the person who claims to have written it.

You might be thinking that authenticity and integrity seem very closely related, and you would be correct; it makes sense that before you can prove that a message came from a particular person, you first have to prove that the message was not changed. In other words, before you can prove authenticity, you first have to be able to prove integrity. However, these are not identical properties and we will take a look at some edge cases as we delve further into the cryptographic unit.

You can think about cryptographic algorithms that ensure integrity and authenticity as adding a seal on the message that is being sent. Alice uses the key to add a special seal, like a piece of tape on the envelope, on the message. She then sends the sealed message over the unsecure channel. If Mallory tampers with the message, she will break the tape on the envelope, and therefore break the seal. Without the key, Mallory cannot create her own seal. When Bob receives the message, he checks that the seal is untampered before unsealing the envelope and revealing the message.

Most cryptographic algorithms that guarantee integrity and authenticity work as follows: Alice generates a *tag* or a *signature* on a message. She sends the message with the tag to Bob. When Bob receives the message and the tag, he verifies that the tag is valid for the message that was sent. If the attacker modifies the message, the tag should no longer be valid, and Bob's verification will fail. This will let Bob detect if the message has been altered and is no longer the original message from Alice. The attacker should not be able to generate valid tags for their malicious messages.

A related property that we may want our cryptosystem to have is *deniability*. If Alice and Bob communicate securely, Alice might want to publish a message from Bob and show it to a judge, claiming that it came from Bob. If the cryptosystem has deniability, there is no cryptographic proof available to guarantee that Alice's published message came from Bob. For example, consider a case where Alice and Bob use the same key to generate a signature on a message, and Alice publishes a message with a valid signature. Then the judge cannot

be sure that the message came from Bob–the signature could have plausibly been created by Alice.

5.3.4 Overview of schemes

We will look at cryptographic primitives that provide confidentiality, integrity, and authentication in both the symmetric-key and asymmetric-key settings.

	Symmetric-key	Asymmetric-key
Confidentiality	Block ciphers with chaining modes (e.g., AES-CBC)	Public-key encryption (e.g., El Gamal, RSA encryption)
Integrity and authentication	MACs (e.g., HMAC)	Digital signatures (e.g., RSA signatures)

In symmetric-key encryption, Alice uses her secret key to encrypt a message, and Bob uses the same secret key to decrypt the message.

In public-key encryption, Bob generates a matching public key and private key, and shares the public key with Alice (but does not share his private key with anyone). Alice can encrypt her message under Bob's public key, and then Bob will be able to decrypt using his private key. If these schemes are secure, then no one except Alice and Bob should be able to learn anything about the message Alice is sending.

In the symmetric-key setting, *message authentication codes (MACs)* provide integrity and authenticity. Alice uses the shared secret key to generate a MAC on her message, and Bob uses the same secret key to verify the MAC. If the MAC is valid, then Bob can be confident that no attacker modified the message, and the message actually came from Alice.

In the asymmetric-key setting, *public-key signatures* (also known as digital signatures) provide integrity and authenticity. Alice generates a matching public key and private key, and shares the public key with Bob (but does not share her private key with anyone). Alice computes a digital signature of her message using her private key, and appends the signature to her message. When Bob receives the message and its signature, he will be able to use Alice's public key to verify that no one has tampered with or modified the message, and that the message actually came from Alice.

We will also look at several other cryptographic primitives. These primitives don't guarantee confidentiality, integrity, or authenticity by themselves, but they have desirable properties that will help us build secure cryptosystems. These primitives also have some useful applications unrelated to cryptography.

- *Cryptographic hashes* provide a one way digest: They enable someone to condense a long message into a short sequence of what appear to be random bits. Cryptographic hashes are irreversible, so you can't go from the resulting hash back to the original message but you can quickly verify that a message has a given hash.

- Many cryptographic systems and problems need a lot of random bits. To generate these we use a *pseudo random number generator*, a process which takes a small amount of true randomness and stretches it into a long sequence that should be indistinguishable from actual random data.
- *Key exchange* schemes (e.g. Diffie-Hellman key exchange) allow Alice and Bob to use an insecure communication channel to agree on a shared random secret key that is subsequently used for symmetric-key encryption.

5.3.5 Kerckhoff's Principle

Let's now examine the threat model, which in this setting involves answering the question: How powerful are the attackers Eve and Mallory?

To consider this question, recall *Kerckhoff's principle* from the earlier notes about security principles:

> Cryptosystems should remain secure even when the attacker knows all internal details of the system. The key should be the only thing that must be kept secret, and the system should be designed to make it easy to change keys that are leaked (or suspected to be leaked). If your secrets are leaked, it is usually a lot easier to change the key than to replace every instance of the running software. (This principle is closely related to *Shannon's Maxim: Don't rely on security through obscurity.*)

Consistent with Kerckhoff's principle, we will assume that the attacker knows the encryption and decryption algorithms.[1] The only information the attacker is missing is the secret key(s).

5.3.6 Threat models

When analyzing the confidentiality of an encryption scheme, there are several possibilities about how much access an eavesdropping attacker Eve has to the insecure channel:

1. Eve has managed to intercept a single encrypted message and wishes to recover the plaintext (the original message). This is known as a *ciphertext-only attack*.
2. Eve has intercepted an encrypted message and also already has some partial information about the plaintext, which helps with deducing the nature of the encryption. This case is a *known plaintext attack*. In this case Eve's knowledge of the plaintext is partial, but often we instead consider complete knowledge of one instance of plaintext.
3. Eve can capture an encrypted message from Alice to Bob and re-send the encrypted message to Bob again. This is known as a *replay attack*. For example, Eve captures the encryption of the message "Hey Bob's Automatic Payment System: pay Eve $100" and sends it repeatedly to Bob so Eve gets paid multiple times. Eve might not know

[1] The story of the Enigma gives one possible justification for this assumption: given how widely the Enigma was used, it was inevitable that sooner or later the Allies would get their hands on an Enigma machine, and indeed they did.

the decryption of the message, but she can still send the encryption repeatedly to carry out the attack.

4. Eve can trick Alice to encrypt arbitrary messages of Eve's choice, for which Eve can then observe the resulting ciphertexts. (This might happen if Eve has access to the encryption system, or can generate external events that will lead Alice to sending predictable messages in response.) At some other point in time, Alice encrypts a message that is unknown to Eve; Eve intercepts the encryption of Alice's message and aims to recover the message given what Eve has observed about previous encryptions. This case is known as a *chosen-plaintext attack*.

5. Eve can trick Bob into decrypting some ciphertexts. Eve would like to use this to learn the decryption of some other ciphertext (different from the ciphertexts Eve tricked Bob into decrypting). This case is known as a *chosen-ciphertext attack*.

6. A combination of the previous two cases: Eve can trick Alice into encrypting some messages of Eve's choosing, and can trick Bob into decrypting some ciphertexts of Eve's choosing. Eve would like to learn the decryption of some other ciphertext that was sent by Alice. (To avoid making this case trivial, Eve is not allowed to trick Bob into decrypting the ciphertext sent by Alice.) This case is known as a *chosen-plaintext/ciphertext attack*, and is the most serious threat model.

Today, we usually insist that our encryption algorithms provide security against chosen-plaintext/ciphertext attacks, both because those attacks are practical in some settings, and because it is in fact feasible to provide good security even against this very powerful attack model.

However, for simplicity, this class will focus primarily on security against chosen-plaintext attacks.

6 Symmetric-Key Encryption

In this section, we will build symmetric-key encryption schemes that guarantee confidentiality. Because we are in the symmetric key setting, in this section we can assume that Alice and Bob share a secret key that is not known to anyone else. Later we will see how Alice and Bob might securely exchange a shared secret key over an insecure communication channel, but for now you can assume that only Alice and Bob know the value of the secret key.

For modern schemes, we are going to assume that all messages are bitstrings, which is a sequence of bits, 0 or 1 (e.g. *1101100101010101*). Text, images, and most other forms of communication can usually be converted into bitstrings before encryption, so this is a useful abstraction.

6.1 IND-CPA Security

In the previous section, we defined confidentiality to mean that an attacker cannot read our messages. Recall from the previous chapter that confidentiality was defined to mean that an attacker cannot read our messages. This definition, while intuitive, is quite open-ended. If the attacker can read the first half of our message but not the second half, is that confidential? What if the attacker can deduce that our message starts with the words "Dear Bob?" It might also be the case that the attacker had some partial information about the message M to begin with. Perhaps she knew that the last bit of M is a 0, or that 90% of the bits of M are 1's, or that M is one of BUY! or SELL but does not know which.

A more formal, rigorous definition of confidentiality is: the ciphertext C should give the attacker no additional information about the message M. In other words, the attacker should not learn any new information about M beyond what they already knew before seeing C (seeing C should not give the attacker any new information).

We can further formalize this definition by designing an experiment to test whether the attacker has learned any additional information. Consider the following experiment: Alice has encrypted and sent one of two messages, either M_0 or M_1, and the attacker Eve has no idea which was sent. Eve tries to guess which was sent by looking at the ciphertext. If the encryption scheme is confidential, then Eve's probability of guessing which message was sent should be 1/2, which is the same probability as if she had not intercepted the ciphertext at all, and was instead guessing at random.

We can adapt this experiment to different threat models by allowing Eve to perform further actions as an attacker. For example, Eve might be allowed to trick Alice into encrypting some messages of Eve's choosing. Eve might also be allowed to trick Alice into decrypting some ciphertexts of Eve's choosing. In this class, we will be focusing on the chosen-plaintext attack model, which means Eve can trick Alice into encrypting some messages, but she cannot trick Alice into decrypting some messages.

In summary, our definition of confidentiality says that even if Eve can trick Alice into encrypting some messages, she still cannot distinguish whether Alice sent M_0 or M_1 in the experiment. This definition is known as indistinguishability under chosen plaintext attack,

or IND-CPA. We can use an experiment or game, played between the adversary Eve and the challenger Alice, to formally prove that a given encryption scheme is IND-CPA secure or show that it is not IND-CPA secure.

The IND-CPA game works as follows:

1. The adversary Eve chooses two different messages, M_0 and M_1, and sends both messages to Alice.
2. Alice flips a fair coin. If the coin is heads, she encrypts M_0. If the coin is tails, she encrypts M_1. Formally, Alice chooses a bit $b \in \{0, 1\}$ uniformly at random, and then encrypts M_b. Alice sends the encrypted message $Enc(K, M_b)$ back to Eve.
3. Eve is now allowed to ask Alice for encryptions of messages of Eve's choosing. Eve can send a plaintext message to Alice, and Alice will always send back the encryption of the message with the secret key. Eve is allowed to repeat this as many times as she wants. Intuitively, this step is allowing Eve to perform a chosen-plaintext attack in an attempt to learn something about which message was sent.
4. After Eve is finished asking for encryptions, she must guess whether the encrypted message from step 2 is the encryption of M_0 or M_1.

If Eve can guess which message was sent with probability $> 1/2$, then Eve has won the game. This means that Eve has learned some information about which message was sent, so the scheme is not IND-CPA secure. On the other hand, if Eve cannot do any better than guess with $1/2$ probability, then Alice has won the game. Eve has learned nothing about which message was sent, so the scheme is IND-CPA secure.

There are a few important caveats to the IND-CPA game to make it a useful, practical security definition:

The messages M_0 and M_1 must be the same length. In almost all practical cryptosystems, we allow ciphertexts to leak the length of the plaintext. Why? If we want a scheme that doesn't reveal the length of the plaintext, then we would need every ciphertext to be the same length. If the ciphertext is always n bits long, then we wouldn't be able to encrypt any messages longer than n bits, which makes for a very impractical system. You could make n very large so that you can encrypt most messages, but this would mean encrypting a one-bit message requires an enormous n-bit ciphertext. Either way, such a system would be very impractical in real life, so we allow cryptosystems to leak the length of the plaintext.

If we didn't force M_0 and M_1 to be the same length, then our game would incorrectly mark some IND-CPA secure schemes as insecure. In particular, if a scheme leaks the plaintext length, it can still be considered IND-CPA secure. However, Eve would win the IND-CPA game with this scheme, since she can send a short message and a long message, see if Alice sends back a short or long ciphertext, and distinguish which message was sent. To account for the fact that cryptosystems can leak plaintext length, we use equal-length messages in the IND-CPA game.

Eve is limited to a practical number of encryption requests. In practice, some schemes may be vulnerable to attacks but considered secure anyway, because those attacks are computa-

tionally infeasible. For example, Eve could try to brute-force a 128-bit secret key, but this would take 2^{128} computations. If each computation took 1 millisecond, this would take 10^{28} years, far longer than the age of our solar system. These attacks may be theoretically possible, but they are so inefficient that we don't need to worry about attackers who try them. To account for these computationally infeasible attacks in the IND-CPA game, we limit Eve to a practical number of encryption requests. One commonly-used measure of practicality is polynomially-bounded runtime: any algorithm Eve uses during the game must run in $O(n^k)$ time, for some constant k.

Eve only wins if she has a non-negligible advantage. Consider a scheme where Eve can correctly which message was sent with probability $1/2 + 1/2^{128}$. This number is greater than $1/2$, but Eve's advantage is $1/2^{128}$, which is astronomically small. In this case, we say that Eve has *negligible* advantage–the advantage is so small that Eve cannot use it to mount any practical attacks. For example, the scheme might use a 128-bit key, and Eve can break the scheme if she guesses the key (with probability $1/2^{128}$). Although this is theoretically a valid attack, the odds of guessing a 128-bit key are so astronomically small that we don't need to worry about it. The exact definition of negligible is beyond the scope of this class, but in short, Eve only wins the IND-CPA game if she can guess which message was sent with probability greater than $1/2 + n$, where n is some non-negligible probability.

You might have noticed that in step 3, there is nothing preventing Eve from asking Alice for the encryption of M_0 or M_1 again. This is by design: it means any deterministic scheme is not IND-CPA secure, and it forces any IND-CPA secure scheme to be non-deterministic. Informally, a deterministic scheme is one that, given a particular input, will always produce the same output. For example, the Caesar Cipher that was seen in the previous chapter is a deterministic scheme since giving it the same input twice will always produce the same output (i.e. inputting "abcd" will always output "cdef" when we shift by 2). As we'll see later, deterministic schemes do leak information, so this game will correctly classify them as IND-CPA insecure. In a later section we'll also see how to win the IND-CPA game against a deterministic scheme.

6.2 XOR review

Symmetric-key encryption often relies on the bitwise XOR (exclusive-or) operation (written as $\oplus$), so let's review the definition of XOR:

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

Given this definition, we can derive some useful properties:

$x \oplus 0 = x$	0 is the identity
$x \oplus x = 0$	x is its own inverse
$x \oplus y = y \oplus x$	commutative property
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	associative property

One handy identity that follows from these is: $x \oplus y \oplus x = y$. In other words, given $(x \oplus y)$, you can retrieve y by computing $(x \oplus y) \oplus x$, effectively "cancelling out" the x.

We can also perform algebra with the XOR operation:

$y \oplus 1 = 0$	goal: solve for y
$y \oplus 1 \oplus 1 = 0 \oplus 1$	XOR both sides by 1
$y = 1$	simplify left-hand side using the identity above

6.3 One Time Pad

The first symmetric encryption scheme we'll look at is the *one-time pad (OTP)*. The one time pad is a simple and idealized encryption scheme that helps illustrate some important concepts, though as we will see shortly, it is impractical for real-world use.

In the one-time pad scheme, Alice and Bob share an n-bit secret key $K = k_1 \cdots k_n$ where the bits $k_1, \ldots k_n$ are picked uniformly at random (they are the outcomes of independent unbiased coin flips meaning that to pick k_1 a coin is flipped and if it lands on heads, then k_1 is assigned 1, but if it lands on tails, k_1 is assigned 0).

Suppose Alice wishes to send the n-bit message $M = m_1 \cdots m_n$.

The desired properties of the encryption scheme are:

1. It should scramble up the message, i.e., map it to a ciphertext $C = c_1 \cdots c_n$.
2. Given knowledge of the secret key K, it should be easy to recover M from C.
3. Eve, who does not know K, should get *no* information about M.

Encryption in the one-time pad is very simple: $c_j = m_j \oplus k_j$. In words, you perform a bitwise XOR of the message and the key. The jth bit of the ciphertext is the jth bit of the message, XOR with the jth bit of the key.

We can derive the decryption algorithm by doing some algebra on the encryption equation:

$c_j = m_j \oplus k_j$	encryption equation, solve for m_j
$c_j \oplus k_j = m_j \oplus k_j \oplus k_j$	XOR both sides by k_j
$c_j \oplus k_j = m_j$	simplify right-hand side using the handy identity from above

In words, given ciphertext C and key K, the jth bit of the plaintext is the jth bit of the ciphertext, XOR with the jth bit of the key.

To sum up, the one-time pad is described by specifying three procedures:

- Key generation: Alice and Bob pick a shared random key K.
- Encryption algorithm: $C = M \oplus K$.
- Decryption algorithm: $M = C \oplus K$.

Now let's prove that the one-time pad is IND-CPA secure. In other words, we want to show that in the IND-CPA game, the adversary Eve's probability of guessing which message was sent is $1/2$.

Proof: For a fixed choice of plaintext M, every possible value of the ciphertext C can be achieved by an appropriate and unique choice of the shared key K: namely $K = M \oplus C$. Since each such key value K is equally likely, it follows that C is also equally likely to be any n-bit string. Thus Eve sees a uniformly random n bit string no matter what the plaintext message was, and thus gets no information about which of the two messages was encrypted.

Here's another way to see that Eve's probability of guessing successfully is $1/2$. Suppose Eve observes the ciphertext C, and she knows that the message M is either M_0 or M_1, but she does not know which. The probability space here has size 2^{n+1}: it represents the 2^n choices for the n-bit key K, as well as the challenger's choice of whether to send M_0 or M_1. All 2^{n+1} choices are equally likely. We can assume the key K is generated uniformly at random; then the challenger randomly chooses a bit $b \in \{0, 1\}$, and Alice sends the encryption of M_b. So, if Eve observes that the ciphertext has some specific value C, what is the conditional probability that $b = 0$ given her observation? It is:

$$\begin{aligned}
\Pr[b = 0 \mid \text{ciphertext} = C] &= \frac{\Pr[b = 0 \wedge \text{ciphertext} = C]}{\Pr[\text{ciphertext} = C]} \\
&= \frac{\Pr[b = 0 \wedge K = M_0 \oplus C]}{\Pr[\text{ciphertext} = C]} \\
&= \frac{1/2 \cdot 1/2^n}{1/2^n} \\
&= \frac{1}{2}.
\end{aligned}$$

The one time pad has a major drawback. As its name suggests, the shared key cannot be reused to transmit another message M'. If the key K is reused to encrypt two messages M and M', then Eve can take the XOR of the two ciphertexts $C = M \oplus K$ and $C' = M' \oplus K$ to obtain $C \oplus C' = M \oplus M'$. This gives partial information about the two messages. In particular, if Eve happens to learn M, then she can deduce the other message M'. In other words, given $M \oplus M'$ and M, she can calculate $M' = (M \oplus M') \oplus M$. Actually, in this case, she can reconstruct the key K, too. Question: How?[2]

In practice, even if Eve does not know M or M', often there is enough redundancy in messages that merely knowing $M \oplus M'$ is enough to recover most of M and M'. For instance, the US

[2]Answer: Given M and $C = M \oplus K$, Eve can calculate $K = M \oplus C$.

exploited this weakness to read some World War II era Soviet communications encrypted with the one-time pad, when US cryptanalysts discovered that Soviet officials in charge of generating random keys for the one-time pad got lazy and started re-using old keys. The VENONA project, although initiated just shortly after World War II, remained secret until the early 1980s.

We can see that the one-time pad with key reuse is insecure because Eve has learned something about the original messages (namely, the XOR of the two original messages). We can also formally prove that the one-time pad with key reuse is not IND-CPA secure by showing a strategy for the adversary Eve to correctly guess which message was encrypted, with probability greater than 1/2.

Eve sends two messages, M_0 and M_1 to the challenger. The challenger randomly chooses one message to encrypt and sends it back to Eve. At this point, Eve knows she has received either $M_0 \oplus K$ or $M_1 \oplus K$, depending on which message was encrypted. Eve is now allowed to ask for the encryption of arbitrary messages, so she queries the challenger for the encryption of M_0. The challenger is using the same key for every message, so Eve will receive $M_0 \oplus K$. Eve can now compare this value to the encryption she is trying to guess: if the value matches, then Eve knows that the challenger encrypted M_0 and sent $M_0 \oplus K$. If the value doesn't match, then Eve knows that the challenger encrypted M_1 and sent $M_1 \oplus K$. Thus Eve can guess which message the challenger encrypted with 100% probability! This is greater than 1/2 probability, so Eve has won the IND-CPA game, and we have proven that the one-time pad scheme with key reuse is insecure.

Consequently, the one-time pad is not secure if the key is used to encrypt more than one message. This makes it impractical for almost all real-world situations–if Alice and Bob want to encrypt an n-bit message with a one-time pad, they will first need to securely send each other a new, previously unused n-bit key. But if they've found a method to securely exchange an n-bit key, they could have just used that same method to exchange the n-bit message![3]

6.4 Block Ciphers

As we've just seen, generating new keys for every encryption is difficult and expensive. Instead, in most symmetric encryption schemes, Alice and Bob share a secret key and use this single key to repeatedly encrypt and decrypt messages. The block cipher is a fundamental building block in implementing such a symmetric encryption scheme.

Intuitively, a block cipher transforms a fixed-length n-bit input into a fixed-length n-bit output. The block cipher has 2^k different settings for scrambling, so it also takes in a k-bit

[3]This is why the only primary users of one-time-pads are spies in the field. Before the spy leaves, they obtain a large amount of key material. Unlike the other encryption systems we'll see in these notes, a one-time pad can be processed entirely with pencil and paper. The spy then broadcasts messages encrypted with the one-time pad to send back to their home base. To obfuscate the spy's communication, there are also "numbers stations" that continually broadcast meaningless sequences of random numbers. Since the one-time pad is IND-CPA secure, an adversary can't distinguish between the random number broadcasts and the messages encoded with a one time pad.

key as input to determine which scrambling setting should be used. Each key corresponds to a different scrambling setting. The idea is that an attacker who doesn't know the secret key won't know what mode of scrambling is being used, and thus won't be able to decrypt messages encrypted with the block cipher.

A block cipher has two operations: encryption takes in an n-bit plaintext and a k-bit key as input and outputs an n-bit ciphertext. Decryption takes in an n-bit ciphertext and a k-bit key as input and outputs an n-bit plaintext. Question: why does the decryption require the key as input?[4]

Given a fixed scrambling setting (key), the block cipher encryption must map each of the 2^n possible plaintext inputs to a different ciphertext output. In other words, given a specific key, the block cipher encryption must be able to map every possible input to a unique output. If the block cipher mapped two plaintext inputs to the same ciphertext output, there would be no way to decrypt that ciphertext back into plaintext, since that ciphertext could correspond to multiple different plaintexts. This means that the block cipher must also be *deterministic*. Given the same input and key, the block cipher should always give the same output.

In mathematical notation, the block cipher can be described as follows. There is an encryption function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. This notation means we are mapping a k-bit input (the key) and an n-bit input (the plaintext message) to an n-bit output (the ciphertext). Once we fix the key K, we get a function mapping n bits to n bits: $E_K : \{0,1\}^n \to \{0,1\}^n$ defined by $E_K(M) = E(K, M)$. E_K is required to be a *permutation* on the n-bit strings. In other words, it must be an invertible (bijective) function. The inverse mapping of this permutation is the decryption algorithm D_K. In other words, decryption is the reverse of encryption: $D_K(E_K(M)) = M$.

The block cipher as defined above is a category of functions, meaning that there are many different implementations of a block cipher. Today, the most commonly used block cipher implementation is called Advanced Encryption Standard (AES). It was designed in 1998 by Joan Daemen and Vincent Rijmen, two researchers from Belgium, in response to a competition organized by NIST.[5]

AES uses a block length of $n = 128$ bits and a key length of $k = 128$ bits. It can also support $k = 192$ or $k = 256$ bit keys, but we will assume 128-bit keys in this class. It was designed to be extremely fast in both hardware and software.

6.5 Block Cipher Security

Block ciphers, including AES, are not IND-CPA secure on their own because they are deterministic. In other words, encrypting the same message twice with the same key produces the same output twice. The strategy that an adversary, Eve, uses to break the security of AES is exactly the same as the strategy from the one-time pad with key reuse. Eve sends

[4] Answer: The key is needed to determine which scrambling setting was used to generate the ciphertext. If decryption didn't require a key, any attacker would be able to decrypt encrypted messages!

[5] Fun fact: Professor David Wagner, who sometimes teaches this class, was part of the team that came up with a block cipher called TwoFish, which was one of the finalists in the NIST competition.

M_0 and M_1 to the challenger and receives either $E(K, M_0)$ or $E(K, M_1)$. She then queries the challenger for the encryption of M_0 and receives $E(K, M_0)$. If the two encryptions she receives from the challenger are the same, then Eve knows the challenger encrypted M_0 and sent $E(K, M_0)$. If the two encryptions are different, then Eve knows the challenger encrypted M_1 and sent $E(K, M_1)$. Thus Eve can win the IND-CPA game with probability $100\% > 1/2$, and the block cipher is not IND-CPA secure.

Although block ciphers are not IND-CPA secure, they have a desirable security property that will help us build IND-CPA secure symmetric encryption schemes: namely, a block cipher is *computationally indistinguishable* from a random permutation. In other words, for a fixed key K, E_K "behaves like" a random permutation on the n-bit strings.

A random permutation is a function that maps each n-bit input to exactly one random n-bit output. One way to generate a random permutation is to write out all 2^n possible inputs in one column and all 2^n possible outputs in another column, and then draw 2^n random lines connecting each input to each output. Once generated, the function itself is not random: given the same input twice, the function gives the same output twice. However, the choice of which output is given is randomly determined when the function is created.

Formally, we perform the following experiment to show that a block cipher is indistinguishable from a random permutation. The adversary, Eve, is given a box which contains either (I) the encryption function E_K with a randomly chosen key K, or (II) a permutation π on n bits chosen uniformly at random when the box was created (in other words, map each n-bit input to a different random n-bit output). The type of box given to Eve is randomly selected, but we don't tell Eve which type of box she has been given. We also don't tell Eve the value of the key K.

Eve is now allowed to play with the box as follows: Eve can supply an input x to the box and receive a corresponding output y from the box (namely, $y = E_K(x)$ for a type-I box, or $y = \pi(x)$ for a type-II box). After playing with the box, Eve must guess whether the box is type I or type II. If the block cipher is truly indistinguishable from random, then Eve cannot guess which type of box she received with probability greater than $1/2$.

AES is not truly indistinguishable from random, but it is believed to be *computationally* indistinguishable from random. Intuitively, this means that given a practical amount of computation power (e.g. polynomially-bounded runtime), Eve cannot guess which type of box she received with probability greater than $1/2$. Another way to think of computational indistinguishability is: Eve can guess which type of box she received with probability $1/2$, plus some negligible amount (e.g. $1/2^{128}$). With infinite computational time and power, Eve could leverage this tiny $1/2^{128}$ advantage to guess which box she received, but with only a practical amount of computation power, this advantage is useless for Eve.

The computational indistinguishability property of AES gives us a strong security guarantee: given a single ciphertext $C = E_K(M)$, an attacker without the key cannot learn anything about the original message M. If the attacker could learn something about M, then AES would no longer be computationally indistinguishable: in the experiment from before, Eve could feed M into the box and see if given only the output from the box, she can learn something about M. If Eve learns something about M, then she knows the output came

from a block cipher. If Eve learns nothing about M, then she knows the output came from a random permutation. However, since we believe that AES is computationally indistinguishable from random, we can say that an attacker who receives a ciphertext learns *nothing* about the original message.

There is no proof that AES is computationally indistinguishable from random, but it is believed to be computationally indistinguishable. After all these years, the best known attack is still *exhaustive key search*, where the attacker systematically tries decrypting some ciphertext using every possible key to see which one gives intelligible plaintext. Given infinite computational time and power, exhaustive key search can break AES, which is why it is not truly indistinguishable from random. However, with a 128-bit key, exhaustive key search requires 2^{128} computations in the worst case (2^{127} on average). This is a large enough number that even the fastest current supercomputers couldn't possibly mount an exhaustive key search attack against AES within the lifetime of our Solar system.

Thus, AES behaves very differently than the one-time pad. Even given a very large number of plaintext/ciphertext pairs, there appears to be no effective way to decrypt any new ciphertexts. We can leverage this property to build symmetric-key encryption schemes where there is no effective way to decrypt *any* ciphertext, even if it's the encryption of a message we've seen before.

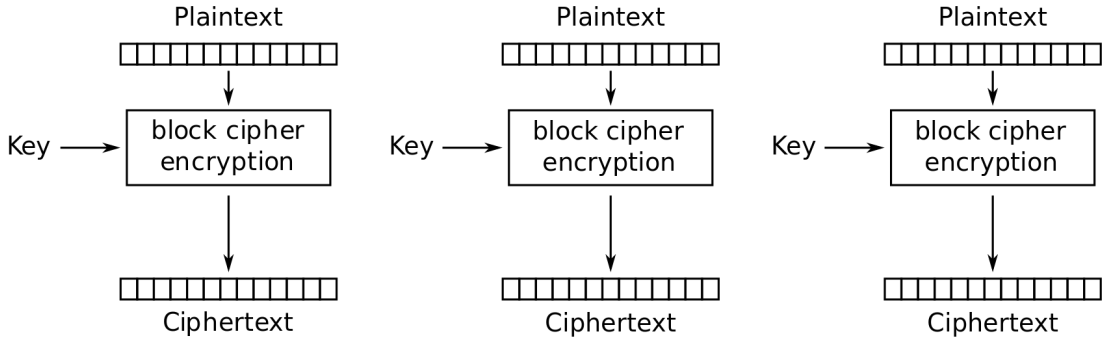
6.6 Block Cipher Modes of Operation

There are two main reasons AES by itself cannot be a practical IND-CPA secure encryption scheme. The first is that we'd like to encrypt arbitrarily long messages, but the block cipher only takes fixed-length inputs. The other is that if the same message is sent twice, the ciphertext in the two transmissions is the same with AES (i.e. it is deterministic). To fix these problems, the encryption algorithm can either be randomized or stateful—it either flips coins during its execution, or its operation depends upon some state information. The decryption algorithm, however, is neither randomized nor stateful.

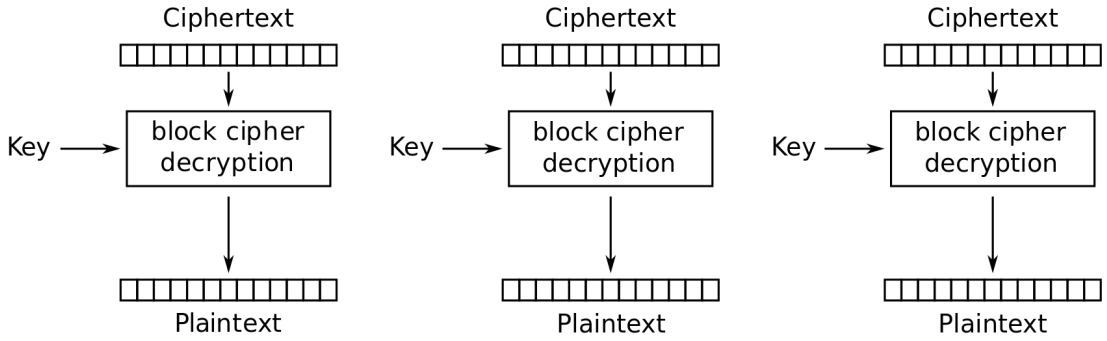
There are several standard ways (or modes of operation) of building an encryption algorithm, using a block cipher:

ECB Mode (Electronic Code Book): In this mode the plaintext M is simply broken into n-bit blocks $M_1 \cdots M_l$, and each block is encoded using the block cipher: $C_i = E_K(M_i)$. The ciphertext is just a concatenation of these individual blocks: $C = C_1 \cdot C_2 \cdots C_l$. This scheme is **flawed**. Any redundancy in the blocks will show through and allow the eavesdropper to deduce information about the plaintext. For instance, if $M_i = M_j$, then we will have $C_i = C_j$, which is visible to the eavesdropper; so ECB mode **leaks information** about the plaintext.

- ECB mode encryption: $C_i = E_K(M_i)$
- ECB mode decryption: $M_i = D_K(C_i)$



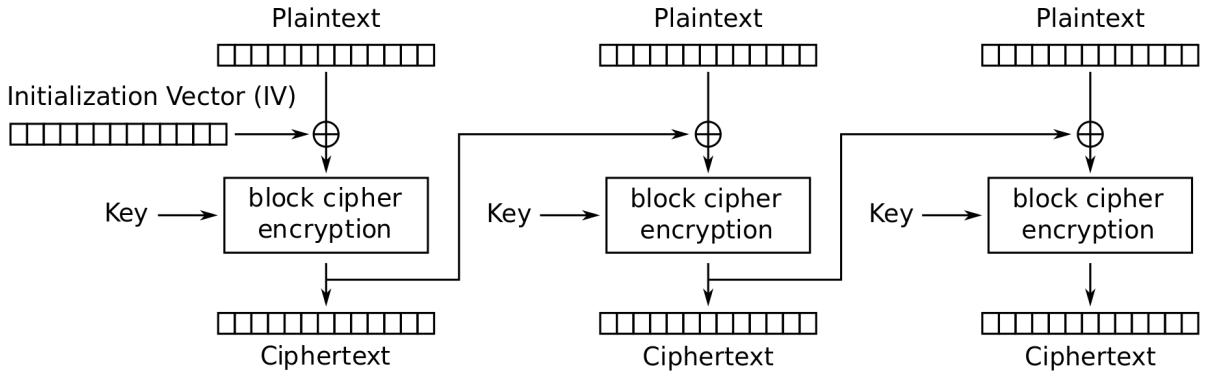
Electronic Codebook (ECB) mode encryption



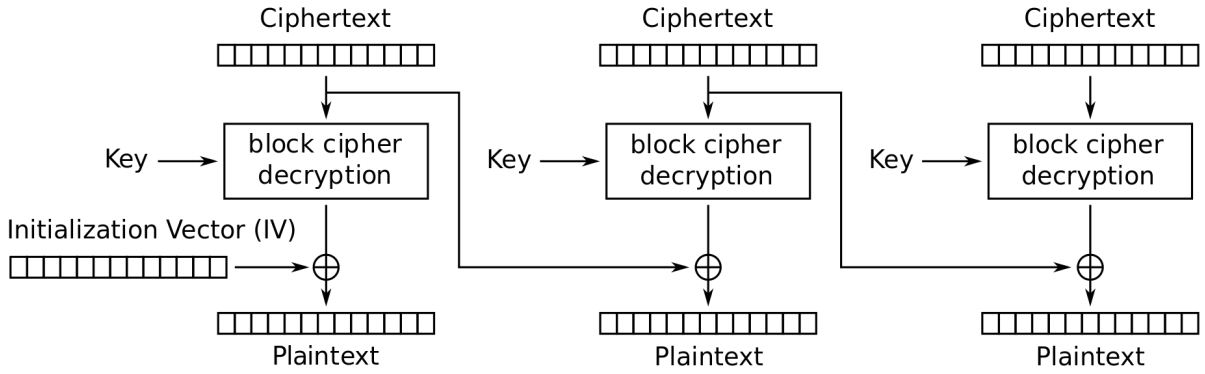
Electronic Codebook (ECB) mode decryption

CBC Mode (Cipher Block Chaining): This is a popular mode for commercial applications. For each message the sender picks a random n-bit string, called the *initial vector* or IV. Define $C_0 = IV$. The i^{th} ciphertext block is given by $C_i = E_K(C_{i-1} \oplus M_i)$. The ciphertext is the concatenation of the initial vector and these individual blocks: $C = IV \cdot C_1 \cdot C_2 \cdots C_l$. CBC mode has been proven to provide strong security guarantees on the privacy of the plaintext message (assuming the underlying block cipher is secure).

- CBC mode encryption: $\begin{cases} C_0 = IV \\ C_i = E_K(P_i \oplus C_{i-1}) \end{cases}$
- CBC mode decryption: $P_i = D_K(C_i) \oplus C_{i-1}$



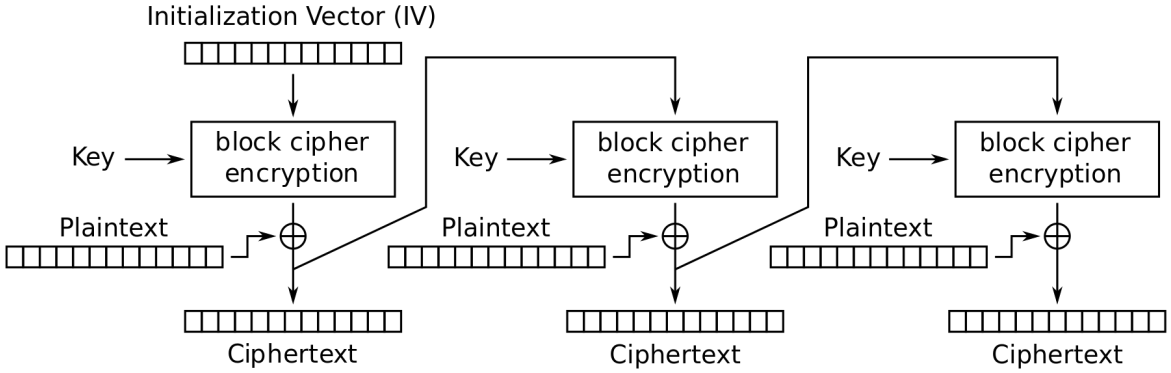
Cipher Block Chaining (CBC) mode encryption



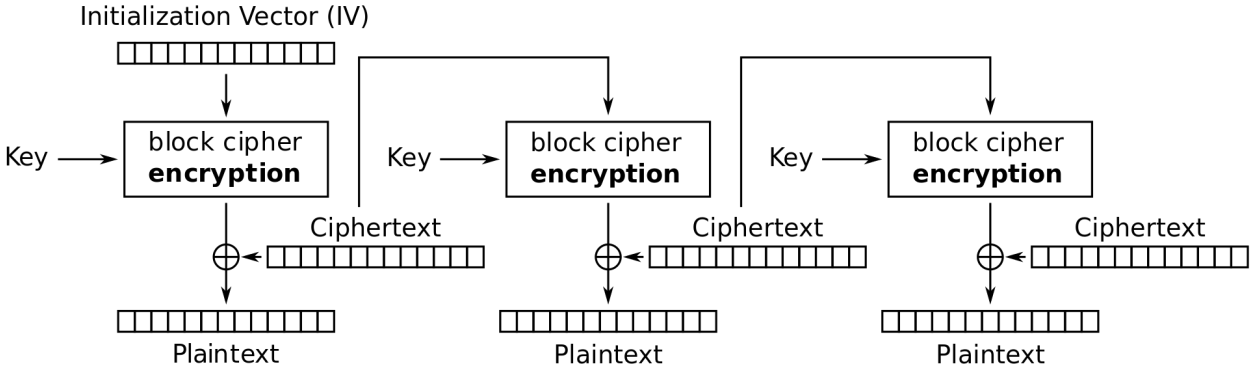
Cipher Block Chaining (CBC) mode decryption

CFB Mode (Ciphertext Feedback Mode): This is another popular mode with properties very similar to CBC mode. Again, C_0 is the IV. The i^{th} ciphertext block is given by $C_i = E_K(C_{i-1}) \oplus M_i$.

- CFB mode encryption: $\begin{cases} C_0 = IV \\ C_i = E_K(C_{i-1}) \oplus P_i \end{cases}$
- CFB mode decryption: $P_i = E_K(C_{i-1}) \oplus C_i$



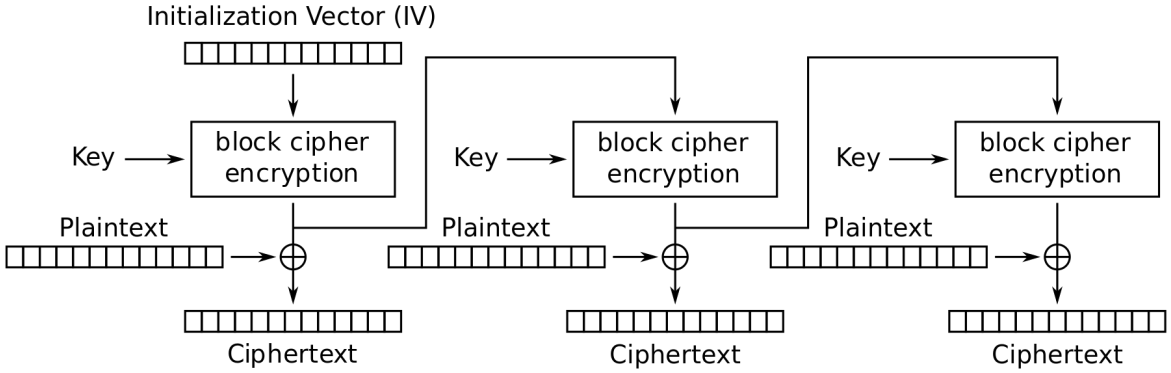
Cipher Feedback (CFB) mode encryption



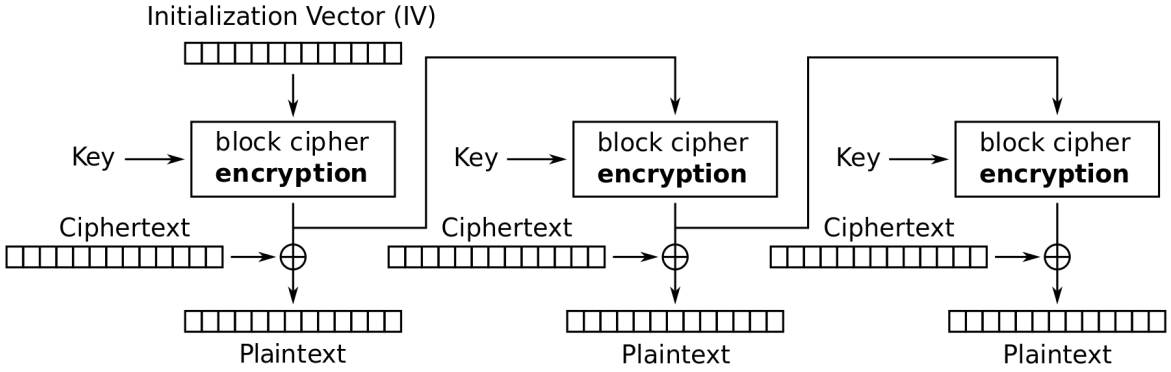
Cipher Feedback (CFB) mode decryption

OFB Mode (Output Feedback Mode): In this mode, the initial vector IV is repeatedly encrypted to obtain a set of values Z_i as follows: $Z_0 = IV$ and $Z_i = E_K(Z_{i-1})$. These values Z_i are now used as though they were the key for a one-time pad, so that $C_i = Z_i \oplus M_i$. The ciphertext is the concatenation of the initial vector and these individual blocks: $C = IV \cdot C_1 \cdot C_2 \cdots C_l$. In OFB mode, it is very easy to tamper with ciphertexts. For instance, suppose that the adversary happens to know that the j^{th} block of the message, M_j, specifies the amount of money being transferred to his account from the bank, and suppose he also knows that $M_j = 100$. Since he knows both M_j and C_j, he can determine Z_j. He can then substitute any n-bit block in place of M_j and get a new ciphertext C'_j where the 100 is replaced by any amount of his choice. This kind of tampering is also possible with other modes of operation as well (so don't be fooled into thinking that CBC mode is safe from tampering); it's just easier to illustrate on OFB mode.

- OFB mode encryption: $\begin{cases} Z_0 = IV \\ Z_i = E_K(Z_{i-1}) \\ C_i = M_i \oplus Z_i \end{cases}$
- OFB mode decryption: $P_i = C_i \oplus Z_i$



Output Feedback (OFB) mode encryption

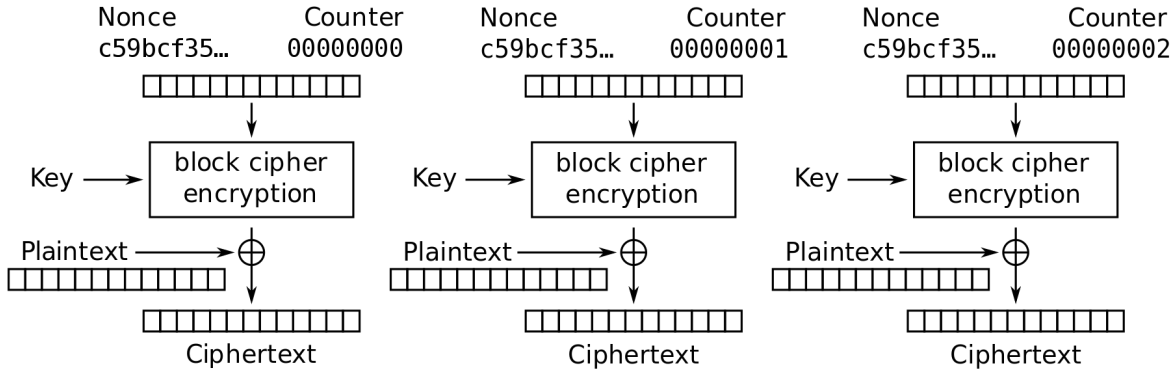


Output Feedback (OFB) mode decryption

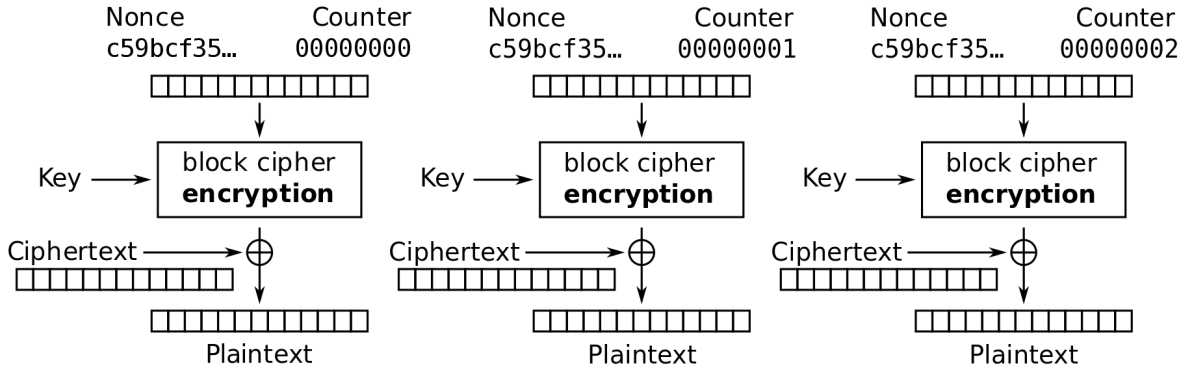
Counter (CTR) Mode: In CTR mode, a counter is initialized to IV and repeatedly incremented and encrypted to obtain a sequence that can now be used as though they were the keys for a one-time pad: namely, $Z_i = E_K(IV + i)$ and $C_i = Z_i \oplus M_i$. In CTR mode, the IV is sometimes renamed the *nonce*. This is just a terminology difference–nonce and IV can be used interchangeably for the purposes of this class.

Note that in CTR and OFB modes, the decryption algorithm uses the block cipher *encryption* function instead of the decryption function. Intuitively, this is because Alice used the encryption function to generate a one-time pad, so Bob should also use the encryption function to generate the same pad. The plaintext is never passed through the block cipher encryption, so the block cipher decryption is never used.

- CTR mode encryption: $C_i = E_K(IV + i) \oplus M_i$
- CTR mode decryption: $M_i = E_K(IV + i) \oplus C_i$



Counter (CTR) mode encryption



Counter (CTR) mode decryption

For the rest of these notes, we will focus on analyzing CBC and CTR modes. As an exercise, you can try performing similar analysis on the other modes as well.

6.7 Parallelization

In some modes, successive blocks must be encrypted or decrypted sequentially. In other words, to encrypt the ith block of plaintext, you first need to encrypt the $i-1$th block of plaintext and see the $i-1$th block of ciphertext output. For high-speed applications, it is often useful to parallelize encryption and decryption.

Of the schemes described above, which ones have parallelizable encryption? Which ones have parallelizable decryption?

CBC mode encryption cannot be parallelized. By examining the encryption equation $C_i = E_K(P_i \oplus C_{i-1})$, we can see that to calculate C_i, we first need to know the value of C_{i-1}. In other words, we have to encrypt the $i-1$th block first before we can encrypt the ith block.

CBC mode decryption can be parallelized. Again, we examine the decryption equation $P_i = D_K(C_i) \oplus C_{i-1}$. To calculate P_i, we need C_i and C_{i-1}. Neither of these values need to

be calculated–when we're decrypting, we already have all of the ciphertext blocks. Thus we can compute all the P_i in parallel.

CTR mode encryption and decryption can both be parallelized. To see this, we can examine the encryption and decryption diagrams. Note that each block cipher only takes the nonce and counter as input, and there is no reliance on any previous ciphertext or plaintext.

6.8 Padding

We have already reasoned that block ciphers let us encrypt messages that are longer than one block long. What happens if we want to send a message that is not a multiple of the block size? It turns out the answer depends on which mode is being used. For this section, assume that the block size is 128 bits, or 16 bytes (16 characters).

In CBC mode, if the plaintext length isn't a multiple of 128 bits, then the last block of plaintext will be slightly shorter than 128 bits. Then the XOR between the 128-bit previous ciphertext and the less-than-128-bit last block of plaintext would be undefined–bitwise XOR only works if the two inputs being XORed are the same length.

Suppose the last block of plaintext is only 100 bits. What if we just XOR the first 100 bits of the previous ciphertext with the 100 bits of plaintext, and ignore the last 28 bits of the previous ciphertext? Now we have a 100-bit input to the block cipher, which only takes 128-bit inputs. This input is undefined for the block cipher.

The solution to this problem is to add padding to the plaintext until it is a multiple of 128 bits.

If we add padding to make the plaintext a multiple of 128 bits, we will need to be able to remove the padding later to correctly recover the original message. Some forms of padding can create ambiguity: for example, consider a padding scheme where we pad a message with all 1s. What happens if we need to pad a message `0000000010111`? We would add 1s until it's a multiple of the block size, e.g. `0000000010111111`. When we try to depad the message, we run into some ambiguity. How many 1s do we remove from the end of the message? It's unclear.

One correct padding scheme is PKCS#7[6] padding. In this scheme, we pad the message by the number of padding bytes used. For example, the message above would be padded as `0000000010111333`, because 3 bytes of padding were needed. To remove the padding, we note that the message ends in a 3, so 3 bytes of padding were used, so we can unambiguously remove the last 3 bytes of padding. Note that if the message is already a multiple of a block size, an entire new block is appended. This way, there is always one unique padding pattern at the end of the message.

Not all modes need padded plaintext input. For example, let's look at CTR mode next. Again, suppose we only have 100 bits in your last block of plaintext. This time, we can actually XOR the 100 bits of plaintext with the first 100 bits of block cipher output, and ignore the last 28 bits of block cipher output. Why? Because the result of the XOR never

[6] PKCS stands for Public Key Cryptography Standards.

has to be passed into a block cipher again, so we don't care if it's slightly shorter than 128 bits. The last ciphertext block will just end up being 100 bits instead of 128 bits, and that's okay because it's never used as an input to a block cipher.

How does decryption work? From our encryption step, the last ciphertext block is only 100 bits instead of 128 bits. Then to retrieve the last 100 bits of plaintext, all we have to do is XOR the 100 bits of ciphertext with the first 100 bits of the block cipher output and ignore the last 28 bits of block cipher output.

Recall that CTR mode can be thought of as generating a one-time pad through block ciphers. If the pad is too long, you can just throw away the last few bits of the pad in both the encryption and decryption steps.

6.9 Reusing IVs is insecure

Remember that ECB mode is not IND-CPA secure because it is deterministic. Encrypting the same plaintext twice always results in the same output, and this causes information leakage. All the other modes introduce a random initialization vector (IV) that is different on every encryption in order to ensure that encrypting the same plaintext twice with the same key results in different output.

This also means that when using secure block cipher modes, it is important to always choose a different, random, unpredictable IV for each new encryption. If the same IV is reused, the scheme becomes deterministic, and information is potentially leaked. The severity of information leakage depends on what messages are being encrypted and which mode is being used.

For example, in CTR mode, reusing the IV (nonce) is equivalent to reusing the one-time pad. An attacker who sees two different messages encrypted with the same IV will know the bitwise XOR of the two messages. However, in CBC mode, reusing the IV on two different messages only reveals if two messages start with the same blocks, up until the first difference.

Different modes have different tradeoffs between usability and security. Although proper use of CBC and CTR mode are both IND-CPA, insecure use of either mode (e.g. reusing the IV) breaks IND-CPA security, and the severity of information leakage is different in the two modes. In CBC mode, the information leakage is contained, but in CTR mode, the leakage is catastrophic (equivalent to reusing a one-time pad). On the other hand, CTR mode can be parallelized, but CBC can not, which is why many high performance systems use CTR mode or CTR-mode based encryption schemes.

7 Cryptographic Hashes

7.1 Overview

A cryptographic hash function is a function, H, that when applied on a message, M, can be used to generate a fixed-length "fingerprint" of the message. As such, any change to the message, no matter how small, will change many of the bits of the hash value with there being no detectable patterns as to how the output changes based on specific input changes. In other words, any changes to the message, M, will change the resulting hash-value in some seemingly random way.

The hash function, H, is deterministic, meaning if you compute $H(M)$ twice with the same input M, you will always get the same output twice. The hash function is unkeyed, as it only takes in a message M and no secret key. This means anybody can compute hashes on any message.

Typically, the output of a hash function is a fixed size: for instance, the SHA256 hash algorithm can be used to hash a message of any size, but always produces a 256-bit hash value. In a secure hash function, the output of the hash function looks like a random string, chosen differently and independently for each message—except that, of course, a hash function is a deterministic procedure.

To understand the intuition behind hash functions, let's take a look at one of its many uses: document comparisons. Suppose Alice and Bob both have a large, 1 GB document and wanted to know whether the files were the same. While they could go over each word in the document and manually compare the two, hashes provide a quick and easy alternative. Alice and Bob could each compute a hash over the document and securely communicate the hash values to one another. Then, since hash functions are deterministic, if the hashes are the same, then the files must be the same since they have the same "fingerprint". On the other hand, if the hashes are different, it must be the case that the files are different. Determinism in hash functions ensures that providing the same input twice (i.e. providing the same document) will result in the same hash value; however, providing different inputs (i.e. providing two different documents) will result in two different hash values.

7.2 Properties of Hash Functions

Cryptographic hash functions have several useful properties. The most significant include the following:

- **One-way:** The hash function can be computed efficiently: given x, it is easy to compute $H(x)$. However, given a hash output y, it is infeasible to find any input x such that $H(x) = y$. (This property is also known as "**preimage resistant**.") Intuitively, the one-way property claims that given an output of a hash function, it is infeasible for an adversary to find *any* input that hashes to the given output.
- **Second preimage resistant:** Given an input x, it is infeasible to find another input x' such that $x' \neq x$ but $H(x) = H(x')$. This property is closely related to *preimage*

resistance; the difference is that here the adversary also knows a starting point, x, and wishes to tweak it to x' in order to produce the same hash—but cannot. Intuitively, the second preimage resistant property claims that given an input, it is infeasible for an adversary to find another input that has the same hash value as the original input.

- **Collision resistant:** It is infeasible to find *any* pair of messages x, x' such that $x' \neq x$ but $H(x) = H(x')$. Again, this property is closely related to the previous ones. Here, the difference is that the adversary can freely choose their starting point, x, potentially designing it specially to enable finding the associated x'—but again cannot. Intuitively, the collision resistance property claims that it is infeasible for an adversary to find *any* two inputs that both hash to the same value. While it is impossible to design a hash function that has absolutely no collisions since there are more inputs than outputs (remember the pigeonhole principle), it is possible to design a hash function that makes finding collisions *infeasible* for an attacker.

By "infeasible", we mean that there is no known way to accomplish it with any realistic amount of computing power.

Note, the third property implies the second property. Cryptographers tend to keep them separate because a given hash function's resistance towards the one might differ from its resistance towards the other (where resistance means the amount of computing power needed to achieve a given chance of success).

Under certain threat models, hash functions can be used to verify message integrity. For instance, suppose Alice downloads a copy of the installation disk for the latest version of the Ubuntu distribution, but before she installs it onto her computer, she would like to verify that she has a valid copy of the Ubuntu software and not something that was modified in transit by an attacker. One approach is for the Ubuntu developers to compute the SHA256 hash of the intended contents of the installation disk, and distribute this 256-bit hash value over many channels (e.g., print it in the newspaper, include it on their business cards, etc.). Then Alice could compute the SHA256 hash of the contents of the disk image she has downloaded, and compare it to the hash publicized by Ubuntu developers. If they match, then it would be reasonable for Alice to conclude that she received a good copy of the legitimate Ubuntu software. Because the hash is collision-resistant, an attacker could not change the Ubuntu software while keeping the hash the same. Of course, this procedure only works if Alice has a good reason to believe that she has the correct hash value, and it hasn't been tampered with by an adversary. If we change our threat model to allow the adversary to tamper with the hash, then this approach no longer works. The adversary could simply change the software, hash the changed software, and present the changed hash to Alice.

7.3 Hash Algorithms

Cryptographic hashes have evolved over time. One of the earliest hash functions, MD5 (Message Digest 5) was broken years ago. The slightly more recent SHA1 (Secure Hash Algorithm) was broken in 2017, although by then it was already suspected to be insecure. Systems which rely on MD5 or SHA1 actually resisting attackers are thus considered insecure. Outdated hashes have also proven problematic in non-cryptographic systems. The `git`

version control program, for example, identifies identical files by checking if the files produce the same SHA1 hash. This worked just fine until someone discovered how to produce SHA1 collisions.

Today, there are two primary "families" of hash algorithms in common use that are believed to be secure: SHA2 and SHA3. Within each family, there are differing output lengths. SHA-256, SHA-384, and SHA-512 are all instances of the SHA2 family with outputs of 256b, 384b, and 512b respectively, while SHA3-256, SHA3-384, and SHA3-512 are the SHA3 family members.

For the purposes of the class, we don't care which of SHA2 or SHA3 we use, although they are in practice very different functions. The only significant difference is that SHA2 is vulnerable to a *length extension attack*. Given $H(M)$ and the length of the message, but not M itself, there are circumstances where an attacker can compute $H(M||M')$ for an arbitrary M' of the attacker's choosing. This is because SHA2's output reflects all of its internal state, while SHA3 includes internal state that is never outputted but only used in the calculation of subsequent hashes. While this does not violate any of the aforementioned properties of hash functions, it is undesirable in some circumstances.

In general, we prefer using a hash function that is related to the length of any associated symmetric key algorithm. By relating the hash function's output length with the symmetric encryption algorithm's key length, we can ensure that it is equally difficult for an attacker to break either scheme. For example, if we are using AES-128, we should use SHA-256 or SHA3-256. Assuming both functions are secure, it takes 2^{128} operations to brute-force a 128 bit key and 2^{128} operations to generate a hash collision on a 256 bit hash function. For longer key lengths, however, we may relax the hash difficulty. For example, with 256b AES, the NSA uses SHA-384, not SHA-512, because, let's face it, 2^{196} operations is already a hugely impractical amount of computation.

7.4 Lowest-hash scheme

Cryptographic hashes have many practical applications outside of cryptography. Here's one example that illustrates many useful properties of cryptographic hashes.

Suppose you are a journalist, and a hacker contacts you claiming to have stolen 150 million records from a website. The hacker is keeping the records for ransom, so they don't want to present all 150 million files to you. However, they still wish to prove to you that they have actually stolen 150 million different records, and they didn't steal significantly fewer records and exaggerate the number. How can you be sure that the hacker isn't lying, without seeing all 150 million records?

Remember that the outputs of cryptographic hashes look effectively random–two different inputs, even if they only differ in one bit, give two unpredictably different outputs. Can we use these random-looking outputs to our advantage?

Consider a box with 100 balls, numbered from 1 to 100. You draw a ball at random, observe the value, and put it back. You repeat this n times, then report the lowest number you saw in the n draws. If you drew 10 balls (n=10), you would expect the lowest number to be

approximately 10. If you drew 100 balls (n=100), you might expect the lowest number to be somewhere in the range 1-5. If you drew 150 million balls (n=150,000,000), you would be pretty sure that the lowest number was 1. Someone who claims to have drawn 150 million balls and seen a lowest number of 50 has either witnessed an astronomically unlikely event, or is lying about their claim.

We can apply this same idea to hashes. If the hacker hashes all 150 million records, they are effectively generating 150 million unpredictable fixed-length bitstrings, just like drawing balls from the box 150 million times. With some probability calculations (out of scope for this class), we can determine the expected range of the lowest hash values, as well as what values would be astronomically unlikely to be the lowest of 150 million random hashes.

With this idea in mind, we might ask the hacker to hash all 150 million records with a cryptographic hash and return the 10 lowest resulting hashes. We can then check if those hashes are consistent with what we would expect the lowest 10 samples out of 150 million random bitstrings to be. If the hacker only hashed 15 million records and returned the lowest 10 hashes, we should notice that the probability of getting those 10 lowest hashes from 150 million records is astronomically low and conclude that the hacker is lying about their claim.

What if the hacker tries to cheat? If the hacker only has 15 million records, they might try to generate 150 million fake records, hash the fake records, and return the lowest 10 hashes to us. We can make this attack much harder for the attacker by requiring that the attacker also send the 10 records corresponding to the lowest hashes. The hacker won't know which of these 150 million fake records results in the lowest hash, so to guarantee that they can fool the reporter, all 150 million fake records would need to look valid to the reporter. Depending on the setting, this can be very hard or impossible: for example, if we expect the records to be in a consistent format, e.g. `lastname, firstname`, then the attacker would need to generate 150 million fake records that follow the same format.

Still, the hacker might decide to spend some time *precomputing* fake records with low hashes before making a claim. This is called an *offline attack*, since the attacker is generating records offline before interacting with the reporter. We will see more offline attacks when we discuss password hashing later in the notes. We can prevent the offline attack by having the reporter choose a random word at the start of the interaction, like "fubar," and send it to the hacker. Now, instead of hashing each record, the hacker will hash each record, concatenated with the random word. The reporter will give the attacker just enough time to compute 150 million hashes (but no more) before requesting the 10 lowest values. Now, a cheating hacker cannot compute values ahead of time, because they won't know what the random word is.

A slight variation on this method is to hash each record 10 separate times, each with a different reporter-chosen random word concatenated to the end (e.g. "fubar-1," "fubar-2," "fubar-3," etc.). In total, the hacker is now hashing 1.5b (150 million times 10) records. Then, instead of returning the lowest 10 hashes overall, the hacker returns the record with the lowest hash for each random word. Another way to think of this variation is: the hacker hashes all 150 million records with the first random word concatenated to each record, and returns the record with the lowest hash. Then the hacker hashes all 150 million records again with the second random word concatenated to each record, and returns the record with the

lowest hash. This process repeats 10 times until the hacker has presented 10 hashes. The math for using the hash values to estimate the total number of lines is slightly different in this variation (the original uses random selection without substitution, while the variant uses random selection with substitution), but the underlying idea is the same.

8 Message Authentication Codes (MACs)

When building cryptographic schemes that guarantee integrity and authentication, the threat we're concerned about is adversaries who send messages pretending to be from a legitimate participant (*spoofing*) or who modify the contents of a message sent by a legitimate participant (*tampering*). To address these threats, we will introduce cryptographic schemes that enable the recipient to detect spoofing and tampering.

In this section, we will define *message authentication codes (MACs)* and show how they guarantee integrity and authenticity. Because MACs are a symmetric-key cryptographic primitive, in this section we can assume that Alice and Bob share a secret key that is not known to anyone else. Later we will see how Alice and Bob might securely exchange a shared secret key over an insecure communication channel, but for now you can assume that only Alice and Bob know the value of the secret key.

8.1 MAC: Definition

A MAC is a keyed checksum of the message that is sent along with the message. It takes in a fixed-length secret key and an arbitrary-length message, and outputs a fixed-length checksum. A secure MAC has the property that any change to the message will render the checksum invalid.

Formally, the MAC on a message M is a value $F(K, M)$ computed from K and M; the value $F(K, M)$ is called the *tag* for M or the MAC of M. Typically, we might use a 128-bit key K and 128-bit tags.

When Alice wants to send a message with integrity and authentication, she first computes a MAC on the message $T = F(K, M)$. She sends the message and the MAC $\langle M, T \rangle$ to Bob. When Bob receives $\langle M, T \rangle$, Bob will recompute $F(K, M)$ using the M he received and check that it matches the MAC T he received. If it matches, Bob will accept the message M as valid, authentic, and untampered; if $F(K, M) \neq T$, Bob will ignore the message M and presume that some tampering or message corruption has occurred.

Note that MACs must be deterministic for correctness–when Alice calculates $T = F(K, M)$ and sends $\langle M, T \rangle$ to Alice, Bob should get the same result when he calculates $F(K, M)$ with the same K and M.

MACs can be used for more than just communication security. For instance, suppose we want to store files on a removable USB flash drive, which we occasionally share with our friends. To protect against tampering with the files on our flash drive, our machine could generate a secret key and store a MAC of each file somewhere on the flash drive. When our machine reads the file, it could check that the MAC is valid before using the file contents. In a sense, this is a case where we are "communicating" to a "future version of ourselves," so security for stored data can be viewed as a variant of communication security.

8.2 MAC: Security properties

Given a secure MAC algorithm F, if the attacker replaces M by some other message M', then the tag will almost certainly[7] no longer be valid: in particular, $F(K, M) \neq F(K, M')$ for any $M' \neq M$.

More generally, there will be no way for the adversary to modify the message and then make a corresponding modification to the tag to trick Bob into accepting the modified message: given M and $T = F(K, M)$, an attacker who does not know the key K should be unable to find a different message M' and a tag T' such that T' is a valid tag on M' (i.e., such that $T' = F(K, M')$). Secure MACs are designed to ensure that even small changes to the message make unpredictable changes to the tag, so that the adversary cannot guess the correct tag for their malicious message M'.

Recall that MACs are deterministic–if Alice calculates $F(K, M)$ twice on the same message M, she will get the same MAC twice. This means that an attacker who sees a pair $M, F(K, M)$ will know a valid MAC for the message M. However, if the MAC is secure, the attacker should be unable to create valid MACs for messages that they have never seen before.

More generally, secure MACs are designed to be secure against known-plaintext attacks. For instance, suppose an adversary Eve eavesdrops on Alice's communications and observes a number of messages and their corresponding tags: $\langle M_1, T_1 \rangle, \langle M_2, T_2 \rangle, \ldots, \langle M_n, T_n \rangle$, where $T_i = F(K, M_i)$. Then Eve has no hope of finding some new message M' (such that $M' \notin \{M_1, \ldots, M_n\}$) and a corresponding value T' such that T' is the correct tag on M' (i.e., such that $T' = F(K, M')$). The same is true even if Eve was able to choose the M_i's. In other words, even though Eve may know some valid MACs $\langle M_n, T_n \rangle$, she still cannot generate valid MACs for messages she has never seen before.

Here is a formal security definition that captures both properties described above. We imagine a game played between Georgia (the adversary) and Reginald (the referee). Initially, Reginald picks a random key K, which will be used for all subsequent rounds of the game. In each round of the game, Georgia may query Reginald with one of two kinds of queries:

- **Generation query:** Georgia may specify a message M_i and ask for the tag for M_i. Reginald will respond with $T_i = F(K, M_i)$.
- **Verification query:** Alternatively, Georgia may specify a pair of values $\langle M_i, T_i \rangle$ and ask Reginald whether T_i is a valid tag on M_i. Reginald checks whether $T_i \stackrel{?}{=} F(K, M_i)$ and responds "Yes" or "No" accordingly.

Georgia is allowed to repeatedly interact with Reginald in this way. Georgia wins if she ever asks Reginald a verification query $\langle M_n, T_n \rangle$ where Reginald responds "Yes", and where M_n did not appear in any previous generation query to Reginald. In this case, we say that Georgia has successfully forged a tag. If Georgia can successfully forge, then the MAC

[7] Strictly speaking, there is a very small chance that the tag for M will also be a valid tag for M'. However, if we choose tags to be long enough—say, 128 bits—and if the MAC algorithm is secure, the chances of this happening should be about $1/2^{128}$, which is small enough that it can be safely ignored.

algorithm is insecure. Otherwise, if there is no strategy that allows Georgia to forge (given a generous allotment of computation time and any reasonable number of rounds of the game), then we say that the MAC algorithm is secure.

This game captures the idea that Georgia the Forger can try to observe the MAC tag on a bunch of messages, but this won't help her forge a valid tag on any new message. In fact, even if Georgia carefully selects a bunch of chosen messages and gets Alice to transmit those messages (i.e., she gets Alice to compute the MAC on those messages with her key, and then transmit those MAC tags), it still won't help Georgia forge a valid tag on any new message. Thus, MACs provide security against chosen-plaintext/ciphertext attacks, the strongest threat model.

8.3 AES-EMAC

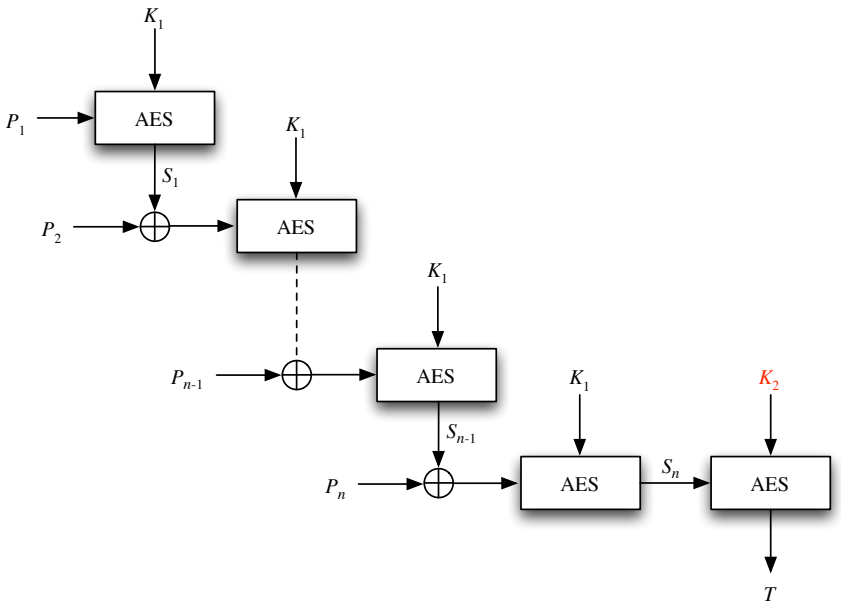
How do we build secure MACs?

There are a number of schemes out there, but one good one is AES-CMAC, an algorithm standardized by NIST. Instead of showing you AES-CMAC, we'll look at a related algorithm called AES-EMAC. AES-EMAC is a slightly simplified version of AES-CMAC that retains its essential character but differs in a few details.

In AES-EMAC, the key K is 256 bits, viewed as a pair of 128-bit AES keys: $K = \langle K_1, K_2 \rangle$. The message M is decomposed into a sequence of 128-bit blocks: $M = P_1||P_2||...||P_n$. We set $S_0 = 0$ and compute

$$S_i = \text{AES}_{K_1}(S_{i-1} \oplus P_i), \qquad \text{for } i = 1, 2, \ldots, n.$$

Finally we compute $T = \text{AES}_{K_2}(S_n)$; T is the tag for message M. Here is what it looks like:



Assuming AES is a secure block cipher, this scheme is provably secure, using the unforgeability definition and security game described in the previous section. An attacker cannot forge a valid AES-EMAC for a message they haven't seen before, even if they are allowed to query for MACs of other messages.

8.4 HMAC

One of the best MAC constructions available is the HMAC, or Hash Message Authentication Code, which uses the cryptographic properties of a cryptographic hash function to construct a secure MAC algorithm.

HMAC is an excellent construction because it combines the benefits of both a MAC and the underlying hash. Without the key, the tag does not leak information about the message. Even with the key, it is computationally intractable to reconstruct the message from the hash output.

There are several specific implementations of HMAC that use different cryptographic hash functions: for example, HMAC_SHA256 uses SHA256 as the underlying hash, while HMAC_SHA3_256 uses SHA3 in 256 bit mode as the underlying hash function. The choice of underlying hash depends on the application. For example, if we are using HMACs with a block cipher, we would want to choose an HMAC whose output is twice the length of the keys used for the associated block cipher, so if we are encrypting using AES_192 we should use HMAC_SHA_384 or HMAC_SHA3_384.

The output of HMAC is the same number of bits as the underlying hash function, so in both of these implementations it would be 256 bits of output. In this section, we'll denote the number of bits in the hash output as n.

To construct the HMAC algorithm, we first start with a more general version, NMAC:

$$\text{NMAC}(K_1, K_2, M) = H(K_1||H(K_2||M))$$

In words, NMAC concatenates K_2 and M, hashes the result, concatenates the result with K_1, and then hashes that result.

Note that NMAC takes two keys, K_1 and K_2, both of length n (the length of the hash output). If the underlying hash function H is cryptographic and K_1 and K_2 are unrelated[8], then NMAC is provably secure.

HMAC is a more specific version of NMAC that only requires one key instead of two unrelated keys:

$$\text{HMAC}(M, K) = H((K' \oplus opad)||H((K' \oplus ipad)||M))$$

The HMAC algorithm actually supports a variable-length key K. However, NMAC uses K_1 and K_2 that are the same length as the hash output n, so we first transform K to be length n. If K is shorter than n bits, we can pad K with zeros until it is n bits. If K is longer than

[8]The formal definition of "unrelated" is out of scope for these notes. See this paper to learn more.

n bits, we can hash K to make it n bits. The transformed n-bit version of K is now denoted as K'.

Next, we derive two unrelated keys from K'. It turns out that XORing K' with two different pads is sufficient to satisfy the definition of "unrelated" used in the NMAC security proof. The HMAC algorithm uses two hardcoded pads, *opad* (outer pad) and *ipad* (inner pad), to generate two unrelated keys from a single key. The first key is $K_1 = K' \oplus opad$, and the second key is $K_2 = K' \oplus ipad$. *opad* is the byte `0x5c` repeated until it reaches n bits. Similarly, *ipad* is the byte `0x36` repeated until it reaches n bits.[9]

In words, HMAC takes the key and pads it or hashes it to length n. Then, HMAC takes the resulting modified key, XORs it with the ipad, concatenates the message, and hashes the resulting string. Next, HMAC takes the modified key, XORs it with the opad, and then concatenates it to the previous hash. Hash this final string to get the result.

Because NMAC is provably secure, and HMAC is a special case of NMAC that generates the two unrelated keys from one key, HMAC is also provably secure. This proof assumes that the underlying hash is a secure cryptographic hash, which means if you can find a way to break HMAC (forge a valid HMAC without knowing the key), then you have also broken the underlying cryptographic hash.

Because of the properties of a cryptographic hash, if you change just a single bit in either the message or the key, the output will be a completely different, unpredictable value. Someone who doesn't know the key won't be able to generate tags for arbitrary messages. In fact, they can't even distinguish the tag for a message from a random value of the same length.

HMAC is also very efficient. The inner hash function call only needs to hash the bits of the message, plus n bits, and the outer hash function call only needs to hash $2n$ bits.

8.5 MACs are not confidential

A MAC does not guarantee confidentiality on the message M to which it is applied. In the examples above, Alice and Bob have been exchanging non-encrypted plaintext messages with MACs attached to each message. The MACs provide integrity and authenticity, but they do nothing to hide the contents of the actual message. In general, MACs have no confidentiality guarantees–given $F(K, M)$, there is no guarantee that the attacker cannot learn something about M.

As an example, we can construct a valid MAC that guarantees integrity but does not guarantee confidentiality. Consider the MAC function F' defined as $F'(K, M) = F(K, M)||M$. In words, F' contains a valid MAC of the message, concatenated with the message plaintext. Assuming F is a valid MAC, then F' is also valid MAC. An attacker who doesn't know K won't be able to generate $F'(K, M')$ for the attacker's message M', because they won't be able to generate $F(K, M')$, which is part of $F'(K, M')$. However, F' does not provide any confidentiality on the message–in fact, it leaks the entire message!

[9] The security proof for HMAC just required that ipad and opad be different by at least one bit but, showing the paranoia of cryptography engineers, the designers of HMAC chose to make them very different.

There is no notion of "reversing" or "decrypting" a MAC, because both Alice and Bob use the same algorithm to generate MACs. However, there is nothing that says a MAC algorithm can't be reversed if you know the key. For example, with AES-MAC it is clear that if the message is a single block, you can run the algorithm in reverse to go from the tag to the message. Depending on the particular MAC algorithm, this notion of reversing a MAC might also lead to leakage of the original message.

There are some MAC algorithms that don't leak information about the message because of the nature of the underlying implementation. For example, if the algorithm directly applies a block cipher, the block cipher has the property that it does not leak information about the plaintext. Similarly, HMAC does not leak information about the message, since it maintains the properties of the cryptographic hash function.

In practice, we usually want to guarantee confidentiality in addition to integrity and authenticity. Next we will see how we can combine encryption schemes with MACs to achieve this.

8.6 Authenticated Encryption

An *authenticated encryption* scheme is a scheme that simultaneously guarantees confidentiality and integrity on a message. As you might expect, symmetric-key authenticated encryption modes usually combine a block cipher mode (to guarantee confidentiality) and a MAC (to guarantee integrity and authenticity).

Suppose we have an IND-CPA secure encryption scheme Enc that guarantees confidentiality, and an unforgeable MAC scheme MAC that guarantees integrity and authenticity. There are two main approaches to authenticated encryption: encrypt-then-MAC and MAC-then-encrypt.

In the *encrypt-then-MAC* approach, we first encrypt the plaintext, and then produce a MAC over the ciphertext. In other words, we send the two values $\langle \mathsf{Enc}_{K_1}(M), \mathsf{MAC}_{K_2}(\mathsf{Enc}_{K_1}(M)) \rangle$. This approach guarantees *ciphertext integrity*–an attacker who tampers with the ciphertext will be detected by the MAC on the ciphertext. This means that we can detect that the attacker has tampered with the message without decrypting the modified ciphertext. Additionally, the original message is kept confidential since neither value leaks information about the plaintext. The MAC value might leak information about the ciphertext, but that's fine; we already know that the ciphertext doesn't leak anything about the plaintext.

In the *MAC-then-encrypt* approach, we first MAC the message, and then encrypt the message and the MAC together. In other words, we send the value $\mathsf{Enc}_{K_1}(M || \mathsf{MAC}_{K_2}(M))$. Although both the message and the MAC are kept confidential, this approach does not have ciphertext integrity, since only the original message was tagged. This means that we'll only detect if the message is tampered after we decrypt it. This may not be desirable in some applications, because you would be running the decryption algorithm on arbitrary attacker inputs.

Although both approaches are theoretically secure if applied correctly, in practice, the MAC-then-Encrypt approach has been attacked through side channel vectors. In a side channel attack, improper usage of a cryptographic scheme causes some information to leak through

some other means besides the algorithm itself, such as the amount of computation time taken or the error messages returned. One example of this attack was a padding oracle attack against a particular TLS implementation using the MAC-then-encrypt approach. Because of the possibility of such attacks, encrypt-then-MAC is generally the better approach.

In both approaches, the encryption and MAC functions should use different keys, because using the same key in an authenticated encryption scheme makes the scheme vulnerable to a large category of potential attacks. These attacks take advantage of the fact that two different algorithms are called with the same key, as well as the properties of the particular encryption and MAC algorithms, to potentially leak information about the original message. The easiest way to avoid this category of attacks is to simply use different keys for the encryption and MAC functions.

8.7 AEAD Encryption Modes

There are also some special block cipher operation modes, known as AEAD (Authenticated Encryption with Additional Data) that, in addition to providing confidentiality like other appropriate block cipher modes, also provide integrity/authenticity.

The "additional data" component means that the integrity is provided not just over the encrypted portions of the message but some additional unencrypted data. For example, if Alice wants to send a message to Bob, she may want to include that the message is "From Alice to Bob" in plaintext (for the benefit of the system that routes the message from Alice to Bob) but also include it in the set of data protected by the authentication.

While powerful, using these modes improperly will lead to catastrophic failure in security, since a mistake will lead to a loss of both confidentiality and integrity at the same time.

One such mode is called AES-GCM (Galois Counter Mode). The specifics are out of scope for these notes, but at a high level, AES-GCM is a stream cipher that operates similarly to AES-CTR (counter) mode. The security properties of AES-GCM are also similar to CTR–in particular, IV reuse also destroys the security of AES-GCM. Since the built-in MAC in AES-GCM is also a function of the CTR mode encryption, improper use of AES-GCM causes loss of both confidentiality and integrity.

Some other modes include CCM mode, CWC mode, and OCB mode, but these are out of scope for these notes.

9 Pseudorandom Number Generators

9.1 Randomness and entropy

As we've seen in the previous sections, cryptography often requires randomness. For example, symmetric keys are usually randomly-generated bits, and random IVs and nonces are required to build secure block cipher chaining modes.

In cryptography, when we say "random," we usually mean "random and unpredictable." For example, flipping a biased coin that comes up heads 99% of the time is random, but you can predict a pattern–for a given coin toss, if you guess heads, it's very likely you're correct. A better source of randomness for cryptographic purposes would be flipping a fair coin, because the outcome is harder to predict than the outcome of the biased coin flip. Consider generating a random symmetric key: you would want to use outcomes of the fair coin to generate the key, because that makes it harder for the attacker to guess your key than if you had used outcomes of the biased coin to generate the key.

We can formalize this concept of unpredictability by defining *entropy*, a measure of uncertainty or surprise, for any random event. The biased coin has low entropy because you expect a given outcome most of the time. The fair coin has high entropy because you are very uncertain about the outcome. The specifics of entropy are beyond the scope of this class, but an important note is that the uniform distribution (all outcomes equally likely) produces the greatest entropy. In cryptography, we generally want randomness with the most entropy, so ideally, any randomness should be bits drawn from a uniform distribution (i.e. the outcomes of fair coin tosses).

However, true, unbiased randomness is computationally expensive to generate. True randomness usually requires sampling data from an unpredictable physical process, such as an unpredictable circuit on a CPU, random noise signals, or the microsecond at which a user presses a key. These sources may be biased and predictable, making it even more challenging to generate unbiased randomness.

Instead of using expensive true randomness each time a cryptographic algorithm requires randomness, we instead use *pseudo-randomness*. Pseudorandom numbers are generated deterministically using an algorithm, but they look random. In particular, a good pseudorandom number algorithm generates bits that are *computationally indistinguishable* from true random bits–there is no efficient algorithm that would let an attacker distinguish between pseudorandom bits and truly random bits.

9.2 Pseudorandom Number Generators (pRNGs)

A *pseudorandom number generator (pRNG)* is an algorithm that takes a small amount of truly random bits as input and outputs a long sequence of pseudorandom bits. The initial truly random input is called the *seed*.

The pRNG algorithm is deterministic, so anyone who runs the pRNG with the same seed will see the same pseudorandom output. However, to an attacker who doesn't know the

seed, the output of a secure pRNG is computationally indistinguishable from true random bits. A pRNG is not completely indistinguishable from true random bits–given infinite computational time and power, an attacker can distinguish pRNG output from truly random output. If the pRNG takes in an n-bit seed as input, the attacker just has to input all 2^n possible seeds and see if any of the 2^n outputs matches the bitstring they received. However, when restricted to any practical computation limit, an attacker has no way of distinguishing pRNG output from truly random output.

It would be very inefficient if a pRNG only outputted a fixed number of pseudorandom bits for each truly random input. If this were the case, we would have to generate more true randomness each time the pRNG output has all been used. Ideally, we would like the pRNG to take in an initial seed and then be available to generate as many pseudorandom bits as needed on demand. To achieve this, the pRNG maintains some internal state and updates the state any time the pRNG generates new bits or receives a seed as input.

Formally, a pRNG is defined by the following three functions:

- *Seed(entropy)*: Take in some initial truly random entropy and initialize the pRNG's internal state.
- *Reseed(entropy)*: Take in some additional truly random entropy, updating the pRNG's internal state as needed.
- *Generate(n)*: Generate n pseudorandom bits, updating the internal state as needed. Some pRNGs also support adding additional entropy directly during this step.

9.3 Rollback resistance

In the previous section, we defined a secure pRNG as an algorithm whose output is computationally indistinguishable from random if the attacker does not know the seed and internal state. However, this definition does not say anything about the consequences of an attacker who does manage to compromise the internal state of a secure pRNG.

An additional desirable property of a secure pRNG is *rollback resistance*. Suppose a pRNG has been used to generate 100 bits, and an attacker is somehow able to learn the internal state immediately after bit 100 has been generated. If the pRNG is rollback-resistant, then the attacker cannot deduce anything about any previously-generated bit. Formally, the previously-generated output of the pRNG should still be computationally indistinguishable from random, even if the attacker knows the current internal state of the pRNG.

Not all secure pRNGs are rollback-resistant, but rollback resistance is an important property for any practical cryptographic pRNG implementation. Consider a cryptosystem that uses a single pRNG to generate both the secret keys and the IVs (or nonces) for a symmetric encryption scheme. The pRNG is first used to generate the secret keys, and then used again to generate the IVs. If this pRNG was not rollback-resistant, then an attacker who compromises the internal state at this point could learn the value of the secret key.

9.4 HMAC-DRBG

There are many implementations of pRNGs, but one commonly-used pRNG in practice is HMAC-DRBG[10], which uses the security properties of HMAC to build a pRNG.

HMAC-DRBG maintains two values as part of its internal state, K and V. K is used as the secret key to the HMAC, and V is used as the "message" input to the HMAC.

To generate a block of pseudorandom bits, HMAC-DRBG computes HMAC on the previous block of pRNG output. This can be repeated to generate as many pseudorandom bits as needed. Recall that the output of HMAC looks random to an attacker who doesn't know the key. As long as we keep the internal state (which includes K) secret, an attacker cannot distinguish the output of the HMAC from random bits, so the pRNG is secure.

We also use HMAC to update the internal state K and V each time. If additional true randomness is provided, we add it to the "message" input to HMAC.

Algorithm 1 GENERATE(n): Generate n pseudorandom bits, with no additional true random input.

```
1: output = ''
2: while len(output) < n do
3:     V = HMAC(K, V)
4:     output = output || V
5: end while
6: K = HMAC(K, V || 0x00)
7: V = HMAC(K, V)
8: return output[0:n]
```

At line 3, we are repeatedly calling HMAC on the previous block of output. The while loop repeats this process until we have at least n bits of output. Once we have enough output, we update the internal state with two additional HMAC calls, and then return the first n bits of pseudorandom output.

Next, let's see how to seed the pRNG. The seed and reseed algorithms use true randomness as input to the HMAC, and uses the output of slight variations on the HMAC input to update K and V.

[10] DRBG stands for Deterministic Random Bit Generator

Algorithm 2 SEED(s): Take some truly random bits s and initialize the internal state.

1: K = 0
2: V = 0
3: K = HMAC(K, V || 0x00 || s)
4: V = HMAC(K, V)
5: K = HMAC(K, V || 0x01 || s)
6: V = HMAC(K, V)

The reseed algorithm is identical to the seed algorithm, except we don't need to reset K and V to 0 (steps 1-2).

Finally, if we want to generate pseudorandom output and add entropy at the same time, we combine the two algorithms above:

Algorithm 3 GENERATE(n, s): Generate n pseudorandom bits, with additional true random input s.

1: output = ''
2: **while** len(output) < n **do**
3: V = HMAC(K, V)
4: output = output || V
5: **end while**
6: K = HMAC(K, V || 0x00 || s)
7: V = HMAC(K, V)
8: K = HMAC(K, V || 0x01 || s)
9: V = HMAC(K, V)
10: **return** output[0:n]

The specific design decisions of HMAC-DRBG, such as why it uses 0x00 and 0x01, are not so important. The main takeaway is that because HMAC output is indistinguishable from random, the output of HMAC-DRBG (which is essentially lots of HMAC outputs) is also indistinguishable from random.

The use of the cryptographic hash function in both the seeding and reseeding algorithms means that HMAC-DRBG can accept an arbitrary long initial seed. For example, if each bit of the input seed really only has 0.1 bits of entropy (e.g. because it is a highly biased coin), using 2560 bits of seed material will leave HMAC-DRBG with 256b of actual entropy for its internal operations. Furthermore, adding in additional strings that contain *no* entropy (such as a string of 0 bits or the number π) doesn't make the internal state worse.

Additionally, HMAC-DRBG has rollback resistance: if you can compute the previous state from the current state you have successfully reversed the underlying hash function!

9.5 Stream ciphers

As we've seen in the previous section, an attacker without knowledge of the internal state of a secure, rollback-resistant pRNG cannot predict the pRNG's past or future output, and the attacker cannot distinguish the pRNG output from random bits. This sounds very similar to the properties we want in a random, unpredictable one-time pad. In fact, we can use pRNGs to generate a one-time pad that we then use for encrypting messages. This encryption scheme is an example of a class of algorithms known as *stream ciphers*.

Recall that in block ciphers, we encrypted and decrypted messages by splitting them into fixed-size blocks. Stream ciphers use a different approach to encryption, in which we encrypt and decrypt messages as they arrive, one bit at a time. You can imagine a stream cipher operating on an encrypted file being downloaded from the Internet: as each subsequent bit is downloaded, the stream cipher can immediately decrypt the bit while waiting for the next bit to download. This is different from a block cipher, where you might need a block of bits, several blocks of bits, or the entire message to be downloaded before you can start decrypting.

A common class of stream cipher algorithms involves outputting an unpredictable stream of bits, and then using this stream as the key of a one-time pad. In other words, each bit of the plaintext message is XORed with the corresponding bit in the key stream.

The output of a secure pRNG can be used as the key for this one-time pad scheme. Formally, in a pRNG-based stream cipher, the secret key is the initial seed used to seed the pRNG. To encrypt an n-bit message, Alice runs the pRNG until it generates n pseudorandom bits. Then she XORs the pseudorandom bits with the plaintext message. Since the pRNG can generate as many bits as needed, this algorithm can encrypt arbitrary-length messages.

To decrypt the message, Bob uses the same secret key to seed the pRNG. Since the pRNG is deterministic with the same seed, Bob will generate the same pseudorandom bits when he runs the pRNG. Then Bob XORs the pseudoranom bits with the ciphertext to learn the original plaintext.

To avoid key reuse, Alice and Bob can both seed the pRNG with a random IV in addition to their secret key so that the pRNG output is different and unpredictable each time. In short, the pRNG algorithm is:

- Encryption: $Enc(K, M) = \langle IV, PRNG(K, IV) \oplus M \rangle$
- Decryption: $Dec(K, IV, C_2) = PRNG(K, IV) \oplus C_2$

AES-CTR is effectively a stream cipher. Although technically AES appears to be a pseudo-random permutation rather than a pseudo-random generator, in practice the results are similar. As long as the total ciphertext encrypted with a given key is kept to a reasonable level (2^{64}b), the one-time pad output of AES-CTR should be effectively indistinguishable from pRNG output. Beyond this threshold, there is a significant probability with CTR mode that there will be two blocks with identical ciphertext, which would leak information that the underlying plaintext blocks are different.

Although theoretically we could use any cryptographically secure pRNG (like HMAC-DRBG) as a stream cipher, dedicated stream ciphers (such as the ChaCha20 cipher) have properties that we would consider a disadvantage in a secure pRNG but are actually advantages for stream ciphers. In particular, both AES-CTR mode encryption and ChaCha20 include a counter value in the computation of the stream.

One desirable consequence of including a counter is the ability to encrypt or decrypt an arbitrary point in the message without starting from the beginning. If you have a 1 terabyte file encrypted using either AES-CTR mode or ChaCha20 and you wish to read just the last bytes, you can set the counter to the appropriate point and just decrypt the last bytes, while if you used HMAC-DRBG as the stream cipher, you would need to start at the beginning of the message and compute 1 terabytes of HMAC-DRBG output before you could read the end of the file.[11]

[11] This use of a counter value means that if you view it as a pRNG, AES-CTR and ChaCha20 lack rollback resistance as the key and counter are the internal state. But on the other hand, it is this ability to rollback and jump-forward into the output space that makes them more useful as stream ciphers.

10 Diffie-Hellman key exchange

In the previous sections, we discussed symmetric-key schemes such as block ciphers and MACs. For these schemes to work, we assumed that Alice and Bob both share a secret key that no one else knows. But how would they be able to exchange a secret key if they can only communicate through an insecure channel? It turns out there is a clever way to do it, first discovered by Whit Diffie and Martin Hellman in the 1970s.

The goal of Diffie-Hellman is usually to create an *ephemeral* key. An ephemeral key is used for some series of encryptions and decryptions and is discarded once it is no longer needed. Thus Diffie-Hellman is effectively a way for two parties to agree on a *random* value in the face of an eavesdropper.

10.1 Diffie-Hellman intuition

A useful analogy to gain some intuition about Diffie-Hellman key exchange is to think about colors. Alice and Bob want to both share a secret color of paint, but Eve can see any paint being exchanged between Alice and Bob. It doesn't matter what the secret color is, as long as only Alice and Bob know it.

Alice and Bob each start by deciding on a secret color–for example, Alice generates amber and Bob generates blue. If Eve wasn't present, Alice and Bob could just send their secret colors to each other and mix the two secret colors together to get the final color. However, since Eve can see any colors sent between Alice and Bob, they must somehow hide their colors when exchanging them.

To hide their secret colors, Alice and Bob agree on a publicly-known common paint color–in this example, green. They each take their secret color, mix it with the common paint color, and then send it over the insecure channel. Alice sends a green-amber mixture to Bob, and Bob sends a green-blue mixture to Alice. Here we're assuming that given a paint mixture, Eve cannot separate the mixture into its original colors.

At this point, Alice and Bob have each other's secret colors, mixed with the common color. All that's left is to add their own secret. Alice receives the green-blue mixture from Bob and adds her secret amber to get green-blue-amber. Bob receives the green-amber mixture from Alice and adds his secret blue to get green-amber-blue. Alice and Bob now both have a shared secret color (green-amber-blue).

The only colors exchanged over the insecure channel were the secret mixtures green-amber and green-blue. Given green-amber, Eve would need to add blue to get the secret, but she doesn't know Bob's secret color, because it wasn't sent over the channel, and she can't separate the green-blue mixture. Eve could also try mixing the green-amber and green-blue mixtures, but you can imagine this wouldn't result in exactly the same secret, since there would be too much green in the mix. The result would be more like green-amber-green-blue than green-amber-blue.

10.2 Discrete logarithm problem

The secret exchange in the color analogy relied on the fact that mixing two colors is easy, but separating a mixture of two colors is practically impossible. It turns out that there is a mathematical equivalent of this. We call these *one-way functions*: a function f such that given x, it is easy to compute $f(x)$, but given y, it is practically impossible to find a value x such that $f(x) = y$.

A one-way function is also sometimes described as the computational equivalent of a process that turns a cow into hamburger: given the cow, you can produce hamburger, but there's no way to restore the original cow from the hamburger.

There are many functions believed to be one-way functions. The simplest one is exponentiation modulo a prime: $f(x) = g^x \pmod{p}$, where p is a large prime and g is a specially-chosen generator[12].

Given x, it is easy to calculate $f(x)$ (you may recall the repeated squaring algorithm from CS 70). However, given $f(x) = g^x \pmod{p}$, there is no known efficient algorithm to solve for x. This is known as the *discrete logarithm problem*, and it is believed to be computationally hard to solve.

Using the hardness of the discrete log problem and the analogy from above, we are now ready to construct the Diffie-Hellman key exchange protocol.

10.3 Diffie-Hellman protocol

In high-level terms, the Diffie-Hellman key exchange works like this.

Alice and Bob first establish the public parameters p and g. Remember that p is a large prime and g is a generator in the range $1 < g < p - 1$. For instance, Alice could pick p and g and then announce it publicly to Bob. Today, g and p are often hardcoded or defined in a standard so they don't need to be chosen each time. These values don't need to be specific to Alice or Bob in any way, and they're not secret.

Then, Alice picks a secret value a at random from the set $\{0, 1, \ldots, p-2\}$, and she computes $A = g^a \bmod p$. At the same time, Bob randomly picks a secret value b and computes $B = g^b \bmod p$.

Now Alice announces the value A (keeping a secret), and Bob announces B (keeping b secret). Alice uses her knowledge of B and a to compute

$$S = B^a = (g^b)^a = g^{ba} \pmod{p}.$$

Symmetrically, Bob uses his knowledge of A and b to compute

$$S = A^b = (g^a)^b = g^{ab} \pmod{p}.$$

[12] You don't need to worry about how to choose g, just know that it satisfies some special number theory properties. In short, g must satisfy the following properties: $1 < g < p-1$, and there exists a k where $g^k = a$ for all $1 \leq a \leq p - 1$.

Note that $g^{ba} = g^{ab} \pmod{p}$, so both Alice and Bob end up with the same result, S.

Finally, Alice and Bob can use S as a shared key for a symmetric-key cryptosystem (in practice, we would apply some hash function to S first and use the result as our shared key, for technical reasons).

The amazing thing is that Alice and Bob's conversation is entirely public, and from this public conversation, they both learn this secret value S—yet eavesdroppers who hear their entire conversation cannot learn S.

As far as we know, there is no efficient algorithm to deduce $S = g^{ab} \bmod p$ from the values Eve sees, namely $A = g^a \bmod p$, $B = g^b \bmod p$, g, and p. The hardness of this problem is closely related to the discrete log problem discussed above. In particular, the fastest known algorithms for solving this problem take $2^{cn^{1/3}(\log n)^{2/3}}$ time, if p is a n-bit prime. For $n = 2048$, these algorithms are far too slow to allow reasonable attacks.

Here is how this applies to secure communication among computers. In a computer network, each participant could pick a secret value x, compute $X = g^x \bmod p$, and publish X for all time. Then any pair of participants who want to hold a conversation could look up each other's public value and use the Diffie-Hellman scheme to agree on a secret key known only to those two parties. This means that the work of picking p, g, x, and X can be done in advance, and each time a new pair of parties want to communicate, they each perform only one modular exponentiation. Thus, this can be an efficient way to set up shared keys.

Here is a summary of Diffie-Hellman key exchange:

- **System parameters:** a 2048-bit prime p, a value g in the range $2 \ldots p-2$. Both are arbitrary, fixed, and public.
- **Key agreement protocol:** Alice randomly picks a in the range $0 \ldots p-2$ and sends $A = g^a \bmod p$ to Bob. Bob randomly picks b in the range $0 \ldots p-2$ and sends $B = g^b \bmod p$ to Alice. Alice computes $K = B^a \bmod p$. Bob computes $K = A^b \bmod p$. Alice and Bob both end up with the same random secret key K, yet as far as we know no eavesdropper can recover K in any reasonable amount of time.

10.4 Elliptic-curve Diffie-Hellman

In the section above, we used the discrete log problem to construct a Diffie-Hellman protocol, but we can generalize Diffie-Hellman key exchange to other one-way functions. One commonly used variant of Diffie-Hellman relies on the elliptic curve discrete logarithm problem, which is based on the math around elliptic curves instead of modular arithmetic. Although the underlying number theory is more complicated, elliptic-curve Diffie-Hellman can use smaller keys than modular arithmetic Diffie-Hellman and still provide the same security, so it has many useful applications.

For this class, you don't need to understand the underlying math that makes elliptic-curve Diffie-Hellman work, but you should know that the idea behind the protocol is the same.

Alice and Bob start with a publicly known point on the elliptic curve G. Alice chooses a secret integer a and Bob chooses a secret integer b.

Alice computes $A = a \cdot G$ (this is a point on the curve A, obtained by adding the point G to itself a times), and Bob computes $B = b \cdot G$. Alice sends A to Bob, and Bob sends B to Alice.

Alice computes

$$S = a \cdot B = a \cdot b \cdot G$$

and Bob computes

$$S = b \cdot A = b \cdot a \cdot G$$

Because of the properties of the elliptic curve, Alice and Bob will derive the same point S, so they now have a shared secret. Also, the elliptic-curve Diffie-Hellman problem states that given $A = a \cdot G$ and $B = b \cdot G$, there is no known efficient method for Eve to calculate S.

10.5 Difficulty in Bits

It is generally believed that the discrete log problem is hard, but how hard? In practice, it is generally believed that computing the discrete log modulo a 2048b prime, computing the elliptic curve discrete log on a 256b curve, and brute forcing a 128b symmetric key algorithm are all roughly the same difficulty. (Brute-forcing the 128b key is believed to be slightly harder than the other two.)

Thus, if we are using Diffie-Hellman key exchange with other cryptoschemes, we try to relate the difficulty of the schemes so that it is equally difficult for an attacker to break any scheme. For example, 128b AES tends to be used with SHA-256 and either 256b elliptic curves or 2048b primes for Diffie-Hellman. Similarly, for top-secret use, the NSA uses 256b AES, 384b Elliptic Curves, SHA-384, and 3096b Diffie-Hellman and RSA.

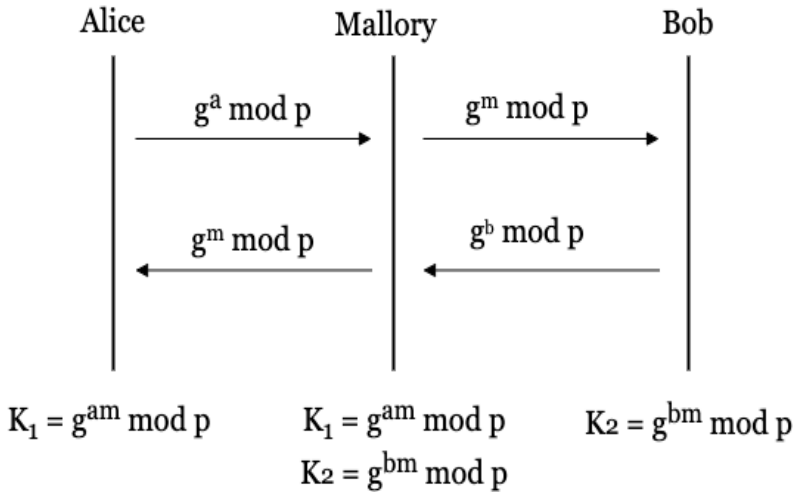
10.6 Attacks on Diffie-Hellman

As we've seen, Diffie-Hellman is secure against an eavesdropper Eve, who observes the messages sent between Alice and Bob, but does not tamper with them. What if we replace Eve with Mallory, an active adversary (man-in-the-middle) who can tamper with messages?

It turns out the Diffie-Hellman key exchange protocol is only secure against a passive adversary and not an active adversary. If Mallory can tamper with the communication between Alice and Bob, she can fool them into thinking that they've agreed with a shared key, when they have actually generated two different keys that Mallory knows.

The following figure demonstrates how an active attacker (Mallory) can agree on a key ($K_1 = g^{am} \pmod p$) with Alice and another key ($K_2 = g^{bm} \pmod p$) with Bob in order to man-in-the-middle (MITM) their communications.

When Alice sends $g^a \pmod p$ to Bob, Mallory intercepts the message and replaces it with $g^m \pmod p$, where m is Mallory's secret. Bob now receives $g^m \pmod p$ instead of $g^a \pmod p$.



Now, when Bob wants to calculate his shared key, he will calculate $K = A^b \pmod p$, where A is the value he received from Alice. Since he received a tampered value from Mallory, Bob will actually calculate $K = (g^m)^b = g^{mb} \pmod p$.

Likewise, when Bob sends $g^b \pmod p$ to Alice, Mallory intercepts the message and replaces it with $g^m \pmod p$. Alice receives $g^m \pmod p$. To calculate her shared key, she calculates $K = B^a \pmod p$, where B is the value she received from Bob. Since Alice received a tampered value, she will actually calculate $K = (g^m)^a = g^{ma} \pmod p$.

After the exchange, Alice thinks the shared key is $g^{ma} \pmod p$ and Bob thinks the shared key is $g^{mb} \pmod p$. They no longer have the same shared secret.

Even worse, Mallory knows both of these values too. Mallory intercepted Alice sending $g^a \pmod p$, which means Mallory knows the value of $g^a \pmod p$. She also knows her own chosen secret m. Thus she can calculate $(g^a)^m = g^{am} \pmod p$, which is what Alice thinks her shared secret is. Likewise, Mallory intercepted $g^b \pmod p$ from Bob and can calculate $(g^b)^m = g^{bm} \pmod p$, which is what Bob thinks his shared secret is.

If Alice and Bob fall victim to this attack, Mallory can now decrypt any messages sent from Alice with Alice's key $g^{ma} \pmod p$, make any changes to the message, re-encrypt the message with Bob's key $g^{mb} \pmod p$, and send it to Bob. In other words, Mallory would pretend to Alice that she is Bob, and pretend to Bob that she is Alice. This would not only allow Mallory to eavesdrop on the entire conversation but also make changes to the messages without Alice and Bob ever noticing that they are under attack.

The main reason why the Diffie-Hellman protocol is vulnerable to this attack is that the messages exchanged between Alice and Bob have no integrity or authenticity. To defend against this attack, Alice and Bob will need to additionally use a cryptoscheme that provides integrity and authenticity, such as digital signatures. If the messages sent during the Diffie-Hellman exchange have integrity and authenticity, then Alice and Bob would be able to detect Mallory's tampering with the messages.

11 Public-Key (Asymmetric) Encryption

11.1 Overview

Previously we saw symmetric-key encryption, where Alice and Bob share a secret key K and use the same key to encrypt and decrypt messages. However, symmetric-key cryptography can be inconvenient to use, because it requires Alice and Bob to coordinate somehow and establish the shared secret key. *Asymmetric cryptography*, also known as *public-key cryptography*, is designed to address this problem.

In a public-key cryptosystem, the recipient Bob has a publicly available key, his *public key*, that everyone can access. When Alice wishes to send him a message, she uses his public key to encrypt her message. Bob also has a secret key, his *private key*, that lets him decrypt these messages. Bob publishes his public key but does not tell anyone his private key (not even Alice).

Public-key cryptography provides a nice way to help with the key management problem. Alice can pick a secret key K for some symmetric-key cryptosystem, then encrypt K under Bob's public key and send Bob the resulting ciphertext. Bob can decrypt using his private key and recover K. Then Alice and Bob can communicate using a symmetric-key cryptosystem, with K as their shared key, from there on.

11.2 Trapdoor One-way Functions

Public-key cryptography relies on a close variant of the one-way function. Recall from the previous section that a one-way function is a function f such that given x, it is easy to compute $f(x)$, but given y, it is hard to find a value x such that $f(x) = y$.

A *trapdoor one-way function* is a function f that is one-way, but also has a special backdoor that enables someone who knows the backdoor to invert the function. As before, given x, it is easy to compute $f(x)$, but given only y, it is hard to find a value x such that $f(x) = y$. However, given both y and the special backdoor K, it is now easy to compute x such that $f(x) = y$.

A trapdoor one-way function can be used to construct a public encryption scheme as follows. Bob has a public key PK and a secret key SK. He distributes PK to everyone, but does not share SK with anyone. We will use the trapdoor one-way function $f(x)$ as the encryption function.

Given the public key PK and a plaintext message x, it is computationally easy to compute the encryption of the message: $y = f(x)$.

Given a ciphertext y and only the public key PK, it is hard to find the plaintext message x where $f(x) = y$. However, given ciphertext y and the secret key SK, it becomes computationally easy to find the plaintext message x such that $y = f(x)$, i.e., it is easy to compute $f^{-1}(y)$.

We can view the private key as "unlocking" the trapdoor. Given the private key SK, it

becomes easy to compute the decryption f^{-1}, and it remains easy to compute the encryption f.

Here are two examples of trapdoor functions that will help us build public encryption schemes:

- *RSA Hardness*: Suppose $n = pq$, i.e. n is the product of two large primes p and q. Given $c = m^e \pmod{n}$ and e, it is computationally hard to find m. However, with the factorization of n (i.e. p or q), it becomes easy to find m.
- *Discrete log problem*: Suppose p is a large prime and g is a generator. Given g, p, $A = g^a \pmod{p}$, and $B = g^b \pmod{p}$, it is computationally hard to find $g^{ab} \pmod{p}$. However, with a or b, it becomes easy to find $g^{ab} \pmod{p}$.

11.3 RSA Encryption

Under construction

For now, you can refer to these notes from CS 70 for a detailed proof of RSA encryption. For this class, you won't need to remember the proof of why RSA works. All you need to remember is that we use the public key to encrypt messages, we use the corresponding private key to decrypt messages, and an attacker cannot break RSA encryption unless they can factor large primes, which is believed to be hard.

There is a tricky flaw in the RSA scheme described in the CS 70 notes. The scheme is deterministic, so it is not IND-CPA secure. Sending the same message multiple times causes information leakage, because an adversary can see when the same message is sent. This basic variant of RSA might work for encrypting "random" messages, but it is not IND-CPA secure. As a result, we have to add some randomness to make the RSA scheme resistant to information leakage.

RSA introduces randomness into the scheme through a *padding mode*. Despite the name, RSA padding modes are more similar to the IVs in block cipher modes than the padding in block cipher modes. Unlike block cipher padding, public-key padding is not a deterministic algorithm for extending a message. Instead, public-key padding is a tool for mixing in some randomness so that the ciphertext output "looks random," but can still be decrypted to retrieve the original plaintext.

One common padding scheme is OAEP (Optimal Asymmetric Encryption Padding). This scheme effectively generates a random symmetric key, uses the random key to scramble the message, and encrypts both the scrambled message and the random key. To recover the original message, the attacker has to recover both the scrambled message and the random key in order to reverse the scrambling process.

11.4 El Gamal encryption

The Diffie-Hellman protocol doesn't quite deliver public-key encryption directly. It allows Alice and Bob to agree on a shared secret that they could use as a symmetric key, but

it doesn't let Alice and Bob control what the shared secret is. For example, in the Diffie-Hellman protocol we saw, where Alice and Bob each choose random secrets, the shared secret is also a random value. Diffie-Hellman on its own does not let Alice and Bob send encrypted messages to each other. However, there is a slight variation on Diffie-Hellman that would allow Alice and Bob to exchange encrypted messages.

In 1985, a cryptographer by the name of Taher Elgamal invented a public-key encryption algorithm based on Diffie-Hellman. We will present a simplified form of El Gamal encryption scheme. El Gamal encryption works as follows.

The public system parameters are a large prime p and a value g satisfying $1 < g < p-1$. Bob chooses a random value b (satisfying $0 \le b \le p-2$) and computes $B = g^b \bmod p$. Bob's public key is B, and his private key is b. Bob publishes B to the world, and keeps b secret.

Now, suppose Alice has a message m (in the range $1 \ldots p-1$) she wants to send to Bob, and suppose Alice knows that Bob's public key is B. To encrypt the message m to Bob, Alice picks a random value r (in the range $0 \ldots p-2$), and forms the ciphertext

$$(g^r \bmod p, m \times B^r \bmod p).$$

Note that the ciphertext is a pair of numbers, each number in the range $0 \ldots p-1$.

How does Bob decrypt? Well, let's say that Bob receives a ciphertext of the form (R, S). To decrypt it, Bob computes

$$R^{-b} \times S \bmod p,$$

and the result is the message m Alice sent him.

Why does this decryption procedure work? If $R = g^r \bmod p$ and $S = m \times B^r \bmod p$ (as should be the case if Alice encrypted the message m properly), then

$$R^{-b} \times S = (g^r)^{-b} \times (m \times B^r) = g^{-rb} \times m \times g^{br} = m \pmod p.$$

If you squint your eyes just right, you might notice that El Gamal encryption is basically Diffie-Hellman, tweaked slightly. It's a Diffie-Hellman key exchange, where Bob uses his long-term public key B and where Alice uses a fresh new public key $R = g^r \bmod p$ chosen anew just for this exchange. They derive a shared key $K = g^{rb} = B^r = R^b \pmod p$. Then, Alice encrypts her message m by multiplying it by the shared key K modulo p.

That last step is in effect a funny kind of one-time pad, where we use multiplication modulo p instead of xor: here K is the key material for the one-time pad, and m is the message, and the ciphertext is $S = m \times K = m \times B^r \pmod p$. Since Alice chooses a new value r independently for each message she encrypts, we can see that the key material is indeed used only once. And a one-time pad using modular multiplication is just as secure as xor, for essentially the same reason that a one-time pad with xor is secure: given any ciphertext S and a hypothesized message m, there is exactly one key K that is consistent with this hypothesis (i.e., exactly one value of K satisfying $S = m \times K \bmod p$).

Another way you can view El Gamal is using the discrete log trapdoor one-way function defined above: Alice encrypts the message with $B^r = g^{br} \pmod p$. Given only g, p, $R = g^r$

(mod p), and $B = g^b \pmod{p}$, it is hard for an attacker to learn $g^{-br} \pmod{p}$ and decrypt the message. However, with Bob's secret key b, Bob can easily calculate $g^{-br} \pmod{p}$ and decrypt the message.

Note that for technical reasons that we won't go into, this simplified El Gamal scheme is actually *not* semantically secure. With some tweaks, the scheme can be made semantically secure. Interested readers can read more at this link.

Here is a summary of El Gamal encryption:

- **System parameters:** a 2048-bit prime p, and a value g in the range $2 \ldots p-2$. Both are arbitrary, fixed, and public.
- **Key generation:** Bob picks b in the range $0 \ldots p-2$ randomly, and computes $B = g^b \bmod p$. His public key is B and his private key is b.
- **Encryption:** $E_B(m) = (g^r \bmod p, m \times B^r \bmod p)$ where r is chosen randomly from $0 \ldots p-2$.
- **Decryption:** $D_b(R, S) = R^{-b} \times S \bmod p$.

11.5 Public Key Distribution

This all sounds great—almost too good to be true. We have a way for a pair of strangers who have never met each other in person to communicate securely with each other. Unfortunately, it is indeed too good to be true. There is a slight catch. The catch is that if Alice and Bob want to communicate securely using these public-key methods, they need some way to securely learn each others' public key. The algorithms presented here don't help Alice figure out what is Bob's public key; she's on her own for that.

You might think all Bob needs to do is broadcast his public key, for Alice's benefit. However, that's not secure against *active attacks*. Attila the attacker could broadcast his own public key, pretending to be Bob: he could send a spoofed broadcast message that appears to be from Bob, but that contains a public key that Attila generated. If Alice trustingly uses that public key to encrypt messages to Bob, then Attila will be able to intercept Alice's encrypted messages and decrypt them using the private key Attila chose.

What this illustrates is that Alice needs a way to obtain Bob's public key through some channel that she is confident cannot be tampered with. That channel does not need to protect the *confidentiality* of Bob's public key, but it does need to ensure the *integrity* of Bob's public key. It's a bit tricky to achieve this.

One possibility is for Alice and Bob to meet in person, in advance, and exchange public keys. Some computer security conferences have "key-signing parties" where like-minded security folks do just that. In a similar vein, some cryptographers print their public key on their business cards. However, this still requires Alice and Bob to meet in person in advance. Can we do any better? We'll soon see some methods that help somewhat with that problem.

11.6 Session Keys

There is a problem with public key: it is *slow*. It is very, very slow. When encrypting a single message with a 2048b RSA key, the RSA algorithm requires exponentiation of a 2048b number to a 2048b power, modulo a 2048b number. Additionally, some public key schemes only really work to encrypt "random" messages. For example, RSA without OAEP leaks when the same message is sent twice, so it is only secure if every message sent consists of random bits. In the simplified El Gamal scheme shown in these notes, it is easy for an attacker to substitute the message $M' = 2M$. If the messages have meaning, this can be a problem.

Because public key schemes are expensive and difficult to make IND-CPA secure, we tend to only use public key cryptography to distribute one or more *session keys*. Session keys are the keys used to actually encrypt and authenticate the message. To send a message, Alice first generates a random set of session keys. Often, we generate several different session keys for different purposes. For example, we may generate one key for encryption algorithms and another key for MAC algorithms. We may also generate one key to encrypt messages from Alice to Bob, and another key to encrypt messages from Bob to Alice. (If we need different keys for each message direction and different keys for encryption and MAC, we would need a total of four symmetric keys.) Alice then encrypts the message using a symmetric algorithm with the session keys (such as AES-128-CBC-HMAC-SHA-256 [13]) and encrypts the random session keys with Bob's public key. When he receives the ciphertext, Bob first decrypts the session keys and then uses the session keys to decrypt the original message.

[13] That is, using AES with 128b keys in CBC mode and then using HMAC with SHA-256 for integrity

12 Digital Signatures

We can use the ideas from public-key encryption to build asymmetric cryptographic schemes that guarantee integrity and authentication too. In this section, we will define *digital signatures*, which are essentially the public-key version of MACs, and show how they can help guarantee integrity and authentication.

12.1 Digital signature properties

Recall that in public-key encryption, anyone could use Bob's public key to encrypt a message and send it to him, but only Bob could use his secret key to decrypt the message. However, the situation is different for digital signatures. It would not really make sense if everyone could generate a signature on a message and only Bob could verify it. If anyone could generate a signature with a public key, what's stopping an attacker from generating a malicious message with a valid signature?

Instead, we want the reverse: only Bob can generate a signature on a message, and everyone else can verify the signature to confirm that the message came from Bob and has not been tampered with.

In a digital signature scheme, Bob generates a public key (also known as a *verification* key) and a private key (also known as a *signing* key). Bob distributes his public verification key to everyone, but keeps his signing key secret. When Bob wants to send a message, he uses his secret signing key to generate a signature on the message. When Alice receives the message, she can use Bob's public verification key to check that the signature is valid and confirm that the message is untampered and actually from Bob.

Mathematically, a digital signature scheme consists of three algorithms:

- **Key generation:** There is a randomized algorithm KeyGen that outputs a matching public key and private key: $(PK, SK) = \text{KeyGen}()$. Each invocation of KeyGen produces a new keypair.
- **Signing:** There is a signing algorithm Sign: $S = \text{Sign}(SK, M)$ is the signature on the message M (with private key SK).
- **Verification:** There is a verification algorithm Verify, where $\text{Verify}(PK, M, S)$ returns true if S is a valid signature on M (with public key PK) or false if not.

If PK, SK are a matching pair of private and public keys (i.e., they were output by some call to KeyGen), and if $S = \text{Sign}(SK, M)$, then $\text{Verify}(PK, M, S) = \text{true}$.

12.2 RSA Signatures: High-level Outline

At a high level, the RSA signature scheme works like this. It specifies a trapdoor one-way function F. The public key of the signature scheme is the public key U of the trapdoor function, and the private key of the signature scheme is the private key K of the trapdoor function. We also need a one-way function H, with no trapdoor; we typically let H be some

cryptographic hash function, per § 7. The function H is standardized and described in some public specification, so we can assume that everyone knows how to compute H, but no one knows how to invert it.

We define a signature on a message M as a value S that satisfies the following equation:

$$H(M) = F_U(S).$$

Note that given a message M, an alleged signature S, and a public key U, we can verify whether it satisfies the above equation. This makes it possible to verify the validity of signatures.

How does the signer sign messages? It turns out that the trapdoor to F, i.e., the private key K, lets us find solutions to the above equation. Given a message M and the private key K, the signer can first compute $y = H(M)$, then find a value S such that $F_U(S) = y$. In other words, the signer computes $S = F^{-1}(H(M))$; that's the signature on M. This is easy to do for someone who knows the private key K, because K lets us invert the function F, but it is hard to do for anyone who does not know K. Consequently, anyone who has the private key can sign messages.

For someone who does not know the private key K, there is no easy way to find a message M and a valid signature S on it. For instance, an attacker could pick a message M, compute $H(M)$, but then the attacker would be unable to compute $F^{-1}(H(M))$, because the attacker does not know the trapdoor for the one-way function F. Similarly, an attacker could pick a signature S and compute $y = F(S)$, but then the attacker would be unable to find a message M satisfying $H(M) = y$, since H is one-way.

This is the general idea underpinning the RSA signature scheme. Now let's look at how to build a trapdoor one-way function, which is the key idea needed to make this all work.

12.3 Number Theory Background

Here are some basic facts from number theory, which will be useful in deriving RSA signatures. As previously discussed in lecture, we use $\varphi(n)$ to denote Euler's *totient function* of n: the number of positive integers less than n that share no common factor with n.

Fact 1 *If* $\gcd(x, n) = 1$*, then* $x^{\varphi(n)} = 1 \pmod{n}$*.* ("Euler's theorem.")

Fact 2 *If* p *and* q *are two different odd primes, then* $\varphi(pq) = (p-1)(q-1)$*.*

Fact 3 *If* $p = 2 \pmod 3$ *and* $q = 2 \pmod 3$*, then there exists a number* d *satisfying* $3d = 1 \pmod{\varphi(pq)}$*, and this number* d *can be efficiently computed given* $\varphi(pq)$*.*

Let's assume that p and q are two different odd primes, that $p = 2 \pmod 3$ and $q = 2 \pmod 3$, and that $n = pq$.[14] Let d be the positive integer promised to exist by Fact 3. As a consequence of Facts 2 and 3, we can efficiently compute d given knowledge of p and q.

[14] Why do we pick those particular conditions on p and q? Because then $\varphi(pq) = (p-1)(q-1)$ will not be a multiple of 3, which is going to allow us to have unique cube roots.

Theorem 1 *With notation as above, define functions F, G by $F(x) = x^3 \bmod n$ and $G(x) = x^d \bmod n$. Then $G(F(x)) = x$ for every x satisfying $\gcd(x, n) = 1$.*

Proof: By Fact 3, $3d = 1 + k\varphi(n)$ for some integer k. Now applying Fact 1, we find

$$G(F(x)) = (x^3)^d = x^{3d} = x^{1+k\varphi(n)} = x^1 \cdot (x^{\varphi(n)})^k = x \cdot 1^k = x \pmod{n}.$$

The theorem follows.

If the primes p, q are chosen to be large enough—say, 1024-bit primes—then it is believed to be computationally infeasible to recover p and q from n. In other words, in these circumstances it is believed hard to factor the integer $n = pq$. It is also believed to be hard to recover d from n. And, given knowledge of only n (but not d or p, q), it is believed to be computationally infeasible to compute the function G. The security of RSA will rely upon this hardness assumption.

12.4 RSA Signatures

We're now ready to describe the RSA signature scheme. The idea is that the function F defined in Theorem 1 will be our trapdoor one-way function. The public key is the number n, and the private key is the number d. Given the public key n and a number x, anyone can compute $F(x) = x^3 \bmod n$. As mentioned before, F is (believed) one-way: given $y = x^3 \bmod n$, there is no known way to recover x in any reasonable amount of computing time. However, we can see that the private key d provides a trapdoor: given d and y, we can compute $x = G(y) = y^d \bmod n$. The intuition underlying this trapdoor function is simple: anyone can cube a number modulo n, but computing cube roots modulo n is believed to be hard if you don't know the factorization of n.

We then apply this trapdoor one-way function to the basic approach outlined earlier. Thus, a signature on message M is a value S satisfying

$$H(M) = S^3 \bmod n.$$

The RSA signature scheme is defined by the following three algorithms:

- **Key generation:** We can pick a pair of random 1024-bit primes p, q that are both 2 mod 3. Then the public key is $n = pq$, and the private key is the value of d given by Fact 3 (it can be computed efficiently using the extended Euclidean algorithm).
- **Signing:** The signing algorithm is given by

$$\textsc{Sign}_d(M) = H(M)^d \bmod n.$$

- **Verification:** The verification algorithm Verify is given by

$$\textsc{Verify}_n(M, S) = \begin{cases} \text{true} & \text{if } H(M) = S^3 \bmod n, \\ \text{false} & \text{otherwise.} \end{cases}$$

Theorem 1 ensures the correctness of the verification algorithm, i.e., that $\textsc{Verify}_n(M, \textsc{Sign}_d(M)) = \text{true}$.

A quick reminder: in these notes we're developing the conceptual basis underlying MAC and digital signature algorithms that are widely used in practice, but again don't try to implement them yourself based upon just this discussion! We've omitted some technical details that do not change the big picture, but that are essential for security in practice. For your actual systems, use a reputable crypto library!

12.5 Definition of Security for Digital Signatures

Finally, let's outline a formal definition of what we mean when we say that a digital signature scheme is secure. The approach is very similar to what we saw for MACs.

We imagine a game played between Georgia (the adversary) and Reginald (the referee). Initially, Reginald runs KeyGen to get a keypair $\langle K, U \rangle$. Reginald sends the public key U to Georgia and keeps the private key K to himself. In each round of the game, Georgia may query Reginald with a message M_i; Reginald responds with $S_i = \textsc{Sign}_K(M_i)$. At any point, Georgia can yell "Bingo!" and output a pair $\langle M, S \rangle$. If this pair satisfies $\textsc{Verify}_U(M, S) = \text{true}$, and if Reginald has not been previously queried with the message M, then Georgia wins the game: she has forged a signature. Otherwise, Georgia loses.

If Georgia has any strategy to successfully forge a signature with non-negligible probability (say, with success probability at least $1/2^{40}$), given a generous amount of computation time (say, 2^{80} steps of computation) and any reasonable number of rounds of the game (say, 2^{40} rounds), then we declare the digital signature scheme insecure. Otherwise, we declare it secure.

This is a very stringent definition of security, because it declares the signature scheme broken if Georgia can successfully forge a signature on any message of her choice, even after tricking Alice into signing many messages of Georgia's choice. Nonetheless, modern digital signature algorithms—such as the RSA signature scheme—are believed to meet this definition of security.

Note however that the security of signatures do rely on the underlying hash function. Signatures have been broken in the past by taking advantage of the ability to create hash collisions when the hash function, not the public key algorithm, is compromised.

13 Certificates

So far we've seen powerful techniques for securing communication such that the only information we must carefully protect regards "keys" of various sorts. Given the success of cryptography in general, arguably the biggest challenge remaining for its effective use concerns exactly those keys, and how to *manage* them. For instance, how does Alice find out Bob's public key? Does it matter?

13.1 Man-in-the-middle Attacks

Suppose Alice wants to communicate securely with Bob over an insecure communication channel, but she doesn't know his public key (and he doesn't know hers). A naive strategy is that she could just send Bob a message asking him to send his public key, and accept whatever response she gets back (over the insecure communication channel). Alice would then encrypt her message using the public key she received in this way.

This naive approach is insecure. An *active attacker* (Mallory, in our usual terminology) could tamper with Bob's response, replacing the public key in Bob's response with the attacker's public key. When Alice encrypts her message, she'll be encrypting it under Mallory's public key, not Bob's public key. When Alice transmits the resulting ciphertext over the insecure communication channel, Mallory can observe the ciphertext, decrypt it with his private key, and learn the secret message that Alice was trying to send to Bob.

You might think that Bob could detect this attack when he receives a ciphertext that he is unable to decrypt using his own private key. However, an active attacker can prevent Bob from noticing the attack. After decrypting the ciphertext Alice sent and learning the secret message that Alice wanted to send, Mallory can re-encrypt Alice's message under Bob's public key, though not before possibly tampering with Alice's packet to replace her ciphertext with new ciphertext of Mallory's choosing. In this way, neither Alice nor Bob would have any idea that something has gone wrong. This allows an active attacker to spy on—*and alter*—Alice's secret messages to Bob, without breaking any of the cryptography.

If Alice and Bob are having a two-way conversation, and they both exchange their public keys over an insecure communication channel, then Mallory can mount a similar attack in both directions. As a result, Mallory will get to observe all of the secret messages that Alice and Bob send to each other, but neither Alice nor Bob will have any idea that something has gone wrong. This is known as a "*man-in-the-middle*" (MITM) attack because the attacker interposes between Alice and Bob.

Man-in-the-middle attacks were possible in this example because Alice did not have any way of authenticating Bob's alleged public key. The general strategy for preventing MITM attacks is to ensure that every participant can verify the authenticity of other people's public keys. But how do we do that, specifically? We'll look next at several possible approaches to secure key management.

13.2 Trusted Directory Service

One natural approach to this key management problem is to use a trusted directory service: some organization that maintains an association between the name of each participant and their public key. Suppose everyone trusts Dirk the Director to maintain this association. Then any time Alice wants to communicate with someone, say Bob, she can contact Dirk to ask him for Bob's public key. This is only safe if Alice trusts Dirk to respond correctly to those queries (e.g., not to lie to her, and to avoid being fooled by imposters pretending to be Bob): if Dirk is malicious or incompetent, Alice's security can be compromised.

On first thought, it sounds like a trusted directory service doesn't help, because it just pushes the problem around. If Alice communicates with the trusted directory service over an insecure communication channel, the entire scheme is insecure, because an active attacker can tamper with messages involving the directory service. To protect against this threat, Alice needs to know the directory service's public key, but where does she get *that* from? One potential answer might be to **hardcode** the public key of the directory service in the source code of all applications that rely upon the directory service. So this objection can be overcome.

A trusted directory service might sound like an appealing solution, but it has a number of shortcomings:

- *Trust:* It requires complete trust in the trusted directory service. Another way of putting this is that everyone's security is contingent upon the correct and honest operation of the directory service.
- *Scalability:* The directory service becomes a bottleneck. Everyone has to contact the directory service at the beginning of any communication with anyone new, so the directory service is going to be getting a lot of requests. It had better be able to answer requests very quickly, lest everyone's communications suffer.
- *Reliability:* The directory service becomes a single central point of failure. If it becomes unavailable, then no one can communicate with anyone not known to them. Moreover, the service becomes a single point of vulnerability to denial-of-service attacks: if an attacker can mount a successful DoS attack on the directory service, the effects will be felt globally.
- *Online:* Users will not be able to use this service while they are disconnected. If Alice is composing an email offline (say while traveling), and wants to encrypt it to Bob, her email client will not be able to look up Bob's public key and encrypt the email until she has connectivity again. As another example, suppose Bob and Alice are meeting in person in the same room, and Alice wants to use her phone to beam a file to Bob over infrared or Bluetooth. If she doesn't have general Internet connectivity, she's out of luck: she can't use the directory service to look up Bob's public key.
- *Security:* The directory service needs to be available in real time to answer these queries. That means that the machines running the directory service need to be Internet-connected at all times, so they will need to be carefully secured against remote

attacks.

Because of these limitations, the trusted directory service concept is not widely used in practice, except in the context of messengers (such as Signal), where in order to send a message, Alice already has to be online.

In this case, the best approach is described as "trust but verify" using a key transparency mechanism. Suppose Alice and Bob discovered each others keys through the central keyserver. If they are ever in person, they can examine their devices to ensure that Alice actually has the correct key for Bob and vice versa. Although inconvenient, this acts as a check on a rogue keyserver, as the rogue keyserver would know there is at least a chance of getting caught.

However, some of these limitations—specifically, the ones relating to scalability, reliability, and the requirement for online access to the directory service—can be addressed through a clever idea known as digital certificates.

13.3 Digital Certificates

Digital certificates are a way to represent an alleged association between a person's name and their public key, as attested by some certifying party.

Let's look at an example. As a professor at UC Berkeley, David Wagner is an employee of the state of California. Suppose that the state maintained a list of each state employee's public key, to help Californians communicate with their government securely. The governor, Jerry Brown, might control a private key that is used to sign statements about the public key associated with each employee. For instance, Jerry could sign a statement attesting that "David Wagner's public key is `0x092...3F`", signed using the private key that Jerry controls.

In cryptographic protocol notation, the certificate would look like this:

$$\{\text{David Wagner's public key is } \texttt{0x092...3F}\}_{K^{-1}_{\text{Jerry}}}$$

where here $\{M\}_{K^{-1}}$ denotes a digital signature on the message M using the private key K^{-1}. In this case, K^{-1}_{Jerry} is Jerry Brown's private key. This certificate is just some digital data: a sequence of bits. The certificate can be published and shared with anyone who wants to communicate securely with David.

If Alice wants to communicate securely with David, she can obtain a copy of this certificate. If Alice knows Jerry's public key, she can verify the signature on David's digital certificate. This gives her high confidence that indeed Jerry consented to the statement about the bit pattern of David's public key, because the valid signature required Jerry to decide to agree to apply his private key to the statement.

If Alice also considers Jerry trustworthy and competent at recording the association between state employees and their public keys, she can then conclude that David Wagner's public key is `0x092...3F`, and she can use this public key to securely communicate with David.

Notice that Alice did not need to contact a trusted directory service. She only needed to receive a copy of the digital certificate, but she could obtain it from *anyone*—by Googling it, by obtaining it from an untrusted directory service, by seeing it scrawled on a whiteboard, or by getting a copy from David himself. It's perfectly safe for Alice to download a copy of the certificate over an insecure channel, or to obtain it from an untrustworthy source, as long as she verifies the signature on the digital certificate and trusts Jerry for these purposes. The certificate is, in some sense, self-validating. Alice has *bootstrapped* her trust in the validity of David's public key based on her existing trust that she has a correct copy of Jerry's public key, *plus* her belief that Jerry takes the act of signing keys seriously, and won't sign a statement regarding David's public key unless Jerry is sure of the statement's correctness.

13.4 Public-Key Infrastructure (PKI)

Let's now put together the pieces. A *Certificate Authority* (CA) is a party who issues certificates. If Alice trusts some CA, and that CA issues Bob a digital certificate, she can use Bob's certificate to get a copy of Bob's public key and securely communicate with him. For instance, in the example of the previous section, Jerry Brown acted as a CA for all employees of the state of California.

In general, if we can identify a party who everyone in the world trusts to behave honestly and competently—who will verify everyone's identity, record their public key accurately, and issue a public certificate to that person accordingly—that party can play the role of a trusted CA. The public key of the trusted CA can be hardcoded in applications that need to use cryptography. Whenever an application needs to look up David Wagner's public key, it can ask David for a copy of his digital certificate, verify that it was properly signed by the trusted CA, extract David's public key, and then communicate securely with David using his public key.

Some of the criticisms of the trusted directory service mentioned earlier also apply to this use of CAs. For instance, the CA must be trusted by everyone: put another way, Alice's security can be breached if the CA behaves maliciously, makes a mistake, or acts without sufficient care. So we need to find a single entity whom everyone in the world can agree to trust—a tall order. However, digital certificates have better scalability, reliability, and utility than an online directory service.

For this reason, digital certificates are widely used in practice today, with large companies (e.g., Verisign) having thriving businesses acting as CAs.

This model is also used to secure the web. A web site that wishes to offer access via SSL (`https:`) can buy a digital certificate from a CA, who checks the identity of the web site and issues a certificate linking the site's domain name (e.g., `www.amazon.com`) to its public key. Every browser in the world ships with a list of trusted CAs. When you type in an `https:` URL into your web browser, it connects to the web site, asks for a copy of the site's digital certificate, verifies the certificate using the public key of the CA who issued it, checks that the domain name in the certificate matches the site that you asked to visit, and then establishes secure communications with that site using the public key in the digital certificate.

Web browsers come configured with a list of many trusted CAs. As a fun exercise, you might try listing the set of trusted CAs configured in your web browser and seeing how many of the names you can recognize. If you use Firefox, you can find this list by going to Preferences / Advanced / Certificates / View Certificates / Authorities. Firefox currently ships with about 88 trusted CAs preconfigured in the browser. Take a look and see what you think of those CAs. Do you know who those CAs are? Would you consider them trustworthy? You'll probably find many unfamiliar names. For instance, who is Unizeto? TURKTRUST? AC Camerfirma? XRamp Security Services? Microsec Ltd? Dhimyotis? Chunghwa Telecom Co.? Do you trust them?

The browser manufacturers have decided that, whether you like it or not, those CAs are trusted. You might think that it's an advantage to have many CAs configured into your browser, because that gives each user a choice depending upon whom they trust. However, that's not how web browsers work today. Your web browser will accept *any* certificate issued by *any* of these 88 CAs. If Dhimyotis issues a certificate for `amazon.com`, your browser will accept it. Same goes for all the rest of your CAs. This means that if any one of those 88 CAs issues a certificate to the wrong person, or behaves maliciously, that could affect the security of everyone who uses the web. The more CAs your browser trusts, the greater the risk of a security breach. That CA model is under increasing criticism for these reasons.

13.5 Certificate Chains and Hierarchical PKI

Above we looked at an example where Jerry Brown could sign certificates attesting to the public keys of every California state employee. However, in practice that may not be realistic. There are over 200,000 California state employees, and Jerry couldn't possibly know every one of them personally. Even if Jerry spent all day signing certificates, he still wouldn't be able to keep up—let alone serve as governor.

A more scalable approach is to establish a hierarchy of responsibility. Jerry might issue certificates to the heads of each of the major state agencies. For instance, Jerry might issue a certificate for the University of California, delegating to UC President Janet Napolitano the responsibility and authority to issue certificates to UC employees. Napolitano might sign certificates for all UC employees. We get:

$$\{\text{The University of California's public key is } K_{\text{Napolitano}}\}_{K^{-1}_{\text{Jerry}}}$$

$$\{\text{David Wagner's public key is } K_{\text{daw}}\}_{K^{-1}_{\text{Napolitano}}}$$

This is a simple example of a *certificate chain*: a sequence of certificates, each of which authenticates the public key of the party who has signed the next certificate in the chain.

Of course, it might not be realistic for President Napolitano to personally sign the certificates of all UC employees. We can imagine more elaborate and scalable scenarios. Jerry might issue a certificate for UC to Janet Napolitano; Napolitano might issue a certificate for UC Berkeley to UCB Chancellor Nicholas Dirks; Dirks might issue a certificate for the UCB EECS department to EECS Chair Randy Katz; and Katz might issue each EECS professor a certificate that attests to their name, public key, and status as a state employee. This would lead to a certificate chain of length 4.

In the latter example, Jerry acts as a Certificate Authority (CA) who is the authoritative source of information about the public key of each state agency; Napolitano serves as a CA who manages the association between UC campuses and public keys; Dirks serves as a CA who is authoritative regarding the public key of each UCB department; and so on. Put another way, Jerry delegates the power to issue certificates for UC employees to Napolitano; Napolitano further sub-delegates this power, authorizing Dirks to control the association between UCB employees and their public keys; and so on.

In general, the hierarchy forms a tree. The depth can be arbitrary, and thus certificate chains may be of any length. The CA hierarchy is often chosen to reflect organizational structures.

13.6 Revocation

What do we do if a CA issues a certificate in error, and then wants to invalidate the certificate? With the basic approach described above, there is nothing that can be done: a certificate, once issued, remains valid forever.

This problem has arisen in practice. A number of years ago, Verisign issued bogus certificates for "Microsoft Corporation" to . . . someone other than Microsoft. It turned out that Verisign had no way to revoke those bogus certificates. This was a serious security breach, because it provided the person who received those certificates with the ability to run software with all the privileges that would be accorded to the real Microsoft. How was this problem finally resolved? In the end, Microsoft issued a special patch to the Windows operating system that revoked those specific bogus certificates. The patch contained a hardcoded copy of the bogus certificates and inserted an extra check into the certificate-checking code: if the certificate matches one of the bogus certificates, then treat it as invalid. This addressed the particular issue, but was only feasible because Microsoft was in a special position to push out software to address the problem. What would we have done if a trusted CA had handed out a bogus certificate for Amazon.com, or Paypal.com, or BankofAmerica.com, instead of for Microsoft.com?

This example illustrates the need to consider revocation when designing a PKI system. There are two standard approaches to revocation:

- *Validity periods.* Certificates can contain an expiration date, so they're no longer considered valid after the expiration date. This doesn't let you immediately revoke a certificate the instant you discover that it was issued in error, but it limits the damage by ensuring that the erroneous certificate will eventually expire.

  With this approach, there is a fundamental tradeoff between efficiency and how quickly one can revoke an erroneous certificate. On the one hand, if the lifetime of each certificate is very short—say, each certificate is only valid for a single day, and then you must request a new one—then we have a way to respond quickly to bad certificates: a bad certificate will circulate for at most one day after we discover it. Since we won't re-issue certificates known to be bad, after the lifetime elapses the certificate has effectively been revoked. However, the problem with short lifetimes is that legitimate parties must frequently contact their CA to get new certificates; this puts a heavy load

on all the parties, and can create reliability problems if the CA is unreachable for a day. On the other hand, if we set the lifetime very long, then reliability problems can be avoided and the system scales well, but we lose the ability to respond promptly to erroneously issued certificates.

- *Revocation lists.* Alternatively, the CA could maintain and publish a list of all certificates it has revoked. For security, the CA could date and digitally sign this list. Every so often, everyone could download the latest copy of this revocation list, check its digital signature, and cache it locally. Then, when checking the validity of a digital certificate, we also check that it is not on our local copy of the revocation list.

  The advantage of this approach is that it offers the ability to respond promptly to bad certificates. There is a tradeoff between efficiency and prompt response: the more frequently we ask everyone to download the list, the greater the load on the bandwidth and on the CA's revocation servers, but the more quickly we can revoke bad certificates. If revocation is rare, this list might be relatively short, so revocation lists have the potential to be more efficient than constantly re-issuing certificates with a short validity period.

  However, revocation lists also pose some special challenges of their own. What should clients do if they are unable to download a recent copy of the revocation list? If clients continue to use an old copy of the revocation list, then this creates an opportunity for an attacker who receives a bogus certificate to DoS the CA's revocation servers in order to prevent revocation of the bogus certificate. If clients err on the safe side by rejecting all certificates if they cannot download a recent copy of the revocation list, this creates an even worse problem: an attacker who successfully mounts a sustained DoS attack on the CA's revocation servers may be able to successfully deny service to all users of the network.

  Today, systems that use revocation lists typically ignore these denial-of-service risks and hope for the best.

13.7 Web of Trust

Another approach is the so-called *web of trust*, which was pioneered by PGP, a software package for email encryption. The idea is to democratize the process of public key verification so that it does not rely upon any single central trusted authority. In this approach, each person can issue certificates for their friends, colleagues, and others whom they know.

Suppose Alice wants to contact Doug, but she doesn't know Doug. In the simplest case, if she can find someone she knows and trusts who has issued Doug a certificate, then she has a certificate for Doug, and everything is easy.

If that doesn't work, things get more interesting. Suppose Alice knows and trusts Bob, who has issued a certificate to Carol, who has in turn issued a certificate to Doug. In this case, PGP will use this certificate chain to identify Doug's public key.

In the latter scenario, is this a reasonable way for Alice to securely obtain a copy of Doug's

public key? It's hard to say. For example, Bob might have carefully checked Carol's identity before issuing her a certificate, but that doesn't necessarily indicate how careful or honest Carol will be in signing other people's keys. In other words, Bob's signature on the certificate for Carol might attest to Carol's *identity*, but not necessarily her honesty, integrity, or competence. If Carol is sloppy or malicious, she might sign a certificate that purports to identify Doug's public key, but actually contains some imposter's public key instead of Doug's public key. That would be bad.

This example illustrates two challenges:

- *Trust isn't transitive.* Just because Alice trusts Bob, and Bob trusts Carol, it doesn't necessarily follow that Alice trusts Carol. (More precisely: Alice might consider Bob trustworthy, and Bob might consider Carol trustworthy, but Alice might not consider Carol trustworthy.)
- *Trust isn't absolute.* We often trust a person for a specific purpose, without necessarily placing absolute trust in them. To quote one security expert: "I trust my bank with my money but not with my children; I trust my relatives with my children but not with my money." Similarly, Alice might trust that Bob will not deliberately act with malicious intent, but it's another question whether Alice trusts Bob to very diligently check the identity of everyone whose certificate he signs; and it's yet another question entirely whether Alice trusts Bob to have good judgement about whether third parties are trustworthy.

The web-of-trust model doesn't capture these two facets of human behavior very well.

The PGP software takes the web of trust a bit further. PGP certificate servers store these certificates and make it easier to find an intermediary who can help you in this way. PGP then tries to find *multiple* paths from the sender to the recipient. The idea is that the more paths we find, and the shorter they are, the greater the trust we can have in the resulting public key. It's not clear, however, whether there is any principled basis for this theory, or whether this really addresses the issues raised above.

One criticism of the web-of-trust approach is that, empirically, many users find it hard to understand. Most users are not experts in cryptography, and it remains to be seen whether the web of trust can be made to work well for non-experts. To date, the track record has not been one of strong success. Even in the security community, it is only partially used—not due to lack of understanding, but due to usability hurdles, including lack of integration into mainstream tools such as mail readers.

13.8 Leap-of-Faith Authentication

Another approach to managing keys is exemplified by SSH. The first time that you use SSH to connect to a server you've never connected to before, your SSH client asks the server for its public key, the server responds in the clear, and the client takes a "leap of faith" and

trustingly accepts whatever public key it receives.[15] The client remembers the public key it received from this server. When the client later connects to the same server, it uses the same public key that it obtained during the first interaction.

This is known as *leap-of-faith authentication*[16] because the client just takes it on faith that there is no man-in-the-middle attacker the first time it connects to the server. It has also sometimes been called *key continuity management*, because the approach is to ensure that the public key associated with any particular server remains unchanged over a long time period.

What do you think of this approach?

- A rigorous cryptographer might say: this is totally insecure, because an attacker could just mount a MITM attack on the first interaction between the client and server.
- A pragmatist might say: that's true, but it still prevents many kinds of attacks. It prevents passive eavesdropping. Also, it defends against any attacker who wasn't present during the first interaction, and that's a significant gain.
- A user might say: this is easy to use. Users don't need to understand anything about public keys, key management, digital certificates or other cryptographic concepts. Instead, the SSH client takes care of security for them, without their involvement. The security is invisible and automatic.

Key continuity management exemplifies several design principles for "usable security". One principle is that "there should be only one mode of operation, and it should be secure." In other words, users should not have to configure their software specially to be secure. Also, users should not have to take an explicit step to enable security protections; the security should be ever-present and enabled automatically, in all cases. Arguably, users should not even have the power to disable the security protections, because that opens up the risk of social engineering attacks, where the attacker tries to persuade the user to turn off the cryptography.

Another design principle: "Users shouldn't have to understand cryptography to use the system securely." While it's reasonable to ask the designers of the system to understand cryptographic concepts, it is not reasonable to expect users to know anything about cryptography.

[15] The client generally asks the user to confirm the trust decision, but users almost always ok the leap-of-faith.

[16] Another term is TOFU = Trust On First Use.

14 Passwords

Passwords are widely used for authentication, especially on the web. What practices should be used to make passwords as secure as possible?

14.1 Risks and weaknesses of passwords

Passwords have some well-known usability shortcomings. Security experts recommend that people pick long, strong passwords, but long random passwords are harder to remember. In practice, users are more likely to choose memorable passwords, which may be easier to guess. Also, rather than using a different, independently chosen password for each site, users often reuse passwords across multiple sites, for ease of memorization. This has security consequences as well.

From a security perspective, we can identify a number of security risks associated with password authentication:

- *Online guessing attacks.* An attacker could repeatedly try logging in with many different guesses at the user's password. If the user's password is easy to guess, such an attack might succeed.
- *Social engineering and phishing.* An attacker might be able to fool the user into revealing his/her password, e.g., on a phishing site. We've examined this topic previously, so we won't consider it further in these notes.
- *Eavesdropping.* Passwords are often sent in cleartext from the user to the website. If the attacker can eavesdrop (e.g., if the user is connecting to the Internet over an open Wifi network), and if the web connection is not encrypted, the attacker can learn the user's password.
- *Client-side malware.* If the user has a keylogger or other client-side malware on his/her machine, the keylogger/malware can capture the user's password and exfiltrate it to the attacker.
- *Server compromise.* If the server is compromised, an attacker may be able to learn the passwords of people who have accounts on that site. This may help the attacker break into their accounts on other sites.

We'll look at defenses and mitigations for each of these risks, below.

14.2 Mitigations for eavesdropping

There is a straightforward defense against eavesdropping: we can use SSL (also known as TLS). In other words, instead of connecting to the web site via http, the connection can be made over https. This will ensure that the username and password are sent over an encrypted channel, so an eavesdropper cannot learn the user's password.

Today, many sites do use SSL, but many do not.

Another possible defense would be to use more advanced cryptographic protocols. For instance, one could imagine a challenge-response protocol where the server sends your browser a random challenge r; then the browser takes the user's password w, computes $H(w, r)$ where H is a cryptographic hash (e.g., SHA256), and sends the result to the server. In this scheme, the user's password never leaves the browser and is never sent over the network, which defends against eavesdroppers. Such a scheme could be implemented today with Javascript on the login page, but it has little or no advantage over SSL (and it has some shortcomings compared to using SSL), so the standard defense is to simply use SSL.

14.3 Mitigations for client-side malware

It is very difficult to protect against client-side malware.

To defend against keyloggers, some people have proposed using randomized virtual keyboards: a keyboard is displayed on the screen, with the order of letters and numbers randomly permuted, and the user is asked to click on the characters of their password. This way, a keylogger (which only logs the key strokes you enter) would not learn your password. However, it is easy for malware to defeat this scheme: for instance, the malware could simply record the location of each mouse click and take a screen shot each time you click the mouse.

In practice, if you type your password into your computer and your computer has malware on it, then the attacker learns your password. It is hard to defend against this; passwords are fundamentally insecure in this threat model. The main defense is two-factor authentication, where we combine the password with some other form of authentication (e.g., a SMS sent to your phone).

14.4 Online guessing attacks

How easy are online guessing attacks? Researchers have studied the statistics of passwords as used in the field, and the results suggest that online guessing attacks are a realistic threat. According to one source, the five most commonly used passwords are `123456`, `password`, `12345678`, `qwerty`, `abc123`. Of course, a smart attacker will start by guessing the most likely possibilities for the password first before moving on to less likely possibilities. A careful measurement study found that with a dictionary of the 10 most common passwords, you can expect to find about 1% of users' passwords. In other words, about 1% of users choose a password from among the top 10 most commonly used passwords. It also found that, with a dictionary of the 2^{20} most commonly used passwords, you can expect to guess about 50% of users' passwords: about half of all users will have a password that is in that dictionary.

One implication is that, if there are no limits on how many guesses an attacker is allowed to make, an attacker can have a good chance of guessing a user's password correctly. We can distinguish targeted from untargeted attacks. A *targeted attack* is where the attacker has a particular target user in mind and wants to learn their password; an *untargeted attack* is where the attacker just wants to guess some user's password, but doesn't care which user gets hacked. An untargeted attack, for instance, might be relevant if the attacker wants to

take over some existing Gmail account and send lots of spam from it.

The statistics above let us estimate the work an attacker would have to do in each of these attack settings. For an untargeted attack, the attacker might try 10 guesses at the password against each of a large list of accounts. The attacker can expect to have to try about 100 accounts, and thus make a total of about 1000 login attempts, to guess one user's password correctly. Since the process of guessing a password and seeing if it is correct can be automated, resistance against untargeted attacks is very low, given how users tend to choose their passwords in practice.

For a targeted attack, the attacker's workload has more variance. If the attacker is extremely lucky, he might succeed within the first 10 guesses (happens 1% of the time). If the attacker is mildly lucky, he might succeed after about one million guesses (happens half of the time). If the attacker is unlucky, it might take a lot more than one million guesses. If each attempt takes 1 second (to send the request to the server and wait for the response), making 2^{20} guesses will take about 11 days, and the attack is very noticeable (easily detectable by the server). So, targeted attacks are possible, but the attacker is not guaranteed a success, and it might take quite a few attempts.

14.5 Mitigations for online guessing attacks

Let's explore some possible mitigations for online guessing:

- *Rate-limiting.* We could impose a limit on the number of consecutive incorrect guesses that can be made; if that limit is exceeded, the account is locked and the user must do something extra to log in (e.g., call up customer service). Or, we can impose a limit on the maximum guessing rate; if the number of incorrect guesses exceeds, say, 5 per hour, then we temporarily lock the account or impose a delay before the next attempt can be made.

  Rate-limiting is a plausible defense against targeted attacks. It does have one potential disadvantage: it introduces the opportunity for denial-of-service attacks. If Mallory wants to cause Bob some grief, Mallory can make enough incorrect login attempts to cause Bob's account to be locked. In many settings, though, this denial-of-service risk is acceptable. For instance, if we can limit each account to 5 incorrect guesses per hour, making 2^{20} guesses would take at least 24 years—so at least half of our user population will become essentially immune to targeted attacks.

  Unfortunately, rate-limiting is not an effective defense against untargeted attacks. An attacker who can make 5 guesses against each of 200 accounts (or 1 guess against each of 1000 accounts) can expect to break into at least one of them. Rate-limiting probably won't prevent the attacker from making 5 guesses (let alone 1 guess).

  Even with all of these caveats, rate-limiting is probably a good idea. Unfortunately, one research study found that only about 20% of major web sites currently use rate-limiting.

- *CAPTCHAs.* Another approach could be to try to make it harder to perform *automated*

online guessing attacks. For instance, if a login attempt for some user fails, the system could require that the next time you try to log into that same account, you have to solve a CAPTCHA. Thus, making n guesses at the password for a particular user would require solving $n-1$ CAPTCHAs. CAPTCHAs are designed to be solvable for humans but (we hope) not for computers, so we might hope that this would eliminate automated/scripted attacks.

Unfortunately, this defense is not as strong as we might hope. There are black-market services which will solve CAPTCHAs for you. They even provide easy-to-use APIs and libraries so you can automate the process of getting the solution to the CAPTCHA. These services employ human workers in countries with low wages to solve the CAPTCHAs. The market rate is about \$1–2 per thousand CAPTCHAs solved, or about 0.1–0.2 cents per CAPTCHA solved. This does increase the cost of a targeted attack, but not beyond the realm of possibility.

CAPTCHAs do not stop an untargeted attack. For instance, an attacker who makes one guess at each of 1000 accounts won't have to solve any CAPTCHAs. Or, if for some reason the attacker wants to make 10 guesses at each of 100 accounts, the attacker will only have to solve 900 CAPTCHAs, which will cost the attacker maybe a dollar or two: not very much.

- *Password requirements or nudges.* A site could also impose password requirements (e.g., your password must be 10 characters long and contain at least 1 number and 1 punctuation symbol). However, these requirements offer poor usability, are frustrating for users, and may just tempt some users to evade or circumvent the restriction, thus not helping security. Therefore, I would be reluctant to recommend stringent password requirements, except possibly in special cases.

  Another approach is to apply a gentle "nudge" rather than impose a hard requirement. For instance, studies have found that merely showing a password meter during account creation can help encourage people to choose longer and stronger passwords.

14.6 Mitigations for server compromise

The natural way to implement password authentication is for the website to store the passwords of all of its passwords in the clear, in its database. Unfortunately, this practice is bad for security. If the site gets hacked and the attacker downloads a copy of the database, then now all of the passwords are breached; recovery may be painful. Even worse, because users often reuse their passwords on multiple sites, such a security breach may now make it easier for the attacker to break into the user's accounts on other websites.

For these reasons, security experts recommend that sites avoid storing passwords in the clear. Unfortunately, sites don't always follow this advice. For example, in 2009, the Rockyou social network got hacked, and the hackers stole the passwords of all 32 million of their users and posted them on the Internet; not good. One study estimates that about 30–40% of sites still store passwords in the clear.

14.6.1 Password hashing

If storing passwords in the clear is not a good idea, what can we do that is better? One simple approach is to hash each password with a cryptographic hash function (say, SHA256), and store the hash value (not the password) in the database.

In more detail, when Alice creates her account and enters her password w, the system can hash w to get $H(w)$ and store $H(w)$ in the user database. When Alice returns and attempts to log in, she provides a password, say w'; the system can check whether this is correct by computing the hash $H(w')$ of w' and checking whether $H(w')$ matches what is in the user database.

Notice that the properties of cryptographic hash functions are very convenient for this application. Because cryptographic hash functions are one-way, it should be hard to recover the password w from the hash $H(w)$; so if there is a security breach and the attacker steals a copy of the database, no cleartext passwords are revealed, and it should be hard for the attacker to invert the hash and find the user's hashes. That's the idea, anyway.

Unfortunately, this simple idea has some shortcomings:

- *Offline password guessing.* Suppose that Mallory breaks into the website and steals a copy of the password database, so she now has the SHA256 hash of Bob's password. This enables her to test guesses at Bob's password very quickly, on her own computer, without needing any further interaction with the website. In particular, given a guess g at the password, she can simply hash g to get $H(g)$ and then test whether $H(g)$ matches the password hash in the database. By using lists of common passwords, English words, passwords revealed in security breaches of sites who didn't use password hashing, and other techniques, one can generate many guesses. This is known as an *offline guessing attack*: offline, because Mallory doesn't need to interact with the website to test a guess at the password, but can check her guess entirely locally.

  Unfortunately for us, a cryptographic hash function like SHA256 is very fast. This lets Mallory test many guesses rapidly. For instance, on modern hardware, it is possible to test something in the vicinity of 1 billion passwords per second (i.e., to compute about 1 billion SHA256 hashes per second). So, imagine that Mallory breaks into a site with 100 million users. Then, by testing 2^{20} guesses at each user's password, she can learn about half of those users' passwords. How long will this take? Well, Mallory will need to make 100 million $\times 2^{20}$ guesses, or a total of about 100 trillion guesses. At 1 billion guesses per second, that's about a day of computation. Ouch. In short, the hashing of the passwords helps some, but it didn't help nearly as much as we might have hoped.

- *Amortized guessing attacks.* Even worse, the attack above can be sped up dramatically by a more clever algorithm that avoids unnecessarily repeating work. Notice that we're going to try guessing the same 2^{20} plausible passwords against each of the users. And, notice that the password hash $H(w)$ doesn't depend upon the user: if Alice and Bob both have the same password, they'll end up with the same password hash.

  So, consider the following optimized algorithm for offline password guessing. We com-

pute a list of 2^{20} pairs $(H(g), g)$, one for each of the 2^{20} most common passwords g, and sort this list by the hash value. Now, for each user in the user database, we check to see whether their password hash $H(w)$ is in the sorted list. If it is in the list, then we've immediately learned that user's password. Checking whether their password hash is in the sorted list can be done using binary search, so it can be done extremely efficiently (with about $\lg 2^{20} = 20$ random accesses into the sorted list). The attack requires computing 2^{20} hashes (which takes about one millisecond), sorting the list (which takes fractions of a second), and doing 100 million binary searches (which can probably be done in seconds or minutes, in total). This is *much* faster than the previous offline guessing attack, because we avoid repeated work: we only need to compute the hash of each candidate password once.

14.6.2 Password hashing, done right

With these shortcomings in mind, we can now identify a better way to store passwords on the server.

First, we can eliminate the amortized guessing attack by *incorporating randomness into the hashing process*. When we create a new account for some user, we pick a random *salt* s. The salt is a value whose only purpose is to be different for each user; it doesn't need to be secret. The password hash for password w is $H(w, s)$. Notice that the password hash depends on the salt, so even if Alice and Bob share the same password w, they will likely end up with different hashes (Alice will have $H(w, s_A)$ and Bob $H(w, s_B)$, where most likely $s_A \neq s_B$). Also, to enable the server to authenticate each user in the future, the salt for each user is stored in the user database.

Instead of storing $H(w)$ in the database, we store $s, H(w, s)$ in the database, where s is a random salt. Notice that s is stored in cleartext, so if the attacker gets a copy of this database, the attacker will see the value of s. That's OK; the main point is that each user will have a different salt, so the attacker can no longer use the amortized guessing attack above. For instance, if the salt for Alice is s_A, the attacker can try guesses $g_1, g_2, \ldots, g_n$ at her password by computing $H(g_1, s_A), \ldots, H(g_n, s_A)$ and comparing each one against her password hash $H(w_A, s_A)$. But now when the attacker wants to guess Bob's password, he can't reuse any of that computation; he'll need to compute a new, different set of hashes, i.e., $H(g_1, s_B), \ldots, H(g_n, s_B)$, where s_B is the salt for Bob.

Salting is good, because it increases the attacker's workload to invert many password hashes. However, it is not enough. As the back-of-the-envelope calculation above illustrated, an attacker might still be able to try 2^{20} guesses at the password against each of 100 million users' password hashes in about a day. That's not enough to prevent attacks. For instance, when LinkedIn had a security breach that exposed the password hashes of all of their users, it was discovered that they were using SHA256, and consequently one researcher was able to recover 90% of their users' passwords in just 6 days. Not good.

So, the second improvement is to *use a slow hash*. The reason that offline password guessing is so efficient is because SHA256 is so fast. If we had a cryptographic hash that was very slow—say, it took 1 millisecond to compute—then offline password guessing would be much

slower; an attacker could only try 1000 guesses at the password per second.

One way to take a fast hash function and make it slower is by iterating it. In other words, if H is a cryptographic hash function like SHA256, define the function F by

$$F(x) = H(H(H(\cdots(H(x))\cdots))),$$

where we have iteratively applied H n times. Now F is a good cryptographic hash function, and evaluating F will be n times slower than evaluating H. This gives us a tunable parameter that lets us choose just how slow we want the hash function to be.

Therefore, our final construction is to store $s, F(w, s)$ in the database, where s is a randomly chosen salt, and F is a slow hash constructed as above. In other words, we store

$$s, H(H(H(\cdots(H(w, s))\cdots)))$$

in the database.

How slow should the hash function F be? In other words, how should we choose n? On the one hand, for security, we'd like n to be as large as possible: the larger it is, the slower offline password guessing will be. On the other hand, we can't make it too large, because that will slow down the legitimate server: each time a user tries to log in, the server needs to evaluate F on the password that was provided. With these two considerations, we can now choose the parameter n to provide as much security as possible while keeping the performance overhead of slow hashing down to something unnoticeable.

For instance, suppose we have a site that expects to see at most 10 logins per second (that would be a pretty high-traffic site). Then we could choose n so that evaluating F takes about one millisecond. Now the legitimate server can expect to spend 1% of its CPU power on performing password hashes—a small performance hit. The benefit is that, if the server should be compromised, offline password guessing attacks will take the attacker a lot longer. With the example parameters above, instead of taking 1 day to try 2^{20} candidate passwords against all 100 million users, it might take the attacker about 3000 machine-years. That's a real improvement.

In practice, there are several existing schemes for slow hashing that you can use: Scrypt, Bcrypt, or PBKDF2. They all use some variant of the "iterated hashing" trick mentioned above.

14.7 Implications for cryptography

The analysis above has implications for the use of human-memorable passwords or passphrases for cryptography.

Suppose we're building a file encryption tool. It is tempting to prompt the user to enter in a password w, hash it using a cryptographic hash function (e.g., SHA256), use $k = H(w)$ as a symmetric key, and encrypt the file under k. Unfortunately, this has poor security. An attacker could try the 2^{20} most common passwords, hash each one, try decrypting under that key, and see if the decryption looks plausibly like ciphertext. Since SHA256 is fast, this

attack will be very fast, say one millisecond; and based upon the statistics mentioned above, this attack might succeed half of the time or so.

You can do a little bit better if you use a slow hash to generate the key instead of SHA256. Unfortunately, this isn't enough to get strong security. For example, suppose we use a slow hash tuned to take 1 millisecond to compute the hash function. Then the attacker can make 1000 guesses per second, and it'll take only about 15 minutes to try all 2^{20} most likely passwords; 15 minutes to have a 50% chance of breaking the crypto doesn't sound so hot.

The unavoidable conclusion is that deriving cryptographic keys from passwords, passphrases, or human-memorable secrets is usually not such a great idea. Password-based keys tend to have weak security, so they should be avoided whenever possible. Instead, it is better to use a truly random cryptographic key, e.g., a truly random 128-bit AES key, and find some way for the user to store it securely.

14.8 Alternatives to passwords

Finally, it is worth noting that there are many alternatives to passwords, for authenticating to a server. Some examples include:

- Two-factor authentication.
- One-time PINs (e.g., a single-use code sent via SMS to your phone, or a hardware device such as RSA SecurID).
- Public-key cryptography (e.g., SSH).
- Secure persistent cookies.

We most likely won't have time to discuss any of these further in this class, but they are worth knowing about, for situations where you need more security than passwords can provide.

14.9 Summary

The bottom line is: don't store passwords in the clear. Instead, sites should store passwords in hashed form, using a slow cryptographic hash function and a random salt. If the user's password is w, one can store

$$s, H(H(H(\cdots(H(w,s))\cdots)))$$

in the database, where s is a random salt chosen randomly for that user and H is a standard cryptographic hash function.

15 Case Studies

TODO: Under construction.

16 Bitcoin

16.1 Problem Statement

Bitcoin is a digital cryptocurrency, which means it should have all the same properties as physical currency (e.g. the United States dollar). In our simplified model, a functioning currency should have the following properties:

- Each person has a bank account, in which they can store units of currency they own.
- Alice cannot impersonate Bob and perform actions as Bob.
- Any two people can engage in a *transaction*. Alice can send Bob n units of currency. This will cause Alice's bank account balance to decrease by n units, and Bob's bank account to increase by n units.
- If Alice has n units of currency in her account, she cannot spend any more than n units in any transaction.

In traditional physical currency, these properties are enforced by a trusted, centralized party such as a bank. Everyone trusts the bank to keep an accurate list of account holders with their appropriate account balances, and ensure that the identity of each user is correct before proceeding with a transaction. So, if Alice sends n units to Bob, both Alice and Bob trust that the bank will correctly decrease Alice's balance by n and increase Bob's balance by n. Everyone also trusts that the bank will not let Alice spend $n+1$ units of currency if she only has n units in her account.

The goal of Bitcoin is to replicate these basic properties of a functioning currency system, but without any centralized party. Instead of relying on a trusted entity, Bitcoin uses cryptography to enforce the basic properties of currency.

16.2 Cryptographic Primitives

Bitcoin uses two cryptographic primitives that you have already seen in this class. Let's briefly review their definitions and relevant properties.

A *cryptographic hash* is a function H that maps arbitrary-length input x to a fixed-length output $H(x)$. The hash is collision-resistant, which means it is infeasible to find two different inputs that map to the same output. In math, it is infeasible to find $x \neq y$ such that $H(x) = H(y)$.

A *digital signature* is a cryptographic scheme that guarantees authenticity on a message. Alice generates a public verification key PK and a secret signing key SK. She broadcasts the public key to the world and keeps the secret key to herself. When Alice writes a message, she uses the secret key to generate a signature on her message and attaches the signature to the message. Anyone else can now use the public key to verify that the signature is valid, proving that the message was written by Alice and nobody tampered with it.

With these two cryptographic primitives in mind, we can now start designing Bitcoin.

16.3 Identities

Since there is no centralized party to keep track of everyone's accounts, we will need to assign a unique identity to everyone. We also need to prevent malicious users from pretending to be other users.

Every user of Bitcoin generates a public key and private key. Their identity is the public key. For example, Bob generates PK_B and SK_B and publishes PK_B to the world, so now his identity in Bitcoin is PK_B. When Bob is interacting with Bitcoin, he can prove that he is the user corresponding to PK_B by creating a message and signing it with SK_B. Then anybody can use PK_B to verify his signature and confirm that he is indeed the PK_B user. Because digital signatures are unforgeable, an attacker who doesn't know Bob's secret signing key will be unable to impersonate Bob, because the attacker cannot generate a signature that validates with PK_B.

16.4 Transactions

Without a centralized party to validate transactions, we will need a way to cryptographically verify that Alice actually wants to send n units of currency to Bob. Fortunately, this problem is essentially solved with our identity scheme above. If Alice wants to send n units of currency to Bob, she can create a message "PK_A sends n units of currency to PK_B" and sign it with her secret key. Note how she uses her public key PK_A as her identity and Bob's public key PK_B as his identity. Now anybody can verify the signature with Alice's public key to confirm that the user PK_A did intend to make this transaction. Bitcoin doesn't validate the recipient–if someone wanted to refuse a transaction, they could create another transaction to send the money back.

16.5 Balances

In our transaction scheme so far, nothing is stopping Alice from creating and signing a message "PK_A sends $100n$ units of currency to PK_B," even though she may only have n units of currency to spend. We need some way to keep track of each user's balances.

For now, assume that there is a *trusted ledger*. A ledger is a written record that everybody can view. It is append-only and immutable, which means you can only add new entries to the ledger, and you cannot change existing entries in the ledger. You can think of the ledger like a guest book: when you visit, you can add your own entry, and you can view existing entries, but you cannot (or should not) change other people's old entries. Later we will see how to build a decentralized ledger using cryptography.

Bitcoin does not explicitly record the balance of every user. Instead, every completed transaction (along with its signature) is recorded in the public ledger. Since everyone can view the ledger, anybody can identify an invalid transaction, such as Alice trying to spend more than she has. For example, suppose Bob starts with \$10 and everyone else starts with \$0. (We will discuss where Bob got the \$10 later.) Consider the following ledger:

- PK_B (Bob) sends PK_A (Alice) \$5. Message signed with SK_B.

- PK_B (Bob) sends PK_M (Mallory) \$2. Message signed with SK_B.
- PK_M (Mallory) sends PK_A (Alice) \$1. Message signed with SK_M.
- PK_A (Alice) sends PK_E (Eve) \$9. Message signed with SK_A.

Can you spot the invalid transaction? Although we don't have the balances of each user, the transaction ledger gives us enough information to deduce every user's balance at any given time. In this example, after the first three transactions, Bob has \$3, Mallory has \$1, and Alice has \$6. In the fourth transaction, Alice is trying to spend \$9 when she only has \$6, so we know it must be an invalid transaction. Because the ledger is trusted, it will reject this invalid transaction.

Thus, the idea is to have each block have a list of the transactions that show where the money being used in this transaction came from, which also means that blocks have to be sorted in order of creation. Now, our ledger looks as follows (again assuming that Bob starts with $10B$ and everyone else starts with $0B$:

- TX_1 = PK_B (Bob) sends PK_A (Alice) $5B$, and the money came from the initial budget. TX_1 signed with SK_B - TX_2 = PK_A (Alice) sends PK_E (Eve) $5B$, and the money came from TX_1. TX_2 signed with SK_A

So, to check a transaction, we follow four steps:

1. Check that the signature on the transaction is verified using the PK of the sender
2. Check that the sender in this transaction was the receiver in some previous transaction
3. Check that the sender in this transaction has not spent the money in some previous transaction
4. Check that the sender has the appropriate amount of money

If we were checking TX_2, we first check that TX_2 was actually signed by Alice. Then, we check that Alice received some money in the past by checking the previous transactions. In TX_2, we see that Alice received the money from TX_1, and checking TX_1 verifies that Alice was the receiver. Next, we check that Alice has not spent the money earlier, so we scan the history of the blockchain and we don't see anywhere where the money from TX_1 was used. Finally, we check that Alice has 5 B by again checking TX_1 and seeing that she did receive 5 B from Bob. At this point, we have verified that TX_2 is a valid transaction, and we thus append it to the blockchain ledger.

At this point, we have created a functioning currency:

- Each person has a unique account, uniquely identified by public key.
- Users cannot impersonate other users, because each user can be validated by a secret signing key that only that user knows.
- Users can engage in a transaction by having the sender add their transaction to the ledger, with a signature on the transaction.

- Users cannot spend more than their current balance, because the trusted ledger is append-only, and everyone is able to calculate balances from the ledger.

The only remaining design element is creating a decentralized append-only ledger, which we will discuss next.

16.6 Hash chains

Recall that we need a public ledger that is append-only and immutable: everyone can add entries to the ledger, but nobody can modify or delete existing entries.

To build this ledger, we will start with a *hash chain*. Suppose we have five messages, $m_1, m_2, \ldots, m_5$ that we want to append to the ledger. The resulting hash chain would look like this:

Block 1	Block 2	Block 3	Block 4	Block 5
m_1	$m_2, H(\text{Block 1})$	$m_3, H(\text{Block 2})$	$m_4, H(\text{Block 3})$	$m_5, H(\text{Block 4})$

Note that each block contains the hash of the previous block, which in turn contains the hash of the previous block, etc. In other words, each time we append a new message in a new block, the hash of the previous block contains a digest of all the entries in the hash chain so far.

Another way to see this is to write out the hashes. For example:

$$\begin{aligned}\text{Block 4} &= m_4, H(\text{Block 3}) \\ &= m_4, H(m_3, H(\text{Block 2})) \\ &= m_4, H(m_3, H(m_2, H(\text{Block 1}))) \\ &= m_4, H(m_3, H(m_2, H(m_1)))\end{aligned}$$

Note that Block 4 contains a digest of all the messages so far, namely m_1, m_2, m_3, m_4.

16.7 Properties of Hash Chains

Assume that Alice is given the $H(\text{Block } i)$ from a trusted source, but she downloads blocks 1 through i from an untrusted source. Only using the $H(\text{Block } i)$, Alice can verify that the blocks she downloaded from the untrusted source are not compromised by recomputing the hashes of each block, checking that they match the hash in the next block, and so on, until the last block, which she checks against the hash she received from the trusted source. Let's walk through an example:

Say Alice received the $H(\text{Block 4})$ from somewhere she trusts and then fetches the entire blockchain from a compromised server (so she downloads blocks 1 through 4). Can an attacker give Alice an incorrect chain, say with block 2 being incorrect, without her detecting it? No! Since we use cryptographic hashes, which are collision resistant, two different blocks cannot hash to the same value. Say that block 2 is incorrect and Alice instead received block $2'$, then $H(\text{Block } 2') \neq H(\text{Block 2})$. Since block 3 includes the hash of block $2'$,

block 3 will also be incorrect, so the third block that Alice received is block $3' \neq$ block 3. So, $H(\text{Block } 3') \neq H(\text{Block } 3)$. Then, since block 4 includes the hash of block $3'$, block 4 will also be incorrect, so the fourth block that Alice received is block $4' \neq$ block 4. So, $H(\text{Block } 4') \neq H(\text{Block } 4)$. Since Alice received H(block 4) from a trusted source, and it does not match up with $H(\text{Block}4')$, Alice is able to detect misbehavior. On the other hand, if the $H(\text{Block } 4')$ did match $H(\text{Block } 4)$, then the blockchain that Alice downloaded is correct, and we have no misbehavior.

So, perhaps the most important property in a hash chain is that if you get the hash of the latest block from a trusted source, then you can verify that all of the previous history is correct.

16.8 Consensus in Bitcoin

In Bitcoin, every participant in the network stores the entire blockchain (and thus all of its history) since we don't utilize a centralized server. When someone wants to create a new transaction, they broadcast that transaction to everyone, and each user on the network has to check the transaction. If the transaction is correct, they will append it to their local blockchain.

The issue is that some users might be malicious, meaning that they might not append certain transactions or might not check certain transactions correctly or might replay certain transactions or might allow invalid transactions. Bitcoin, however, assumes that the majority of users are honest.

Perhaps one of the biggest issues is forks, which are essentially different versions of the blockchain that exist at the same time. For example, say that Mallory bought a house from Bob for 500 B, and this transaction is appended to the ledger. Mallory can then try "go back in time" and start the blockchain from just before this transaction was added to it, and can start appending new transaction entries from there. If Mallory can get other users to accept this new forked chain, she can get her 500 B back!

This means that we need a way for all users to agree on the content of the blockchain: *consensus via proof of work*.

16.9 Consensus via Proof of Work

In Bitcoin, while every user locally stores the entire blockchain, not every user can add a block. This special privilege is reserved for certain users, known as *miners*, who can only add a block if they have a valid proof of work. A miner validates transactions before solving a *proof of work*, which, if completed before any other miner, allows the miner to append the block to the blockchain. The proof of work is a computational puzzle that takes the hash of the current block concatenated with a random number. This random number can be incremented so that the hash changes, until the proof of work is solved. The proof of work is considered solved when the resulting hash starts with N zero bits, where the value of N (e.g. 33) is determined by the Bitcoin algorithm.

Miners then broadcast blocks with their proof of work. All honest miners listen for such blocks, check the blocks for correctness, and *accept the longest correct chain*. If a miner appends a block with some incorrect transaction, the block is ignored. The key idea for consensus is that everyone will always prefer the longest correct chain. Thus, if multiple miners append blocks at the same time, consensus is gained by the longest correct chain, and the rest of the "versions" are discarded. When two different miners at the same time solve a proof of work and append two different blocks, thus forking the network, the next miner that appends onto one of these chains invalidates the other chain.

Say for example that an honest miner M_1 stores the current local blockchain $b_1 \rightarrow b_2 \rightarrow b_3$, and hears about transaction T. M_1 checks T, then tries to mine (solve for the proof of work) for a new block b_4 to now include transaction T. However, if miner M_2 mines b_4 first, M_2 will broadcast $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$. M_1 checks b_4, accepts it, gives up mining block 4, then starts to mine for block 5. M_1 now has the blockchain $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$ stored locally and has started to mine b_5. However, if M_1 hears miner 3 broadcasts $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b'_4 \rightarrow b'_5$, M_1 will discard the shorter blockchain ($b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$) in favor of the longer one ($b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b'_4 \rightarrow b'_5$). By always accepting the longest blockchain, all the miners are ensured to have the same blockchain view.

Remember that Bitcoin assumes that more than half of the users are honest, meaning that more than half of the computing power is in the hands of honest miners, thus ensuring that honest miners will always have an advantage to mine the longest chain. Going back to the example about forks that prompted this discussion, if proof of consensus is implemented, Mallory cannot fork the blockchain since she does not have ¿50% of the computing power in the world. Since the longest chain is always taken as the accepted, Mallory's forked chain will be shorter unless she can mine new entries faster than the aggregate mining power of everyone else in the world.

Go forth and mine!