Cryptography — CS 161 Computer Security — Notes

Lecture notes by David Wagner

Last update: January 18, 2021

Contact for corrections: Peyrin Kao (peyrin at berkeley.edu)

1 Symmetric-Key Cryptography

1.1 Overview

Over the next several lectures we'll be studying techniques for securing information and communication in several fundamental ways: *confidentiality* (preventing adversaries from reading our private data), *integrity* (preventing them from altering it), and *authenticity* (determining who created a given document). In a nutshell, cryptography is about communicating securely over insecure communications media.

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.

We will start by examining *symmetric-key cryptography*, where both endpoints of a communication share the same key. This is the most classical notion of cryptography. We will study several primitives. *Symmetric-key encryption* provides confidentiality: Alice can encrypt her messages under a key K shared with Bob, and then send the ciphertext to Bob; Bob will be use K to decrypt, but eavesdroppers won't be able to learn anything about the messages Alice is sending. *Message authentication codes (MACs)* provide integrity and authenticity, and act more or less like a keyed checksum. When Alice sends a message M, she can append $F_K(M)$, a "checksum" computed on M using key K; now if Bob has the same key K, he can check that the checksum is valid. MACs are designed so that a man-in-the-middle who doesn't know the key K will be unable to modify the message without invalidating the "checksum".

Then, we will examine *public-key cryptography. Public-key encryption* provides confidentiality. Bob generates a matching public key and private key, and shares the public key with his correspondents (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, but no one else will be able to learn anything about the message. *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 her correspondents (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 in transit.

Then, we will study how to combine these basic building blocks to achieve useful things with cryptography.

1.2 Disclaimer

Caution! Here lies dragons. 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. You need to understand them at a conceptual level to get a good feeling for how industrial systems use cryptography in practice.

However, using these cryptographic building blocks securely in a real system requires attention to a lot of intricate details—and we won't have time to teach all of those details and pitfalls to you in CS161. Therefore, we do not recommend that you try to implement your own cryptography using the algorithms we teach you in class.

Later in the class, we'll give you some practical advice on what you *should* do instead. For now, know that we're going to teach you just enough to be dangerous, but not enough to implement industrial-strength cryptography in practice.

1.3 Brief History of Cryptography

The word "cryptography" comes from the Latin roots *crypt*, meaning secret, and *graphia*, meaning writing. So cryptography is 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 simply 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 developement 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 engineering feat. 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.

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 later in these notes) is due to Shannon. The early 1970s saw the the introduction by NIST (the National Institute for Standards in Technology) of a standardized cryptosystem, *DES*. DES answered the growing need for digital encryption standards in banking and other business. The decade starting in the late 1970s then saw an explosion of work on a computational theory of cryptography.

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. In actual fact they only have available a telephone line or an Internet connection subject to tapping by an eavesdropping adversary, *Eve*. 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 content of their exchange. In other words, we wish to simulate the ideal communication channel using only the available insecure channel.

Encryption focuses on ensuring the *confidentiality* of communications between Alice and Bob. In the *symmetric-key model*, Alice and Bob share a secret key K that is unknown to Eve. Generally K is generated in a random fashion, and for now we don't concern ourselves with how Alice and Bob manage to both have copies of it.

Alice encrypts her message M using the key K, and Bob decrypts the received ciphertext (to recover the original message) using the same key K. The unencrypted message M is often referred to as the *plaintext*, and its encryption as the *ciphertext*.

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

To consider this question, recall (from the earlier notes about principles for secure systems) *Kerkhoff's principle*:

> *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* Don't rely on security through obscurity.*)*

Consistent with Kerkhoff's principle, we will assume that Eve knows the encryption and decryption algorithms.[1] The only information Eve is missing is the secret key K. There are several possibilities about how much access Eve has to the insecure channel:

[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 Engima machine, and indeed they did.

1. Eve has managed to intercept a single encrypted message and wishes to recover the plaintext (the original message).
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.
3. Eve can trick Alice to encrypt messages $M_1, M_2, \ldots, M_n$ 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 M that is unknown to Eve; Eve intercepts the encryption of M and aims to recover M given what Eve has observed about the encryptions of $M_1, M_2, \ldots, M_n$.
4. Eve can trick Bob into decrypting some ciphertexts $C_1, \ldots, C_n$. Eve would like to use this to learn the decryption of some other ciphertext C (different from $C_1, \ldots, C_n$).
5. A combination of cases 3 and 4: 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 (and in particular did not occur as a result of Eve's trickery).

The first case is known as a *ciphertext-only attack*.

The second 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.

The third case is known as a *chosen-plaintext attack*, and the fourth as a *chosen-ciphertext attack*.

The fifth 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 we *can*, i.e., it is in fact feasible to provide good security even against this very powerful attack model. However, to keep the presentation simple, for the moment these notes will focus primarily on security against chosen-plaintext attack. We will come back to chosen-plaintext/ciphertext attacks later.

1.4 One Time Pad

The one time pad is a simple and idealized encryption scheme that helps illustrate some important concepts. In this 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).

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: $c_j = m_j \oplus k_j$, where $\oplus$ is the *XOR* (exclusive-or) of the two bits (0 if the two bits are the same, 1 if they are different).[2]

Decryption is equally simple: $m_j = c_j \oplus k_j$.

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: $M = C \oplus K$.

Let us now analyze how much information Eve gets about the plaintext M by intercepting the ciphertext C. What is the correct measure of this information gained by Eve? It might be the case that Eve had some partial information about 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 we do not know which. The security property will be: intercepting the ciphertext C should give Eve no additional information about the message M (i.e., she should not learn any new information about M beyond what she already knew before she intercepted C).

Let's prove that the one-time pad meets this security property. Consider the following experiment. Suppose Alice has sent one of two messages M_0 or M_1, and Eve has no idea which was sent. Eve tries to guess which was sent by looking at the ciphertext. We will show that Eve's probability of guessing correctly is 1/2, which is no different than it would be if she had not intercepted the ciphertext at all.

The proof is very simple: 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 the plaintext.

[2] Since we'll be making heavy use of *XOR* as we explore various cryptographic schemes, be sure to keep in mind its basic properties:

$$\begin{aligned} x \oplus 0 &= x && \text{0 is the identity} \\ x \oplus x &= 0 && x \text{ is its own inverse} \\ x \oplus y &= y \oplus x && \text{commutative property} \\ (x \oplus y) \oplus z &= x \oplus (y \oplus z) && \text{associative property} \end{aligned}$$

One handy identity that follows from these is:

$$x \oplus y \oplus x = y.$$

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 Alice's choice of whether to send M_0 or M_1. All 2^{n+1} choices are equally likely. We can imagine that Alice and Bob randomly and uniformly choose the key K; then Alice 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:

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

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'. (Actually in this case she can reconstruct the key K, too.)

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 exploited this weakness to read 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.

Consequently, the one-time pad is not secure if the key is used to encrypt more than one message. Alas, this makes it impractical for almost all practical situations.

1.5 Block Ciphers

In symmetric encryption schemes, Alice and Bob share a random key and use this single key to repeatedly exchange information securely despite the existence of an eavesdropping adversary, Eve. The block cipher is a fundamental building block in implementing a symmetric encryption scheme.

In a block cipher, Alice and Bob share a k-bit random key K, and use this to encrypt an n-bit message into an n-bit ciphertext. In mathematical notation this can be said as follows. There is an encryption function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. 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. Decryption is the reverse of encryption: $D_K(E_K(M)) = M$.

The Advanced Encryption Standard (AES) is an example of a block cipher. It was designed in 1998 by Joan Daemen and Vincent Rijmen, two researchers from Belgium, in response to a competition organized by NIST.

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). It was designed to be extremely fast in both hardware and software. In terms of security, AES has proved to be an impressively strong algorithm. After all these years, the best practical attack known is *exhaustive key search*, where the attacker systematically tries decrypting some ciphertext using every possible key to see which one gives intelligible plaintext. In the case of AES, 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 keysearch 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.

Crypto-theoreticians formalize these security properties of AES by saying: for a randomly chosen key K, E_K "behaves like" a random permutation on the n-bit strings. Formally, we measure the security of the block cipher by performing the following experiment: the adversary, Eve, is given a box which contains either (I) the encryption function E_K with a random key K, or (II) a permutation π on n bits chosen uniformly at random when the box was created. (The type of box given to Eve is randomly selected, but we don't tell Eve which type of box she has been given.) Eve is allowed T steps in which to play with the box. In each step, 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. The "advantage" of the adversary Eve is $\mathrm{Adv}(\mathrm{Eve}) = 2|p - 1/2|$, where p is the probability that Eve guesses correctly which type of box she was given.

Informally, the advantage of Eve measures how effective she is at distinguishing between the block cipher and a truly random permutation. If Eve guesses blindly, she will be correct with probability $p = 1/2$, so her advantage will be 0. If Eve can always deduce which type of box she was given (so she is perfect at guessing), she will be correct with probability $p = 1$, so her advantage will be 1. Intuitively, a small advantage means that Eve is doing only a little bit better than chance; a large advantage means that Eve has gained significant information about which type of box she received.

If Eve's advantage is at most ϵ then we say that the block cipher is (T, ϵ)-secure. For AES, the above discussion says that if Eve wants advantage $\epsilon = 1$, she needs $T \geq 2^{128}$ steps. In general there is a tradeoff between T and ϵ, and so we could expect that for a secure algorithm such as AES, we have $T/\epsilon \geq 2^{128}$ for all successful attacks. In some sense $\log(T/\epsilon)$ is the "effective key length" of the block cipher, i.e., a measure of how much work Eve must exert as equated with brute-forcing a key of the effective length. For instance, if Eve only has enough budget to do 2^{64} steps of computation, then as far as we know, she can only get advantage of about $1/2^{64}$ at distinguishing AES from a random permutation. This is remarkable: it means that Eve is learning almost nothing.

1.6 Symmetric Encryption Schemes

A symmetric encryption scheme allows Alice and Bob to privately exchange a sequence of messages in the presence of an eavesdropper Eve. We will assume that Alice and Bob share a random secret key K. How Alice and Bob managed to share a key without the adversary's knowledge is not going to be our concern here (we will talk about it later, though). The encryption scheme consists of an encryption algorithm $\mathcal{E}$ that takes as input the key K and the plaintext message $M \in \{0,1\}^*$, and outputs the ciphertext. The decryption algorithm $\mathcal{D}$ takes as input the key and the ciphertext and reconstructs the plaintext message M.

In general the encryption algorithm builds upon a block cipher to accomplish two goals. The first goal is that we'd like to encrypt arbitrarily long messages using a fixed-length block cipher. The other is to make sure that if the same message is sent twice, the ciphertext in the two transmissions is not the same. To achieve these goals, 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 is neither randomized nor stateful.

There are four standard ways 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.

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).

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.

Counter Mode: One drawback of CBC mode is that successive blocks must be encrypted sequentially. For high speed applications it is useful to parallelize these computations. This is achieved by encrypting a counter initialized to IV 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 $C_i = Z_i \oplus M_i$.

2 Message Authentication Codes and Digital Signatures

Previously we looked at symmetric-key and asymmetric-key encryption. Strong encryption provides us with the *confidentiality* of communications over an insecure channel. Now we'll delve into cryptographic schemes that provide *integrity* and *authentication*. In particular, the threat we're concerned about is adversaries who send spoofed messages (pretending to be from a legitimate participant) or who modify the contents of a message sent by a legitimate participant. To address these threats, we will introduce cryptographic schemes that enable the recipient to detect spoofing and tampering.

We'll look at schemes in both the symmetric-key and asymmetric-key models. If Alice and Bob share a secret key K, they can use a *Message Authentication Code* (also called a MAC, for short) to detect tampering with their messages. If they don't have a shared key, but Bob knows Alice's public key, Alice can sign her messages with her private key, using a *digital signature* scheme (also known as a public-key signature scheme). In tabular form, four particularly significant types of cryptographic primitives are:

	Symmetric-key	Asymmetric-key
Confidentiality	Symmetric-key encryption (e.g., AES-CBC)	Public-key encryption (e.g., El Gamal, RSA encryption)
Integrity and authentication	MACs (e.g., AES-CBC-MAC)	Digital signatures (e.g., RSA signatures)

2.1 Message Authentication Codes (MACs)

Suppose Alice and Bob share a secret key K, and Alice wants to send a message to Bob over an insecure channel. The message isn't secret, but she wants to prevent attackers from modifying the contents of the message. The idea of a Message Authentication Code (MAC) is to send a keyed checksum of the message along with the message, chosen so that any change to the message will render the checksum invalid.

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. Typically, we might use a 128-bit key K and 128-bit tags. Alice will send the pair of values $\langle M, T \rangle$ to Bob, where she computed the tag $T = F(K, M)$ using the MAC. When Bob receives $\langle M, T \rangle$, Bob will compute $F(K, M)$ and check that it matches the provided tag T. 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.

The algorithm F is chosen so that if the attacker replaces M by some other message M', then the tag will almost certainly[3] no longer be valid: in particular, $F(K, M) \neq F(K, M')$.

[3] 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.

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 a message M' that Alice has never sent.

Modern MACs are designed to be secure against known-plaintext attack. For instance, suppose Georgia the Forger eavesdrops on Alice's communications and observes a number of messages and their corresponding tags: $\langle M_1, T_1\rangle, \langle M_2, T_2\rangle, \dots, \langle M_n, T_n\rangle$, where $T_i = F(K, M_i)$. Then Georgia has no hope of finding some new message M' (such that $M' \notin \{M_1, \dots, M_n\}$) 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 Georgia was able to choose the M_i's.

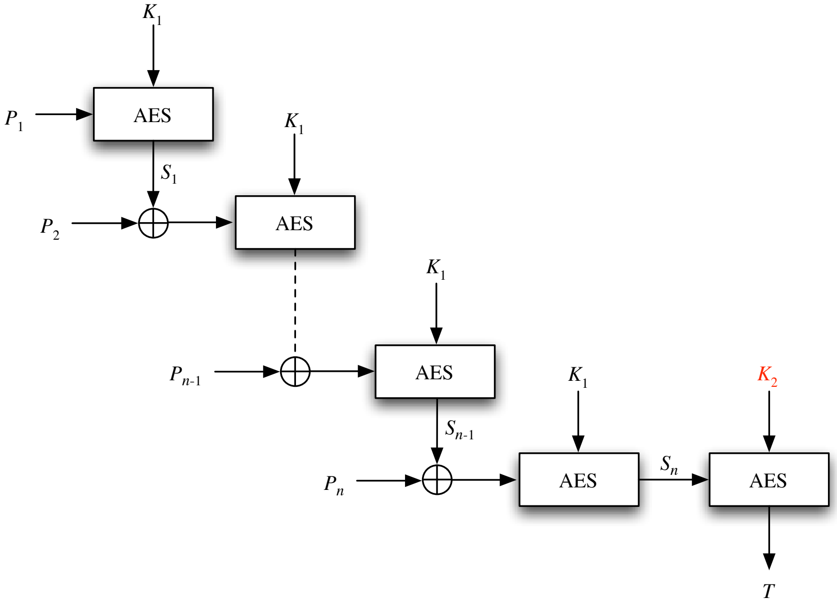
MACs can be used for more than just communication security. For instance, suppose we want to store files on a removable USB flash drive, which we occasionally share with our friends. To protect against tampering with the files on our flash drive, our machine could generate a secret key and store a MAC 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.

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

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

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



This scheme can be proven secure, assuming AES is a secure block cipher. What does it mean to say that a MAC algorithm is secure? Here is a formal definition. 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 $T_i \overset{?}{=} F(K, M_i)$ 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 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.

Something to note: there's no general promise that a MAC algorithm doesn't leak any information about the message M to which it is applied. For some MAC algorithms, it will be clear that they don't leak information because the algorithm directly applies a block cipher (which, if it is any good, indeed has the property that it does not leak information about the plaintext). But for others, it could be that an eavesdropper Eve who observes the value $F(K, M)$ can infer some information about M. So in situations where we want not only *integrity* for M but also *confidentiality*, we need to consider confidentiality for $F(K, M)$, too. One approach is to compute $F(K, \mathcal{E}(M))$ instead; that is, compute the MAC for the ciphertext rather than the plaintext. This value might then leak information about the *ciphertext*, but that's fine; we already assume that Eve can directly view the complete ciphertext.

2.2 Cryptographic Hash Functions

Cryptographic hash functions are another useful primitive. A cryptographic hash function is a deterministic and *unkeyed* function H; $H(M)$ is called the *hash* of the message M. 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, and produces a 256-bit hash value.

A cryptographic hash function can be used to generate a "fingerprint" of a message. Any change to the message, no matter how small, is likely to change many of the bits of the hash value, and there are no detectable patterns to how the output changes. 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.

Cryptographic hash functions have many nice 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 y, it is infeasible to find any input x such that $y = H(x)$. (This property is also known as "**preimage resistant**.")
- **Second preimage resistant:** Given a message x, it is infeasible to find another message x' such that $x' \neq x$ but $H(x) = H(x')$. This property is closely related to *preimage resistant*; 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.
- **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.

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

Note, the third property implies the second property. Cryptographers 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).

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). 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 by the collision-resistance property, it would be reasonable for Alice to conclude that she received a good copy of the legitimate Ubuntu software. 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.

2.3 Digital Signatures

A *digital signature* is the public-key version of a MAC. Suppose Alice wants to send messages to Bob over an insecure channel. In a digital signature scheme, Alice has a public key (also known as a *verification* key) and a private key (also known as a *signing* key) that she generated in advance. Bob needs to know Alice's public key, but Alice will keep her private key absolutely secret from everyone.

Mathematically, a digital signature scheme consists of three algorithms:

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

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

To help us show an example of a working digital signature scheme in more concrete terms, we will now take a tangent and introduce the concept of a *trapdoor one-way function*.

2.4 Trapdoor One-way Functions

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.

A trapdoor one-way function is associated with a public key U and a private key K. Given the public key U, it is computationally easy to compute F, but hard to compute F^{-1}. In other words, given x and U, it is easy to compute $y = F(x)$; but given y and U, it is (very) hard to find x such that $y = F(x)$, i.e., it is hard to compute $F^{-1}(y)$.

We can view the private key as "unlocking" the trapdoor. Given the private key K, it becomes easy to compute F^{-1}, and of course it remains easy to compute F. Put another way, given y and K, it becomes computationally easy to find x such that $y = F(x)$, i.e., it is easy to compute $F^{-1}(y)$.

The RSA signature scheme specifies a particular method for building a trapdoor one-way function.

2.5 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 § 2.2. 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.

2.6 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 $n = pq$.[4] 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.

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.

2.7 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

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

[4] 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.

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 $2 \bmod 3$. 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 $\textsc{Verify}$ 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!

2.8 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 $\textsc{KeyGen}$ 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 $\textsc{Verify}_U(M, S) =$ 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.

3 Asymmetric cryptography

Previously we saw symmetric-key cryptography, where Alice and Bob share a secret key K. However, symmetric-key cryptography can be inconvenient to use, because it requires Alice and Bob to get together in advance to establish the key somehow. *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.

Public-key cryptography is a remarkable thing. Consider a function that, for a given public key, maps the message to the corresponding ciphertext. In a good public-key cryptosystem, this function must be easy to compute, and yet **very hard to invert**. In other words, it must form a *one-way function*: 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 $f(x) = y$. We need 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. It is by no means obvious that it should be possible to accomplish this, but it turns out it is, as we'll soon discuss.

The known methods for public-key cryptography tend to rely heavily upon number theory, so we begin with a brief number theory refresher, and then develop an encryption algorithm based on public-key cryptography.

3.1 Algorithms for modular arithmetic

3.1.1 Simple modular arithmetic

Two n-bit integers can be added, multiplied, or divided by mimicking the usual manual techniques taught in elementary school. For addition, the schoolkid's algorithm takes a constant amount of time to produce each bit of the answer, since each such step only requires dealing with three bits—two input bits and a carry—and anything involving a constant number of bits takes $O(1)$ time. The overall time to add two n-bit integers is therefore $O(n)$, or linear in the bitlength of the integers. Multiplication and division take $O(n^2)$ time, i.e., quadratic time. Also recall that n, the number of bits it takes to represent an integer a in binary, satisfies $n \leq \lceil \log_2 a \rceil$.

Recall that $a \bmod p$ is the remainder of the number a modulo p. For instance, $37 \bmod 10 = 7$.

Modular arithmetic can be implemented naturally using addition, multiplication, and division algorithms. To compute $a \bmod p$, simply return the remainder after dividing a by p. By reducing all inputs and answers modulo p, modular addition, subtraction, and multiplication are easily performed. These operations can all be performed in $O(\log^2 p)$ time, since the numbers involved never grow beyond p and therefore have size at most $\lceil \log_2 p \rceil$ bits.

3.1.2 Modular exponentiation

Modular exponentiation is the task of computing $a^b \bmod p$, given a, b, and p.

A naive algorithm for modular exponentiation is to repeatedly multiply by a modulo p, generating the sequence of intermediate products $a^2 \bmod p$, $a^3 \bmod p$, $a^4 \bmod p$, $\ldots$, $a^b \bmod p$. Each intermediate product can be computed from the prior one with a single modular multiplication, which takes $O(\log^2 p)$ time to compute, so the overall running time to compute $a^b \bmod p$ products is $O(b \log^2 p)$. This is linear in b and thus exponential in the size (bitlength) of b—really slow when b is large.

There is a better way to do it. The key to an efficient algorithm is to notice that the exponent of a number a^j can be doubled quickly, by multiplying the number by itself. Starting with a and squaring repeatedly, we get the powers $a^1, a^2, a^4, a^8, \ldots, a^{2^{\lfloor \log_2 b \rfloor}}$, all modulo p. Each takes just $O(\log^2 p)$ time to compute, and they are all we need to determine $a^b \bmod p$: we just multiply together an appropriate subset of them, those corresponding to ones in the binary representation of b. For instance,

$$a^{25} \quad = \quad a^{11001_2} \quad = \quad a^{10000_2} \cdot a^{1000_2} \cdot a^{1_2} \quad = \quad a^{16} \cdot a^8 \cdot a^1.$$

This repeated squaring algorithm is shown in Algorithm 1. The overall running time is $O(\log^2 p \ \log b)$. When p and b are n-bit integers, the running time is *cubic* in the input size. This is efficient enough that we can easily perform modular exponentiation on numbers that are thousands of bits long.

Algorithm 1 ModExp1(a, b, p): modular exponentiation using repeated squaring.

```
Require: a modulus $p$, a positive integer $a < p$, and a positive exponent $b$
Ensure: the return value is $a^b \bmod p$
Let $b_{n-1} \cdots b_1 b_0$ be the binary form of $b$, where $n = \lceil \log_2 b \rceil$.
// Compute the powers $t_i = a^{2^i} \bmod s$.
$t_0 := a \bmod p$
for $i := 1$ to $n - 1$ do
    $t_i := t_{i-1}^2 \bmod p$
end for
// Multiply together a subset of these powers.
$r := 1$
for $i := 0$ to $n - 1$ do
    if $b_i = 1$ then
        $r := r \times t_i \bmod p$
    end if
end for
return $r$
```

3.1.3 Selecting Large Primes

In the material to follow, we will be working with very large primes: primes that are thousands of bits long. Let's look at how to generate a random n-bit prime.

It turns out that it's easy to test whether a given number is prime. Fermat's Little Theorem forms the basis of a kind of litmus test that helps decide whether a number is prime or not: to test if a number M is prime, we select an $a \bmod M$ at random and compute $a^{M-1} \bmod M$. If the result is not 1, then by Fermat's theorem it follows that M is not a prime. If on the other hand the result is 1, then this provides evidence that M is indeed prime. With a little more work this forms the basis of an efficient probabilistic algorithm for testing primality.

For the purpose of selecting a random large prime (several thousand bits long), it suffices to pick a random number of that length, test it for primality, and repeat until we find a prime of the desired length. The prime number theorem tell us that among the n-bit numbers, roughly a $\frac{1.44}{n}$ fraction of them are prime. So after $O(n)$ iterations of this procedure we expect to find a prime of the desired length.

3.2 Diffie-Hellman key exchange

Now we're ready to see our first public-key algorithm. Suppose Alice and Bob are on opposite sides of a crowded room. They can shout to each other, but everyone else in the room will overhear them. They haven't thought ahead to exchange a secret key in advance. How can they hold a private conversation?

It turns out there is a clever way to do it, first discovered by Whit Diffie and Marti Hellman in the 1970s. In high-level terms, the Diffie-Hellman key exchange works like this.

Alice and Bob first do some work to establish a few parameters. They somehow agree on a large prime p. For instance, Alice could pick p randomly and then announce it so Bob learns p. The prime p does not need to be secret; it just needs to be very large. Also, Alice and Bob somehow agree on a number g in the range $1 < g < p - 1$. The values p and g are parameters of the scheme that could be hardcoded or identified in some standard; they 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

$$S = B^a \bmod p.$$

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

$$S = A^b \bmod p.$$

Note that

$$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \pmod{p},$$

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 $A = g^a \bmod p$, $B = g^b \bmod p$, g, and p. (If there were an efficient algorithm to recover S from A, B, p, g, then this scheme would be insecure, because an eavesdropper could simply apply that algorithm to what she overhears.) 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.

The security of Diffie-Hellman key exchange relies upon the fact that the following function is one-way: $f(x) = g^x \bmod p$. In particular, it is easy to compute $f(\cdot)$ (that's just modular exponentiation), but there is no known algorithm for computing $f^{-1}(\cdot)$ in any reasonable amount of time.

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 $2 \dots p-2$. Both are arbitrary, fixed, and public.
- **Key agreement protocol:** Alice randomly picks a in the range $0 \dots p-2$ and sends $A = g^a \bmod p$ to Bob. Bob randomly picks b in the range $0 \dots p-2$ 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 secret key K, yet as far as we know no eavesdropper can recover K in any reasonable amount of time.

3.3 El Gamal encryption

The Diffie-Hellman protocol doesn't quite deliver public-key encryption directly. It allows Alice and Bob to agree on a key that they then use with symmetric cryptography. An interactive protocol for agreeing on a secret key (like Diffie-Hellman) is somewhat different from a non-interactive algorithm for encrypting messages.

There are also public-key cryptography algorithms that can directly support encryption, if desired. One of these is RSA, which you encountered in CS 70 (and did/will in CS 161 lecture). To cement the idea, here's another scheme for doing so that's actually a slight twist on Diffie-Hellman.

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 system parameters are a large prime p and a value g satisfying $1 < g < p-1$, as in Diffie-Hellman. 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 $1 \dots p-1$) 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 $0 \dots p-2$), 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 $0 \dots p-1$.

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 $S = m \times K \bmod p$).

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 http://crypto.stanford.edu/~dabo/abstracts/DDH.htmlat 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:** $D_b(R, S) = R^{-b} \times S \bmod p$.

3.4 Caveat: Don't try this at home!

A brief warning is in order here. You've now seen the conceptual basis underlying public-key algorithms that are widely used in practice. However, if you should need a public-key encryption algorithm, *don't implement your own based on the description here.* The discussion has omitted some nitty-gritty implementation details that are not all that relevant at the conceptual level, but are essential for robust security. Instead of implementing these algorithms yourself, you should just use a well-tested cryptographic library or protocol, such as TLS or PGP.

3.5 What's the catch?

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.

4 Passwords

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

4.1 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.

4.2 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.

4.3 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).

4.4 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.

4.5 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.

4.6 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.

4.6.1 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 compute 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.

4.6.2 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.

4.7 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.

4.8 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.

4.9 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.

5 Key Management

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?

5.1 Man-in-the-middle Attacks

Suppose Alice wants to communicate security 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.

5.2 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. 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.

5.3 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.

5.4 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.

5.5 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.

5.6 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.

5.7 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.

5.8 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.[5] 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*[6] 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?

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

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

- 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.