CSCI 1515: Applied Cryptography

P. Miao

Spring 2025

These are lecture notes for CSCI 1515: Applied Cryptography taught at Brown University by Peihan Miao in the Spring of 2025.

These notes were originally taken by Jiahua Chen in Spring 2023, and were updated in Spring 2024 by Sudatta Hor. They are currently being maintained by John Wilkinson. There has been gracious help and input from classmates and fellow TAs. Please direct any mistakes/errata to a thread on Ed, or feel free to pull request or submit an issue to the notes repository.

Notes last updated March 18, 2025.

Contents

§1 January 22, 2025

§1.1 Introduction

The course homepage is at https://cs.brown.edu/courses/csci1515/spring-2025/, where you can find information such as the syllabus, projects, homeworks, calendar, lectures and more.

The course is offered in-person in *Salomon 001*, as well as synchronously over Zoom and recorded asynchronously (lectures posted online). Lecture attendance and participation is highly encouraged!

EdStem will be used for course questions, and **Gradescope** is used for assignments.

§1.1.1 Staff

Peihan has been at Brown for a couple of years and this was the second time she is teaching this course. Before Brown, she was at the University of Illinois Chicago. Before that, she finished her PhD at UC Berkeley in 2019 with a focus in cryptography. Afterwards, she worked in industry for a couple of years (Visa) before deciding to come back to academia. She still collaborates with industry to see what problems need to be solved in practice.

During her PhD, she started off doing more theoretical cryptography but also did internships and found applied cryptography fascinating as well. Now she works in both.

Our course staff have all taken or TA-ed the course before and are excited to help you learn!

§1.1.2 Course Philosophy and Logistics

If look up other *applied* cryptography courses online or at other universities, you will find courses that have "applied" in their title. However, if you look at their syllabus or content, it's still mostly theoretical crypto. This may (1) deter students from learning about crypto and (2) leave a gap between theoretical crypto and crypto in practice. (2) is bad because if someone makes a mistake in the crypto domain, the consequences are often significant.

As such, it's helpful for students to get hands-on experience with cryptography:

- How cryptography has been used in practice and
- How cryptography will be used and implemented in the future.

The closest similar course is found at Stanford, which covers theoretical crypto in the first half and more applied crypto in the second. But even that course only covers very basic crypto that are very

well established. In the past 10 years or so, there are new and exciting topics in crypto that are gradually becoming more and more common which we will also cover in this course.

For this course, it will be *much less* about math and proofs, and much more about how you can use these tools to do something more fun. It will be coding heavy and all projects will be implemented in C++ using crypto libraries.

If, however, you are interested in the theoretical or mathematical side, you might consider other courses at Brown like CSCI 1510 and MATH 1580.

There is an option to capstone this course, contact Peihan about this. It would also be best to find a partner who is also capstoning this course.

The following is the grading policy:

Type	*Percentage*
Introductions	1%
Project 0	5%
Projects 1 & 2 & 4	30% (10% each)
Projects 3 & 5	24% (12% each)
Homeworks	25% (5% each)
Final Project	15%

You have 2 free late days for each one of the *projects*. There are **no** late days for *homeworks*, as we want to release the solution guide as soon as possible.

All projects are independent, but collaboration is allowed and encouraged. However, you *must write up your own code.*

If you're sick, let Peihan know with a Dean's note.

§1.2 What is cryptography?

At a high level, *cryptography is a set of techniques that protect sensitive or important information.*

Question. Where is cryptography used in practice? What guarantees do we want in these scenarios?

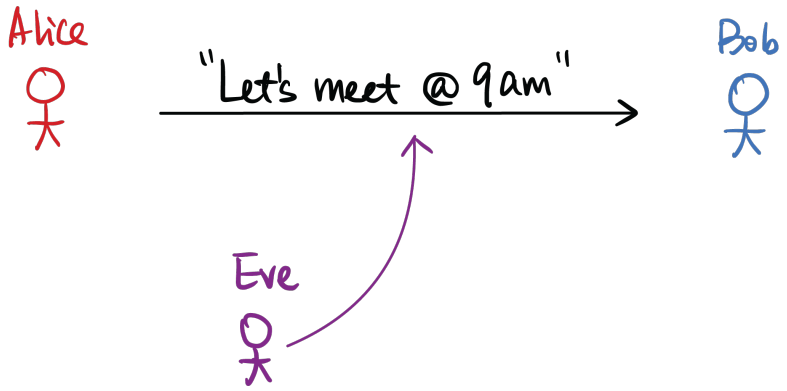
- Online transactions
  - When you make a purchase, you might not want people to see your bank balance, what else you have purchased, etc.

  - You also want to ensure that it was really *you* who purchased the item and not somebody else i.e. authentication

- Secure messaging
  - End-to-end texting, iMessage
  - We don't want anyone else to see our messages
- Online voting
  - Privacy of votes, validity of votes
- Databases
  - Secure storage

§1.3 Secure Communication

We'll start with the most classic form of cryptography: *secure communication*.



Assume Alice wants to communicate to Bob "Let's meet at 9am", what are some security guarantees we want?

- Eve cannot *see* the message from Alice to Bob.
- Eve cannot *alter* the message from Alice to Bob.

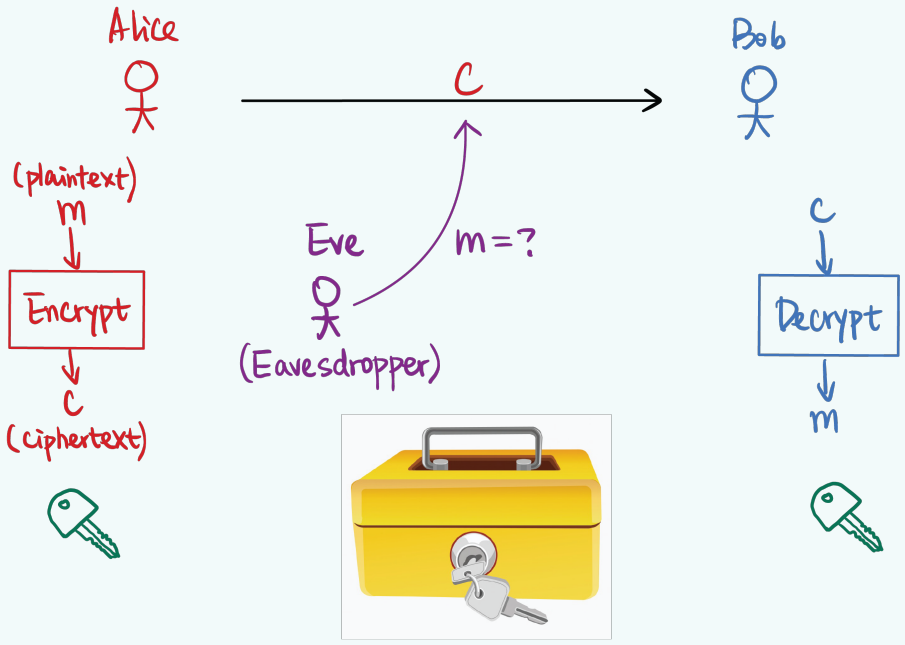
These two guarantees are the most important guarantees! The former is called message secrecy, the latter is called message integrity.

§1.3.1 Message Secrecy

Definition 1.1 (Message Secrecy)

We want cryptography to allow Alice to *encrypt* the message m (which we call *plaintext*) by running an algorithm that produces a *ciphertext* c. We call this an *encryption scheme*.

Bob will be able to receive the ciphertext c and run a *decrypt* algorithm to produce the message m again. This is akin to a secure box that Alice locks up, and Bob unlocks, while Eve does not know the message. The easiest way is for Alice and Bob to agree on a shared secret key.

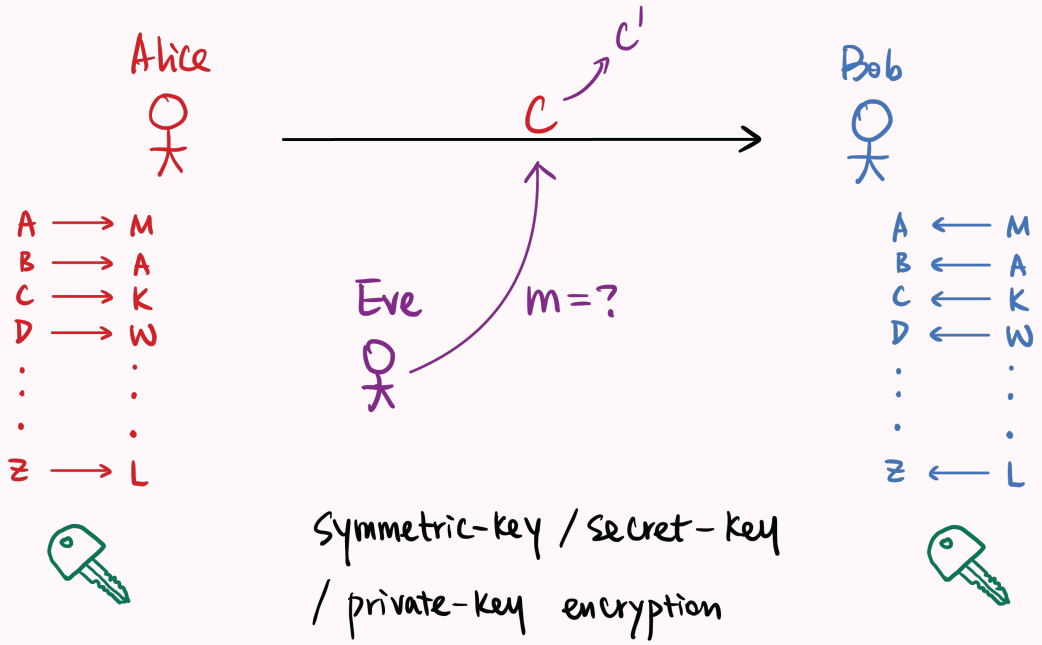


In this model, Eve is a weaker adversary, an *eavesdropper*. Eve can only see the message, not alter it.

Example 1.2 (Substitution Cipher)

The key that Alice and Bob jointly uses is a permutation mapping from $\{A \dots Z\} \to \{A \dots Z\}$. This mapping is the *secret key*.

Bob also has the mapping, and takes the inverse of the permutation to retrieve the message.



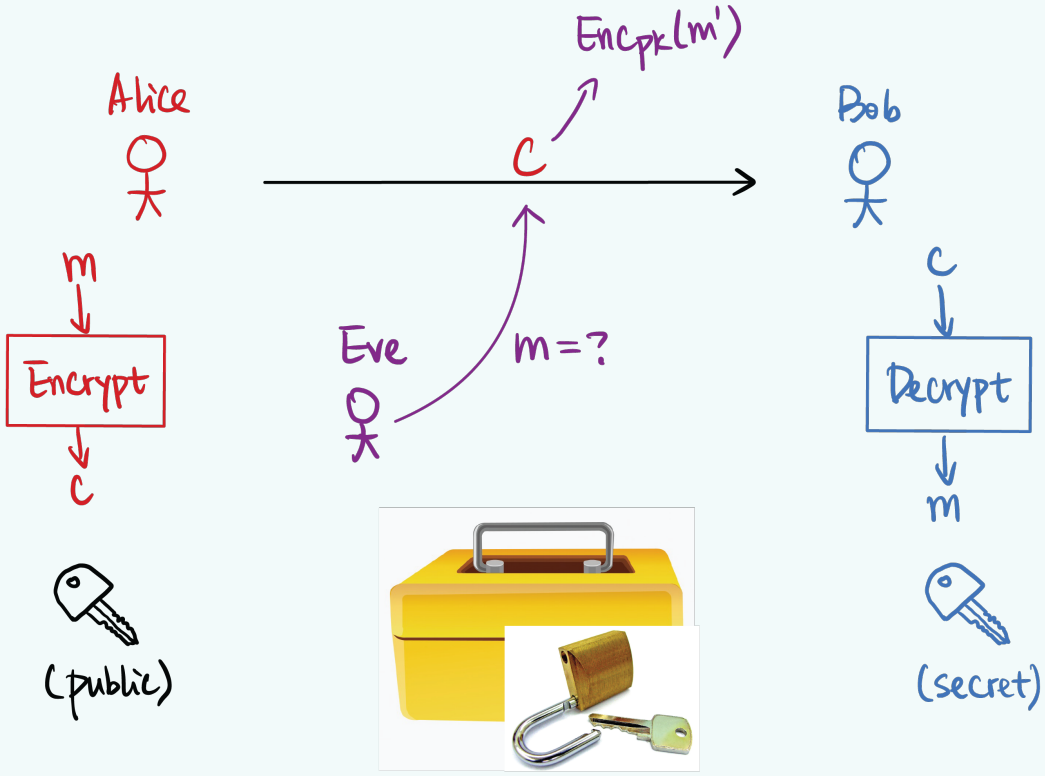
This scheme is not quite secure! *Why?*

Given a large enough text, you can apply frequency analysis and break the substitution cipher.

Remark. This encryption scheme also requires that Alice and Bob meet up in person to exchange this shared private key. Schemes like this are called *symmetric-key, secret-key, or private-key encryption.* They need to somehow agree first on the same secret key.

Definition 1.3 (Public-key Encryption)

There is another primitive that is much stronger: public-key encryption. Bob generate both a *public* key and a *private* key, and then publishes his public key. You can consider a lock where you don't need a key to lock it[1], and only Bob has the key to unlock it.

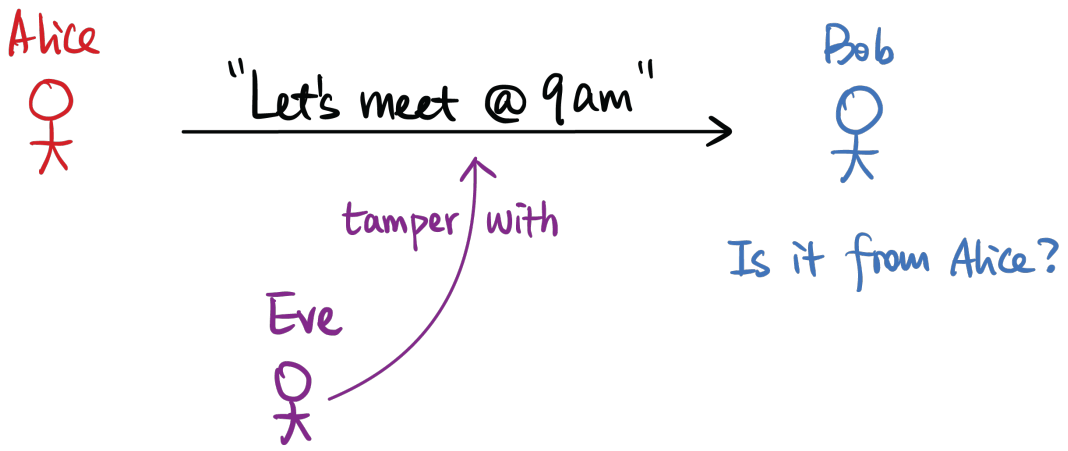


This is seemingly magic! Bob could publish a public key on his homepage, anyone can encrypt using a public key but only Bob can decrypt. *Stay tuned, we will see public-key encryption schemes next lecture!*

§1.3.2 Message Integrity

Alice wants to send a message to Bob again, but Eve is stronger! Eve can now tamper with the message.

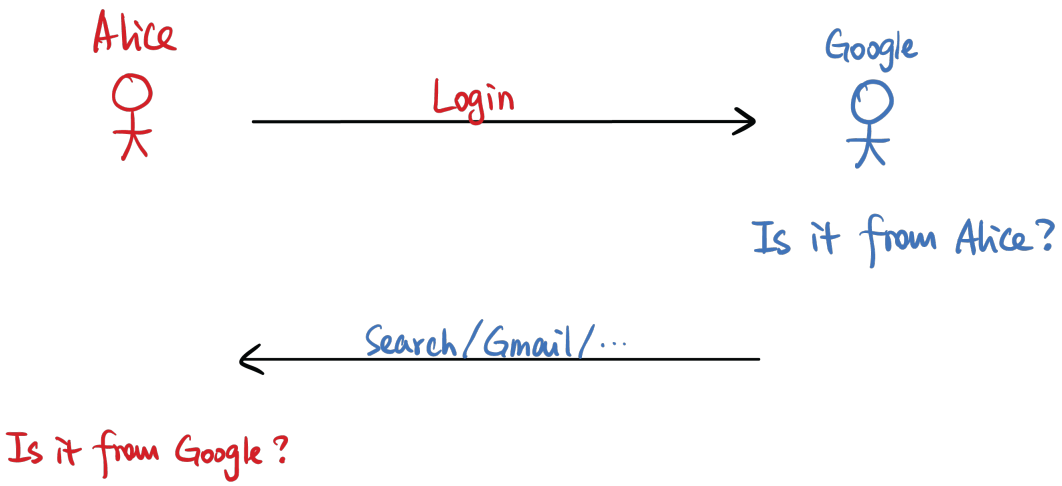
[1] You literally click it closed



Bob wants to ensure that the message *actually* comes from Alice. Does our previous scheme (of encrypting messages) solve this problem? Nope!

Eve can change the ciphertext to something else, they could pretend to be Alice. In secret-key schemes, if Eve figures out the secret-key, they can forge messages from Alice. Even if Eve doesn't know the underlying message, they could still change it to some other ciphertext which might be correlated to the original ciphertext, *without knowing the underlying message*. We'll see how Eve can meaningfully do this in some schemes. Alice could send a message "Let's meet at x AM" and Eve could tamper this to say "Let's meet at $x + 1$ AM."

This is sort of an orthogonal problem to message secrecy. For example, when Alice logs in to Google, Google needs to verify that Alice actually is who she claims to be.



This property that we want is called message integrity.

§1.3.3 Signal and Auth

The first two projects are Signal and Auth, whose aim will be to cover secure messaging and secure authentication.

Projects Overview

0. *Cipher* Warm-up, you will implement some basic cryptographic schemes.
1. *Signal* Secure Communication: how to communicate in secret.
2. *Auth* Secure Authentication: how to authenticate yourself.
3. *Vote* Zero-Knowledge Proofs: we'll use ZKPs to implement a secure voting scheme.
4. *PIR* Fully Homomorphic Encryption: a form of post-quantum cryptography.
5. *Yaos* Secure Multiparty Computation: we'll implement a way to run any function securely between two parties.

We'll now introduce the latter three projects!

§1.4 Zero-Knowledge Proofs

This is to prove something without *revealing* any additional knowledge.

For example, Alice may want to

- Prove she knows the difference in taste between Coke and Pepsi without revealing how
- Prove that you have a bug in your code without revealing the bug
- She has the secret key for this ciphertext without revealing the plaintext
- Prove that she owns two different colored pens to her colorblind friend Bob

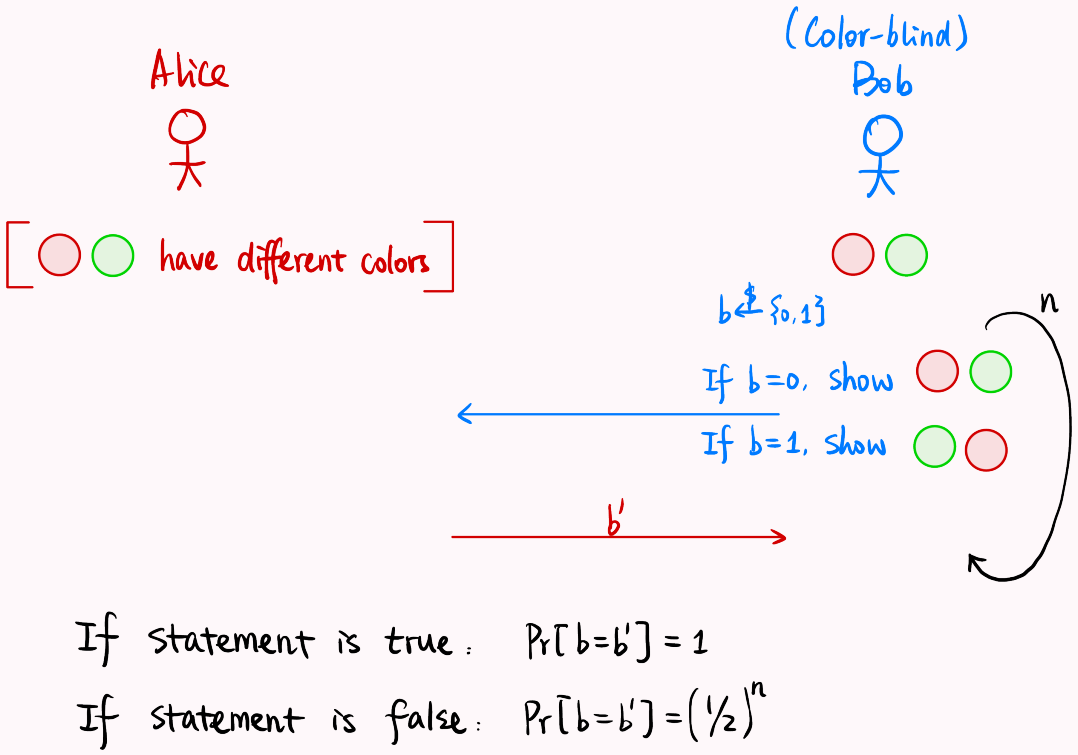
How is this possible? Let's examine this first scenario, which closely follows the red-green pens example from class $orthered - greenballsproblem$.

Example

Alice claims to have two different colored markers. She wants to prove this to her friend Bob, but there is an issue - Bob is colorblind!

Bob will randomly sample a bit $b \xleftarrow{\$} \{0,1\}$, with $b = 0$ being "Stay" and $b = 1$ being "Swap".

Bob will then either swap the order of the markers and show them to Alice, or keep them the same. Alice will give a guess b' of whether Bob swapped or not.



If the statement is true, $\Pr[b' = b] = 1$ (Alice always gives the correct prediction).

If the statement is false, $\Pr[b' = b] = \frac{1}{2}$ (Alice is guessing with 0.5 probability).

To enhance this, we can run this a total of k times. If we run it enough times, Bob will be more and more confident in believing this. Alice getting this correct by chance has a $\frac{1}{2^k}$ probability.

The key idea, however, is that Bob doesn't gain any knowledge of how Alice differentiates.

> **Remark.** This is a similar strategy in proving graph non-isomorphism.
>
> For people who have seen this before, generally speaking, any NP language can be proved in zero-knowledge. Alice has the *witness* to the membership in NP language.

§1.5 Fully Homomorphic Encryption

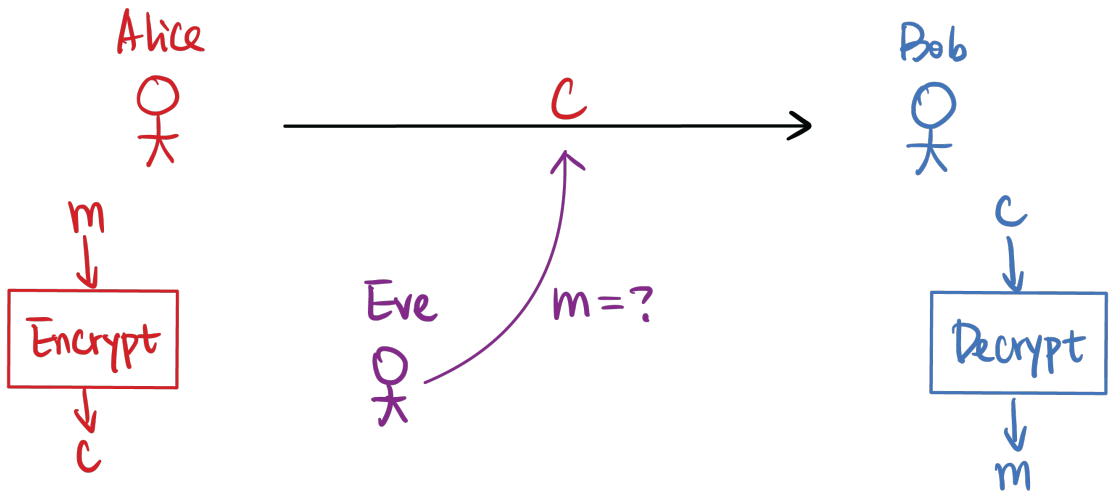
We'll come back to the secure messaging example.

Alice wants to send Bob a message. She encrypts it somehow and sends a ciphertext $c_1 = \mathsf{Enc}(m_1)$.

A nice feature for some encryption schemes is for Eve to do some computation homomorphically on the ciphertexts. Eve might possibly want to add ciphertexts (that leads to plaintext adding)

$$c_1 = \mathsf{Enc}(m_1),\ c_2 = \mathsf{Enc}(m_2) \Rightarrow c' = \mathsf{Enc}(m_1 + m_2)$$

or perhaps $c'' = \mathsf{Enc}(m_1 \cdot m_2)$, or compute arbitrary functions. *Sometimes*, this is simply adding $c_1 + c_2$, but usually not.

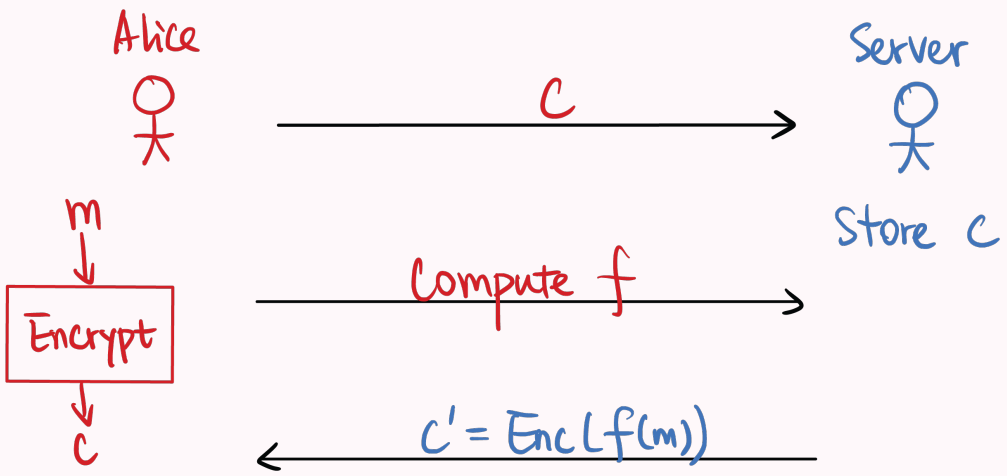


$$c_1 = \mathsf{Enc}(m_1)$$
$$c_2 = \mathsf{Enc}(m_2)$$
$$\Rightarrow \quad c' = \mathsf{Enc}(m_1 + m_2)$$
$$c'' = \mathsf{Enc}(m_1 \cdot m_2)$$

We want to hopefully compute any function in polynomial time!

Example (Outsourced Computation)

Alice has some messages but doesn't have enough compute. There is a server that has *a lot* of compute!



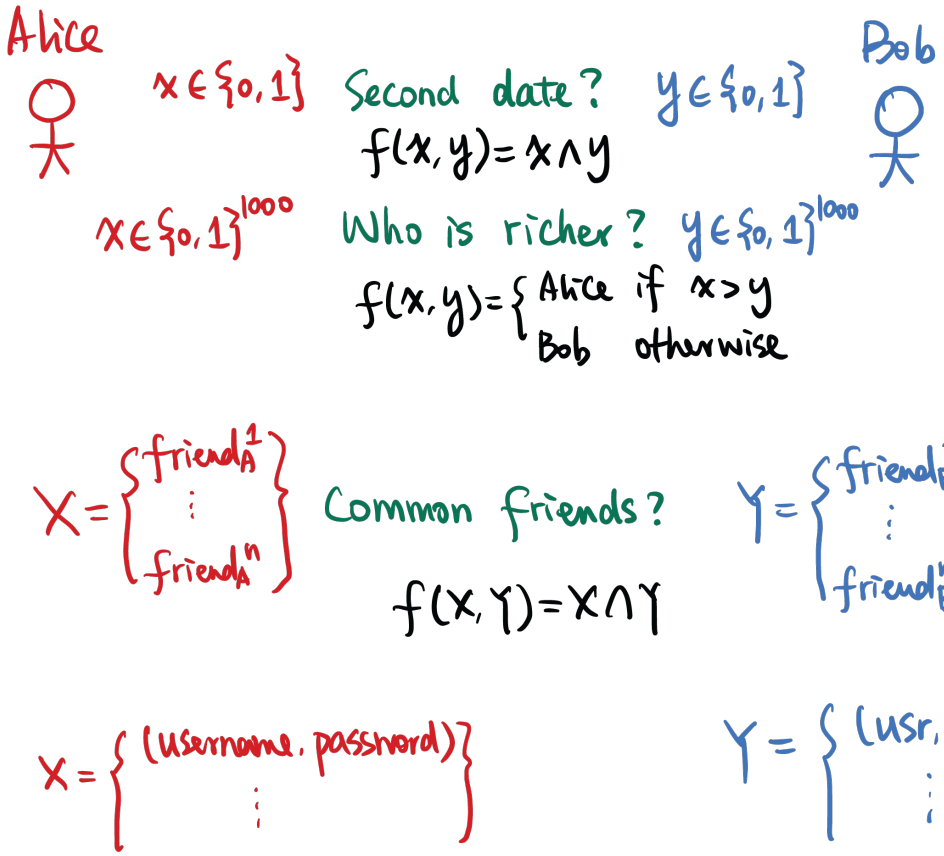
Alice encrypts her data and stores it in the server. At some point, Alice might want to compute a function on the encrypted data on the server, without the server revealing the original data. For example, she may want to search across that data without the server decrypting it (or sending her the entire encrypted DB). With homomorphic encryption, the server may be able to query and return an encrypted value to Alice without learning anything about the query itself.

This is an example of how fully homomorphic encryption can be useful.

Remark. This problem was not solved until 2009 (when Peihan started her undergrad). Theoretically, it doesn't even seem that possible! Being able to compute functions on ciphertexts that correspond to functions on plaintexts.

To construct fully-homomorphic encryption, we'll be using lattice-based cryptography which is a post-quantum secure!

§1.6 Secure Multi-Party Computation



Example (Secure AND)

Alice and Bob go on a first date, and they want to figure out whether they want to go on a second date. They will only go on a second date if and only if both agree to a second date.

How will they agree on this? They could tell each other, but this could be embarrassing. One way is for them to share with a third-party (this is what dating apps do!). However, there might not always be an appropriate third party (in healthcare examples, not everyone can be trusted with the data).

In this case, Alice has a choice bit $x \in \{0,1\}$ and Bob has a choice bit $y \in \{0,1\}$. They are trying to jointly compute $f(x,y) = x \wedge y$.

Remark 1.4. Couldn't a party still figure out how the other party feels? For example, if Bob's bit was 1 and the joint result was 0, Bob can *infer* that Alice's bit was 0.

This is, in effect, the best we can do. The ideal guarantee is that each party only learns any information they can infer from the *output* and their input. However, they should not learn anything more.

What are we trying to achieve here? We want to jointly compute some function, where each party has private input, such that each party only learns the output. They should not learn anything about other partys' inputs.

Example (Yao's Millionaires' Problem)

Perhaps, Alice and Bob wants to figure out who is richer. The inputs are $x \in \{0,1\}^{1000}$ and $y \in \{0,1\}^{1000}$ (for simplification, let's say they can express their wealth in 1000 bits). The output is the person who has the max.

$$f(x,y) = \begin{cases} \text{Alice} & \text{if } x > y \\ \text{Bob} & \text{otherwise} \end{cases}$$

Example (Private Set Intersection)

Alice and Bob meet for the first time and want to determine which of their friends they share. However, they do not want to reveal who specifically are their friends.

X is a set of A's friends $X = \{\mathsf{friend}_A^1, \mathsf{friend}_A^2, \cdots, \mathsf{friend}_A^n\}$ and Bob also has a set $Y = \{\mathsf{friend}_B^1, \mathsf{friend}_B^2, \cdots, \mathsf{friend}_B^m\}$. They want to jointly compute

$$f(X,Y) = X \cap Y.$$

You might need to reveal the cardinality of these sets, but you could also pad them up to a maximum number of friends.

This has a lot of applications in practice! In Google Chrome, your browser will notify you that your password has been leaked on the internet, without having access to your passwords in the clear. X will be a set of *your* passwords, and Google will have a set Y of *leaked* passwords. The *intersection* of these sets are which passwords have been leaked over the internet, without revealing all passwords in the clear.

Question. Isn't the assumption that the size of input records is revealed weaker than using a trusted third-party?

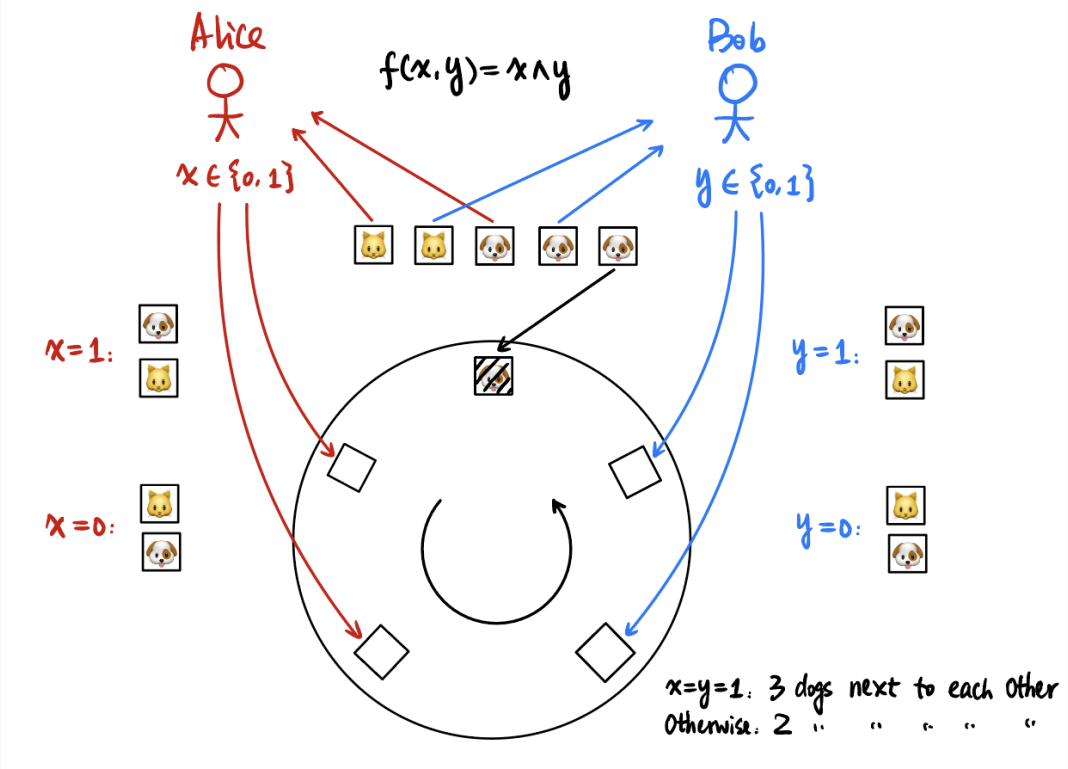
Yes, however in some cases (hospital health records), parties are legally obliged to keep data secure. We wish for security more than the secrecy of cardinality.

In the general case, Alice and Bob have some inputs x and y with bounded length, and they want to jointly compute some function f on these inputs. This is Secure Two-Party Computation. Furthermore, there could be multiple parties $x_1, \ldots, x_n$ that jointly compute $f(x_1, \ldots, x_n)$ that hides each input. This is Secure Multiparty Computation.

We'll explore a toy example with the bit-AND from the dating example.

Example (Private Dating)

Alice and Bob have choice bits $x \in \{0, 1\}$ and $y \in \{0, 1\}$ respectively. There is a *physical* round table with 5 identical slots, one already filled in with a *dog* facing down. Alice and Bob each have identical *dog*, *cat* cards (each of the *dog* and *cat* cards are indistinguishable from cards of the same value). Note that our use of *dog*, *cat* is arbitrary, and could be any other set of two elements: $0, 1$, x, o, etc.



Alice places her cards on the 2 slots in some order, and Bob does the same.

They then spin the table around and reveal all the cards, learning $x \wedge y$.

If $x = 1$, Alice places it as *dog* on top of *cat*, and if $y = 1$, Bob places it as *dog* on top of *cat* as well. Otherwise, they flip them. If $x = y = 1$, then the *dog*'s will be adjacent. If $x \neq y$, the order will be *dog*, *dog*, *cat*, *dog*, *cat* (the *cat*'s are not adjacent), regardless of which of Alice or Bob produced $x = 0$ (or both!).

This is a toy example! It doesn't use cryptography at all! Two parties have to sit in front of a table. This is called card-based cryptography. We will be using more secure primitives.

§1.7 Further Topics

We might cover some other topics:

- Differential Privacy
- Crypto applications in machine learning
- Crypto techniques used in the blockchain[2]

What else would you like to learn? What else do you want to understand? Do go through the semester with these in mind! *How do I log into Google? How do I send messages to friends?*

Feel free to let us know on Ed!

§1.8 Q & A

- *Do I need to have a crypto background?*

  No!

- *Why C++*

  Existing crypto libraries are mostly in C++ and most students have seen C/C++ in either cs33 or cs300. We did, however, consider implementing everything in Rust!

- *Class Participation*

  Course attendance is expected for all students, since a large portion of the class structure relies on students answerin questions in class. Even if you don't actively volunteer, try to think through the answer on your own! Students may join class via Zoom if they are remote-only, traveling, or sick.

- *What is the difference between CSCI 1515 and CSCI 1510, MATH 1580, or CSCI 1410?*

  CSCI 1510 is essentially "theoretical cryptography." It covers formal definitions and constructions and proofs. There is no coding, just proofs.

  MATH 1580 considers crypto from the mathematical perspective. They try to understand some of the computational assumptions we assume from a mathematical standpoint. I.e. why

[2] One important techniques is Zero-Knowledge proofs, for example.

is factoring hard to compute, and what is the best algorithm to compute it? In CS, we simply assume factoring is hard. MATH 1580 is more similar to number theory and group theory.

CSCI 1040 is a cryptography course for students without a computer science or math background. It is self described as "crypto for poets!"

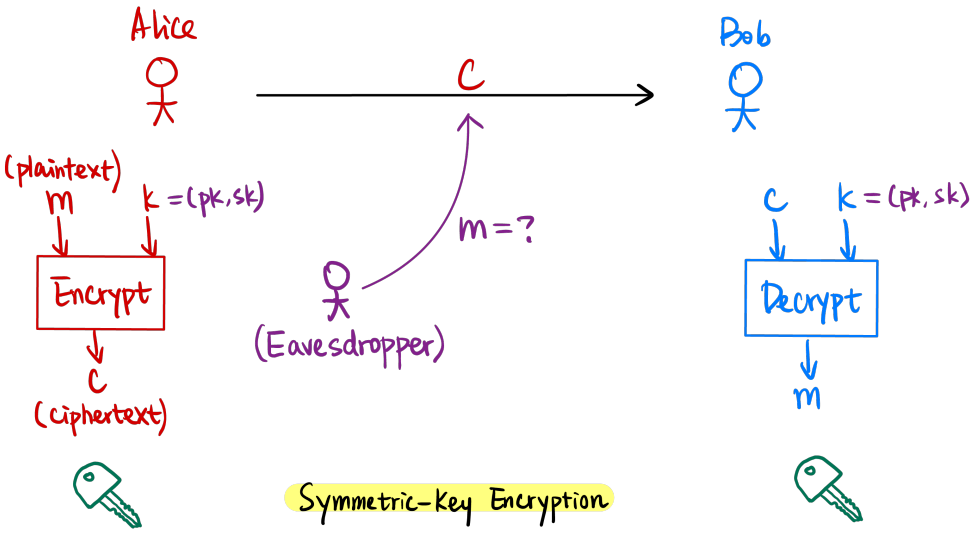
CSCI 1515, on the other hand, takes a hands on approach to cryptography. This class is suitable for both students with limited cryptography exposure more interested in cryptography from a software engineering perspective, as well as students with a cryptography theory background looking to understand what makes an implementation efficient in practice.

§2 January 27, 2025

§2.1 Encryption Scheme Basics

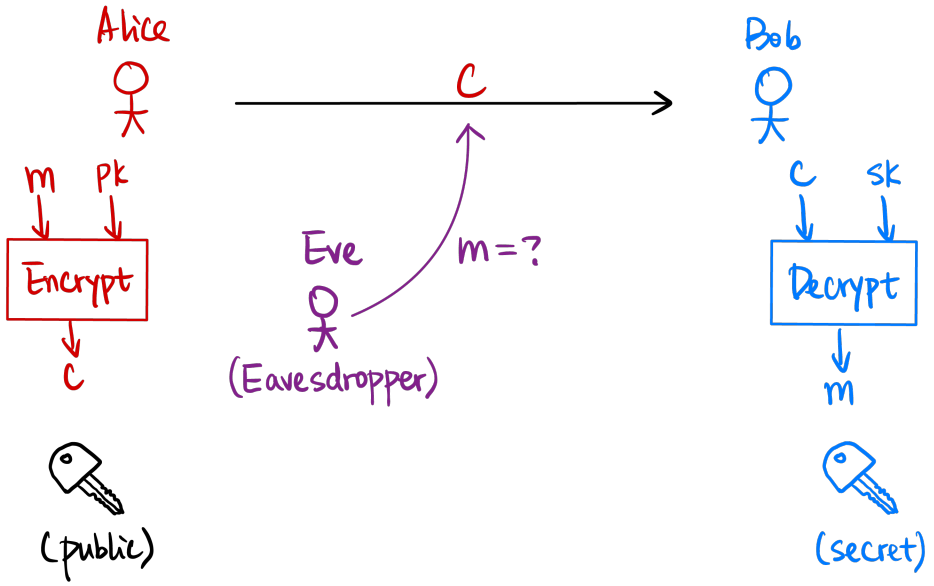
This lecture we'll cover encryption schemes. We briefly mentioned what encryption schemes were last class, we'll dive into the technical content: how we construct them, assumptions, RSA, ElGamal, etc.

Fundamentally, an encryption scheme protects message secrecy. If Alice wants to communicate to Bob, Alice will encrypt a message (plaintext) using some key which gives her a ciphertext. Sending the ciphertext through Bob using a public channel, Bob can use the key to decrypt the ciphertext and recover the message. An eavesdropper in the middle will have no idea what message has been transmitted.



In this case, they are using a shared key, which we called secret-key encryption or symmetric-key encryption.

A stronger version of private-key encryption is called public-key encryption. Alice and Bob do not need to agree on a shared secret key beforehand. There is a keypair (pk, sk), a *public* and *private* key.



§2.1.1 Syntax

Definition 2.1 (Symmetric-Key Encryption)

A symmetric-key encryption (SKE) scheme contains 3 algorithms, $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

Generation. Generates key $k \leftarrow \mathsf{Gen}$.

Encryption. Encrypts message m with key k, $c \leftarrow \mathsf{Enc}(k, m)$, which we sometimes write as $\mathsf{Enc}_k(m)$.

Decryption. Decrypts using key k to retrieve message m, $m := \mathsf{Dec}(k, c)$, or written as $\mathsf{Dec}_k(c)$.

Note the notation $\leftarrow$ and $:=$ is different. In the case of generation and encryption, the produced key k or c follows a *distribution* (is randomly sampled), while we had better want decryption to be deterministic in producing the message.

In other words, we use $\leftarrow$ when the algorithm might involve randomenss and $:=$ when the algorithm is deterministic.

Definition 2.2 (Public-Key Encryption)

A public-key encryption (PKE) scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ contains the same 3 algorithms,

Generation. Generate keys $(pk, sk) \leftarrow \mathsf{Gen}$.

Encryption. Use the public key to encrypt, $c \leftarrow \mathsf{Enc}(pk, m)$ or $\mathsf{Enc}_{pk}(m)$.

Decryption. Use the secret key to decrypt, $m := \mathsf{Dec}(pk, c)$ or $\mathsf{Dec}_{sk}(c)$.

Remark 2.3. Note that all these algorithms are public knowledge. This is known as Kerckhoffs's principle.

Intuitively, one key reason is because if the security of our scheme relies on hiding the *algorithm*, then if it is leaked we need to construct an entirely new algorithm. However, if the security of our scheme relies on, for example, a secret key, then we simply need to generate a new key.

Question. If we can construct public-key encryption, why do we even bother with secret-key encryption? We could just use the (pk, sk) pair for our secret key, and this does the same thing.

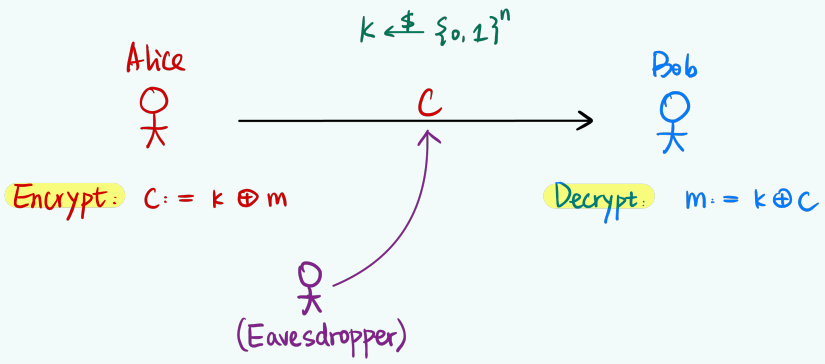
1. First of all, public-key encryption is almost always *more expensive*. Symmetric-key encryption schemes give us efficiency.

2. Public-key encryption relies on much stronger computational assumptions, whereas symmetric key encryption are still post-quantum secure.

§2.1.2 Symmetric-Key Encryption Schemes

Definition 2.4 (One-Time Pad)

Secret key is a uniformly randomly sampled n bit string $k \xleftarrow{\$} \{0,1\}^n$.



Encryption. Alice uses the secret key and bitwise-XOR with the plaintext.

$$\begin{array}{rr} \text{secret key} & k = 0100101 \\ \oplus\ \text{plaintext} & m = 1001001 \\ \hline \text{ciphertext} & c = 1101100 \end{array}$$

Decryption. Bob uses the secret key and again bitwise-XOR with the ciphertext

$$\begin{array}{rr} \text{secret key} & k = 0100101 \\ \oplus\ \text{ciphertext} & c = 1101100 \\ \hline \text{plaintext} & c = 1001001 \end{array}$$

This is widely used in cryptography, called *masking* or *unmasking*.

Question. Why is this correct?

An XOR done twice with the same choice bit b is the identity. In other words, $k \oplus (k \oplus m) = m$

Question. Why is this secure?

We can think about this as the distribution of c. $\forall m \in \{0,1\}^n$, the encryption of m is uniform over $\{0,1\}^n$ (since k was uniform).

Another way to think about this is that for any two messages $m_0, m_1 \in \{0,1\}^n$, $\mathsf{Enc}_k(m_0) \equiv \mathsf{Enc}_k(m_1)$. That is, the encryptions follow the *exact same* distribution. In this case, they are both uniform, but this is not always the case.

Question. Can we reuse k? Should we use the same key again to encrypt another message? Or, it is possible for the eavesdropper to extract information.

For example, $\mathsf{Enc}_k(m)$ is $c := k \oplus m$, and $\mathsf{Enc}_k(m')$ is $c' := k \oplus m'$. If the two messages are the same, the ciphertexts are the same.

By XOR c and c', we get

$$\begin{aligned} c \oplus c' &= (\cancel{k} \oplus m) \oplus (\cancel{k} \oplus m') \\ &= m \oplus m' \end{aligned}$$

This is why this is an *one-time* pad. This is a bit of an issue, to send an n-bit message, we need to agree on an n-bit message.

In fact, this is *the best* that we can do.

Theorem 2.5 (Shannon's Theorem)

Informally, for perfect (information-theoretic[3]) security, the key space must be at least as large as the message space.

$$|\mathcal{K}| \geq |\mathcal{M}|$$

where $\mathcal{K}$ is the key space and $\mathcal{M}$ is the message space.

Question. How can we circumvent this issue?

The high level idea is that we weaken our security guarantees *a little*. Instead of saying that they have to be *exactly the same* distribution, we say that they are *hard to distinguish* for an adversary with limited computational power. This is how modern cryptography gets around these lower bounds in classical cryptography. We can make *computational assumptions* about cryptography.

We can think about computational security,

Definition 2.6 (Computational Security)

We have computational security when two ciphertexts have distribution that cannot be distinguished using a polynomial-time algorithm.

Definition 2.7 (Polynomial-Time Algorithm)

A polynomial time algorithm $A(x)$ is one that takes input x of length n, A's running time is $O(n^c)$ for a constant c.

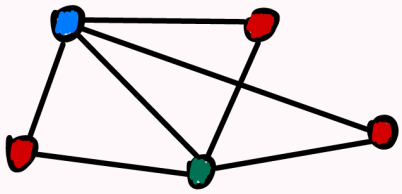
[3]That the distributions of ciphertexts are identical, that $\mathsf{Enc}_k(m_0) \equiv \mathsf{Enc}_k(m_1)$.

Definition 2.8 (NP Problem)

A decision problem is in nondeterministic polynomial-time when its solution can be *verified* in polynomial time.

Example 2.9 (Graph 3-Coloring)

Given a graph, does it have a 3-coloring such that no two edges join the same color? For example,



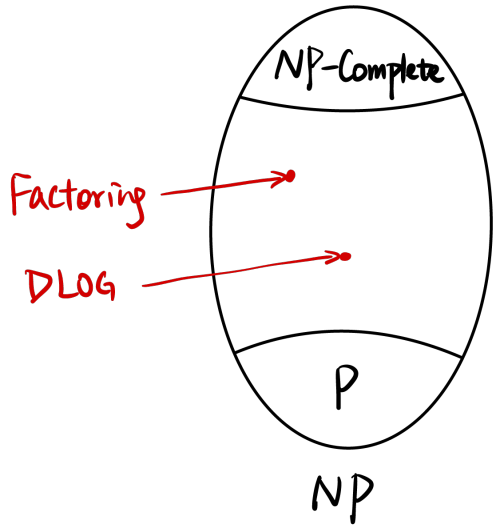
This can be *verified* in polynomial time (we can check if such a coloring is a valid 3-coloring), but it is computed in NP time.

Definition 2.10

An NP-complete problem is a "hardest" problem in NP. Every problem in NP is at least as hard as an NP-complete problem.

Right now, we assume $\mathsf{P} \neq \mathsf{NP}$. As of right now, there is no realistic algorithm that can solve any NP problem in polynomial-time.

Even further, we pick some problems not in NP-complete, not in P. We assume they are neither NP-complete nor in P (we don't yet have a reduction, but we don't know if one could exist) The reasoning behind using these problems is as we have no good cryptoscheme relying solely on NP-complete problems (we need something weaker).



Going back to our definition of computational security definition 2.6,

Definition (Computational Security)

Let the adversary be computationally bounded (i.e. only a *polynomial-time algorithm*). Then $\forall$ probabilistic poly-time algorithm $\mathcal{A}$,

$$\mathsf{Enc}_k(m_0) \overset{c}{\simeq} \mathsf{Enc}_k(m_1)$$

Where $\overset{c}{\simeq}$ is "computationally indistinguishable".

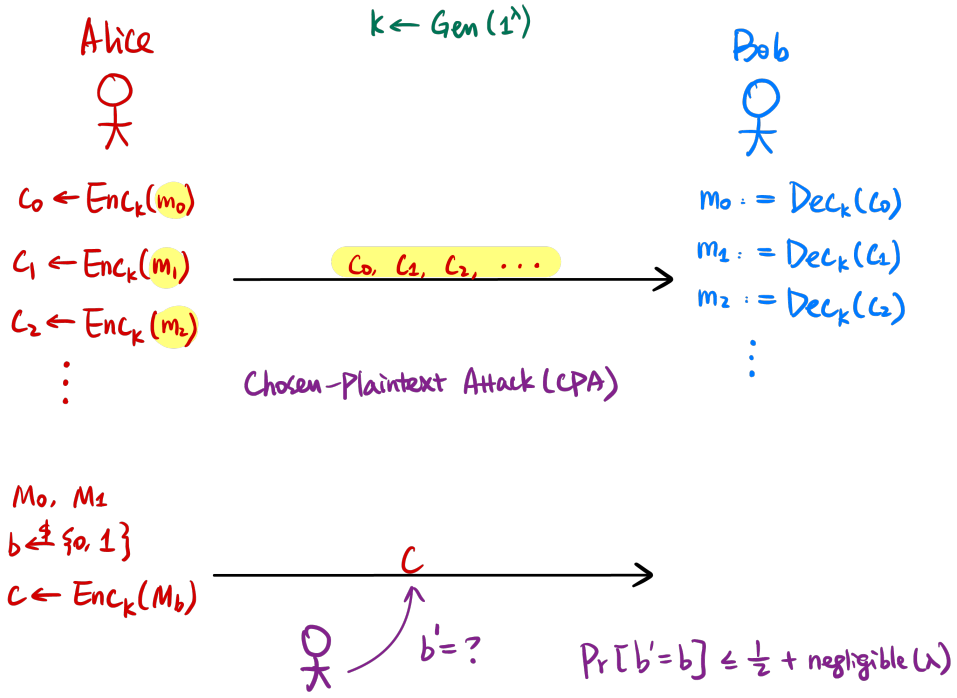
What does it mean for distributions to be "computationally indistinguishable"? Let's say Alice encrypts multiple messages $m_0, m_1, \ldots$ to Bob and produces $c_0, c_1, \ldots$. Even if Eve can see all plaintexts m_i in the open and ciphertexts c_i in the open, between known m_0, m_1 and randomly encrypting one of them $c \leftarrow \mathsf{Enc}_k(m_b)$ where $b \overset{\$}{\leftarrow} \{0,1\}$, the adversary cannot determine what the random choice bit b is. That is, $\Pr[b = b'] \leq \frac{1}{2} + \mathsf{negligible}(\lambda)$[4]. This is Chosen-Plaintext Attack (CPA) Security.

Definition 2.11 (Chosen-Plaintext Attack)

In simpler terms, a Chosen-Plaintext Attack (CPA) allows an adversary to request the encryptions of any number of chosen messages with the same key. The adverrsary can see the ciphertext of each message. Then, the adversary chooses two messages to be encrypted, and receives the ciphertext of one. The adversary must try to determine which message the ciphertext corresponds to.

Note: the adversary is allowed to send the same message as many times as it wishes.

[4] λ is the security parameter, roughly a measure of how secure the protocol is. If it were exactly equal $\frac{1}{2}$, we have information-theoretic security.



For a key generated $k \leftarrow \mathsf{Gen}(1^\lambda)$.

Theoretically, for λ a security parameter and an adversary running in time $\mathsf{poly}(\lambda)$, the adversary should have distinguishing advantage $\mathsf{negligible}(\lambda)$ where

$$\mathsf{negligible}(\lambda) \ll \frac{1}{\lambda^c} \qquad \forall \text{ constant } c.$$

In practice, we set $\lambda = 128$. This means that the best algorithm to break the scheme (e.g. find the secret key) takes time $\sim 2^\lambda$. Currently, this is longer than the age of the universe.

Remark 2.12. *Just how big is* 2^{128}*?* Well, see how long 2^{128} CPU cycles will take.

Let's assume the CPU spec is 3.8 GHz i.e. $3 \cdot 10^9$ cycles per second. Moreover, note that $2^{128} \sim 10^{40}$ CPU cycles.

Then doing the math ...

$$10^{40} \text{ CPU cycles} \cdot \frac{1s}{10^9 \text{ CPU cycles}} \cdot \frac{1 \text{ year}}{31,536,000s} \sim 10^{22} \text{ years}$$

Now let's be generous and say the age of the universe is 10^{11} years ... then we see that it would still take 10^{11} times the current age of the universe!

Example 2.13

If the best algorithm is a brute-force search for k, what should our key length be?

It can just be a λ bit string.

Example

What if the best algorithm is no longer a brute-force search, but instead for a key length l takes $\sim \sqrt{2^l}$?

Our key length should be 2λ. Doing the math, we want $\sqrt{2^l} \equiv 2^\lambda$, solving for l gives 2λ.

Going back to the original problem of secret-key encryption, how can we use our newfound cryptographic constructions to improve this?

From a pseudorandom function/permutation (PRF/PRP), we can reuse our secret key by passing it through the pseudorandom function.

The current practical construction for PRD/PRP is called the block cipher, and the standardized implementation is AES[5]

It is a computational assumption[6] that the AES construction is secure, and the best attack is currently a brute-force search (in both classical and quantum computing realms).

§2.1.3 Public-Key Encryption Schemes

Using computational assumptions, we explore some public-key encryption schemes.

RSA Encryption. This is based on factoring/RSA assumption, that factoring large numbers is hard.

ElGamal Encryption. This is based on the discrete logarithm/Diffie-Hellman Assumption, that finding discrete logs in $\mathbb{Z}_p$ is hard.

Lattice-Based Encryption. The previous two schemes are not quantum secure. Quantum computation will break these schemes. Lattice-based encryption schemes are post-quantum secure. They are associated with the difficulty of finding 'short' vectors in lattices[7].

[5] Determined via a competition for such an algorithm in the early 2000s.
[6] Based on heuristic, not involving any number theory!
[7] Covered later in class, we focus on the first two now.

Theorem 2.14

(Very informally,) It is impossible to construct PKE from SKE in a black-box way. This is called "black-box separation".

We first need to define a bit of number theory background.

Definition 2.15

We denote $a \mid b$ as a divides b, that is, there is integer c such that $b = a \cdot c$.

Definition 2.16 (Primes)

An integer $p > 1$ that only has two divisors: 1 and p

Definition 2.17 (Mod)

$a \mod N$ is the remainder of a when divided by N.

$a \equiv b \pmod N$ means when a and b are congruent modulo N. That is, $a \mod N = b \mod N$.

Question. How might we compute $a^b \mod N$? What is the time complexity? Let a, b, N be n-bit strings.

Naïvely, we can repeatedly multiply. But this takes b steps (2^n).

We can 'repeatedly square'. For example, we can get to a^8 faster by getting a^2, squaring to get a^4, and again to get a^8. We can take the bitstring of b and determine how to compute this.

Example

If $b = 100101_2$, we take $a \cdot a^4 \cdot a^{32} \mod N$ which can be calculated recursively (an example is given in the first assignment).

The time complexity of this is order $O(n)$ for n-bit a, b, N[8].

Definition 2.18

The $\gcd(a, b)$ is the greatest common divisor of a, b. If $\gcd(a, b) = 1$, then a, b are coprime.

[8] Not exactly order n, we should add the complexity of multiplication. However, this should be bounded by N since we can log at every step.

Question. How do we compute gcd? What is its time complexity?

Example

We use the Euclidean Algorithm. Take $\gcd(12, 17)$,

$$\begin{aligned} 17 &\mod 12 = 5 \\ 12 &\mod 5 = 2 \\ 5 &\mod 2 = 1 \\ 2 &\mod 1 = 0 \end{aligned}$$

or take $\gcd(12, 18)$

$$\begin{aligned} 18 &\mod 12 = 6 \\ 12 &\mod 6 = 0 \end{aligned}$$

Theorem 2.19 (Bezout's Theorem, *roughly*)

If $\gcd(a, N) = 1$, then $\exists b$ such that

$$a \cdot b \equiv 1 \pmod{N}.$$

This is to say, a is invertible modulo N. b is its inverse, denoted as a^{-1}.

Question. How do we compute b?

We can use the Extended Euclidean Algorithm!

Example

We write linear equations of a and N that sum to 1, using our previous Euclidean Algorithm. Take the previous example $\gcd(12, 17)$,

$$\begin{aligned} 17 &\mod 12 = 5 \\ 12 &\mod 5 = 2 \\ 5 &\mod 2 = 1 \\ 2 &\mod 1 = 0 \end{aligned}$$

We write this as

$$\begin{aligned} 5 &= 17 - 12 \cdot 1 \\ 2 &= 12 - 5 \cdot 2 = 12 \cdot x + 17 \cdot y \\ 1 &= 5 - 2 \cdot 2 = 12 \cdot x' + 17 \cdot y' \end{aligned}$$

where we substitute the linear combination of 5 into 5 on line 2, substitute linear combination of 2 into 2 on line 1, each producing another linear combination of $12, 17$.

If $\gcd(a, N) = 1$, we use the Extended Euclidean Algorithm to write $1 = a \cdot x + N \cdot y$, then $1 \equiv a \cdot x \pmod{N}$.

Definition 2.20 (Group of Units mod N)

We have set

$$\mathbb{Z}_N^\times := \{a \mid a \in [1, N-1], \gcd(a, N) = 1\}$$

which is the group of units modulo N (they are units since they all have an inverse by above).

Definition 2.21 (Euler's Phi Function)

Euler's phi (or totient) function, $\phi(N)$, counts the number of elements in this set. That is, $\phi(N) = |\mathbb{Z}_N^\times|$.

Theorem 2.22 (Euler's Theorem)

For all a, N where $\gcd(a, N) = 1$, we have that

$$a^{\phi(N)} \equiv 1 \pmod{N}.$$

With this, we can start talking about RSA.

§2.1.4 RSA

We first define the RSA assumption.

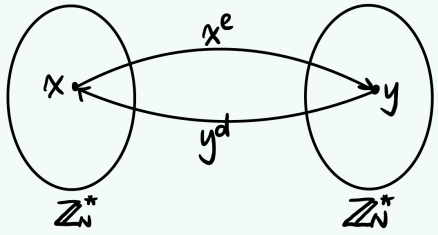
Definition 2.23 (Factoring Assumption)

Given two n-bit primes p, q, we compute $N = p \cdot q$. Given N, it's computationally hard to find p and q (classically).

Definition 2.24 (RSA Assumption)

Given two n-bit primes, we again compute $N = p \cdot q$, where $\phi(N) = (p-1)(q-1)$. We choose an e such that $\gcd(e, \phi(N)) = 1$ and compute $d \equiv e^{-1} \pmod{\phi(N)}$.

Given N and a random $y \xleftarrow{\$} \mathbb{Z}_N^\times$, it's computationally hard to find x such that $x^e \equiv y \pmod{N}$.



However, given p, q, it's easy to find d. We know $\phi(N) = (p-1)(q-1)$, so we can compute d from e by running the Extended Euclidean Algorithm. Then, taking $(x^e)^d \equiv x^{ed} \equiv x$ which allows us to extract x again.

Encrypting is exactly raising by power d, and decrypting is raising again by power e.

Remaining questions:

- How can we generate primes p, q?
- How can we pick e such that $\gcd(e, \phi(N)) = 1$?
- What security issues can you see?

We'll continue next class.

§3 January 29, 2025

§3.1 Basic Number Theory, *continued*

We recall a couple more definitions.

Definition 3.1

We first define the multiplicative group of integers modulo n as

$$\mathbb{Z}_N^* = \{a \in [1, N-1] \mid \gcd(a, N) = 1\}$$

Definition 3.2

We define the Euler's totient function as

$$\phi(N) = |\mathbb{Z}_N^*|$$

Example 3.3

If $N = p \cdot q$ where p, q are prime, then $\phi(N) = (p-1)(q-1)$.

Theorem 3.4 (Euler's Theorem)

$\forall a, N$ where $\gcd(a, N) = 1$, we have that $a^{\phi(N)} \equiv 1 \mod N$.

Corollary 3.5

If

$$d \equiv e^{-1} \mod \phi(N)$$

, then

$$\forall a \in \mathbb{Z}_N^*, (a^d)^e \equiv a \mod N$$

.

§3.2 RSA Encryption, *continued*

Recall: that the RSA encryption algorithm contains 3 components:

$\mathsf{Gen}(1^\lambda)$: Generate two n-bit primes p, q. We compute $N = p \cdot q$ and $\phi(N) = (p-1)(q-1)$. Choose e such that $gcd(e, \phi(N)) = 1$. We compute $d = e^{-1} \mod \phi(N)$. Our public key $pk = (N, e)$, our secret key is $sk = d$.

$\mathsf{Enc}_{pk}(m)$: $c = m^e \mod N$.

$\mathsf{Dec}_{sk}(c)$: $m = c^d \mod N$.

We have a few remaining questions:

1. How do we generate 2 primes p, q? More specifically, how do we generate two *large* primes efficiently?

2. How do we choose such an e?

3. How do we compute $d = e^{-1} \mod \phi(N)$?

4. How do we efficiently compute $m^e \mod N$ and $c^d \mod N$.

How do we resolve these issues to ensure the Gen step is efficient (polynomial time)?

1. We pick an arbitrary number p and check for primality efficiently (using Miller Rabin, a probabilistic primality test). We pick random numbers until they are prime. Since primes are 'pretty dense' in the integers, this can be done efficiently.

2. Since we're unsure whether coprime numbers are dense, we can just pick a small prime e. Guessing, like we did with p, q is also valid, although not strictly necessary.

3. We can compute d using the Extended Euclidean Algorithm.

4. We can repeatedly square (using fast power algorithm).

Question. What happens if we can factor N?

Then we can find p and q and calculate $\phi(N) = (p-1)(q-1)$, and then we can compute $d = e^{-1} \mod \phi(N)$. Thus, RSA relies crucially on the factoring problem being hard.

Question. The above scheme is known as "plain" RSA. Are there any security issues?

- It relies on factoring being difficult (this is the computational assumption). Post-quantum, Shor's Algorithm will break RSA.

- Recall last lecture that CPA (Chosen-Plaintext Attack) security was defined as an adversary not being able to discern between an encryption of m_0 and m_1, *knowing* m_0 and m_1 in the clear.

  Eve could just encrypt m_0 and m_1 themselves using public e, and discern which of the plaintexts the ciphertext corresponds to. For RSA, this is a very concrete attack.

The concrete reason is that the encryption algorithm Enc is *deterministic*. If you encrypt the same message twice, it will be the same ciphertext. Since this scheme is deterministic, it fails CPA security - for example, an adversary can encrypt messages and look for matches. We really want to be sure that $m \xleftarrow{\$} \mathbb{Z}_N^\times$ (that it has enough entropy).

Question. In practice, how can RSA be useful with these limitations?

As long as we pick the plaintext which is randomly sampled, security for RSA holds. There is also a more involved way of using RSA that is CPA-secure, but we will not go in detail of it.

Remark. In practice, we usually set length of p and q to be 1024 bits, and the key length is 2048 bits.

Moreover, note that although exponentiation can be done in polynomial time, it's still a very expensive operation. This is why public key encryption is, in general, more expensive than symmetric key encryption.

§3.3 Intro to Group Theory

Definition 3.6 (Group)

A group is a set $\mathbb{G}$ along with a binary operation $\circ$ with properties:

Closure. $\forall g, h \in \mathbb{G}$, $g \circ h \in \mathbb{G}$.

Existence of an identity. $\exists e \in \mathbb{G}$ such that $\forall g \in \mathbb{G}$, $e \circ g = g \circ e = g$.

Existence of inverse. $\forall g \in \mathbb{G}$, $\exists h \in \mathbb{G}$ such that $g \circ h = h \circ g = e$. We denote the inverse of g as g^{-1}.

Associativity. $\forall g_1, g_2, g_3 \in \mathbb{G}$, $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$.

We say a group is additionally *Abelian* if it satisfies commutativity

Commutativity. $\forall g, h \in \mathbb{G}$, $g \circ h = h \circ g$.

For a finite group, we use $|\mathbb{G}|$ to denote its *order*.

Example 3.7

$(\mathbb{Z}, +)$ is an Abelian group.

We can check so: two integers sum to an integer, identity is 0, the inverse of a is $-a$, addition is associative and commutative.

$(\mathbb{Z}, \cdot)$ is not a group. There is no inverse for 0 such the $0 \cdot h = 1$.

$(\mathbb{Z}_N^\times, \cdot)$ is an Abelian group ($\cdot$ is multiplication mod N).

Definition 3.8 (Cyclic Group)

Let $\mathbb{G}$ be a group of order m. We denote

$$\langle g \rangle := \{e = g^0, g^1, g^2, \ldots, g^{m-1}\}.$$

$\mathbb{G}$ is a cyclic group if $\exists g \in \mathbb{G}$ such that $\langle g \rangle = \mathbb{G}$. g is called a generator of $\mathbb{G}$.

Example

$\mathbb{Z}_p^\times$ (for prime p) is a cyclic group of order $p - 1$[9].

$$\mathbb{Z}_7^\times = \{3^0 = 1, 3^1, 3^2 = 2, 3^3 = 6, 3^4 = 5, 3^5 = 5\}.$$

Question. How do we find a generator?

For every element, we can continue taking powers until $g^\alpha = 1$ for some α. We hope that $\alpha = p - 1$ (the order of g is the order of the group), but we know at least $\alpha \mid p - 1$.

§3.4 Computational Assumptions

We have a few assumptions we make called the Diffie-Hellman Assumptions, in order of **weakest to strongest**[10] assumptions.

Let $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$ be a cyclic group $\mathbb{G}$ or order q (a $\theta(\lambda)$-bit integer) with generator g. For integer groups, keys are usually 2048-bits. For elliptic curve groups, keys are usually 256-bits.

Definition 3.9 (Discrete Lgoarithm (DLOG) Assumption)

Let $x \xleftarrow{\$} \mathbb{Z}_q$. We compute $h = g^x$.

Given $(\mathbb{G}, q, g, h)$, it's computationally hard to find the exponent x (classically).

[9] A proof of this extends beyond the scope of this course, but you are recommended to check out Math 1560 (Number Theory) or Math 1580 (Cryptography). You can take this on good faith.

[10] If one can solve DLOG, we can solve CDH. Given CDH, we can solve DDH. This is why CDH is *stronger* than DDH, and DDH is *stronger* than DLOG. It's not necessarily true the other way around (similar to factoring and DSA assumptions).

Definition 3.10 (Computational Diffie-Hellman (CDH) Assumption)

$x, y \xleftarrow{\$} \mathbb{Z}_q$, compute $h_1 = g^x$, $h_2 = g^y$.

Given $(\mathbb{G}, q, g, h_1, h_2)$, it's computationally hard to find g^{xy}.

Definition 3.11 (Decisional Diffie-Hellman (DDH) Assumption)

$x, y, z \xleftarrow{\$} \mathbb{Z}_q$. Compute $h_1 = g^x$, $h_2 = g^y$.

Given $(\mathbb{G}, q, g, h_1, h_2)$, it's computationally hard to distinguish between g^{xy} and g^z.

$$(g^x, g^y, g^{xy}) \overset{c}{\simeq} (g^x, g^y, g^z).$$

§3.5 ElGamal Encryption

The ElGamal encryption scheme involves the following:

$\mathsf{Gen}(1^\lambda)$: We generate a group $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$. We sample $x \xleftarrow{\$} \mathbb{Z}_q$, compute $h = g^x$. Our public key is $pk = (\mathbb{G}, q, g, h)$, secret key $sk = x$.

$\mathsf{Enc}_{pk}(m)$: We have $m \in \mathbb{G}$. We randomly sample $y \xleftarrow{\$} \mathbb{Z}_q$, which helps prevent our ciphertext from being deterministic. Our ciphertext is $c = \langle g^y, h^y \cdot m \rangle$. Note that $h = g^x$, so $g^{xy} \overset{c}{\simeq} g^z$ is a one-time pad for our message m.

$\mathsf{Dec}_{sk}(c)$: To decrypt $c = \langle c_1, c_2 \rangle$, we raise

$$c_1^x = (g^y)^x = g^{xy}$$
$$m = \frac{g^{xy} \cdot m}{g^{xy}} = c_2 \cdot (c_1^x)^{-1}.$$

Notes about ElGamal:

- Our group can be reused! We can use a public group that is fixed. In fact, there are *popular* groups out there used in practice. Some of these are Elliptic Curve groups which are much more efficient than integer groups. You don't need to use the details, yet you can use it! You can use any group, so long as the group satisfies the DDH assumption.
- Similar to RSA, this is breakable post-quantum. Given Shor's Algorithm, we can break discrete log.

§3.6 Secure Key Exchange

Using DDH, we can construct something very important, *secure key exchange*.

Definition 3.12 (Secure Key Exchange)

Alice and Bob sends messages back and forth, and at the end of the protocol, can agree on a shared key.

An eavesdropper looking at said communications cannot figure out what shared key they came up with.

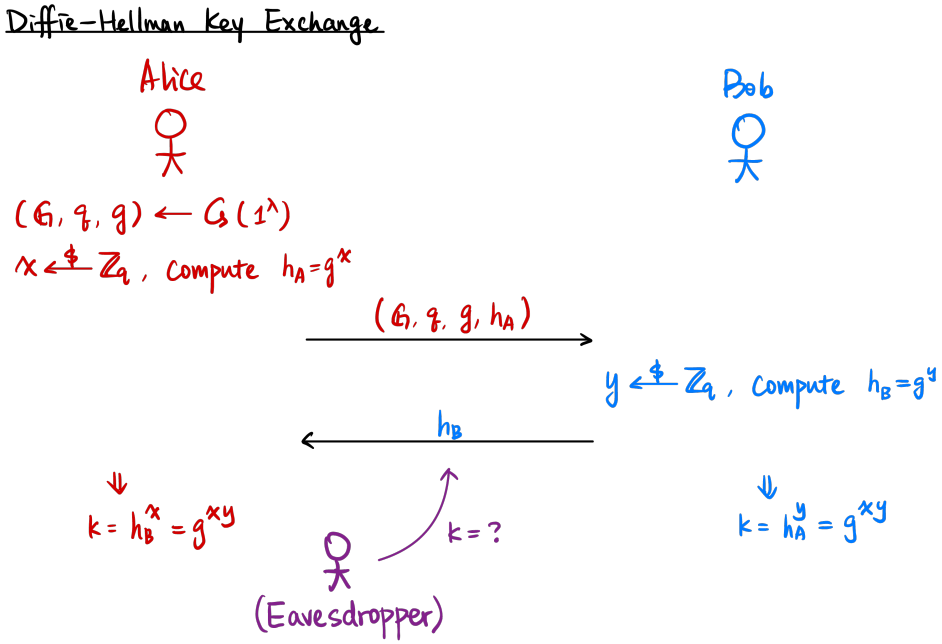
Theorem 3.13

Informally, It's impossible to construct secure key exchange from secret-key encryption in a black-box way.

Question. How do we build a key exchange from public-key encryption?

Bob generates a keypair (pk, sk). Alice generates a shared key $k \xleftarrow{\$} \{0,1\}^\lambda$, and sends $\mathsf{Enc}_{pk}(k)$ to Bob.

Using Diffie-Hellman, it's very easy. We have group $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$. Alice samples $x \xleftarrow{\$} \mathbb{Z}_q$ and sends g^x. Bob also samples $y \xleftarrow{\$} \mathbb{Z}_q$ and sends g^y. Both Alice and Bob compute $g^{xy} = (g^x)^y = (g^y)^x$.



What happens in practice is that parties run Diffie-Hellman key exchange to agree on a shared key. Using that shared key, they run symmetric-key encryption. This gives us efficiency. Additionally, private-key encryptions don't rely on heavy assumptions on the security of protocols (such as the DDH, RSA assumptions).

§3.7 Prime Order Subgroups

The Decisional Diffie-Hellman assumption (DDH) does *not* hold for prime p in cyclic group $\mathbb{Z}_p^\times$ with order $p-1$. We use the prime order subgroup of $\mathbb{Z}_p^\times$.

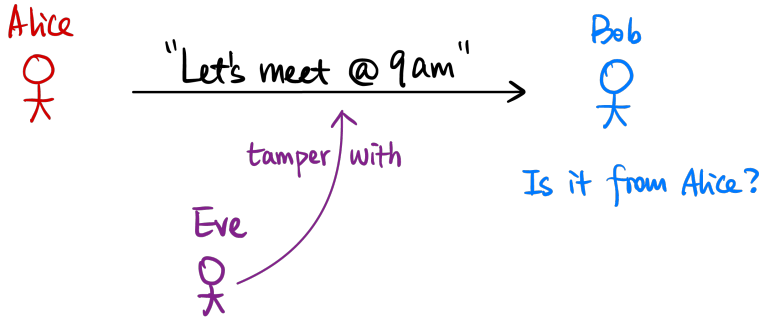
Definition 3.14 (Subgroup)
A subgroup is some subset of a group that is also a group itself.

Definition 3.15 (Safe Prime)
A prime p is a safe prime if $p = 2q + 1$, where q is prime. These are also known as Sophie Germain primes.

Where p is a safe prime, the DDH assumption holds in group $\mathbb{G} := \{x^2 \mod p \mid x \in \mathbb{Z}_p^\times\}$, and $\mathbb{G}$ is a provably a subgroup of $\mathbb{Z}_p^\times$ with order q.

§3.8 Message Integrity

Alice sends a message to Bob, how does Bob ensure that the message came from Alice?



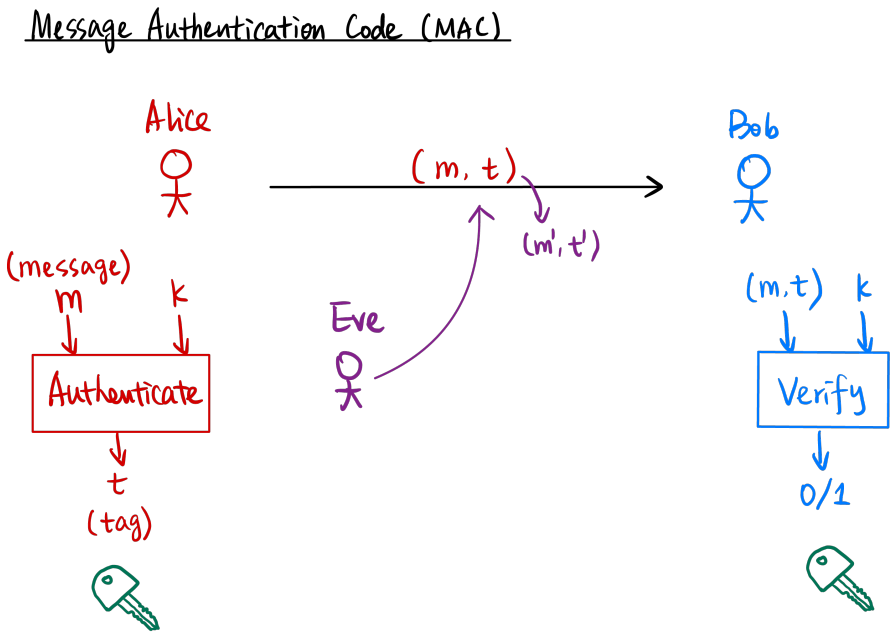
We can build up another line of protocols to ensure message integrity. It's similar to encryption, but the parties run 2 algorithms: *Authenticate* and *Verify*.

Using a message m, Alice can generate a *tag* or *signature*, and Bob can verify (m, t) is either valid or invalid.

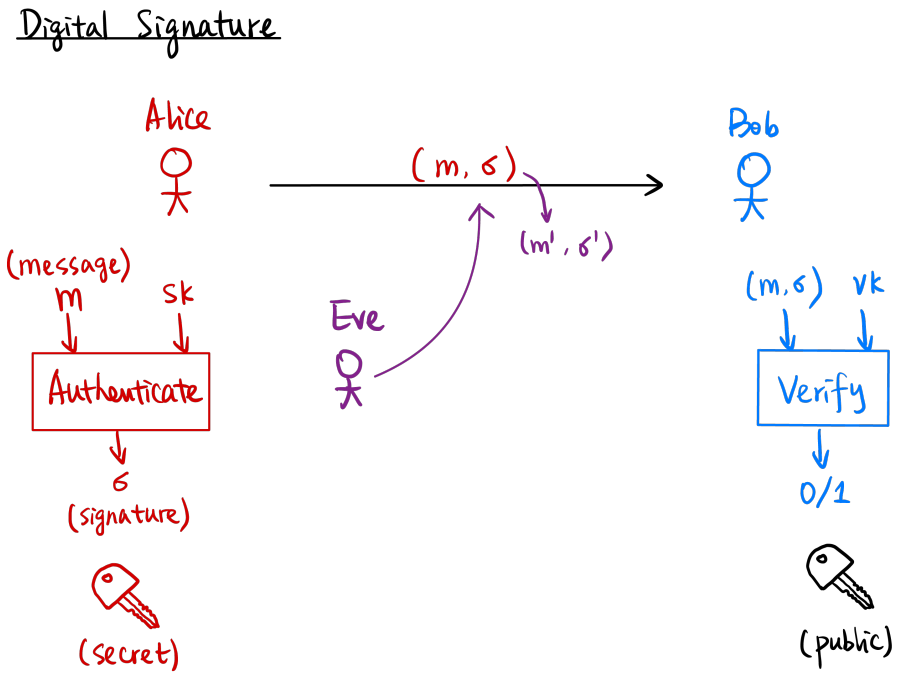
Our adversary has been upgraded to an Eve who can now tamper with messages.

Just like we have symmetric-key and public-key encryption, we also have symmetric-key and public-key authentication and verification.

Using a shared key k, Alice can authenticate m using k to get a tag k. Similarly, Bob can verify whether (m, t) is valid using k. This is called a Message Authentication Code.



Using a public key vk (verification key) and private key sk (secret/signing key), Alice can sign a message m using signing key sk to get a *signature* σ. Bob verifies (m, σ) is valid using vk. This is called a Digital Signature.



Question. Can an adversary tamper with a signed message if the adversary can see the message and signature? *Tune in next week to find out...*

§3.8.1 Syntax

The following is the syntax we use for MACs and digital signatures.

A message authentication code (MAC) scheme consists of $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$.

Generation. $k \leftarrow \mathsf{Gen}(1^\lambda)$.

Authentication. $t \leftarrow \mathsf{Mac}_k(m)$.

Verification $0/1 := \mathsf{Verify}_k(m, t)$.

A digital signature scheme consists of $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.

Generation. $(sk, vk) \leftarrow \mathsf{Gen}(1^\lambda)$.

Authentication. $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.

Verification $0/1 := \mathsf{Verify}_{vk}(m, \sigma)$.

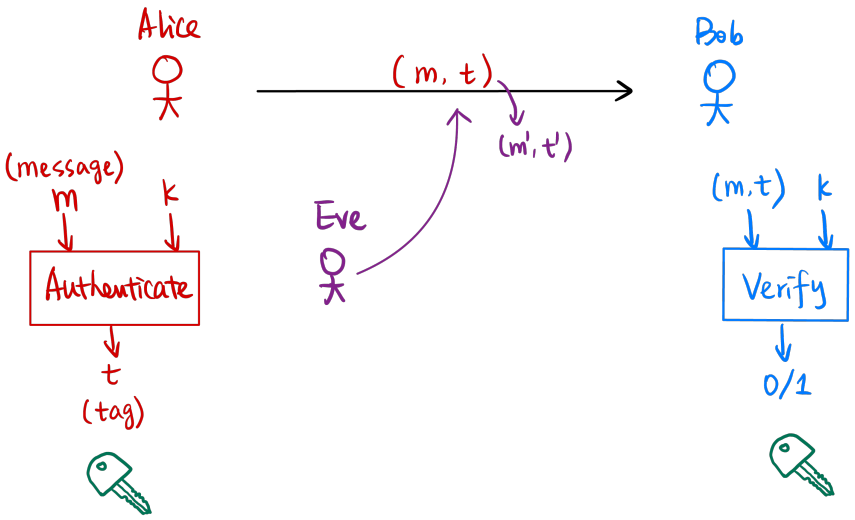
§4 February 3, 2025

§4.1 Message Integrity

Last lecture, we touched upon methods of authenticating a message. The symmetric-key version is called a MAC (message authentication code), the public-key version is called a digital signature. Let's review what we covered last time.

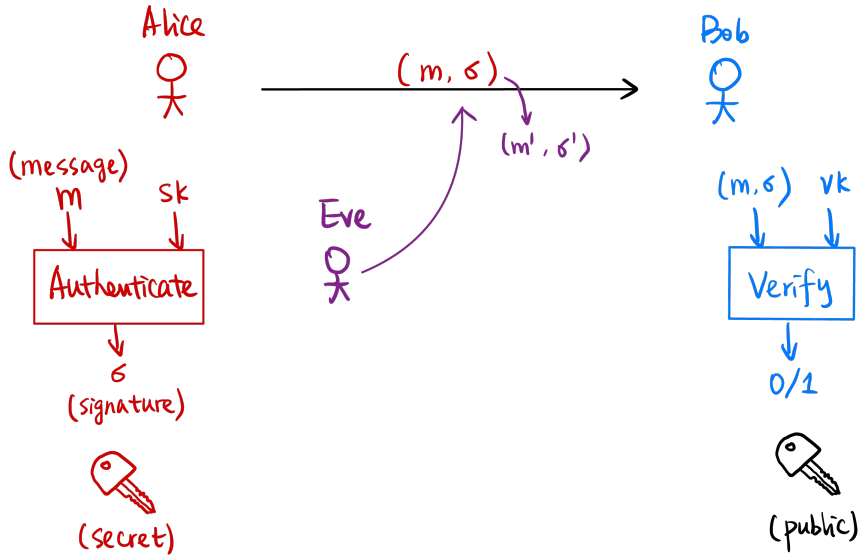
§4.1.1 Message Authentication Code

To authenticate a message, Alice will use the private key k to tag a message m with a tag t. Bob will verify that (m, t) is valid with key k.



§4.1.2 Digital Signature

Similarly we have the public-key version. Alice has a secret key sk to sign message m with signature σ. Bob (or anyone) can verify with the public key pk that (m, σ) is a valid signature.



§4.1.3 Syntax

Recall the syntax of MAC and digital signatures (see section 3.8.1).

A message authentication code (MAC) scheme consists of $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$.

Generation. $k \leftarrow \mathsf{Gen}(1^\lambda)$.

Authentication. $t \leftarrow \mathsf{Mac}_k(m)$.

Verification $0/1 := \mathsf{Verify}_k(m, t)$.

A digital signature scheme consists of $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.

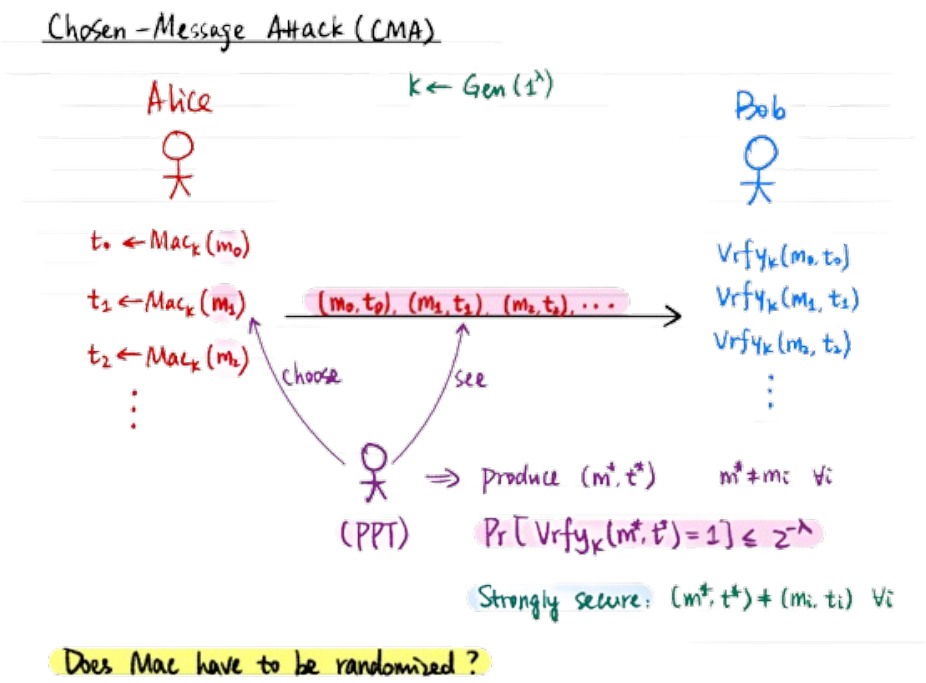
Generation. $(sk, vk) \leftarrow \mathsf{Gen}(1^\lambda)$.

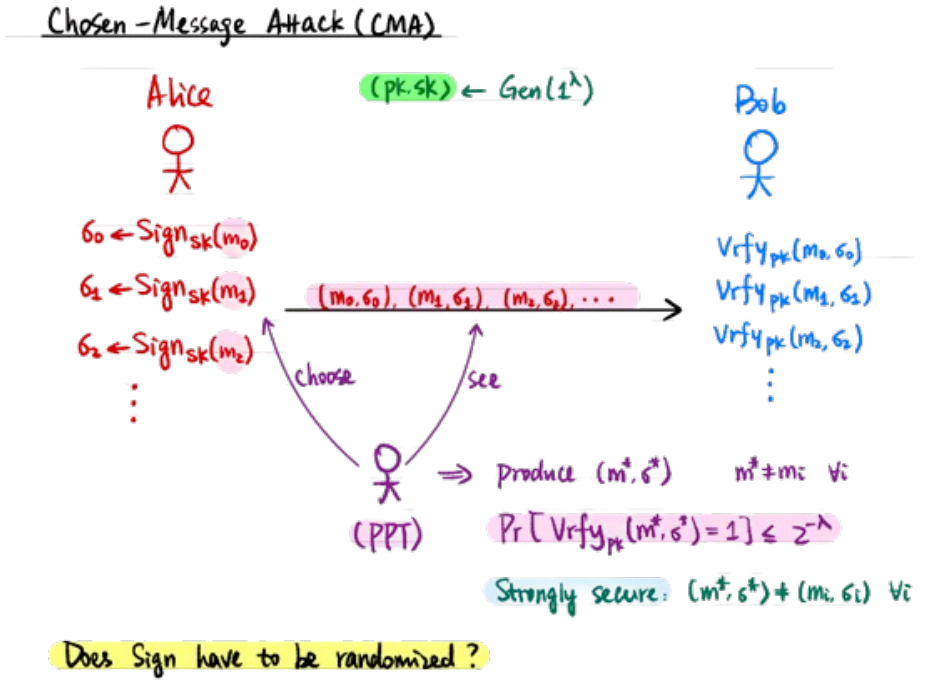
Authentication. $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.

Verification $0/1 := \mathsf{Verify}_{vk}(m, \sigma)$.

§4.1.4 Chosen-Message Attack

Similar to chosen-plaintext attack from encryption, we have chosen-message attack security. An adversary chooses a number of messages to generate signatures or tags for. After that, the adversary will try to generate another valid pair of message and tag. We want to make sure that generating a new pair of message and tag is extremely hard (negligible probability of success).





Question. Do the MAC and signature algorithms need to be randomized to be CMA secure?

No! Unlike CPA security, we don't need these algorithms to be randomized to be CMA secure. It is ok for these algorithms to be deterministic, since it is still difficult for an adversary to produce a *new* message-signature pair.

§4.2 RSA Signatures

Our RSA signatures algorithm works very similarly to RSA encryption.

We generate two n-bit primes p, q. Compute $N := p \cdot q$ and $\phi(N) = (p-1)(q-1)$. Again choose e with $\gcd(e, \phi(N)) = 1$ and invert $d = e^{-1} \mod \phi(N)$. Given N and a random $y \xleftarrow{\$} \mathbb{Z}_N^\times$, it's computationally hard to find x such that $x^e \equiv y \mod N$.

Similarly, $sk := d$ and $vk := (N, e)$. To sign, we compute

$$\mathsf{Sign}_{sk}(m) := m^d \mod N.$$

To verify, we compute

$$\mathsf{Verify}_{vk}(m, \sigma) := \sigma^e \stackrel{?}{\equiv} m \mod N.$$

Question. Are there any security issues with RSA as we have constructed it so far?

Yes, there are several attacks that can be leveraged. A simple on is to sample some $\sigma^* \in \mathbb{Z}_N^*$, and then compute $m^* := (\sigma^*)^e$.

Attacks can be more targeted. If Eve knows many messages and signatures, she can compute another pair of valid message and keys. If we have messages

$$m_0, \sigma_0 = m_0^d \mod N$$
$$m_1, \sigma_1 = m_0^d \mod N$$

We can compute $m^* := m_0 \cdot m_1$ and $\sigma^* := \sigma_0 \cdot \sigma_1 = (m_0 \cdot m_1)^d \mod N$.

We can do linear combinations of messages, as well as raising messages to arbitrary exponents, and we can get other messages with valid signatures.

There is an easy solution, however. We can hash our message m before we sign, like so

$$\mathsf{Sign}_{sk}(m) := H(m)^d \mod N$$
$$\mathsf{Verify}_{vk}(m, \sigma) := \sigma^e \stackrel{?}{\equiv} H(m) \mod N$$

where H is a hash function[11]. This is a commonly known technique called 'hash-and-sign'.

§4.2.1 Other Signature Schemes

There are also other signature schemes that rely on other hardness assumptions.

- RSA signatures rely on the RSA assumption.

[11] A hash function is, briefly, a function that produces some random output that is hard to compute the inverse of.

- Schnorr/DSA signatures rely on the discrete log assumption.
- Lattice-based encryption schemes are post-quantum secure and rely on the hardness of certain lattice problems.

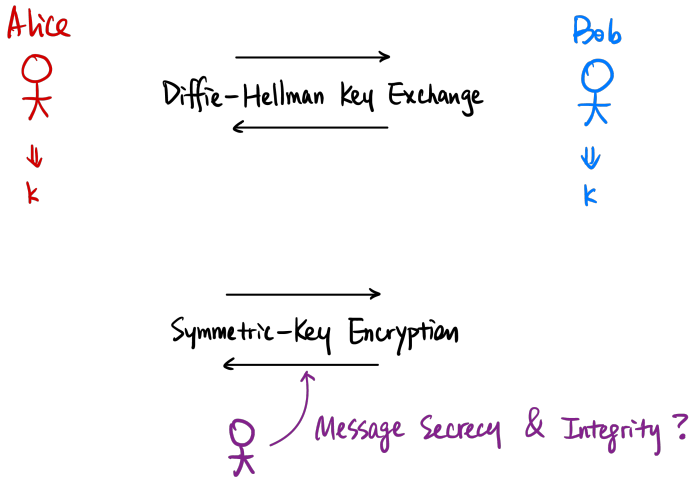
§4.3 A Summary So Far

To summarize, here's all we've covered so far:

	Symmetric-Key	**Public-Key**
Message Secrecy	Primitive: SKE Construction: Block Cipher	Primitive: PKE Constructions: RSA/ElGamal
Message Integrity	Primitive: MAC Constructions: CBC-MAC/HMAC	Primitive: Signature Constructions: RSA/DSA
Secrecy & Integrity	Primitive: AE Construction: Encrypt-then-MAC	
Key Exchange		Construction: Diffie-Hellman
Important Tool	Primitive: Hash function Construction: SHA	

§4.4 Authenticated Encryption

Generally, Alice and Bob will first perform a Diffie-Hellman key exchange, then use that shared key to conduct Symmetric-Key Encryption.



In reality, we want to achieve both message secrecy and integrity *at the same time*. For this, we can introduce Authenticated Encryption.

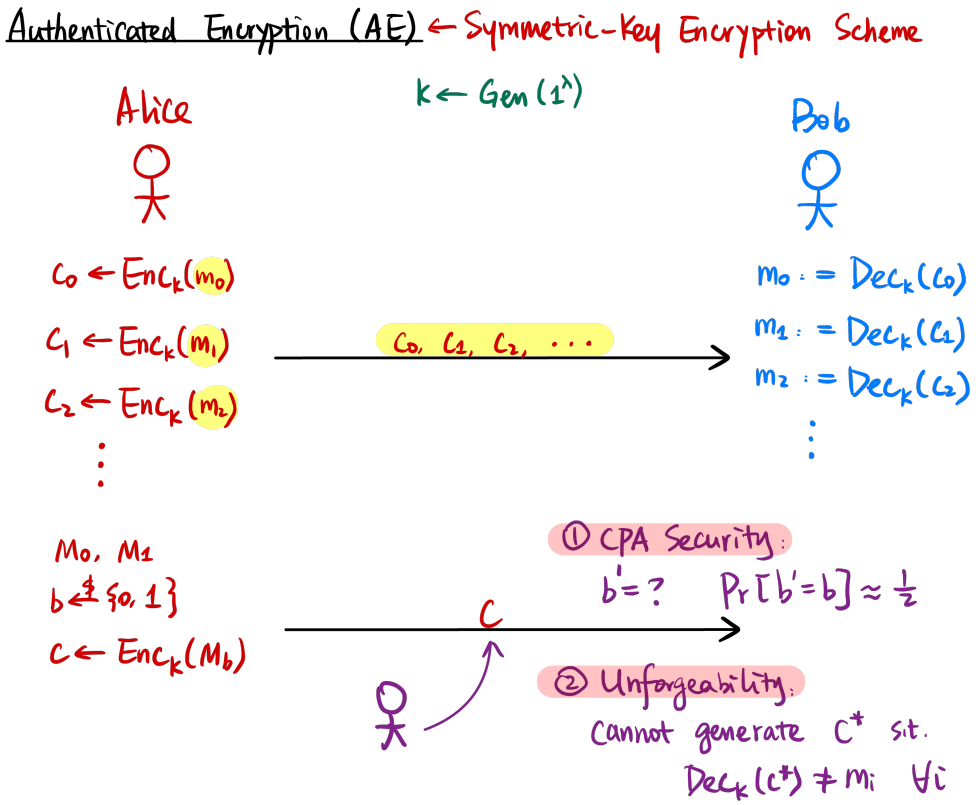
Our security definition is that our adversary can see the encryptions of many messages m_0, m_1, m_2. We want **CCA security**: that an adversary given previous ciphertexts and their decryptions cannot distinguish between the encryptions of a fresh pair of messages m_0 and m_1. *Additionally*, we want the property of **unforgeability**, that our adversary cannot generate a c^* that is a valid encryption, such that $\mathsf{Dec}_k(c^*) \neq m_i$ for any i.

Definition 4.1 (Chosen Ciphertext Attack)

An adversary is allowed to query any number of messages, and receives their corresponding ciphertexts. The adversary can also request decryptions from a decryption oracle for any ciphertext, and will see the resulting decryption.

Definition 4.2 (Unforgeability)

An adversary can query any message. They are unable to create a new message that has not been queried before, and which can be authenticated.



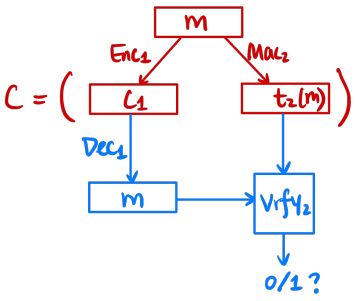
Now that we have two new primitives, we can construct Authenticated Encryption schemes.

Remark 4.3. This section describes three different approaches to AE: Encrypt-and-MAC, Encrypt-then-MAC, and MAC-then-Encrypt. In practice, only Encrypt-then-MAC satisfies CCA security and unforgeability. The other approaches are provided as counterexamples.

§4.4.1 Encrypt-and-MAC?

Given a CPA-secure SKE scheme $\Pi_1(\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ and a CMA-secure MAC scheme $\Pi_2 = (\mathsf{Gen}_2, \mathsf{Mac}_2, \mathsf{Verify}_2)$.

We construct an AE scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ by composing encryption and MAC. We encrypt the plaintext and also compute the MAC the *plaintext*.



$\mathsf{Gen}(1^\lambda)$: $k_1 := \mathsf{Gen}_1(1^\lambda)$, $k_2 := \mathsf{Gen}_2(1^\lambda)$, output (k_1, k_2).

$\mathsf{Enc}(m)$: To encrypt, we first encrypt ciphertext $c_1 := \mathsf{Enc}_1(k_1, m)$ and sign message (in plaintext) $t_2 := \mathsf{Mac}_2(k_2, m)$ and output (c_1, t_2).

$\mathsf{Dec}(m)$: We have ciphertext $c = (c_1, t_2)$. Our message is $m := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (m, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

Question. Is this scheme secure? Assuming the CPA-secure SKE scheme and CMA-secure MAC scheme, does this give us both CPA-security and unforgeability?

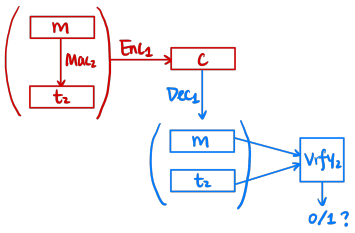
MAC gives you *unforgeability*—but it doesn't even try to hide the message at all. It's possible that the MAC scheme reveals the message in the clear. For example, we might have a MAC scheme that includes the message in the signature *in the clear* (which is still secure)!

Since MAC doesn't try to hide the message. If our MAC reveals something about our message, our composed scheme Π doesn't give us CPA-security. You might still be able to infer something about the message.

We try something else...

§4.4.2 MAC-then-Encrypt?

Similarly, we can also MAC first, encrypt the entire ciphertext and tag concatenated.



$\mathsf{Gen}(1^\lambda)$: $k_1 := \mathsf{Gen}_1(1^\lambda)$, $k_2 := \mathsf{Gen}_2(1^\lambda)$, output (k_1, k_2).

$\mathsf{Enc}(m)$: To encrypt, we first sign message (in plaintext) $t_2 := \mathsf{Mac}_2(k_2, m)$ and then encrypt ciphertext and tag $c_1 := \mathsf{Enc}_1(k_1, m||t_2)$ and output c_1.

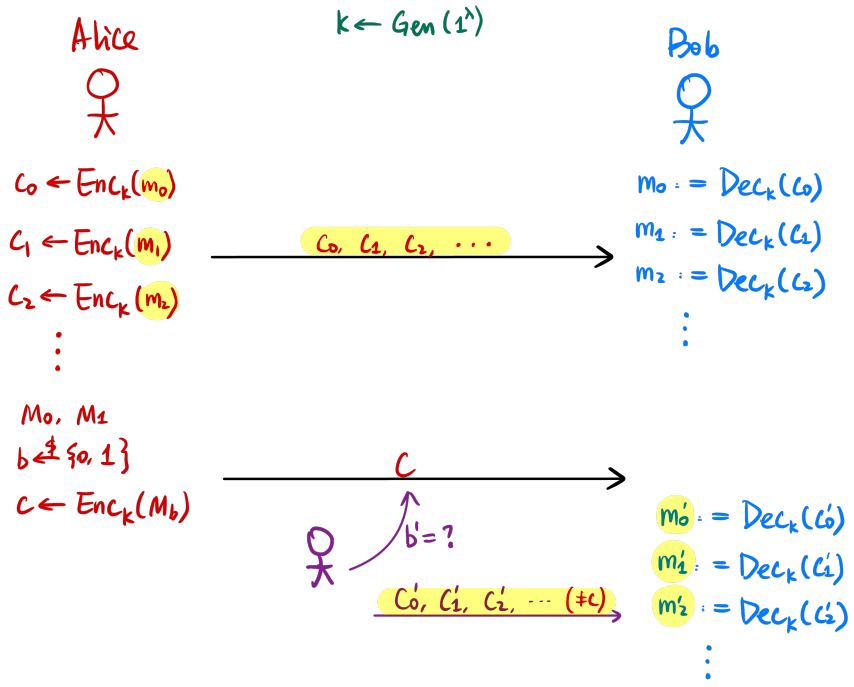
$\mathsf{Dec}(m)$: We have ciphertext c_1. Our message is $m||t_2 := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (m, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

Question. Is this secure?

This doesn't satisfy a stronger security definition called Chosen Ciphertext Attack (CCA) security. We might be able to forge ciphertexts that decrypt to valid message and tags.

§4.4.3 Chosen Ciphertext Attack Security

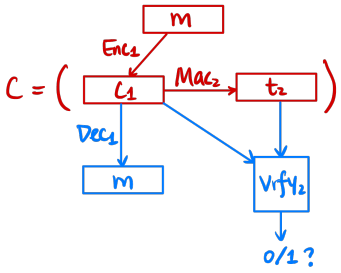
On top of CMA security, the adversary can now request Alice to decrypt ciphertexts $c_0, c_1, \ldots$.



We can prove that MAC-then-Encrypt is not CCA secure.

§4.4.4 Encrypt-then-MAC

We encrypt first, then we MAC on the *ciphertext*.



$\mathsf{Gen}(1^\lambda)$: $k_1 := \mathsf{Gen}_1(1^\lambda)$, $k_2 := \mathsf{Gen}_2(1^\lambda)$, output (k_1, k_2).

$\mathsf{Enc}(m)$: To encrypt, we first encrypt ciphertext $c_1 := \mathsf{Enc}_1(k_1, m)$ and sign message (in ciphertext) $t_2 := \mathsf{Mac}_2(k_2, c_1)$ and output (c_1, t_2).

$\mathsf{Dec}(m)$: We have ciphertext $c = (c_1, t_2)$. Our message is $m := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (c_1, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

You can prove that Encrypt-then-MAC schemes are CPA-secure and unforgeable. In addition, Encrypt-then-MAC is CCA secure, since our decryption oracle will not decrypt if the ciphertext cannot be verified, meaning only valid ciphertext-tag pairs can be passed.

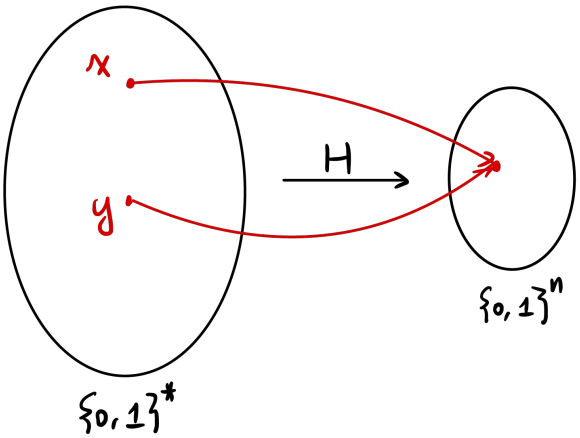
The moral of this is that **you should always use Encrypt-then-MAC.**

§4.5 Hash Function

A hash function is a public function

$$H : 0,1^* \to 0,1^n$$

where n is order $\Theta(\lambda)$.



We want our hash function to be collision-resistant. That is, it's computationally hard to find $x, y \in \{0,1\}^*$ such that $x \neq y$ yet $H(x) = H(y)$ (which is called a collision).

More to come next class!

§5 February 05, 2025

§5.1 Hash Function, *continued*

How might one find a collision for function $H : \{0,1\}^* \to \{0,1\}^n$. We can try $H(x_1), H(x_2), \ldots, H(x_q)$.

If $H(x_1)$ outputs a random value, $0,1^n$, what is the probability of finding a collision?

If $q = 2^n + 1$, our probability is exactly 1 (by pigeon-hole). If $q = 2$, our probability is $\frac{1}{2^\lambda}$ (we have to get it right on the first try). What q do we need for a 'reasonable' probability?

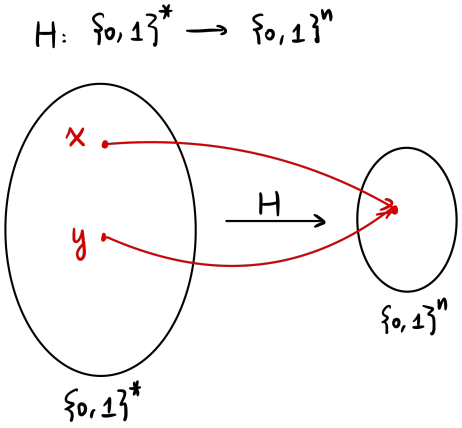
> **Remark.** This is related to the birthday problem. If there are q students in a class, assume each student's birthday is a random day $y_i \xleftarrow{\$} [365]$. What is the probability of a collision? $q = 366$ gives 1, $q = 23$ gives around 50%, and $q = 70$ gives roughly 99.9%.
>
> We can apply this trick to our hash function. If $y_i \xleftarrow{\$} [N]$, then $q = N + 1$ gives us 100%, but $q = \sqrt{N}$ gives 50% probability.

Knowing this, we want $n = 2\lambda$ (output length of hash function). If $\lambda = 128$, we want n to be around 256.

§5.2 Collision-Resistant Hash Function (CRHF)

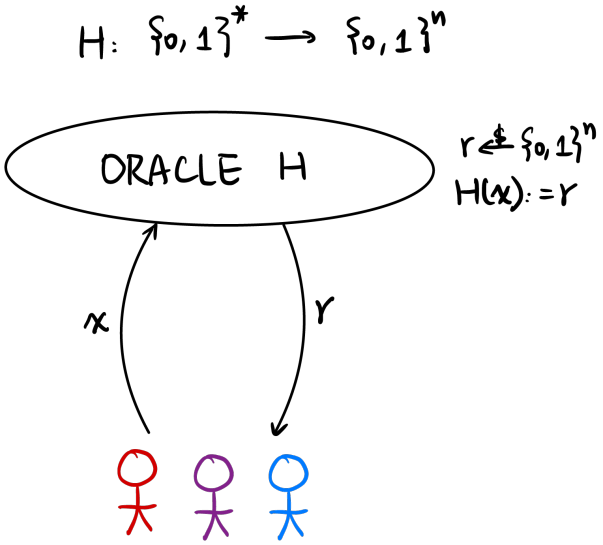
Recall that we defined a hash function to be a public and deterministic function for which it is computationally hard to find a collision. That is, finding two distinct strings x, y such that $H(x) = H(y)$ is computationally difficult.

For the hash function, the input domain is arbitrary-length string, and the output is a fixed-length string - 256 bits. The security guarantee is Collision-Resistant Hash Function(CRHF).



§5.2.1 Random Oracle Model

Another way to model a hash function is the *Random Oracle Model*. We think of our hash function to be an oracle (in the sky) that can *only* take input and a random output (and if you give it the same input twice, the same output).



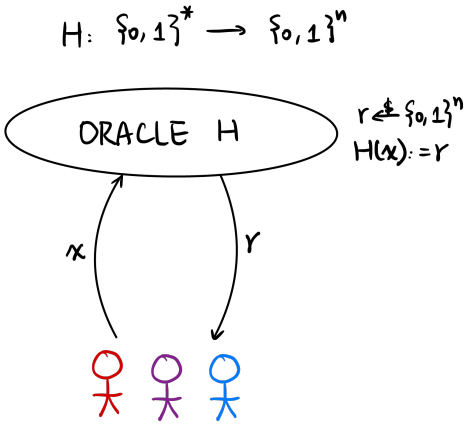
There are proofs that state that no hash functions can be a random oracle. There are schemes that can be secure in the random oracle model, but are not using hash functions[12].

In reality, hash functions are *about as good as*[13] random oracles. Thinking of our hash functions as random oracles gives us a good intuitive understanding of how hash functions can be used in our schemes.

In this model, the best thing that an attacker can do is to try inputs and query for outputs.

If you are given an arbitrary output, it's extremely hard to find its input. But it is not the case for CRHF, since CRHF only assumes you can not find a collision, thus this model is tronger than CRHF.

[12] Some constructions don't rely on this model.
[13] But can never be...



§5.2.2 Constructions for Hash Function

MD5. Output length 128-bit. Best known attack is in 2^{16}. A collision was found in 2004.

And we also have Secure Hash Functions (SHA), founded by NIST.

SHA-0. Standardized in 1993. Output length is 160-bit. Best known attack is in 2^{39}.

SHA-1. Standardized in 1995. Output length is 160-bit. Best known attack is in 2^{63}, and a collision *was* found in 2017.

SHA-2. Standardized in 2001. Output length of 224, 256, 284, 512-bit. The most commonly used is SHA-256.

SHA-3. There was a competition from 2007-2012 for new hash functions. SHA-3 was released in 2015, and has output length 224, 256, 2384, 512-bit. This is *completely different* from SHA-2.

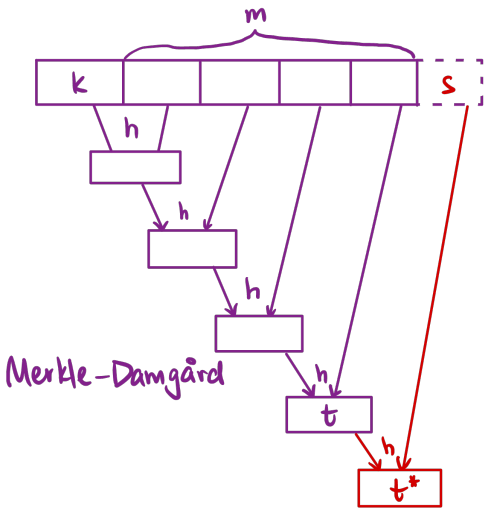
> **Remark.** The folklore is that during a session at a cryptography conference, a mathematician, Xiaoyun Wang, presented slide-after-slide of attacks on MD5 and SHA-0, astounding the audience.

§5.2.3 Applications

HMAC. We can use a hash function to conduct a MAC. Computing a tag involves computing the hash function on the key appended to the message $(k||m)$. It is computationally difficult to find another $k||m'$ that produces the same hash. This is a scheme that looks like

$$\mathsf{Mac}_k(m) = H(k||m).$$

However, an adversary could potentially attach some additional s to m to produce $m' = m||s$ such that they can easily compute $\mathsf{tag}' = H(\mathsf{tag}||s)$. This is due to the Merkle-Damgård construction of SHA-2, which associatively tags blocks of the message one-by-one.



Therefore, in practice, we use a nested MAC like

$$\mathsf{Mac}_k(m) = H(k||H(k||m))$$

and just to be sure (that we're not reusing the key), we produce k_1, k_2 as such

$$\mathsf{HMAC}_k(m) = H(k_1||H(k_2||m))$$

such that $k_1 = k \oplus \mathsf{opad}$ and $k_2 = k \oplus \mathsf{ipad}$, some one-time pads.

Hash-and-Sign. There are some other applications of a hash function. We've seen before with RSA that we want to Hash-and-Sign, removing any homomorphism that an adversary could exploit. Additionally, this allows us to sign larger messages since they are constant size after hashing.

Password Authentication. Another application is password authentication. Instead of storing plaintext passwords on servers, websites can store a hash of the password instead. This means that the passwords are not compromised even if the server is compromised.

Deduplicate Files. We can also use hash functions to deduplicate files. We can hash two files to produce identifiers h_1 and h_2. If $h_1 \neq h_2$, this implies $D_1 \neq D_2$. If $h_1 = h_2$, it almost always[14] implies that $D_1 = D_2$.

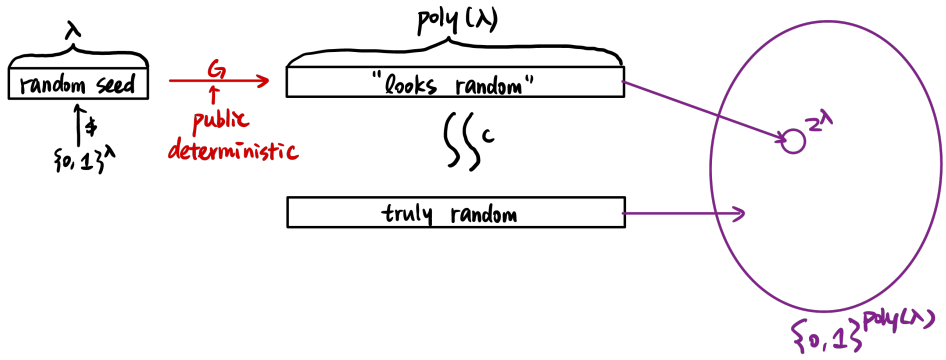
HKDF (Key Derivation Function). We can derive more keys from a shared key, essentially using a hash function as a pseudorandom generator (PRG).

For example, if there is g^{ab} shared key, we can do

$$\mathsf{HMAC}(g^{ab}, \mathsf{salt})$$

Using a random seed, and concatenating a public deterministic salt G, we can generate a random[15] string.

[14] If they are not equal, we've found a collision for our hash function, which is extremely unlikely.
[15] Computationally random, because if our computational power were to be unbounded, we can try all strings.



Pseudorandom Generator (PRG). Given a hash function H, we can generate a PRG easily for any length string by generating

$$\begin{aligned} \mathsf{seed} \xleftarrow{\$} \{0,1\}^\lambda \longrightarrow & H(\mathsf{seed}||00\cdots00) \\ & H(\mathsf{seed}||00\cdots01) \\ & H(\mathsf{seed}||00\cdots10) \\ & \vdots \end{aligned}$$

We can take a bit of randomness (like the way we move our mouse, type keyboard, system properties) and generate our seed.

Fast Membership Proof (Merkle Tree). Using hash functions, we can generate Merkle Trees to prove membership. In blockchains, this is equivalent to checking if a transaction occurred.

SKE Scheme? Could we use this to encrypt? If we have a secret key $k \xleftarrow{\$} \{0,1\}^\lambda$, can we just encrypt by

$$\mathsf{Enc}_k(m) = H(k||m)$$

Well, we can't decrypt for one without having unbounded computational power. If our plaintext m comes from a small set, like $\{0,\ldots,10\}$, we could decrypt properly. However, this is not CPA-secure, since the adversary could just query for all the messages.

> **Remark 5.1.** In general, all deterministic encryption schemes are not CPA-secure.

§5.3 Putting it Together: Secure Communication

This is essentially what we want to do in the second project.

We use Diffie-Hellman Key Exchange between Alice and Bob to get shared g^{ab}. Hashing the shared key using an HKDF, we can get shared key $k = (k_1, k_2)$ (one for AES encryption, one for HMAC). Then, they perform authenticated encryption, namely Encrypt-then-MAC.

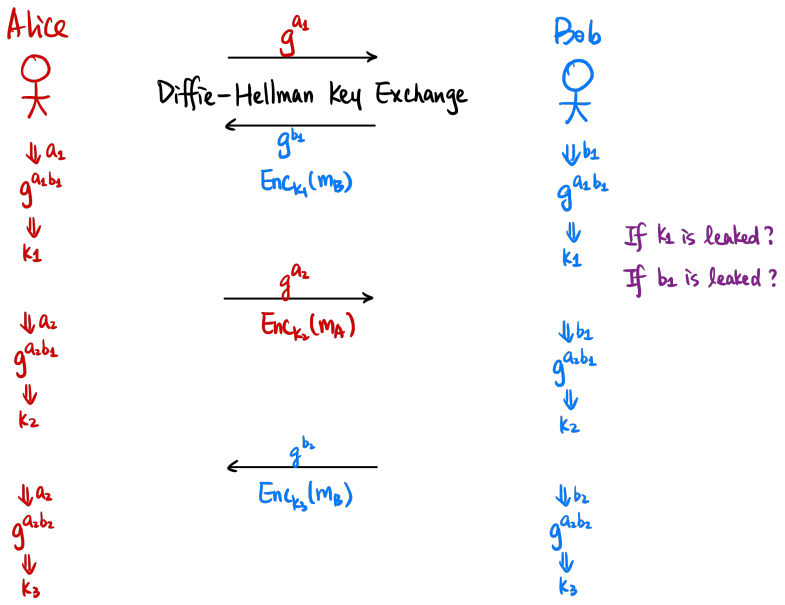
Question. Are there any issues with this scheme?

An Eve could pretend to be Alice to Bob and Bob to Alice, fudging up their shared keys. This is called a *Man-in-the-Middle* attack.

§5.3.1 Diffie-Hellman Ratchet

What if a secret key gets leaked, or cracked? One simple way to fix this is to perform a Diffie-Hellman key exchange on every message. However, this incurs additionall communications costs.

Here's another idea: with every new message (when the direction of communications shifts), the party sending the message sends a new Diffie-Hellman public key for themselves. For example, if Bob is sending a message to Alice and he knows Alice's public key g^{a_1} and his previous secret was b_1 (hence shared $g^{a_1 b_1}$), Bob will generate new key b_2, g^{b_2} and encrypt using $g^{a_1 b_2}$, sending g^{b_2} as public to Alice. Alice can recompute the shared key before decrypting.



This is the protocol used in the Signal messaging app, and is what you will implement for Project 1.

Question. What if k_1 is leaked?

We might have leaked one key, but the other keys are still computationally hard to compute. $k_1 = g^{a_1 b_1}$ is known, but it's equivalent to DDH to compute $g^{a_1 b_2}$ or other keys.

Question. What if b_1 is leaked?

We can compute key $k_1 = g^{a_1 b_1}$ and $k_2 = g^{a_2 b_1}$, but no further keys are leaked, and the next round of communications (after Bob refreshes his private key b_2) is still secure.

§5.4 Block Cipher

To summarize, here's what we've seen so far (this table should be familiar):

	Symmetric-Key	**Public-Key**
Message Secrecy	Primitive: SKE Construction: **Block Cipher**	Primitive: PKE Constructions: RSA/ElGamal
Message Integrity	Primitive: MAC Constructions: CBC-MAC/HMAC	Primitive: Signature Constructions: RSA/DSA
Secrecy & Integrity	Primitive: AE Construction: Encrypt-then-MAC	
Key Exchange		Construction: Diffie-Hellman
Important Tool	Primitive: Hash function Construction: SHA	

The only thing we haven't seen thus far is a block cipher. We first start with the definitions.

We saw earlier that a Pseudorandom Generator (PRG) produces a string that looks random. We also have Pseudorandom Functions (PRF), which are 'random-looking' functions.

§5.4.1 Pseudorandom Function (PRF), *continued*

A block cipher, at a very high level, is a pseudo random function.

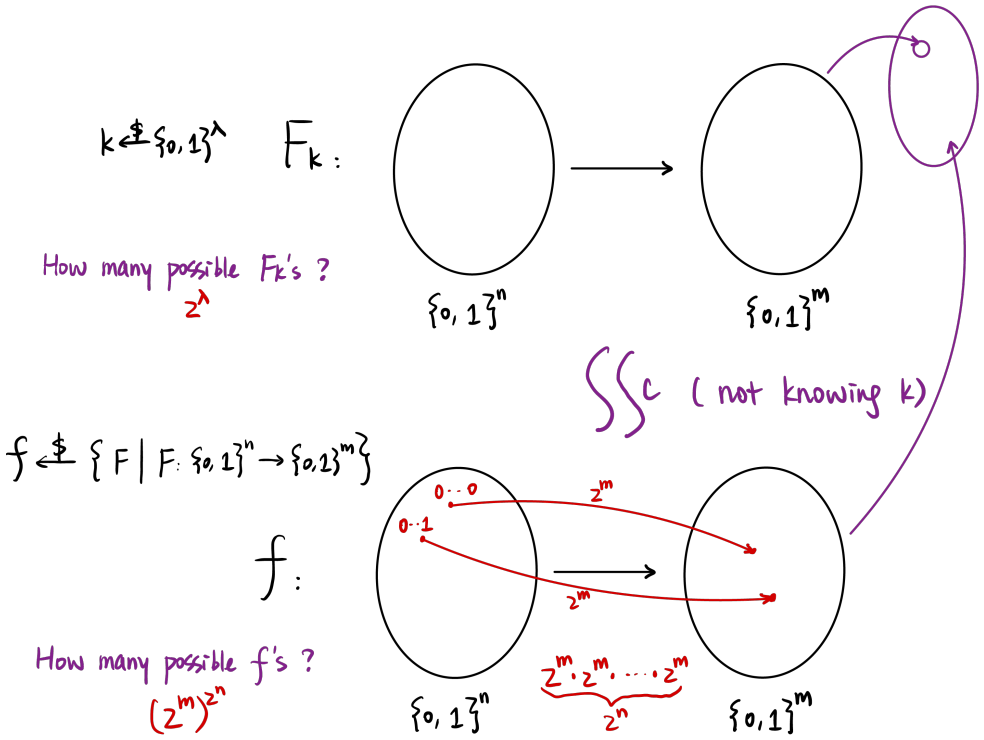
Recall last time that we talked about pseudorandom *generators* (PRG), which takes a seed and expands it into a long string of pseudorandom bits. This "random-looking" **string** is computationally indistinguishable from a truly random string.

A pseudorandom *function* (PRF) is a "random-looking" **function** that takes a key and an input and produces an output. This function is computationally indistinguishable from a truly random function.

More formally, our pseudorandom function F is a keyed function[16] $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^m$, F will take key k and input x to produce output y, $F(k,x) = y$.

Without knowing our key k, F_k is computationally indistinguishable from some random $f \xleftarrow{\$} \{F \mid F\{0,1\}^n \to \{0,1\}^m\}$.

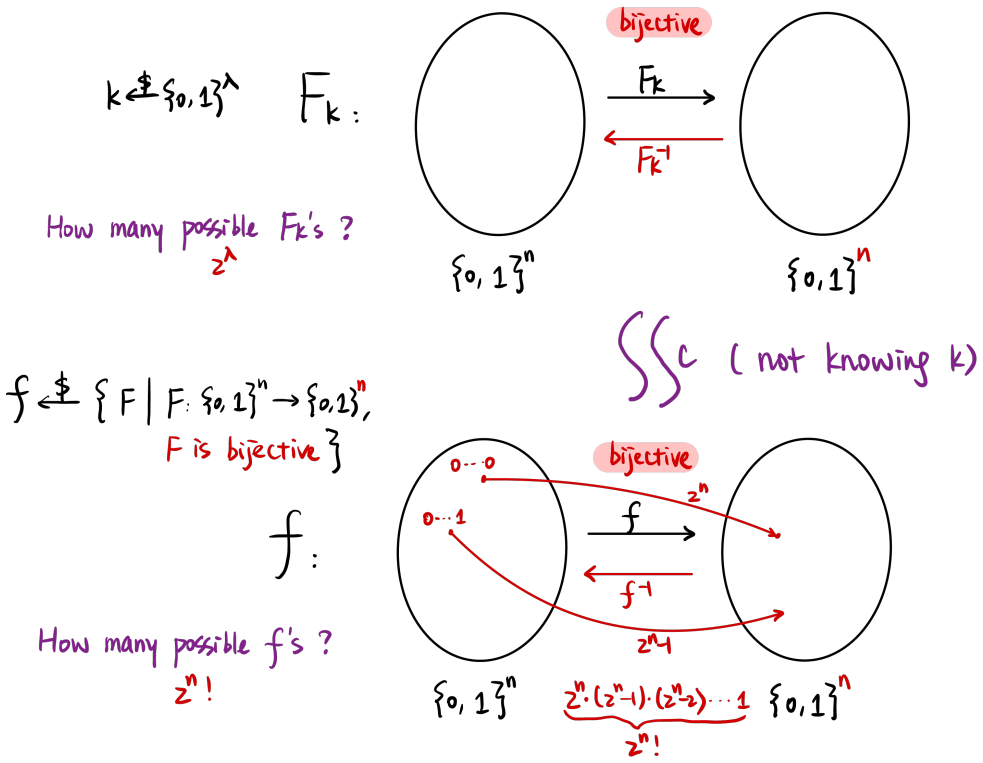
[16] In deterministic polynomial-time.



We have 2^λ possible F_k's, and we have $(2^m)^{2^n}$ possible functions f. A computationally unbounded adversary could try all possible functions and distinguish our function, since F_k lives in a subset of the space of f. However, in reality, we can assume that F_k is computationally indistinguishable from any generic function.

§5.4.2 Pseudorandom Permutation (PRP)

A further assumption is that our function is a bijection. F_k is a keyed function from $F_k : \{0,1\}^n \to \{0,1\}^n$. We still have 2^λ possible F_k's since there are 2^λ



Question. Again, how many possible f's are there?

Our first string has 2^n choices to map to, our second choice has $2^n - 1$, so there are

$$(2^n)(2^n - 1)(2^n - 2) \cdots 1 = 2^n!$$

Still, this is a much larger number than 2^λ, so we still make a computational assumption that our keyed function F_k is still computationally indistinguishable from a random function f.

§6 February 10, 2025

§6.1 Recap

Last lecture, we introduced Pseudorandom Functions (PRFs) and Pseudorandom Permutations (PRPs). We also introduced the concept of a block cipher, which is a special form of a PRP. Refer to last lecture's notes for a refresher.

§6.2 History of AES and DES

AES (Advanced Encryption Standard) was standardized in 2001 by NIST, and is a block cipher with a block length of 128 bits and key lengths of 128, 192, and 256 bits. Its predecessor, DES, had a block length of 64 bits and a key length of 56 bits - the best attack on DES is still a brute force attack.

§6.3 Block Ciphers

Looking back on section 4.3, the last outstanding primitive was the block cipher. We saw this last lecture, we'll continue discussing the block cipher.

Recall that we had seen pseudorandom functions which are keyed functions that are computationally indistinguishable from *all* random functions from $\{0,1\}^n \to \{0,1\}^m$. A stronger form of pseudorandom functions are pseudorandom permutations: a keyed bijective map between $\{0,1\}^n \to \{0,1\}^n$ that is computational indistinguishable from pseudorandom permutations.

Block ciphers are a special form of pseudorandom permutation. It is a keyed function

$$F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$$

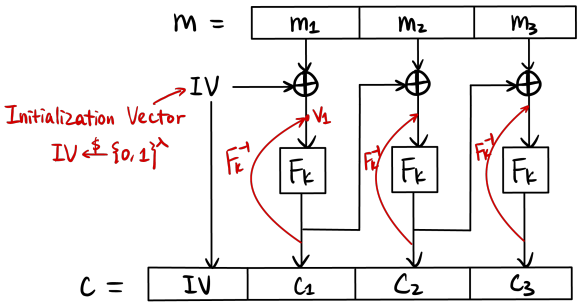
where λ is the key length and n is the block length. The practical construction of which is AES, which takes blocks of $n = 128$ and key length $\lambda = 128, 192, 256$ as choices.

§6.3.1 Modes of Operation

Electronic Code Book (ECB) Mode: We will run our block cipher on each block of our message individually. However, this is not CPA secure, since encryptions are deterministic. We need to 'seed' our encryption with some random value.

In summary: not CPA secure.

Cipher Block Chaining (CBC) Mode: Instead of running on our block cipher on each block individually, every block will get an additional *initialization vector* IV, which is XORed onto each message before running the block cipher.



We waved our hand over the fact that this is CPA secure—but it relies on the initialization vector being random.

What if our IV is not randomly sampled? Consider an IV that is *different* but not randomly sampled. For example, the IV is $0\cdots00$ for the first message, $0\cdots01$ for the second message, and so on. Do we still have security?

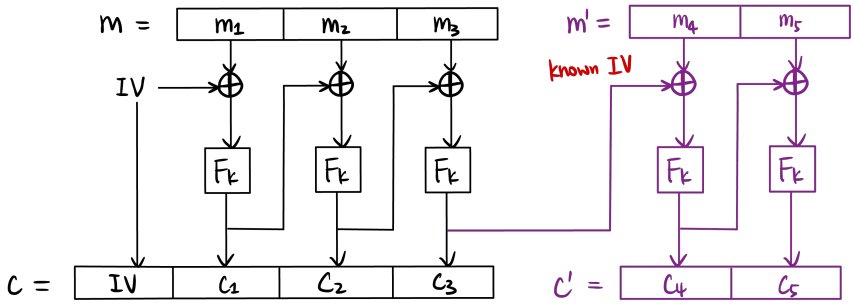
Unfortunately not. Say m_1 is XORed onto $0\cdots01$, an adversary under CPA can choose plaintext that is m_1 with its last bit flipped, such that v_1 is manipulated and the block cipher is again deterministic.

It is crucial that IV is randomly selected, and that the next IVs for future blocks (of the same message) are also pseudorandom (that are the previous ciphertext, which is okay).

Can we parallelize the computation? No, since we need the previous ciphertext to XOR onto the next block. This is a downside of CBC mode.

In summary: CPA secure, but non-parallel.

Chained Cipher Block Chaining (Chained-CBC) Mode: There is a mode of operation of CBC that feeds the last cipher block as the new IV for the next message.



Similar to the case earlier, an adversary here can select a next message *based on* their knowledge of the previous ciphertext and hence the upcoming IV.

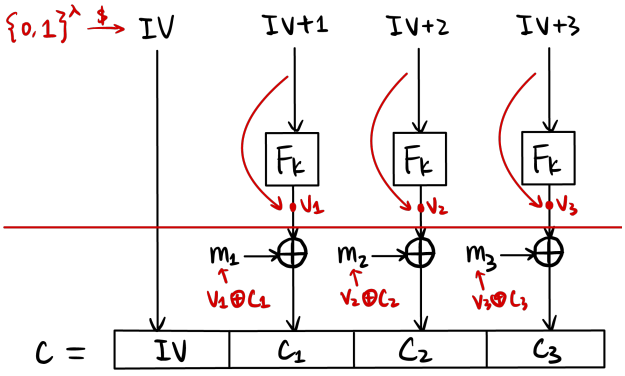
This makes chained-CBC *very subtly* different than CBC. If we squint our eyes enough, it just looks like sending a single message using CBC mode. The key difference is that between rounds of communication m and m', an adversary could influence m' given the knowledge of the previous round.

> **Remark.** Another note that this is *very subtle*! To the extent that when *Signal* was being developed, the course staff initially wrote the solution using Chained-CBC mode. This highlights the difficulty in creating real-world cryptographic systems!

In summary: not CPA secure.

Counter (CTR) Mode: Instead of chaining each successive IV from the previous block ciphertexts, we'll encrypt *only* the IV $\xleftarrow{\$} \{0,1\}^\lambda$, and XOR the encrypted $F_k(\text{IV}+i)$ to mask m_i, like a one-time pad.

Another way to think about the CTR mode is that we're using F_k and a random IV to generate a long enough one-time pad to pad the entire message.



How do we decrypt? Since we know the first IV, we can compute the one-time pads $F_k(\text{IV}+i)$ and XOR with m_is. This scheme is valid.

Is this CPA secure? The XOR *after* F_k might throw you off and cast doubt in your mind. However, this mode of operation is CPA-secure. Even if we know IV, $\text{IV}+1, \text{IV}+2, \cdots$, we can't figure out the output of F_k that becomes our one-time pad (to do so constradicts the CPA security of our block cipher). The CPA security of each F_k being pseudorandom guarantees the CPA security of this scheme.

What about a "stateful CTR mode" which just increments IV every successive time? Instead of sending a new IV for the next message, we'll just increment the IV from before. Similar to Chained-CBC mode, the adversary will know the IV that is going into the next message. However, this

doesn't *really* help the adversary. They've never seen those encrypted IV values before, and hence cannot modify the message given this information.

This is a distinction from last time, where the IV was XORed onto the message directly, which could be tampered with by an adversary who knows the IV.

What if IV is not randomly sampled? Nothing really breaks down, unlike the previous case. We just want to make sure that two IVs are not reused and don't collide. If IVs collide, two blocks will have the same one-time pad, which is potentially a problem. This doesn't prevent us from using $0\cdots00, 0\cdots01, 0\cdots10, \ldots$ as our IV values at all. In practice, however, they are still randomly sampled to prevent collisions.

Can we parallelize this? Yes, we can compute $F_k(\text{IV}+i)$ in parallel and XOR onto each block. Similar for encryption and decryption.

Can we construct a PRG from a PRF? Using a seed (IV, k), we can generate an $n\lambda$ bit string

$$G(k||\text{IV}) = F_k(\text{IV})||F_k(\text{IV}+1)||F_k(\text{IV}+2)||\cdots$$

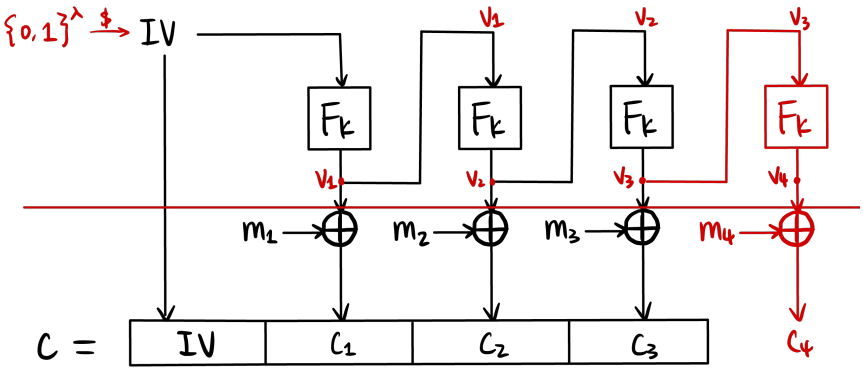
In fact, we can get rid of IV entirely and start at 0,

$$G(k) = F_k(0)||F_k(1)||F_k(2)||\cdots$$

Counter mode essentially uses this PRG with private k to generate a long one-time pad which is used to pad the message. Another note is that in this mode, we don't even require a pseudorandom permutation, since we don't need to invert the function at any point.

In summary: CPA secure, parallel.

Output Feedback (OFB) Mode: This is a mix of CBC and CTR modes. Successive one-time pad blocks are fed into the next F_k as IV, and they are XORed with the message after encryption.



We have the same questions. *How do we decrypt? Is this CPA secure? Is a "stateful" version of OFB secure? Can we use this to construct a PRG?*

We can decrypt similarly: we decrypt the first block, get the IV for the next block and continue on. All security is guaranteed by the same reasoning as in counter mode: we know IV but still cannot compute $F_k(\text{IV})$. Similar to counter mode, this is another form of PRG (which chains successive blocks instead of using IVs in series) that generates a long one-time pad. Again, our IV doesn't need to be randomly sampled, but it should not collide with previous IV values.

A difference to counter mode is that we cannot parallelize this scheme. However, in both CTR and OFB modes, we can precompute the entire one-time pad in both encryption and decryption to happen in the offline phase. The online phase (when parties are communicating) is limited to cheap XOR operations.

Question. We've listed *a lot* of benefits to counter mode or output feedback mode. Why do people use CBC mode at all?

We've seen how things can go wrong catastrophically[17]. This is more true for counter mode than CBC mode. If our IV is reused in counter mode, our entire one-time pad has been exposed previously[18]. However, if our IV is reused in CBC mode, the worst that could happen is something akin to ECB mode, and no messages are compromised.

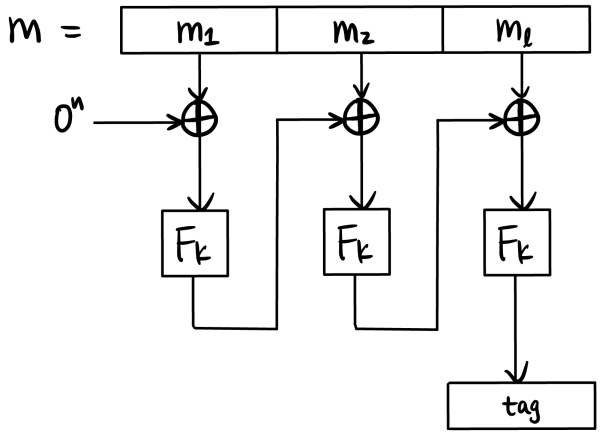
At the end of the day, *engineers are quite oblivious to cryptographic schemes*! Libraries only specify for *some key* and *some IV*, so it is exceedingly easy to screw up your cryptograhic scheme by reusing IVs, etc. CBC mode is simply more foolproof and incurs better outcomes in case it is used incorrectly[19].

[17] We nearly made mistakes in this course!
[18] XORing our ciphertexts will give $m \text{XOR} m'$.
[19] However, if Peihan were to implement a block cipher scheme herself, would opt for counter mode.

§7 February 12, 2025

§7.0.1 CBC-MAC

We can use block ciphers to construct a MAC scheme. Splitting up our message into blocks, we feed blocks into F_k and chain to next blocks. In the end, the final cipher output is our tag.



How do we verify? We can just Mac the message again and check that the tag matches. If F_k is invertible, we can also go the other way.

Is this CMA secure?

- Fixed-length messages of length $l \cdot n$? Yes, since we can only query for fixed-length messages, this gives us no additional information.

- Arbitrary-length messages? This is where problems arise—the adversary could first query for a message of 1 block, then 2 blocks, then 3 blocks, etc. By combining this information, they could produce new valid signatures.

  A concrete attack is an adversary querying for $\mathsf{Mac}(m)$ to produce tag, then querying for $\mathsf{Mac}(\mathsf{tag}) = \mathsf{Mac}(m||0) = \mathsf{tag}'$ which allows the adversary to forge a new message.

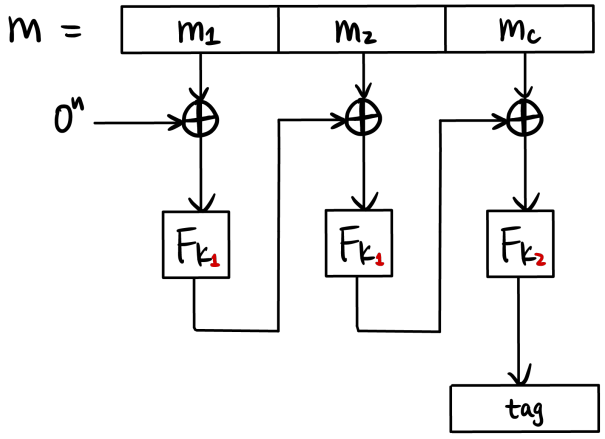
> **Remark.** Our constructions of authenticated encryption calls for an encryption scheme and MAC scheme. It's crucial that the two schemes have *different keys*. Using the same key k for both encryption and MAC can cuase issues (information from one could reveal something about the other).

We have a fix for the CMA-vulnerability in arbitrary-length messages:

§7.0.2 Encrypt-last-block CBC-MAC (ECBC-MAC)

The vulnerability earlier was due to our encryption being *associative*, so to speak.

We can fix this is to use a different key for the last block:



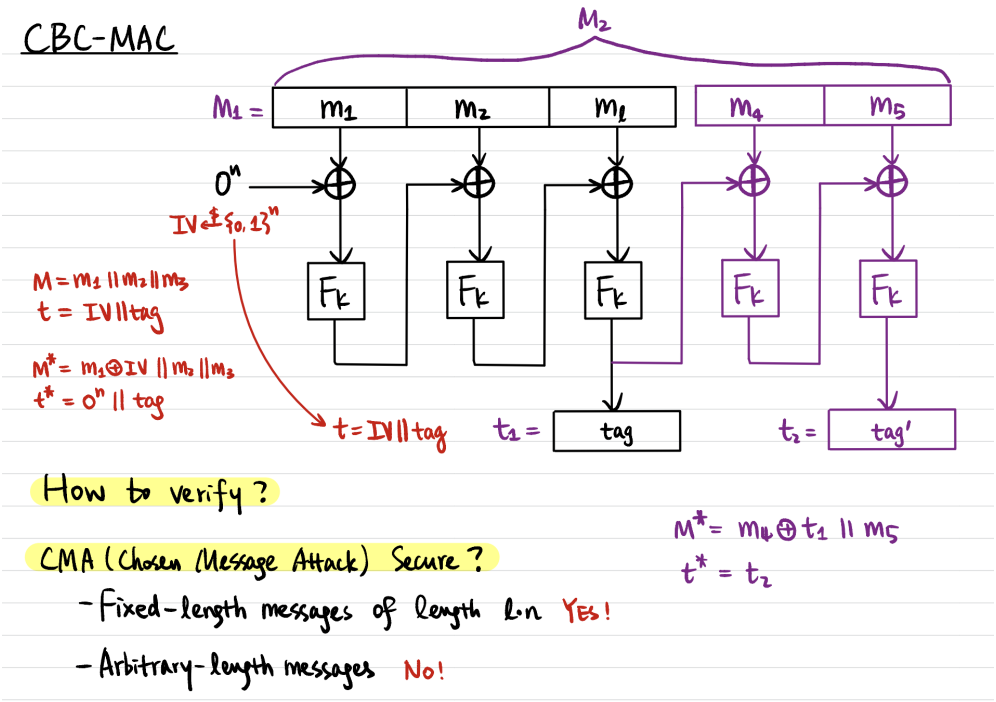
We could also attach length of messages to the first block, or other techniques.

The nuance in CBC-MAC means that realistically, we almost always use HMAC.

Question 7.1. For CBC-MAC, if we randomly sample the IV and include it in the tag, will this be CMA secure?

No! Consider if the adversary queries for the tag of $m := m_1||m_2||m_3$ and receive the tag $t := (\text{IV}, \text{tag})$.

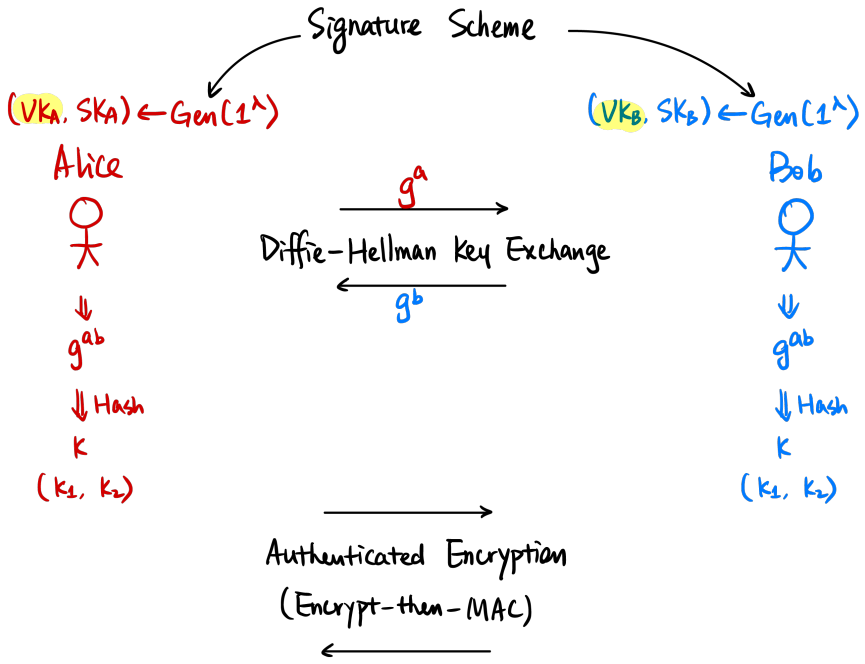
The adversary can generate a new valid tag for $m^* := m_1 \oplus \text{IV}||m_2||m_3$ and $t^* := (0^n, \text{tag})$.



§7.1 Putting it Together

Looking back at section 4.3, we've collected everything we need so far for secure communication.

For Alice and Bob to communicate, they first exchange keys using a Diffie-Hellman key exchange, then perform authenticated encryption.

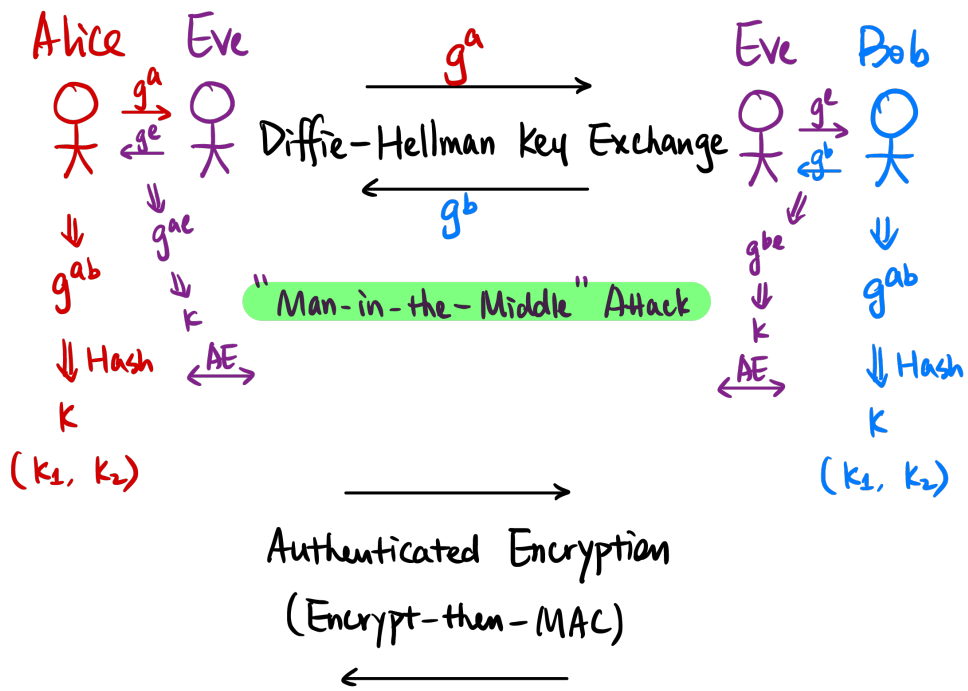


However, this still does not mitigate against a man-in-the-middle attack. Thus, before exchanging keys, Alice and Bob should publish verification keys (to a digital signature scheme, see section 4.1.2). Using this digital signature, Alice and Bob will each sign their Diffie-Hellman public values g^a, g^b using their signing key, which will be attached to the message. They can respectively verify that these values came from each other, and not some Eve in the middle.

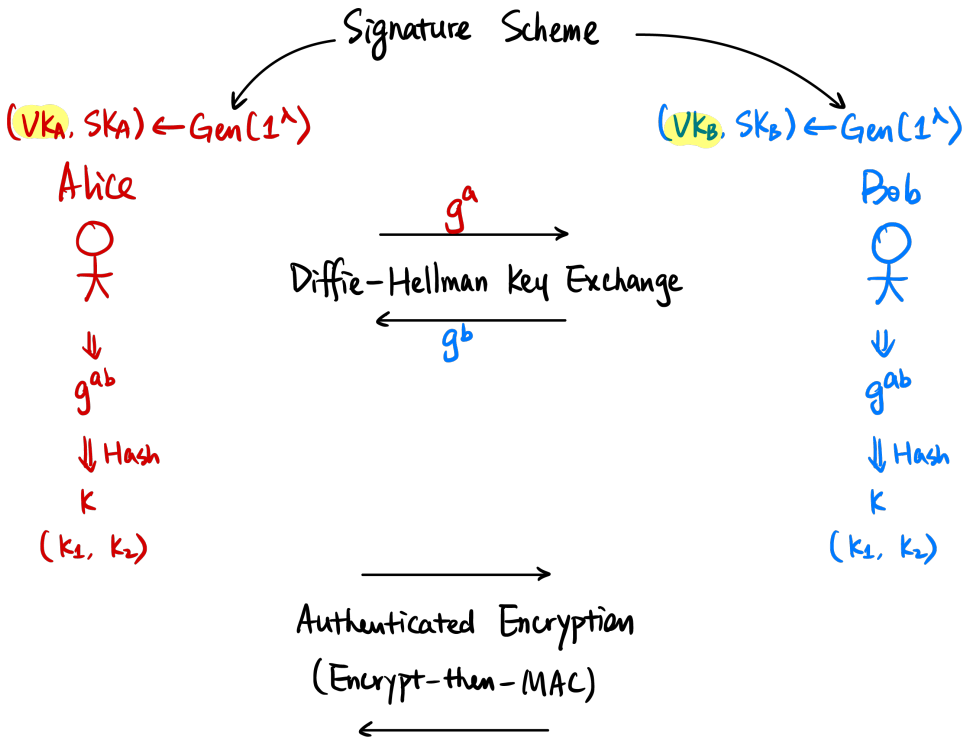
We will now go over the topics needed for the next project, Auth. Namely,

- One-Sided Secure Authentication
- Password-Based Authentication
- Two-Factor Authentication (2FA)
- Putting it All Together: Secure Authentication
- Public Key Infrastructure (PKI)

Recall that we had a way for Alice and Bob to communicate securely, first exchanging a shared Diffie-Hellman key and then performing AES encryption.



However, this is prone to a man-in-the-middle attack. One way to solve this is for parties to *sign* their own Diffie-Hellman public values before sending, and then verify the other party's public value by using their public verification key.

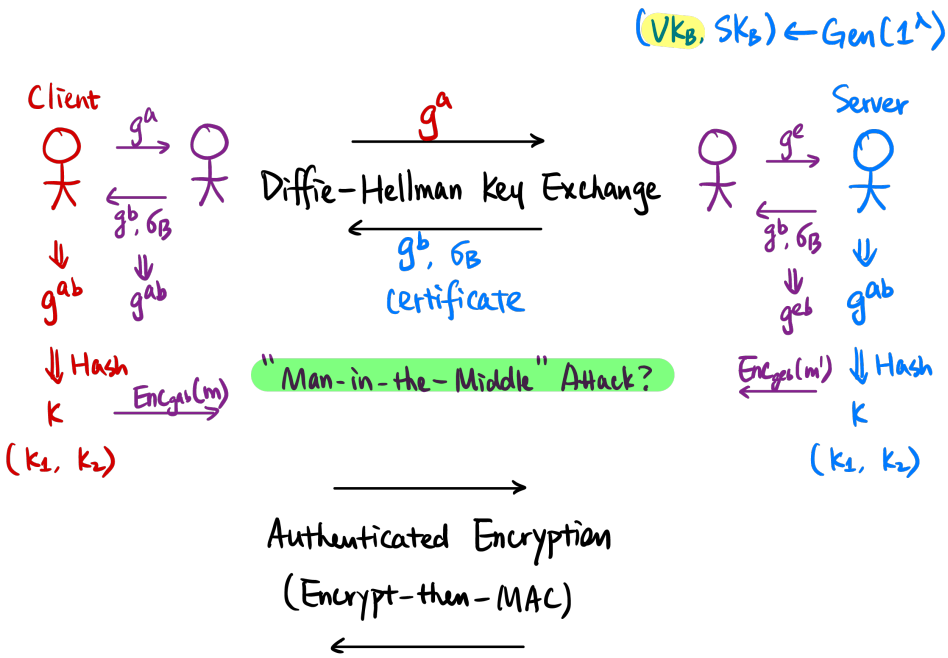


Is this now secure against an adversary in the middle? Yes, because the public values are guaranteed (via our digital signature scheme) by Alice and Bob's signing key. The man in the middle does not have access to the signing key, and cannot sign a phony public value.

However, how do we know the verification keys really belong to who we think they belong to? This is the problem of *authentication*. We are sort of in a chicken and egg problem...

§7.2 One-Sided Secure Authentication

In some circumstances, it's more difficult for a client to communicate their verification key to a server than it is for a server to do so. A server might publish their verification key, and trust that all clients are not compromised.



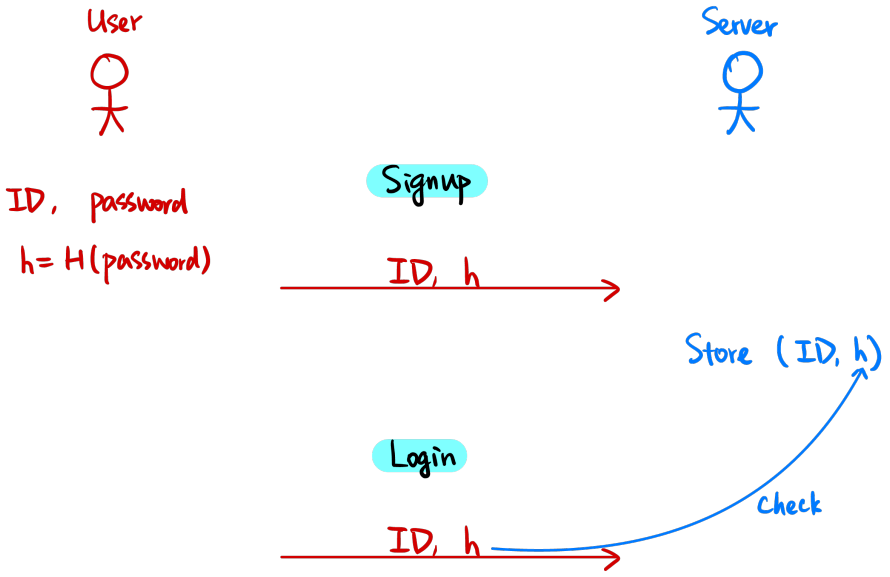
What could an adversary potentially do? The adversary could not pretend to be the server since they have no access to the server's signing key. The adversary *can* pretend to be the client and talk to the server. The adversary could forward all messages sent to the server, and can also communicate g^b, σ_b back to the client (it's a valid signature since it has not been modified).

At the end of this protocol, the client has Diffie-Hellman private g^{ab} and the adversary and server will have g^{eb} (where g^e, e is a Diffie-Hellman keypair the adversary provided to the client). Whatever the client sends to the server cannot be decrypted by the adversary, since it is encrypted with g^{ab}, however, the server's communications *could* be decrypted by the adversary.

This can be easily circumvented by requiring the server and user complete their handshake—the server could request a hash or encryption of the shared secret, and realize that they are communicating to an adversary when this cannot be forged by the man-in-the-middle.

§7.3 Password-Based Authentication

Sometimes, you also want to *authenticate* with a server using a password. The naïve implementation is that a user with an ID sends a hash of the password $h = H(\text{password})$ to the server. The server stores (ID, h).

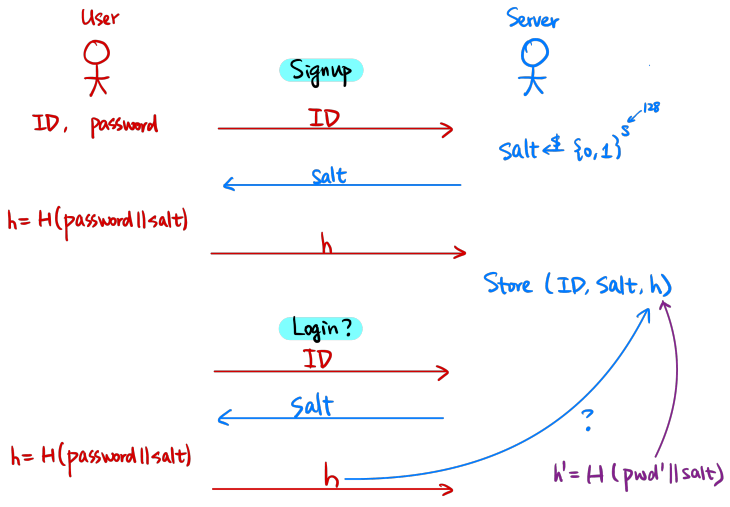


In this case, an adversary could launch an *Online Dictionary Attack* and try a lot of passwords with the server.

If the server were to be compromised, and its database compromised, the adversary can conduct an *Offline Dictionary Attack* on the database. Additionally, the adversary can precompute all hashes and check against the database.

How can we prevent this?

§7.3.1 Salting



One way of ensuring that the hashing is non-deterministic is for servers to generate a salt $\xleftarrow{\$} \{0,1\}^s$

and send it to the user. The user will hash $H(\text{password}||\text{salt})$ and send that to the server. The server stores a database of $(\text{ID}, \text{salt}, \text{h})$.

When logging in, the user first sends their ID to the server, the server will send the salt back, the user hashes their password, and the hash is sent to the server for verification.

Does this allow the user to use a weak password? Nope! The adversary can always brute-force the password.

To further solve these problems, we can use *peppering*. We send the salt to the user and the user computes hash $h = H(\text{password}||\text{salt})$. Then, we pick a random pepper and hash $h^* = H(h||\text{pepper})$ and stores h^*. Now, even if the server is compromised, there is no way to find the preimage of h^*, so adversaries knowing h^* will still have to do try all 2^p possible peppers for each dictionary guess. We still can't log into the server since the server hashes our login hash again.

Additionally, one strategy to make it *even harder* for an adversary is to make hashing more difficult (time-consuming). For example, we can compose SHA256 in certain ways[20]. There are also memory-hard hash functions, like scrypt.

Even with all this, is it still safe to use a weak password? Nope! A dictionary attack is still possible, and with weak passwords will be hard to crack.

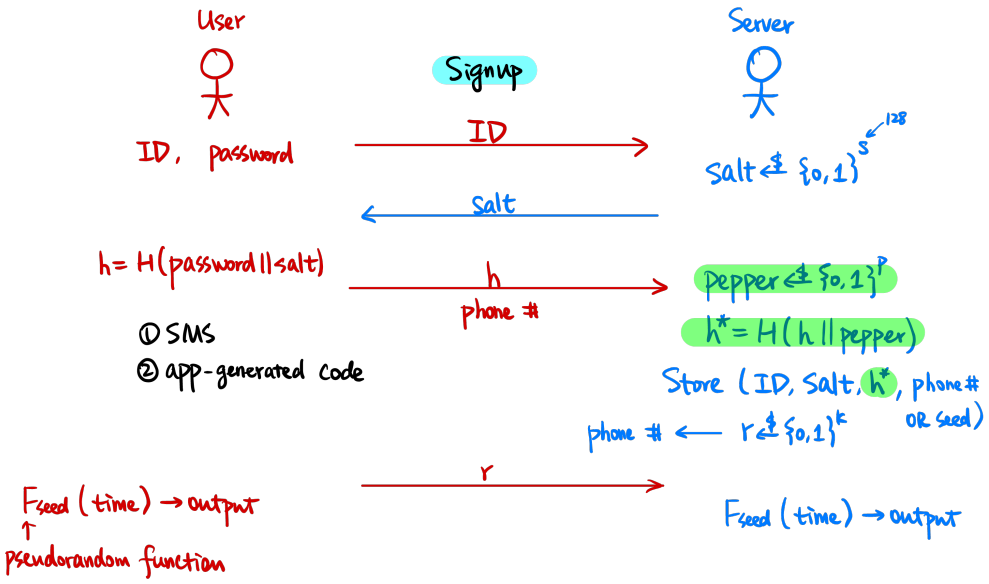
§7.3.2 Two-Factor Authentication

Now we'll discuss how servers implement two-factor authentication.

For phone number verification, on signing up, the user sends a phone number with their password hash. The server stores their phone number. Every time, the server will generate challenge $r \xleftarrow{\$} \{0,1\}^k$

For app-generated codes, the user and server will first share a seed seed and use a pseudorandom function $F_{\mathsf{seed}}(\mathsf{time})$. The server and the user can input the same time, and the outputs will be the same. Generally, the server will test the last 30/60 seconds of values.

[20] The natural way is to hash multiple times, say 100. However, this is actually not more secure in the case of SHA256 but there are specific ways of composition. For example, there are application-specific integrated circuits (ASIC) that can compute hash functions very efficiently.

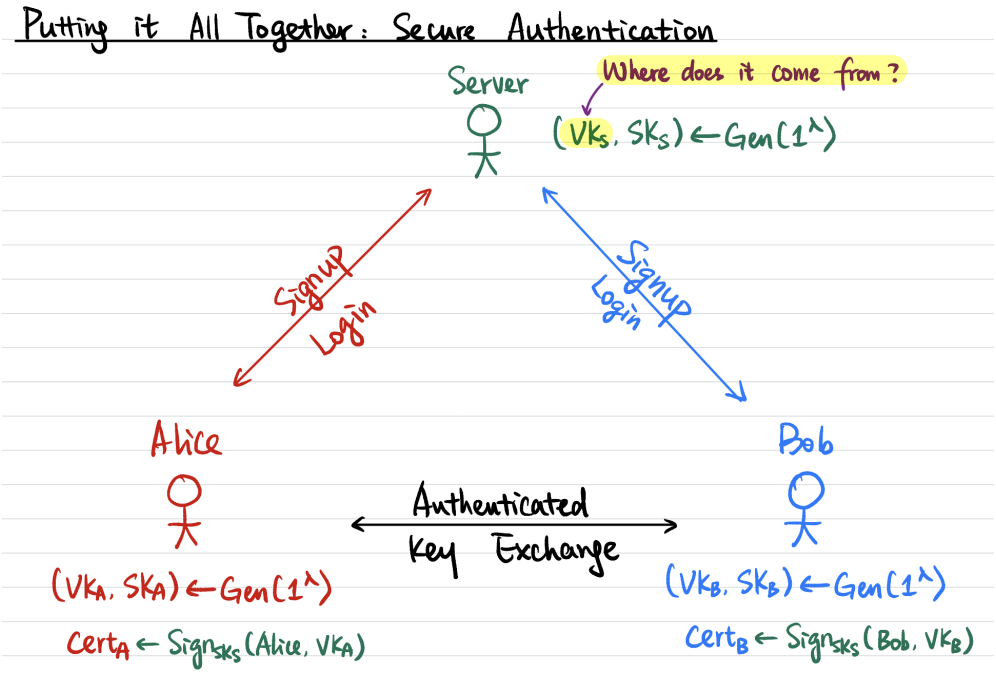

User
Server
Signup
ID
ID, password
Salt
salt ← $ {0,1}^s
128
h = H(password || salt)
h
phone #
pepper ← $ {0,1}^p
h* = H(h || pepper)
① SMS
② app-generated code
Store (ID, Salt, h*, phone # OR seed)
phone # ← r ← $ {0,1}^k
r
F_seed (time) → output
pseudorandom function
F_seed (time) → output

§8 February 19, 2025

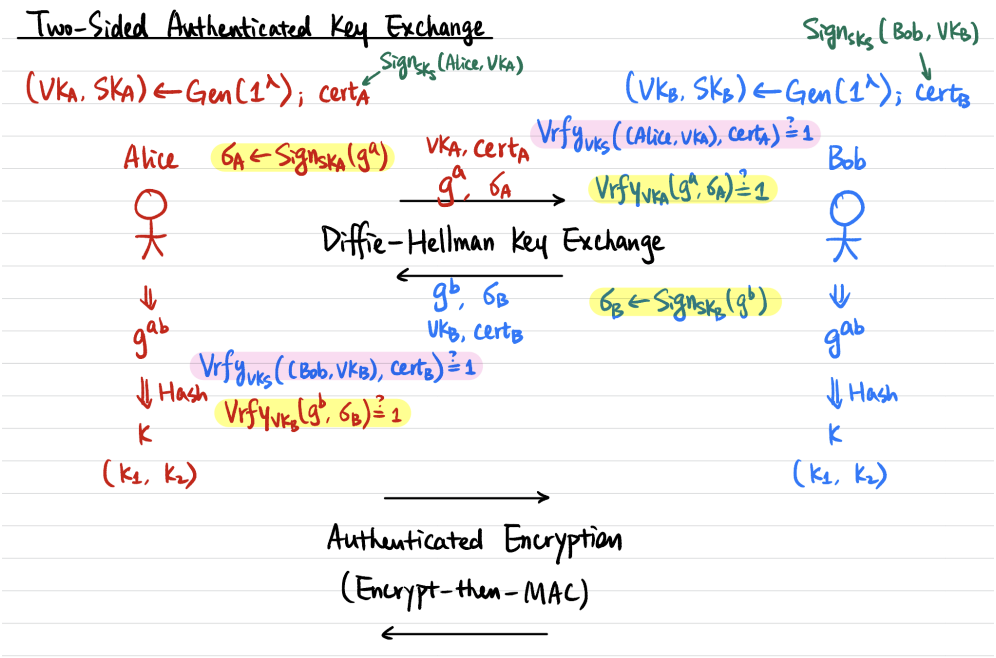
§8.1 A Brief Recap: Secure Authentication

We'll quickly review what we have covered so far.

Recall that we have introduced a third party, the server, who is trusted by all party's and assigns certificates.



After party's have their certificates, they can run two-sided authenticated key-exchange.

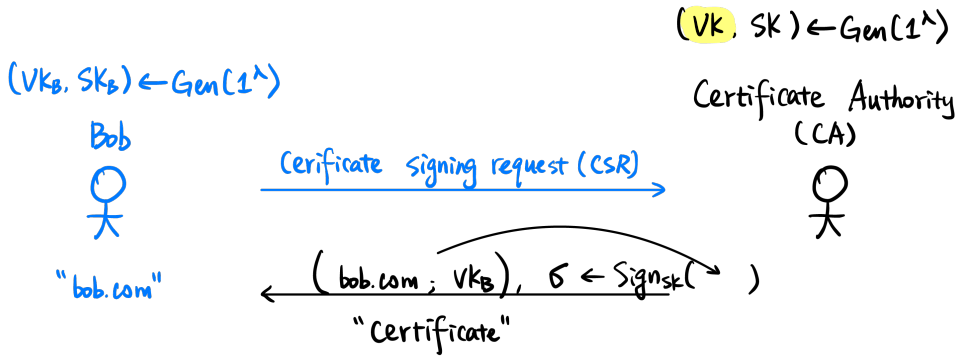


§8.2 Public Key Infrastructure

How can we know who has which public keys on the internet? We can rely on a Public Key Infrastructure (PKI) to know each other's public keys.

If Bob purports to be `bob.com` and wants to prove that vk_B belongs to him, Bob will send a certificate signing request (CSR) to a Certificate Authority (CA)[21].

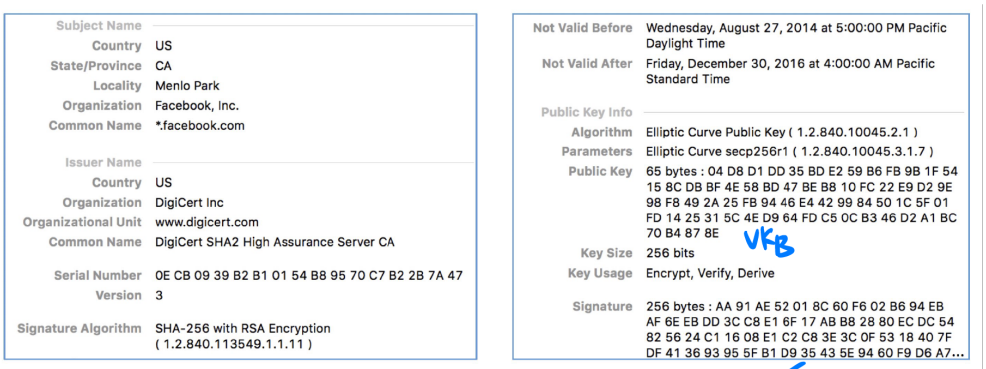
The CA will sign the message $(\texttt{bob.com}, vk_B)$ and send that signature σ back to Bob. This verifies that the user of `bob.com` holds signing key sk_B with public key vk_B.



[21] *The higher beings that be*...this is companies like *DigiCert*, *Let's Encrypt*, etc.

The standard of which is the X.509 certificate.

For example, when we try to access `facebook.com`, we can check that the certificate is valid[22]



An example X.509 certificate

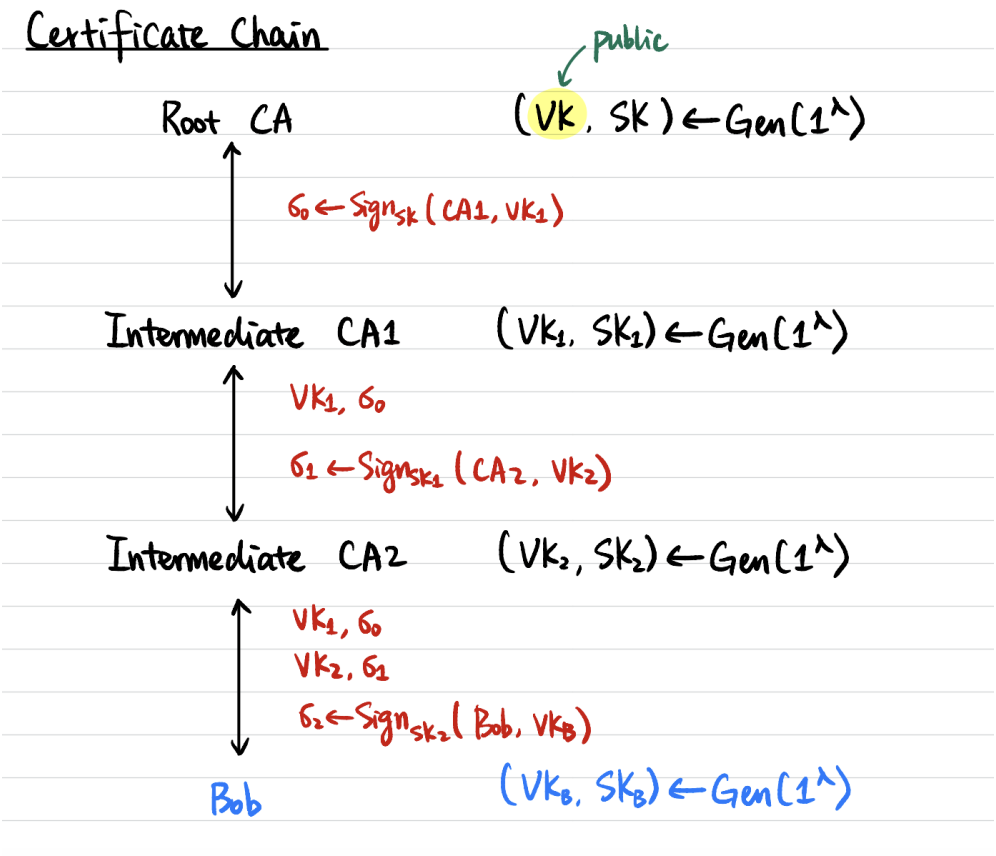
This pivots on the fact that *everyone* must know vk of the certificate authority. We shift our trust from individual sites and users to the certificate authorities. Most devices have the vks of trusted authorities built in.

What happens if a root CA gets compromised? An attacker can issue as many malicious certificates as they want - they could give certificates for Google, or certificates for Amazon, etc. This has actually happened before, and can be very hard to detect.

§8.2.1 Certificate Chain

In reality, there are several certificate authorities, and they also form *chains* of certificate authorities.

[22] In browsers, this is represented by the lock symbol—clicking on that will allow you to verify that certificate.



A Root CA[23] with a known (vk, sk) $\mathsf{Gen}(1^\lambda)$ can first sign the vk_1 of an Intermediate CA1, producing cert $\mathsf{cert}_1 = \sigma \leftarrow \mathsf{Sign}_{sk}(vk_1)$.

Then, the Intermediate CA1 can sign a certificate for Intermediate CA2, but we'll have to preserve this chain. Intermediate CA1 could produce cert $\sigma_1 \leftarrow \mathsf{Sign}_{sk_1}(vk_2)$, but how do we know that sk_1 is valid? So, we'll need to include vk_1 and vk_1's signature signed by sk. That is,

$$\begin{aligned}\mathsf{cert}_2 =& vk_1, \sigma \leftarrow \mathsf{Sign}_{sk}(vk_1),\\ & vk_2, \sigma_2 \leftarrow \mathsf{Sign}_{sk_1}(vk_2)\end{aligned}$$

Finally, Intermediate CA2 can sign Bob's verification key using their chain. Bob's certificate will contain

$$\begin{aligned}\mathsf{cert}_B =& vk_1, \sigma \leftarrow \mathsf{Sign}_{sk}(vk_1),\\ & vk_2, \sigma_2 \leftarrow \mathsf{Sign}_{sk_1}(vk_2)\\ & vk_B, \sigma_B \leftarrow \mathsf{Sign}_{sk_2}(vk_B)\end{aligned}$$

How can an Intermediate CA restrict Bob's use of these certificates? What if Bob will then go on and start signing his own certificates for people? We can concatenate information in each certificate

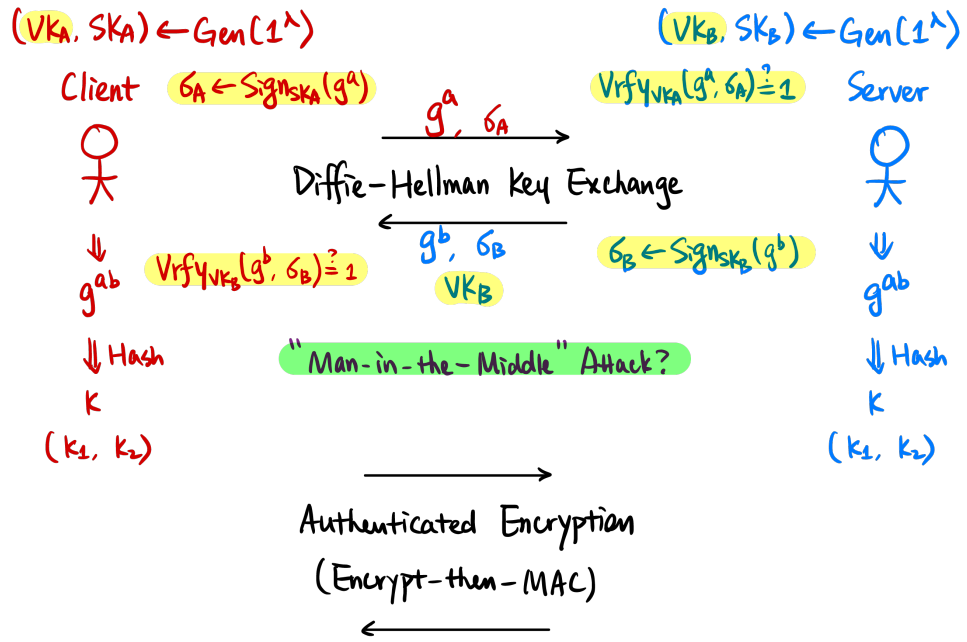
[23] We mentioned earlier that CAs are built into devices. For example, here is a list of all root certificates that are built-in for Apple devices. This can go wrong too! CAs have been misused which causes implications on the security of the internet.

that restricts its use. It could specify whether it is being issued to an *end user*, or even additional information like validity time.

To protect against CAs that get compromised, certificates are short-lived and have set validity times. Additionally, certificate authorities can publish revocation lists that browsers check against when validating a certificate.

§8.3 Case Studies

§8.3.1 SSH



Let's work through the steps of GitHub's SSH setup to see how it works.

The instructions are given for the EDDSA-25519 algorithm, which relies on elliptic curves.

1. We first generate a signing keypair $(vk_A, sk_A) \leftarrow \mathsf{Gen}(1^\lambda)$ via

   ```
   $ ssh-keygen -t ed25519 -C "your_email@example.com"
   ```

   vk_A is the `id_ed25519.pub` (the public key) sk_A is `id_ed25519` (the private key).

2. We upload our public key to our account on GitHub. This is equivalent of communicating our vk_A to GitHub.

3. When we're connecting via SSH to GitHub for the first time, our terminal will prompt us that this is a new server with a new verification key.

   ```
   > The authenticity of host 'github.com (IP ADDRESS)' can't be established.
   > RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
   > Are you sure you want to continue connecting (yes/no)?
   ```

   which we can verify against GitHub's known verification keys[24]. This is the equivalent of receiving a vk_B from GitHub.

§8.3.2 Secure Messaging

How can we design a secure messaging service where two people, Alice and Bob, can communicate across a server?

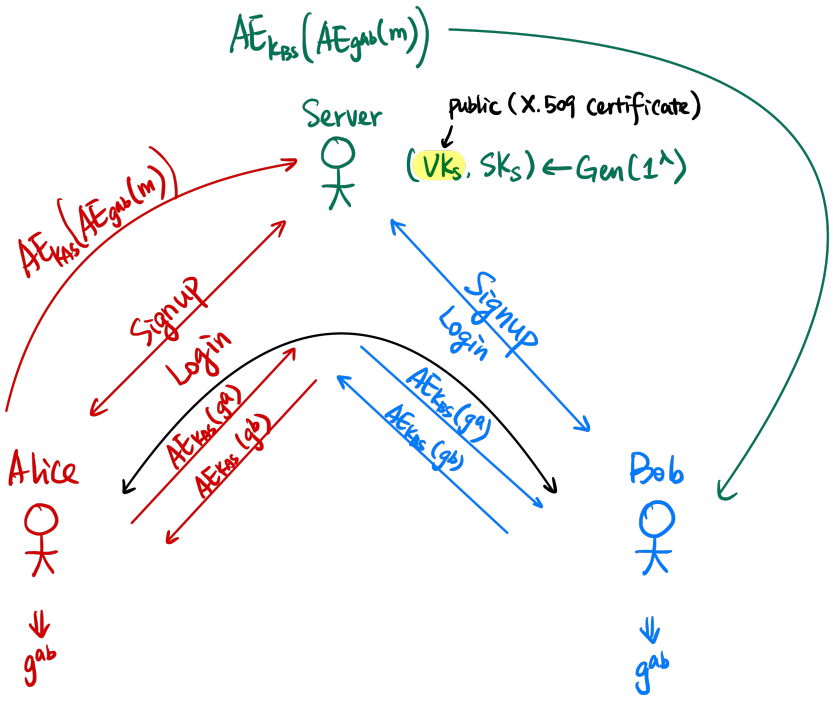
One solution is to have Alice sends an encrypted message, with a noted recipient (under Alice/Server's keys) to the server, the server decrypts it in the clear, and encrypts the message (using Bob/Server's keys) to send to Bob.

However, the message is completely revealed to the server in plaintext. Optimally, we don't want to do this, but many services do nevertheless. Alice and Bob can do a secure key exchange *through* the server to get shared g^{ab}, and encrypt messages between them.

Alice will first encrypt using their shared key, then using their shared secret with the server, encrypt that ciphertext. The server will decrypt the first layer, encrypt that with Bob's key, and send that to Bob.

We note that the server is still the perfect middleman, but our trust assumption is that the server is **semi-honest**—it will honestly follow the protocol but can try to glean any additional information from them.

[24] The security of our web upload to GitHub, or GitHub's site which publishes the verification key, relies on the security of the website, likely through TLS. But you could also imagine exchanging keys in person, etc.

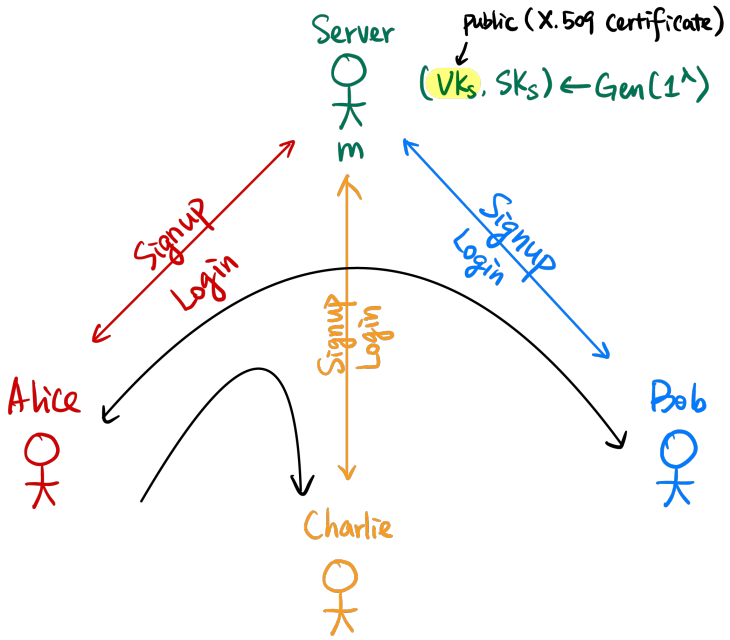


Why might we still adopt the first approach, sending messages in plaintext? Alice and Bob needs to know their private keys, and remain 'online' all the time. If they switch a device, or lose their phone, messages will get lost. Sending messages in plaintext avoids this scenario.

§8.3.3 Group Chats

What about group chats? How might we implement this.

When we move to group chats, there are more things we need to consider. For example, do we want to reveal this message to the server? In this case, Alice can send the message in the clear to the server and it is forwarded. Additionally, we might ask whether we want to hide the group structure from the server.



The first scenario is the same—users can send the encrypted message to the server, the server reveals the message and reencrypts to the group members.

We might posit that Alice, Bob and Charlie share keys g^a, g^b, g^c, then they jointly have shared secret g^{abc}. This is called multi-party key exchange. However, this is in fact very difficult and relies on strong primitives.

Signal and WhatsApp use two different approaches (agree on the same key or pairwise keys), but they both have tradeoffs. We'll continue this next lecture.

In general, there are two paradigms for group messaging. Either everyone uses the *same* key, or everyone has a different key. In WhatsApp, Alice would use a symmetric ratchet with key A, gr (Alice's key and group key) to send the message to the server, and WhatsApp will forward the same encrypted message to Bob and Charlie. While the group structure is revealed to the server, but the message contents are unbeknownst to the server.

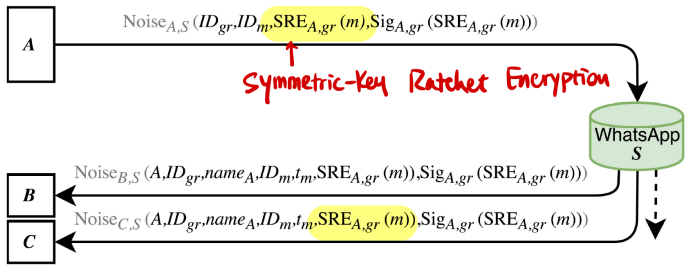


Figure 5. Schematic depiction of traffic, generated for a message m from sender A to receivers B, C in group gr with $\mathcal{G}_{gr} = \{A, B, C\}$ in WhatsApp.

In Signal, on the other hand, every pair of users has a differnt key. If Alice wants to send a message to Bob and Charlie, Alice will encrypt two messages, one with Alice/Bob's key and another with Alice/Charlie's key. The server will forward the encrypted messages to the users respectively. In Signal, a double ratchet encryption is preformed between every pair of parties. Another guarantee is that the group structure can be hidden against the server—Alice sending individual messages to Bob and Charlie is indistinguishable from their group texts.

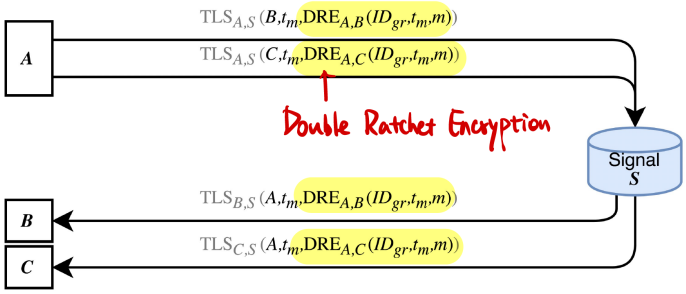


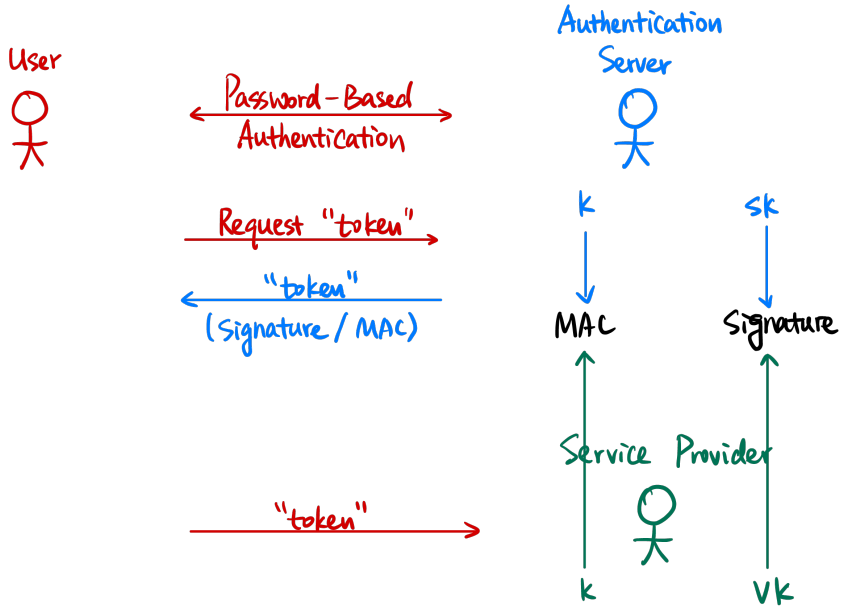
Figure 3. Schematic depiction of Signal's traffic, generated for a message m from sender A to receivers B and C in group gr with $\mathcal{G}_{gr} = \{A, B, C\}$. Transport layer protection is not in the analysis scope (gray).

§9 February 24, 2025

§9.1 Single Sign-On (SSO) Authentication

Often, we'll 'log in with Google' or 'log in with Apple'[25]. A user will authenticate themselves with the authentication server (Google, Apple, Shibboleth), and will be issued a 'token' (usually a signature/MAC) for them to then authenticate themselves against the service provider.

Implementations include OAuth or OpenID, which is the format used by Google/Apple/Facebook, etc. Within enterprises, Kerberos credentials allow for SSO as well as things such as printing, connecting to servers, etc.



§9.2 Zero-Knowledge Proofs

As mentioned in our course outline, a Zero-Knowledge Proof (ZKP) is a scheme that allows a prover to prove to a verifier some knowledge that they have, without revealing that knowledge.

What is a proof? We consider what a 'proof system' is. For example, we'll have a *statement* and a *proof* that is a purported proof of that statement. What guarantees do we want from this proof system? If the statement is true, we should be able to prove it; and if the statement is false, we shouldn't be able to prove this. These are our guarantees of *completeness* and *soundness*.

Completeness. If a statement is true, there exists a proof that proves it is true.

[25] Even Brown has Shibboleth!

Soundness. If a statement is false, any proof cannot prove it is true.

§9.3 Zero-Knowledge Proofs

A Zero-Knowledge Proof (ZKP) is a scheme that allows a prover to prove to a verifier some knowledge that they have, without revealing that knowledge.

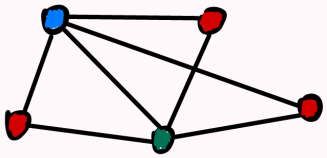
What is a proof? We consider what a 'proof system' is. For example, we'll have a *statement* and a *proof* that is a purported proof of that statement. What guarantees do we want from this proof system? If the statement is true, we should be able to prove it; and if the statement is false, we shouldn't be able to prove this. These are our guarantees of *completeness* and *soundness*.

Completeness. If a statement is true, there exists a proof that proves it is true.

Soundness. If a statement is false, any proof cannot prove it is true.

We can think of NP languages from a proof system perspective.

Example 9.1 (Graph 3-Coloring)
Consider the *Graph 3-coloring*.



We define our language

$$L = \{G : G \text{ has a 3-coloring}\}$$

and relation

$$R_L = \{(G, 3\text{Col})\}$$

Our statement will be that G has a 3-coloring. Our proof is providing such a coloring $(G, 3\text{Col}) \in R_L$.

This satisfies completeness and soundness. Every 3-colorable graph has a proof that is the 3-coloring itself, and if a graph doesn't have a 3-coloring, it will not have a proof.

We can think of NP languages as a proof system. A language L is in NP if $\exists$poly-time$V$ (verifier) such that

Completeness. $\forall x \in L, \exists w$ (witness) such that $V(x, w) = 1$.

Soundness. $\forall x \notin L, \forall w^*, V(x, w^*) = 0$.

The prover will prove to the verifier that they have knowledge of witness w without revealing the witness itself.

Definition 9.2 (Zero-Knowledge Proof System)

Let (P, V) (for *prover* and *verifier*) be a pair of probabilistic poly-time (PPT) interactive machines. (P, V) is a zero-knowledge proof system for a language L with associated relation R_L if

Completeness. $\forall (x, w) \in R_L$, $\Pr[P(x, w) \leftrightarrow V(x) \text{ outputs } 1] = 1$. That is, if there is a $x \in L$ with witness w, a prover will be able to prove to the verifier that they have knowledge of w.

Soundness. $\forall x \notin L$, $\forall P^*$, $\Pr[P^*(x) \leftrightarrow V(x) \text{ outputs } 1] \simeq 0$. That is, for every x not in the language, our prover P^* will not be able to prove its validity to V, with negligible probability. If P^* is PPT, we call the system a *zero-knowledge argument*.

§9.3.1 Proof of Knowledge

Intuitively, if the prover is able to prove, then they must know a witness w. There exists some extractor such that, if you can prove it true, there is some extractor that can extract the witness from it.

$$\exists E \text{ s.t. } \forall p^*, \forall x, \Pr[E^{P^*(\cdot)}(x) \text{ outputs } w \text{ s.t. } (x, w) \in R_L] \cong \Pr[p^* \leftrightarrow V(x) \text{ outputs } 1]$$

§9.3.2 Honest-Verifier Zero-Knowledge

Honest-Verifier Zero-Knowledge (HVZK) can be thought of as security against a semi-honest verifier. In this scenario, the verifier will follows the protocol honestly.

$$\exists \text{PPT } S \text{ s.t. } \forall (x, w) \in R_L, \text{View}_V(P(x, w) \leftrightarrow V(x)) \simeq S(x)$$

That is, that the transcript can be simulated by a simulator S without interaction with the verifier, and without w.

§9.3.3 Zero-Knowledge (Malicious Verifier)

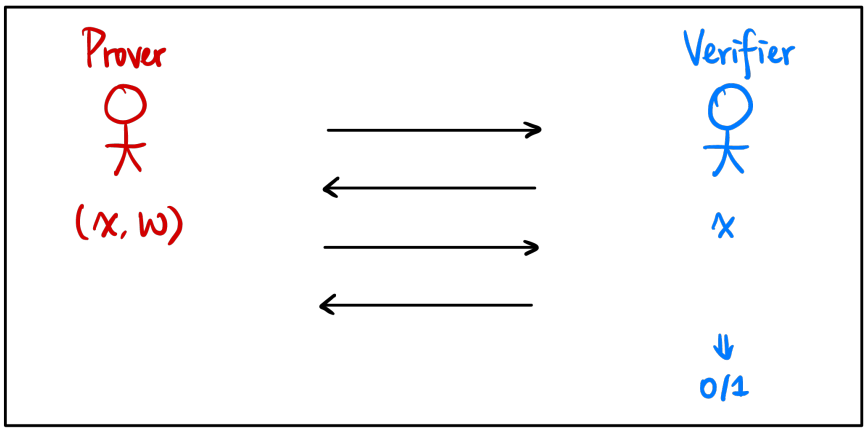
Now assume that the verifier is malicious. In other words, the verifier can deviate from the protocol in attempt to extract more information than intended. Note that a malicious verifier V^* is unable to learn anything about w.

We need an additional property that this is actually *zero-knowledge*[26]. We want to say that the verifier is unable to extract any additional information from the interaction between the verifier and prover. That is, even without the witness, a verifier might be able to 'simulate' this transaction *by themselves*!

We'll say $\forall$PPT $V^*, \exists$PPT S such that $\forall (x, w) \in R_L$,

$$\text{Output}_{V^*}[P(x, w) \leftrightarrow V^*(x)] \simeq S(x).$$

That is to say, for everything in the language, the output transcript between the prover and verifier can be *simulated* by the simulator without knowledge of the witness[27].



§9.3.4 Zero-Knowledge Proof of Knowledge

So we've built up our four properties:

- Completeness: The prover can prove whenever $x \in R_L$.
- Soundness: For any x not in R_L, the prover can only prove $x \in R_L$ with *negligible* probability.
- Zero Knowledge: The verifier does not gain any additional information from the proof. That is, a simulator could have 'thought up' the entire transcript in their head given the ability to rewind.
- Proof of Knowledge: An even stronger guarantee than soundness (this implies soundness)—a prover must have the witness in hand to be able to prove $x \in R_L$. That is, an extractor could

interact with the prover (and rewind) to be able to extract the information of w from the interaction.

§9.4 Example: Schnorr's Identification Protocol

Let G be a cyclic group of prime order q with generator g, $h = g^a$. We wish to prove the relation

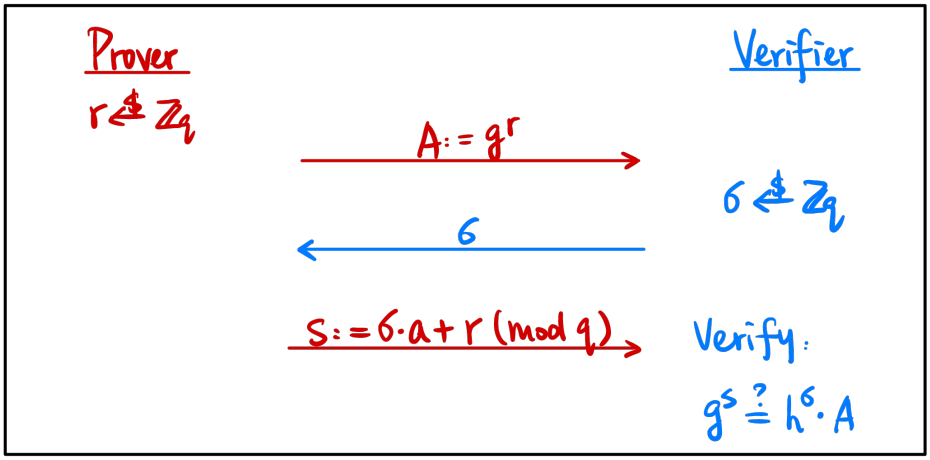
$$R_L = \{(g^\alpha, \alpha)\}_{\alpha \in \mathbb{Z}_q}$$

The language L is given by

$$L = \{h \in G : \exists a \in \mathbb{Z}_q \text{ s.t. } h = g^a\}$$

However, this is exactly the entire group, i.e. $L = G$.

Generator g is known, and the prover wishes to prove that they have the discrete log of h (α where $g^\alpha \equiv h$).



Completeness here is clear, the prover is able to produce such s if the prover has knowledge of α.

§9.4.1 Proof of Knowledge

We wish that

$$\exists \text{PPT } E \text{ s.t. } \forall \text{PPT } P^*, \forall x \in L, E^{P^*(\cdot)}(x) \text{ outputs } w \text{ s.t. } (x, w) \in R_L$$

"That there exists an extractor that by interacting to the prover P^* that can extract w/α"

In this case, E can rewind the prover as well. We do so as follows:

We get the prover to commit to some r and $A := g^r$, and pick 2 σ's (rewinding) such that we have

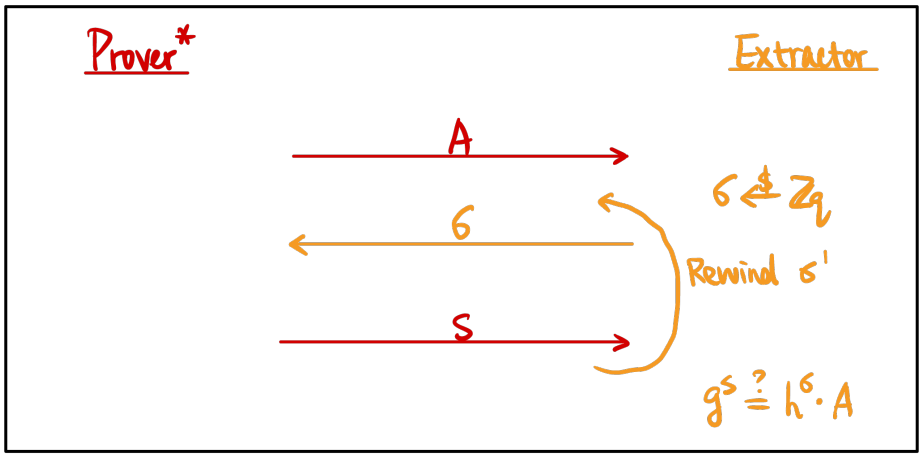
$$g^s = h^\sigma \cdot u$$
$$g^{s'} = h^{\sigma'} \cdot u$$

Given these two equations, we can

$$g^{s-s'} = h^{\sigma-\sigma'}$$

Then we have

$$g^{(s-s')(\sigma-\sigma')^{-1}} = h \implies \alpha = (s-s')(\sigma-\sigma')^{-1}$$



§9.4.2 Honest-Verifier Zero-Knowledge

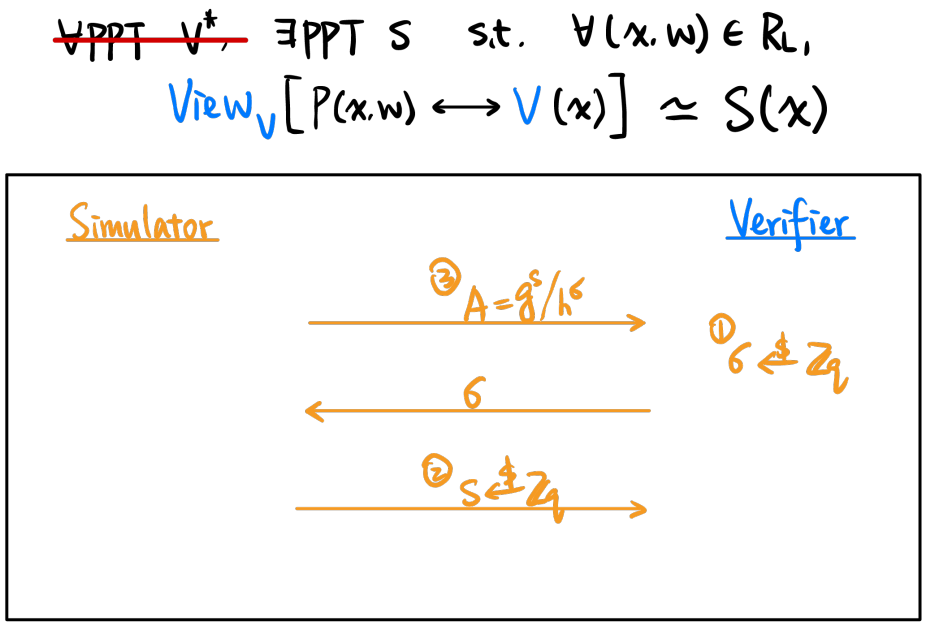
Can we also construct Zero-Knowledge for this protocol?

$$\forall \text{PPT } V^*, \exists \text{PPT } S \text{ s.t. } \forall (x,w) \in R_L, \text{Output}_{V^*}(P(x,w) \leftrightarrow V^*(x)) \overset{C}{\simeq} S(x)$$

We first do this for Honest-Verifier Zero-Knowledge. This can be thought of as security against a semi-honest verifier.

$$\exists \text{PPT } S \text{ s.t. } \forall (x,w) \in R_L, \text{View}_V(P(x,w) \leftrightarrow V(x)) \simeq S(x)$$

That is, that the transcript can be simulated by a simulator S without interaction with the verifier, and without w.



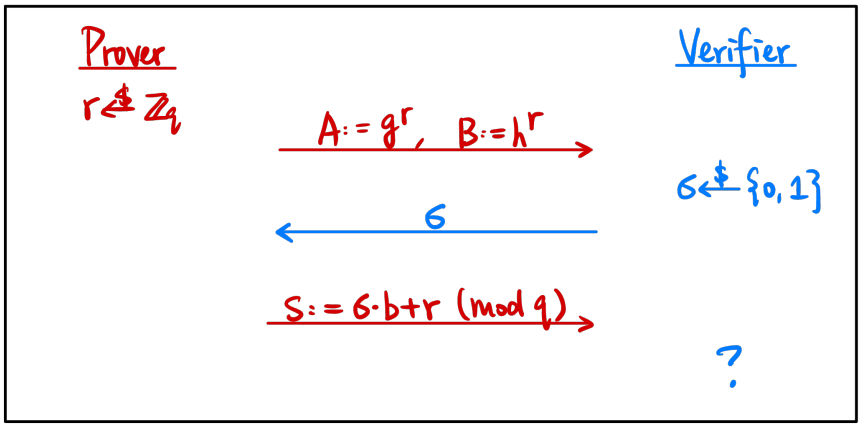
We construct a simulator that gives us specific values of A, σ, s where we can satisfy the equation $g^s = h^\sigma \cdot A$. Once we have s and σ, we can easily compute the A desired.

We don't have to generate them in order. Fixing s and sampling $\sigma \xleftarrow{\$} \mathbb{Z}_g$, we can compute $g^s \cdot h^{-\sigma} = A$.

§9.5 Example: Diffie-Hellman Tuple

We want to prove that $h = g^a, u = g^b, v = g^{ab}$ is a Diffie-Hellman Tuple in a cyclic group $\mathbb{G}$ of order q and generator g.

Our witness is 'private exponent' b. Our statement is that $\exists b \in \mathbb{Z}_q$ such that $u = g^b$ and $v = h^b$.



The prover will randomly sample $r \xleftarrow{\$} \mathbb{Z}_q$ and send to the verifier $A := g^r$ and $B := h^r$. The verifier randomly samples *challenge* $\sigma \xleftarrow{\$} \{0,1\}$, and sends this challenge bit to the prover. The prover will respond with $s := \sigma \cdot b + r \pmod{q}$. If the challenge bit was 0, $s = r$ and the verifier verifies $A = g^S$ and $B = h^S$. If the challenge bit was 1, $s = b + r$ and the verifier verifies $u \cdot A = g^s$ and $v \cdot B = h^S$.

Completeness: If this statement is true, the prover will be able to convince the verifier since they have knowledge of b.

Next lecture, we will prove proof of knowledge and honest-verifier zero-knowledge.

§10 February 26, 2025

Today, we continue our discussion on zero-knowledge proofs. First, we will briefly give an overview on our next project involving anonymous online voting. We will revisit the example on the Diffie-Hellman Tuple, then discuss Non-Interactive Zero-Knowledge proofs. We will show how to achieve this using Fiat-Shamir Heuristic, talk about Elgalmal encryption for homomorphism and threshold decryption, and revisit how these relate to anonymous online voting.

§10.1 Anonymous Online Voting

Say we have n voters with votes $v_1, \ldots, v_n \in \{0, 1\}$. Each voter encrypts their vote $\text{Enc}(v_1), \ldots, \text{Enc}(v_n)$. Our goal is to compute the sum of these votes without having to decrypt each vote individually. Somehow, we must find $\text{Enc}(\sum v_i)$ then decrypt to find $\sum v_i$.

In this scenario, zero-knowledge proofs can be used to ensure that each vote v_1 is 0 or 1 and to verify that the sum $\sum v_i$ was computed correctly.

§10.2 Zero-Knowledge Proof of Knowledge

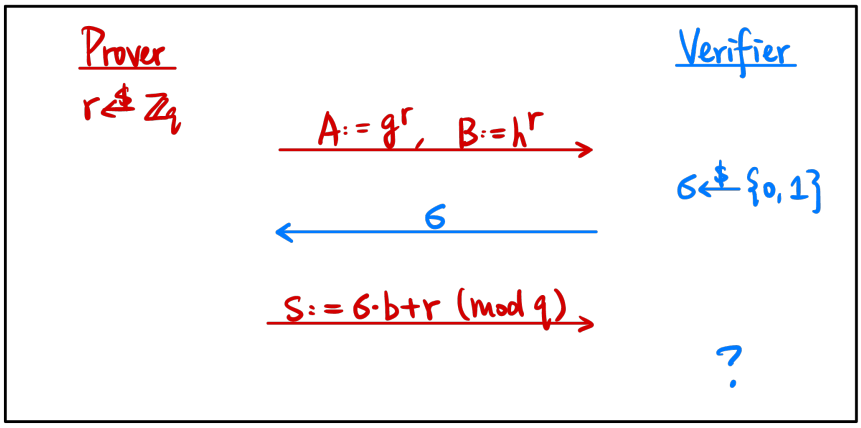
Recall the five properties.

- Completeness: The prover can prove whenever $x \in R_L$.
- Soundness: For any x not in R_L, the prover can only prove $x \in R_L$ with *negligible* probability.
- Proof of Knowledge: If a prover P^* can prove, then they must know w.
- Honest-Verifier Zero-Knowledge (HVZK). An honest verifier doesn't learn anything about w.
- Zero Knowledge: A malicious V^* doesn't learn anything about w.

§10.3 Example: Diffie Hellman Tuple

We want to prove that $h = g^a, u = g^b, v = g^{ab}$ is a Diffie-Hellman Tuple in a cyclic group $\mathbb{G}$ of order q and generator g.

Our witness is 'private exponent' b. Our statement is that $\exists b \in \mathbb{Z}_q$ such that $u = g^b$ and $v = h^b$.



The prover will randomly sample $r \xleftarrow{\$} \mathbb{Z}_q$ and send to the verifier $A := g^r$ and $B := h^r$. The verifier randomly samples *challenge* $\sigma \xleftarrow{\$} \{0,1\}$, and sends this challenge bit to the prover. The prover will respond with $s := \sigma \cdot b + r \pmod q$. If the challenge bit was 0, $s = r$ and the verifier verifies $A = g^S$ and $B = h^S$. If the challenge bit was 1, $s = b + r$ and the verifier verifies $u \cdot A = g^s$ and $v \cdot B = h^S$.

Completeness: If this statement is true, the prover will be able to convince the verifier since they have knowledge of b.

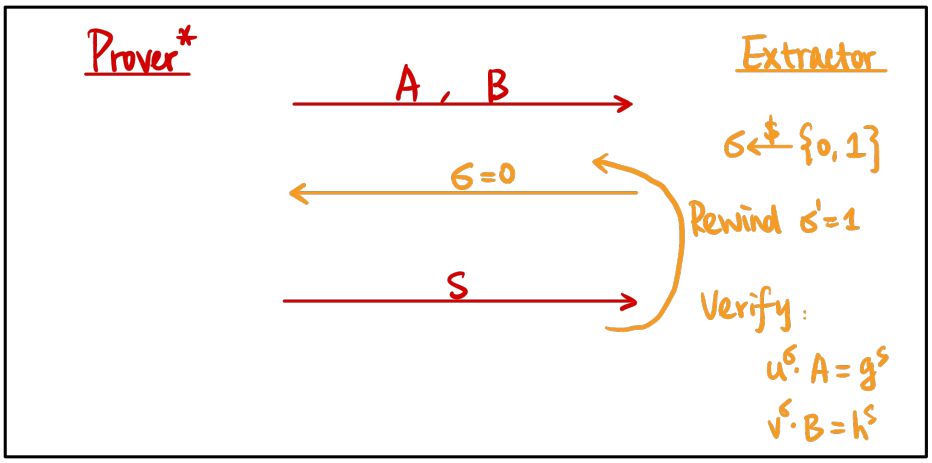
Soundness: Show that if the prover does not have b, the probability that they send a valid $S = \sigma b + r \mod q$ is negligible. The prover can be malicious and does not have to follow the protocol. In the first round, they do not have to send $A = g^r, B = h^r$, but they can send $A = g^{r_1}, B = h^{r_2}$.

Proof of Knowledge: Formally, $\exists$PPT E (called *extractor*) such that $\forall P^*$ (potentially dishonest prover), $\forall x$,

$$\Pr[E^{P^*(\cdot)}(x) \text{ outputs } w \text{ s.t. } (x,w) \in R_L] \simeq \Pr[P^* \leftrightarrow V(x) \text{ outputs } 1].$$

This is to say, the probability that the extractor can extract a witness is computationally indistinguishable from the probability of the prover successfully proving $x \in R_L$.

An extractor, interacting with a prover (not necessarily honest), should be able to *extract* the witness w out of its communication with the prover, with the additional power that it can rewind the prover.



The extractor can first pick $\sigma = 0$, which gives them s such that $A = g^s, B = h^s$. Then, the extractor rewinds the protocol and issues challenge $\sigma' = 1$, gaining s' such that $u \cdot A = g^{s'}$ and $v \cdot B = h^{s'}$.

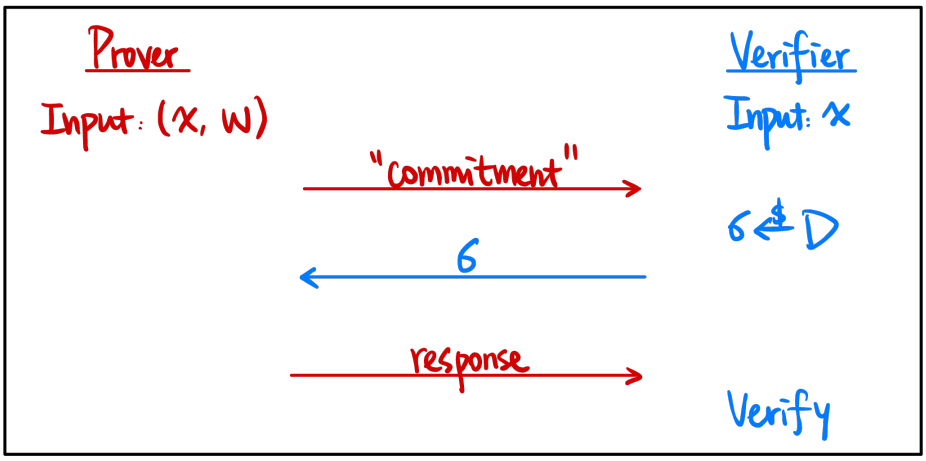
Then, $u = g^{s-s'}$ and $v = h^{s-s'}$, combining these they can extract valid $b = s - s' \pmod q$. If the prover can always convince the verifier, then the extractor will always be able to extract the witness w.

Honest-Verifier Zero-Knowledge

$$\exists \text{PPT } S \text{ s.t. } \forall (x, w) \in R_L, \text{View}_V(P(x, w) \leftrightarrow V(x)) \simeq S(x)$$

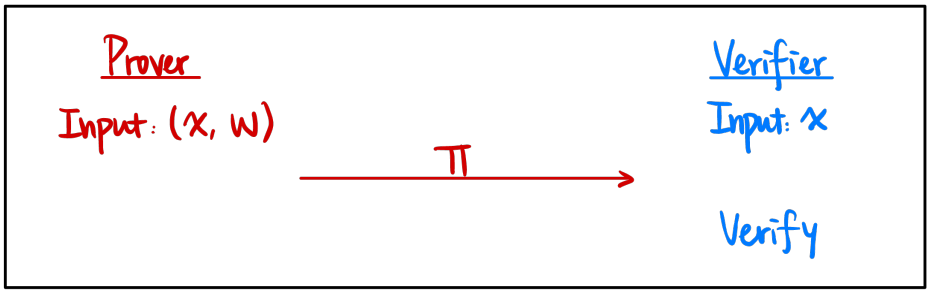
We want to ensure that the verifier does not know anything about the witness.

§10.3.1 Non-Interactive Zero-Knowledge (NIZK) Proofs



These all fall under a class called Sigma Protocols (they look like a capital Σ). A prover will first commit to some A, the verifier issues a challenge σ, then the prover will provide a proof corresponding to their A and σ.

What if we wanted to condense it into a protocol that was *non-interactive*, and only relies on the prover sending *one* round of 'proof' to the verifier.



Completeness and soundness are the same. What about our previous definition of zero-knowledge? For all verifiers, will the simulator be able to simulate the one-way transcript? This is equivalent to the simulator being able to prove the statement itself[28]. Since there is only one round, the simulator 'loses' its ability to rewind the prover and verifier. In the plain model, we cannot achieve a NIZK proof.

To make NIZK proofs possible, there are a few models available to us.

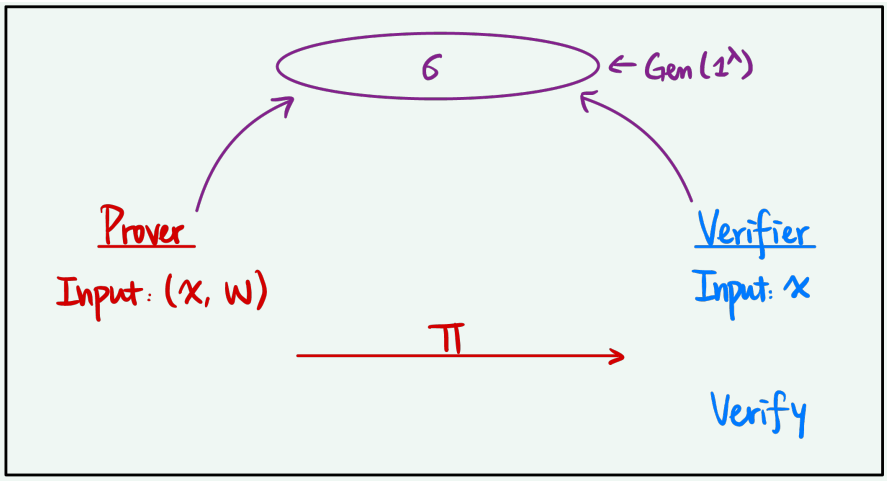
Common Random String/Common Reference String (CRS): There is a trusted third-party that both parties have access to, who generates a shared reference string.

The power that we give to the simulator is that the simulator is allowed to generate this random/reference string together with the proof. This should be indistinguishable against the real-world.

In reality, the CRS can be generated in a *key ceremony* between parties such that no party can interfere with the generated key.

[28] This is generally impossible! For example, a NIZK for the DH tuple that satisfies Zero-Knowledge breaks the Decisional Diffie-Hellman assumption.

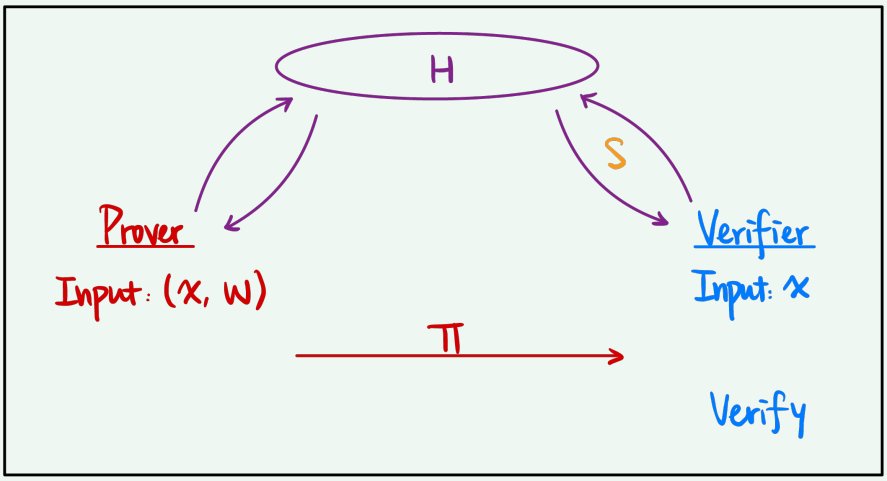
For the sake of contradiction, say we had such a simulator. To distinguish whether $(g^a, g^b, g^c) \overset{c}{\simeq} (g^a, g^b, g^{ab})$, we can feed this to the simulator to get a proof, and check with the verifier whether the proof is valid. The proof is valid if and only if it is a valid tuple. This contradicts DDH. Such a simulator had better not exist.



There are some formal proof definitions of Zero-Knowledge, etc elided here but are in the post-notes.

Random Oracle (RO) Model: The prover and verifier have access to a hash function and it is a random oracle (behaves as if it is a random function).

The additional power we give to the simulator is that they can control the behavior of the random oracle.



[27] That is, the prover could just send the witness in the clear to the verifier, which satisfies completeness and soundness.

[28] This is *counterintuitive*, because if any PPT can simulate the proof by themselves, how do we know we're even talking to a prover that has a witness? This is subtle, but we give extra power to the simulator that they are allowed to *rewind* the verifier to some previous step. If the transcript can be simulated, then surely no information is leaked from the protocol.

§10.3.2 Fiat-Shamir Heuristic

We can convert any Sigma protocol into a NIZK under the random oracle model. Recall that the only thing that could have gone wrong is that $\sigma \xleftarrow{\$} D$ was not computed randomly. Instead of using a challenge from the verifier, the challenge becomes $\sigma : H(x||m_1)$, a hash of the transcript so far. Since both the prover and verifier have access to the hash function, the prover can generate a challenge for themselves. After that, the prover can generate response.

A malicious verifier has no control over σ now, so a malicious verifier cannot do anything more than seeing the proof. A malicious prover also cannot produce a valid σ without committing to a m_1 first before receiving σ.

We can transform any public-coin[29] HVZK of arbitrary number of rounds into a NIZK using the Fiat-Shamir heuristic in the random oracle model.

Every public coin challenge will become the hash of the transcript so far from a random oracle. This condenses the entire proof into a single message that can be sent to the verifier (and that a verifier at a later point in time can also verify).

Example 10.1

The Fiat-Shamir Heuristic can also transform Schnorr's Identification Protocol into Schnorr's Signature Scheme in the RO model, such that it becomes a NIZK proof.

We have a cyclic group $\mathbb{G}$ of order g, and generator g.

Public verification key $vk = g^a$, secret signing key $sk = a$.

We condense the Schnorr's Identification Protocol into a NIZK proof. To sign a message m:

1. prover samples $r \xleftarrow{\$} \mathbb{Z}_q$ and computes $A := g^r$
2. prover computes $\sigma := H(A||M)$
3. prover computes $s := \sigma * a + r \mod q$
4. prover sends (A, s) to verifier
5. verifier checks $g^s \stackrel{?}{=} h^\sigma \cdot A$, which is true only if $h = g$

[29] Every message sent from the verifier is randomly sampled from a public domain distribution, as a challenge.

§10.4 Putting it Together: Anonymous Online Voting

§10.4.1 Homomorphic Encryption

Homomorphic encryption refers to encryption that allows for operations on the ciphertext corresponding to operations on the plaintext.

For example, an additively homomorphic scheme has the property that

$$\mathsf{Enc}(m_1) + \mathsf{Enc}(m_2) = \mathsf{Enc}(m_1 + m_2).$$

Similarly for multiplicatively homomorphic schemes,

$$\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2) = \mathsf{Enc}(m_1 \cdot m_2).$$

Example 10.2 (ElGamal Homomorphism)

Let's look at the ElGamal encryption scheme. We have a cyclic group $\mathbb{G}$ with generator g, public key pk.

$$\mathsf{Enc}_{pk}(m_1) = (g^{r_1}, pk^{r_1} \cdot m_1)$$
$$\mathsf{Enc}_{pk}(m_2) = (g^{r_2}, pk^{r_2} \cdot m_2)$$

We note that this is multiplicatively homomorphic (element-wise):

$$\mathsf{Enc}_{pk}(m_1) \cdot \mathsf{Enc}_{pk}(m_2) = (g^{r_1+r_2}, pk^{r_1+r_2} \cdot (m_1 \cdot m_2)) = \mathsf{Enc}_{pk}(m_1 \cdot m_2).$$

This gives us multiplicative homomorphism, but we want additive homomorphism (we want votes to add, not multiply).

We can consider exponential ElGamal, where the message is a power of g (and exponents add).

$$\mathsf{Enc}_{pk}(m_1) = (g^{r_1}, pk^{r_1} \cdot g^{m_1})$$
$$\mathsf{Enc}_{pk}(m_2) = (g^{r_2}, pk^{r_2} \cdot g^{m_2})$$

then

$$\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2) = (g^{r_1+r_2}, pk^{r_1+r_2} \cdot g^{m_1 \cdot m_2}) = \mathsf{Enc}(m_1 + m_2).$$

but how do we recover the message? We can decrypt (normally) to get $g^{m_1+m_2}$, but solving for $m_1 + m_2$ is *hard*, since it is a discrete log.

However, we're using this in the context of online voting. If $m \in \{0, \ldots, n\}$ for n the total number of voters (some polynomial range). This is fine for our uses![30]

[30] Normally, we talk about exponents from exponentially large sizes. Here, we can solve the discrete log since n is quite small.

§11 March 3, 2025

§11.1 Anonymous Online Voting

Recall anonymous online voting.

Say we have n voters with votes $v_1, \ldots, v_n \in \{0, 1\}$. Each voter encrypts their vote $\text{Enc}(v_1), \ldots, \text{Enc}(v_n)$. Our goal is to compute the sum of these votes without having to decrypt each vote individually. Somehow, we must find $\text{Enc}(\sum v_i)$ then decrypt to find $\sum v_i$.

There are three questions.

1. How do we compute $\text{Enc}(\sum v_i)$?
2. How do we ensure each vote is 0 or 1?
3. Who decrypts $\text{Enc}(\sum v_i)$?

§11.2 Additively Homomorphic Encryption

1. Additively homomorphic encryption is taking two encryptions $\text{Enc}(m_1)$ and $\text{Enc}(m_2)$ and combine them to get $\text{Enc}(m_1 + m_2)$.
2. Multiplicatively homomorphic encryption is taking two encryptions $\text{Enc}(m_1)$ and $\text{Enc}(m_2)$ and getting the product $\text{Enc}(m_1 \cdot m_2)$.

Example 11.1 (Elgalmal Encryption)

Elgamal Encryption: Cyclic group G with generator g, and the public key is given by $\text{pk} = g^{\text{sk}}$. The encryption of message m_1 is given by $\text{Enc}_{\text{pk}}(m_1) = (g^{r_1}, \text{pk}^{r_2} \cdot m_1)$ and the encryption of message m_2 is given by $\text{Enc}_{\text{pk}}(m_2) = (g^{r_2}, \text{pk}^{r_2} \cdot m_2)$. If we multiply the first components together and then the second components together, we get $(g^{r_1+r_2}, \text{pk}^{r_1+r_2} \cdot (m_1 \cdot m_2))$ which is exactly $\text{Enc}(m_1 \cdot m_2)$. Thus, we have multiplicatively homomorphic encryption.

Exponential Elgamal: The encryption of message m_1 is given by $\text{Enc}_{\text{pk}}(m_1) = (g^{r_1}, \text{pk}^{r_1} \cdot g^{m_1})$ and the encryption of message m_2 is given by $\text{Enc}_{\text{pk}}(m_2) = (g^{r_2}, \text{pk}^{r_2} \cdot g^{m_2})$. If we multiply them together element-wise like before, we get $(g^{r_1+r_2}, \text{pk}^{r_1+r_2} \cdot g^{m_1+m_2})$ which is exactly $\text{Enc}(m_1 + m_2)$. thus, we have additively homomorphic encryption.

How do we do decryption? Normally, we take $c_1 = g^{r_1+r_2}$ and $c_2 = \text{pk}^{r_1+r_2} \cdot g^{m_1+m_2}$ and compute c_2/c_1^{sk}. Usually this equals the plaintext, but in this scenario it equals $g^{m_1+m_2}$.

In our anonymous online voting scenario, each vote will be 0 or 1. Thus, the summation of all the votes is at most n, where n is the number of voters. This is a polynomial number of

possibilities, so we can compute g^m for each $m \in \{0, 1, \ldots, n\}$ and see which one matches $g^{\sum m_i}$ to recover the summation of the votes.

§11.3 Threshold Encryption

Definition 11.2 (t-out-of-n threshold)

In t-out-of-n threshold encryption, we must have t parties out of n parties come together in order to decrypt.

Let t parties be denoted by $p_1, \ldots, p_t$. Each party p_i works independently and runs a partial gen algorithm $\text{PartialGen}(1^\lambda)$, which generates a public and secret key pair $(\text{pk}_i, \text{sk}_i)$. After everyone is done, we combine all of the public keys pk_i to get one collective public key pk. A message is encrypted using it $\text{ct} \leftarrow \text{Enc}_{\text{pk}}(m)$. Note that a single party cannot decrypt by themselves.

In order to decrypt, each party p_i runs a partial decryption algorithm $\text{PartialDec}(\text{sk}_i, \text{ct})$ that gives a partial decryption d_i. Then, we combine all of the partial decryptions d_i to get the plaintext m.

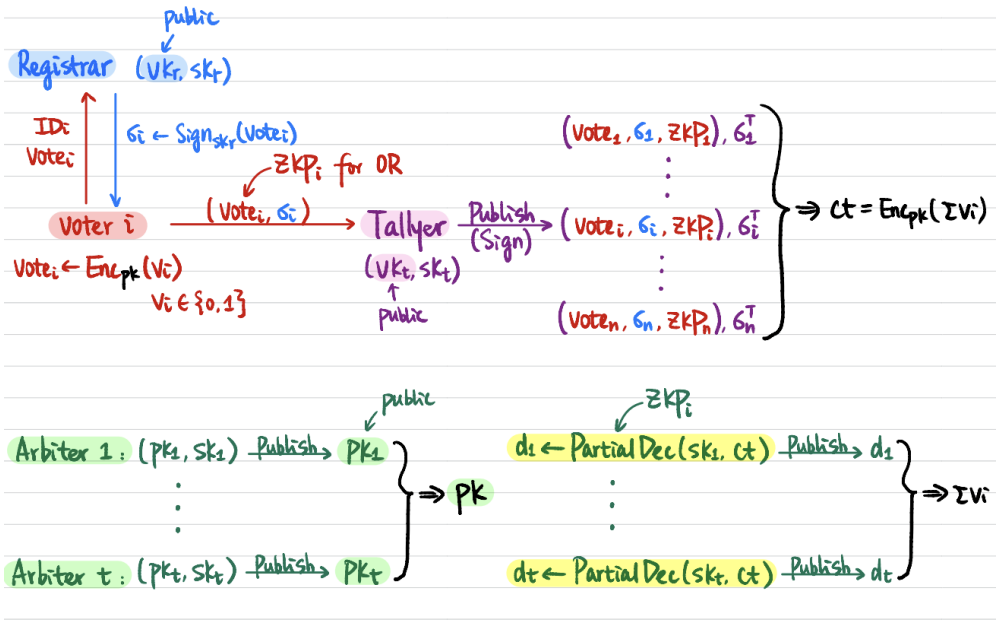
§11.3.1 Threshold Encryption: Elgamal

Now we give an explicit construction of threshold encryption using Elgamal.

Each party p_i generates a random secret key $\text{sk}_i \leftarrow \mathbb{Z}_q$ and public key $\text{pk}_i = g^{\text{sk}_i}$. Let pk be the product of all pk_i, which gives us $g^{\sum \text{sk}_i}$. Next we encrypt $\text{ct} = (c_1, c_2) = (g^r, \text{pk}^r \cdot g^m)$. In order to decrypt this, we need to compute c_2 / c_1^{sk} where $\text{sk} = \sum \text{sk}_i$. However, any single party does not know sk by themselves.

To decrypt, each party p_i does a partial decryption by computing $d_i = c_1^{\text{sk}_i}$. When all parties come together, they can multiply all the partial decryptions d_i which gives us $c_1^{\sum \text{sk}_i} = c_1^{\text{sk}}$. We can use this to compute c_2 / c_1^{sk} and decrypt.

§11.4 Voting Framework



We have some servers:

Registrar. For a voter to be able to vote, they register with the Registrar to obtain a certificate to vote. They get a certificate for their verification key.

Arbiters. The arbiters will generate the threshold encryption keys. There will be t arbiters and each will have their (pk_i, sk_i). They all reveal pk_i to the public, so that everyone can compute the full public key pk.

Voter. The voter, using the public key, will encrypt $v_i \in \{0, 1\}$. The voter will sign this vote using their signing key. They will send this vote to the Tallyer.

Tallyer. The tallyer will check that the signature is valid. Then, they will strip the signature and output $(vote_i, \sigma_i, zkp_i)$ for each vote.

§11.4.1 Correctness of Partial Decryption

Given a cyclic group G of order q with generator g, we have three pieces of public information.

1. The partial public keys of each party $\text{pk}_i \in G$.
2. The ciphertext $c = (c_1, c_2)$.
3. The partial decryption of each party d_i.

The witness is the partial secret key sk_i which is private to each party. The language for ZKP for partial decryption is

$$R_L = \{((c_1, pk_i, d_i), \underbrace{sk_i}_{\text{witness}}) : pk_i = g^{sk_i} \wedge d_i = c_1^{sk_i}\}$$

This is still the Diffie-Hellman tuple! $pk_i = g^{sk_i}, c_1 = g^r, d_i = g^{r \cdot sk_i}$. We can use the NIZK for Diffie-Hellman tuple as discussed in previous lectures.

§11.4.2 Correctness of Encryption

We want voters to prove that their encryption is either of 0 or 1. We're in group $\mathbb{G}$ with order q and generator g. We have public key $pk \in \mathbb{G}$, and ciphertext $c = (c_1, c_2)$. We're trying to prove the statement "c is an encryption of 0 OR c is an encryption of 1."

Our languages are then encryptions of 0 and encryptions of 1:

$$R_{L_0} = \{(\underbrace{(pk, c_1, c_2)}_{x}, \underbrace{r}_{\text{witness}}) : c_1 = g^r \wedge c_2 = pk^r\}$$
$$R_{L_1} = \{(\underbrace{(pk, c_1, c_2)}_{x}, \underbrace{r}_{\text{witness}}) : c_1 = g^r \wedge c_2 = pk^r \cdot g\}$$

where r is our private key. Using this, we can prove that c is an encryption of 0 ($c_2 = pk^r$) or c is an encryption of 1 ($c_2 = pk^r \cdot g$).

§11.5 Proving AND/OR Statements

For AND, our statements are x_1, x_2 and our witnesses are w_1, w_2. The language is given by

$$R_{AND} = \{((x_1, x_2), (w_1, w_2)) : (x_1, w_1) \in R_{L_1} \text{ AND } (x_2, w_2) \in R_{L_2}\}$$

To prove this language, we can use a ZKP for R_{L_1} and a ZKP for R_{L_2}.

For OR, our statements are x_1, x_2 and our witness is w. The language is

$$R_{OR} = \{((x_1, x_2), w) : (x_1, w) \in R_{L_1} \text{ OR } (x_2, w) \in R_{L_2}\}$$

To prove this language, we cannot use a ZKP for R_{L_1} and a ZKP for R_{L_2}. If we do, then we reveal whether $(x_1, w) \in R_{L_1}$ and whether $(x_2, w) \in R_{L_2}$, which is revealing more information than allowed.

To prove R_{OR}, we know that both languages R_{L_1} and R_{L_2} works with a sigma protocol. The prover is going to send (A_1, B_1) for the first language and (A_2, B_2) for the second language, pretending that both are correct. The verifier sends a challenge $\sigma \leftarrow \mathbb{Z}_q$. The prover separates σ into σ_1 and σ_2, and computes responds S_1, S_2 for σ_1, σ_2 respectively. Then the verifier will verify that $\sigma = \sigma_1 + \sigma_2$, as well as the responses $((A_1, B_1), \sigma_1, S_1)$ and $((A_2, B_2), \sigma_2, S_2)$.

How does the Prover compute a response for both statements? Since we are working with OR, we might not have inclusion in one of the languages. For that language, we will simulate it.

§12 March 5, 2025

This lecture we will continue our discussion on zero-knowledge proofs for OR statements, discuss RSA blind signature, and put it all together for anonymous online voting. Then we will discuss prime-order groups and more example of sigma protocols.

§12.1 Blind Signature

As a recap, in the voting framework, each voter receives a σ_i (certificate cert_i) from the Registrar, which is eventually published by the Tallyer. The Registrar is able to figure out who is voting, since they know exactly which signatures have been issued to the voters. To hide this information from the Registrar, we introduce something called blind signatures. This is a new idea this semester, and is used in practice.

Signer		**Requester**
$(\text{vk}, \text{sk}) \leftarrow \text{Gen}(1^\lambda)$		$(m', r) \leftarrow \text{Blind}(m)$
	$\xleftarrow{m'}$	
$\sigma' \leftarrow \text{SignBlind}_{\text{sk}}(m')$		
	$\xrightarrow{\sigma'}$	$\sigma = \text{Unblind}(\sigma', r)$
		$\text{Vrfy}_{\text{vk}}(m, \sigma) = 1$

If the Signer sees σ after this protocol, they will be unable to recognize if they signed it before.

§12.2 RSA Blind Signature

Signer		**Requester**
$(\text{vk}, \text{sk}) \leftarrow \text{Gen}(1^\lambda)$		$\text{Blind}(m)$:
		$r \leftarrow\!\!\$\ \mathbb{Z}_N^*$
		$m' := H(m) \cdot r^e \mod N$
	$\xleftarrow{m'}$	
$\text{SignBlind}_{\text{sk}}(m')$:		
$\sigma' := (m')^d$		
	$\xrightarrow{\sigma'}$	$\text{Unblind}(\sigma', r)$
		$\sigma := \sigma' \cdot r^{-1} \mod N$
		$\text{Vrfy}_{\text{vk}}(m, \sigma) = 1$

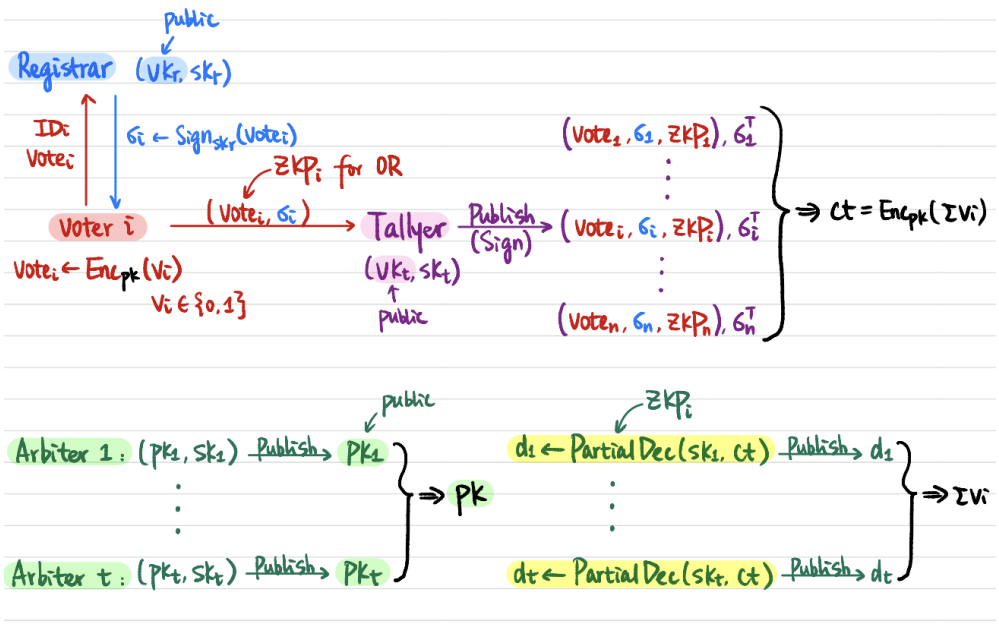
Notice that

$$\begin{aligned}\sigma' &= (m')^d \\ &= (H(m) \cdot r^e)^d \\ &= H(m)^d \cdot r^{ed} \\ &= H(m)^d \cdot r \mod N\end{aligned}$$

Thus $\sigma := \sigma' \cdot r^{-1} = H(m)^d \mod N$.

§12.3 Anonymous Online Voting

Here is a recap of the voting framework.



We have some servers:

Registrar. For a voter to be able to vote, they register with the Registrar to obtain a certificate to vote. They get a certificate for their verification key.

Arbiters. The arbiters will generate the threshold encryption keys. There will be t arbiters and each will have their (pk_i, sk_i). They all reveal pk_i to the public, so that everyone can compute the full public key pk.

Voter. The voter, using the public key, will encrypt $v_i \in \{0, 1\}$. The voter will sign this vote using their signing key. They will send this vote to the Tallyer.

Tallyer. The tallyer will check that the signature is valid and the user has not voted before. Then, they will strip the signature and output $\mathsf{Enc}_{pk}(v_1), \ldots, \mathsf{Enc}_{pk}(v_i), \ldots, \mathsf{Enc}_{pk}(v_n)$.

§12.4 Multiple Candidates

Recall earlier when we talked about homomorphic encryption, each voter has encrypts their vote $v_i \in \{0, 1\}$. This works if we have only two candidates. What if we want to have multiple candidates? Then, we need $v_i \in \{0, 1, \ldots, t-1\}$. Furthermore, $\text{Enc}(\sum v_i)$ no longer gives us the majority vote.

One idea is for each of the t candidates, we use $0/1$ voting. Every voter votes for each candidate, yes or no, and then the votes are summed and decrypted as in two candidate voting. This gives something like an "approval rating" for each candidate, where voters can vote for multiple candidates.

§12.5 More Examples of Sigma Protocols

Remark 12.1. The following are examples of sigma protocols. We do not go in depth in lecture, and we do not expect you to study these examples thoroughly at all - they are just examples of what ZKPs can look like besides the ones we have seen so far.

Example 12.2 (Okamoto's Protocol for Representation)

This is similar to Diffie-Hellman, except there are two group elements as input (h, u) instead of 3. Now we want to prove that $u = g^a h^b$.

The protocol is also similar to the Diffie-Hellman example, except the Prover will sample two random integers instead of one. One corresponds to a, and the other corresponds to b.

Input: Cyclic group G of order q, generator g, h, u.

Witness: (a, b)

Language: $R_L = \{((h, u), (a, b)) : u = g^a h^b\}$

Prover		**Verifier**
$r_1 \leftarrow\$ \mathbb{Z}_q$		
$r_2 \leftarrow\$ \mathbb{Z}_q$	$A = g^{r_1} h^{r_2} \mod q \longrightarrow$	
	$\longleftarrow \sigma$	$\sigma \leftarrow\$ \mathbb{Z}_q$
	$S_1 = \sigma \cdot a + r_1 \mod q$	
	$S_2 = \sigma \cdot b + r_2 \mod q \longrightarrow$	
		Verify: $g^{s_1} \cdot h^{s_2} = u^{\sigma} \cdot A$

For completeness, you can do the math and everything works out. It is saying that if both parties are honest, following the protocol, it will be verified. Proof of Knowledge was not covered in lecture, but it is similar to the examples covered in before lectures. Honest-Verifier Zero-Knowledge can be proved by considering the entire transcript, which can be simulated by a simulator without knowing the witness (a, b). The intuition is the same as before. You change the order of the messages. For example, sample σ, S_1, S_2 randomly in that order, then compute A, and that will be the exactly the same distribution as the reward.

Example 12.3 (Arbitrary Linear Equations)

Input: Cyclic group G of order q, generator g, h, u, v.

Witness: (a, b, c)

Language: $R_L = \{((h, u, v), (a, b, c)) : u = g^a h^b \ \wedge \ h = u^a v^b g^c\}$

The idea for the protocol is the same as before. We sample random r_1, r_2, r_3 which correspond to a, b, c respectively. A corresponds to $u = g^a h^b$ and B corresponds to $h = u^a v^b g^c$. Note that everything in the diagram is taken modulo q.

Prover		**Verifier**
$r_1, r_2, r_3 \leftarrow\$ \mathbb{Z}_q$		
	$A = g^{r_1} h^{r_2}$ $B = u^{r_1} v^{r_2} g^{r_3}$ $\longrightarrow$	
	$\longleftarrow$ σ	$\sigma \leftarrow\$ \mathbb{Z}_q$
	$S_1 = \sigma \cdot a + r_1$ $S_2 = \sigma \cdot b + r_2$ $S_3 = \sigma \cdot c + r_3$ $\longrightarrow$	
		Verify: $g^{S_1} h^{S_2} = u^{\sigma} \cdot A$ $u^{S_1} v^{S_2} g^{S_3} = h^{\sigma} \cdot B$

Completeness follows from assuming that the prover and verifier honestly follow the protocol, and we can confirm by following the math that it will be verified correctly. Proof of Knowledge is saying that we can extract knowledge from the prover. An Extractor will run the protocol with the Prover once, and then rewind and run the protocol again with a different challenge σ'. You can do this exercise offline. For Honest-Verifier Zero-Knowledge, the entire protocol can be simulated by a simulator without knowing the witness by changing the order of the messages. In particular, you can first sample σ and then sample S_1, S_2, S_3 randomly, which can be done because these numbers are random because of r_1, r_2, r_3, then compute A, B.

This example is similar to the previous example, except we have $h = u^a v^b g^c$ added to $u = g^a h^b$. In general, we can keep extending this language to an arbitrary number of equations of this form.

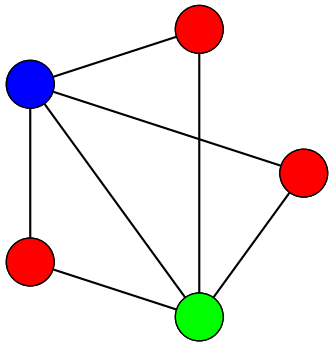
§13 March 10, 2025

This lecture we cover more examples of sigma protocols and zero-knowledge proofs for All NP.

§13.1 Zero-Knowledge Proof for Graph 3-Coloring (All NP)

Given a graph G, with vertices and edges, let the language be defined as the set of all graphs that have a 3-coloring. This is an NP language given by $L = \{G : G \text{ has 3-coloring}\}$. The NP relation is $R_L = \{(G, 3\text{Col})\}$.

Recall that a graph has a k-coloring if it is possible to color each vertex from a set of k colors such that adjacent vertices have different colors. For example, below is a graph with a three coloring.



Now we construct a proof without revealing the 3 coloring. Both the Prover and the Verifier can see the graph structure, but only the Prover can see the coloring. The Prover wants to prove that the graph is 3 colorable without revealing the coloring.

First, the Prover will hide all of the colors in the 3 coloring. Say that the Prover covers up each vertex with a piece of paper. They can color in the coloring, but after the paper goes down, the coloring cannot be modified. There is a way to do this cryptographically, which we will discuss later.

Next, the Verifier will give a random challenge, for example, choose two adjacent vertices. The Prover will reveal their colors, and show that they are different colors. Doing so will reveal the colors of some vertices, which is undesirable. Instead, we can map the original colors to a different color, so that whichever colors are revealed are random. We keep the set of colors the same. For example, we can remap the colors as follows.

$$\begin{aligned} \text{blue} &\to \text{red} \\ \text{red} &\to \text{blue} \\ \text{green} &\to \text{red} \end{aligned}$$

Checking one edge is not sufficient for the Verifier to be convinced that the graph is 3-colorable. If $G \notin L$, i.e. the graph is not 3-colorable, the probability that the Prover is caught is $\Pr[P^* \text{ is caught}] \geq$

$1/|E|$. This is because if the graph is not 3-colorable, then there exists at least 1 edge whose vertices have the same color. The probability of the Verifier picking this edge is $1/|E|$. There is some probability of catching the Prover, so we can amplify this, i.e. amplifying soundness.

One way to do this is for the Verifier to choose multiple edges. Then the Prover must remap the colors to avoid revealing the original coloring. The process is as follows: the Prover remaps the colors, then the Verifier chooses and edge, and the Prover reveals the colors of the vertices on that edge to show that they have distinct colors. This repeats multiple times, with the Prover remapping the colors randomly each time.

The Verifier is allowed to choose the same edge in multiple iterations. If the graph is not 3-colorable, the Prover might try to cheat by setting two adjacent vertices with the same color after they have been checked. Thus, the Verifier may want to check the same edge again to ensure that the Prover does not do so.

If we repeat this n times, the probability that the Prover P^* survives (not caught) is

$$\Pr[P^* \text{ survives}] \leq (1 - 1/|E|)^n.$$

If we pick $n = \lambda|E|$, then

$$\begin{aligned}\Pr[P^* \text{ survives}] &\leq (1 - 1/|E|)^{\lambda|E|} \\ &\approx (1/e)^{\lambda}.\end{aligned}$$

§13.2 Commitment Scheme

In the earlier 3-coloring example, the Prover places down a piece of paper on each of the vertices so that the color is hidden and cannot be modified. We discuss a cryptographic protocol that can achieve this, which is called a commitment scheme.

Sender		**Receiver**
$m \in \{0,1\}$		
Commit:		
$r \leftarrow\!\!\$\ \{0,1\}^\lambda$		
$c := \text{Com}(m; r)$	$\xrightarrow{\quad c \quad}$	
Open:		
	$\xrightarrow{\quad (m, r) \quad}$	Verify:
		$c = \text{Com}(m; r)$

There are two properties with this scheme:

- **Hiding:** The commitment of 0 is roughly the same as the commitment of 1, i.e. $\text{Com}(0; r) \approx \text{Com}(1; s)$.

- **Binding:** If one has committed to some message, then later on they can only open up to the message that they have committed. They cannot open up do something else. In other words, it is hard to find r, s such that $\text{Com}(0; r) = \text{Com}(1; s)$.

Example 13.1 (Hash-based commitment)

Randomly sample $r \leftarrow\!\!\$\ \{0,1\}^\lambda$. Then the commitment is $\text{Com}(m; r) := H(r||m)$ for a hash H. The hash is modeled as a random oracle.

This commitment scheme is **hiding** because the hash function output appears random. The **binding** property follows from collision resistance of H, which means that it is hard to find two inputs that give the same output.

Example 13.2 (Pedersen Commitment)

Take a cyclic group G with order q and generator g. Let $h \leftarrow\!\!\$\ G$ for $h = g^x$ where x is hidden to the sender. h can be generated by the receiver. Then randomly sample $r \leftarrow\!\!\$\ \mathbb{Z}_q$ and the commitment is $\text{Com}(m; r) = g^m \cdot h^r$.

Hiding holds because h^r appears as a random group element, so $g^m \cdot h^r$ is random and can be any group element since g is a generator, sort of like a one-time pad.

Binding follows from the discrete log assumption. If we find two $r_0, r_1 \leftarrow\!\!\$\ \mathbb{Z}_q$ with $\text{Com}(0; r_0) = \text{Com}(1; r_1)$, then

$$\begin{aligned} g^0 \cdot h^{r_0} = c = g^1 \cdot h^{r_1} \\ h^{r_0 - r_1} = g \\ h = g^{(r_0 - r_1)^{-1}} \end{aligned}$$

which essentially solves the Discrete Log problem, which is assumed to be hard. Thus, it is hard to find two such r_0, r_1.

§13.3 Zero-Knowledge Proof for Graph 3-Coloring

Now we give a protocol for a Zero-Knowledge Proof for Graph 3-Coloring.

Input: Graph $G = (V, E)$ with vertices V and edges E.

Witness: A coloring $\phi : V \to \{0, 1, 2\}$ that assigns vertices to colors 1, 2, 3.

Prover		**Verifier**
Randomly sample $\pi : \{0,1,2\} \to \{0,1,2\}$		
$\forall v \in V, r_v \leftarrow\$ \{0,1\}^\lambda, c_v := \text{Comm}(\pi(\phi(V)); r_v)$		
	$\xrightarrow{\{c_v\}_{v \in V}}$	
		Randomly pick an edge $(u, v) \in E$
	$\xleftarrow{(u,v)}$	
Open commitments C_u and C_v		
	$\xrightarrow{\alpha = \pi(\phi(u)), r_u \\ \beta = \pi(\phi(v)), r_v}$	Verify:
		$C_u = \text{Comm}(\alpha; r_u)$
		$C_v = \text{Comm}(\beta; r_v)$
		$\alpha, \beta \in \{0,1,2\}, \alpha \neq \beta$

This lets us prove all NP languages—we can do a reduction to the 3-coloring and prove it that way. In reality, this is expensive and merely a theoretical result.

§13.4 Circuit Satisfiability

In reality, many choose another NP-complete language, the circuit satisfiability problem. The language considers an arbitrary boolean circuit which consists of AND, XOR gates. The input are certain values x for input values, and witnesses w are the rest of the wires. The satisfiability problem is whether there exists some w to make the circuit evaluate to 1. Since the input can be any boolean circuit, this is adaptable and widely used in implementation.

This circuit model is considered a lot.

Example 13.3 (Pre-Image of Hash Function)

The function is $\text{C}(x, w) = H(w) - x + 1$. The circuit will output 1 on w such that $H(w) = x$. w here is the pre-image of x.

This allows us to, say, represent SHA as a boolean circuit to prove the pre-image of a hash function.

The intuition of the zero-knowlege proof is similar. Let's say the prover has some input values. The prover will commit to the bit of every wire.

For example, when a verifier asks to confirm a certain XOR gate, the prover will perform a *small*

zero-knowledge proof to prove that that gate was computed correctly. Composing commitments and using sigma protocols from before will allow us to gain the functionality we want.

Let's say

$$c_1 = \text{Com}(x)$$
$$c_2 = \text{Com}(y)$$
$$c_3 = \text{Com}(z)$$

and $x = y \oplus z$. Using a sigma-OR protocol, we can prove

$$(y = 0, z = 0, x = 0) \text{ OR } (y = 0, z = 1, x = 1) \text{ OR } \cdots$$

This allows us to do ZK-proofs for circuit satisfiability.

§13.4.1 Proof Systems for Circuit Satisfiability

We discuss the proof systems so far for circuit satisfiability.

The naïve proof is to reveal witness w. This is not zero-knowledge, but is non-interactive. Using Σ-protocols, we have zero-knowledge but not non-interaction. Using the Fiat-Shamir heuristic, we get both zero-knowledge and non-interaction.

For the easiest NP proof, communication requires $O(|w|)$ complexity and the verifier verifies in $O(|c|)$ (linear in number of gates) complexity. For Σ-protocols, communication requires a commitment to each wire, which is $O(|c| \cdot \lambda)$ (needs a factor of λ security parameter), and the verifier also verifies in $O(|c| \cdot \lambda)$. This is the same for NIZK.

	NP	Σ-Protocol	(Fiat-Shamir) NIZK
Zero-Knowledge	No	Yes	Yes
Non-Interactive	Yes	No	Yes
Communication	$O(\|w\|)$	$O(\|C\| * \lambda)$	$O(\|C\| * \lambda)$
Verifier's computation	$O(\|C\|)$	$O(\|C\|)$	$O(\|C\|)$

Even if we do our proof with the Fiat-Shamir heuristic, we will incur linear communication costs and computation costs. *Can we make this proof system more succinct?* In other words, can we have communication and verification complexity to be *sublinear* in $|c|$ and $|w|$? In other words, would it be possible to design a protocol that is even more efficient than just sending each witness in the clear?

Yes! We can do this with zk-SNARGs, which we will cover more in-depth next lecture.

§14 March 12, 2025

This lecture we cover in more detail Succinct Non-Interactive Arguments (SNARGs).

§14.1 Succinct Non-Interactive Argument (SNARG)

This brings us to succinct arguments, which are seemingly not quite possible.

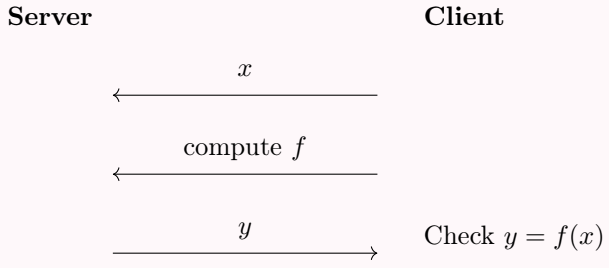
Definition 14.1 (Succinct Non-Interactive Arguments)
A non-interactive proof/argument system is succinct if

- The proof π is of length $|\pi| = \text{poly}(\lambda, \log |c|)$.
- The verifier runs in time $\text{poly}(\lambda, |x|, \log |c|)$.

Additionally, SNARKs are Succinct Non-Interactive Arguments of Knowledge. A zk-SNARG or zk-SNARK additionally guarantees zero-knowledge property.

Why succinct proofs? Here are some examples where we might want succinct proofs.

Example 14.2 (Verifiable Computation)
The client sends some x to the server, along with function f. The server sends back $y = f(x)$ and a proof. The client wants to check if the computation was done correctly.



If we did not have succinct proofs, then the client would still have to run the function again to verify the output. Note this allows interactions, so this is not the go-to example.

Is it possible? This remains as the large problem. Even in the naïve NP situation, we need to send the entire witness w and check the entire witness.

Enter probabilistically checkable proofs (PCP):

The prover prepares a proof and the verifier will only need to check certain bits of the proof.

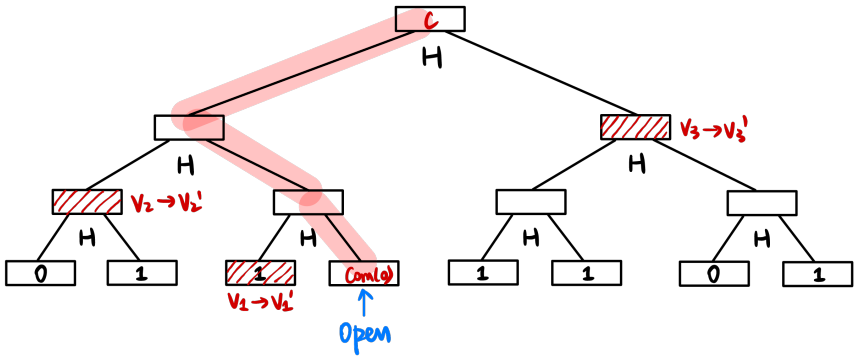
Theorem 14.3 (PCP Theorem, Informally)
Every NP language has a PCP where the verifier reads only a *constant* number of bits of the proof, to gain constant soundness.

The intuition is for the prover to commit the entire proof, the verifier checks certain bits, and the prover opens commitments.

The problem with this is that the first round message is not succinct (the commitment is just as long).

§14.2 Merkle Tree Commitment Scheme

Instead of committing linearly, we'll use a Merkle Tree, and only send the commitment/hash of the root node. We build up a binary tree where each node is the hash of its branches. Opening particular bits, the prover will send the root-to-leaf path along with siblings to prove that this opening was correct. This size will grow logarithmically with the size of the tree.



We hash values in a tree format, with each parent node being the hash of its children. We only send the *root note*. Whenever the verifier requests a certain bit, we send the path from the root to the bit (revealing all hashes, and siblings) to verify that this is indeed.

It's very difficult to change any bit. If we changed a bit, at some point up the path of the tree we'll have found a collision for a hash. That is to say, a specific bit being correct is predicated on whether the path to the root is valid and the root hash matches.

Can we make this hiding? Right now, we don't guarantee the hiding property. If we only had one layer, every bit would be revealed. How can we modify this algorithm to ensure that each bit is hiding?

One solution would be to add a random string r as a sibling to every leaf. However, this would require us to reveal all siblings when we're verifying a certain leaf node. We can easily modify this to *salt every* leaf node. We can add some random r_i to the hash of *every* bit that hides those bits.

Now, instead of sending a commitment of the entire proof, we send a Merkle Tree of the commitment of the proof. Then, when requested for certain bits i, j, k, we'll open those commitments as paths on the tree.

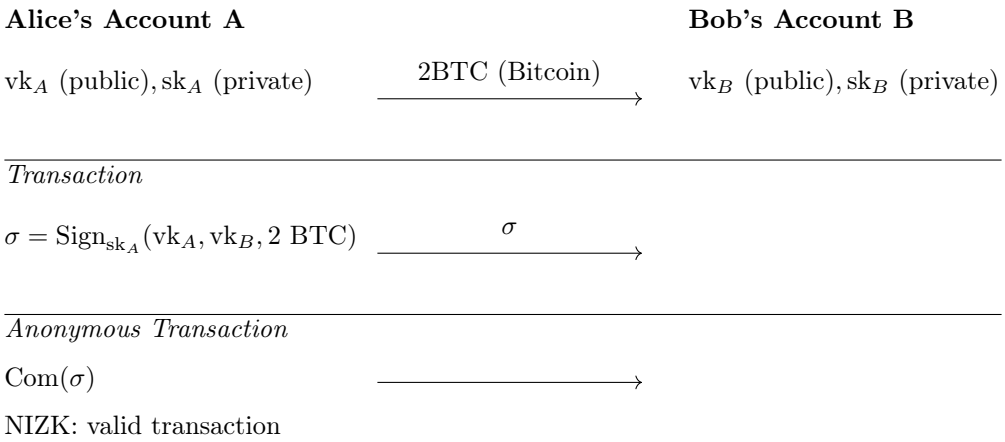
Is this zero-knowledge? Note that in the PCP theorem, we did not have the zero-knowledge property. Our solution is that when opening commitments, we can instead provide ZK proofs for our 'reveals' instead of the actual bits themselves. Asymptotically, this still preserves our succinctness property.

Theoretically, this lets us construct zk-SNARGs. In practice, there are more efficient ways to construct them, but we will not cover them now.

§14.3 Anonymous Transactions on Blockchains

We think of the blockchain as a public ledger. Say Alice wants to send 2 Bitcoin to Bob, Alice will sign the transaction using her signing key and add that transaction onto the ledger. All transactions are public, you know which addresses sent to which addresses. The public nature of the ledger allows all parties to verify transactions.

The blockchain is maintained, in a distributed way, by many parties called "miners." Each block contains a chain of transactions;



There's a lot of work to make transactions anonymous. We'll hide a transaction and hide it, and use a NIZK to prove that it is a valid transaction. We want these proofs to be non-interactive and succinct (we don't want users to spend too long doing verification). This is a major application of SNARK and zk-SNARK

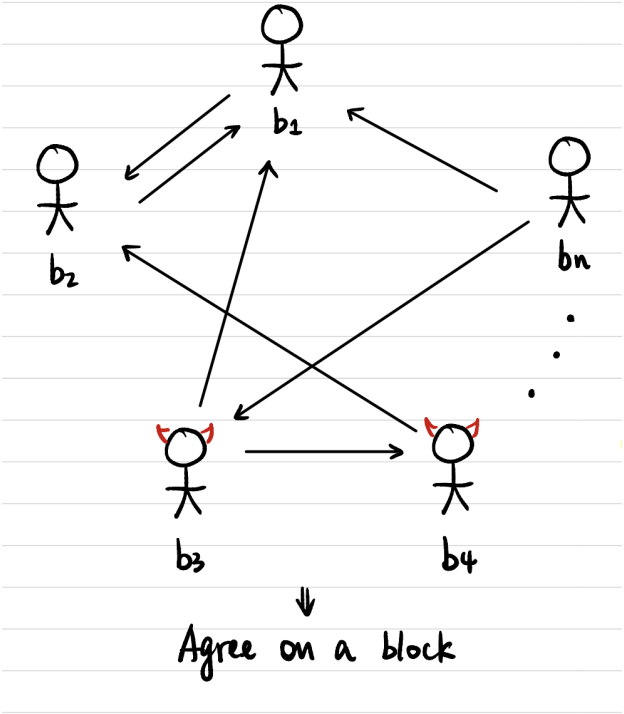
§14.3.1 Byzantine Agreement

Imagine a network where we have n nodes and t faulty nodes. How can we get them to agree? Byzantine Fault Tolerance (BFT) Protocol states that if $n \geq 3t + 1$ it is possible to reach concensus. However, we must assume $t < n/3$ - if we do, we can agree that the next block on the blockchain is a valid block.

In a permissionless protocol like Bitcoin, this is an issue - since anyone can become a node (i.e., become a miner), how do we ensure an honest majority, where an adversary cannot overrun the system with fake nodes? Bitcoin solves this problem with a concept called **proof of work**. Every node must use computation power to contribute to the blockchain.

In practice, the way this works is as follows: in order to find a block, the block (with some nonce) and the previous hash is hashed. Whichever node finds a block which can cash to a value with more than 30 leading zeros 'wins', and that block becomes the next one. This is a problem of raw computation and luck.

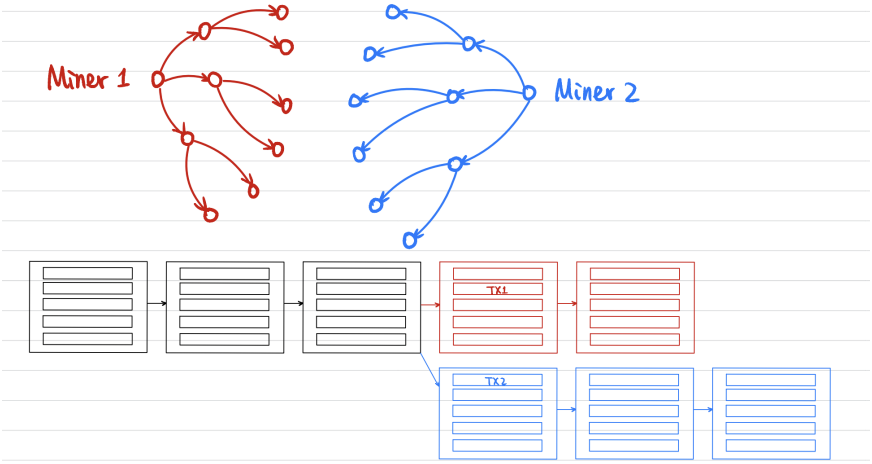
Why would miners want to devote massive amount of computation towards a game of luck? For Bitcoin, every transaction has some transaction fee. When a miner includes a transaction in a block that it solves, it gets that transaction fee. Furthermore, every block generates some new coin, which the winning miner also gets. In practice, miners often pool their computation into mining pools, which distibute the risk and reward across the mining pool.



§14.3.2 Longest Chain Rule

It is possible for two miners to compute two valid blocks at roughly the same time, which can cause nodes to get out of sync. Bitcoin solves this issue with the longest chain rule, which states that a node should always use the longest chain. Assuming honest majority of compute, the longest chain is always valid.

In practice, a block should not be considered valid until a node is confident it is on the longest chain. This can be assumed to be 6+ blocks deep.



§14.3.3 Extensions to Blockchain

The protocol described above is relevant to Bitcoin. Other protocols are built differently, with upsides and downsides to each. Other topics you can research (if interested) include:

1. Proof of Stake
2. Anonymous Transactions
3. Smart Contracts
4. Public bulletin

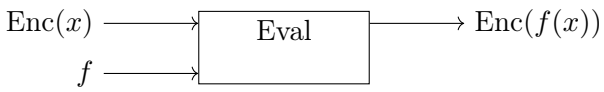
§15 March 17, 2025

In this lecture, we give a brief introduction to Fully Homomorphic Encryption. Then we give a concrete construction of Somewhat Homomorphic Encryption over Integers. We want to use more solid assumptions, so we introduce a new assumption called Learning with Errors, which is a post-quantum assumption, i.e. it is secure against known quantum attacks.

§15.1 Fully Homomorphic Encryption (FHE)

So far, our encryption schemes primarily follow an "all-or-nothing" idea, where if we have the secret key we can decrypt, otherwise we cannot decrypt.

In Homomorphic schemes, we want to have the additional property that an encryption of an input x can be evaluated with a function f to get an encryption of the output $f(x)$ without having to decrypt first.



An additively homomorphic scheme means we can combine $\mathsf{Enc}(m_1)$ and $\mathsf{Enc}(m_2)$ to get

$$\mathsf{Enc}(m_1 + m_2)$$

as we saw in Exponential ElGamal or Paillier.

Similarly, we can have a multiplicatively homomorphic scheme which means we can combine $\mathsf{Enc}(m_1)$ and $\mathsf{Enc}(m_2)$ to get

$$\mathsf{Enc}(m_1 \cdot m_2)$$

as we saw in ElGamal or RSA.

Fully homomorphic encryption means we can get both $\mathsf{Enc}(m_1 \cdot m_2)$ and $\mathsf{Enc}(m_1 + m_2)$.

§15.2 Applications

Example 15.1 (Oursourcing Storage & Computation)

Let's say a client stores some data on a server. A client has data x and key sk. $ct \leftarrow \mathsf{Enc}(x)$ is sent to the server. If the client wants to run some computation on the server, the client sends f and the server evaluates $ct' \leftarrow \mathsf{Eval}(f, ct)$ and sends ct' to the client, which gives the client $f(x)$ without the server knowing x.

Server		**Client**
		Data x
		Key sk
		$\text{ct} \leftarrow \text{Enc}(x)$
	$\text{ct}, f \longleftarrow$	
$\text{ct}' \leftarrow \text{Eval}(f, \text{ct})$		
	$\text{ct}' \longrightarrow$	
		$f(x) \leftarrow \text{Dec}_{\text{sk}}(\text{ct}')$

Example 15.2 (Privacy-Preserving Query)

The client wants to make a query to the server, e.g. Google search or GPT-4. However, the client does not want to reveal what their query is. Thus, the client homomorphically encrypts their query and the server homomorphically processes the query and sends back the evaluated ciphertext ct', so that the client gets the query result and the server does not know the query.

Server		**Client**
		Query x
		Key sk
		$\text{ct} \leftarrow \text{Enc}(x)$
	$\text{ct} \longleftarrow$	
$\text{ct}' \leftarrow \text{Eval}(f, \text{ct})$		
f can be Search, ML, GPT, etc.		
	$\text{ct}' \longrightarrow$	
		$f(x) \leftarrow \text{Dec}_{\text{sk}}(\text{ct}')$

The function f has a database embedded in it, and outputs $D[i]$ for some location i. How we build this function will be the focus of project 4 (PIR).

Example 15.3 (Private Information Retrieval (PIR))

In this application (We'll implement this in the next project!), we have some server with a database. A client wants to retrieve the i-th element without revealing the index i to the server.

Server		**Client**
Database D		Want: $D[i]$ while hiding i from the server
		Query: i
		Key: sk
	$\xleftarrow{\text{ct}}$	$\text{ct} \leftarrow \mathsf{Enc}(i)$
$\text{ct}' \leftarrow \text{Eval}(f, \text{ct})$		
where $f(i) = D[i]$		
	$\xrightarrow{\text{ct}'}$	
		$D[i] \leftarrow \mathsf{Dec}_{\text{sk}}(\text{ct}')$

Note two differences from 1-out-of-n OT.

- **Security:** In 1-out-of-n OT, the server does not want to reveal any information about their database other than the 1 out of n chosen entry. However, in PIR, we do not enforce such security. In PIR, we allow that the client may learn something else about the database from the ciphertext ct' other than the desired result $f(x)$.

- **Efficiency:** In 1-out-of-n OT, the server will generate n ciphertexts and send them to the client. However, in PIR, we want to be more succinct and only send one ciphertext.

A naïve solution is for the server to send the *entire* database to the client and the client can just access their desired element. In fact, this is the best we can do information-theoretically.

§15.3 FHE Definition

Definition 15.4 (Homomorphic Encryption)

A (public-key) homomorphic encryption scheme is some

$$\pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$$

with respect to some function family $\mathcal{F}$ with

- $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
- $ct \leftarrow \mathsf{Enc}_{pk}(m) \qquad m \in \{0,1\}$.
- $m \leftarrow \mathsf{Dec}_{sk}(ct)$.
- $ct_f \leftarrow \mathsf{Eval}(f, ct_1, \ldots, ct_n)$ with $f : \{0,1\}^n \to \{0,1\}$ in family $\mathcal{F}$.

Correctness requires that $\forall f \in \mathcal{F}$, if $ct_f \leftarrow \mathsf{Eval}(f, ct_1, \ldots, ct_n)$ for $ct_i \leftarrow \mathsf{Enc}_{pk}(m_i)$, then $\mathsf{Dec}_{sk}(ct_f) = f(m_1, \ldots, m_n)$. That is, that evaluating functions does indeed give the ciphertext of the function evaluated on the plaintexts.

CPA security, as we've seen before, requires that

$$(pk, \mathsf{Enc}_{pk}(m_0)) \overset{c}{\simeq} (pk, \mathsf{Enc}_{pk}(m_1)).$$

Compactness, that $|ct_f| \leq \mathsf{poly}(\lambda)$, that each ciphertext is bounded by some constant that is polynomial in λ and fixed for security parameter λ. This is necessary for several reasons: practically, when outsourcing compute, we want to decrease the netork costs we incur. Furthermore, this allows us to prevent the 'loophole' of evaluations returning itself.

Why do we need compactness? If we just have correctness and CPA security, our current definition is nearly trivial. For example, if our evaluation just output the tuple $(f, ct_1, \ldots, ct_n)$. To decrypt, we decrypt each ct_i individually and apply f on top, and this satisfies our definitions! We need to add some notion that our ciphertext must stay within some size, and that we're actually *combining* our ciphertexts.

If the family of functions $\mathcal{F}$ contains all polynomial sized Boolean circuits, we say that the scheme is **fully** homomorphic.

§15.3.1 Constructions

All FHE constructions follow two steps:

1. Somewhat Homomorphic Encryption (SWHE). We'll see versions over integers, from LWE (GSW), and from RLWE (BFV).

2. Then, we bootstrap our SWHE schemes to be fully homomorphic.

§15.3.2 SWHE over Integers

Attempt 1. This is our first attempt is a secret-key scheme:

Our secret key will be odd p.

> **Remark 15.5.** *Why must p be odd?* If p is even, then the ciphertext $p \cdot q + m$ has the same parity as m. The message is not secure because we can find it just by checking even/odd. If p is odd, then $p \cdot q$ can be either even or odd, so the parity of the ciphertext is not solely determined by parity of m.

- To $\mathsf{Enc}(m)$ with $m \in \{0, 1\}$, we sample a random q and compute $ct = p \cdot q + m$. Encryption of 0 is a multiple of p.
- Decryption is $ct \mod p$.
- Add: $\mathrm{ct} \leftarrow \mathrm{ct}_1 + \mathrm{ct}_2$.
- Multiply: $\mathrm{ct} \leftarrow \mathrm{ct}_1 \cdot \mathrm{ct}_2$.

Is this CPA secure? No! If we have a lot of encryptions of 0s, taking the gcd of them will give us p exactly.

Attempt 2. Our next attempt is to add a small error noise.

- To encrypt, instead of $ct = p \cdot q + m$, we'll do
$$ct = p \cdot q + m + 2e$$
for small even $2e$ with $e \ll p$. Encryption of 0 is small and even modulo p.
- Then, decryption becomes $\mathsf{Dec}(ct) = [ct \mod p] \mod 2$.
- Addition and multiplication work the same way. Check that the decryption gives the correct result by modding p then modding 2. If
$$\begin{aligned} \mathrm{ct}_1 &= p \cdot q_1 + m_1 + 2e_1 \\ \mathrm{ct}_2 &= p \cdot q_2 + m_2 + 2e_2 \end{aligned}$$
then
$$\begin{aligned} \mathrm{ct}_1 + \mathrm{ct}_2 &= p(q_1 + q_2) + (m_1 + m_2) + (2e_1 + 2e_2) \\ \mathrm{ct}_1 \cdot \mathrm{ct}_2 &= pq_1(pq_2 + m_2 + 2e_2) + m_1 \cdot pq_2 + m_1 \cdot m_2 \\ &\quad + m_1 \cdot 2e_2 + 2e_1(pq_2 + m_2 + 2e_2) \end{aligned}$$

Attempt 3. We can also construct a public key version of this, where the secret key is still our odd p but the public key are a set of encryptions of 0, $\{x_i = p \cdot q_i + 2e_i\}$.

Since our scheme is homomorphic over addition, we can take a random subset sum of $\{x_i\}$ and add m and $2e$:

- To encrypt, we'll do
$$ct = (\text{subset sum of } \{x_i\}) + m + 2e$$
for small even $2e$ with $e \ll p$. Encryption of 0 is small and even modulo p.
- Then, decryption becomes $\mathsf{Dec}(ct) = [ct \mod p] \mod 2$.
- Addition and multiplication work the same way.

What could go wrong? Note that this is still only a *somewhat* homomorphic encryption scheme. Are there limits to how much we can add or multiply? Specifically, could the noise grow to an extent that interferes with our encryption?

1. If we add ciphertexts, our noise grows linearly:
$$\begin{aligned} ct_1 &= p \cdot q_1 + m_1 + 2e_1 \\ ct_2 &= p \cdot q_2 + m_2 + 2e_2 \\ ct_1 + ct_2 &= p \cdot (q_1 + q_2) + (m_1 + m_2) + 2(e_1 + e_2) \end{aligned}$$
2. If we multiply ciphertexts, our noise grows exponentially:
$$ct_1 \cdot ct_2 = p \cdot (\cdots) + (m_1 \cdot m_2) + m_1 \cdot 2e_2 + 2e_1 m_2 + 4e_1 e_2$$

Addition is cheaper, but multiplication has our noise grow much much faster. In our parameters, we can support roughly $O(\lambda)$ multiplications, but addition is almost for free (we can do exponentially many additions).

This is why this is *somewhat* homomorphic encryption—if the noise exceeds p, then it might affect the plaintext.

This analysis works similarly for public-key encryption[31].

Note that this protocol is quite expensive—p and q will tend to get quite large. We'll pivot into lattice-based cryptography which will solve some of these issues.

[31] In fact, we took a very generic way of converting the secret-key scheme into a public-key scheme. We can always take subset sums of encryptions of 0 and add it to our message.

§15.4 Learning With Errors (LWE) Assumption

We'll introduce a new assumption, and which is assumed to be post-quantum secure. There is no known quantum algorithm to solve this in polynomial time.

We have security parameter n, and $q \sim 2^{n^\epsilon}$ for constant ϵ. $m = \Theta(n \log q)$. We have some distribution χ which is a distribution over $\mathbb{Z}_q$ concentrated on "small integers". For example, this can be Gaussian.

We require that for $e \leftarrow\!\!\$\ \chi$,

$$\Pr[|e| < \alpha \cdot q] \simeq 1$$

with $\alpha \ll 1$. That is, the probability that our error deviates significantly from 0 is negligible.

The assumption states the following: sampling a random matrix $A \xleftarrow{\$} \mathbb{Z}_q^{m \times n}$ and randomly sample a vector $s \xleftarrow{\$} \mathbb{Z}_q^n$ with error $e \xleftarrow{\$} \chi^m$. We have

$$b := A \times s + e$$

The computational assumption is that

$$(A, b) \overset{c}{\simeq} (A, b')$$

for $b' \xleftarrow{\$} \mathbb{Z}_q^m$. That is, (A, b) is indistinguishable from a truly random (A, b').

Why do we need the error term e? Without e, in our equation

$$b := A \times s$$

, is it possible to solve for s