Class Notes

These are notes taken from the lectures for CS 255: Introduction to Cryptography. They are mostly taken from the lecture recordings for the class, and I try to update them within a day of each lecture, though relevant drawings and diagrams might be added later. All diagrams are either drawn by me or are screenshots from the lecture recording or relevant slide deck. If you find any typos, errors, or confusingly worded section of the notes, please let me know! You can contact me at `atalati [at] stanford.edu`.

Table of Contents

Lecture 1: Perfect Secrecy and the One-Time Pad

What is this course about?

- the use of cryptography
- cryptography is everywhere!

Goal: to learn how cryptography works and how to use it correctly

Two Steps for protocols like SSH and TLS:

1. session-setup: public-key cryptography (ex. Elbamal, RSA) and certificate
2. use shared key to encrypt traffic (using a symmetric cipher) for **privacy** and **data integrity**

The beginning of the course will be part 2, the next part of the course will be session setup, and the last part will be some advanced forms of cryptography.

Thing to Remember: in cryptography, no security by obscurity

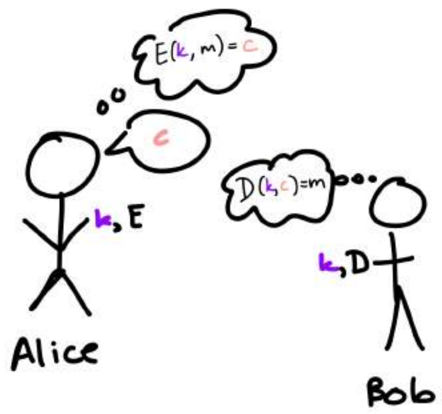
A cryptographic system should be secure even when the entire code is open source - the only secret we are allowed to have is a short secret key. (on the scale of 16 bytes)

Course Organization:

1. using a shared key for confidentiality and data integrity
2. session setup using public key encryption and digital signatures
3. protocols: session setup, zero knowledge proofs, etc.

If you want to read about the history of cryptography, read the famous book *The Codebreakers* by David Kahn.

Shared Cipher: Alice and Bob are trying to communicate. They both have a shared (secret) key K. Alice has a (public) encryption algorithm E and Bob has a (public) decryption algorithm D such that for any message m, Alice can generate a ciphertext c by computing $E(K, m)$. She sends the ciphertext to Bob, and Bob generates the message m by computing $D(K, c)$.



Definition 1. A **cipher** defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ (keyspace, message space, ciphertext space) is a pair of "efficient" algorithms (E, D):

$$E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}, \qquad D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$$

such that for every message $m \in \mathcal{M}$ and every key $K \in \mathcal{K}$,

$$D(K, E(K, m)) = m.$$

Note: the encryption algorithm can be randomized (the same message, encrypted twice, can produce different ciphertexts) but the decryption algorithm must be deterministic

Example 2. The oldest cipher is a **substitution cipher**, where the key is a certain permutation of the letters of the alphabet, and the encryption algorithm maps each letter to its position in the permutation.

e.g. For key $k =$ `zmbfg ... ha`, our encryption function $E(k, m)$ would take the message string, and replace `a` with `z`, `b` with `m`, and so on, and our decryption function $D(k, c)$ would take the ciphertext, and replace `z` with `a`, `m` with `b`, and so on.

The Caeser cipher was a "shift by 3" but this was not secure because there is no key. Any shift is insecure, because the keyspace $\mathcal{K}$ has size 26, so it is trivial to try all keys until we reveal the message.

To break the **substitution cipher**:

1. use the frequency of English letters
2. use the frequency of digrams (pairs of letters)
3. use the frequency of trigrams
4. and so on ...

This is a **ciphertext-only attack** because we have no information other than the ciphertext, and we are still able to reveal the message.

Definition 3. The **one-time pad** (invented by Vernam, 1917 or possibly by a banker in Oakland 50 years before) is essentially a substitution cipher where the permutation changes every bit:

$$\mathcal{M} = \mathcal{C} = \{0, 1\}^n, \qquad \mathcal{K} = \{0, 1\}^n$$

so the secret key is a random bit string as long as the message.

Our encryption algorithm is

$$c = E(K, m) = K \oplus m$$

and our decryption algorithm is

$$m = D(K, c) = K \oplus c.$$

This is a very fast encryption algorithm, but the problem with the one-time pad is the key is as long as the message. (If Alice can send a length-n string to Bob securely, she might as well just send the message.)

What is a secure cipher?

Definition 4 (Shannon, 1949)**.** The idea of **information-theoretic security** is the ciphertext should reveal no information about the plaintext message.

A cipher (E, D) over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ has **perfect secrecy** if for all $m_0, m_1 \in \mathcal{M}$ such that m_0 and m_1 are both the same length, and for all $c \in \mathcal{C}$,

$$\Pr[E(K, m_0) = c] = \Pr[E(K, m_1) = c],$$

where K is selected uniformly at random from $\mathcal{K}$.

In words, if an attacker receives c, the attacker cannot tell whether it came from m_0 or m_1 (for all c, m_0, m_1).

This implies a ciphertext-only attack is impossible, because the ciphertext could have come from any message (of a given length).

Theorem 5. The one-time pad has perfect secrecy.

Proof. For any $m \in \mathcal{M}$, $c \in \mathcal{C}$, the probability $E(K, m) = c$ is just the number of keys that map m to c, divided by the total number of keys. We know that the keyspace is $\{0, 1\}^n$, so the number of possible keys is 2^n. Moreover, the key maps m to c means $K \oplus m = c$, which means $K = m \oplus c$; there is only one such key. Thus, for all m, c,

$$\Pr[E(K, m) = c] = \frac{1}{2^n}.$$

Since this is the same for all m and all c, this means the one-time pad has perfect secrecy. □

Theorem 6 (Bad News Theorem)**.** Every cipher that has perfect secrecy must have $|\mathcal{K}| = |\mathcal{M}|$.

Proof. (Proof is left as an exercise; hint is that for a fixed $c \in \mathcal{C}$, for every $m \in \mathcal{M}$, there must be at least one key $K \in \mathcal{K}$ such that $E(K, m) = c$) □

Lecture 2: Pseudo-Random Generators and Stream Ciphers

Definition 7. A **pseudo-random generator** is a function $G : \{0,1\}^s \to \{0,1\}^n$, where s is significantly smaller than n, which is "efficiently" computable by a deterministic algorithm.

Definition 8. A **stream cipher** makes the one-time pad practical by replacing a random key by a "pseudorandom" key.

The stream cipher (E, D) is defined over $\mathcal{C} = \mathcal{M} = \{0,1\}^n$ and $\mathcal{K} = \{0,1\}^s$, where

$$E(K, m) = m \oplus G(K)$$

and

$$D(K, c) = c \oplus G(K).$$

Since $|\mathcal{K}| << |\mathcal{M}|$, the stream cipher is not perfectly secure. Thus, we need a new definition of security, which will depend on the pseudo-random generator.

Specifically, we will see that the pseudo-random generator must be unpredictable.

Definition 9. A pseudo-random generator $G(K)$ is **unpredictable** if no efficient algorithm $\mathcal{A}$ can compute the i^{th} bit of $G(K)$ if they just know the first i bits of $G(K)$ (and not they key).
More precisely: For all efficient algorithms $\mathcal{A}$, we want the value

$$\left|\Pr\left[\mathcal{A}\left(G(K)[0,\ldots,i-1]\right) = G(K)[i]\right] - \frac{1}{2}\right|$$

to be negligible; the adversary shouldn't do much better than guessing at random.

(In practical terms, we can consider "negligible" to be at most 2^{-80}.)

A predictable key implies that the stream cipher is insecure (knowing part of the message means that we can reveal part of the key, and then we can use this to reveal the rest of the key by predicting the rest of $G(K)$).

Weak Generators (do not use for crypto):

- glibc `rand()`
- Java `Math.random()`
- Microsoft VisualBasic `RND()`

These are useful for generating randomness in statistical algorithms or physics simulators, but they are completely predictable. Instead, use cryptographic libraries such as OpenSSL, and properly seed the random generator.

Definition 10. A pseudo-random generator $G : S \to R$ is **secure** if for all efficient adversaries $\mathcal{A}$, its advantage

$$\text{Adv}(\mathcal{A}, G) = \left|\Pr[\mathcal{A}(G(K)) = 1] - \Pr[\mathcal{A}(r) = 1]\right|$$

is negligible, where K is selected uniformly at random from S, r is selected uniformly at random from R, and $\mathcal{A}(x)$ is any efficient function of strings in R.

(Intuitively, we are looking at adversaries that are trying to distinguish the output of G from a randomly selected string, such as one that outputs 1 only if it thinks the string was generated truly at random.)

Thus, if G is secure, we can treat $G(K)$ as if it was selected uniformly at random from R.

Lemma 11. If the pseudo-random generator is secure, that implies it is unpredictable.

Proof. The proof is left as an exercise, but the idea is that if a pseudo-random generator is predictable, we can construct an adversary that tries to predict part of the string from a prefix, and outputs 1 only if it does not succeed. $\square$

(The converse of this is also true, but proving it is beyond the scope of this class.)

A pseudo-random generator being secure implies that the derived stream cipher is secure.

Definition 12. Two distributions P_1 and P_2 are **indistinguishable** if for all efficient adversaries $\mathcal{A}$, its advantage

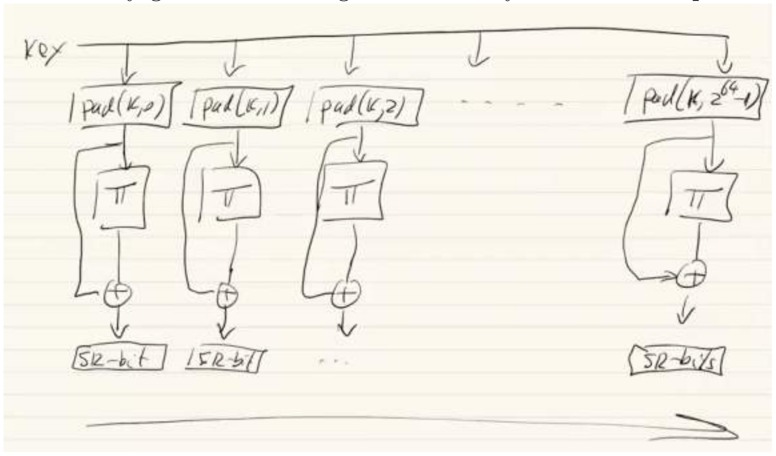
$$\text{Adv}(\mathcal{A}, P_1, P_2) = \left|\Pr[\mathcal{A}(p_1) = 1] - \Pr[\mathcal{A}(p_2) = 1]\right|$$

is negligible, where p_1 is randomly sampled from P_1, p_2 is randomly sampled from P_2, and $\mathcal{A}(x)$ is any efficient function of strings in R.

Real-World Pseudo-Random Generators

1. RC4:
   This is now broken, but is used in 802.11b WEP. It's broken because the first 256 bytes of output are biased, so after large samples statistical analysis can be used to reveal the message.

2. ChaCha20:
   This is very good and is designed to be very fast on 64-bit processors. Works as follows:



   so that the input key is 256 bits and the output is $2^{64} \times 32$ bytes.

3. CSS pseudo-random generator:
   This was used for DVD encryption and is broken (will be an example on a homework problem).

Attack on One-Time Pad and Stream Ciphers:

Two-Time Pad
If the same key is used for two messages:

$$c_1 = m_1 \oplus K$$
$$c_2 = m_2 \oplus K,$$

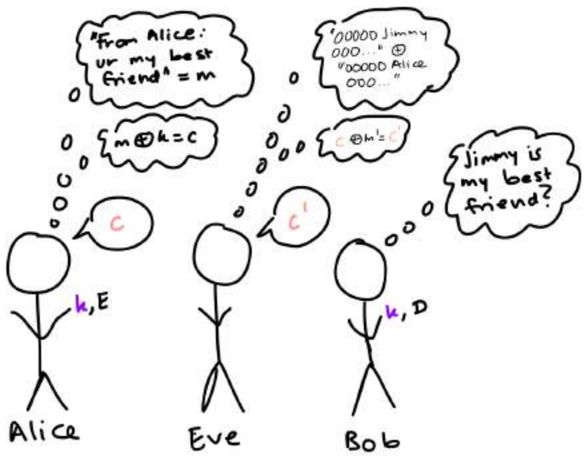
then we can see that $c_1 \oplus c_2 = m_1 \oplus m_2$, and written text has enough patterns in it that this is enough to recover m_1 and m_2.

For a historical example, see Project Venona, which was a project to break encrypted ciphertexts in the Cold War when secret keys were being reused for the one-time pads in encrypted messages.

A stream cipher key should only be used once!

No Integrity

If an adversary can intercept a message, it can xor the ciphertext with whatever changes it wants to make, and the receiver won't know the difference.


"From Alice: ur my best friend" = m
m ⊕ k = c
c
k, E
"00000 Jimmy 000..." ⊕ "00000 Alice 000..."
c ⊕ m' = c'
c'
Jimmy is my best friend?
k, D
Alice
Eve
Bob

Note that this can be done without the adversary knowing the actual message, if, for example, they know where the "from" location in an email message is.

Lecture 3: Block Ciphers

Modern Stream Ciphers have a way of allowing you to use the same key more than once, using a concept called the **nonce**. Specifically, the pseudo-random generator is a function of both the key and a short nonce, where the nonce is a public value. Then, the stream cipher is secure as long as no key, nonce *pair* is repeated (rather than when no key is repeated). This is surprising because by adding a changing known value to the function, the key is not compromised even when using the same key multiple times.

A Short Aside: How do we generate the key for the pseudo-random generator?
(a.k.a. how to generate crypto-randomness on a computer) Every crypto-random generator has two functions:

- `add_entropy(rand)`: called by the OS whenever there is new randomness to be added - ex. could add the current timestamp whenever a keyboard interrupt occurs
- `get_random()`: called by the developer to obtain a random string

Every time `add_entropy` is called, the new entropy is xored with the previous state, until the state of the pseudo-random generator is close enough to uniformly random. This means that `get_random` will not return a properly random value until enough entropy has been added to the state. Moreover, since entropy keeps being added, if an attacker obtains the state of the crypto-random generator at one point, this will no longer reveal information about the generator once more entropy has been added.

Block Ciphers

Definition 13. A **block cipher** is a pair of efficient algorithms (E, D) defined over $(\mathcal{K}, \mathcal{X})$ such that E and D both map n-bit strings to n-bit strings. Moreover, for every key $K \in \mathcal{K}$ and every $x \in \mathcal{X}$, $D(K, E(K, x)) = x$.

Two current examples of block ciphers:

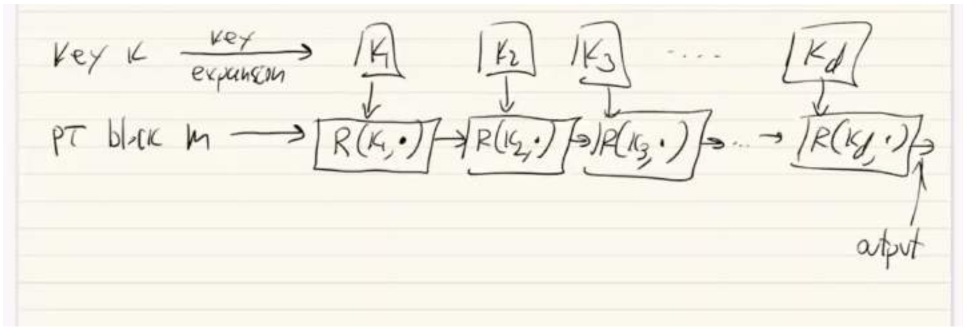
- 3DES: n=64-bit blocks, key size=168 bits
- AES: n=128 bit blocks, key size=128, 192, or 256 bits

A secure block cipher will be formally defined next lecture, but it means $E(K, x)$ should look like a truly random one-to-one function from $\mathcal{X}$ to $\mathcal{X}$.

Advanced Encryption Standard (AES) (invented in 2000)

The AES Pledge: I promise that once I see how simple AES is, I will not implement it in production code even though it will be really fun. This pledge will remain in effect until I learn all about side-channel attacks and countermeasures to the point where I lose all interest in implementing AES myself.

Most block ciphers are built by iteration:



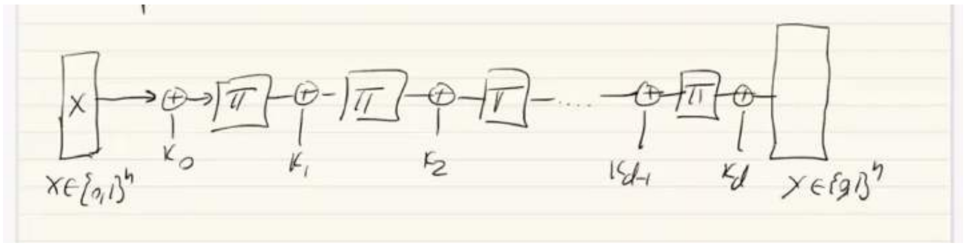
The function R is a deterministic function such that R^{-1} is easy to compute, so it is decrypted by computing R^{-1} of the ciphertext on each of the keys in reverse order.

Number of Rounds Needed to Make Each Cipher Secure:

- 3DES: $d = 48$ rounds
- AES128: $d = 10$ rounds
- AES192: $d = 12$ rounds
- AES256 $d = 14$ rounds

Abstract AES: This is a type of iterated Evan-Mansour cipher.

We have a permutation function $\pi : \{0,1\}^n \to \{0,1\}^n$ which is fixed and publicly known. Then our encryption algorithm $E(K, x)$ works as follows: First, perform the key-expansion to generate $K_1, \ldots, K_d \in \{0,1\}^n$. Then, apply the permutation and xor with the keys as follows:



with the minor detail that the π in the last round is slightly different from the first $d-1$ rounds, just for convenience.

This is a vague statement of a theorem that is beyond the scope of the class, but provides some motivation for the iterated Evan-Mansour cipher:

"Theorem" 14. If π is a random one-to-one function, then $E(K, x)$ "looks like" a random one-to-one function on $\mathcal{X}$, provided n and d are sufficiently large.

Lecture 4: Pseudorandom Functions

Definition 15. A **pseudo-random function** (PRF) defined over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$ is a function

$$F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$$

such that there exists an efficient algorithm to evaluate $F(K, x)$.

Definition 16. A **pseudo-random permutation** (PRP) defined over $(\mathcal{K}, \mathcal{X})$ is a function

$$E : \mathcal{K} \times \mathcal{X} \to \mathcal{X}$$

such that:

- there exists an efficient algorithm to evaluate $E(K, x)$
- the map $E(K, \cdot)$ is one-to-one
- there exists an efficient inversion algorithm $D(K, x)$

(This means the same thing as a block cipher, so we will use the terms interchangeably.)

Any pseudo-random permutation is also a pseudo-random function (a specific type of pseudo-random function that is also one-to-one and invertible).

Definition 17. A pseudo-random function is **secure** if a random function in Funs$[X, Y]$ (the set of all functions from X to Y) is indistinguishable from a random function in the set $\{F(K, \cdot)$ where $K \in \mathcal{K}\}$.

Thus, we define the following experiment to determine whether a pseudo-random function is secure: The challenger is running either experiment 0, where it has selected a key K randomly from $\mathcal{K}$ and will pass each query into the function $f(x) = F(K, x)$, or it is running experiment 1, where it has selected a random function from Funs$[X, Y]$ and will pass each query into $f(x)$ where $f(x)$ is this random function.

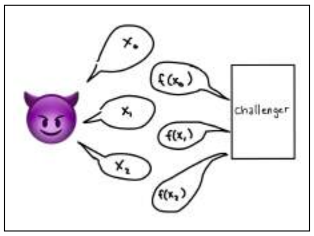
Then, the adversary can query the challenger as many times as it likes, sending the challenger some x_i and receiving $f(x_i)$. Once it is done querying the challenger, it outputs either 0 or 1 depending on which experiment the adversary thinks it is in.

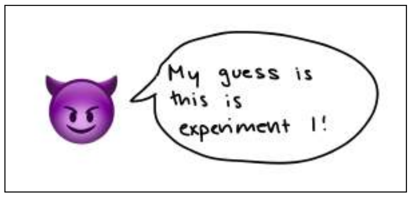
Let EXP(b) be the output the adversary produces in experiment b. Then, a pseudo-random function is **secure** if for all efficient adversaries $\mathcal{A}$ (note that being efficient puts an upper bound on the number of queries an adversary can ask),

$$\mathrm{Adv}_{\mathrm{PRF}}(\mathcal{A}, F) = \left|\Pr[\mathrm{EXP}(0) = 1] - \Pr[\mathrm{EXP}(1) = 1]\right|$$

is negligible.

Challenger
This is experiment 0, so I will pick a random key K, and f(x) = F(K,x)





Example 18. The pseudo-random function $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ defined by $F(K,x) = K \oplus x$ is insecure; we can construct an adversary that passes $x_0 \neq x_1$ to the challenger and outputs 0 if $f(x_0) \oplus f(x_1) = x_0 \oplus x_1$ and 1 otherwise. We can compute the advantage of this to see that adversary gets advantage $1 - 1/2^n$; this is close to one and therefore not negligible.

The definition for **secure pseudo-random permutation** is the same except the adversary is trying to distinguish between $E(K,\cdot)$ and a random function chosen from Perms$[X]$ (the set of one-to-one functions from X to itself).

Example 19. Ciphers such as 3DES and AES are examples of secure permutations; we believe that no adversary that runs in time less than 2^{80} can get advantage more than 2^{-40} on AES.

Lemma 20. Let E be a pseudo-random permutation over $(\mathcal{K}, \mathcal{X})$. Then, for any q-query adversary $\mathcal{A}$,

$$\left|\mathrm{Adv}_{\mathrm{PRP}}(\mathcal{A}, E) - \mathrm{Adv}_{\mathrm{PRF}}(\mathcal{A}, E)\right| < q^2/2|\mathcal{X}|\,.$$

Thus, for large enough $\mathcal{X}$, a secure pseudo-random permutation is also a secure pseudo-random function.

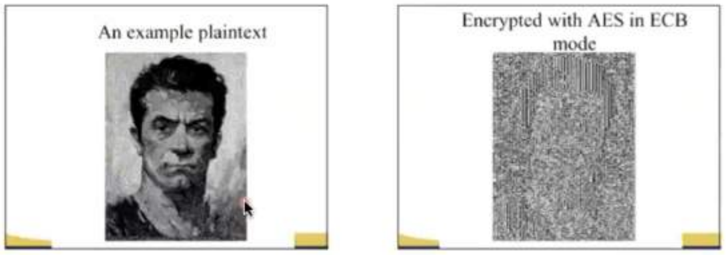
Building a Secure Encryption using PRP and PRF

Security is defined using two questions:

- What power does the adversary have? (only asking one query, asking multiple queries, etc)
- What is the adversary trying to do? (fully decrypt a ciphertext, learn information about the plaintext, etc)

Incorrect Use of a PRP:

You might intuitively think to break a long message into blocks, and then encrypt each block using the pseudo-random permutation to get a string of resulting ciphertexts. This is called the Electronic Code Book (ECB) method. But this is a bad idea because it reveals a lot of information about the plaintext - any two blocks that are identical will have the same resulting ciphertext.



(this picture was marked as courtesy B. Preneel in the slides)

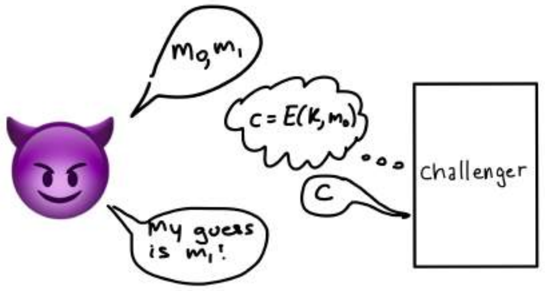
Definition 21. We say that a cipher with a one-time key is **semantically secure** if the ciphertext does not reveal any information about the plaintext after a single query.

Specifically, we define the adversarial game as follows: the adversary sends the challenger two messages, m_0 and m_1, of equal length. The challenger picks at random whether they want to encrypt m_0 or m_1 and sends to the adversary either $E(K, m_0)$ or $E(K, m_1)$. Then, the adversary guesses which message was encrypted.

Let W_b be the output the adversary produces in experiment b. Then, a cipher is **semantically secure** if for all efficient adversaries $\mathcal{A}$,

$$\text{Adv}_{\text{SS}}(\mathcal{A}, E) = \left|\Pr[W_0 = 1] - \Pr[W_1 = 1]\right|$$

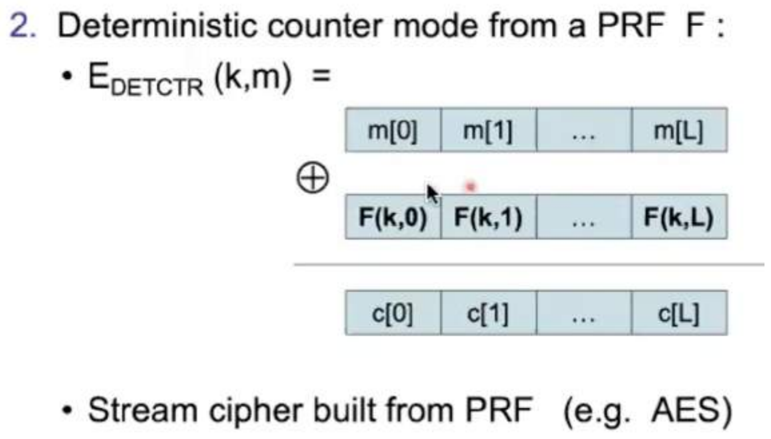
is negligible.



Example 22. If we have an efficient adversary $\mathcal{A}$ that can learn the least significant bit of the plaintext from the ciphertext, then E is not semantically secure.

In the game, the adversary picks m_0 and m_1 such that $|m_0| = |m_1|$ and the least significant bit of m_0 is 0 but the least significant bit of m_1 is 1. Then, if it passes m_0 and m_1 to the challenger, and receives the ciphertext c, then the adversary figures out what the least siginficant bit of the plaintext was. If it was 0, it outputs 0, otherwise it outputs 1. This gives the adversary an advantage of 1.

Clearly, the one-time pad is semantically secure. We will look at some ways of generating semantically secure ciphers from a pseudo-random function.



2. Deterministic counter mode from a PRF F :
- E_{DETCTR} (k,m) =

- Stream cipher built from PRF (e.g. AES)

Theorem 23. For any $L > 0$ and any secure pseudo-random function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{X}$, E_{DETCTR} is a semantically secure cipher from $\mathcal{K} \times \mathcal{X}^L$ to $\mathcal{X}^L$.

Proof. Specifically, we can show that for any adversary $\mathcal{A}$ attacking E_{DETCTR}, there is an adversary $\mathcal{B}$ attacking F with exactly half the advantage. Then, since the maximum advantage $\text{Adv}[\mathcal{B}, F]$ is negligible, the maximum advantage $\text{Adv}[\mathcal{A}, E_{\text{DETCTR}}]$ must also be negligible.

(Refer to the textbook for the construction of $\mathcal{B}$) $\square$

Definition 24. A cipher is **CPA secure** (or secure against chosen-plaintext attack) if the ciphertext does not reveal information about the plaintext even after many queries using the same key.

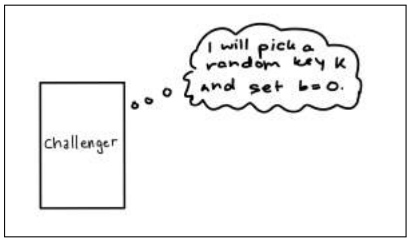
Thus, we define the following game to determine whether a cipher is CPA secure:
The challenger first selects either $b = 0$ or $b = 1$ and selects a key K at random.

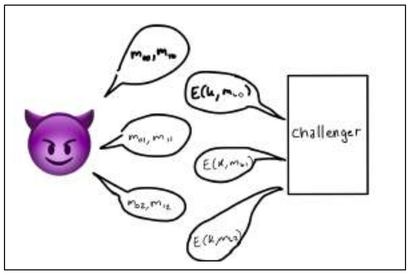
Then, the adversary can query the challenger as many times as it likes, where each query is of the form of the adversary sending the challenger some pair of messages of the same length, m_0 and m_1, and receiving $E(K, m_b)$. Once it is done querying the challenger, it outputs either 0 or 1 depending on which message it thinks the adversary is encrypting.

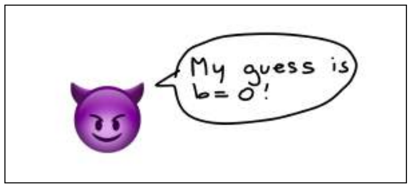
Let W_b be the output the adversary produces when the challenger selects b. Then, a cipher is **CPA secure** if for all efficient adversaries $\mathcal{A}$ (note that being efficient puts an upper bound on the number of queries an adversary can ask),

$$\text{Adv}_{\text{CPA}}(\mathcal{A}, E) = \left|\Pr[W_0 = 1] - \Pr[W_1 = 1]\right|$$

is negligible.





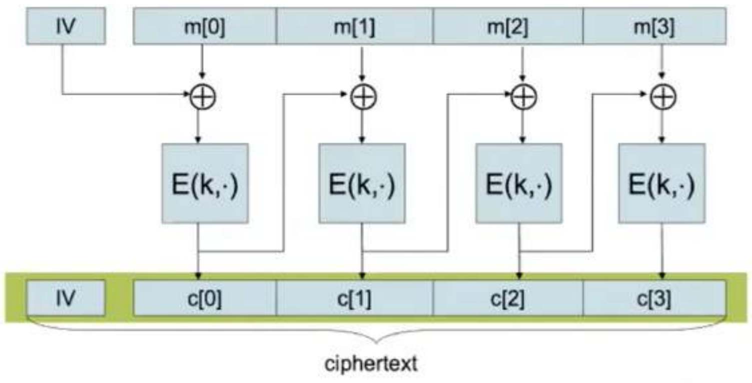


Note that if $E(K, m)$ always produces the same c for the same m, then it is not secure: the adversary can first pass in m_0, m_0 to get $c_0 = E(K, m_0)$ and then pass in m_0, m_1 to get $c_b = E(K, m_b)$ and then check whether $c_b = c_0$.

We fix this by adding a nonce to the encryptor - this is a value that changes from message to message, so that $E(K, m, n)$ changes each time because the pair (K, n) never repeats.

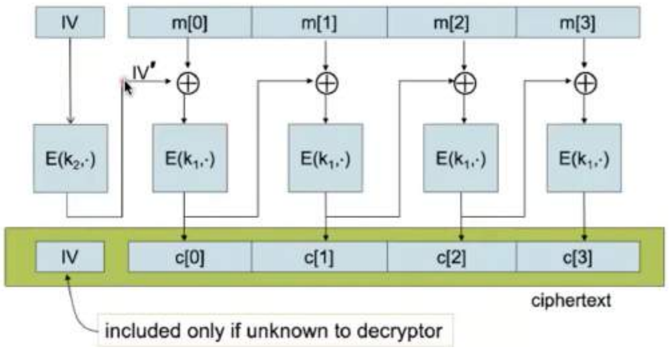
Two common nonces are either a nonce chosen randomly from the nonce space, or a counter that increases at each message sent.

One way of using the nonce is **cipher block chaining**, which works as follows (where IV is the nonce):



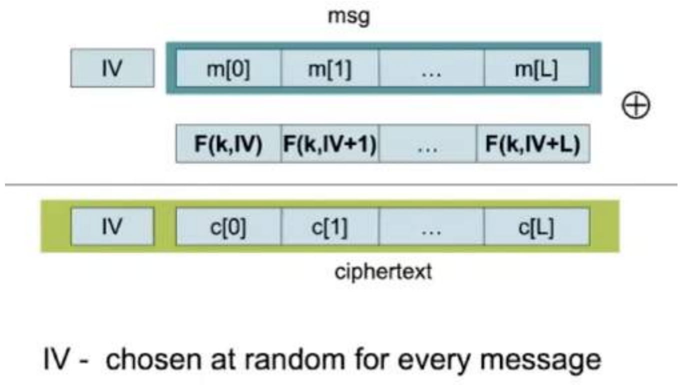
However, this requires a random IV (there is a homework problem on showing that if the nonce is predictable this is not secure) and because it is a sequential process, we can't use parallelization to speed it up.

To fix the first problem, we can take a known (unique) nonce and pass it through E to get a pseudorandom nonce, as follows:



A side note: we may need to pad the last message to fill the final block. IN TLS 1, they did this by padding with "$nnnn...n$" where $n+1$ was the number of bytes in the pad, so the decryptor could read the last byte and know how many bytes of padding to remove. If the message is a multiple of the block size, we need to add a dummy block so that the decryptor doesn't think the end of the message is part of the pad.

Instead of cipher block chaining (which is not that good and mainly only used by banks now), we should use **randomized counter mode**.



(This is parallelizable, unlike CBC ☺)

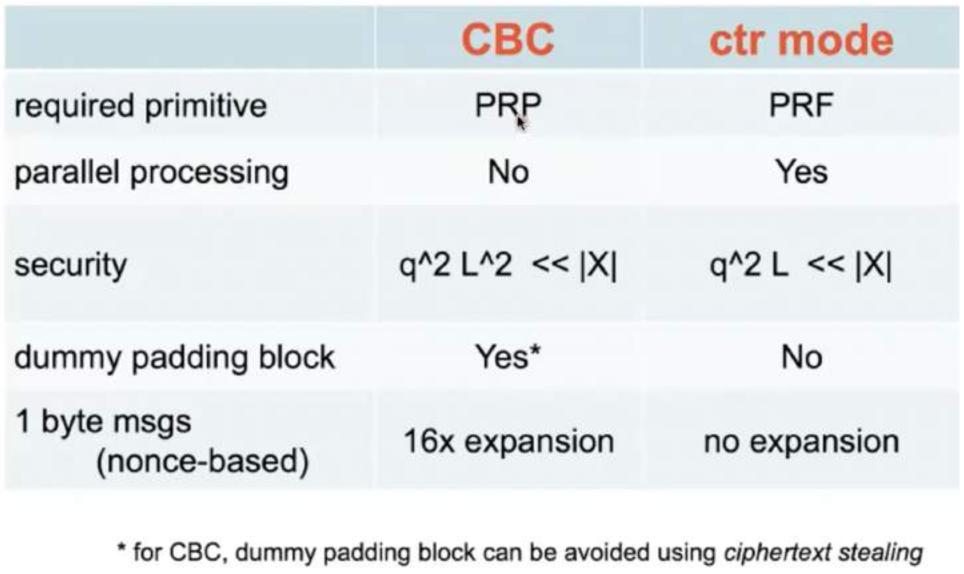
Lecture 5: Data Integrity and MACs

For the randomized counter mode, to make sure that the $F(k, x)$ pair is never repeated, we set the IV to be a 96-bit nonce and then 32 bits of zeroes. This way, the counter can increment from 0 to $2^{32} - 1$ without bleeding into a nonce that may have been used before.

Theorem 25 (Counter-Mode Theorem)**.** For any $L > 0$ and any secure pseudo-random function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{X}$, E_{CTR} is a CPA-secure cipher from $\mathcal{K} \times \mathcal{X}^L$ to $\mathcal{X}^{L+1}$ (where the extra block comes from the IV).
In particular, for a q-query adversary $\mathcal{A}$ attacking E_{CTR}, there exists a pseudo-random function adversary $\mathcal{B}$ such that

$$\text{Adv}_{\text{CPA}}(\mathcal{A}, E_{\text{CTR}}) \leq 2\,\text{Adv}_{\text{PRF}}(\mathcal{B}, F) + 2q^2 L/|\mathcal{X}|\,.$$

This means that counter-mode is secure as long as $q^2 L \ll |\mathcal{X}|$, which is a better bound than for cipher block chaining.



	CBC	ctr mode
required primitive	PRP	PRF
parallel processing	No	Yes
security	q^2 L^2 << |X|	q^2 L << |X|
dummy padding block	Yes*	No
1 byte msgs (nonce-based)	16x expansion	no expansion

* for CBC, dummy padding block can be avoided using *ciphertext stealing*

Note that in counter mode, we also do not need padding because the extra part of the last block just gets cut off in xoring.

Attacks on Block Ciphers

Goal is just to distinguish a block cipher from a random permutation - if this can be done efficiently, the block cipher is broken. (As the adversary), we do not need to recover the entire message from the ciphertext in order to declare the block cipher insecure.

Linear and Differential Attacks

Suppose the key selection is very slightly biased, so there is some combination of indices $i_1, \ldots, i_r$, $j_1, \ldots, j_v$, $\ell_1, \ldots \ell_u$ such that

$$\Pr\Big[m[i_1] \oplus m[i_2] \oplus \cdots \oplus m[i_r] \bigoplus c[j_1] \oplus c[j_2] \oplus \cdots \oplus c[j_v] = k[\ell_1] \oplus k[\ell_2] \oplus \cdots \oplus k[\ell_u]\Big] = \frac{1}{2} + \epsilon$$

for some $\epsilon > 0$. This is equivalent to one bit of the key being slightly biased.

For DES, this exists for $\epsilon = 1/2^{21}$.

Theorem 26. Given $1/\epsilon^2$ random pairs (m, c) then

$$k[\ell_1] \oplus k[\ell_2] \oplus \cdots \oplus k[\ell_u] = \text{MAJ}\left[m[i_1] \oplus m[i_2] \oplus \cdots \oplus m[i_r] \bigoplus c[j_1] \oplus c[j_2] \oplus \cdots \oplus c[j_v]\right]$$

with probability at least 97.7%.

For DES, the key is 2^{56} bits. Using this bias, we can compute the key in time approximately 2^{43}, which is much less than brute-force, which would take 2^{56}.

The lesson is: don't design ciphers yourself !!

Quantum Attacks

Grover's theorem implies a quantum computer can find k in time $O(|\mathcal{K}|^{1/2})$. This doesn't exist today, but is relevant if you are encrypting something that needs to be secure for thirty to fifty years.

Message Integrity

The problem now is: Alice wants to send a message to Bob. The message is not secret, but we want to make sure the message is not changed by an attacker before Bob receives it.

Generally, the way we do this is by appending a tag to the message, where Alice has an algorithm S that takes in m and a key K, and generates a tag t, and Bob has a verifier V that takes in K, m, t and outputs whether or not t is the correct tag for m. (If not, this means the message was tampered with.)

Why does this require a key?

Because if an attacker wants to modify the message, and S doesn't take a key, then the attacker can just compute $S(m')$ and replace the original tag with the tag for m', and then Bob will not realize the message has been tampered with. Thus, the pair S, V is only secure if the adversary cannot generate their own valid (m, t) pairs.

Definition 27. A **message authentication code** (MAC) defined over $(\mathcal{K}, \mathcal{M}, \mathcal{T})$ is a pair of efficient algorithms $S : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ and $V : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \{\text{yes}, \text{no}\}$ such that for all $K \in \mathcal{K}$ and all $m \in \mathcal{M}$,

$$V(K, m, S(K, m)) = \text{yes}.$$

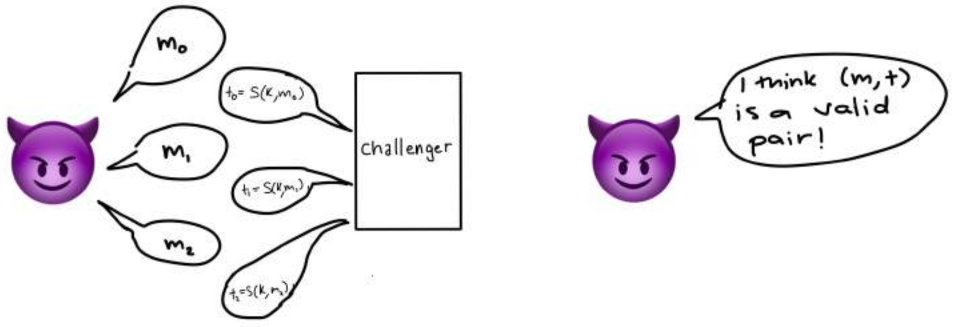
What is a secure message authentication code?

- attacker's power: chosen message attack
  Attacker can send $m_1, m_2, \ldots \in \mathcal{M}$ to the challenger and get back $t_1, t_2, \ldots$ where $t_i = S(K, m_i)$.
- attacker's goal: existential forgery
  Generate a new valid (m, t) pair (so that (m, t) is not in $\{(m_1, t_1), (m_2, t_2), \ldots\}$.

Definition 28. For a message authentication code $I = (S, V)$ and adversary $\mathcal{A}$, we define the MAC game as follows:

The challenger selects a random key K. Then, the adversary queries the challenger with any number of messages m_i, and for each message receives a tag $t_i = S(K, m_i)$. Then, the adversary tries to generate a new pair (m, t) such that $(m, t) \neq (m_i, t_i)$ for any i and $V(K, m, t) =$ yes.

A message authentication code is **secure** if for all efficient adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ wins the MAC game against I is negligible.



How do we construct a secure MAC?

Every secure pseudo-random function (where the range is sufficiently large) gives us a secure message authentication code.

Let F be a pseudo-random function over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$. Then, $I_F = (S, V)$ where $S(K, m) = F(K, m)$ and $V(K, m, t) =$ yes if and only if $t = F(K, m)$.

Theorem 29. If F is a secure pseudo-random function over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$ where $1/|\mathcal{Y}|$ is negligible, then I_F is a secure message authentication code.

The general proof idea is based on the idea that we can model F as a truly random function over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$, so that the probability $\Pr\left[t = F(K, m)\right] = 1/|\mathcal{Y}|$ for any (m, t).

So, AES gives us a secure MAC for 16-byte messages.

This leads us to the question of whether we can build a MAC for big messages from a MAC for small messages. And this is actually the question: how do we build a PRF over a large domain from a PRF over a small domain?

Three Constructions

- CBC-MAC: used in banking, standardized version called CMAC
- HMAC: used in internet protocols, very popular (still sequential)
- PMAC: parallel MAC, not very popular mainly because it arose too late

Remark 30. Suppose we have a MAC built from a PRF F that outputs n-bit tags ($\mathcal{Y} = \{0, 1\}^n$). Then, it is still secure if we truncate the tag to $w < n$ bits as long as $1/2^w$ is still negligible.

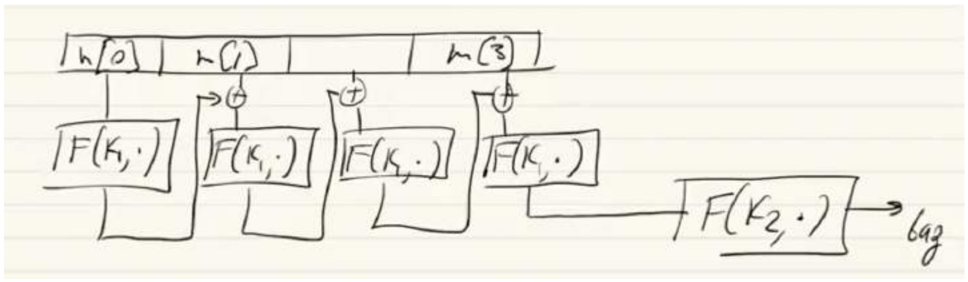
(a truncated version of a secure PRF is still a secure PRF)

.

Example 31. If I_F is a MAC that produces 2-bit tags, this is insecure because any (m, t) pair has a 1/4 chance of being correct (non-negligible).

Constructing a CBC-MAC

To construct a CBC-MAC, we take our message, apply CBC to it to get an encrypted one-block string, and then apply our PRF F to it one more time with a different key to generate our tag.



Lecture 6: Collision Resistance

We define F_{CBC} to be our CBC-constructed pseudo-random function, which takes in two keys (K_1, K_2).

Theorem 32 (CBC-MAC Theorem)**.** If F is a secure pseudo-random function, then F_{CBC} is also a secure pseudo-random function.

In particular, for every q-query adversary $\mathcal{A}$ against F_{CBC}, there is an adversary $\mathcal{B}$ that runs in around the same time against F, such that

$$\text{Adv}_{\text{PRF}}[\mathcal{A}, F_{\text{CBC}}] \leq \text{Adv}_{\text{PRF}}[\mathcal{B}, F] + q^2 L^{O(1)}/|\mathcal{X}|\,.$$

Why the last step?

"Why do we do anything in crypto? The answer is if we don't, then it's insecure."

We can show that raw CBC (without applying F one last time at the end) is insecure, because we can have an adversary $\mathcal{A}$ that does the following:

- choose an arbitrary $m \in \mathcal{X}$ (this means it is one block long)
- request tag for m; since this is only one block, we can see that $t = F(K_1, m)$
- create a forgery with the two-block message $m, m \oplus t$ and tag t (if we compute CBC on this, we can see that this is actually the tag)

Thus, there is a forger for raw CBC.

But, this forger created a forgery with a different-length message. This is not a coincidence - raw CBC is secure for fixed-length messages.

CBC-MAC padding

If the message length is not a multiple of the block size, we need to pad the last block.

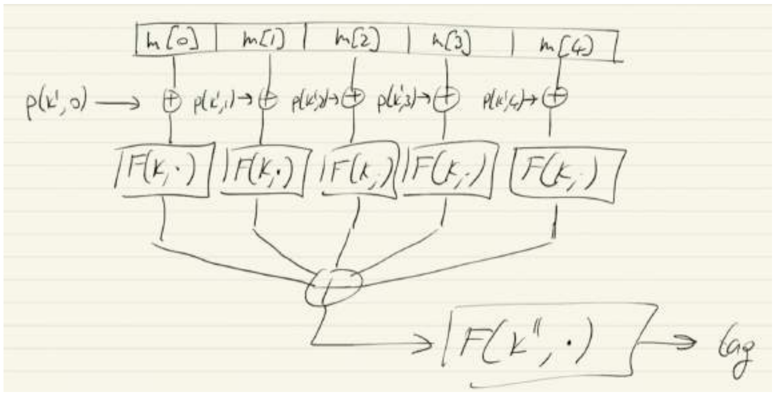
Padding with all zeroes is not secure, because an adversary can pass in an m that is not a multiple of the block size, get a tag t, and then $(m \parallel 0, t)$ is a valid forgery.

Thus, we need a one-to-one padding function. We will pad with "10000" until the message is a multiple of the block size, using a dummy block if the message is already a multiple of the block size.

This is interesting because even though we never need to "decrypt" CBC-MAC, we still need a one-to-one padding function because collisions in the padding process means that a message forgery is possible.

There is an alternative called CMAC which uses a clever padding system (which relies on randomization) so that the dummy block is not needed.

CBC-MAC is sequential, so it is not parallizable. PMAC (parallel MAC) is much better, but it is not widely used for historic reasons.



(a diagram of how PMAC works)

In this diagram $p(K', i)$ is an easy-to-compute function; if it were not there, this would be insecure because a swapping of the blocks would produce the same tag. Another benefit of PMAC is that we can change a single block of the file and easily modify the tag without having to recompute the entire process.

Collision Resistant Hashing

Let $H : \mathcal{M} \to \mathcal{T}$ be a hash function. ($|\mathcal{T}| \ll |\mathcal{M}|$)
A **collision** for H is a pair (m_0, m_1) such that $H(m_0) = H(m_1)$. By pigeonhole principle there must be many collisions on H, yet for a collision-resistant hash function these are very difficult to find.

Definition 33. A hash function $H : \mathcal{M} \to \mathcal{T}$ is **collision resistant** if for all (explicit) efficient adversaries $\mathcal{A}$,

$$\text{Adv}_{\text{CR}}[\mathcal{A}, H] = \Pr(\mathcal{A} \text{ outputs a collision for } H)$$

is negligible.

There is a problem with this definition - we know that there exists a collision (m_0, m_1), and we know there exists an algorithm that does nothing but print m_0 and m_1, which is efficient. However, this algorithm should not be "findable." In complexity theory terms, we want all uniform algorithms to fail at finding a collision.

Standard Examples

- 2001: SHA256, SHA384, SHA512 (numbered based on how many bits the hash values are)
- 2014: SHA3-256, SHA3-384, SHA3-512

But the 2001 versions can be computed very quickly and those are still the norm.

An Immediate Application

To get from a small MAC to a big MAC:
Let's say (S, V) is a secure MAC over $(\mathcal{K}, \mathcal{M}, \mathcal{T})$ for short messages, and $H : \mathcal{M}^{\text{big}} \to \mathcal{M}$ is a collision-resistant hash.

Then, we can define (S', V') to be a MAC over $(\mathcal{K}, \mathcal{M}^{\text{big}}, \mathcal{T})$ where

$$S'(K, m) = S(K, H(m))$$
$$V'(K, m, t) = V(K, H(m), t).$$

Theorem 34. If (S, V) is a secure MAC and H is a collision-resistant hash, then (S', V') is also a secure MAC.

The idea behind the proof is the following:

Suppose an efficient adversary $\mathcal{A}$ successfully attacks (S', V'); that is, it sent $m_1, m_2, \ldots, m_q$ to the challenger and got back $t_1, t_2, \ldots, t_q$. Then, it generated a valid forgery (m, t).
Then, either $H(m) = H(m_i)$ for some i, in which case the hash function is not collision-resistant, or $H(m) \neq H(m_i)$ for all i, in which case $(H(m), t)$ is a valid forgery on (S, V). Since both options break our assumptions, (S', V') must be secure.

Note that if the hash were not collision resistant (we can efficiently compute a pair (m_0, m_1) such that $H(m_0) = H(m_1)$) then our new MAC is not secure because if we get the tag t for m_0 we can produce the forgery (m_1, t).

How do we construct collision-resistant hashes?

(I have not taken notes on the birthday paradox, but an explanation can be found here.)

By the birthday paradox, if $H : \mathcal{M} \to \mathcal{T}$ outputs ℓ-bit strings, we can find a collision in time $2^{\ell/2} = O(\sqrt{|\mathcal{T}|})$. Specifically, we choose $2^{\ell/2}$ strings uniformly at random, compute their hashes, and look for a collision. Because this gives a collision with probability $1/2$, we can expect that after two iterations, we find a collision.

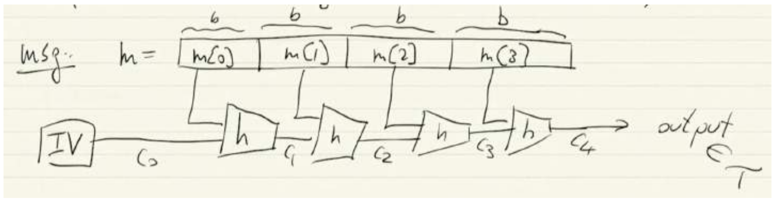
For a 128-bit hash, this means we can find a collision in time 2^{64}, which is not secure. For a 256-bit hash, the collision time is 2^{128}, which is much better.

Hard puzzle: We can find a collision in time $O(2^{\ell/2})$ using $O(1)$ space - how?

Quantum: There is some evidence that quantum computers can find a collision in time $2^{\ell/3}$, but this is mainly still an open problem (because the size necessary to do this is very unpractical).

Constructing a Collision-Resistant Hash

Merkle-Damgard construction (MD)



- $h : \{0,1\}^b \times \mathcal{T} \to \mathcal{T}$ is known as the compression function
- $c_0, c_1, c_2, \ldots$ are the chaining variables
- IV is a fixed publicly known value

(In SHA256, the block size is 512 bits.) We pad the message to ensure that it has a length that's a multiple of b, by adding a string "10000" and then the message length. If there is no space for the pad, we add a dummy block. This doesn't increase the size of the output, just the number of steps the hash function takes.

Theorem 35. If the compression function h is a collision-resistant hash, H_{MD} is a collision-resistant hash.

Lecture 7: Authenticated Encryption

Applications for Collision-Resistant Hashing

There are three general ways to create data integrity: one of them is the MAC we discussed before, if the sender and recipient both have a secret key.

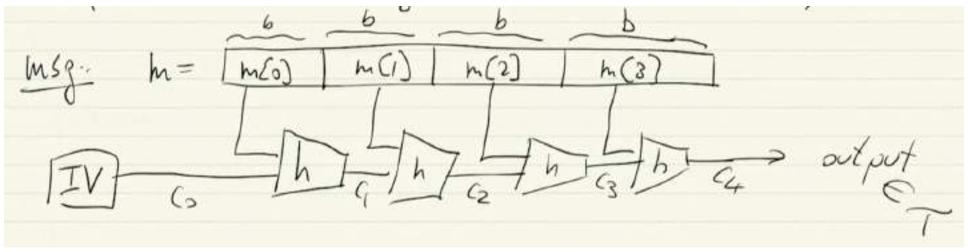
The other way is that if there is some small public read-only space (so only authorized people can write to the space, and the data is guaranteed to be unmodified), we can store the hashes of each of the files, so that in order to modify the file, the adversary would need to find a replacement file that has the same hash as before.

Later in the course, we will learn about the third way, which involves digital signatures.

Back to...

Constructing a Collision-Resistant Hash

Last time, we discussed the Merkle-Damgard construction, which converted a smaller compression function into one that worked on larger messages, while remaining collision-resistant.



Last time, we didn't get to the proof of Theorem 35, so we will prove that now.

Proof. The idea behind the proof is that if two messages $m_0 \neq m_1$ had the same final output under H_{MD}, then the hash h to produce the last output produced the same output for both messages. Thus, either we have found a collision on h, or the last block and the second-last chaining variable are the same for both messages; specifically, since the last block contains the message length, both messages must be the same length.

Then, since the second-last chaining variable is the same for both messages, we can again see that either we have found a collision on h or the third-last chaining variable and the second-last block are the same for both messages. We can repeat this backwards to the start of the message, and we get that either at some point we have found a collision on h, which contradicts the fact that h is collision-resistant, or $m_0 \neq m_1$, in which case we have not actually found a collision on H_{MD}. □

Now, we have to think about how to make these smaller compression functions h.

Constructing compression functions h

Davies-Meyers

Let $E(K, x)$ be a block cipher over $(\mathcal{K}, \mathcal{X})$ where $\mathcal{X} = \{0,1\}^n$.

Then, $h(m, c) = E(m, c) \oplus c$ (the message is being used as the key).

For example, AES would not be a secure cipher to use here because it is not designed to be used when the adversary controls the key.

The following is a vague statement of a theorem:

"Theorem" 36. If E is an ideal cipher (a random collection of permutations), then finding a collision on $h(m, c)$ takes time at least $2^{n/2}$.

This is as good of a lower bound as we can hope for, because we know the Birthday Attack runs in time $O(2^{n/2})$.

SHA256 uses a specific E called SHACAL2, with the Davies-Meyers method.

In summary, there are three steps to building collision-resistant hashes:

1. Merkle-Damgard, so that we need to build h
2. Davies-Meyers, so that we need to build E
3. building E

Building a MAC from a Hash Function

We know already that a PRF leads to a MAC, so we're really looking at how to build a PRF from a hash function $H : \mathcal{M} \to \mathcal{T}$.

Attempt 1: $F(K, m) = H(K \parallel m)$. This is a bad idea for Merkle-Damgard hash functions, because we know that if m' is one block, then $F(K, m \parallel \text{pad}(m) \parallel m') = h(m', F(K, m))$, so by knowing $F(K, m)$ the adversary can generate a new valid message, tag pair.

Standard Method:

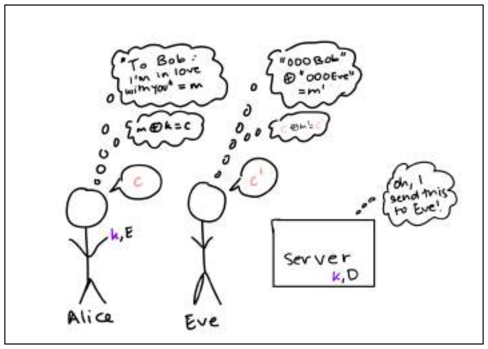
$$F_{\text{HMAC}}(K, m) = H\left((K \oplus \text{opad}) \parallel H((K \oplus \text{ipad}) \parallel m)\right).$$

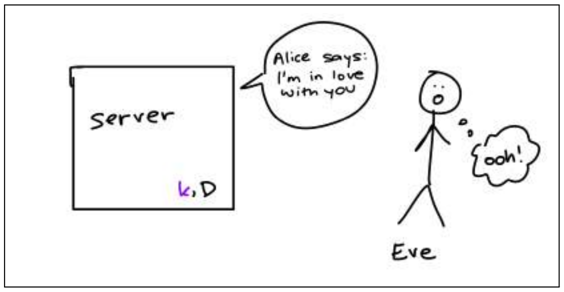
That is, we are taking the hash of the key (xored with some inner pad) appended with m, adding the key (xored with some outer pad) to the beginning, and taking the hash again. The outer and inner pads are fixed known strings.

Theorem 37. If the compression function $h(m, c)$ is a secure pseudo-random function with either m or c as the key, then HMAC is a secure PRF.

Thus, a crypto library that implements SHA256 can be used to create a secure MAC.

So far, we have looked at two things: how to send private messages (protect against eavesdropping) and how to protect data integrity. However, if we do not protect against tampering attacks, we don't have confidentiality - for example if an attacker Eve can modify a ciphertext email so that the plaintext email says "to Eve" rather than "to Bob," and the server reads that and sends the email to Eve, we no longer have confidentiality.





Definition 38. We define **authenticated encryption** to be the combination of confidentiality and data integrity.

Definition 39. We define the **ciphertext integrity** game as follows:

The challenger first picks a random key K. Then, the adversary $\mathcal{A}$ can send as many queries as they want, where a query occurs when $\mathcal{A}$ sends a message m_i and receives $c_i = E(K, m_i)$.

Then, the adversary sends a ciphertext c, and wins the game if this was not one of the c_i's and the decryption algorithm does not reject the ciphertext.

Definition 40. An cipher (E, D) provides **authenticated encryption** if (E, D) is CPA secure and provides ciphertext integrity.

Constructions Using MAC and Cipher Assume $I = (S, V)$ is a secure MAC and (E, D) is a CPA-secure cipher. We also have a key $K = (K_e, K_m)$ (one key for encryption and one for the MAC). How do we create authenticated encryption?

First Option: MAC then Encrypt
(This was used in TLS 1.0) We first compute the tag $t = S(K_m, m)$ and then compute $c = E(K_e, m \parallel t)$ and then send c.
This is **insecure** because it does not provide ciphertext integrity and is also vulnerable to timing-based attacks where the attacker checks how long the decrypter takes in order to figure out whether the tag was valid or not.

Second Option: Encrypt then MAC
(This was used in GCM and IPSec) We first encrypt the message to get $c = E(K_e, m)$, then compute the tag $t = S(K_m, c)$ and then send (c, t).

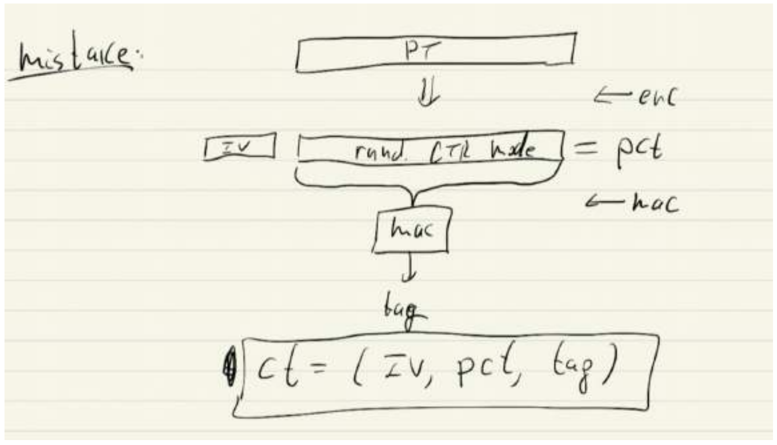
Theorem 41. If $I = (S, V)$ is secure and (E, S) are secure (and they have independent keys), then this option has authenticated encryption.

Proof. We can see that if this was insecure, then either it would not provide ciphertext integrity, which would imply our MAC was insecure (the adversary should not be able to create a new ciphertext with a valid tag), or it would not be CPA secure, which implies E was not CPA secure. $\square$

Third Option: Encrypt and MAC
(This was used in SSH) We compute the tag $t = S(K_m, m)$ and encrypt the message to get $c = E(K_e, m)$ and send (c, t).
This is **insecure** in the general case because our tag is allowed to reveal information about the message. In the specific case for SSH, they use a MAC that does not reveal such information.

Lecture 8: Intro To Public-Key Cryptography

There are many common mistakes in trying to achieve authenticated encryption, especially before it was standardized. One such mistake (made by Apple in Developer Mode) was the following:



(but using CBC mode, not CTR mode) This is bad because the IV was included in the final ciphertext but not in generating the tag, so it could be modified, and in CBC mode it can be modified to change the decrypted plaintext in small desired ways.

Galois Counter Mode (GCM) is the standard for authenticated encryption now; it uses a nonce-based counter mode encryption and then MACs.

Authenticated Encryption with Associated Data (AEAD)

In the interface for standard crypto libraries (such as boringSSL), the encryption function allows you to include associated data along with the plaintext, key, and IV in the input.

(As a sidenote, the boringSSL encrypt function outputs the ciphertext and the tag as two separate variables, which is bad because the developer using it is then able to use the ciphertext without the tag and create insecure ciphers.)

The associated data gets concatenated to the beginning of the ciphertext when generating the tag, and then gets passed as a parameter into the decryption function so that the decryption algorithm can make sure the ciphertext is tied to the correct associated data.

For example, a packet sent across a network is sent with a public header, so only the payload is encrypted, but the header is passed as associated data, so we can make sure the header did not get modified before decrypting the payload.

Case Study: GCM in TLS 1.3

The browser and server are trying to communicate, and they are sending TLS records back and forth, of size at most 2^{14} bytes.

We agree on two sets of keys, $K_{B\to S}$ and $K_{S\to B}$. Both the browser and server know both keys (they each need one for encryption and one for decryption), and these are derived from a master secret using a process called Hash-Based Key Derivation Function (HKDF).

They use what is called stateful encryption; each side maintains two 64-bit write sequence counters, WSC_B and WSC_S. They both start at zero at setup, and WSC_B counts the number of records the browser sent to the server, and WSC_S counts the number of records the server sent to the browser. For example, the browser's WSC_B would increase each time it sends a record, and the server's WSC_B would increase each time it recieves a record.

Then, when the browser sends a record to the server, it sends it with the associated data (WSC_B, `record_type`, "protocol version 3.1"). [It's 3.1 and not 1.3 for historical reasons] The nonce for the encryption is $WSC_B \oplus$`iv`, where the IV is generated at session setup. It then sends the ciphertext, with header information appended at the beginning.

Note that because TLS is built on top of PCP, which guarantees in-order delivery, the server can keep its own copy of the counter and compute the nonce on its own, without it having to be sent with the packet. This prevents a replay attack - for example, if you order a book from Amazon, a replay attack could send the ciphertext for the same order later, and you will end up buying two books.

This is the end of Part 1 of the course.

In summary, part 1 of the course was about symmetric cryptography. You should now be able to answer the question:
"Alice and Bob have a shared key. Alice wants to send a message to Bob. What do they do?"
with:
"They need to use an authenticated encryption mode such as AES-GCM."

The two fundamental points you should know by now are:

1. If you only need integrity: use a MAC.
2. If you need confidentiality: use an authenticated encryption scheme.

Now, we turn to the question:
Where does the shared key K come from?

Answer: it comes from a key exchange protocol!

Basic Key Exchange

Method 1: Trusted Third Party (TTP)

Each person out of our millions of people (or four) have a shared secret key with the trusted third party. Specifically, we can say that Alice has shared secret key K_A with the TTP and Bob has shared secret key K_B with the TTP. How do Alice and Bob get a shared secret key?

Here is a toy protocol for a key exchange (because it is only secure against eavesdropping).

We have a CPA-secure cipher (E, D) with key space $\mathcal{K}$. The TTP generates a new shared key K_{ab} randomly from $\mathcal{K}$. Then, the TTP sends Alice $E(K_A, K_{ab})$ and the ticket $E(K_B, K_{ab})$. Alice decrypts the first ciphertext to get K_{ab}, and she sends the ticket to Bob, who is able to decrypt it and also get K_{ab}. Since (E, D) is CPA-secure, the adversary learns nothing about K_{ab}, so this is safe against eavesdropping.

However...

- the TTP is needed for every key exchange, so if it goes offline, no one can communicate

- the TTP knows all secret keys (backdoor heaven) - reminder that there are major societal implications to the cryptography we are discussing !!
- basis of Kerberos system (Windows) and used in many corporate networks

Basic Question: Can we generate a session key without an online TTP?
Answer: Yes! This is the basis of public-key cryptography.

How?

- Merkle (1974): Merkle Puzzles can do it using only symmetric ciphers, but would require sending gigabytes of data (and cannot be improved using this method), so completely impractical.
- instead, use *algebra*:
  - Diffie-Helman (1976)
  - RSA (1977)
  - Elliptic Curve Cryptography (1984)
  - ... and more

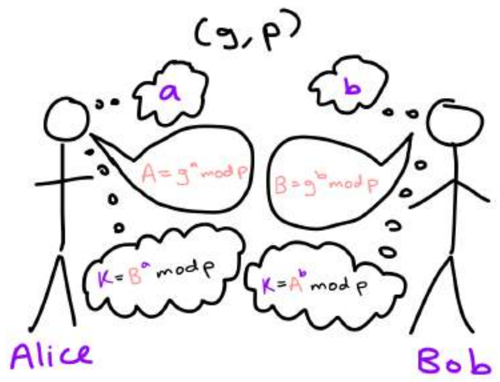
Key Exchange without an Online TTP

Note that for now we are just focusing on eavesdropping security.

Basic Diffie-Helman Key Exchange

Fix a large prime p (on the scale of 600 digits - we will see next lecture how this can be done quickly) and an integer g such that $1 < g < p$. Then, Alice knows some integer a such that $1 \leq a \leq p-1$, and Bob knows some integer b such that $1 \leq b \leq p-1$.

Then, Alice computes $A = g^a \pmod{p}$ (we will see next lecture how this can be done quickly) and sends this to Bob.
Bob computes $B = g^b \pmod{p}$ and sends this to Alice.



Then, we define $K_{ab} = g^{ab} \pmod{p}$ (note that this is *not* the product of A and B). Alice can compute this easily as $B^a \pmod{p}$ and Bob can compute this easily as $A^b \pmod{p}$, but the adversary, who only has A, B, g, and p, cannot easily compute this when p is large enough. (However, this is easily computable for an adversary with a quantum computer - once quantum computers become readily available, there is a different algorithm ready that will replace this.)

We will also see next lecture why this is secure (and $g^{ab} \pmod p$ is not easily computable by the adversary), and why this is dependent on p being prime.

You can find a lot of explanations of this topic online, but many of them miss that the important part of this protocol is not why it works - it's "why is this secure?"

This protocol is used everywhere on the internet.

Next time: how do we compute $g^a \pmod p$ quickly?

Lecture 9: Cryptography Using Finite Cyclic Groups

Modular Arithmetic

We will use n to denote a positive integer, p and q to denote primes, $\mathbb{Z}_n$ to denote the integers modulo n (the set $\{0, 1, \ldots, n-1\}$, where we add, subtract, and multiply numbers mod n to keep the results of arithmetic operations within the set).

We say that n and m are **relatively prime** if $\gcd(n, m) = 1$.

Remark 42 (Euclid, 200BC)**.** For all integers $n, m > 0$ there exists integers a, b such that

$$a \cdot n + b \cdot m = \gcd(m, n)$$

We find a, b using Euclid's algorithm, which runs in $O(\log(m+n))$. This is a very interesting algorithm - if you do not know it, it is useful to look up and learn.

Example 43. The gcd of 12 and 18 is 6. We can see that

$$2(12) + (-1)(18) = 6,$$

so we can set $a = 2$ and $b = -1$.

To learn how to divide in modular arithmetic, we need to discuss inverses.

Definition 44. The inverse of $x \in \mathbb{Z}_n$ is $y \in \mathbb{Z}_n$ such that

$$x \cdot y = 1$$

in $\mathbb{Z}_n$; that is $xy = 1 \pmod{n}$. Then, $y = x^{-1}$.

Example 45. When n is an odd integer, we can see that 2^{-1} in $\mathbb{Z}_n$ is equal to $\frac{n+1}{2}$, because $2\left(\frac{n+1}{2}\right) = n + 1 = 1 \pmod{n}$.

Which elements in $\mathbb{Z}_n$ have an inverse?

Lemma 46. An element $x \in \mathbb{Z}_n$ has an inverse if and only if $\gcd(x, n) = 1$.

Proof. If $\gcd(x, n) = 1$, then by Euclid's Algorithm we can find $a, b \in \mathbb{Z}$ such that $a \cdot x + b \cdot n = 1$. But then taking everything mod n, we get that $a \times x = 1$ in $\mathbb{Z}_n$, so $a = x^{-1}$ in $\mathbb{Z}_n$.

If $\gcd(x, n) > 1$, then for all $a \in \mathbb{Z}$, we can see that $\gcd(ax, n) > 1$ as well. But then $ax \neq 1$ in $\mathbb{Z}_n$, and since this is true for all a, x has no inverse. $\square$

More notation: $\mathbb{Z}_n^*$ is the set of invertible elements in $\mathbb{Z}_n$, or the set $\{0 \leq x < n : \gcd(x, n) = 1\}$.

Example 47. For any prime p, $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$; that is, it is everything in $\mathbb{Z}_p$ besides 0.

For any $x \in \mathbb{Z}_n^*$, we can compute x^{-1} in $\mathbb{Z}_n$ in time $O(\log^2 n)$.

Now we can solve modular linear equations:
If we have to solve

$$a \cdot x + b = 0$$

in $\mathbb{Z}_n$, where $a \in \mathbb{Z}_n^*$, we know that

$$x = (-b) \cdot a^{-1}.$$

Moreover, we can use this to solve a system of linear equations using Gaussian elimination, since this just requires adding, subtracting, multiplying, and inverses.

What about quadratic equations?

In order to understand this, we need to understand the structure of $\mathbb{Z}_p^*$. (We are focusing on primes for now, we will get to $\mathbb{Z}_n^*$ for composite n later.)

Theorem 48 (Fermat, 1640)**.** Let p be a prime. Then, for all $x \in \mathbb{Z}_p^*$,

$$x^{p-1} = 1$$

in $\mathbb{Z}_p$.

Example 49. When $p = 5$, $3^4 = 81 = 1$ in $\mathbb{Z}_5$.

Moreover, this means that for $x \in \mathbb{Z}_p^*$, we have that $x(x^{p-2}) = 1$ in $\mathbb{Z}_p$, so $x^{p-2} = x^{-1}$. This gives us another way to compute inverses in $\mathbb{Z}_p^*$, but its running time is $O(\log^3 p)$.

We can also use this to generate a (probable) 2048-bit prime: pick p randomly from the set of 2048-bit integers. If $2^{p-1} = 1$ in $\mathbb{Z}_p$, output p. If not, pick a different p and continue until we find one that works. The probability that we output a non-prime is small but nonzero.

Example 50. In $\mathbb{Z}_{11}^*$, what is 3^{2022}? This is equal to

$$(3^{10})^{202} 3^2 = 1^{202}(3^2) = 9.$$

In general, for all $a \in \mathbb{Z}_p$,

$$a^k = a^{k \pmod{p-1}} \text{ in } \mathbb{Z}_p.$$

Theorem 51 (Euler)**.** The structure theorem of $\mathbb{Z}_p^*$ says that $\mathbb{Z}_p^*$ is a **finite cyclic group**. This means there exists some $g \in \mathbb{Z}_p^*$ such that

$$\left\{1, g, \ldots, g^{p-2}\right\} = \mathbb{Z}_p^*.$$

Such a g is called a **generator**.

Example 52. We can express $\mathbb{Z}_7^*$ in terms of powers of 3 as

$$\{1, 3, 2, 6, 4, 5\},$$

so 3 is a generator of $\mathbb{Z}_7^*$. But we can see that the powers of 2 in this group are $\{1, 2, 4\}$, so 2 is not a generator of this group.

We can define a finite cyclic group more generally.

Definition 53. A **group** is a pair $(G, \cdot)$ where G is a set and $\cdot : G \times G \to G$ is a multiplication operation.

For $(G, \cdot)$ to be a group it must be **closed** under the operation, which means for any $a, b \in G$, $a \cdot b$ must also be in G.

Moreover, it must contain an identity e, such that $e \cdot a = a$ for any $a \in G$, every element $a \in G$ must have an inverse $a^{-1} \in G$ such that $a \cdot a^{-1} = e$, and the multiplication operation must be associative.

Definition 54. A **finite** group is one in which the set G is finite.

Definition 55. A **cyclic** group is one which has a generator g as defined above.

(Cyclic groups also have the property that the multiplication operation is commutative, but this is implied by the fact that they have a generator.)

With weird definitions of the multiplication operations, we can come up with some strange finite cyclic groups.

Definition 56. For an element $h \in G$, the **order** of h is the size of the set $\{1, h, h^2, \dots\}$.

Example 57. In $\mathbb{Z}_7^*$, the order of 3 is 6 but the order of 2 is 3, based on the sets from the previous example.

In general, if g is a generator of G, its order must be $|G|$.

Remark 58. For all $g \in G$, $g^{\text{order}(g)} = 1$.

Theorem 59 (Lagrange)**.** For all $g \in G$, order(g) will be a factor of $|G|$.

As a corollary, for all $g \in G$, $g^{|G|} = 1$ in G.

How do we compute roots?

Suppose that G is a finite cyclic group with known prime order and with a generator $g \in G$.

Example 60. Fix a prime p. Then, pick an element h in $\mathbb{Z}_p^*$ such that order$(h) = q$, where q is prime. Then, we can let G be the subgroup of $\mathbb{Z}_p^*$ generated by h.

Our problem is, given $h \in G$ and some $1 \le e \le |G|$, find

$$y = h^{1/e}.$$

This means, find y such that $y^e = h$.

Our algorithm to do this is quite simple:

1. Compute $\alpha = e^{-1} \pmod{q}$ where $q = |G|$.
2. Output $y = h^\alpha \in G$.

Why is $y^e = h$? Well, $\alpha \cdot e = 1 \pmod{q}$. Then, there is some integer k such that $\alpha e = 1 + kq$ by definition of mod. But then,

$$y^e = (h^\alpha)^e = h^{\alpha e} = h^{1+kq} = h(h^q)^k = h,$$

since $h^q = 1$ in G.

How do we compute exponents?

Let g be a finite cyclic group of order q. We take as input $h \in G$, and $x \in \mathbb{Z}$, and we want to output $h^x \in G$.

We use an algorithm called the repeated squaring algorithm.

Example 61. Suppose we want to compute h^{13}. We begin by writing 13 in binary; $13 = 1101_2$. But this means that $13 = 8 + 4 + 1$. But this means that

$$h^{13} = (h^8)(h^4)(h).$$

The repeated squaring algorithm does the following:
First, take the binary expansion of x, and let $x[i]$ be the i^{th} digit in the binary expansion (where $x[0]$ is the least significant digit and $x[n]$ is the most significant digit). Then, run the following process:

```
z = 1, y = h
for i = 0, 1, ..., n:
    if x[i] == 1:
        z = z*y (in G)
    y = y*y (in G)

output z
```

We can see that using repeated squaring, y contains the current relevant power of h, and we multiply that into our accumulator z if it is one of the powers of two that sum to the power we are computing.

This runs in at most $2\log_2 x$ multiplication steps, which is very efficient. If the exponent is larger than the size of the group, we can just mod it by the size of the group, so this process takes at most $2\log_2|G|$ multiplication steps.

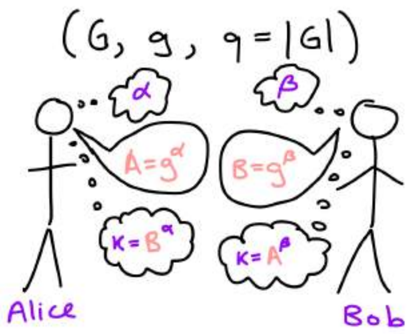
In the cryptographic group we're interested in, multiplication takes time $O(\log^2|G|)$. Thus, overall, exponentiation takes time $O(\log^3|G|)$.

Lecture 10: Public-Key Encryption Schemes

We will go back to looking at the Diffie-Helman protocol, but now more generally in a finite cyclic group. In practice, this group G is an elliptic curve, not the integers mod a prime p.

Diffie-Helman (in a cyclic group)

Alice and Bob both agree on (G, q, g), where G is a finite cyclic group, q is the order of that group, and g is a generator in G.
Then, Alice picks some random $0 \leq \alpha < q$ and Bob picks some random $0 \leq \beta < q$. Alice sends $A = g^\alpha \in G$ to Bob, and Bob sends $B = g^\beta \in G$ to Alice.
Then, the key is $g^{\alpha\beta} \in G$, which Alice can compute as B^α and Bob can compute A^β.



This means an eavesdropper sees (g, g^α, g^β) and wants to compute $g^{\alpha\beta}$ - we want this to be a hard problem.

Definition 62. The **computational Diffie-Helman assumption** (CDH) holds in (G, g) if for all efficient adversaries $\mathcal{A}$, the probability

$$\Pr\left[\mathcal{A}(g, g^\alpha, g^\beta) = g^{\alpha\beta}\right]$$

is negligible when α, β are randomly chosen from the integers mod q.

An algorithm trying to compute $g^{\alpha\beta}$ from (g, g^α, g^β) is called a CDH algorithm.

For the Diffie-Helman protocol, the group G and the generator g are standardized at a national level.

cyclic group:	$\mathbb{Z}_p^*$	**the elliptic curve over** $\mathbb{Z}_p$
best known CDH algorithm:	General Number Field Sieve (GNFS)	baby step-giant step (BSGS)
running time of that algorithm:	$e^{\sqrt[3]{\ln(p)}}$	$\sqrt{p} = e^{\frac{1}{2}\ln(p)}$
to be secure...	p must be $>$ 2048 bits	p can be 256 bits
speed	slow	not so slow !!

Table 1: Our two examples of cyclic groups where CDH holds.

(A problem for you to think about: for any group of size p, how do we compute the CDH in time $\sqrt{p}$?)

Now we move on to a related problem:

The Discrete Log in G

The discrete log problem is: given $h \in G$, find $\alpha \in \mathbb{Z}_q$ (the integers mod q) such that $h = g^\alpha$.

Example 63. In the group $\mathbb{Z}_{11}^*$, we can compute the discrete log base 2:

n	1	2	3	4	5	6	$\cdots$
$\text{Dlog}_2(n)$	0	1	8	2	4	9	$\cdots$

Remark 64. If discrete log in G is easy, this means that CDH in G is easy.

Proof. If discrete log is easy, given (g, g^α, g^β), we can compute α easily as $\text{Dlog}_g\, g^\alpha$. Then, once we have alpha, we can just compute $g^{\alpha\beta}$ as $\left(g^\beta\right)^\alpha$. □

Thus, for CDH to be hard in G, the discrete log problem must also be hard.

The converse, "CDH is easy implies discrete log is easy" is an open problem that Professor Boneh has worked for a while on.

We have one more fact about modular arithmetic to learn:

Arithmetic mod composites n

Definition 65. Let the **Euler totient function**, $\varphi(n)$ be equal to $|\mathbb{Z}_n^*|$. That is, let $\varphi(n)$ be equal to the number of elements that are invertible mod n, or the number of elements relatively prime to n.

Example 66.

$$\varphi(12) = \left|\{1, 5, 7, 11\}\right| = 4.$$

Theorem 67 (Euler)**.** For all $x \in \mathbb{Z}_n^*$, $x^{\varphi(n)} = 1$ in $\mathbb{Z}_n$.

This is a generalization of Fermat's theorem, applying it also to composites.

Example 68. We can see that in $\mathbb{Z}_{12}$,

$$5^{\varphi(n)} = 5^4 = 625 = 1$$

in $\mathbb{Z}_{12}$.

We know that computing $\varphi(n)$ is as hard as factoring n, which we believe to be a hard problem. If we don't know the factorization of n, then for $e > 1$, computing e^{th} roots of elements of $\mathbb{Z}_n^*$ is also believed to be difficult.

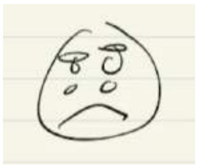
With a quantum computer...

- Dlog for all finite cyclic groups,
- factoring n,

- CDH

are all easy (cubic time) on a quantum computer.

So do we just go home and cry?



No, there's a new (very productive) field called **post-quantum cryptography** that explores cryptography that we can do on a regular computer that would be secure even against a quantum attacker.

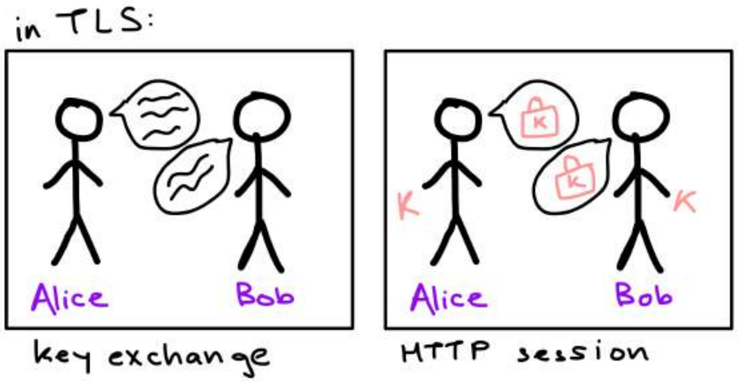
Examples of this are lattice systems and isogeny systems. But these are computationally heavier than the methods we use right now, which is why we won't switch to them until they become necessary.

The reason we believe Diffie-Helman is hard is because people have tried for forty years to break it and have not succeeded - there is no mathematical proof that it is a difficult problem.

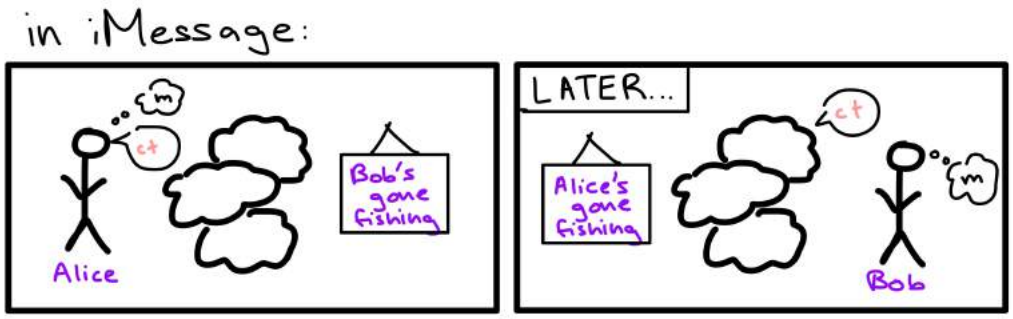
We are now switching topics, and we will start talking about public-key cryptosystems.

Two settings for communication: interactive vs. non-interactive

The below is an example of interactive communication:



while the below here is an example of non-interactive communication:



In particular, in non-interactive communication, since Alice and Bob are not online at the same time, instead communicating through a third party, they cannot set up a session key.

So how do they communicate??

This is the topic of public-key encryption (PKE)

Alice has a message m she wants to send to Bob. She takes PK_b, which is Bob's public key, and encrypts the message using Bob's public key. Then, she sends over $c = E(PK_b, m)$.

Then, Bob gets the ciphertext c, and he can decrypt the message using his secret key: $m = D(SK_b, c)$.

This is almost exactly the same as the process for symmetric encryption, except the key for encryption and the key for decryption are different. We call the pair (PK_b, SK_b) a key pair.

Definition 69. A **public-key encryption scheme** (PKE) over $(\mathcal{M}, \mathcal{C})$ is a tuple of algorithms:

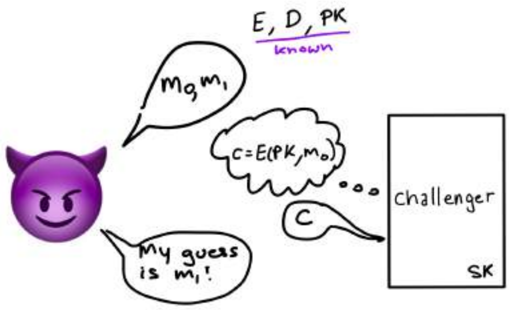
- Gen() produces a key pair (PK, SK) and is a randomized algorithm
- $E(PK, m) \to c$ is a randomized algorithm
- $D(SK, c) \to m$ or `reject` is a deterministic algorithm

such that for any (PK, SK) produced by Gen() and for any message $m \in \mathcal{M}$,

$$D(SK, E(PK, m)) = m.$$

For public-key encryption, we can again define semantic security:

Definition 70. For public-key encryption, we define **semantic security** (security against eavesdropping) using the following game:
The challenger chooses $b = 0$ or $b = 1$ and generates a (PK, SK) pair. Then, the adversary sends two messages m_0 and m_1 of the same length, and the challenger returns $E(PK, m_b)$. Finally, the adversary guesses the value b.



A public-key encryption scheme $\mathcal{E}$ is **semantically secure** if for all efficient adversaries $\mathcal{A}$:

$$\text{Adv}[\mathcal{A}, \mathcal{E}] = \left|\Pr[W_0 = 1] - \Pr[W_1 = 1]\right|$$

is negligible, where W_b is the output of the adversary in experiment b.

Remark 71. A semantically secure public-key encryption scheme means E must be randomized.

Since the adversary knows PK, against a deterministic algorithm E the adversary could just do the following:
First, send two messages $m_0 =$"hello" and $m_1 =$"world." Then, get back a ciphertext c. If $c = E(PK, \text{"hello"})$ return 0, otherwise return 1. This has an advantage of almost 1. Thus, deterministic E's are not semantically secure.

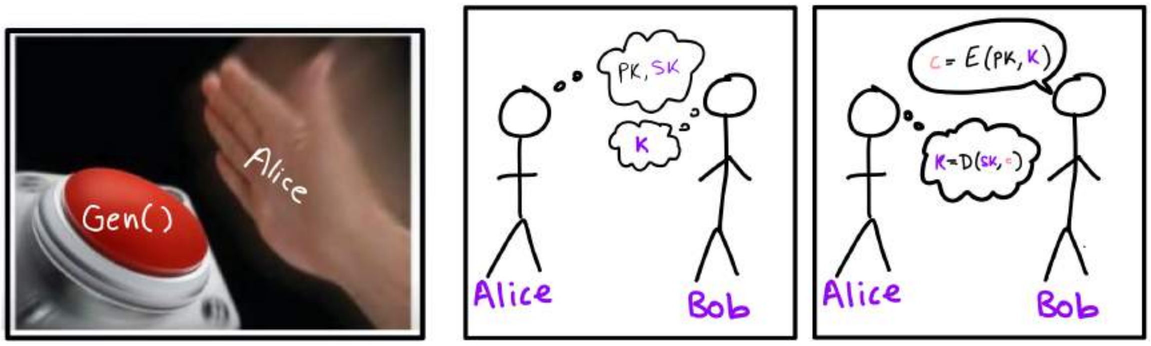
But we can't forget about ...
Data Integrity for Public-Key Encryption
But we won't discuss this right now either. Stay tuned for next time ☺
(Specifically, as we have defined this now, the recipient knows nothing about the identity of the sender. Next time, we will fix this using digital signatures).

Now, we will turn to applications of public-key encryption. The first is:

Toy Key Exchange from PKE

Again, this will be secure against eavesdropping.

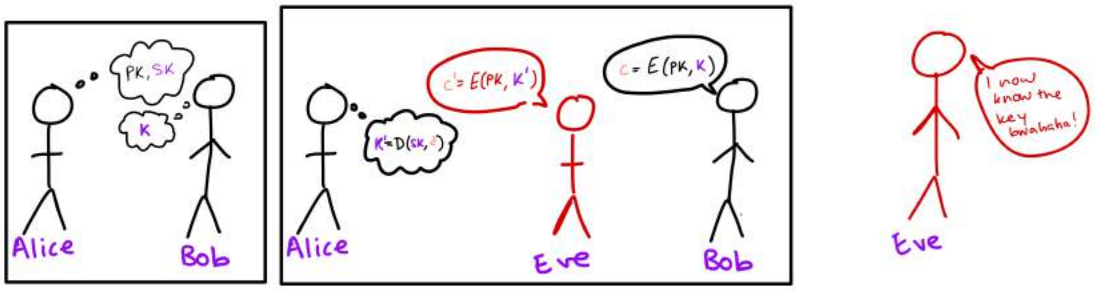
Alice generates a random pair (PK, SK) from Gen(). Then, she sends PK to Bob, and Bob randomly generates a shared key K from $\mathcal{M}$, and then sends Alice $c = E(PK, K)$. Using her secret key, Alice can decrypt this and get $K = D(SK, c)$. Now, they both have the secret key K.

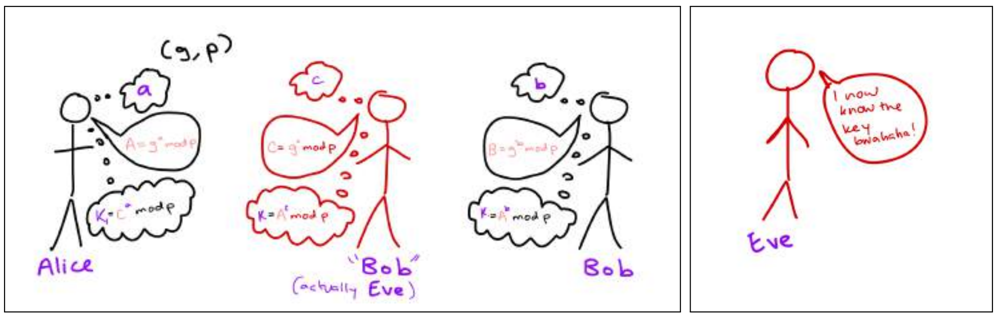


To show this is secure: an eavesdropper sees PK, c and wants K (which is the unencrypted version of c). But since the PKE is semantically secure, an efficient adversary $\mathcal{A}$ cannot distinguish c from the encryption of any other message $m \in \mathcal{M}$ of the same length. Thus, an efficient adversary can't obtain K, so this is secure against eavesdropping.

The difference between this and the Diffie-Helman key exchange is that everyone has a permanent public key registered with the cloud (in the iMessage example), so Bob can generate and encrypt their shared key while Alice is offline, and then Alice will know the shared key as soon as she goes online. This requires a lot less back-and-forth before actually getting the shared key, compared to Diffie-Helman.

<u>Both toy key-exchange protocols are insecure against person-in-the-middle attacks</u>





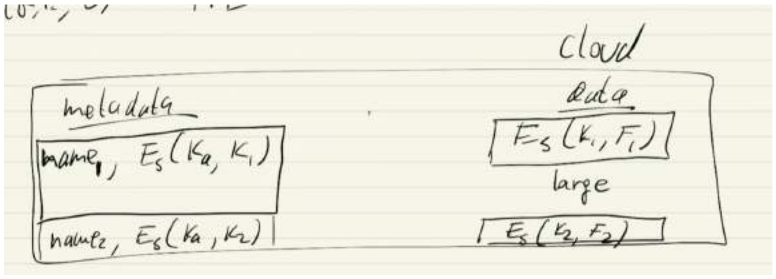
This is why both key exchange protocols are toy protocols: you need more machinery to defend against person-in-the-middle attacks. (This is why we're learning about digital signatures next time.)

Lecture 11: RSA Encryption

Three Applications of Public-Key Encryption

1. Toy Key Exchange (as discussed last lecture)
2. File Sharing in an Encrypted Filesystem:
   if (E_s, D_s) is a symmetric cipher over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ and (Gen, E, D) is a PKE

   Alice has K_a which is a key randomly selected from $\mathcal{K}$. She wants to upload a file F_1 to the cloud. So, she will choose a random key K_1 from $\mathcal{K}$, and compute $E_s(K_1, F_1)$ and $E_s(K_a, K_1)$. She uploads $E_s(K_a, K_1)$ to the metadata section, along with information such as the filename, and she uploads $E_s(K_1, F_1)$ to the data section.

   Then, Alice wants to give Bob access to F_1 but not F_2; that is, she wants him to know K_1 but not K_2. Moreover, she can't directly send him K_1 because she cannot interact with him directly - just the cloud. So, she does the following:

   - get PK_B from the cloud (assume this can be done reliably, the how will be discussed later)
   - append the metadata section for F_1 to include $E(PK_B, K_1)$
   - now Bob can use the metadata section to access K_1, and read/write to F_1

   When Windows uses EFS, this is exactly what happens!

   Answers to Questions:

   - Once we have given Bob access, we can revoke access by removing $E(PK_B, K_1)$ but this assumes he hasn't saved F_1 or K_1 already. To protect against the latter, we can change the encryption key for F_1 and re-encrypt it, but we cannot protect against the former.
   - The reason we use a separate key K_1 for encrypting F_1 is that this prevents us from separately encrypting F_1 using each person's keys - since F_1 is large and K_1 is small, having multiple encryptions of K_1 is a much more viable option.
3. Encrypted Chat System (like iMessage): Alice and Bob both give their public keys to the cloud. Then, when Alice wants to send a message to Bob, she asks for PK_B and sends $E(PK_B, m)$ to the cloud - this can sit in the cloud until Bob is ready to decrypt and read it. Bob does the same when he wants to send a message to Alice.

Now, we have seen how PKEs work and some applications, so we turn to the question of:

Constructing a PKE: ElGamal encryption (1982, Stanford)

This is a PKE constructed from Diffie-Helman.

Ingredients:

- G: a finite cyclic group of order q with generator $g \in G$
- (E_s, D_s): a symmetric cipher over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$
- $H : (G, G) \to \mathcal{K}$ is a hash function

First, we construct Gen():
We select an α randomly from $\mathbb{Z}_q$ and we compute $h = g^\alpha$. Then $SK = \alpha$ and $PK = h$. We know that computing α from h is hard because this is the dlog problem.

Then, we construct $E(PK, m)$:
To encrypt a message m, where $PK = h$, we select a β randomly from $\mathbb{Z}_q$ and compute $u = g^\beta$ and $v = h^\beta = g^{\alpha\beta}$.
Then, we create a key $K = H(u, v) \in \mathcal{K}$. (What's important here is that the key is derived from v, which is the Diffie-Helman secret.)
Then, we compute $c = E_s(K, m)$, and we output (u, c).

Then, we construct $D(SK, (u, c))$:
We know that $SK = \alpha$. Then, we can see that $v = g^{\alpha\beta} = u^\alpha$. So given u, we can compute v. Then, we can compute $K = H(u, v)$ as before, and then $m = D_s(K, c)$.

Answers to Questions:

- H is a deterministic hash function - in reality, people use SHA-256
- When we want to add authenticated encryption to this, we will add all the authenticated encryption to the symmetric cipher (e.g. using AES-GCM as our symmetric cipher).
- Putting u in the hash mainly just allows for a cleaner security proof, it's not strictly necessary.

Performance: encryption takes two exponentiations in G and one symmetric encryption, decryption takes one exponentiation in G and one symmetric decryption - this is not bad!
As a standard, we use ECIES (elliptic curve integrated encryption system).

Security: these theorems will be stated but not proved. The proofs are in the book (not too difficult) if you would like to see them.

Theorem 72. Our PKE (Gen, E, D) is **semantically secure** assuming

- CDH holds in our group
- (E_s, D_s) is semantically secure (but we don't need CPA security)
- H is a **secure key derivation** function (vaguely, this means it preserves the entropy in v)

Theorem 73. Our PKE (Gen, E, D) is **CCA secure** (will be defined later today, but vaguely means secure against tampering) assuming:

- "interactive D-H" assumption holds in our group (not going to explain, but a stronger assumption than CDH)
- (E_s, D_s) provides authenticated encryption
- H is a "random oracle" (vaguely means an ideal hash function)

We've been talking about cryptography using finite cyclic groups, which is what is primarily being used today. Now, we will move on to cryptography using the problem of factoring large integers. (not factoring large primes - that's easy!) RSA is on its way out, though, because it is difficult to implement securely in practice.

Definition 74. A PKE (Gen, E, D) is secure against chosen ciphertext attack (**CCA secure**) if all efficient adversaries have negligible advantage in the following game:

First, the challenger selects $b = 0$ or $b = 1$ at random, calls Gen() to generate a public key and secret key, and sends PK to the adversary. Then, the adversary sends an encryption query m_0, m_1 to the challenger, and gets back $c = E(PK, m_b)$.

Then, the adversary can send as many decryption queries as they want, where in a decryption query the adversary sends $c_i \neq c$ to the challenger and gets back $D(SK, c_i)$.

Finally, the adversary guesses b.

The adversary can't just ask the challenger to decrypt c, but it can encrypt/decrypt whatever else it wants. Again, we can see that we need the encryption algorithm to be randomized in order for this to be possible. Moreover, it shouldn't be the case that we can modify the ciphertext and predict how that would modify the decrypted message (because then we could send c with those modifications to the challenger in a decryption query). Thus, this protects from tampering because if the adversary tampers the ciphertext, the decryption will (very likely) fail or return garbage values.

Now, before we switch gears to talk about RSA, we will talk about trapdoor functions.

Definition 75. A **trapdoor function** $X \to Y$ is a triple of efficient algorithms (Gen$()$, F, F^{-1}) where:

- Gen() is a randomized algorithm that generates (PK, SK)
- $F(PK, \cdot) : X \to Y$ is a deterministic function
- $F^{-1}(SK, \cdot) : Y \to X$ is a deterministic function that inverts $F(PK, \cdot)$

More precisely, for all PK, SK that could be output by Gen() and for all $x \in X$, $F^{-1}(SK, F(PK, x)) = x$.

It's extremely difficult to construct good trapdoor functions, because we're looking for *deterministic* functions that anyone can compute using the public key but cannot be inverted without the secret key.

Definition 76. A trapdoor function is **secure** if $F(PK, \cdot)$ can be evaluated but cannot be inverted without SK. Specifically, we can consider the following game:

The challenger generates (PK, SK) and chooses $x \in X$ at random. Then, the adversary is given a public key PK and $y = F(PK, x)$ and outputs a guess x'.

The trapdoor function is secure if for all efficient adversaries $\mathcal{A}$,

$$\text{Adv}[\mathcal{A}, F] = \Pr[x = x']$$

is negligible.

We can see that if the adversary guesses an x', it can compute $F(PK, x')$ and check if it equals y - we need a space X large enough that if an efficient adversary randomly guesses an arbitrary number of times, it has negligible probability of hitting the correct x.

ElGamal is not a trapdoor function - we can compute g^α given α very easily, but there is no secret key (we know of) that would make it easy to compute α given g^α. Public-key encryption is also not a trapdoor function - the encryption algorithm uses randomness, so it is not a function.

How do we use this to construct a public-key encryption scheme?

We have

- $(\text{Gen}(), F, F^{-1})$ our trapdoor function
- (E_s, D_s) a symmetric encryption scheme over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$
- $H : X \to \mathcal{K}$ a hash function

To construct Gen(): we just call Gen() from our trapdoor function.

To construct $E(PK, m)$: we select x randomly from X. Then, we compute $K = H(x)$, and encrypt the message to get $c = E_s(K, m)$. Moreover, we compute $y = F(PK, x)$ and output (y, c).

To construct $D(SK, (y, c))$: we compute $F^{-1}(SK, y)$ to get x. Then, we compute $K = H(x)$ and run $D_s(K, c)$ to get the original message.

Theorem 77. If $(\text{Gen}(), F, F^{-1})$ is a secure trapdoor function, (E_s, D_s) has authenticated encryption, and H is a "random oracle," then (Gen, E, D) is CCA secure.

(Again, we can see that our encryption function here is randomized.) Moreover, the adversary cannot tamper with the message by messing with y - this will just produce a bogus key which means the ciphertext cannot be decrypted (the decryption algorithm will reject with high probability).

Note that we should **never** use an encryption algorithm that just applies F directly to the message - this is deterministic so it is not semantically secure!

Now we turn to the question of how to construct a trapdoor function.

The RSA Trapdoor Function

(This is a good time to go back and review Lecture 9 for relevant facts about modular arithmetic.)

To construct $\text{Gen}()$**:**

- choose random distinct primes p, q that have around 1024 bits
- set $N = pq$
- choose integers e, d such that $ed = 1 \pmod{\varphi(N)}$ (this means $e = d^{-1}$ in $\mathbb{Z}_N$)
- output $PK = (N, e)$ and $SK = (N, d)$

If I give you N, it is difficult to recover p and q - this is the factoring problem.

To construct $F(PK, x)$**:** $F((N, e), x) = x^e \pmod N$.

To construct $F^{-1}(SK, y)$**:** $F^{-1}((N, d), y) = y^d \pmod N$. We can see that

$$y^d = (x^e)^d = x^{ed} = x^{k\varphi(N)+1} = (x^{\varphi(N)})^k x = x \pmod N,$$

where k is an arbitrary integer, from the fact that $ed = 1 \pmod{\varphi(N)}$.

Definition 78. The **RSA assumption** says that RSA with exponent e is a one-way permutation.

Specifically, for any p, q, N defined as above and y randomly selected from $\mathbb{Z}_n$, the RSA assumption is that for all efficient adversaries $\mathcal{A}$,

$$\Pr\left[\mathcal{A}(N, e, y) = y^{1/e}\right]$$

is negligible.

Then, RSA public-key encryption is defined the same way we defined public key encryption for a general trapdoor function:
We have

- $(\text{Gen}(), F, F^{-1})$ our RSA trapdoor function
- (E_s, D_s) a symmetric encryption scheme over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$
- $H : \mathbb{Z}_N \to \mathcal{K}$ a hash function

To construct Gen(): we just call Gen() from the RSA trapdoor function (this gives us N, e, and d).

To construct $E(PK, m)$: we select x randomly from $\mathbb{Z}_N^*$. Then, we compute $K = H(x)$, and encrypt the message to get $c = E_s(K, m)$. Moreover, we compute $y = F(PK, x) = x^e \pmod N$ and output (y, c).

To construct $D(SK, (y, c))$: we compute $F^{-1}(SK, y) = y^d \pmod N$ to get x. Then, we compute $K = H(x)$ and run $D_s(K, c)$ to get the original message.

Lecture 12: RSA Encryption and Digital Signatures

We should generally think of RSA as a specific example of a trapdoor function, rather than thinking about the specific mechanics of how it works.

Remember that RSA used the trapdoor function $F((N,e),x) = x^e \pmod{N}$.
When $e = 1$ or $e = -1$, this is easily invertible, so it is not a trapdoor function. A fun problem is: when $e = 2$, being able to invert f means we would be able to factor N. But $e = 2$ still doesn't work for RSA, because 2 will always be a factor of $\varphi(N)$, so we cannot find $d = e^{-1} \pmod{\varphi(N)}$. So, the smallest valid e is 3.

Many textbooks incorrectly explain RSA as the encryption scheme that has $E((N,e),m) = m^e \pmod{N}$; this is insecure because it is deterministic - never do this!!

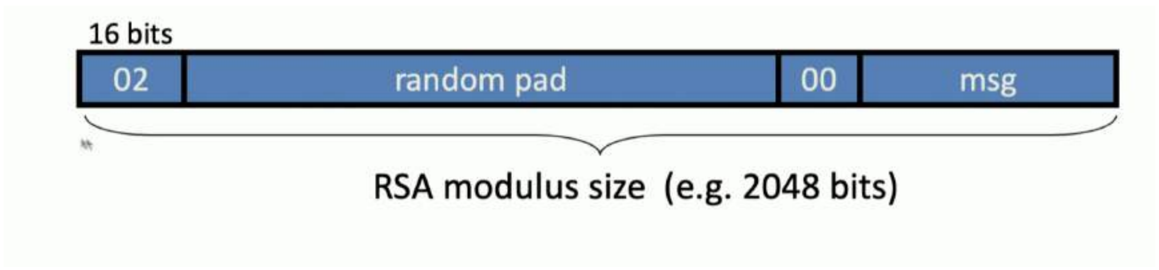
RSA doesn't work super well in practice - it was done before we really understood cryptography, so it is not used under the ISO standard.

RSA encryption in practice

We are trying to use RSA to encrypt and send over a secret key (for the rest of communication to be done via a AES), so when we say "message" for the sake of RSA, it actually refers to the secret key we are trying to encrypt.

We take a 128-bit message, apply some preprocessing to it so that its length is equal to the length of the modulus N, and then we apply RSA to it.
So what is this preprocessing?

In PKCS1 v1.5:



We write a `02`, then a random pad that doesn't contain `00`, then a `00` and then the message. This is the preprocessing used in TLS 1.2. But this is insecure!

Attack on PKCS1 v1.5 (Bleichenbacher 1998)

The way that TLS 1.2, for example, implemented this padding method, it would decrypt a ciphertext, and then check whether the first two bits were `02`. If those were not the first two bits, it would know that this was not a correctly preprocessed message, and it would reject. But this is a problem - from a ciphertext, the attacker can learn whether the plaintext started with `02` or not based on whether the server rejects or not.

Specifically, the attacker has a chosen ciphertext attack where, given a ciphertext c it can pick any $r \in \mathbb{Z}_n$, and then pass in

$$c' = r^e c = (r \cdot \text{PKCS1}(m))^e.$$

This means it can find out whether $(r \cdot \text{PKCS1}(m))$ starts with `02` for any r.

What can it do with this?

Baby Bleichenbacher Attack

Say, instead, that N is a power of 2 and the server actually checks whether the most significant bit of the message is a 1.

Then, the attacker can do the following:

- Send c to get the most significant bit of the message.
- Send $2^e c$ to get the second bit of the message.
- Send $4^e c$ to get the third bit of the message.
- and so on...

so that they reveal the entire message.

It is left to us as a challenge to see how to get from this to the actual Bleichenbacher attack, but it is not that difficult. For a 2000-bit modulus, this would take about 4 million queries for the attacker to reveal the message - this can be done in a couple hours.

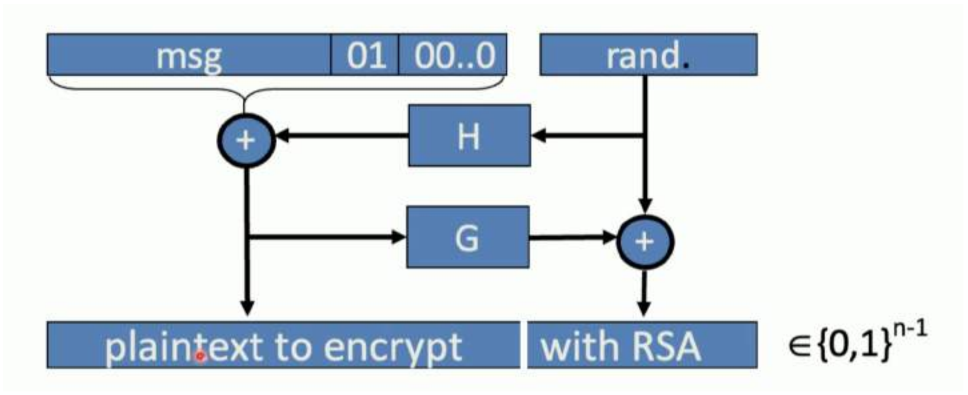
This was a major deal when the attack was discovered! People were sending their credit cards over HTTP servers, and now this was completely insecure.

The fix implemented:

1. Generate a random 46-byte string R.
2. Decrypt the ciphertext to recover the plaintext.
3. If the plaintext is invalid, pretend the message was R and continue.

This means we have a garbage message, so if we try to use it as a key things will probably break later on (which it should, because we are being called by an attacker) but the attacker won't get that information about the message.
(This is pretty ad-hoc, it is used today and there is a paper now showing that this is secure when implemented with the rest of TLS.)

A better version (PKCS1 v2.0: OAEP)
We are not going into the details of OAEP, but this is what the preprocessing looks like:



This is better because it hashes the randomness into the message instead of just incorporating randomness in the padding.

Theorem 79. Assuming RSA is a trapdoor permuation, RSA-OAEP is CCA (chosen ciphertext attack) secure when H and G are random oracles.

In practice, SHA-256 is used as the hash function.

This is not used in practice, because all browsers and all servers need to be changed at the same time. But this is very difficult to do, so we used the previous fix, which only required server-side changes.

There are many subtleties in implementing OAEP. For example, if you have code that looks like the following:

```
OAEP-decrypt(ct):
    error = 0;
    ...
    if(RSA^{-1}(ct) > 2^{n-1})
        {error=1; goto exit;}
    ...
    if(pad(OAEP^{-1}(RSA^{-1}(ct))) != "01000")
        {error=1; goto exit;}
```

You will be vulnerable to a timing attack! Based on whether the program throws an error quickly or slowly, an attacker will be able to tell what the issue with the ciphertext was.
To fix this, simply remove the `goto exit` line from each of the functions, and only exit the program at the end.

There are three main types of side-channel attacks:

- **timing**
  This can leak a lot of information about the message! In CS 155 next quarter, they will discuss timing attacks with a lot of detail.
  Memory attacks usually devolve into timing attacks in terms of how long it takes to read memory, because that tells you whether something is in the cache or not.

- **power**
  The amount of power that we use to encrypt a message can reveal information about the message. This is very difficult to defend against, and most of the industry has given up on doing so.
  We can also leak information through noise - based on the amount of power the processor is using, the frequency of the humming of the power converter changes.

- **radiation**
  There's a company in San Francisco that builds super-sensitive antennae to measure a processor's electromagnetic field as it is computing - this leaks information about the computations! There was a (sort of contrived) demo done a few years ago where electromagnetic analysis from ten meters away revealed a secret key from an iPhone.

Is RSA a one-way permutation?

To invert the RSA one-way function (without d) the attacker must compute an e^{th} root mod N.

After 50 years of studying this problem, the best known way to compute e^{th} roots is still:

1. factor N to get p and q (hard)
2. compute e^{th} roots mod p and q (easy)

This gives you the e^{th} root mod N using the Chinese Remainder Theorem - don't worry about it if you don't know what that is.

Again, factoring will be made easy using a quantum computer, so RSA will be completely dead once quantum computers arise.

Open problem: does a shortcut exist?

To prove there is no shortcut we would need to show that an efficient algorithm for e^{th} roots would imply an efficient algorithm for factoring N. There is some evidence we cannot prove this - (BV '98) says that an "algebraic" reduction from e^{th} roots to factoring would imply that factoring is easy.

How **not** to improve RSA perfomance

To speed up RSA decryption, you could use a small private key d. But ...

Wiener '87: If $d < N^{0.25}$ then RSA is insecure.
BD '98: If $d < N^{0.292}$ then RSA is insecure.
Open question whether RSA is secure when $d < N^{0.5}$.

Wiener's attack
Since $ed = 1 \pmod{\varphi(N)}$, there exists some k such that $ed = k\varphi(N) + 1$. Then,

$$\left|\frac{e}{\varphi(N)} - \frac{k}{d}\right| = \frac{1}{d\varphi(N)} \leq \frac{1}{\sqrt{N}}.$$

But $\varphi(N) = N - p - q + 1$, which is well-approximated by N - specifically, $N - \varphi N \leq 3\sqrt{N}$. Then, if $d \leq \frac{N^{0.25}}{3}$,

$$\left|\frac{e}{N} - \frac{k}{d}\right| \leq \left|\frac{e}{N} - \frac{e}{\varphi(N)}\right| + \left|\frac{e}{\varphi(N)} - \frac{k}{d}\right| \leq \frac{1}{2d^2}.$$

Again, we know e and N, we don't know k or d. But this is a very small error term - small enough that there would be very few satisfying fractions. Thus, using a process called the continued fraction algorithm, we can solve for k/d exactly and since k and d are relatively prime, this gives us d.

This is actually *faster* than the standard RSA process to generate e and d!

Instead, what we do is...

How to improve RSA performance

Use a small e! The standard is $2^{16} + 1$, which would take 17 multiplications to encrypt, but with the right implementation $e = 3$ would work as well.

This is called the asymmetry of RSA: encryption is much faster than decryption. ElGamal does not have this property; it takes about the same time to encrypt and decrypt.

This also means that if you are encrypting something using a very small device - such as an IoT device - RSA encryption is a good bet.

We are using RSA encryption to generate a secret key for another encryption scheme, which means we want our encryption *of* the key to be at least as secure as encryption *using* the key. How big does the modulus have to be to accomplish this? Thus, RSA is just far too expensive to be used in practice, and instead we primarily use elliptic curve-based ElGamal.

Key Size	**Modulus Size For RSA**	**Modulus Size for EC**
80 bits	1024 bits	160 bits
128 bits	3072 bits	256 bits
256 bits (AES)	15360 bits	512 bits

RSA Key Generation Trouble

Heninger et al. and Lenstra et al. downloaded all the N's being used for RSA encryption on the internet, and took the gcd of each pair of them. What they should find, since each N is the product of separate randomly generated p and q, is that for two distinct N, their gcd is 1. Instead, they found that there was a significant number of pairs of Ns whose gcd was nonzero.
But if $\gcd(N, N') \neq 1$ then $\gcd(N, N') = p$ and $q = N/p$ and $q' = N/p'$, which is easily computable. This broke 0.4% of all HTTPS public keys.

What happened was: there were many small devices (such as IoT devices) that were trying to generate an RSA public key at startup. But their random number generator didn't have enough entropy the first time it was called after startup, so the ps generated were with the same seed, and ended up being the same.

Now we are moving onto ... digital signatures

The big picture: one-way functions

Definition 80. A **one-way function** $f : X \to Y$ is a function such that:

- there exists an efficient algorithm to evaluate f
- for all efficient adversaries $\mathcal{A}$,
$$\Pr[f(\mathcal{A}(f(x)) = f(x)]$$
is negligible when x is selected randomly from X

This means not only that the adversary can't get back x from $f(x)$, but also that they can't get back any preimage of $f(x)$.

One Way Functions we have seen so far:

1. general one-way functions:
Let (E, D) be a block cipher. Then,
$$f^E(K) = \big(E(K,1), E(K,2), \ldots, E(K,10)\big),$$
where $f^E : \mathcal{K} \to \mathcal{C}^{10}$ (or some arbitrary number). This is a one-way function because (E, D) is secure. This has no special properties - in particular, it is bad for a key exchange.

2. discrete log:
When G is a finite cyclic group of order q and $g \in G$ is a generator,
$$f^{\text{Dlog}}(x) = g^x \in G$$
where $f^{\text{Dlog}} : 2q \to G$. Inverting this function is hard when Dlog in G is hard.
properties: $f(x+y) = f(x)f(y)$, $f(x)^\alpha = f(\alpha x)$ for $\alpha \in \mathbb{Z}$
Those properties are what allow Diffie Helman and ElGamal key exchange.

3. RSA:
   If we have $N = pq$, $e \in \mathbb{Z}^*_{\varphi(N)}$,
   $$f^{\text{RSA}}(x) = x^e \text{ in } \mathbb{Z}_N$$
   where $f^{\text{RSA}} : \mathbb{Z}_N \to \mathbb{Z}_N$. Inverting this is hard when the RSA assumption holds.
   properties: $f(xy) = f(x)f(y)$, RSA has a trapdoor
   These properties allow RSA encryption and RSA digital signatures (as we will see soon)

Digital Signatures

We want a digital signature to be evidence that you wrote/approved a message. Since on the internet we can copy signatures between documents easily, we need the digital signature to be a function of m.

We want a pair (S, V) where the signer gets a message and a secret key and outputs a signature σ, and the verifier gets a message and a signature and uses a public key to output `yes` or `no`.

This is the same as the process for MACs, except it uses an (SK, PK) pair instead of a single shared key.

Definition 81. A **signature scheme** is a triple of algorithms $(\text{Gen}(), S, V)$ where:

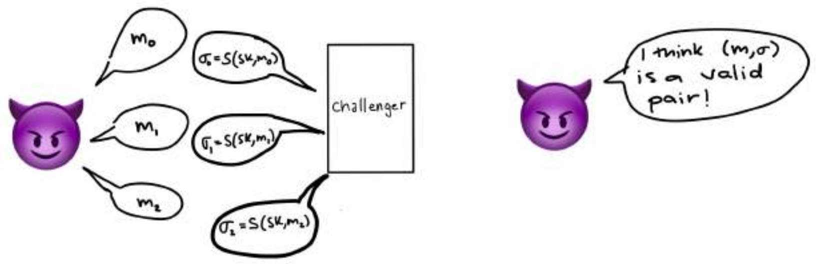
- $\text{Gen}()$ is a randomized algorithm that outputs (PK, SK)
- $S(SK, m) \to \sigma$ is a randomized algorithm that outputs a signature
- $V(PK, (m, \sigma))$ is a deterministic algorithm that outputs `yes` or `no`

and for all (SK, PK) outputted by $\text{Gen}()$ and all messages m,
$$V(PK, (m, S(SK, m))) = \texttt{yes}.$$

The signer signs the message once, and anyone with the public key and (m, σ) can verify that it has been signed correctly.

Definition 82. Security for a signature scheme (Gen, S, V) is defined using the following game:

First, the challenger calls $\text{Gen}()$ to generate a public key and secret key, and sends PK to the adversary. Then, the adversary sends as many messages m_i as they want, and the challenger will return $\sigma_i = S(SK, m_i)$. Then, the adversary outputs a pair (m, σ) and wins if $m \neq m_i$ for any i and $V(PK, (m, \sigma))$ accepts.

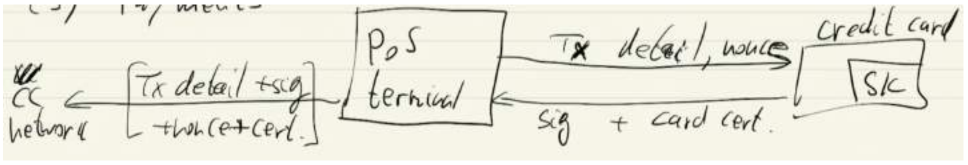


A signature scheme is **secure** if for all efficient adversaries $\mathcal{A}$, the probability that the adversary wins the security game is negligible.

Lecture 13: Certificates

Many Applications of Signatures:

1. Android phones ship with a built in public key. All software updates are signed with the secret key, so the phone can verify whether the updates are coming from a trusted source.
2. Certificates (this is what we will discuss today):
   The certificate authority (CA) creates a signed message:
   $m =$"Bob's public key is PK", $S(SK_{CA}, m)$
   Then, we can see that anyone who trusts the certificate authority can get Bob's public key and trust that it is not a bogus public key.
3. payments:
   Your credit card contains a secret key SK. When you swipe/tap/etc your credit card on a point of sale terminal, the terminal sends the transaction details and a nonce (recall that a nonce is a non-repeating value) to your card, and the credit card responds with its signature on the transaction details and its certificate. Finally, all of this is sent to the credit card network.



So far, we have seen three approaches to data integrity:

1. collision-resistant hash:
   This requires a read-only public space to store the hash, but anyone can verify it.
2. digital signature:
   Again, there is one signer and many verifiers, but they all need the public key.
3. MACs (message authentication codes):
   This requires a shared secret key, so it is generally used with one signer and one recipient.

Extending the Domain of a Signature Scheme

Let $(\text{Gen}(), S, V)$ be a signature scheme for short messages; say, $\mathcal{M} = \{0,1\}^{256}$. How do we extend this to a signature scheme for long messages?

Let $H : \mathcal{M}^{\text{big}} \to \mathcal{M}$ be a collision-resistant hash function (such as SHA-256).

Then, like with MACs, we can define

$$S^{\text{big}}(SK, m) = S(SK, H(m))$$
$$V^{\text{big}}(PK, m) = V(PK, H(m), \sigma).$$

Theorem 83. If $(\text{Gen}(), S, V)$ is secure and H is a collision-resistant hash, then $(\text{Gen}(), S^{\text{big}}, V^{\text{big}})$ is also secure.

Thus, it suffices to build a signature scheme for 256-bit messages and use SHA-256 to expand it.

As we discussed at the end of last lecture, we can build signature schemes from different primitives.

Primitives that Imply Secure Signatures

1. general one-way functions
   Surprisingly, we **can** build signature schemes from general one-way functions. An example is Lamport-Merkle.
   Problem: signatures are long ($\geq$ 30KB for a stateless signer, $\geq$ 4KB for a *stateful signer*)
   These are suitable for software update, and they are quantum resistant!

2. discrete log
   We **can** build signature schemes from discrete log. Examples include ECDSA, Schnorr, and BLS.
   The signature size is 48 or 64 bytes and the public key size is 32 or 48 bytes.
   These will not be described until we discuss zero-knowledge proofs, so that the math makes sense to us ☺
   (Schnorr is a better signature scheme, but ECDSA was not patented at first, so ECDSA is being used more generally.)

3. trapdoor permuations (RSA)
   It is very easy to build a signature scheme from RSA - this will be our first example.
   These are being phased out - the discrete log-based signature scheme is much better.

(A *stateful signer* is a signer that keeps some sort of state information each time it signs a new message - if it reuses an old state, then all security is lost.)

Generating a Signature Scheme from a Trapdoor Permutation

We have a trapdoor permuation $(\text{Gen}, F : X \to X, F^{-1} : X \to X)$ and a hash function $H : \mathcal{M} \to X$.

Then, the signature scheme is:

- Gen(): the Gen() from the trapdoor permutation
- $S(SK, m)$: output $F^{-1}(SK, H(m))$
- $V(PK, m, \sigma)$: output `yes` if $F(PK, \sigma) = H(m)$, otherwise output `no`

Theorem 84. The triple $(\text{Gen}(), S, V)$ is a secure signature scheme assuming $(\text{Gen}(), F, F^{-1})$ is a secure trapdoor permutation and H is a random oracle.

This is a very popular design, when used with RSA it is called:

RSA-FDH (full-domain hash)

This requires a hash function $H : \mathcal{M} \to \mathbb{Z}_N$. Then, we can construct the following signature scheme:

- Gen(): choose $N = pq$ and e, d such that $ed = 1 \pmod{\varphi(N)}$
  output $PK = (N, e)$ and $SK = (N, d)$
- $S(SK, m)$: output $\sigma = (H(m))^d \pmod N$
- $V(PK, m, \sigma)$: output `yes` if $\sigma^e = H(m)$ in $\mathbb{Z}_n$, and `no` otherwise

Problem: the range of H depends on the public key. But this is not that bad - in practice for the hash functions we use we can take the output of H mod N and still have collision-resistance.
Note: We can take $e = 3$, and then we have a superfast verification function.

Why do we hash?

There's only ever one answer to this question - if we don't it's insecure.

If we had a signature scheme based on $S(SK, m) = m^d \pmod N$, then we are vulnerable to the following attacks:

Attack 1: an existential forgery just given PK
Step 1: choose $\sigma \in \mathbb{Z}_n$
Step 2: choose $m = \sigma^e \pmod N$
Step 3: output (m, σ) as a forgery
We can see that V will just check if $\sigma^e = m$, which is true by definition.

Attack 2: also an existential forgery given PK
We want to compute the signature for a message m.
Step 1: choose r randomly from $\mathbb{Z}_N$ and compute $\hat{m} = r^e m \pmod N$.
Step 2: Request a signature on $\hat{m}$. This gives us $\hat{\sigma}$, where $\hat{\sigma}^e = \hat{m}$ in $\mathbb{Z}_N$.
Step 3: output $(m, \hat{\sigma}/r)$
We can see that V will check if $(\hat{\sigma}/r)^e = m$ in $\mathbb{Z}_n$. But

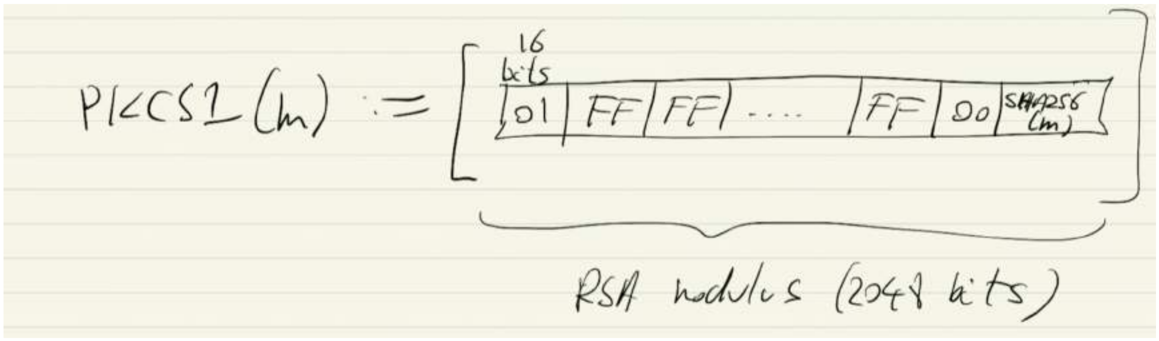
$$(\hat{\sigma}/r)^e = \hat{m}/r^e = (r^e m)/r^e = m.$$

So this is a valid forgery.

Attack 2 gives us a **blind signature**; we can get someone to sign a message without knowing what message they are signing. This is very useful - for example, in a voting process, the authority could sign and give someone a voting ticket without knowing which ticket they gave the person, so each person can vote anonymously but nobody can vote twice.

RSA in practice: PKCS1 v1.5

This uses a hash function PKCS1: $\mathcal{M} \to \mathbb{Z}_N$, which is defined as follows:



Then, our signature is $\sigma = (\text{PKCS1}(m))^d \in \mathbb{Z}_N$.

This is called a partial-domain hash because it hashes only to a specific fraction of $\mathbb{Z}_N$. Thus, we don't know how to prove that this is secure, but it is what is used in practice.

Puzzle: What if the pad was all zeroes, instead of containing the FF's?
Then, this would be insecure because (for example when $e = 3$) in order to compute the signature of a message, we just need to compute the cube root of a small number. For example, if SHA-256(m) ends up being a perfect cube, it would be fairly easy to compute the signature of PKCS1(m). Moreover, there is an algorithm for computing cube roots of small numbers that works faster than factoring.

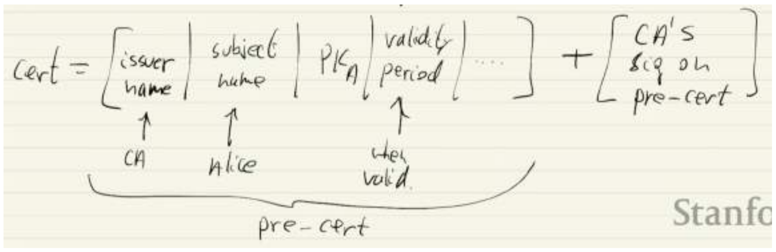
Public Key Management: certificates

How does Bob obtain Alice's PK_A? If Bob gets the wrong public key, then he would trust signatures that are not from Alice.

Single-Domain Certificate Authority

Alice generates a (SK_A, PK_A) pair for herself. She goes to the certificate authority (CA) and has to prove to the authority that she is Alice, then tell the authority PK_A. (The "prove she is Alice" step is usually dependent on a real-world mechanism, such a passport.) Then, the authority sends a certificate to Alice.

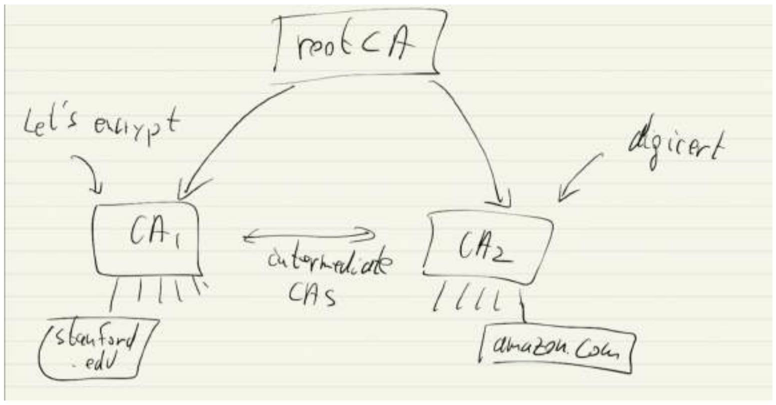
The certificate binds PK_A to the physical identity "Alice."



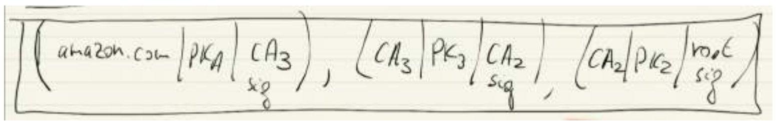
Then, Alice sends Bob the certificate. He can verify the certificate (because he knows the CA's public key), and then he knows PK_A.

This is a lot better than the trusted third party in the symmetric world, because the CA has no one's secret key.

In reality, we have root CA's and intermediate CA's.



So then, the certificate for Amazon would actually be a lot longer than just having a signature from the root CA, since the root CA didn't actually verify Amazon's public key. Instead, it would look like this:



Since everything is dependent on the root CA secret key, the root CA secret key is typically stored offline.
There are about 60 root CA's and 1200 intermediate CA's in the world.
For next time: what happens if one of the intermediate CA's decides to go rogue and make and sign their own proxies?

Lecture 14: Identification Protocols

Certificate Revocation

What happens when someone's private key is stolen, and we need to revoke their certificate?
(A couple years ago, there was a bug in the Apache web server, which revealed around a million secret keys, and those certificates needed to be revoked all at once.)

1. **expiration date:**
   Every certificate is only useful until its expiration date, so if a secret key is stolen, it is only tied to a valid certificate for a limited amount of time. But certificates typically expire after a year, so there is still a lot of time to cause damage before they expire.

2. **CRL (certificate revocation list) set:**
   This is shipped daily from Google to Chrome, Firefox, and etc. Then, the browser can check whether the serial number for your certificate is on the revocation list, and consider it invalid if it is. (If you realize your private key has been compromised, you fill out a form to add your certificate to the revocation list via offline methods.)
   These are more difficult to block because they are being sent out by Google, so you can't prevent access to the CRL set without also blocking all traffic to Google.

3. **OCSP (online certificate status protocol):**
   This is an alternative to the CRL set that means the browser does not have to download data for all revoked certificates in the world. Each certificate comes with an OSCP link, where the browser can ask an "OSCP responder" (created by the CA) whether or not a given certificate is valid. The responder gives a yes or no response, but it is signed by the responder so that the browser knows it is accurate. But there are multiple problems with this:

   - The OSCP responder often goes offline and does not respond. Because the browser does not want to completely block traffic because the responder is offline, they let traffic through when there is no response, which means that an authoritarian regime could use revoked certificates simply by blocking the responder.
   - This is also a privacy violation, because the OSCP responder (and therefore the CA) now knows what websites you are trying to visit at what times.
   - This significantly slows down the browser if the OSCP responder is sometimes slow.

   Because of these problems, OSCP is on its way out, and people primarily use the CRL set (which is being optimized).

4. **Short-Lived Certificates:**
   This is an idea that Professor Boneh has been trying to push for a while, but has not been widely adopted. The idea is that certificates expire after around four days, rather than a year or three months. The main issue with this is that it would require the CAs to be more active and that websites would have to be a lot more active in asking for a certificate renewal every four days automatically. (It's another thing that would have worked better if we had started with it in the first place, but now is harder to switch into.)

Negligent CAs:

What happens if a CA is negligent or compromised?

In 2011, there was a CA called DigiNotar that was hacked, and the adversary was able to get certificates for URLs like `*.google.com` and `gmail.com`, which means it could essentially pretend to actually be those

sites, and your browser would trust it because they had valid certificates.
As a consequence, the CA became untrusted by all browsers (certificates from that CA were no longer considered valid) and the company went out of business.

But since then, this has happened several times a year, where sometimes a CA issues bogus certificates. What to do in this case is still a big research area, but there are two solutions that took hold:

1. **pinning**:
   The browser ships with a list of allowable CAs for some domains (for example, `gmail.com` is pinned to the GTS CA, so a certificate for `gmail.com` from a different CA is considered invalid).
   If a browser receives a certificate from an invalid CA, it rejects the certificate and sends a report to one of the pinned CAs that this other CA has been compromised.

2. **certificate transparency (CT)**:
   The idea is that there are a handful of public CT logs, and each time a CA generates a new certificate, it publishes that certificate on two of the logs. It then receives an SCT (signed certificate timestamp) from each of the logs, and includes that in the certificate. Browsers only accept certificates that have two valid SCTs in them. Then, the site admin for each website periodically scans the CT logs to make sure there are no bogus certificates issued for its website.

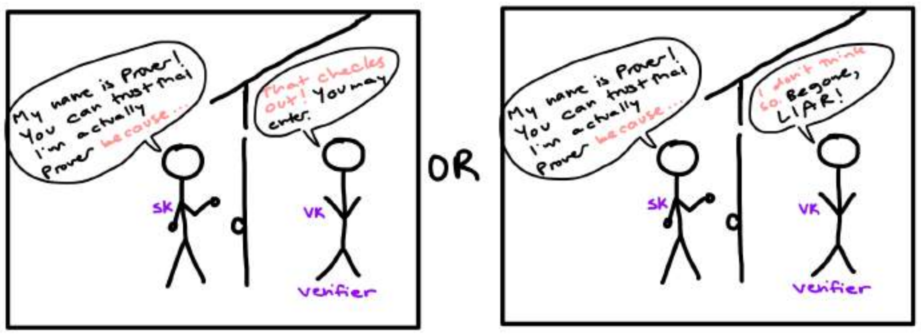
This is the end of Part 2 of the course.

Now we are on part 3 of the class, which is about protocols.

We begin with ID protocols.

Definition 85. The setup of an **ID protocol** is the following:

- A setup algorithm G outputs a pair (SK, VK), giving SK to the prover and VK to the verifier.
- SK is a secret key belonging to the prover, and VK may or may not be secret, depending on the protocol.
- The prover, using SK, sends a single message to the verifier.
- The verifier outputs `accept` if the prover has shown they are the correct person, and `reject` otherwise.


My name is Prover! You can trust that I'm actually Prover because...
That checks out! You may enter.
SK
VK
verifier
OR
My name is Prover! You can trust that I'm actually Prover because...
I don't think so. Begone, LIAR!
SK
VK
verifier

Examples: (these are called friend-or-foe applications)

- using a physical key to enter a locked door

- using a key fob to remotely unlock your car
- using your card to try to access your bank account at an ATM
- using a password login to tell a website who you are, after connecting via a protocol like HTTPS

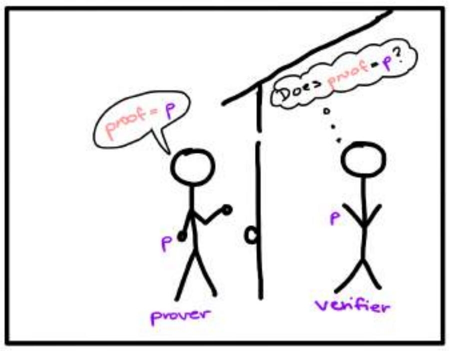
For an ID protocol, there are three levels of security:

1. **direct attacker:** This is an attacker that has no information about the prover. They may have VK if VK is public. For example, an attacker that breaks into a locked door using a lock pick.
2. **eavesdropping attacker:** This is an attacker that listens to a few conversation between the prover and the verifier, and then tries to convince the verifier. For example, we can think of an adversary that listens to the signals sent by your car fob when you are remotely unlocking your car, and then tries to use that to unlock your car door.
3. **active attacker:** This is an attacker that queries the prover and then tries to impersonate the prover. For example, an attacker that puts up a fake ATM at a shopping mall, and then uses the data collected from the fake ATM to try to connect to someone's bank account at a real ATM.

Basic Password Protocol v0:

If PWD is some finite set of passwords, then we can have the following password protocol:

- The setup algorithm G picks a password p randomly from PWD and sets $SK = VK = p$. (Here, VK is also secret.)
- The prover sends over SK.
- If the verifier gets a message that equals p, they accept, and otherwise they reject.



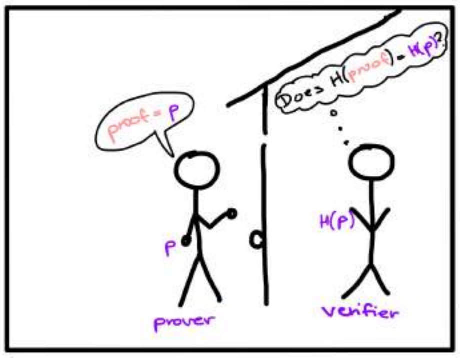
Problem with this protocol: The password VK must be kept secret. If it is stored in the clear on the server, then if the server is ever compromised, all the passwords are revealed. So, we have...

Basic Password Protocol v1:

If PWD is some finite set of passwords and H is a one-way hash function, then we can have the following password protocol:

- The setup algorithm G picks a password p randomly from PWD and sets $SK = p$ and $VK = H(p)$. (Here, VK is also secret.)

- The prover sends over SK.
- The verifier computes the hash of the message they receive. If it equals VK, they accept, and otherwise they reject.



This means the database only needs to store hashed passwords, so if the database is revealed the passwords are not automatically revealed, and because the hash function is a one-way function, an adversary does not get the passwords from their hashes.

Problem with this protocol: weak passwords
The top 10 most used passwords, as of 2018, are:

1. `123456`
2. `password`
3. `1223456789`
4. `12345678`
5. `12345`
6. `111111`
7. `1234567`
8. `sunshine`
9. `querty`
10. `iloveyou`

And a dictionary of $360,000,000$ words covers about 25% of the passwords used on the internet.

About 3% of people use `123456` as their password. This means an adversary that has a list of usernames for the site and tries the **online dictionary attack**, where they go through their list and try `123456` as the password for each username, will get the password in an average of $\boxed{\text{33 tries.}}$

This problem can be mitigated by things like IP-based rate limiting, where they limit the amount of login tries per second from the same IP address. But there is an even worse problem!

Offline-based dictionary attack

If the attacker gets a single $VK = H(p)$ from the server, they can run through their entire dictionary and hash each word until they find a match (without interfacing with the server). This takes $O(|\text{dictionary}|)$.

For example, there is a tool called John the ripper that will scan through all possible 7-letter passwords and then the 360,000,000 word dictionary in a few minutes, and recover 23% of passwords from their hash.

But, if the attacker gets the *entire* table of hashed passwords, they can attack everyone's password at once! For example, in 2012, LinkedIn's table of 6 million passwords, which were hashed using SHA1, was exposed.

In 6 days, 90% of the passwords were exposed using this attack.

What the attacker does is the following: For every word w in the dictionary, check if $H(w)$ appears anywhere in the table. Using fast lookup, this check takes $O(1)$ time, so revaling the passwords of the entire dictionary takes $O(|\text{dictionary}| + |\text{table}|)$, which is very fast.

To fix this, we have:

Basic Password Protocol v2: using a public salt

If PWD is some finite set of passwords and H is a one-way hash function, then we can have the following password protocol:

- The setup algorithm G picks a password p randomly from PWD and a random salt S. Then, they set $SK = p$ and $VK = S, H(S, p)$. (Here, VK is also secret.)
- The prover sends over SK.
- The verifier computes $H(S, p)$. If it equals VK, they accept, and otherwise they reject.

This means that now the offline dictionary attack runs in $O(|\text{dictionary}| \cdot |\text{table}|)$, because now the hash of each password is different for each user, since each user has their own public salt. This means the adversary has to recompute all the hashes with the given user's salt, which slows down their attack.

To hash the passwords, we want to use a **slow**, **space-hard** hash function, so that the dictionary attacks cannot run quickly.

For a slow hash, we can use hashes like PBKDF2, which essentially runs SHA-256 multiple times on the password (it hashes the password and the salt, then hashes the output of that, then hashes the output of that, and so on). We use enough iterations of SHA-256 that it takes, for example, 0.1 seconds to verify.

Since the server is only hashing one password, the server is not significantly slowed down when verifying a user's password, but since someone running the dictionary attack is hashing millions of passwords, this slows them down.

The problem is: the server trying to verify the user's password is probably running on an Intel processor. Here, most of the processor is dedicated to cache, and not to trying to compute SHA-256. An attacker could build a custom ASIC processor that is just devoted to computing hashes, and it would run much faster (on the scale of 50,000 times faster). (The bitcoin mining industry taught us how to do this.)

A better approach is scrypt, which is a hash function that takes a lot of time and a lot of memory to evaluate. This means that a custom processor that is built just for compute speed would be limited by the amount of memory it has, and it is difficult to build a custom processor that would compute these hashes significantly faster than the commodity CPUs.

This was all security against a direct attacker. We will now look at security against eavesdropping.

In this model, the adversary is given VK (if it is public) and a transcript of interactions between an honest prover and the verifier. They are trying to impersonate the prover (convince the verifier that they are actually the prover.)

Prover: Here's my proof.
Verifier: OK!
DAY 2
Prover: Here's my proof
Verifier: OK!
DAY 3
Prover:
proof
Eve
My name is Prover! You can trust me! I'm actually Prover because...
Eve
VK
verifier

Clearly, our previous password protocol was insecure here! Instead, we consider a:

<u>One-Time Password Protocol</u>

This uses a pseudorandom function F (such as AES).

- The setup algorithm G picks a random key k. Then, they set $SK = (k, 0)$ and $VK = (k, 0)$. (Here, VK is also secret.)
- The prover sends over $F(k, i)$ (where i is initialized to 0 and increments over time).
- The verifier computes $F(k, i)$ (they store their local copy of i and also increment over time). If this matches what the prover sent, they accept. Otherwise, they reject.

Then, an eavesdropper who knows $F(k, i)$ for any set of i's cannot compute $F(k, i + 1)$ without knowing both k and i, so they cannot imitate the prover.

Theorem 86. If F is a secure pseudorandom function then this protocol is secure against eavesdropping.

For example, the little RSA SecurID key uses AES-128 on a 128-bit key and a 32-bit counter, which increments every sixty seconds.
(The time-based one-time password (TOTP) won over the user-input based one, which would increment every time the user pressed a button, because there were more problems with the user-input based one.)

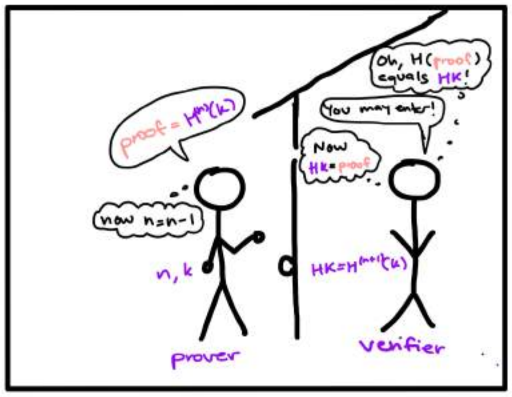
The verification key *has* to be kept secret, because it contains the user's secret key and the counter, so if it is compromised, the prover can be spoofed directly.

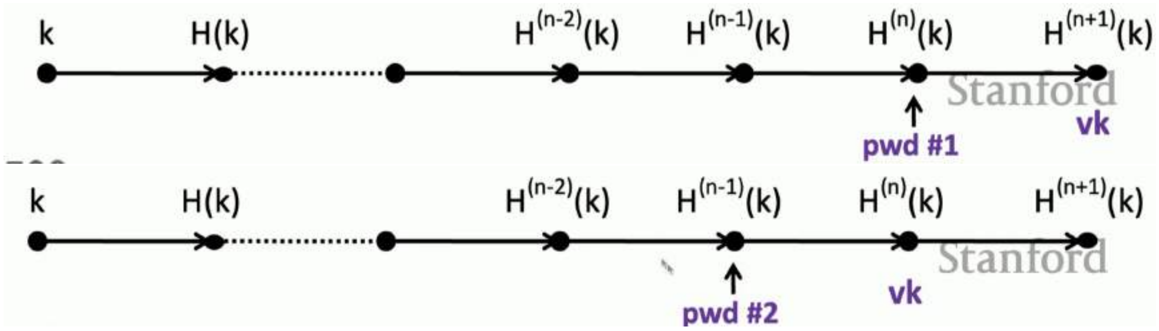
<u>The S/Key system</u>

This is a system (which is not used today) which generates one-time passwords without needing a secret verification key.

We have a hash function H, where $H^{(n)}(x)$ means we hash x, and then hash the output, and then hash that output, n times.

- The setup algorithm G picks a random key k. Then, they set $SK = (k, n)$ and $VK = H^{(n+1)}(k)$
- The prover sends over $H^{(n)}(k)$ and then decrements n.
- The verifier computes the hash of the input. If that matches VK, they accept and set VK to be the input, and otherwise they reject.





We can see that even if we steal VK, we can see that VK is the hash of the next password, so since we can't get the password from the hash, the adversary still can't imitate the prover.

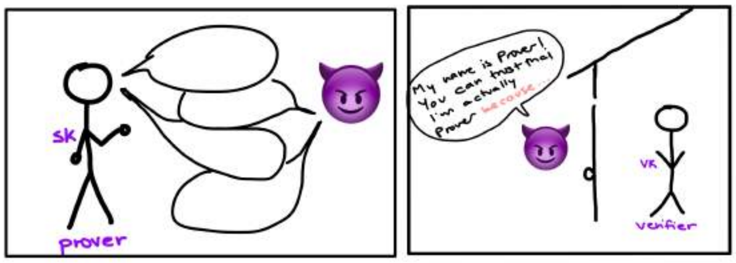
Theorem 87. If the hash function H is a one-way function even when applied n times, S/Key is secure against eavesdropping even if the eavesdropper gets access to VK.

But SecurID has an unlimited number of authentications, and the key being sent over is much shorter, so it is used over S/Key.

Lecture 15: Authenticated Key Exchange

Last class, we learned to protect against a direct attacker and an eavesdropping attacker. Today, we will learn to protect against active attacks.

Definition 88. In a **active attack** on an ID protocol, an adversary can query the prover as many times as they want. Then, once it is done, it begins talking to the verifier, and it is successful if it is able to convince the verifier that it is actually the prover.


SK
prover
My name is Prover! You can trust me! I'm actually Prover because
VK
verifier

Note that these are two separate steps; the adversary only starts talking to the verifier once it is *done* talking to the prover.

Example 89. A real-world example of this is if a scammer puts up a fake ATM in a shopping mall, then uses the credit card data collected from the fake ATM to try to access someone's real bank account.

Online, phishing sites are an example of this. They present a fake password login page to the user (prover) and once the prover has entered their password data, the adversary can use that to log into the prover's account in the actual website.

All basic password protocols we have discussed so far are vulnerable to active attacks.

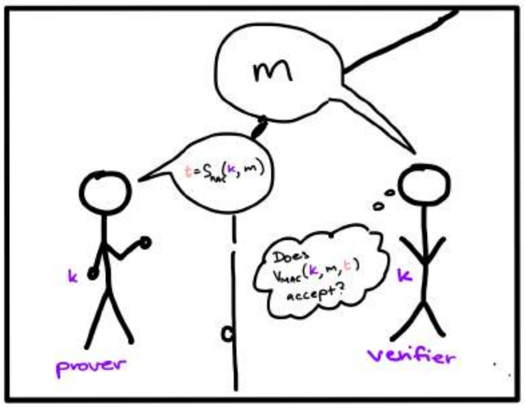
We will now look at protocols that are not vulnerable to these attacks:

MAC-based challenge response

This is a protocol that is secure against active attacks but requires the verifier to have a secret VK.

We have a message authentication scheme $(S_{\text{MAC}}, V_{\text{MAC}})$, with keyspace $\mathcal{K}$ and message space $\mathcal{M}$. Then, we do the following:

- The setup algorithm G picks a random key $k \in \mathcal{K}$. Then, they set $SK = k$ and $VK = k$.
- The verifier sends over a random message $m \in \mathcal{M}$, which we call a *challenge*.
- The prover sends back $t = S_{\text{MAC}}(k, m)$.
- The verifier computes $V_{\text{MAC}}(k, m, t)$, and accepts if V_{MAC} accepts, and otherwise rejects.



Theorem 90. This protocol is secure against active attacks given that VK is secret, $(S_{\text{MAC}}, V_{\text{MAC}})$ is a secure MAC, and $\mathcal{M}$ is bigger than $\{0,1\}^{128}$.

This is because we can see that an adversary who queries the prover a bunch of times just gets the tag for a bunch of messages, and by MAC security as long as the verifier prompts them with a message they have not seen before, they will not be able to generate a valid tag for it.

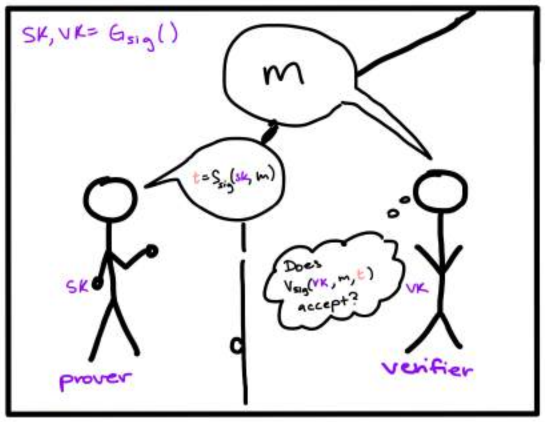
We can see that if the verification key is revealed, then the adversary knows k for the MAC, and then they can easily impersonate the prover. For similar reasons, this is less secure when k is a human-selected password, because then it is vulnerable to the dictionary attack on k.

But, it has the benefit of allowing a short m and t. A CryptoCard is a physical card where the prover gets a challenge m, enters it into the card, and the card gives them a response t which they enter into their terminal to be let in. This uses only 8-character long messages and tags.

In order to allow for a public VK, we have the Signature-Based Challenge Response.

We have a signature scheme $(G_{\text{sig}}, S_{\text{sig}}, V_{\text{sig}})$, with message space $\mathcal{M}$. Then, we do the following:

- The setup algorithm G calls G_{sig} and outputs the relevant SK and VK, where VK is the public key for the signature scheme.
- The verifier sends over a random message $m \in \mathcal{M}$, which we call a *challenge*.
- The prover sends back $t = S_{\text{sig}}(SK, m)$.
- The verifier computes $V_{\text{sig}}(VK, m, t)$, and accepts if V_{sig} accepts, and otherwise rejects.



Theorem 91. This ID protocol is secure against active attacks given that our signature scheme is a secure signature scheme and $\mathcal{M}$ is bigger than $\{0,1\}^{128}$.

But, we can see that the tag t can no longer be as short (it must now be along the lines of 20 bytes). This is because the VK for a signature scheme is public, so if the tag space is small, the adversary can brute-force all possible tags and check it themselves using the verification algorithm.

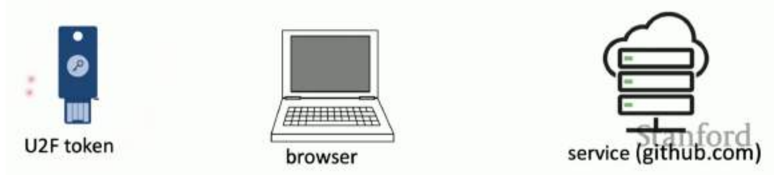
This is currently implemented in ...

The U2F Standard (and WebAuthn)

(Here, U2F stands for universal second factor).

We have a U2F token talking to a service, and their communication goes through a brower. Our goals are:

- the browser malware cannot steal user credentials
- U2F should not enable tracking users across sites
- U2F uses counters to defend against token cloning



The U2F protocol has two parts. First, a device gets registered with the service:

- The U2F token has a secret key SK that it keeps across websites and interactions.
- The service sends over a challenge m and an ID (such as the site URL), through the browser.
- The U2F token uses SK and ID (and some sort of PRF) to generate SK_{ID} and PK_{ID}, as the key pair for a signature scheme.
- The U2F token sends PK_{ID}, $S(SK_{ID}, m)$, and a handle to the service through the browser.

- Then, the service registers your handle with PK_{ID} (i.e. stores it in its databases so it recognizes you as a user next time)

What is the handle for?

The reason there is a new public key for each website is to prevent tracking across websites, because if you had a public key that stayed constant across websites, it would be very easy to tell what sequence of websites you were registered for.

But the small devices don't have enough memory to store a separate key for each website you need to visit. So instead they are stored on the server - each handle is actually an encryption of SK_{ID} using SK, so that if the U2F token gets back the handle, they can re-gain SK_{ID}.

(Though SK_{ID} was computed from SK and ID, this process used some amount of randomness or auxiliary information, so the handle could also just contain this information. The point is primarily that it contains the information so that someone with SK can compute SK_{ID}, but no one else can.)

Then, when you want to authenticate yourself to the server, you do the following:

- The service sends over the ID, a challenge m, and the handle, to the U2F token through the browser.
- The U2F token uses this to recompute SK_{ID} and sends back $S(SK_{ID}, m)$ and a counter.
- Then, the service can verify that this is a valid signature, using PK_{ID}.

This is publicly available as WebAuthn, and you should use this on websites that you are making. Primarily, the users who have U2F tokens are employees at larger companies, because end-users are mostly unwilling to buy their own U2F token.

This is the conclusion of our section on ID protocols, which we should think of as protocols that are useful when the adversary can*not* interact with the user *during* the impersonation attempt.

Now, we are going to talk about authenticated key exchange protocols.

Again, we want Alice and Bob to generate a shared key for the sake of communication. The protocols we saw so far were secure against eavesdropping, but not against a person-in-the-middle attack.

An Authenticated Key Exchange is supposed also to be secure against active adversaries.

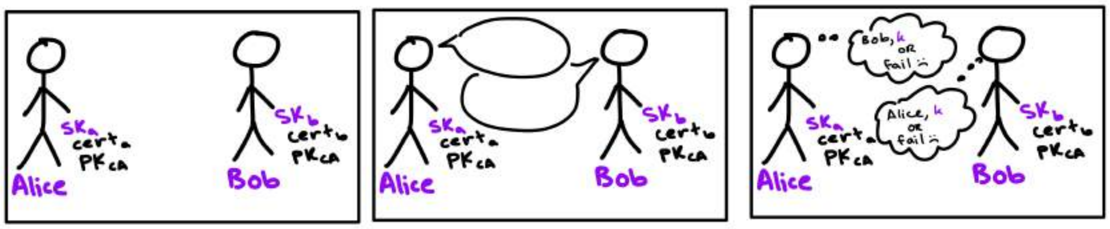
Definition 92. An **active adversary** is an adversary that has complete control of the network. This means they can modify, inject, and delete packages on their way from one user to another.

Moreover, in this setting, we also have corrupt users which are controlled by the adversary, and we want to make sure that doing a key exchange with a corrupt user does not compromise the security of a key exchange with a not-corrupt user.
(If Alice does a key exchange with Mr. EvilMan before she does a key exchange with Bob, her key exchange with Bob should not be compromised because of her previous conversation.)

All authenticated key exchange (AKE) protocols require a trusted third party to certify user identities. We talked about this in the previous lectures: there are two types of trusted third parties.
Offline trusted third parties are only contacted during registration and revocation, and we called them Certificate Authorities.
Online TTPs are part of every key exchange protocol and end up knowing every symmetric key, so they are not widely used.

The Syntax of an AKE

We have Alice and Bob. Alice has SK_a, cert$_a$, and PK_{CA} (the public key of the certificate authority). Bob has SK_b, cert$_b$, and PK_{CA}. Then, at the end of the protocol, either Alice has (k, Bob), as in she has the shared secret key and knows she's talking to Bob, or she has `fail`. Similarly, Bob now either has (k, Alice), or he has `fail`.



The following is an informal definition of AKE security:

Suppose Alice successfully completes AKE and gets (k, Bob). If Bob is not corrupt, then:

Definition 93. If the AKE protocol has **authenticity** for Alice, either Bob received the shared key k or no one received it. (The key could not have been shared with anyone else.)

Definition 94. If the AKE protocol has **secrecy** for Alice, then the shared key k looks indistinguishable from random for the adversary, even if the adversary has seen shared keys that Alice generated with other people or shared keys that Bob generated with other people.

Definition 95. If the AKE protocol has **consistency**, then if Bob successfully completes AKE it gets (k, Alice).

But, there's actually three levels of core security. These are the following:

Definition 96. An AKE has **static security** if it has authenticity, secrecy, and consistency.

Definition 97. An AKE has **forward secrecy** if it has static security, and moreover if the adversary learns SK_a at time T, then it cannot learn anything about Alice's messages sent before time t.

Definition 98. An AKE has **HSM security** (where HSM stands for Hardware Security Module) if it has forward secrecy and if an adversary queries an HSM and gets SK_a n times, then at most n sessions are compromised. (Having the secret key from n different timestamps does not reveal information about the secret key at other timestamps.)

There is also a variance of AKE where only one side has a certificate:

The Syntax of One-Sided AKE

We have Alice and Bank. Alice has VK_{CA} (the public key of the certificate authority). Bank has SK_b, cert$_b$, and VK_{CA}. Then, at the end of the protocol, either Alice has (k, Bank), as in she has the shared secret key and knows she's talking to Bank, or she has `fail`. But Bank does not learn Alice's identity - it now either has k, or it has `fail`.

Warning: These AKE protocols are extremely fragile. Do not try to design it yourself. Instead, just use the latest version of TLS.

Building Blocks of AKE

The certificate cert_b has PK_b, and the bank has SK_b.

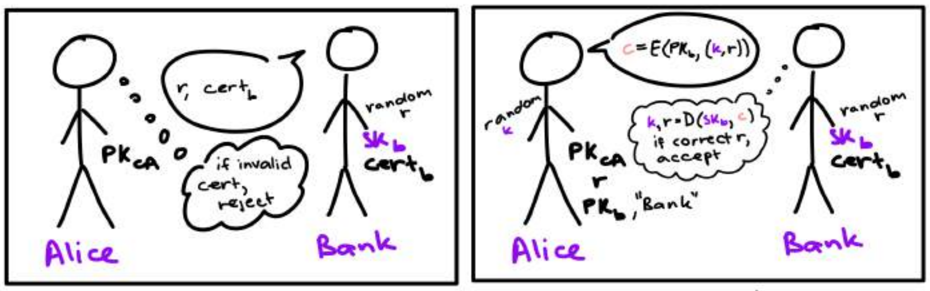
We also have an encryption algorithm $E_b(m, r) = E(PK_b, m, r)$ where E is chosen-ciphertext secure.

And, we have $S_a(m, r) = S(SK_a, m, r)$ where S is a secure signature scheme.

Finally, we have the nonce space R, which is a large set, such as $\{0, 1\}^{256}$.

Then, we can have the following simple one-sided AKE protocol:

- The Bank sends over r, which is sampled randomly from R and cert_b.
- Alice selects a random key k and sends over $c = E_b(k, r)$.
- The Bank decrypts c and checks that the same r was received, and if so it keeps k as its secret key.
- Meanwhile, Alice knows k because it generated k, and it knows Bank's identity from the certificate.



Theorem 99. This simple one-sided AKE protocol has static security.

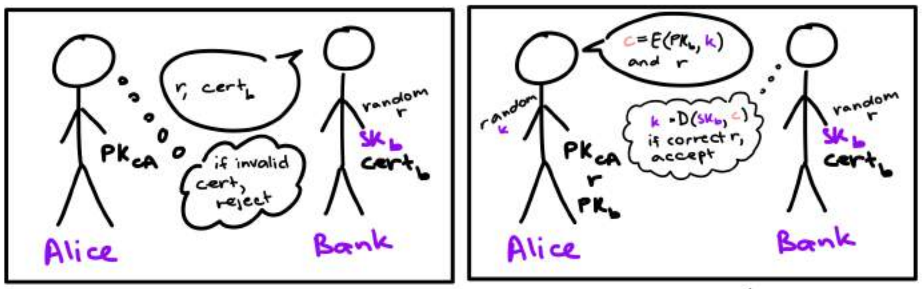
Informally, this means that if Alice and the Bank are not corrupt then we have secrecy and authenticity for Alice.

Now we will look at some variants of this that are insecure.

Insecure Variant 1: r not encrypted.

- The Bank sends over r, which is sampled randomly from R and cert_b.
- Alice selects a random key k and sends over $c = E_b(k)$ and r.

- The Bank decrypts c and checks that the same r was received, and if so it keeps k as its secret key.
- Meanwhile, Alice knows k because it generated k, and it knows Bank's identity from the certificate.



This is not secure against **replay attacks**: if Alice uses the shared key later on to encrypt a message such as "I am Alice, pay Eve \$30" then Eve can save that ciphertext and replay it later to get the Bank to keep sending Eve money.
In equations, if the above message is m, and our symmetric encryption scheme is E_s, then Alice:

- gets r, cert$_b$
- sends $c = E(k)$ and r
- is now authenticated and then sends $E_s(k, m)$ to the Bank to get it to send money to Eve

But, an eavesdropping adversary can then set up their own session with the Bank using the same secret key as Alice: the adversary

- gets r', cert$_b$
- sends $c = E(k)$ and r'
- is now authenticated and then sends $E_s(k, m)$ to the Bank to get it to send more money to Eve

(where we see that the second step could not have been done in the secure version of AKE)

Here, note that each time Alice or the adversary has a separate conversation with the Bank, they are doing it in a new session. Thus, unless the new session was set up using the same shared key as before, the replay attack would not work.

Now, we will look at how the two-sided AKE works.

Simple Two-Sided AKE

Ingredients:

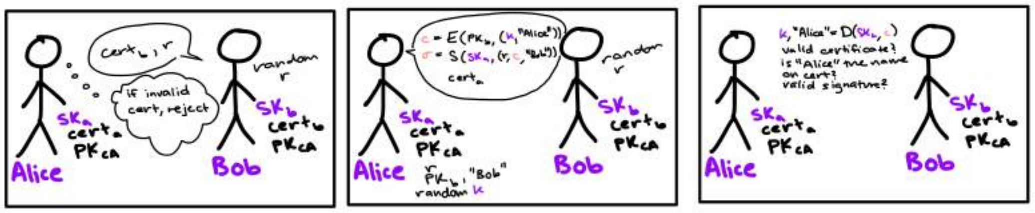
The certificate cert$_b$ has PK_b, and Bob has SK_b. Similarly, cert$_a$ has PK_a and Alice has SK_a.

We also have an encryption algorithm $E_b(m, r) = E(PK_b, m, r)$ where E is chosen-ciphertext secure.

And, we have $S_a(m, r) = S(SK_a, m, r)$ where S is a secure signature scheme.

Finally, we have the nonce space R, which is a large set, such as $\{0,1\}^{256}$.

- Bob sends over r, which is sampled randomly from R and cert_b.
- Alice selects a random key k and sends over $c = E_b(k, \texttt{Alice})$. Alice also sends $\sigma = S_a(r, c, \texttt{Bob})$ and cert_a.
- Bob decrypts c and checks that the name encrypted matches the name on the certificate. Then, he checks that σ is a valid signature on $(r, c, \texttt{Bob})$, and if that is the case, he accepts that `Alice` is the person he is talking to and k is the shared key.
- Meanwhile, Alice knows k because she generated k, and she knows Bob's identity from his certificate.

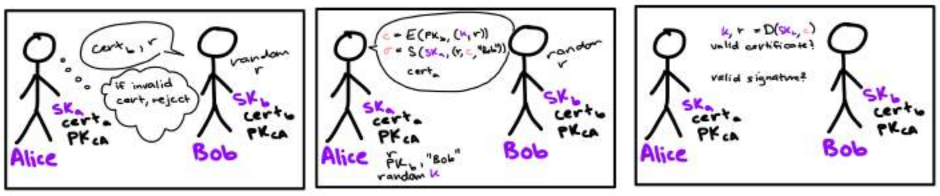


Theorem 100. This is a statically secure AKE.

Again, we will look at small changes to this protocol that make it insecure.

Insecure Variant: encrypt r instead of `Alice`

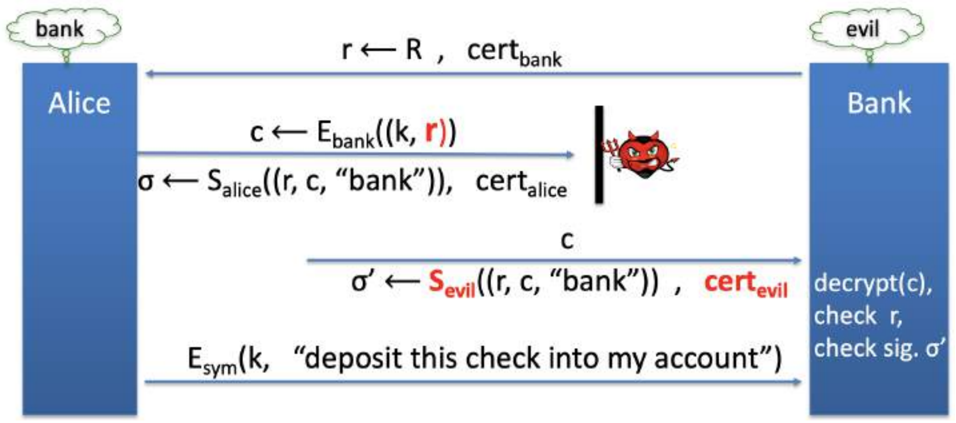
- Bob sends over r, which is sampled randomly from R and cert_b.
- Alice selects a random key k and sends over $c = E_b(k, \mathbf{r})$. Alice also sends $\sigma = S_a(r, c, \texttt{Bob})$ and cert_a.
- Bob decrypts c and checks that it contains the correct $\mathbf{r}$. Then, he checks that σ is a valid signature on $(r, c, \texttt{Bob})$, and if that is the case, he accepts that `Alice` is the person he is talking to and k is the shared key.
- Meanwhile, Alice knows k because she generated k, and she knows Bob's identity from his certificate.



This is vulnerable to an identity misbinding attack.

This works as follows:

- Bob sends over r, which is sampled randomly from R and cert_b.
- Alice selects a random key k and sends over $c = E_b(k, \mathbf{r})$. Alice also sends $\sigma = S_a(r, c, \texttt{Bob})$ and cert_a.
- The adversary blocks this message, and instead sends over c, $\sigma = S_{\text{evil}}(r, c, \texttt{Bob})$ and $\text{cert}_{\text{evil}}$.
- Bob decrypts c and checks that it contains the correct $\mathbf{r}$. Then, he checks that σ is a valid signature on $(r, c, \texttt{Bob})$, and if that is the case, he accepts that `Mr EvilMan` is the person he is talking to and k is the shared key.
- Meanwhile, Alice knows k because she generated k, and she knows Bob's identity from his certificate.


bank
evil
Alice
Bank
r ← R , cert_bank
c ← E_bank((k, r))
σ ← S_alice((r, c, "bank")), cert_alice
c
σ' ← S_evil((r, c, "bank")) , cert_evil
decrypt(c), check r, check sig. σ'
E_sym(k, "deposit this check into my account")

How does this hurt you? For example, if you submit your homework assignment using this protocol, and the adversary performs an identity misbinding attack, then Prof Boneh will think that the adversary has submitted a homework assignment, and give him credit instead of you.

Note that putting `Alice` in the ciphertext instead of r protects against this, because if the adversary tries to perform this attack, Bob will notice that there is a mismatch between the name on the ciphertext and the name on the certificate being sent over, and reject.

Now, we return to the simple one-sided AKE. Note that we said this had static secrecy. But the problem is that it does not have forward secrecy: every shared key k can be decrypted using the Bank's (unchanging) secret key SK_b, so if the adversary steals SK_b, they can get all past k's and therefore read all past messages.

(This protocol was used in TLS 1.2 but was deprecated for TLS 1.3)

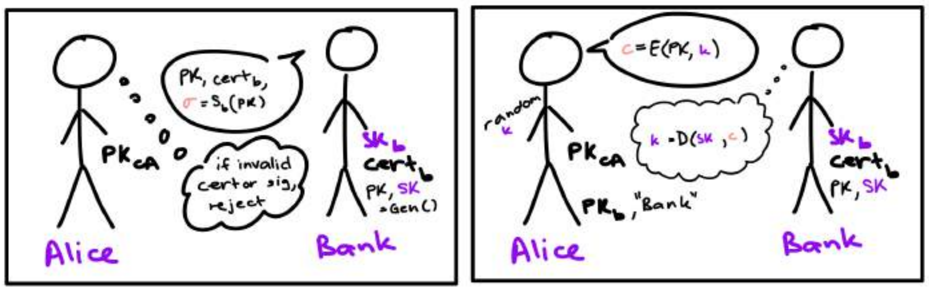
So, we can now look at a slightly modified version of this protocol that does have forward secrecy:

<u>One-Sided AKE with Forward Secrecy</u>

Along with the ingredients we had last time, we now have a public-key encryption scheme $(\text{Gen}(), E, D)$.

- The Bank calls Gen() to generate (SK, PK).
- The Bank sends over PK, cert_b, and $S_b(PK)$.
- Alice verifies that $S_b(PK)$ is actually the public key signed using PK_b. selects a random key k and sends over $c = E(PK, k)$.

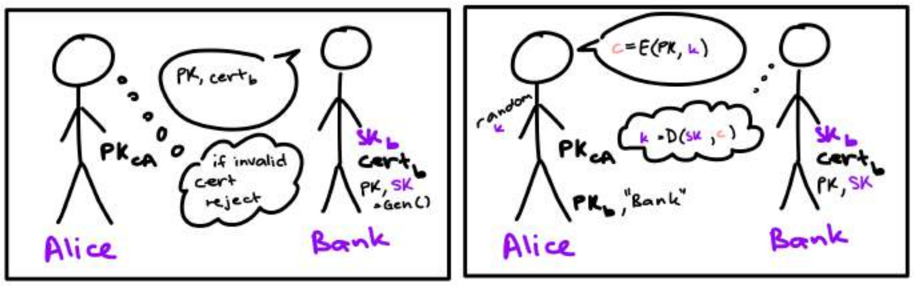
- The Bank computes $k = D(SK, c)$, rejects if D rejects, and then throws away SK. Now the bank knows k.
- Meanwhile, Alice knows k because it generated k, and it knows Bank's identity from the certificate.



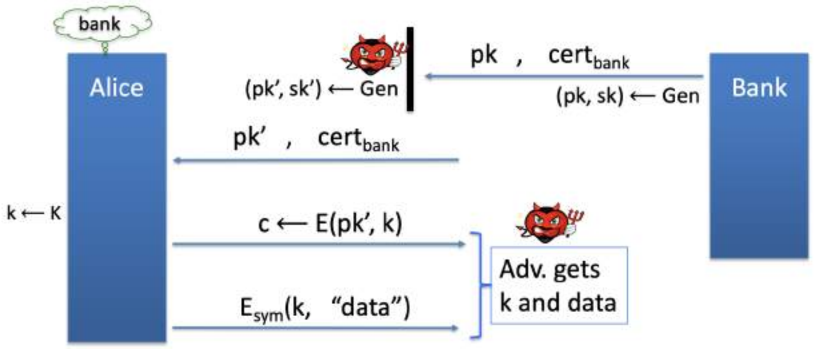
Since SK is deleted when the protocol completes, this has forward secrecy.

Insecure Variant: do not sign PK

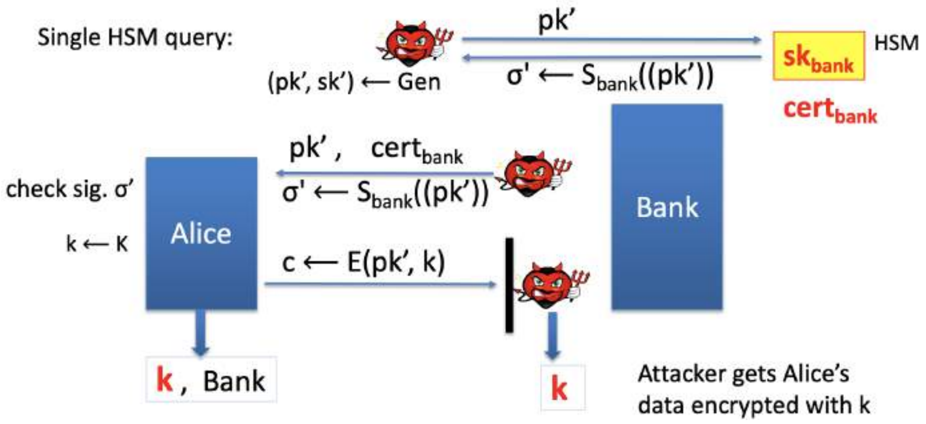
- The Bank calls Gen() to generate (SK, PK).
- The Bank sends over PK and cert_b. (**without any signature**)
- Alice selects a random key k and sends over $c = E(PK, k)$.
- The Bank computes $k = D(SK, c)$, rejects if D rejects, and then throws away SK. Now the bank knows k.
- Meanwhile, Alice knows k because it generated k, and it knows Bank's identity from the certificate.



This completely exposes the shared key k, because an attacker can just block the bank's first message and send over their own PK' with the bank's certificate, and then it will be the adversary, not the bank, that finds out k.



But, the forward secrecy protocol is still not HSM secure. We can see that an adversary that makes a single HSM query to get a signature on some PK' using SK_b can now use that PK' to intercept any AKE that anyone is trying to have with the Bank.

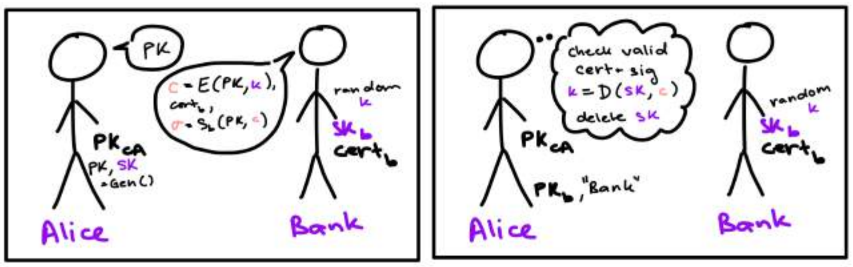


So now we will look at a protocol that also provides HSM security:

<u>Simple One-Sided AKE with HSM Security</u>

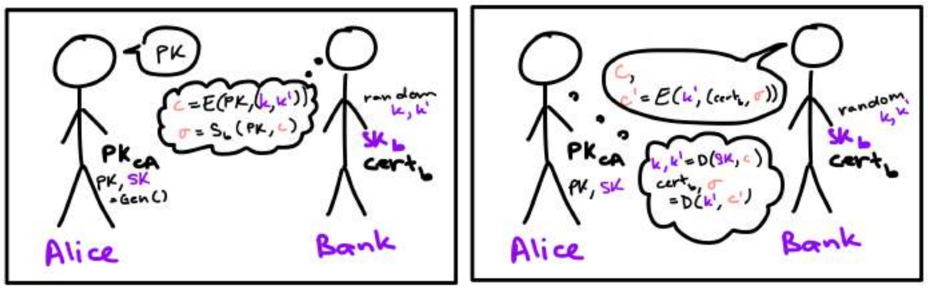
Along with the ingredients we had last time, we now have a public-key encryption scheme $(\text{Gen}(), E, D)$.

- Alice calls Gen() to generate (SK, PK), and sends over PK.
- The Bank generates a random shared key k and sends over $c = E(PK, k)$, cert_b, and $\sigma = S_b(PK, c)$.
- Alice checks that the σ is actually a signature on PK, c from the Bank (using PK_b from the certificate), and then computes $k = D(SK, c)$. Then, it deletes SK. She now knows k and that she is talking to the Bank.
- Meanwhile, the Bank knows k because it generated k.

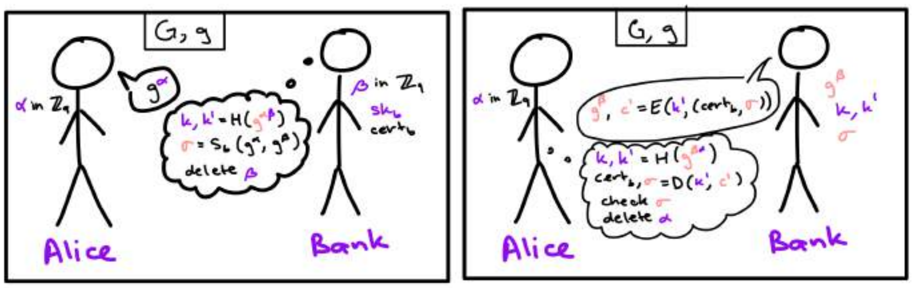


Since SK_b is now being used to sign an ephemeral PK, c that Alice sends, a new HSM query needs to be made each time we are setting up a session, so the adversary can no longer make a single HSM query to break all of HSM security.

The last problem is this is not private: since the bank's certificate is sent in the clear, an eavesdropper can see exactly who Alice is talking to. But this is easily fixed: we just need to modify the protocol to also encrypt cert$_b$ and σ.



In practice, we use DHAKE. This is very similar to the above, except instead of using ElGamal as the encryption scheme, we directly use Diffie-Helman, as below:


G, g
α in $\mathbb{Z}_q$
g^α
$k, k' = H(g^{\alpha\beta})$
$\sigma = S_b(g^\alpha, g^\beta)$
delete β
β in $\mathbb{Z}_q$
sk_b
$cert_b$
Alice
Bank
G, g
α in $\mathbb{Z}_q$
$g^\beta, c' = E(k', (cert_b, \sigma))$
g^β
k, k'
σ
$k, k' = H(g^{\beta\alpha})$
$cert_b, \sigma = D(k', c')$
check σ
delete α
Alice
Bank

Lecture 16: Zero-Knowledge Protocols

More Variants of AKE (that we won't explain here)

- AKE based on a shared secret (either a high-entropy secret, or a low-entropy secret like a password): used to get forward secrecy by generating a new shared secret from the previous one
- deniable AKE:
  In the previous version of AKE, Alice gets the bank's signature on her public key and ciphertext, which means she can prove that the bank was talking to her. In deniable AKE, we want both parties to be able to deny that they had communicated with her - no signature on non-secret information

TLS 1.3 Session Setup

A simplified version of the "full handshake" is the following, where the client and the server are trying to communicate:

1. the client sends a `clientHello` message which includes all the cipher suites it supports, and a Diffie-Hellman public key
2. the server sends a `serverHello` message which includes a cipher suite selected from the ones the client sent, their Diffie-Helman public key, and the server's encrypted certificate
3. the client sends a "finished" message to indicate that the session has been set up correctly

(Then the session has been set up, both parties have the shared secret key, and the client can now send HTTP requests to the server.)

But this means we need three rounds of communication before the client can even send an HTTP get request, which will not be practical in high-latency networks. There's a new field of crypto called crypto in space where they come up with workarounds for issues like this, so that we can communicate with devices on the moon, for example.

The finish state for this is a hash of the entire transcript so far, so that they agree that they have received the same keys and the setup was successful.

The reason we have a set of cipher suites that the client can support is because TLS is meant to be a worldwide protocol. So all browsers are required to support AES-GCM, but a Russian server might prefer to use the Russian standard GOST if that is an option, and an older server might prefer to use an older encryption scheme.

Session Setup from Pre-Shared Keys

Let us say that a server and browser have almost completed an old session, but would like to resume this old session later. Since the server would require a lot of storage to save every prior session it's had, it instead has the browser store the previous session data. The way this works is the browser saves the `preSharedKey` which is generated from the current shared key. Then, the server sends the browser a `newSessionTicket` which contains all the session data, including the `preSharedKey`, encrypted under the server's main secret key.
The browser cannot read the `newSessionTicket` because it does not have the server's secret key. But it can store it for the server. Then, the next time it is trying to connect to the server and resume its old session, it sends the `newSessionTicket` with its `clientHello`. Then, the server can decrypt this to get all the old session data, and the browser and server can engage in an abbreviated handshake to generate the new shared key from the `preSharedKey` (for forward secrecy) rather than having to engage in the full

Diffie-Helman handshake.

PSK 0-RTT (pre-shared key zero roundtrip time)

People really dislike the TLS round trips because since each ad has to load separately, the ads take a longer time to load, so they become less effective.

This is a feature that allows you to complete a GET request, for example, before the TLS handshake is completed. It can be beneficial in speeding things up, but it's also very dangerous.

You do the same process as before (with a pre-shared key), but in the ClientHello, you also send the 0-RTT application data, which in this case would be the GET request. Then, as the handshake is being done, the server can also complete this request, so it can send you the requested data the moment the handshake is done.

The 0-RTT application data is encrypted using k_{CE}, which is the client early key-exchange key. This is derived from the previous shared key and the ClientHello.

BUT the 0-RTT application data is very vulnerable to replay attacks, because someone could continually replay the ClientHello with the encrypted application data without getting to the rest of the handshake.

So, we have to make sure that the application data does not have side effects; for example, if it was a request to Amazon to buy a book, then someone could use the replay attack to make you buy 50 copies of the book. We generally only allow application data that is in the form of a GET request (banning POST requests, for example) but there are still GET requests that could have dangerous side effects when replayed.

Now, we will turn to talking about zero-knowledge protocols. This is one of the two "bonus topics" of the course.

Zero-Knowledge Protocols

First, we will review what the class NP is.

Definition 101. We say that a language L is in NP if L is a set of strings ($L \subseteq \{0,1\}^*$) and there exists a machine M that runs in polynomial time such that a string x is in L if and only if there exists some witness $w \in \{0,1\}^*$ such that $M(x,w) = 1$.

That is M is a verification machine for L; for every statement x in L, they can provide a valid "witness" w that causes M to believe it actually is in L, and moreover x that is not in L can forge such a valid witness.

Example 102. The example we will discuss is the equality of Dlog. Specifically, we have a cyclic group G of prime order q. Then the language we care about is

$$L_{EDL} = \left\{(g, h, g^\alpha, h^\alpha) \in G^4 \mid \alpha \in \mathbb{Z}_q, g, h \neq 1\right\}.$$

This is the set of four-tuples $g, h, u, v)$ such that $\mathrm{Dlog}_g\, u = \mathrm{Dlog}_h\, v$.
Then, we can see that given the witness α, it is easy to check that (g, h, u, v) is in the language; we just check if $u = g^\alpha$ and $v = h^\alpha$.

Definition 103. A **zero-knowledge proof system** for a language $L \in NP$ is a pair of probablistic polynomial time (PPT) algorithms P and V such that the prover P takes in (x, w) and the verifier V just takes in x.

Then, the prover and the verifier are going to send messages back and forth, and after that, the verifier outputs `yes` or `no`. So the prover is trying to convince the verifier that x is in the language.

Moreover, a zero-knowledge proof system has the following properties:

Definition 104. A proof system is **complete** if for all x, w such that $M(x, w) = 1$ (so x is in the language) then

$$\Pr\left[(P(x, w) \leftrightarrow V(x)) = \texttt{yes}\right] = 1,$$

or the verifier outputs `yes` after interacting with a prover that has x, w.

Definition 105. A proof system is **sound** if $x \notin L$, then no cheating prover can convince the verifier that $x \in L$. So when $x \notin L$, for all provers $\hat{P}$ (even ones that run in exponential time) and all w,

$$\Pr\left[(\hat{P}(x, w) \leftrightarrow V(x)) = \texttt{yes}\right]$$

should be negligible.

Then, a trivial proof system is one in which P just sends w to V, and then V can verify for themselves whether $M(x, w)$ is in the language. Since M runs in polynomial time, V can just run $M(x, w)$ and see if it accepts or rejects. But this is not a zero-knowledge proof because of the third property.
The third property states that the verifier should learn nothing about w besides the fact that $x \in L$. To define this formally, we first say that transcript$(P(x, w) \leftrightarrow V(x))$ is the sequence of messages sent back and forth between the prover and the verifier. This is a random variable, because P and V are randomized machines.

Definition 106. An **honest verifier zero knowledge** (HVZK) proof system is one in which there exits a PPT algorithm S called the simulator, where for all $x \in L$, the distribution $\{S(x)\}$ is computationally indistinguishable from the distribution $\{\text{transcript}(P(x, w) \leftrightarrow V(x))\}$.

What does this mean? Well we want the verifier to learn nothing about w. So we want everything in its conversation with P to be something it could have generated on its own given the knowledge that $x \in L$, because that means P isn't telling it anything besides this knowledge. So we ask whether a simulator that also runs in PPT and does not know w could generate the same probability distribution of transcripts.

Example 107. A zero-knowledge proof of *Where's Waldo* means you have to convince the verifier you know where Waldo is without revealing to the verifier anything about Waldo's location. So you enter a room, the verifier checks that you have brought nothing with you, and you're given scissors and the Where's Waldo page. You cut out Waldo, burn the rest of the page, and hand the verifier the picture of Waldo.

Example 108. A story from *The Lady Tasting Tea* was that the lady in the cafeteria where Fischer (the statistician) was working claimed that it was always important to pour the milk before the tea when making tea. Fischer wanted to test whether she could really tell the difference between the milk and the tea, so what he did was turn around and make tea, randomly deciding whether to pour the milk first or the tea first. Then he asked her to taste it and guess which was poured first.

If she really could tell the difference, she should be able to guess it right with probability 1, and if she couldn't, she would be able to guess it correctly with probability 1/2. If we repeat this experiment 10 times, and Fischer only believes she actually knows the difference if she never guesses incorrectly, then we can see that if she didn't really know the difference, she would only be able to trick him with probability 2^{-10}.

Moreover, Fischer learns nothing through this conversation; using the formal definition, their transcript would just be him handing her the tea and her guesses on which item was poured first. But if she did actually know the difference, her guesses would be correct every time, and he could predict them because he knows which item he actually did pour first. But he learns nothing about how to actually taste the difference between the two teas.

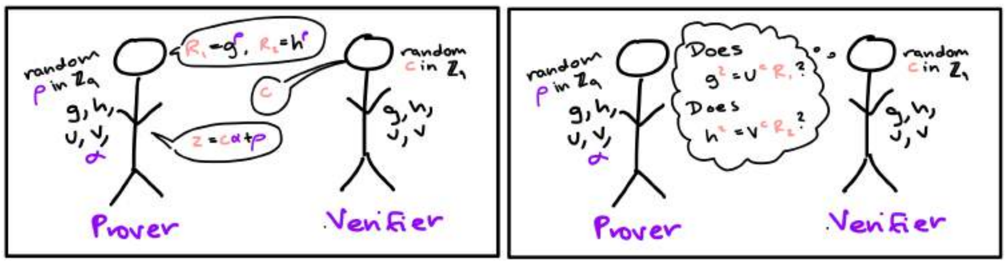
Example 109. We have the following zero-knowledge proof system for L_{EDL}.

The prover P knows (g, h, u, v, α) and the verifier knows (g, h, u, v).

The prover choses a random ρ from $\mathbb{Z}_q$ (where q is the order of our group G). Then, the prover sends $R_1 = g^\rho$ and $R_2 = h^\rho$.

Then, the verifier is going to choose a random "challenge" $c \in \mathbb{Z}_q$ and send that to the prover.

Finally, the prover returns $z = c\alpha + \rho \in \mathbb{Z}_q$, and the verifier accepts if $g^z = u^c R_1$ and $h^z = v^c R_2$.



Note: in this protocol, the verifier has no secrets, so we call this a public coin protocol.
Now, we will prove that this is an HVZK protocol.

1. completeness:
   If $(g, h, u, v) \in L_{EDL}$ and specifically $u = g^\alpha$ and $v = h^\alpha$ so $M(g, h, u, v, \alpha)$ accepts, then we can see that
   $$g^z = g^{c\alpha+\rho} = (g^\alpha)^c g^\rho = u^c R_1,$$
   and the parallel statement is true for h^z so the verifier will accept.

2. soundness:
   If $(g, h, u, v) \notin L_{EDL}$, so $u = g^\alpha$ and $v = h^\beta$ where $\alpha \neq \beta$, then for all malicious provers $\hat{P}$, the transcript will look like:
   $$(R_1 = g^{\rho_1}, R_2 = h^{\rho_2}, c, z),$$
   where the prover does not control c. Then
   $$\Pr[V \text{ accepts}] = \Pr[z = \alpha c + \rho_1 \text{ and } z = \beta c + \rho_2] \leq \Pr\left[c = \frac{\rho_1 - \rho_2}{\beta - \alpha}\right].$$
   Note that we as the verifier choose c randomly after $\rho_1, \rho_2, \alpha, \beta$ are set, and that $\alpha \neq \beta$ so we aren't dividing by zero. So the probability of this being true is exactly $\boxed{1/q}$.

3. HVZK:
   If $(g, h, u, v) \in L_{EDL}$, the simulator S needs to output transcript$(P(x, w) \leftrightarrow V(x))$. But the simulator doesn't need to generate the items of the transcript in the same order as the prover and verifier. So what the prover does is:

   (a) choose c, z randomly from $\mathbb{Z}_q$.

(b) set

$$R_1 = \frac{g^z}{u^c},\ R_2 = \frac{h^z}{v^c}.$$

So the reason the simulator can "cheat" is they can pick R_1, R_2 after c. We can see that $R_1 = g^\rho$ and $R_2 = h^\rho$ where $\rho = z - c\alpha$, c is chosen at random from $\mathbb{Z}_q$, and z is selected such that the above two conditions hold.
So this simulated transcript has the same probability distribution as the actual transcript, and the verifier must therefore learn nothing about α.

Lecture 17: Zero Knowledge-Based Signature Schemes

Soundness vs. Knowledge

In a normal zero-knowledge proof, we prove to the witness that there exists a w such that $M(x, w) = 1$, so $x \in L$. But, what if V wants a proof that P actually knows what w is? That is, we can prove soundness just by saying there is a witness, but not that the prover knows what the witness is. But a zero knowledge proof of knowledge (ZKPK) system also involves the prover showing that they know what w is.

Definition 110. A **zero knowledge proof of knowledge system** is one that has completeness, honest verifier zero knowledge, and for any prover $\hat{P}$ that can convince the verifier, the verifier can "extract" w from $\hat{P}$.

Example 111. We will show a ZKPK proof of Dlog. Specifically, let G be a finite cyclic group of prime order q. Then, let L be the language

$$L = \left\{(g, h = g^\alpha) \mid \alpha \in \mathbb{Z}_q, g \neq 1\right\}.$$

Then, showing that (g, h) is in the language is fairly trivial, because all pairs with $g \neq 1$ are in the group. But a ZKPK proof means the prover also convinces the verifier that they know what α is.

Schorr proof of knowledge of Dlog

The prover has $(g, h = g^\alpha, \alpha)$, while the verifier just has (g, h).
The prover chooses a random ρ from $\mathbb{Z}_q$. Then, the prover sends $R = g^\rho$ to the verifier.
Then, the verifier chooses a random c from $\mathbb{Z}_q$ and sends this to the prover.
The prover returns $z = c\alpha + \rho \in \mathbb{Z}_q$, and the verifier accepts if $g^z = R \cdot h^c$.

Theorem 112. The Schorr proof is complete, HVZK, and knowledge-sound, so it is a zero knowledge proof of knowledge system.

We can prove each of the three parts:

- completeness:
  This follows from the fact that if $h = g^\alpha$, then

  $$g^z = g^{c\alpha+\rho} = g^{c\alpha}g^\rho = h^c R.$$

- HVZK:
  A simulator, given (g, h) must produce a transcript (R, c, z) where $g^z = R \cdot h^c$, and every $R \in G$, $c \in \mathbb{Z}_q$ is equally likely.

  To do so, they choose z and c randomly from $\mathbb{Z}_q$, and then computes R as g^z/h^c. Then, they can output (R, c, z).

  So an honest verifier has the power to compute the transcript, so they learn nothing from the protocol. (Note again that this is an *honest* verifier zero knowledge protocol, so a dishonest verifier might choose c in such a way that they actually do learn something from the protocol.)

- knowledge sound:

  If a prover $\hat{P}$ can convince V that (g, h) is in the language, we can build an extractor that extracts α from $\hat{P}$.

  This works as follows:

  We have a $\hat{P}$ that ouputs a randomly generated R. Then, we send a randomly generated challenge c_1 and receive z_1 such that $g^{z_1} = Rh^{c_1}$.

  But, we also take a snapshot of the memory state of $\hat{P}$ right before we send the challenge c_1. Then, once we are done with the first challenge, we can rewind $\hat{P}$ to the previous state, where it has just sent R.

  At this state, we randomly generate another challenge c_2 and get back z_2, such that $g^{z_2} = Rh^{c_2}$.

  But then, we can see that $g^{z_1 - z_2} = h^{c_1 - c_2}$, and with high probability $c_1 \neq c_2$, so then we can compute

  $$\alpha = \frac{z_1 - z_2}{c_1 - c_2}.$$

  Thus, this is a proof of knowledge: if you can convince the verifier (g, h) is in the language, then the extractor can extract α from you, so you must have known α.

  (Note that a formal proof of this requires that even if the prover only works with non-negligible probability, we can still extract α with high probability - this extractor does work in that situation, but our proof above only shows that it works when the prover is always correct.)

But any public-coin protocol (where everything the verifier does is public) can be made non-interactive!

Fiat-Shamir Transform

Let us consider a "generic" public coin protocol, where the prover has (x, w) and the verifier has x. Then, the prover sends over a randomly generated R, the verifier sends a randomly generated challenge c, and the prover responds with z. Then, the verifier does some test on (R, c, z), and if the test passes, they accept.

So the only step the verifier really does here to make the proof interactive is generate c. So if we can have the prover also pick the randomly generated c, they can eliminate the need for interaction in this proof.

So the prover and verifier are given a hash function H, which works as a pseudorandom generator. Then, the prover does the following, given (x, w):

- randomly generate R
- generate $c = H(x, R)$
- generate z based on R and c
- send over $\pi = (R, z)$

For the verifier to accept, they can generate $c = H(x, R)$, and then run the same test as before on (R, c, z).

The question of what hash function to use for this and how to prove it is secure is a pretty major topic, and this will be covered further in CS 355.

But the reason this is interesting is once we have a non-interactive ZKPK, we can use this to build a digital signature. Basically, the signature for this signature scheme is a proof that we know the secret key for a given public key. We do this by including the message within the hash, so that our $c = H(x, m, R)$ instead of just $H(x, R)$.

Schnorr Signature Schemes

We have a cyclic group G of prime order q with generator g. Also, assume we have a hash function $H : G \times \mathcal{M} \times G \to \mathbb{Z}_q$.
Then, our signature scheme (Gen, S, V) is as follows:

- Gen() generates an α randomly from $\mathbb{Z}_q$ and $h = g^\alpha \in G$. Then, $SK = \alpha$ and $PK = h$.
- $S(SK = \alpha, m)$ plays the role of the prover:

  We generate ρ randomly from $\mathbb{Z}_q$ and set $R = g^\rho \in G$. Then, we generate $c = H(h, m, R) \in \mathbb{Z}_q$, and set $z = c\alpha + \rho$. We output (c, z).

  (Note that outputting c and outputting R are equivalent; the optimized Schnorr signature outputs c.)
- $V(PK = h, m, \sigma = (c, z))$ plays the role of the verifier:

  We generate $R = g^z/h^c$. Then, we check whether $H(h, m, R) = c$, and accept if it is.

Theorem 113. The signature scheme (Gen, S, V) is secure assuming Dlog is hard in G and H is modeled as a "random oracle."

Note that this is dependent on R being randomly generated each time; if not, we can use the two signatures to reveal α (as we did with the extractor). This is very easy to mess up, because even a small bias can break security, so specs now say to choose ρ as a PRF applied to (SK, m). But in general, Schnorr is very dangerous to implement yourself - it's better to use a trusted library.