CSCI 1515: Applied Cryptography

P. Miao

Spring 2023

These are lecture notes for CSCI 1515: Applied Cryptography taught at Brown University by Peihan Miao in the Spring of 2023.

These notes are taken by Jiahua Chen with gracious help and input from classmates and fellow TAs. Please direct any mistakes/errata to me via email, post a thread on Ed, or feel free to pull request or submit an issue to the notes repository.

Notes last updated April 13, 2023.

Contents

§1 January 26, 2023

§1.1 Introduction

If you find a lot of courses on cryptography online or at universities, you'll find a lot of theoretical content. However, it's helpful for students to get hands-on experience with cryptography:

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

The goal of this course is to

Introduction: Peihan Miao, please refer by Peihan (you will be corrected if you address as professor). Joined Brown last semester, taught a seminar on research topic (Secure Computation). Before that, was in Chicago, and Visa Research even before that. Really loved math and theoretical computer science, started PhD with theoretical computer science. Shifted to applied side of cryptography; it's exciting to see cryptography implemented in practice. However, realized that most crypto courses are very theoretical—there are not a lot of courses that build up advanced applications using the tools that have been set up.

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, 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[1]

§1.2 Course Logistics

The course homepage is at https://brownappliedcryptography.github.io/.

Huge kudos to Nick and Jack for developing a new course from the ground up!

Please view the syllabus here. *If there are differences between this document and the syllabus, the syllabus trumps this document.*

The course is offered in-person in CIT 368, as well as synchronously over Zoom and recorded asynchronously (lectures posted online). You can do either[2] but try to attend online because there will be interaction! Lecture attendance is encouraged!

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

[1] CSCI 1510 and Math 1580 are good candidates. 1510 covers cryptographic proofs, and Math 1580 covers cryptography from a number theoretic perspective.

[2] It is a 9AM class...

For assignments: *Project 0* is a warmup for C++. Projects 1 & 2 is to develop secure communication or authentication systems using the underlying cryptographic libraries. The later project, 3, 4 & 5 will be on more advanced topics. The first 2 are more basic that are developed in practice, and the latter ones are more experimental in practice (so this is recent research in applied cryptography). The final project will be a combination of the existing projects or a project entirely new (separate from the earlier projects, but using the same cryptographic primitives).

Projects 1 through 5 will be accompanied with homework assignments (appropriately numbered 1 through 5) that develop a conceptual understanding of the materials.

Projects 1 & 2 are to be done individually, the later projects are done in pairs (you can choose to go solo if you so wish as well). You are encouraged to find partners earlier on to discuss and work with them from the beginning. You are *encouraged* to communicate with your partners on projects 1 & 2 so you gain a conceptual understanding. You should complete your own write-up and code for the first two projects, however.

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*
Project 0	5%
Projects 1 & 2	20% (10% each)
Projects 3, 4 & 5	45% (15% each)
Homeworks	20% (2% each)
Final Project	20%

You have 6 late days for *projects*, of which at most two can be used on a single project. Additionally, you have 3 late days for *homeworks*, of which at most one can be used on a single homework.

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

§1.3 What is cryptography?

...or more importantly, what is cryptography used for?

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

Question. What are some cryptographic techniques used in practice and what does it achieve?

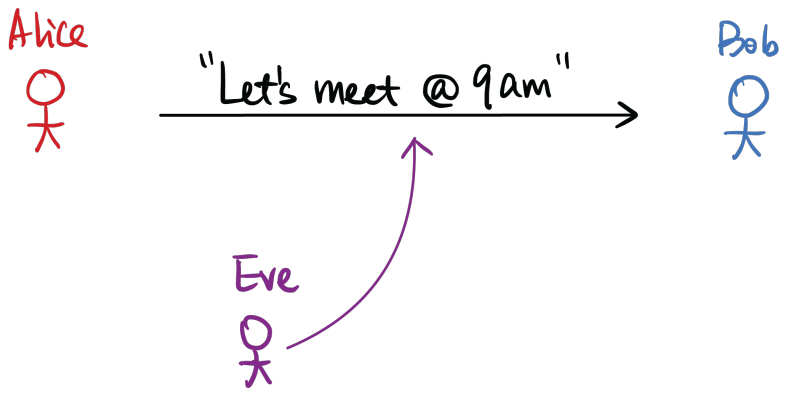
- Used in financial institutions.

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

- Might be used in voting schemes. Making sure that ballots are kept private.
- Messaging systems, recently.
  - End-to-end texting.
- Storing medical records.
  - More generally, any sensitive information.
  - You don't want anyone else getting access to this sensitive information.
- Used in military or war.
  - Historically, it was used a lot in the military for secure communication.
- Authentication systems. Login systems.
  - There is authentication going on that ensures that *only you* can log in.

§1.4 Secure Communication

We'll start with the most primitive 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.

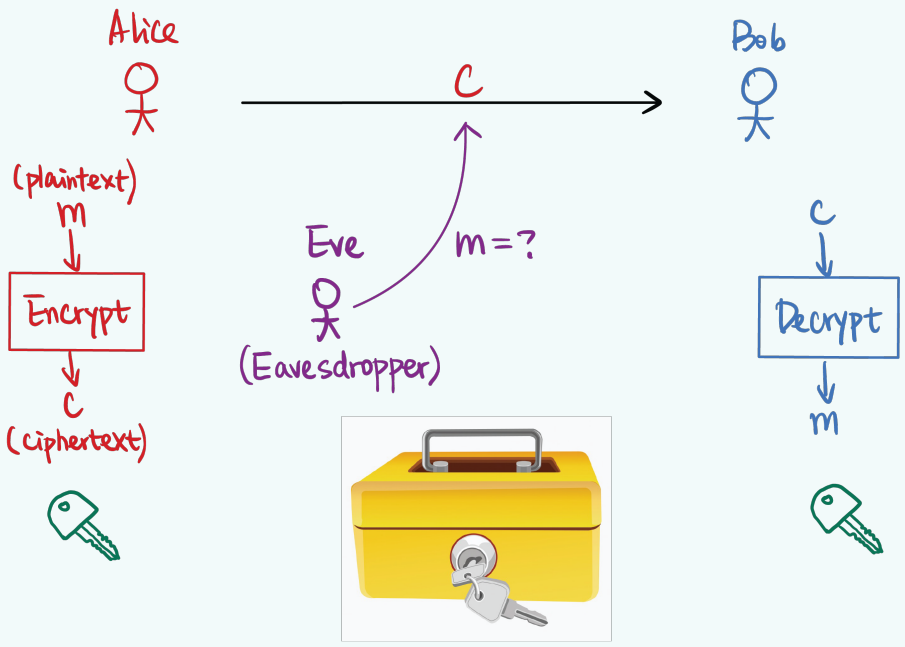
These two guarantees are the most important guarantees! The former is called message secrecy, the latter is called message integrity.

§1.4.1 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.

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.



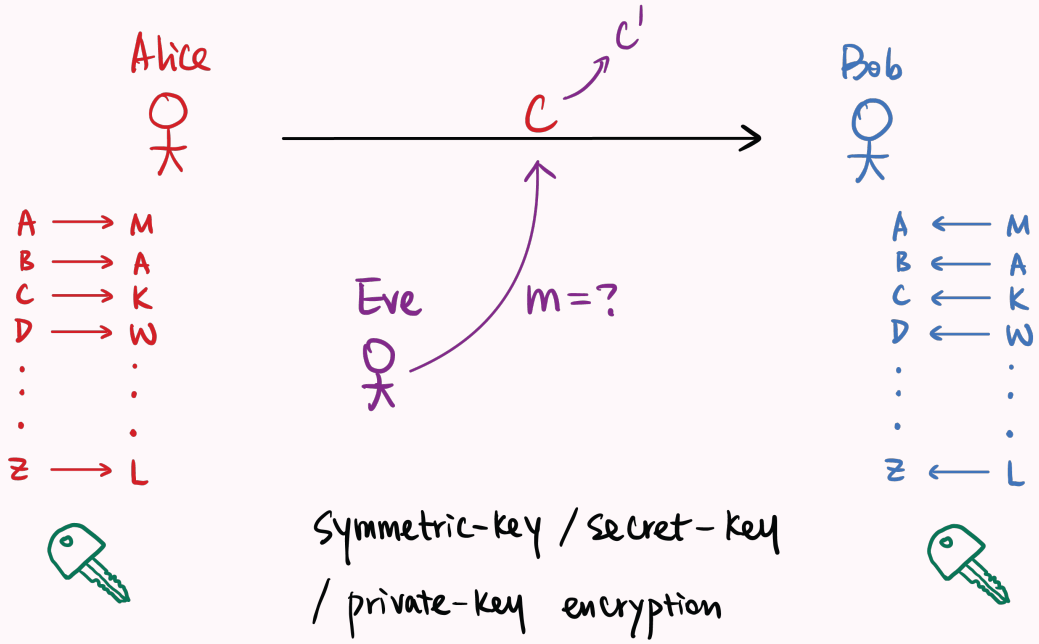
In this model, Eve is a weaker adversary, an *eavesdropper*. Eve can only see the message, not alter it.

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.



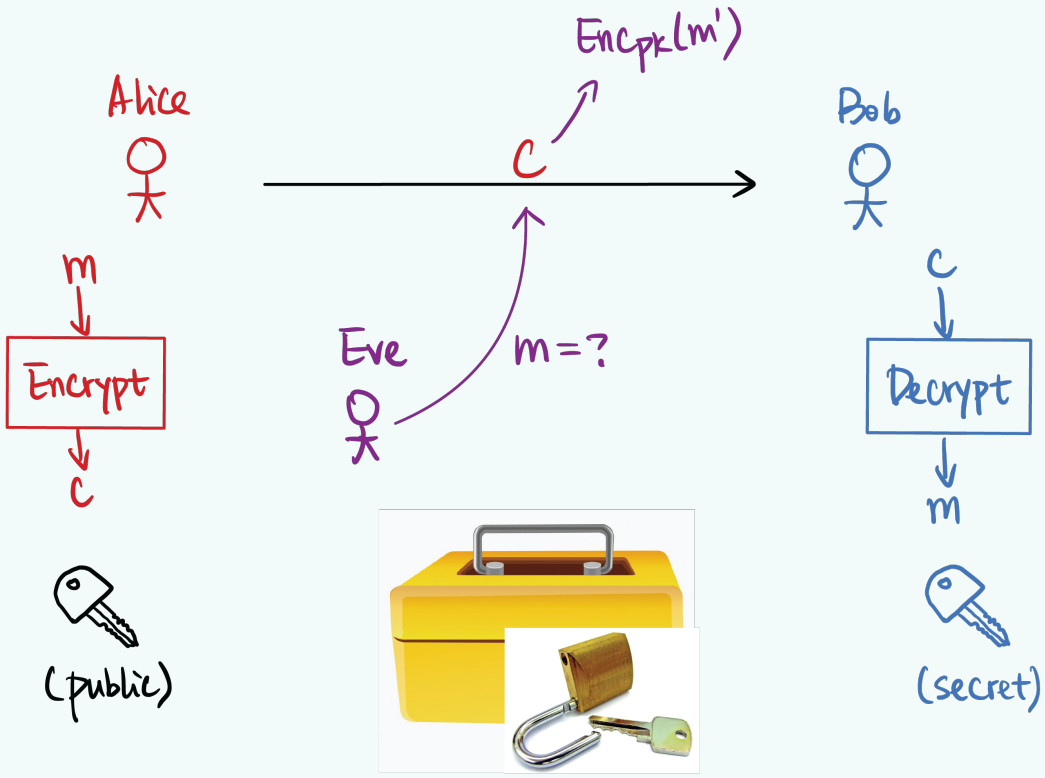
This scheme is not quite secure! *Why?*

You could guess a bunch of vowels and see what words could make up. If you have a long enough message, you can see which letters appear more often. We know that in English, the vowels appear more often; and you can make a lot of guesses.

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 meet up first to exchange secret keys.

Definition 1.3 (Public-key Encryption)

There is another primitive that is much more ideal/stronger, public-key encryption. Bob publishes both a *public* key and a *private* key. You can consider a lock where you don't need a key to lock it[3], and only Bob has the key to unlock it.

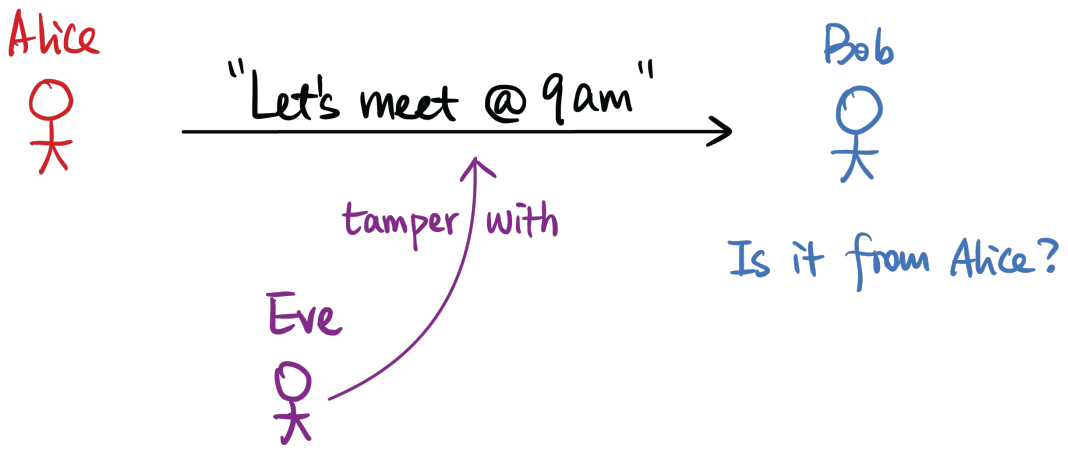


This is seemingly magic! Bob could publish a public key on his homepage, anyone can encrypt using a public key but only Bob can decrypt. *Stay tuned, we will see public-key encryption schemes next lecture!*

§1.4.2 Message Integrity

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

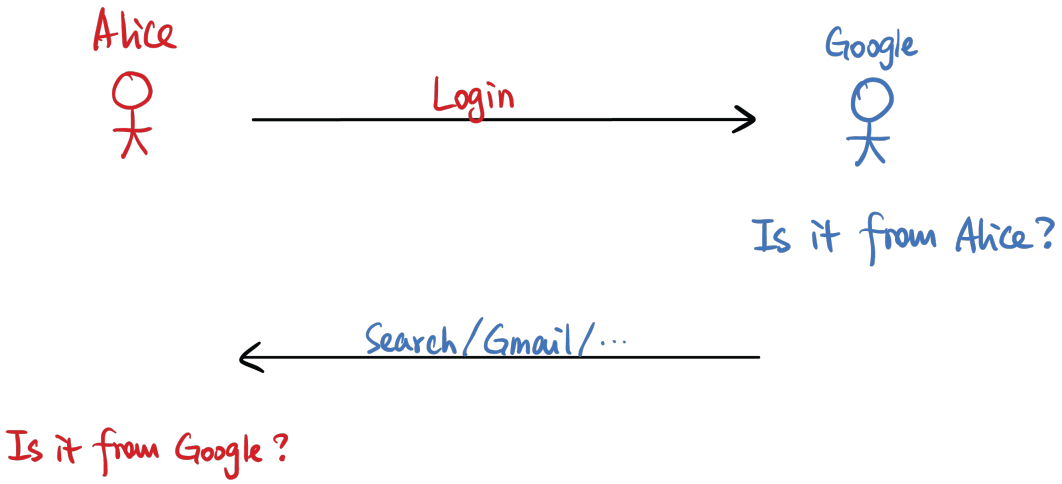
[3] You literally click it closed



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.



§1.5 Project Overview

0. Warm-up, you will implement some basic cryptographic schemes.
1. Secure Communication: what was just introduced.
2. Secure Authentication: how to authenticate yourself to a server, also mentioned just now.
3. Zero-Knowledge Proofs: we'll use ZKPs to implement a secure voting scheme.
4. Secure Multiparty Computation: we'll implement a way to run any function securely between two parties.
5. Fully Homomorphic Encryption: a form of post-quantum cryptography.

We'll quickly introduce the concepts used in the projects.

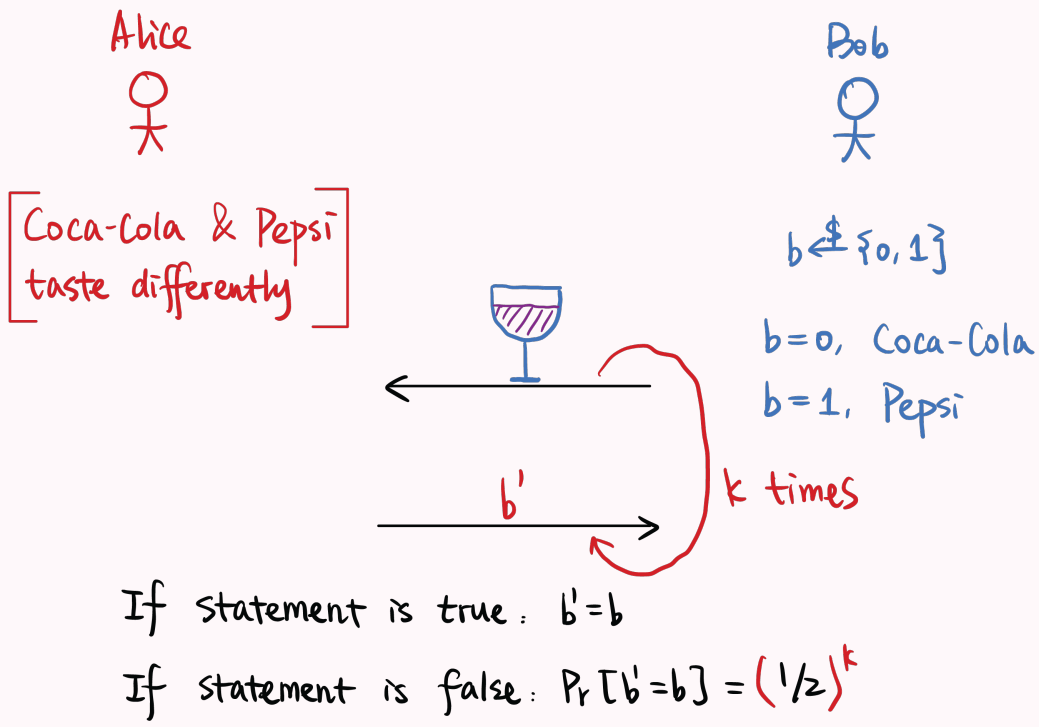
§1.5.1 Zero-Knowledge Proofs

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

Example

Alice claims to be able to differentiate between Coca-Cola and Pepsi! She wants to prove this to Bob without revealing her secrets.

Bob will randomly sample a bit $b \xleftarrow{\$} \{0,1\}$, with $b = 0$ being Coca-Cola and $b = 1$ being Pepsi. Bob will let Alice taste this drink. Alice will give a guess b' of what drink it is.



If the statement is true, $b' = b$ (Alice gives the correct prediction).

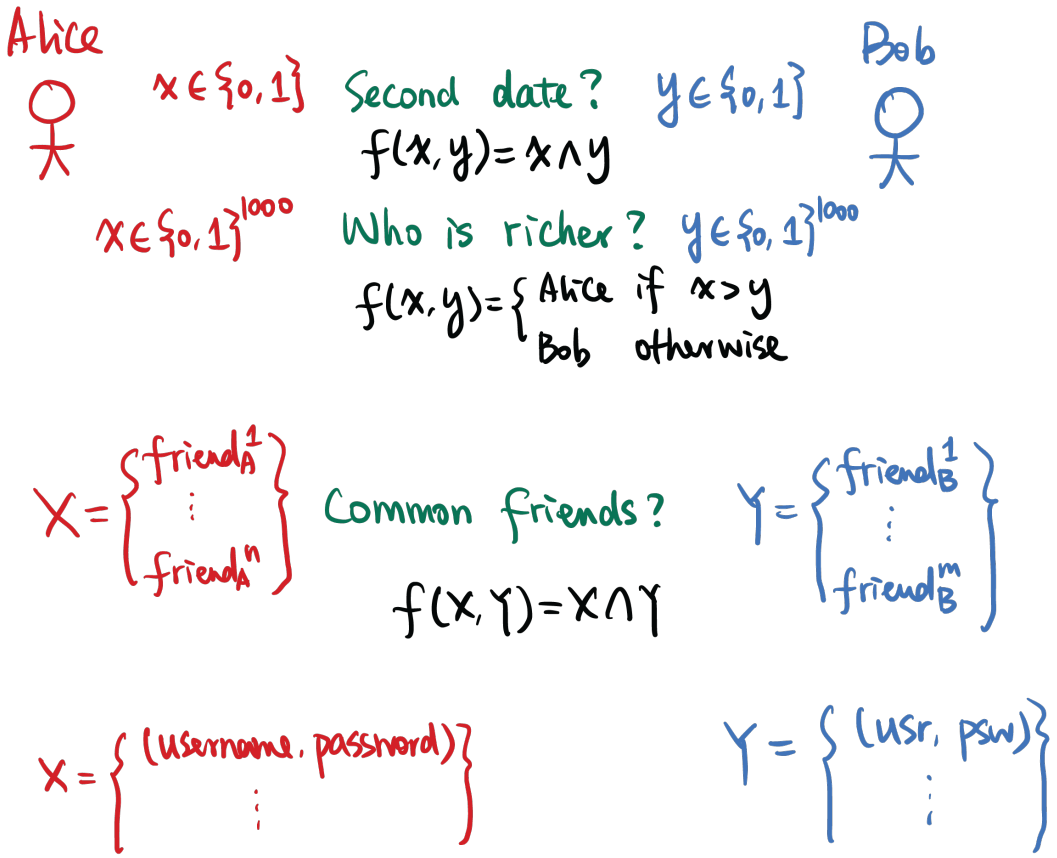
If the statement is false, $\Pr[b' = b] = \frac{1}{2}$ (Alice is guessing at 0.5 probability).

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.

§1.5.2 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 $f(x,y) = x \wedge y$.

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 is revealed weaker than using a trusted third-party?

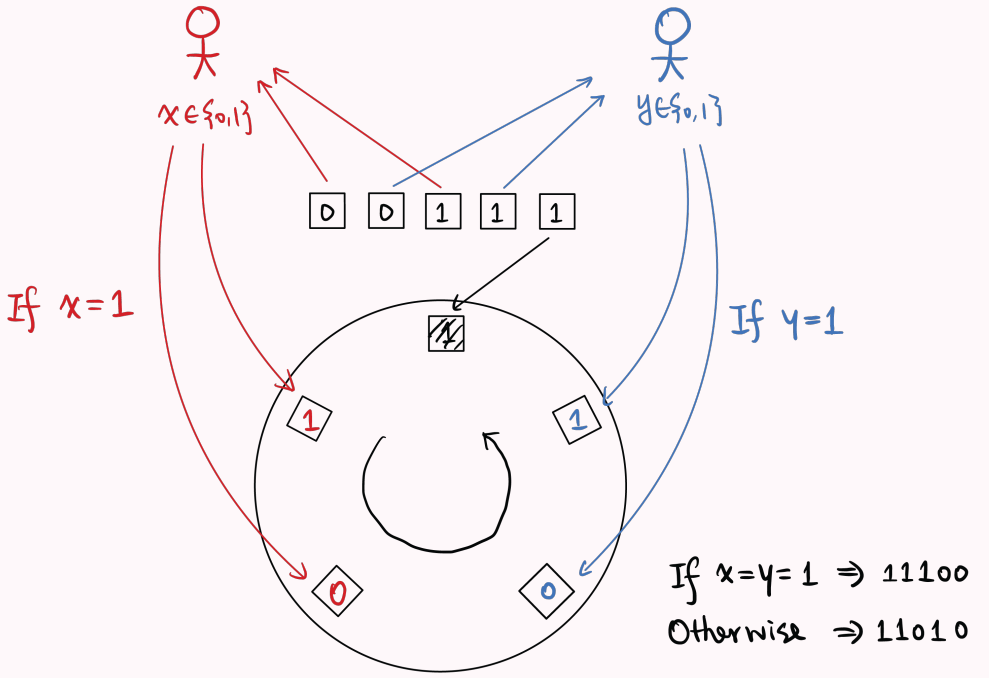
Yes, however in some cases (hospital health records), parties are legally obliged to keep data secure. We wish for security more than the secrecy of cardinality.

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 1 facing down.



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

Question. If Alice puts 1, and the output is 0, she could infer the information from Bob. However, this is allowed. Whatever can be inferred from the desired output is inferred.

The more sensitive part is when if you put a 0, you don't learn whether the other party also put down a 0.

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.

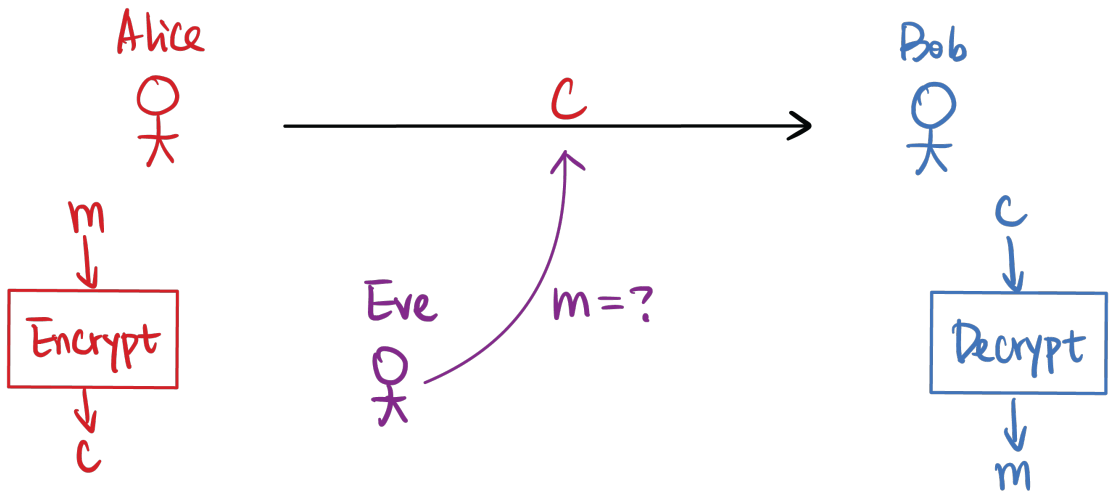
§1.5.3 Fully Homomorphic Encryption

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.

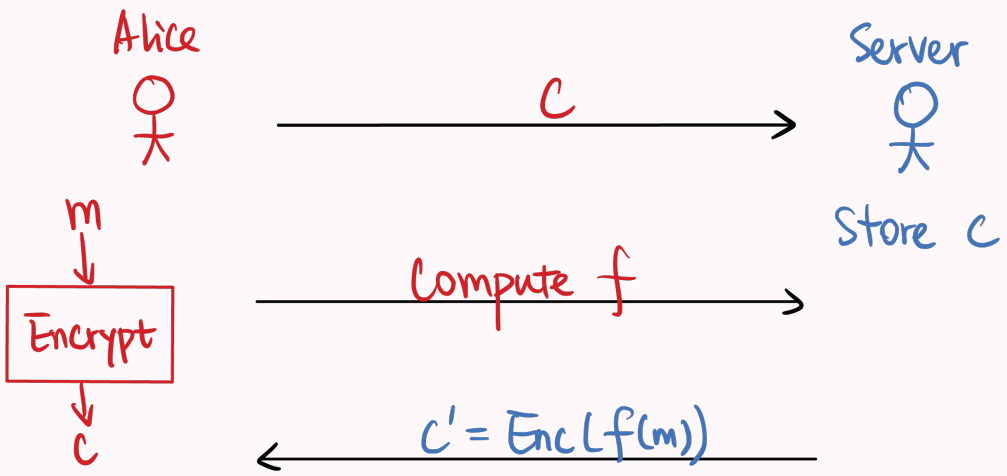


$$c_1 = \mathsf{Enc}(m_1)$$
$$c_2 = \mathsf{Enc}(m_2)$$
$$\Rightarrow \quad c' = \mathsf{Enc}(m_1 + m_2)$$
$$c'' = \mathsf{Enc}(m_1 \cdot m_2)$$

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.

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. This is also the only cryptographic primitive that is quantum-secure[4].

§1.5.4 Further Topics

We might cover some other topics:

- Differential Privacy
- Crypto applications in machine learning
- Crypto techniques used in the blockchain[5]

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?*

Peihan will collect responses at the start and middle of the semester to shape course content.

[4] Everything before this can be broken if quantum computers become mainstream!
[5] One important techniques is Zero-Knowledge proofs, for example.

§1.6 A Quick Survey

Peihan conducted the following poll in-class to gauge content for future lectures. ***By all means, you don't need to know any/all of this going into this course! These will be self-contained in this course!***

Do you know what the following means?

- Polynomial-time algorithm.
- NP-hard problems.
- "a divides b" ($a \mid b$)
- GCDs
- (Extended) Euclidean Algorithms
- Groups
- One-Time pads
- RSA encryption/signature
- Diffie-Hellman Key Exchange
- SHA (hash functions)

§2 January 31, 2023

§2.1 Logistics

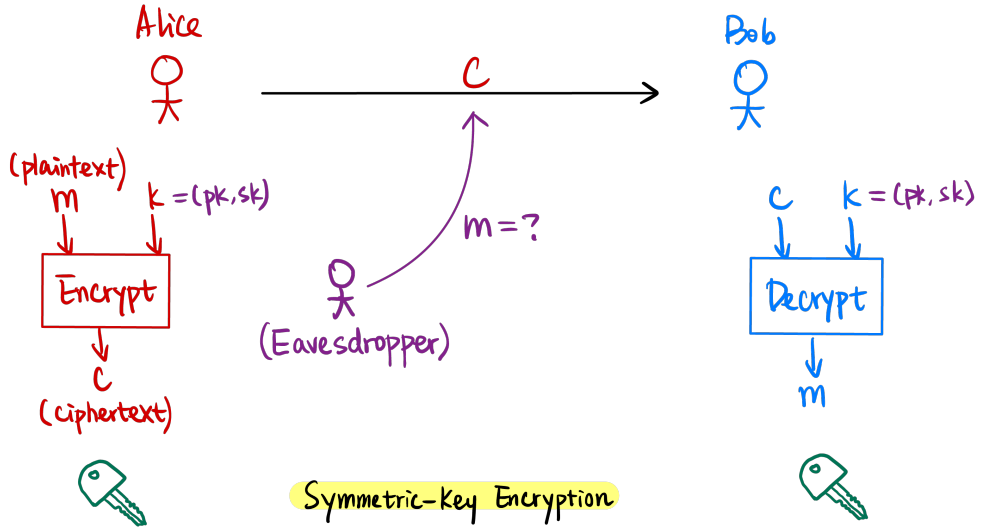
There's an EdStem post asking about topics you're interested, feel free to keep on posting!

We acknowledge the synchronization issues with Panopto. For now, if you want to watch the lecture recordings, you can use the Zoom link linked from the course home page. We can manually sync up EdStem but Panopto cannot be synced up, unfortunately.

§2.2 Encryption Schemes

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.

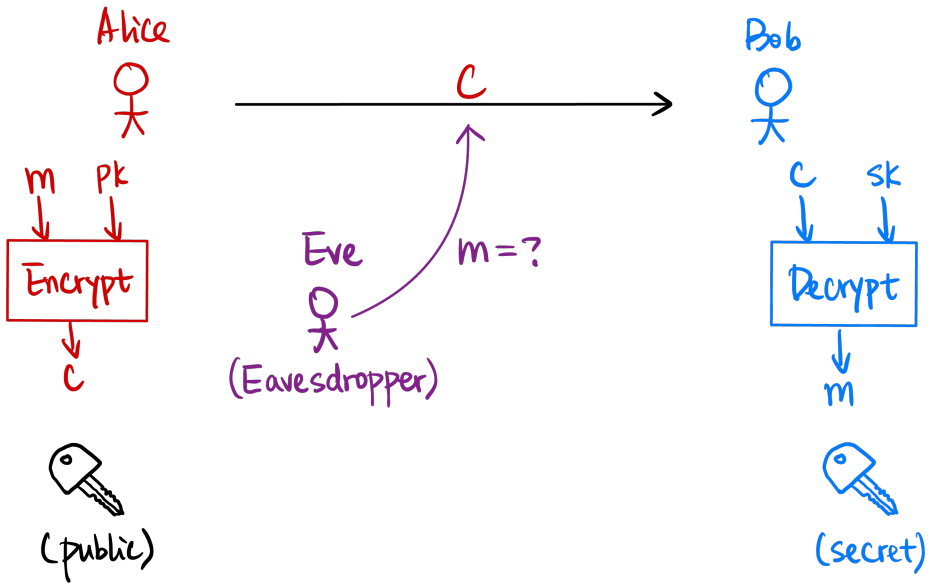
Fundamentally, an encryption scheme protects message secrecy. If Alice wants to communicate to Bob, Alice will encrypt a message (plaintext) using some key which gives her a ciphertext. Sending the ciphertext through Bob using a public channel, Bob can use the key to decrypt the ciphertext and recover 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.



§2.2.1 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), but we had better want decryption to be deterministic in producing the message.

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 $\mathsf{Dec}_{sk}(c)$.

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.

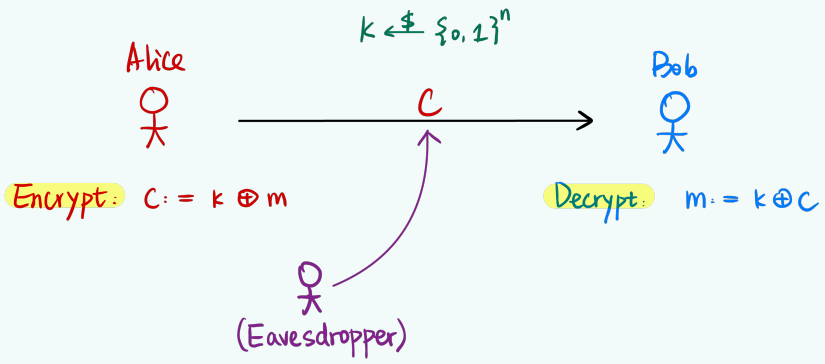
1. First of all, public-key encryption is almost always *more expensive*. Symmetric-key encryption schemes give us efficiency.

2. Public-key encryption relies on much stronger computational assumptions.

§2.2.2 Symmetric-Key Encryption Schemes

Definition 2.3 (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. Or, an element is its own inverse with the XOR operator.

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 2.4

Informally, for perfect (information-theoretic[6]) 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 2.5 (Computational Security)

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

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

Definition 2.6 (Polynomial-Time Algorithm)

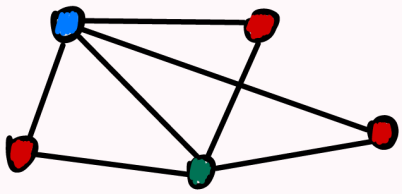
A polynomial time algorithm $A(x)$ is one that takes input x of length n, A's running time is $O(n^c)$ for a constant c.

Definition 2.7 (NP Problem)

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

Example 2.8 (Graph 3-Coloring)

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



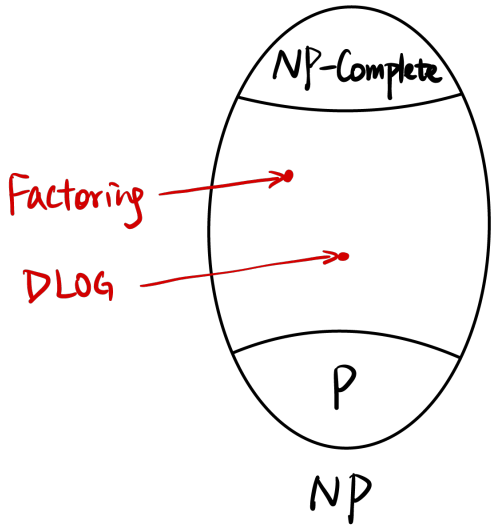
This can be *verified* in polynomial time (we can check if such a coloring is a valid 3-coloring), but it is computed in NP time.

Definition 2.9

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 2.5,

Definition (Computational Security)

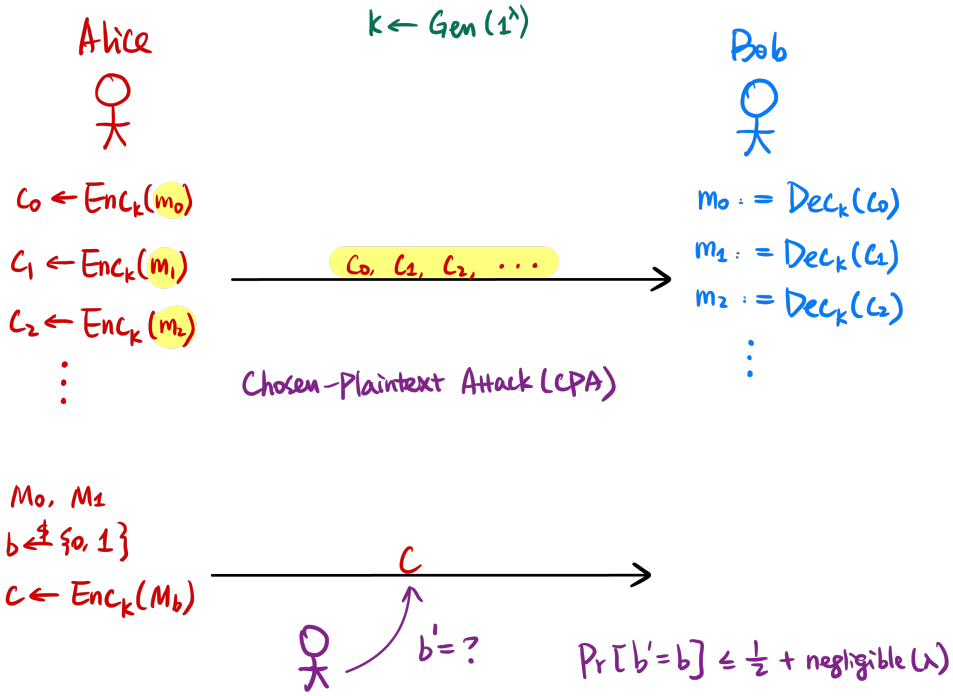
We say that the adversary is computationally bounded (is only a *polynomial-time algorithm*), that $\forall$probabilic poly-time algorithm $\mathcal{A}$,

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

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, $\Pr[b = b'] \leq \frac{1}{2} + \mathsf{negligible}(\lambda)$[7]. This is Chosen-Plaintext Attack (CPA) Security.

[7] 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.

Example 2.10

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[8]

It is a computational assumption[9] that the AES construction is secure, and the best attack is currently a brute-force search (in both classical and quantum computing realms).

§2.2.3 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[10].

Another thing worth mentioning is that

Theorem 2.11

(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 2.12

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

Definition 2.13

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

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

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

Example

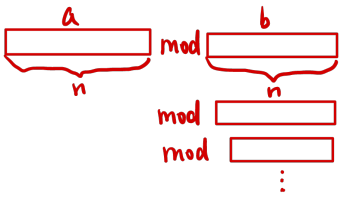
We use the Euclidean Algorithm. Take $\gcd(12, 17)$,

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

or take $\gcd(12, 18)$

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

If we have two bitstrings of length n bits, what is the running time of the Euclidan Algorithm?



Very informally, we see that every step, the length of a, b decrease by approximately 1 bit. Then, finding gcd is roughly order $O(n)$.

Definition 2.14 (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[11].

Theorem 2.15 (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 $1 \equiv a \cdot x \pmod{N}$.

[11] 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.

Definition 2.16 (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 2.17 (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 2.18 (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.

§2.2.4 RSA

We first define the RSA assumption.

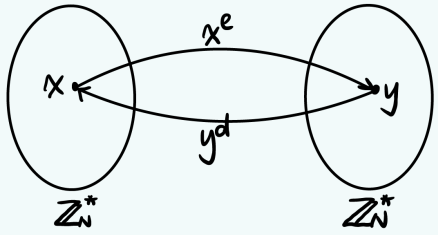
Definition 2.19 (Factoring Assumption)

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 2.20 (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 February 2, 2023

§3.1 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?
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. We can also guess! Since we're unsure whether coprime numbers are dense, we can pick small prime e.
3. We can compute d using the Extended Euclidean Algorithm.
4. We can repeatedly square (using fast power algorithm).

A note that we have correctness with this scheme: $(m^e)^d = m^{ed} = m \mod N$.

Question. Still, 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. Using RSA, you *really* have to be careful. 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. We really want to be sure that $m \xleftarrow{\$} \mathbb{Z}_N^\times$ (that it has enough entropy).

  Returning on the RSA assumption. It's crucial that the $y \xleftarrow{\$} \mathbb{Z}_N^\times$ is randomly sampled.

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.

> **Remark.** In practice, we usually set length of p and q to be 1024 bits, and the key length is 2048 bits. This is because of better algorithms for finding big primes.

Question. Asked on Ed: What happens if $p \mid m$ (or $q \mid m$)?

Correctness still holds[12].

However, security will be broken. If $p \mid (m^2 \mod N)$, then $gcd(c, N) = p$ will factor N.

However, this is *very* unlikely! Sampling $m \xleftarrow{\$} [1, N-1]$, then

$$\Pr[p \mid m] \equiv \frac{q}{N} = \frac{1}{p} = \frac{1}{2^{\theta(\lambda)}} = \mathsf{negligible}(\lambda)$$

When sending k messages,

$$\Pr[p \mid m \text{ for any } m] \leq \frac{k}{p} \equiv \frac{\mathsf{poly}(\lambda)}{2^{\theta(\lambda)}}$$

which is still negligible. Put simply, Alice has just as much chance to break RSA by randomly factoring as she is to send a message that is a multiple of p or q.

[12] Can do out, or by using Chinese Remainder Theorem to see that it preserves the qualities we need mod p and mod q.

§3.2 Intro to Group Theory

Definition 3.1 (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. $\forall g, h \in \mathbb{G}$, $g \circ h = h \circ g$.

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

Example 3.2

$(\mathbb{Z}, +)$ 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.

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

Definition 3.3 (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 $p - 1$[13].

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

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 $\alpha \mid p - 1$.

§3.3 Computational Assumptions

We have a few assumptions we make called the Diffie-Hellman Assumptions, in order of **weakest to strongest**[14] 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 3.4 (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).

Definition 3.5 (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 3.6 (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.

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

§3.4 ElGamal Encryption

The ElGamal encryption scheme involves the following:

[13] 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.

[14] 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).

$\mathsf{Gen}(1^\lambda)$: 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 sample $y \xleftarrow{\$} \mathbb{Z}_q$. 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.

§3.5 Secure Key Exchange

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

Definition 3.7 (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.

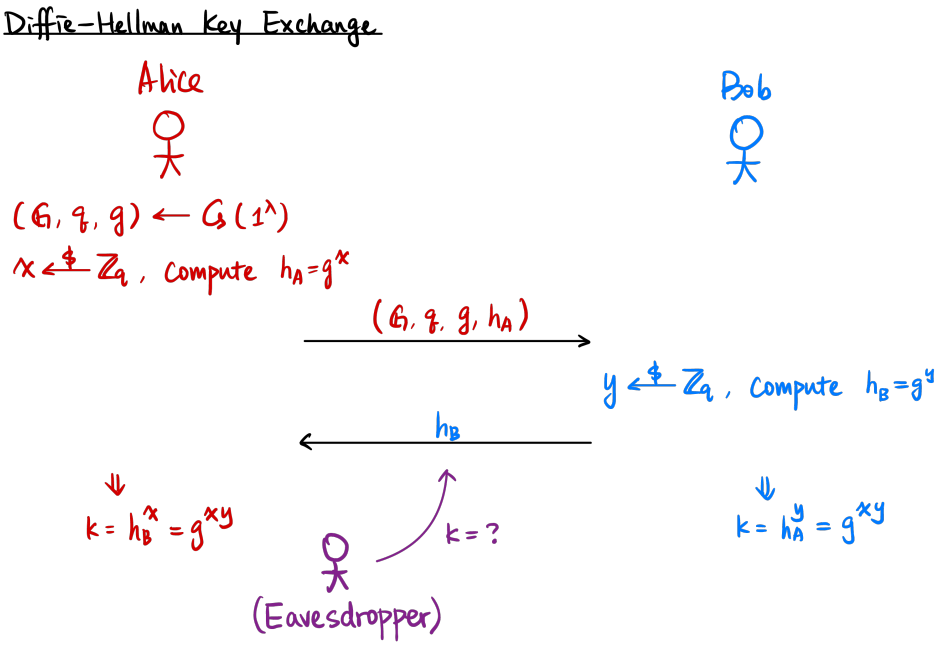
Theorem 3.8

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.

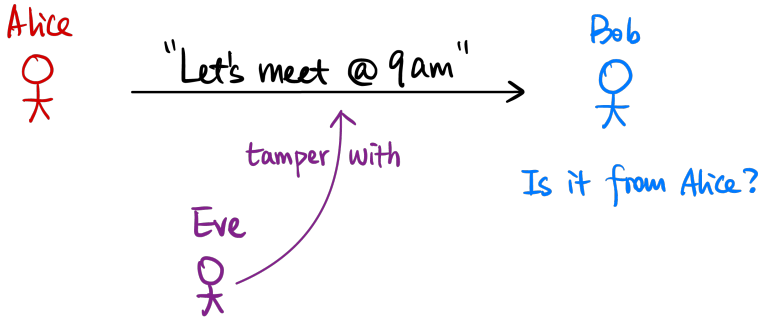
Using Diffie-Hellman, it's very easy. We have group $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$. Alice samples $x \xleftarrow{\$} \mathbb{Z}_q$ and sends g^x. Bob also samples $y \xleftarrow{\$} \mathbb{Z}_q$ and sends g^y. Both Alice and Bob compute $g^{xy} = (g^x)^y = (g^y)^x$.



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).

§3.6 Message Integrity

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



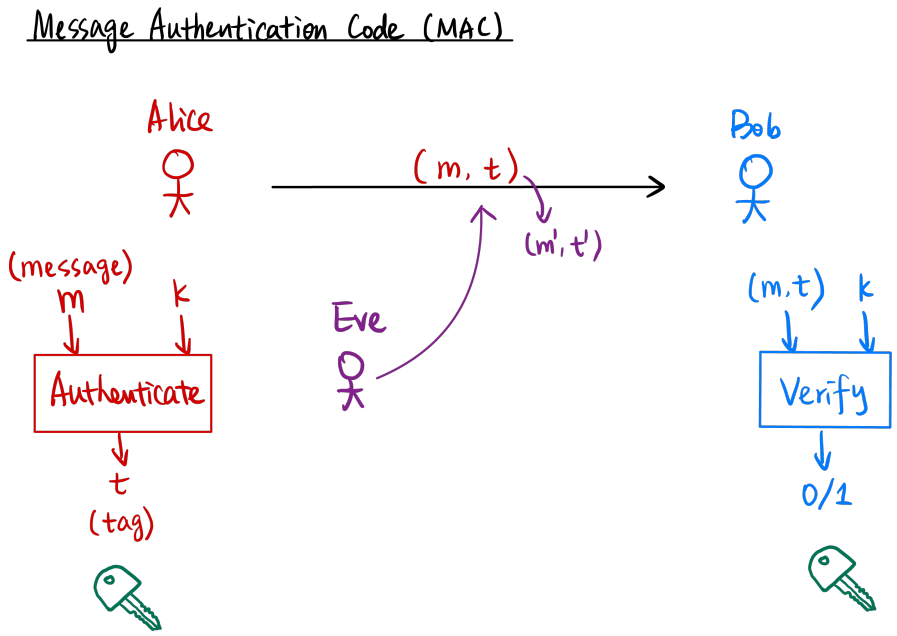
We can build up another line of protocols to ensure message integrity. It's similar to encryption, but the parties run 2 algorithms: *Authenticate* and *Verify*.

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

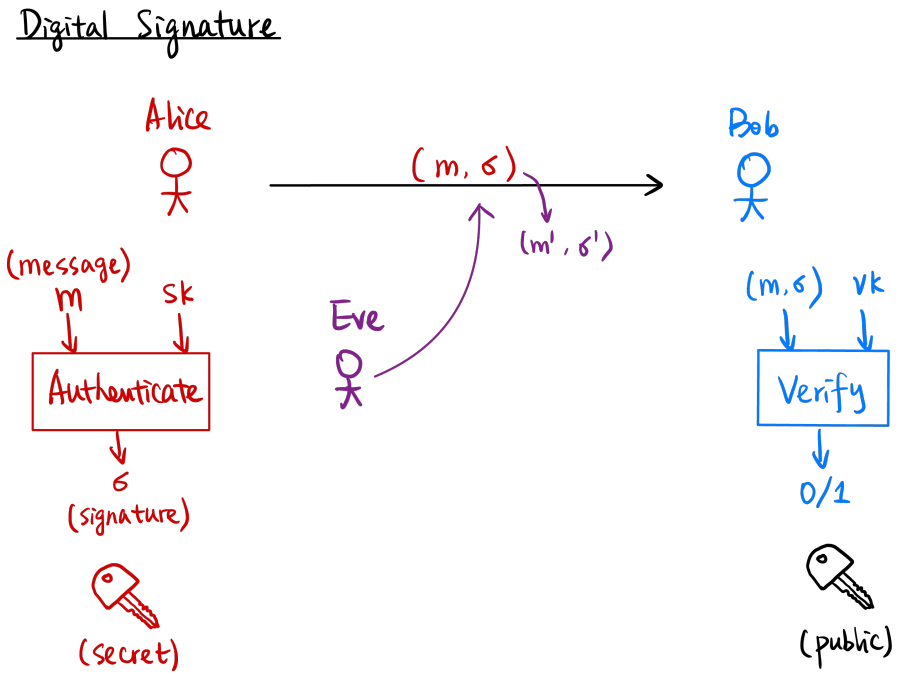
Our adversary has been upgraded to an Eve who can now tamper with messages.

Similarly to encryption, we have symmetric-key and public-key encryption.

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.



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

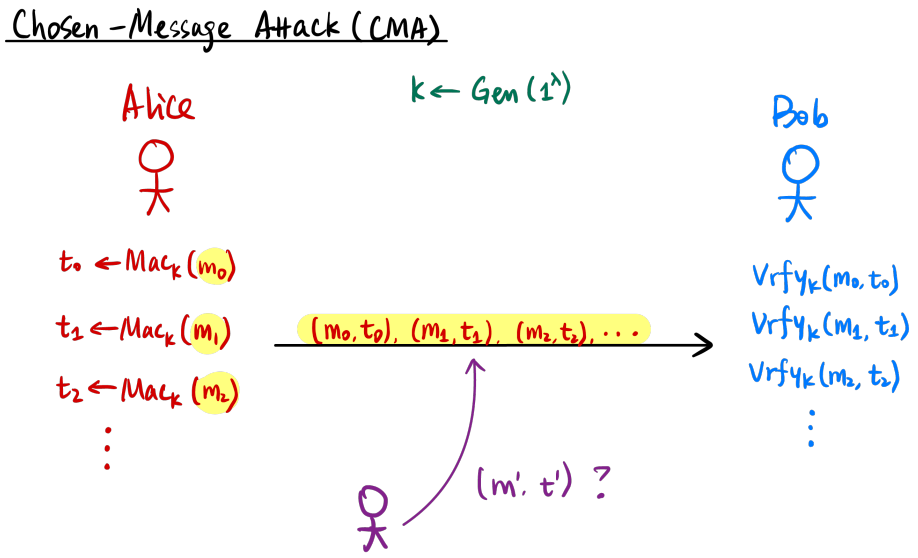
Authentication. $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.

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

§3.6.2 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 hard.



§3.6.3 Constructions

Very briefly, we discuss constructions for MAC and digital signatures.

Using block ciphers, we have CBC-MAC. Using a hash function, we have HMAC.

For digital signatures, we have RSA which relies on the RSA assumption, or DSA which relies on discrete-log algorithms. There are also lattice signature schemes for post-quantum digital signatures.

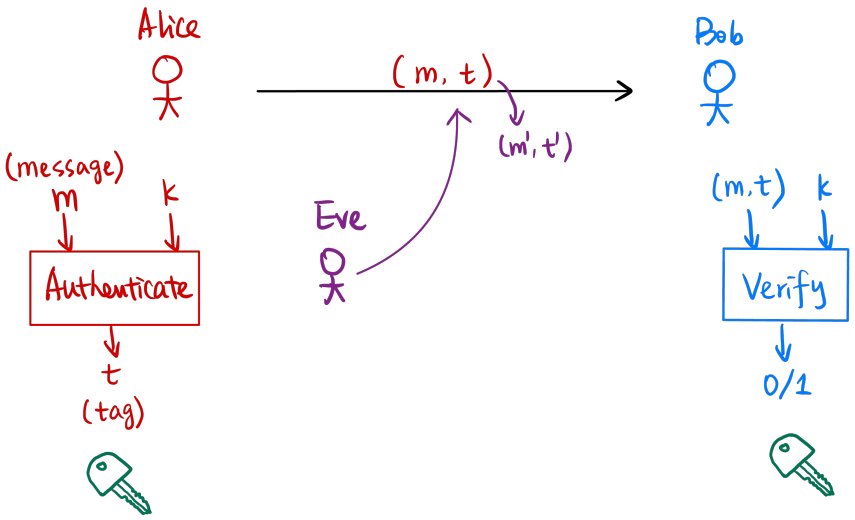
§4 February 7, 2023

§4.1 Message Integrity, *reviewed*

Last lecture, we discussed 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.

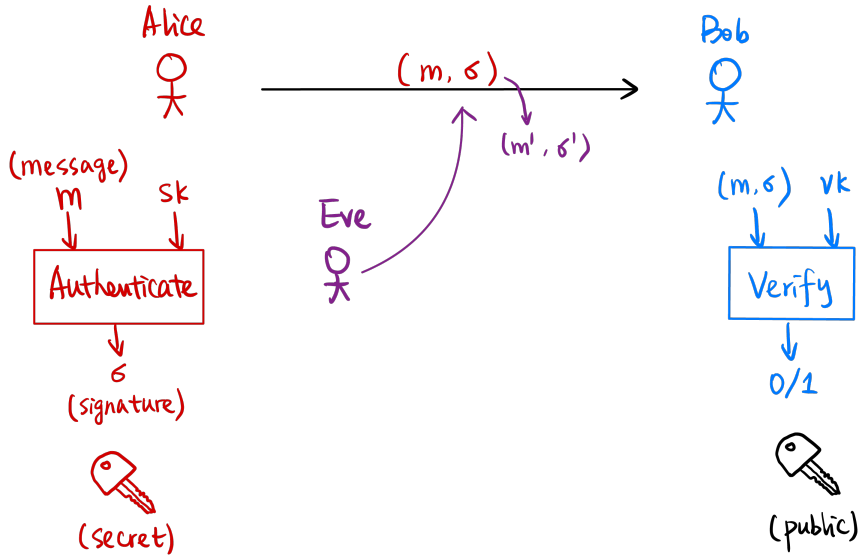
§4.1.1 Message Authentication Code

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

In 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 3.6.1).

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 Constructions

We can construct MAC pratically using

- Block Cipher: CBC-MAC
- Hash Functions: HMAC

We can construct digital signatures using

- RSA signature: RSA Assumption.
- DSA signature: Discrete Log Assumption
- Lattice-Based Encryption Schemes (post-quantum secure).

§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?

Thinking back to our definition of chosen-message attack, 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[15]. This is a commonly known technique called 'hash-and-sign'.

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

§4.3 DSA Signatures

DSA signatures are a bit more involved; they rely on the discrete log assumption and that it is hard to compute discrete logs in $\mathbb{Z}_p^\times$ and $\mathbb{Z}_q^\times$ a subgroup.

We give this definition using integer groups. There is also ECDSA which uses elliptic curve groups, which is used in cryptocurrencies[16]. Elliptic curves are much more efficient (especially when chosen correctly) and provide similar security guarantees.

$\mathsf{Gen}(1^k)$: We generate an n-bit[17] prime p and an m-bit[18] prime q such that $q \mid (p-1)$. We pick a random $\alpha \in \mathbb{Z}_p^\times$ such that $g = \alpha^{\frac{p-1}{q}} \mod p \neq 1$[19]

We pick $x \xleftarrow{\$} \mathbb{Z}_q^\times$ and compute $h = g^x \mod p$.

Our verification key is $vk := (p, q, g, h)$ and our signing key is $sk = x$.

$\mathsf{Sign}_{sk}(m)$: Sample $y \xleftarrow{\$} \mathbb{Z}_q^\times$, compute $r = (g^y \mod p) \mod q$. Compute $s = y^{-1} \cdot (H(m) + x \cdot r) \mod q$ and $\sigma := (r, s)$.

$\mathsf{Verify}_{vk}(m, \sigma)$: We can compute

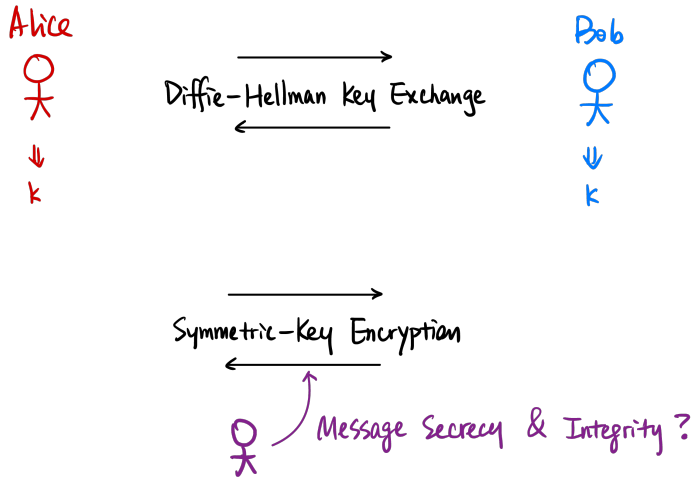
$$\begin{aligned} w &= s^{-1} \mod q \\ u_1 &= H(m) \cdot w \mod q \\ u_2 &= r \cdot w \mod q \end{aligned}$$

and verify $r \stackrel{?}{=} (g^{u_1} \cdot h^{u_2} \mod p) \mod q$.

§4.4 Authenticated Encryption

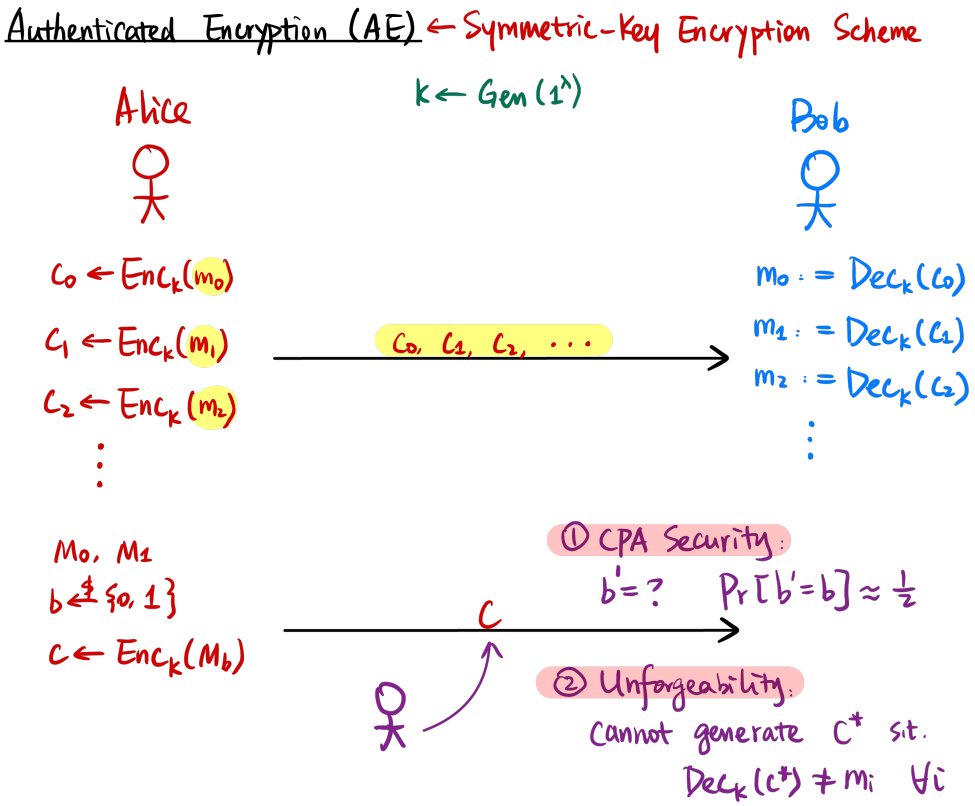
Generally, Alice and Bob will first perform a Diffie-Hellman key exchange, then use that shared key to conduct Symmetric-Key Encryption.

[16] The specific curve used by Bitcoin, for example, is called **secp256k1**.
[17] Usually, 2048.
[18] Usually, 256. We assume discrete log in *both* of these groups.
[19] This means that $\langle g \rangle$ has order q, a subgroup in $\mathbb{Z}_p^{-1}$. $g^q \equiv \alpha^{p-1} \equiv 1 \mod p$.



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 **CPA security**: given an encryption of either m_0, m_1, our adversary cannot distinguish between encryptions of the two. *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.

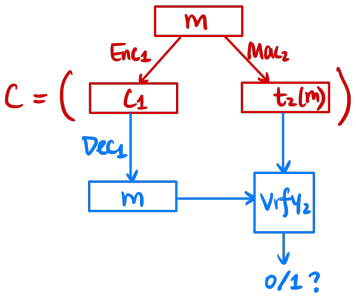


Now that we have two new primitives, we can construct Authenticated Encryption schemes.

§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?

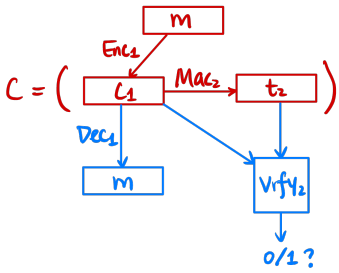
MAC gives you *unforgeability*—it doesn't even try to hide the message at all. It's possible that the MAC scheme reveals the message in the clear. For example, we might have a MAC scheme that includes the message in the signature *in the clear* (which is still secure)!

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 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).

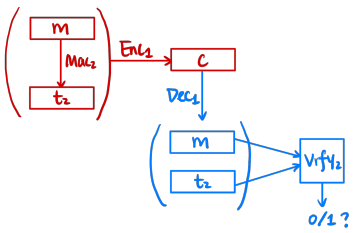
$\mathsf{Enc}(m)$: To encrypt, we first encrypt ciphertext $c_1 := \mathsf{Enc}_1(k_1, m)$ and sign message (in ciphertext) $t_2 := \mathsf{Mac}_2(k_2, c_1)$ and output (c_1, t_2).

$\mathsf{Dec}(m)$: 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.

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

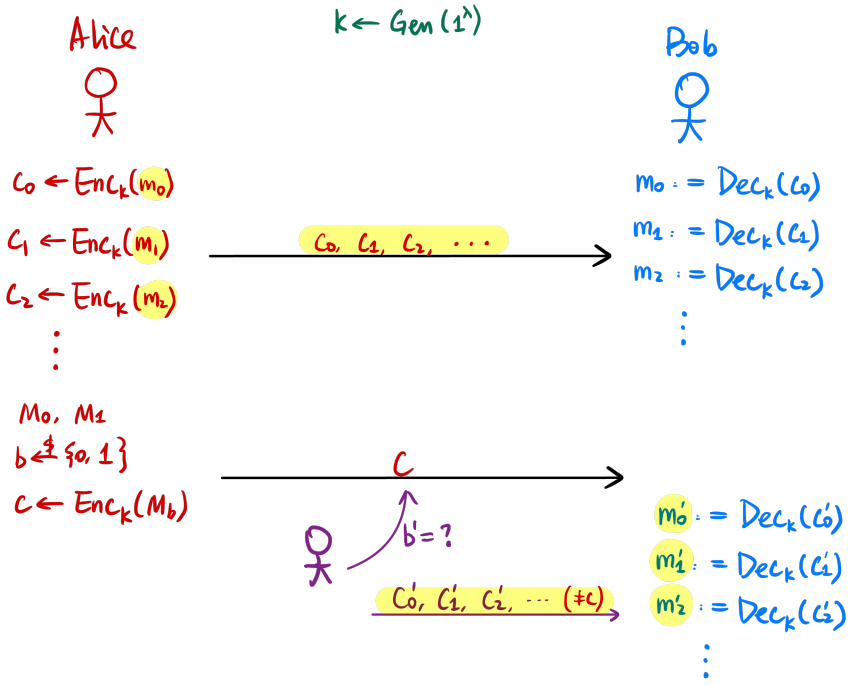
$\mathsf{Dec}(m)$: We have ciphertext $c = (c_1, t_2)$. Our message is $m||t_2 := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (m, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

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.

§4.4.4 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.

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

§4.5 A Summary So Far

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

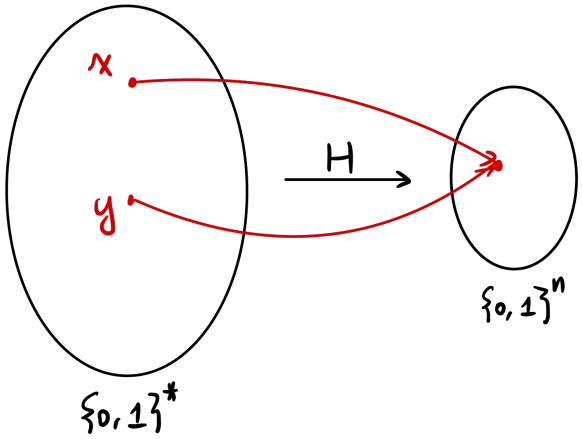
	Symmetric-KePy	**Public-Key**
Message Secrecy	Primitive: SKE Construction: Block Cipher	Primitive: PKE Constructions: RSA/ElGamal
Message Integrity	Primitive: MAC Constructions: CBC-MAC/HMAC	Primitive: Signature Constructions: RSA/DSA
Secrecy & Integrity	Primitive: AE Construction: Encrypt-then-MAC	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

§4.6 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).

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?

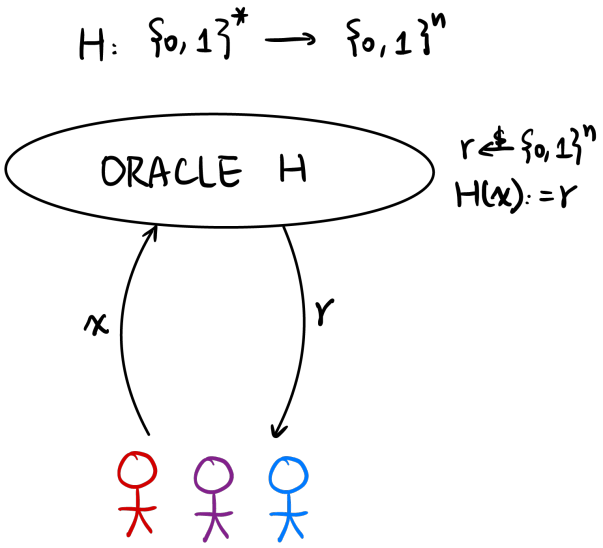
Remark. This is related to the birthday problem. If there are q students in a class, assume each student's birthday is a random day $y_i \xleftarrow{\$} [365]$. What is the probability of a collision? $q = 366$ gives 1, $q = 23$ gives around 50%, and $q = 70$ gives roughly 99.9%.

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.

§4.6.1 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[20].

In reality, hash functions are *about as good as*[21] 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.

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

§4.6.2 Constructions for Hash Function

MDS. 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.

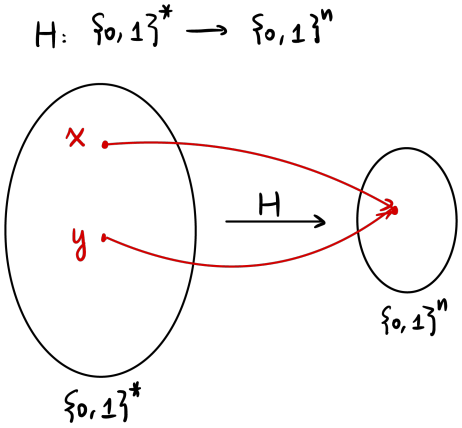
> **Remark.** The folklore is that during a session at a cryptography conference, a mathematician, Xiaoyun Wang, presented slide-after-slide of attacks on MDS and SHA-0, astounding the audience.

§5 February 9, 2023

Logistics: Please let us know if you have outstanding Ed or Gradescope issues!

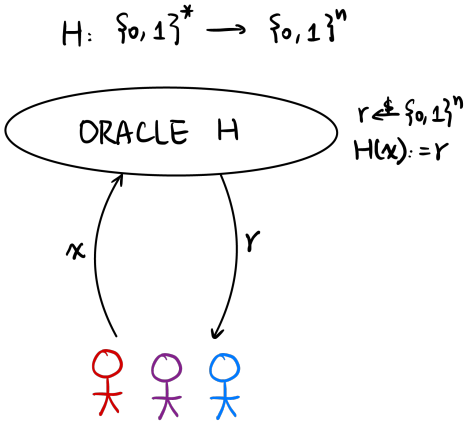
§5.1 Hash Functions, *continued*

Recall that we defined a hash function to be a 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.



One model that we sometimes use to facilitate our analysis is the random oracle model. We assume our hash function is a random oracle 'in the sky'that produces random hashes.

By guessing, analyzing it via the birthday problem, we require time approximately $\sqrt{n}$.



§5.1.1 Constructions

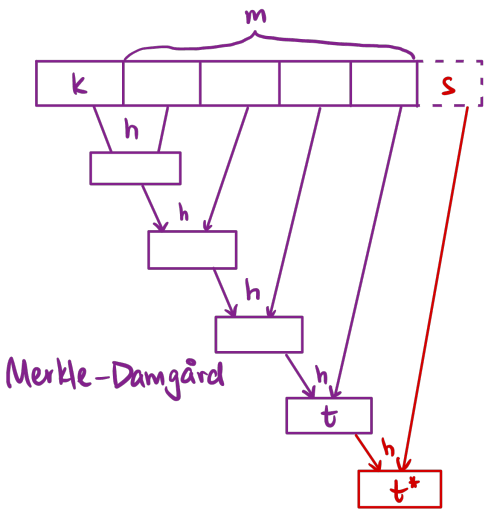
The hash function constructions that are still in practice (and unbroken) are SHA-2 and SHA-3.

§5.1.2 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.

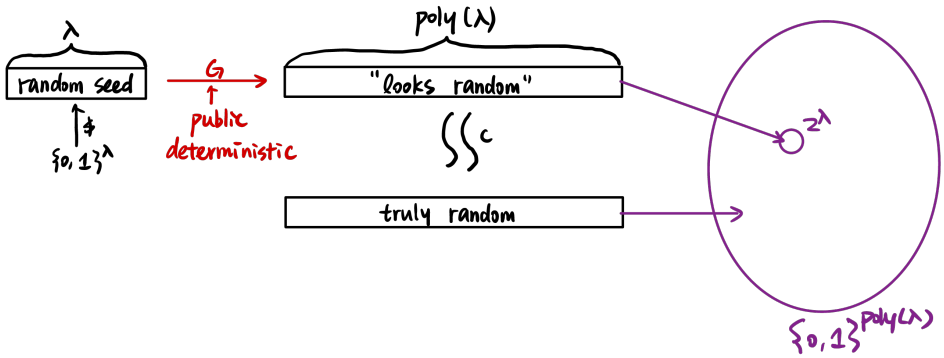
Deduplicate Files. We can also use hash functions to deduplicate files. We can hash two files to produce identifiers h_1 and h_2. If $h_1 \neq h_2$, this implies $D_1 \neq D_2$. If $h_1 = h_2$, it almost always[22] implies that $D_1 = D_2$.

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 adding a public deterministic salt G, we can generate a random[23] string.



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.

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

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.

§5.2 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 shraed key $k = (k_1, k_2)$ (one for AES one for HMAC). Then, they perform authenticated encryption, namely Encrypt-then-MAC.

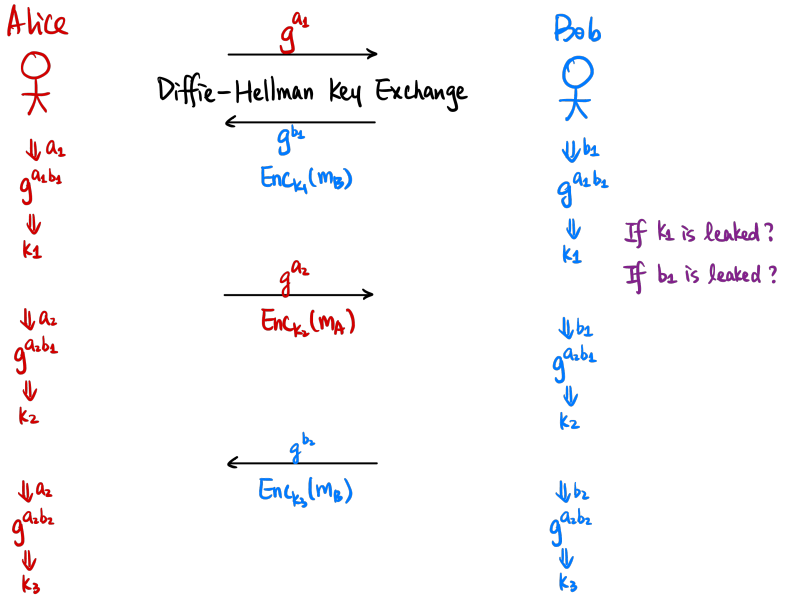
Question. Are there any issues with this scheme?

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.

§5.2.1 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.

§5.3 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.

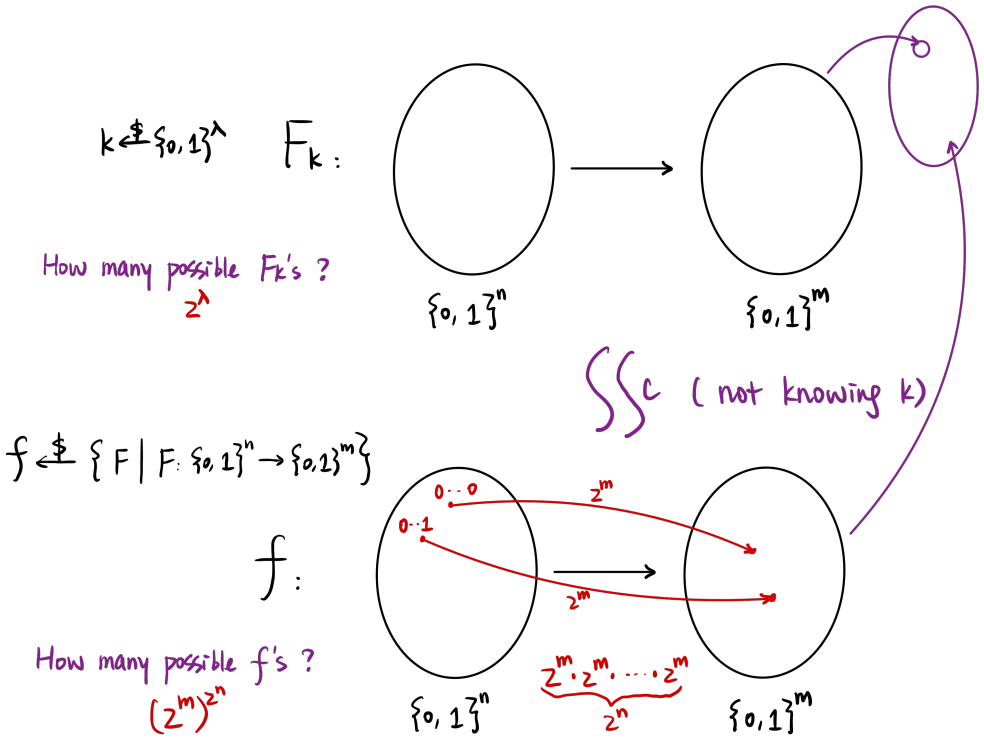
We saw earlier that a Pseudorandom Generator (PRG) produces a string that looks random. We also have Pseudorandom Functions (PRF), which are 'random-looking' functions.

§5.3.1 Pseudorandom Function (PRF)

Our Pseudorandom Function F is a keyed function[24] $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^m$, 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 $f \xleftarrow{\$} \{F \mid F\{0,1\}^n \to \{0,1\}^m\}$.

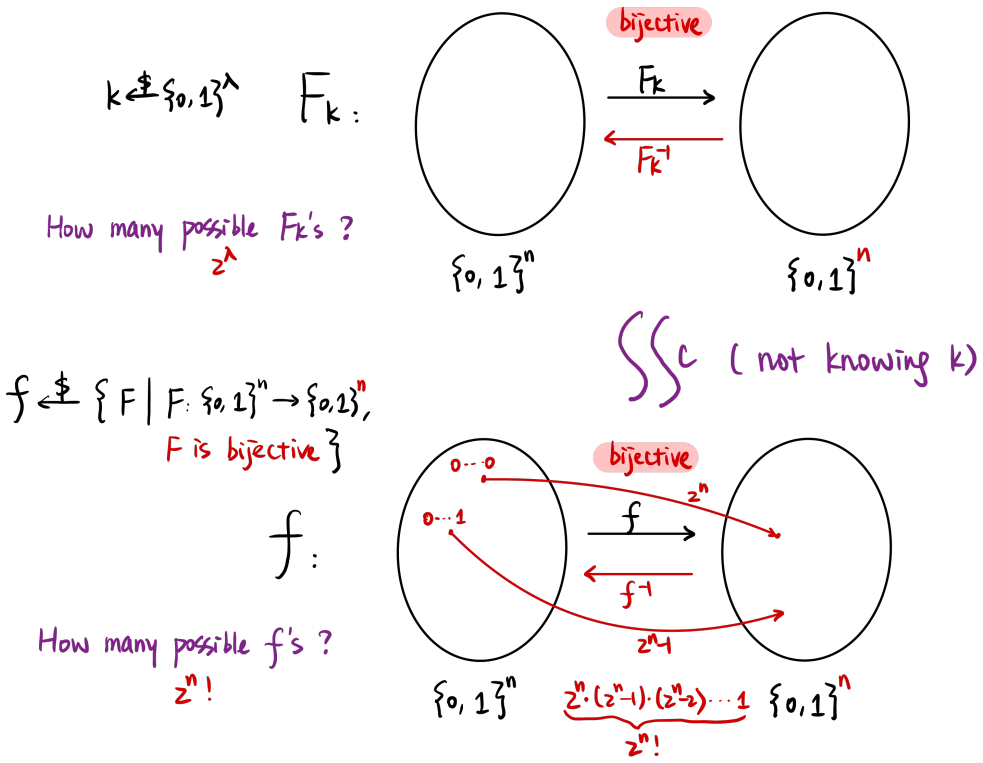
[24] In deterministic polynomial-time.



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.

§5.3.2 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.

§5.3.3 Block Cipher Definition

A block cipher is a 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. A block cipher is assumed to be a pseudorandom permutation (PRP).

Our practical construction is the Advanced Encryption Standard (AES).

- Our key and block sizes are $\lambda = n = 128$.
- This was standardized by NIST in 2001.

- There was a competition from 1997-2000 to come up with a block cipher scheme. It is said that the end of the competition, competitors were simply trying to attack each other's schemes. AES was eventually selected for its efficiency.

- Before AES, we used the Data Encryption Standard (DES) with $\lambda = 56$ and $n = 64$. The best attack is *still* a brute-force search, but key and block lengths are relatively fixed (cannot be extended).

Currently, the best attack for AES is still a brute-force search, which takes time 2^{128}.

§5.3.4 Block Cipher Modes of Operation

We have block cipher

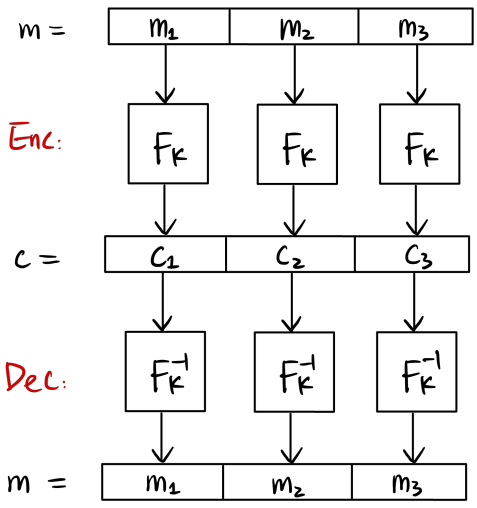
$$F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$$

We want to construct an SKE scheme from F for arbitrary-length messages. We have some $k \xleftarrow{\$} \{0,1\}^\lambda$. We encrypt with $\mathsf{Enc}_k(m)$ and decrypt $\mathsf{Dec}_k(c)$.

Our goal is to construct an SKE scheme that is CPA (Chosen Plaintext Attack) secure.

Electronic Code Book (ECB) Mode:

The easiest solution is to split up our message into blocks, and run our function F on each of those blocks.

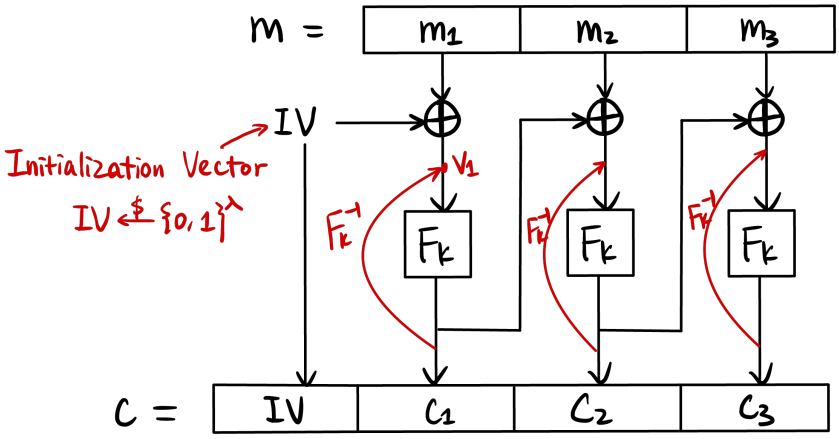


However, this is not CPA secure, since each block is deterministically computed.

Cipher Block Chaining (CBC) Mode:

We can do something else to ensure each block's plaintext is different using an Initialization Vector (IV), sampled from $\{0,1\}^\lambda$.

After every block, we XOR that block's c_i with the next block's message m_{i+1}.



Question. How do we decrypt this?

We can decrypt the first block, then XOR c_i with $F_k^{-1}(c_{i+1})$.

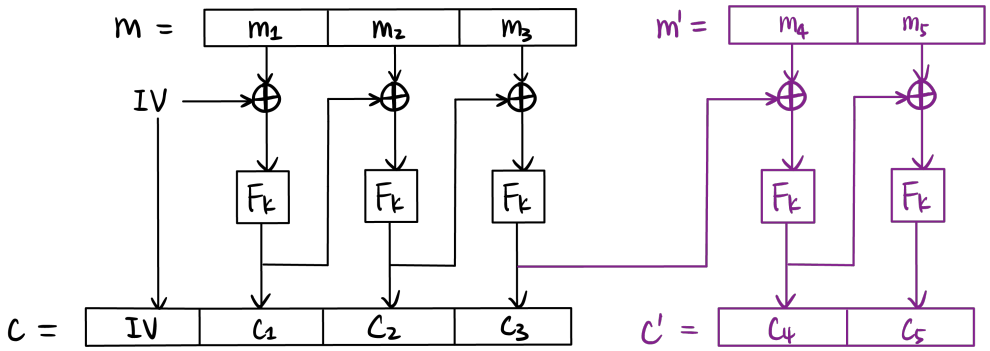
Question. Is this secure?

Assuming F_k is a valid pseudorandom permutation, this is. We'll elide the proof here.

Question. Can we parallelize this? Especially the F_k of F_k^{-1} steps?

We can't in the case of encryption. For decryption, we can perform F_k^{-1} all at once, and do all the XOR operations in series.

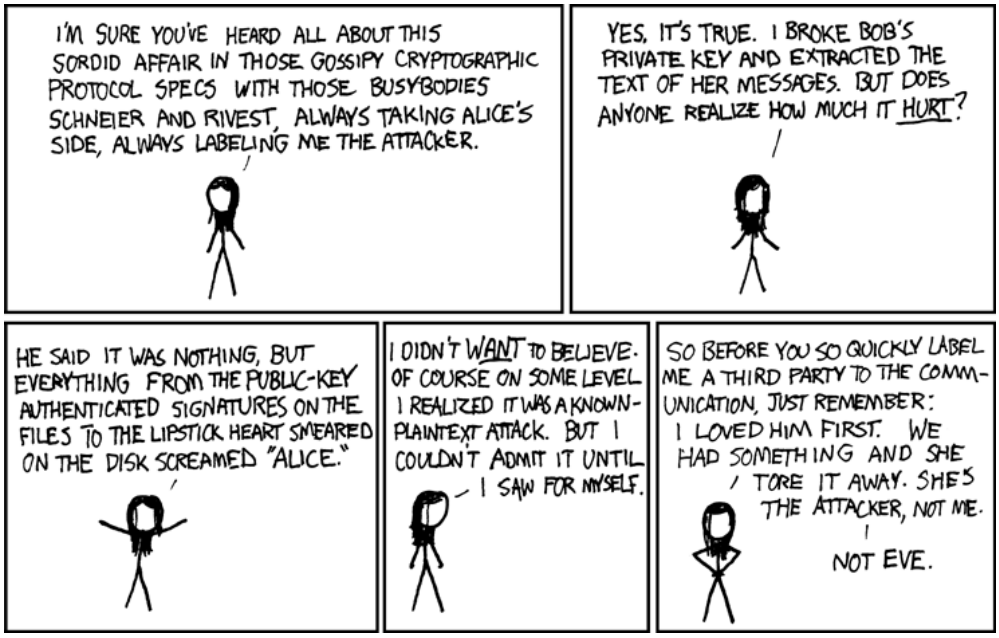
Chained Cipher Block Chaining (Chained-CBC) Mode: We *could* also use the previous messages' c values as the IV for future messages.



Question. Is this now CPA-secure?

We'll continue this next time...

§6 February 14, 2023



xkcd 177. Happy Valentine's Day! Stay safe by selecting authenticated encryption schemes (see section 4.4) that are CPA secure and unforgeable to communicate with your date.

§6.1 Block Ciphers, *continued*

Looking back on section 4.5, 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

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

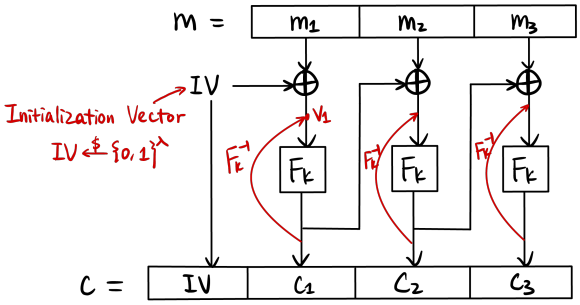
where λ is the key length and n is the block length. The practical construction of which is AES, which takes blocks of $n = 128$ and key length $\lambda = 128, 192, 256$ as choices.

§6.1.1 Modes of Operation

You can see section 5.3.4 for a recap as well.

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.

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.



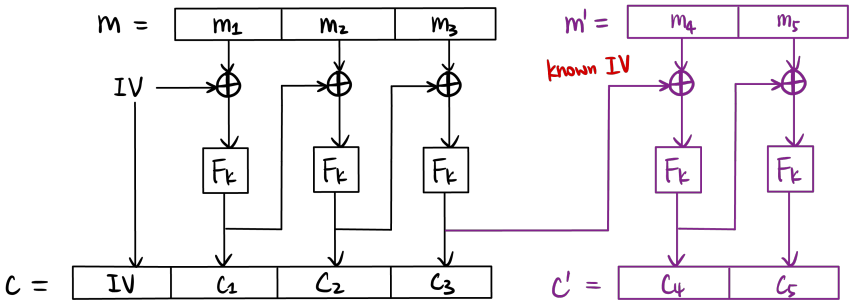
We waved our hand over the fact that this is CPA secure—but it relies on the initialization vector being random.

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).

Chained Cipher Block Chaining (Chained-CBC) Mode: We touched on this earlier, but 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.

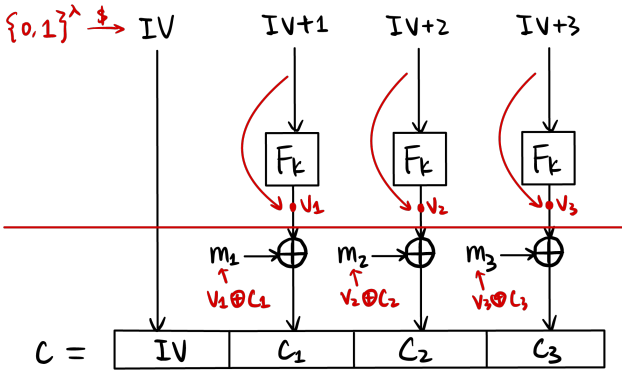
This makes chained-CBC *very subtly* different than CBC. If we squint our eyes enough, it just looks like sending a single message using CBC mode. The key difference is that between rounds of communication m and m', an adversary could influence m' given the knowledge of the previous round.

> **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!

The following will be new modes not covered last lecture:

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

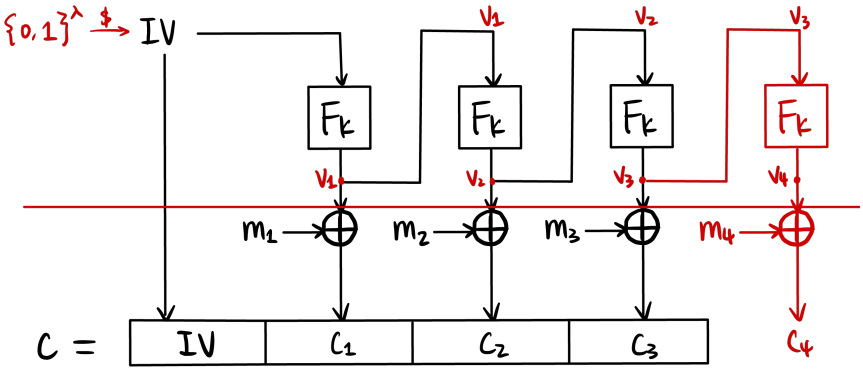
$$G(k||\text{IV}) = F_k(\text{IV})||F_k(\text{IV}+1)||F_k(\text{IV}+2)||\cdots$$

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.

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.

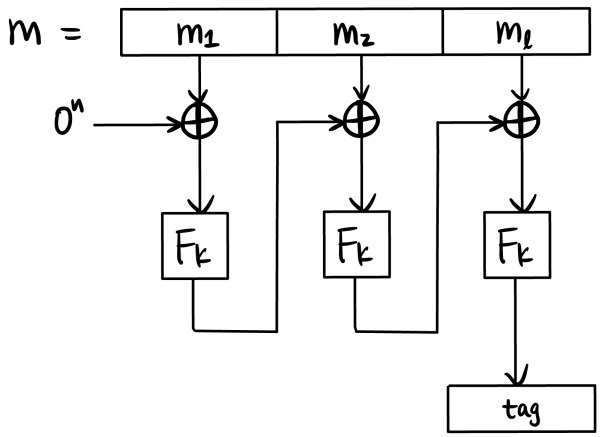
Question. We've listed *a lot* of benefits to counter mode or output feedback mode. Why do people use CBC mode at all?

We've seen how things can go wrong catastrophically[25]. 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[26]. 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[27].

§6.1.2 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.



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

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.

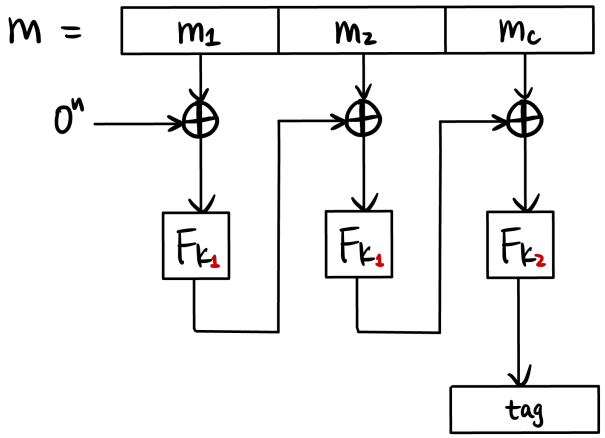
> **Remark.** Our constructions of authenticated encryption calls for an encryption scheme and MAC scheme. It's crucial that the two schemes have *different keys*. Using the same key k for both encryption and MAC can cuase issues (information from one could reveal something about the other).

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

§6.1.3 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:



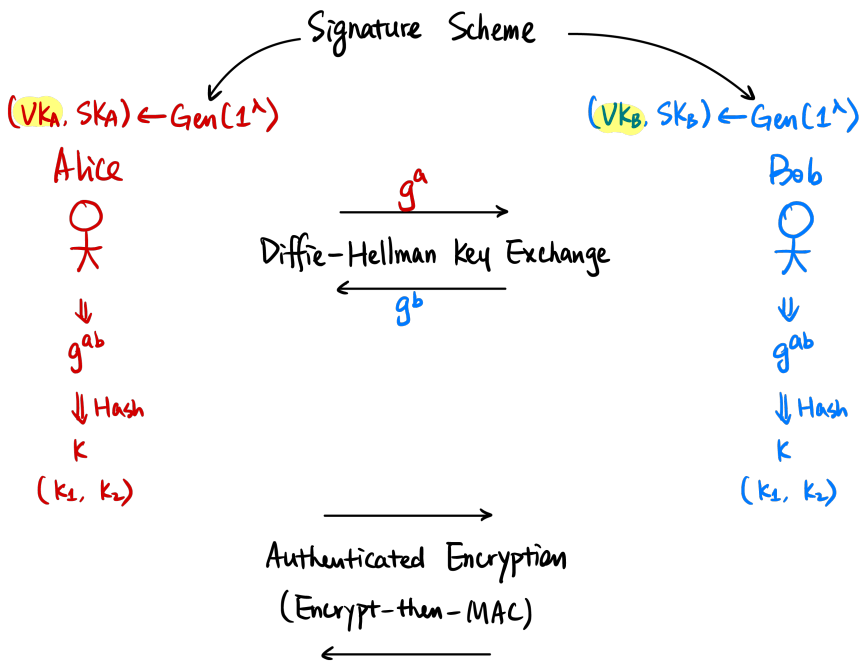
We could also attach length of messages to the first block, or other techniques.

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

§6.2 Putting it Together

Looking back at section 4.5, 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.

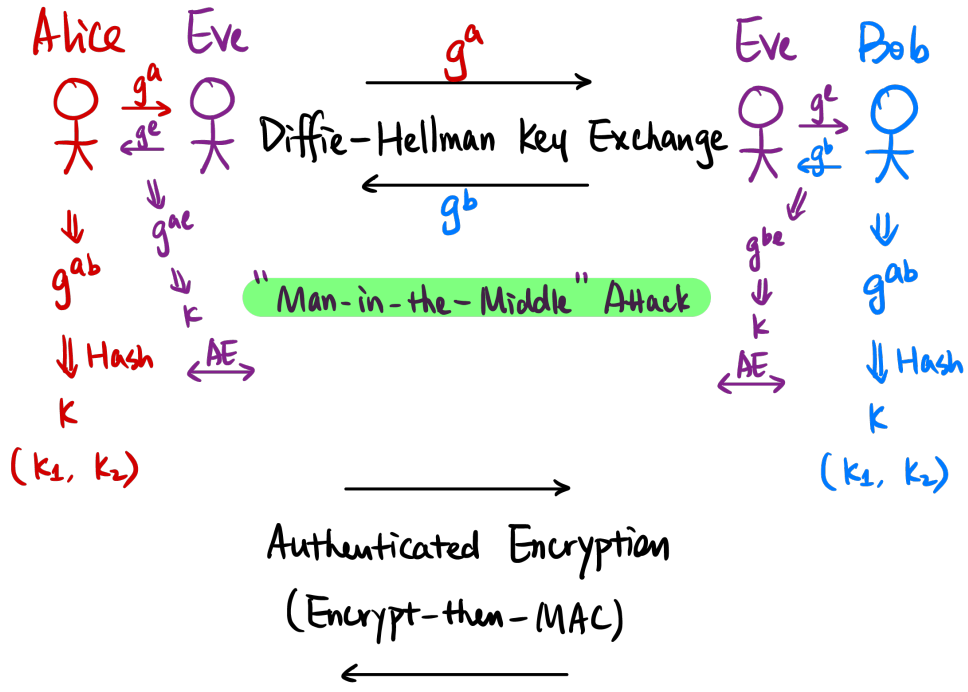


However, this still does not mitigate against a man-in-the-middle attack. Thus, before exchanging keys, Alice and Bob should publish verification keys (to a digital signature scheme, see section 4.1.2). Using this digital signature, Alice and Bob will each sign their Diffie-Hellman public values g^a, 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.

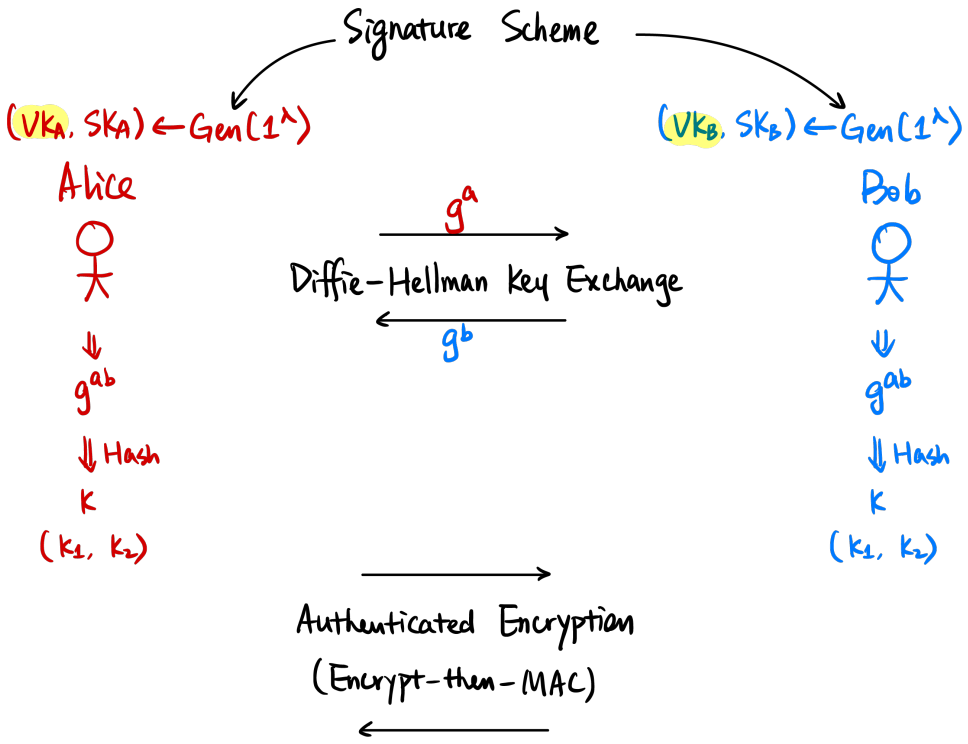
§7 February 16, 2023

We've now learned all the cryptographic primitives we need in this course.

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.



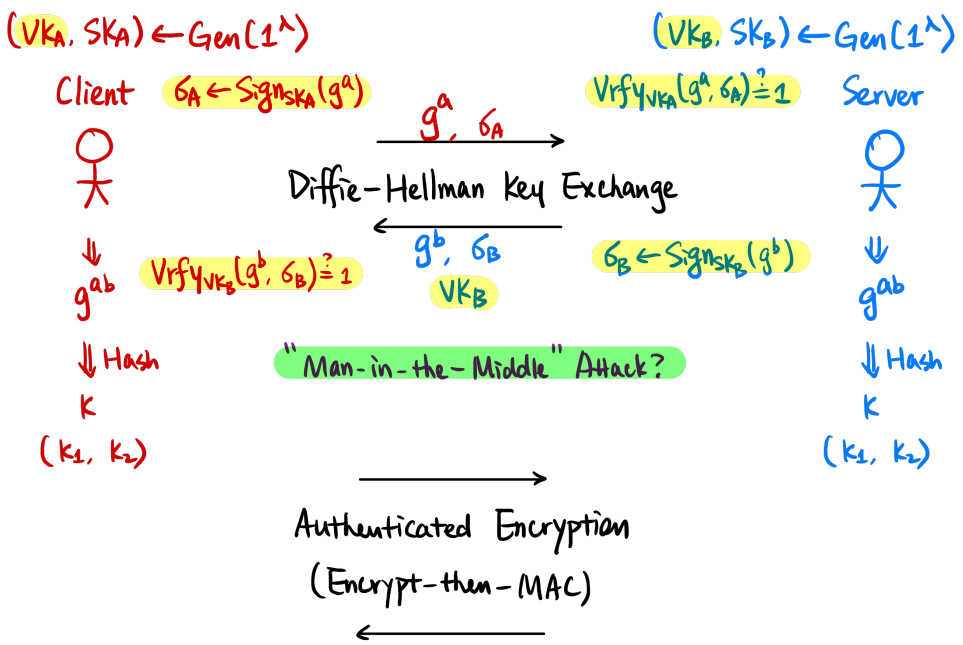
However, this is prone to a man-in-the-middle attack. One way to solve this is for parties to *sign* their Diffie-Hellman publics, and verify using a publicly known verification key.



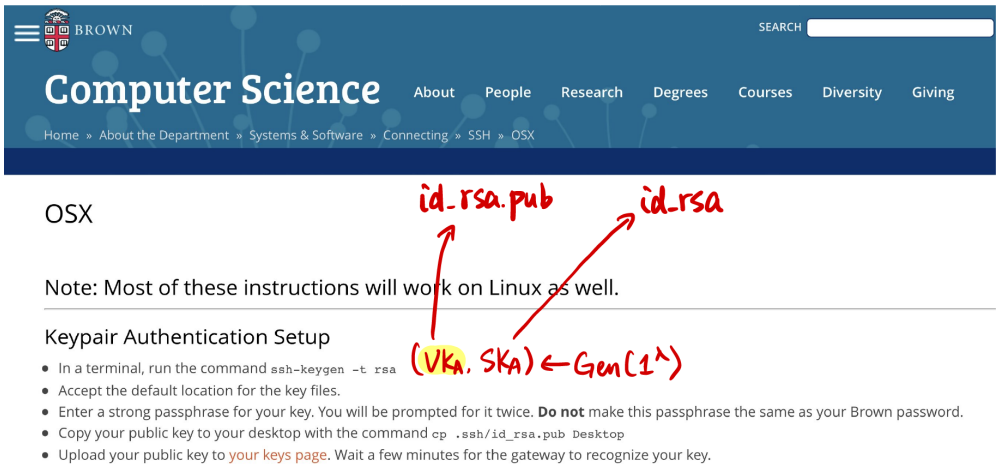
Is this now secure against an adversary in the middle? Yes, because the public values are guaranteed (via our digital signature scheme) by Alice and Bob's signing key. The man in the middle does not have access to the signing key, and cannot sign a phony public value.

§7.1 SSH

This is *exactly* how the SSH algorithm works.



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.

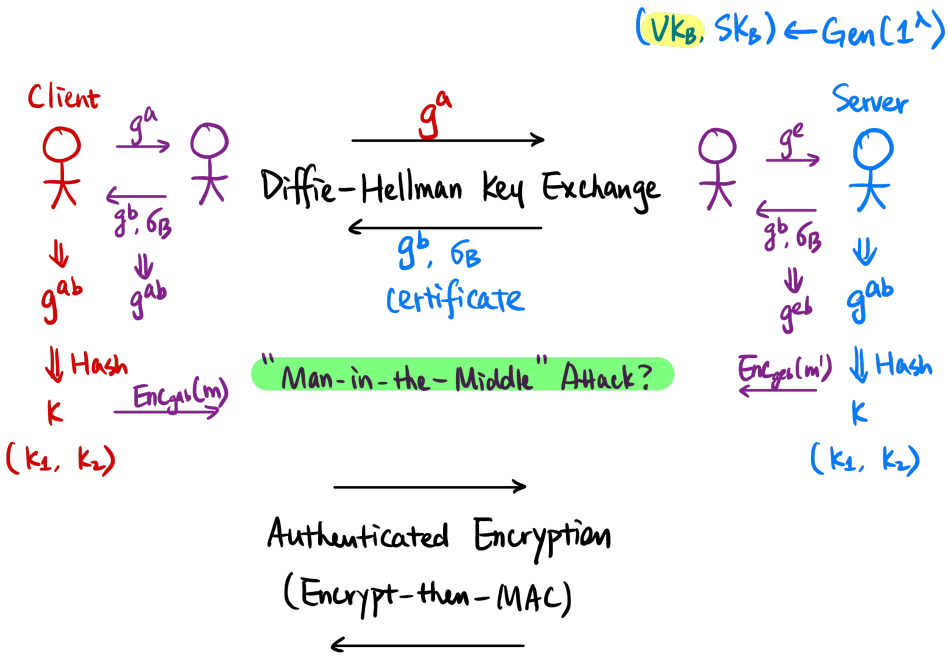
3. When we're connecting via SSH to GitHub for the first time, our terminal will prompt us that this is a new server with a new verification key.

```
> 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[28]. This is the equivalent of receiving a vk_B from GitHub.

§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

[28] 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.

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.

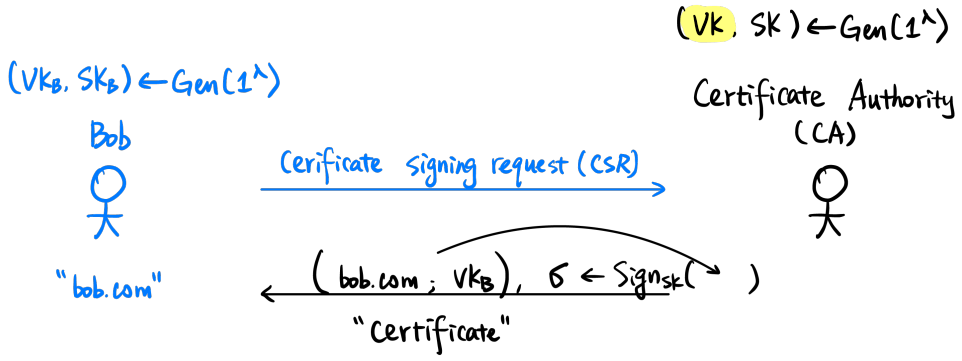
This can be easily circumvented by requiring the server and user complete their handshake—the server could request a hash or encryption of the shared secret, and realize that they are communicating to an adversary when this cannot be forged by the man-in-the-middle.

§7.3 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)[29].

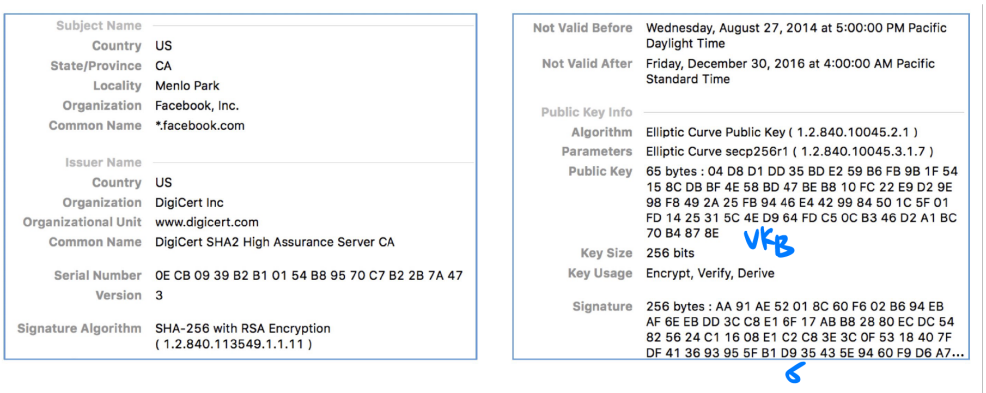
The CA will sign the message $(\texttt{bob.com}, vk_B)$ and send that signature σ back to Bob. This verifies that the user of `bob.com` holds signing key sk_B with public key vk_B.



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[30]

[29] *The higher beings that be*...this is companies like *DigiCert*, *Let's Encrypt*, etc.
[30] In browsers, this is represented by the lock symbol—clicking on that will allow you to verify that certificate.

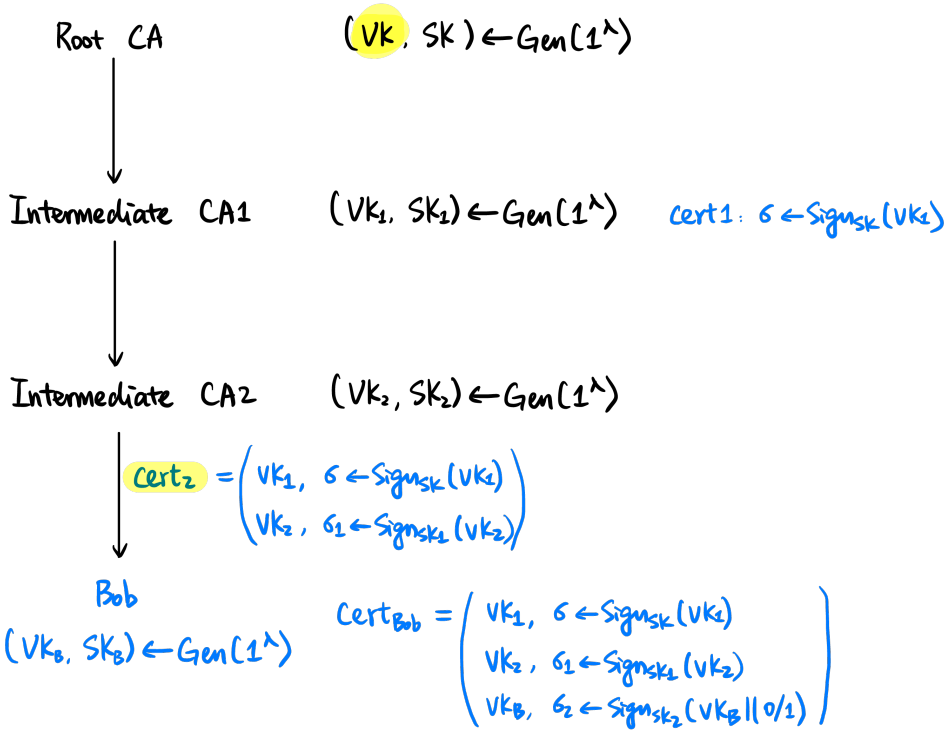


An example X.509 certificate

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.

§7.3.1 Certificate Chain

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



A Root CA[31] 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

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

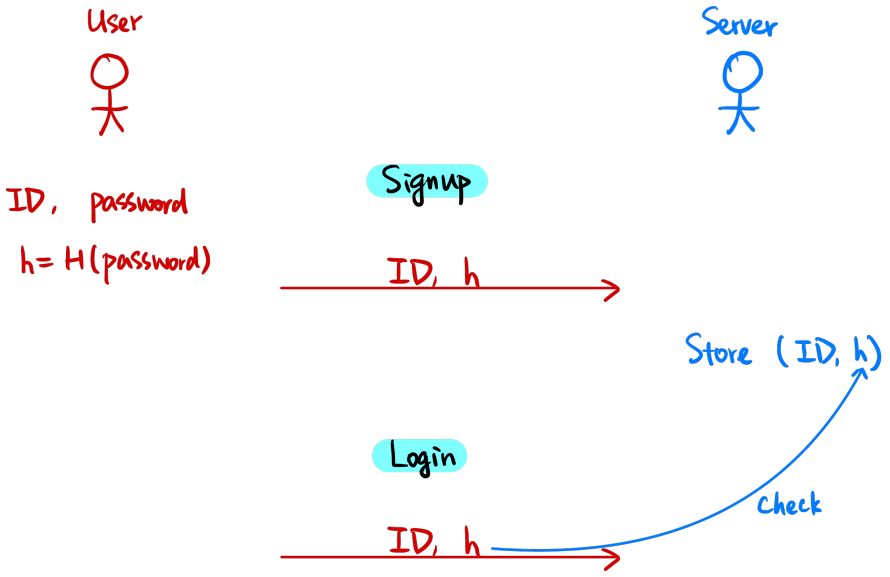
How can an Intermediate CA restrict Bob's use of these certificates? What if Bob will then go on and start signing his own certificates for people? We can concatenate information in each certificate that restricts its use. It could specify whether it is being issued to an *end user*, or even additional 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.

§7.4 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).

[31] 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.

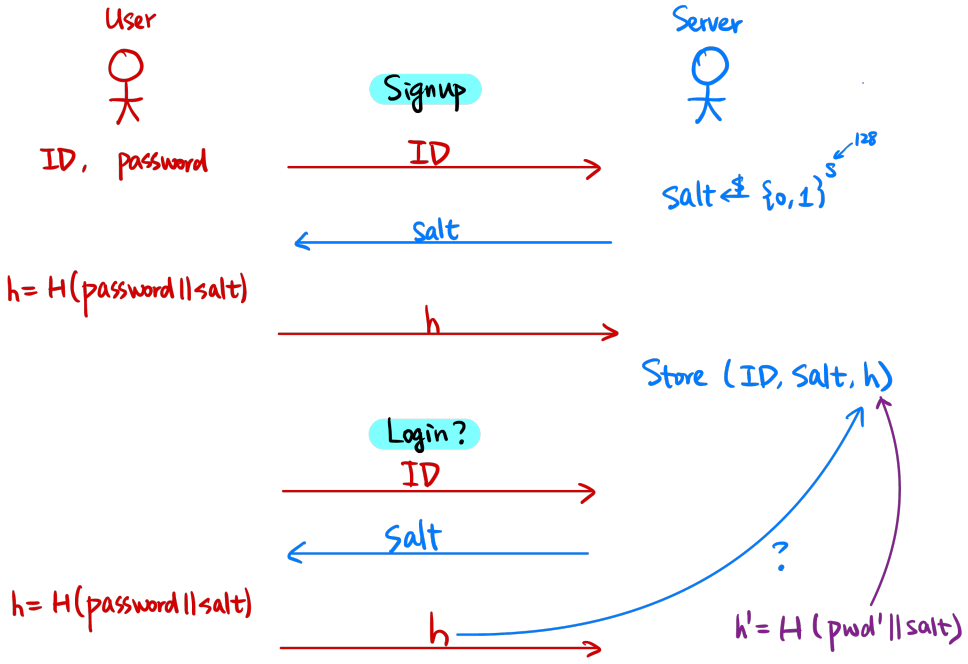


In this case, an adversary could launch an *Online Dictionary Attack* and try a lot of passwords with the server.

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?

§7.4.1 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.

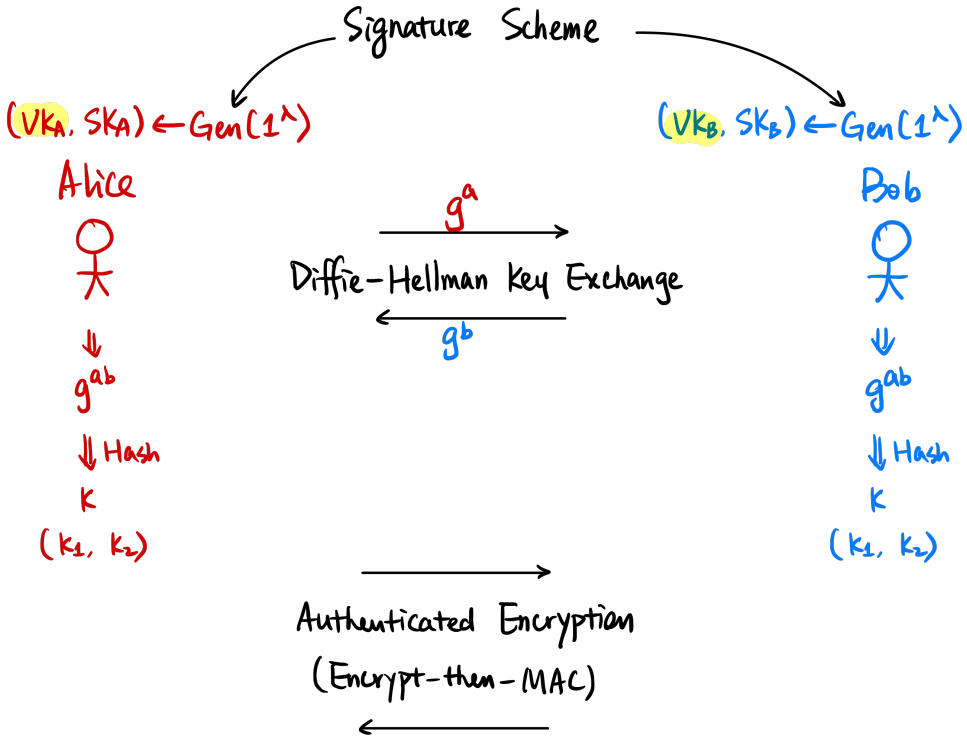
However, there are still issues with this scheme...we'll discuss another technique next time, *peppering*, that will make the adversary's life even harder.

§8 February 23, 2023

§8.1 Review

We'll quickly review what we covered last lecture.

Recall that we had a problem with authenticated encryption earlier—that we cannot prevent man-in-the-middle attacks. We require that we have public keys and sign our Diffie-Hellman Key Exchange step.



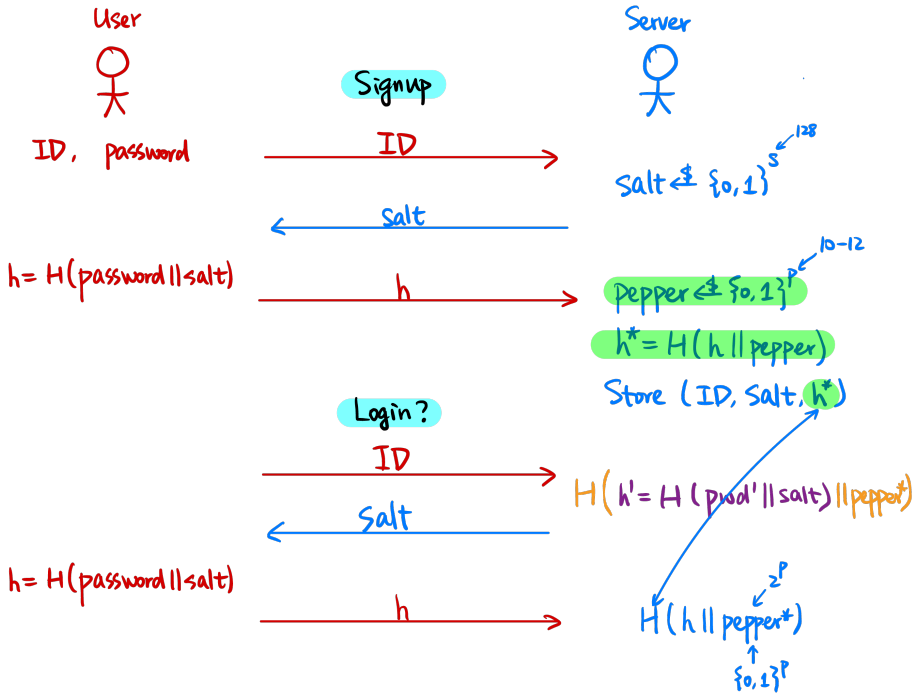
We solve this by using certificates and certificate authorities. We assume there is a known verification key for a Root CA (built into computers), and certificate authorities sign down a chain to verify an individual user.

This is most commonly used in the context of one-sided secure authentication. This is how TLS/HTTPS works in practice.

§8.2 Password Authentication, *continued*

We also discussed how a user might authenticate themselves with a server. The user will send the hash of a password to the server, and the server will verify against this hash.

We saw two attacks, an online dictionary attack and an offline dictionary attack. An online dictionary attack will query known hashes against the server. This can be prevented by rate limiting. An offline dictionary attack happens when the entire database of passwords is leaked—and an adversary can test known passwords.[32]



To solve these problems, we can use *salting* and *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[33]. 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.

[32] Once the database is leaked, they can already authenticate themselves with this server. However, the plaintext password is *more* important, since users reuse passwords. Additionally, a server will usually notice that their database has been compromised, and prompt users to re-enter passwords.

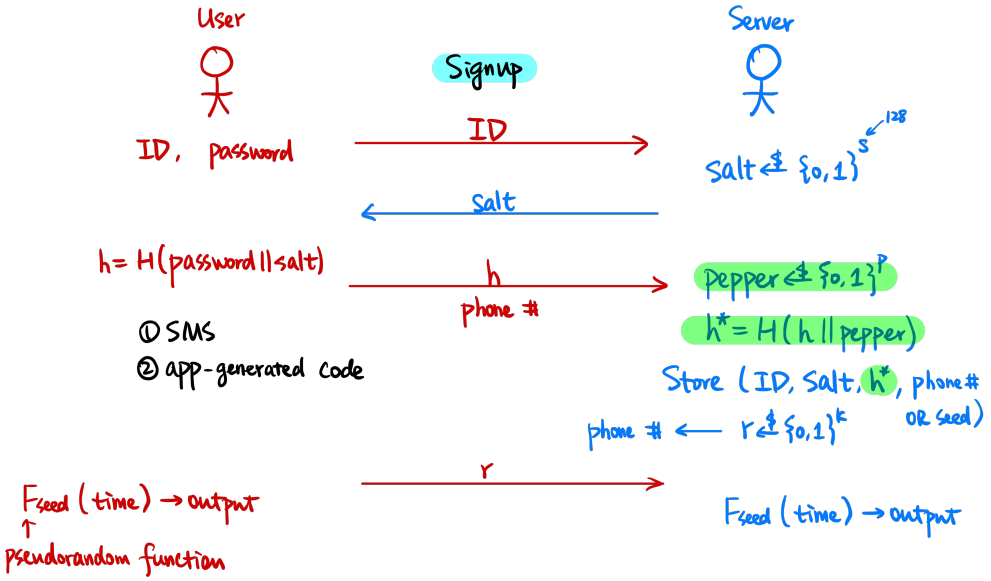
[33] The natural way is to hash multiple times, say 100. However, this is actually not more secure in the case of SHA256 but there are specific ways of composition. For example, there are application-specific integrated circuits (ASIC) that can compute hash functions very efficiently.

§8.2.1 Two-Factor Authentication

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

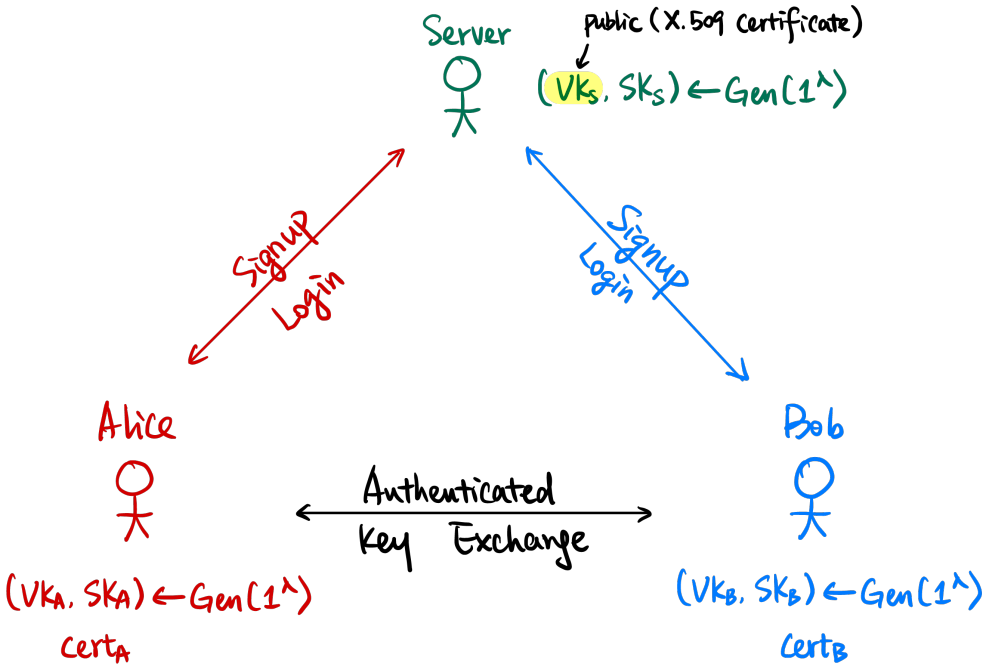
For phone number verification, on signing up, the user sends a phone number with their password hash. The server stores their phone number. Every time, the server will generate challenge $r \xleftarrow{\$} \{0,1\}^k$

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.



§8.3 Putting it Together: Secure Authentication

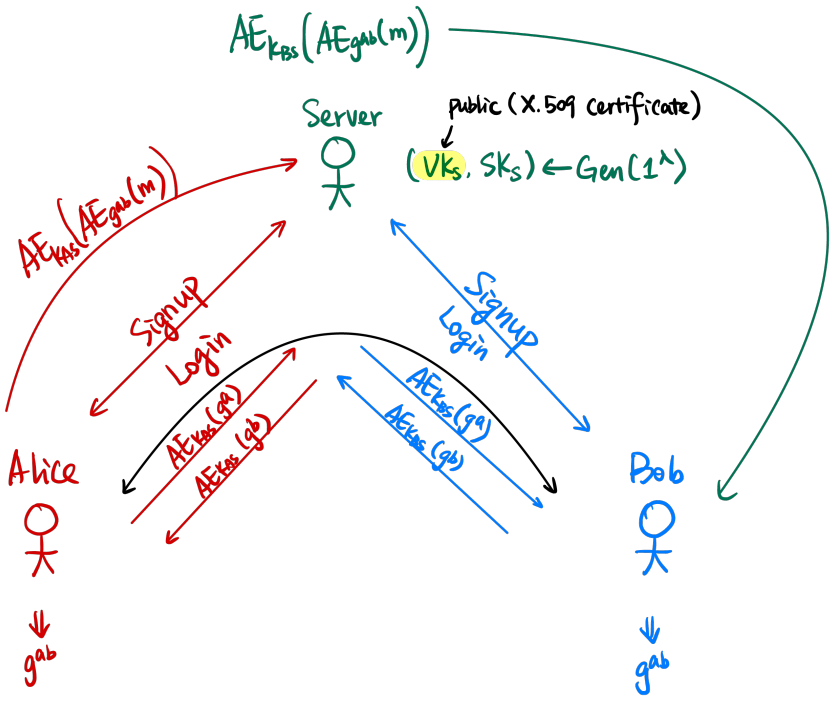
Recall that we had an issue from last project that the protocol was vulnerable to man-in-the-middle attacks. Now, we can have a server with a known verification key authenticate users who are communicating between each other.



Now, the server can authenticate users, say Alice and Bob. *How might Alice and Bob send messages amongst each other?*

§8.3.1 Secure Messaging

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.

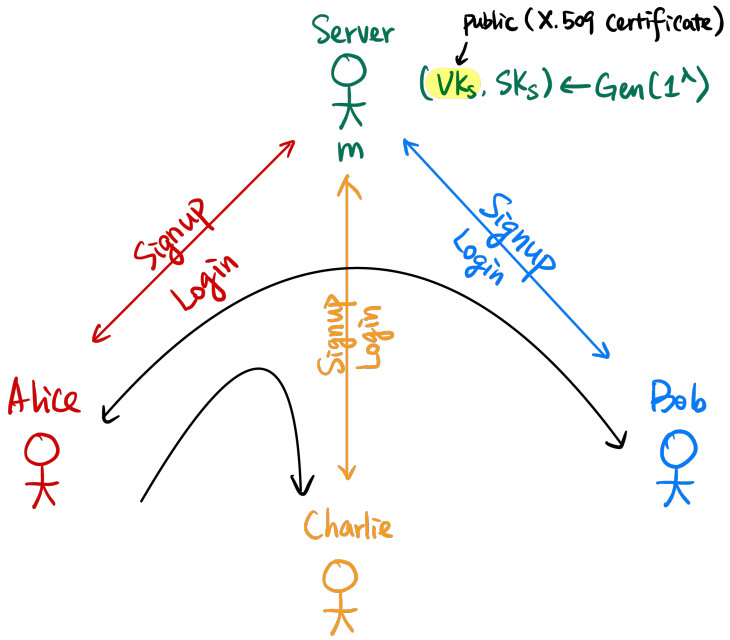
Alice will first encrypt using their shared key, then using their shared secret with the server, encrypt that ciphertext. The server will decrypt the first layer, encrypt that with Bob's key, and send that to Bob.

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.

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.

§8.3.2 Group Chats

What about group chats? How might we implement this.



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.

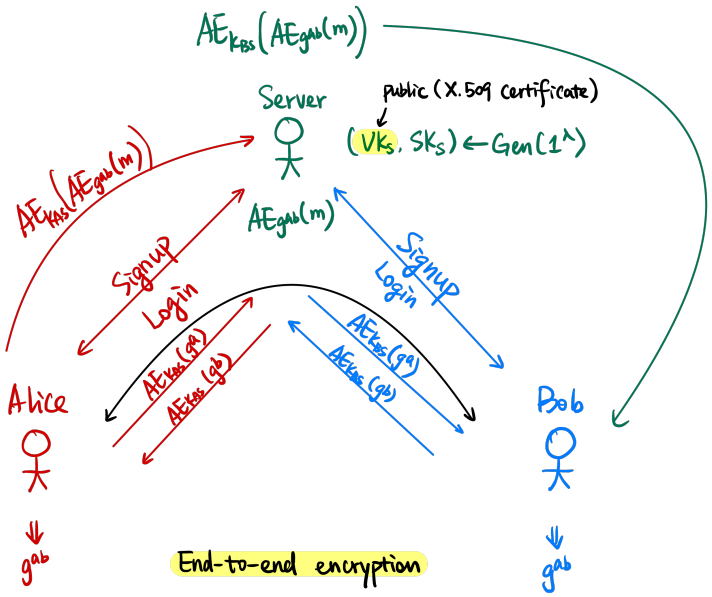
We might posit that Alice, Bob and Charlie share keys g^a, g^b, g^c, then they jointly have shared secret g^{abc}. This is called multi-party key exchange. However, this is in fact very difficult and relies on strong primitives.

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.

§9 February 28, 2023

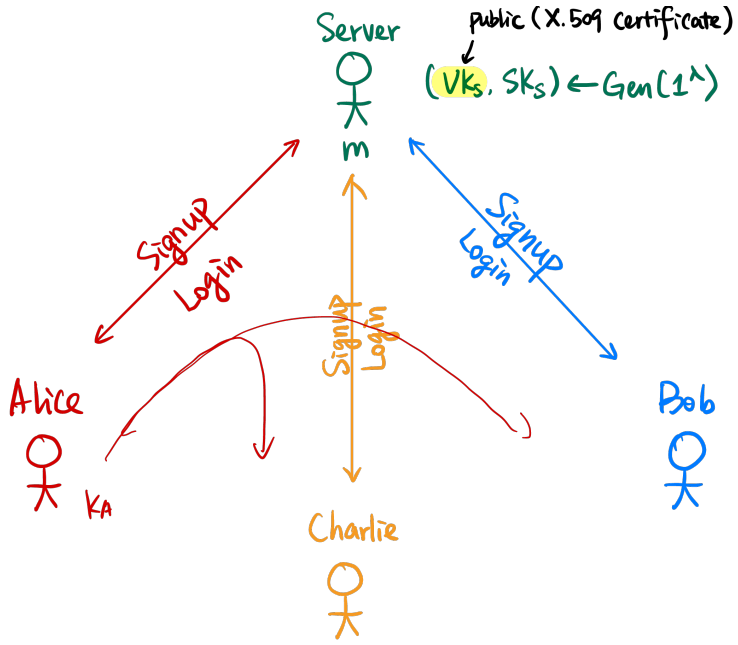
§9.1 Secure Messaging, *continued*

This was just a review of section 8.3.1. The server with a known verification key will sign public keys for Alice and Bob. Alice and Bob will exchange keys via the server and communicate messages via the server, encrypted with their shared secret. This allows the server to pass messages that remain secret to the server.



§9.1.1 Group Messaging

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.



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.

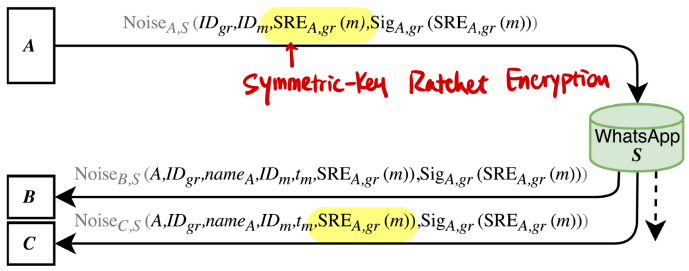


Figure 5. Schematic depiction of traffic, generated for a message m from sender A to receivers B, C in group gr with $\mathcal{G}_{gr} = \{A, B, C\}$ in WhatsApp.

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.

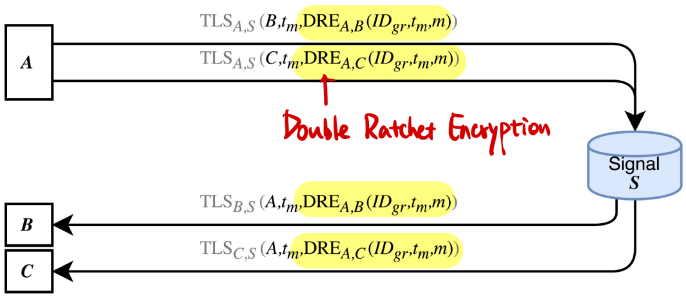
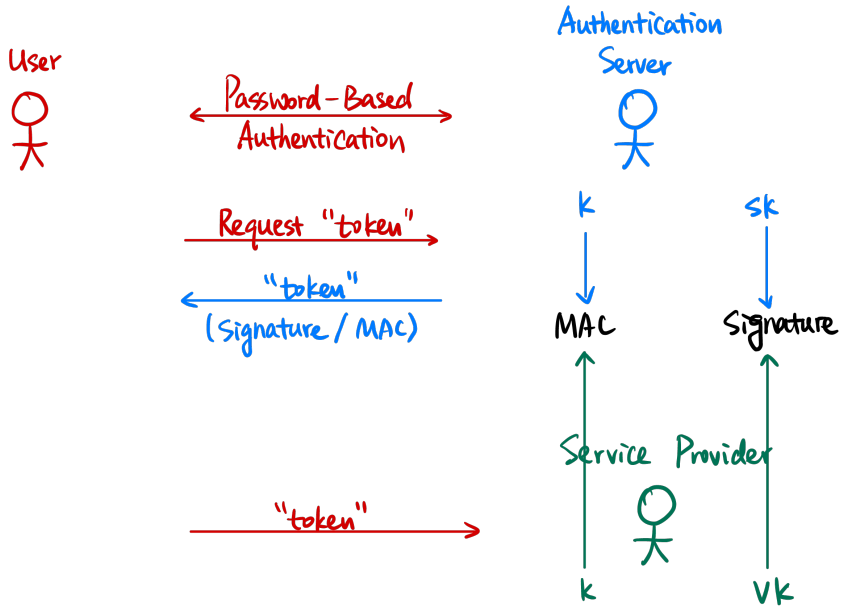


Figure 3. Schematic depiction of Signal's traffic, generated for a message m from sender A to receivers B and C in group gr with $\mathcal{G}_{gr} = \{A, B, C\}$. Transport layer protection is not in the analysis scope (gray).

§9.2 Single Sign-On (SSO) Authentication

Often, we'll 'log in with Google' or 'log in with Apple'[34]. 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.



[34] Even Brown has Shibboleth!

§9.3 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.

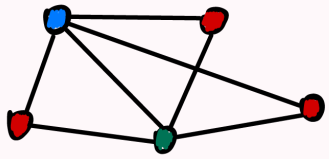
What is a proof? We consider what a 'proof system' is. For example, we'll have a *statement* and a *proof* that is a purported proof of that statement. What guarantees do we want from this proof system? If the statement is true, we should be able to prove it; and if the statement is false, we shouldn't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

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.

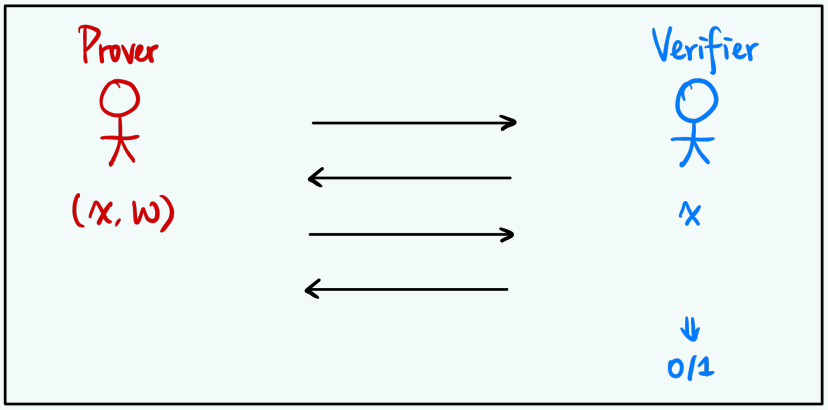
Soundness. $\forall x \notin L$, $\forall P^*$, $\Pr[P^*(x) \leftrightarrow V(x) \text{ outputs } 1] \simeq 0$. That is, for every x not in the language, our prover P^* will not be able to prove its validity to V, with negligible probability. If P^* is PPT, we call the system a *zero-knowledge argument*.

We need an additional property that this is actually *zero-knowledge*[35]. 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[36].



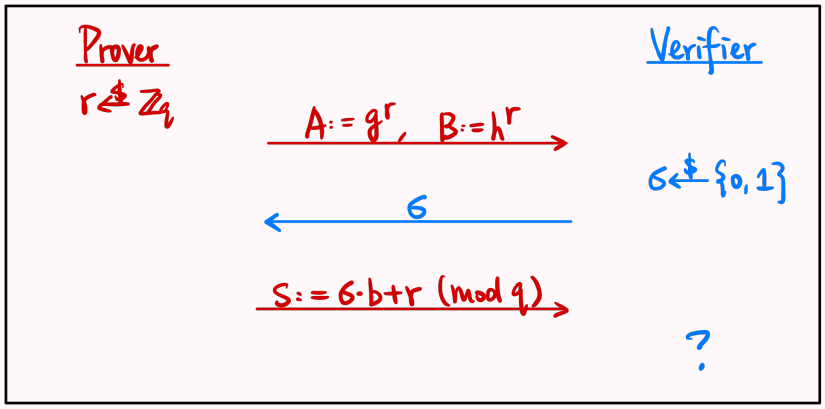
[35] That is, the prover could just send the witness in the clear to the verifier, which satisfies completeness and soundness.

[36] 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

Example 9.3 (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: If the statement is *not true*, what is the probability that the prover will be able to convince the verifier? When $\sigma = 0$, then it's easy for the prover to pass validation. When $\sigma = 1$, is it possible for the prover to send back a valid s (or, knowing that $\sigma = 1$, can they generate a first round message that makes the message valid in the third round). For example, the prover will first randomly sample s and compute $A = g^s \cdot u^{-1}$ and $B = g^s \cdot v^{-1}$ to send in the first round. However, the prover cannot *simultaneously* pass $\sigma = 0$ *and* $\sigma = 1$ without knowing b. So the probability that the prover can convince the verifier is exactly $\frac{1}{2}$.

Can we bootstrap this to be negligible? We repeat this protocol λ times and the probability will be $\frac{1}{2^\lambda}$. r, σ are randomly sampled each time. Note that we are running the protocol a polynomial number of times for an exponential decrease in probability.

We can deconstruct this probability of $\frac{1}{2}$. If the prover sends $A = g^r, B = h^r$, if $\sigma = 0$, the prover can prove this. If $\sigma = 1$, the prover will need to produce such s such that $u \cdot g^r = g^s$ and $v \cdot h^r = h^s$,

allowed to *rewind* the verifier to some previous step. If the transcript can be simulated, then surely no information is leaked from the protocol.

but then this reduces to solving for b (we can get $b = s - r$ out of such an s). Otherwise, if $A = g^{r_1}$ and $B = g^{r_2}$, and $\sigma = 0$, we'll 'catch' the phony prover.

In essence, the prover is using a one-time pad r to mask b, but when prompted, the prover can also be asked to reveal r in the clear that it is valid.

What might a simulator do? Since the simulator can 'rewind' the verifier to a previous step. The simulator will guess whether the challenge bit, and prepare the first round message. If the guess is correct, we proceed. If not, the simulator will just rewind and do it again until it is correct. This gives the basis of our intuition behind the zero-knowledge aspect of this proof system. Rewinding λ times per iteration (at most) gives us $1 - \frac{1}{\lambda^2}$ times that it will succeed, and we repeat this over λ iterations.

To bootstrap the soundness error, what if we did the entire procedure in parallel? So there will be a total of 3 messages, each containing λ iterations of information. In particular, is it still zero-knowledge? We can't rewind anymore, since we need to get λ choice bits correct, which means we rewind an exponential number of times (violating PPT simulator). *Maybe there are other ways to construct simulators...* There are some questions:

1. Can we prove that this is zero-knowledge?

2. If not, can we disprove that this is not zero-knowledge?

3. Can we, as the verifier, get more information from this protocol running in parallel?

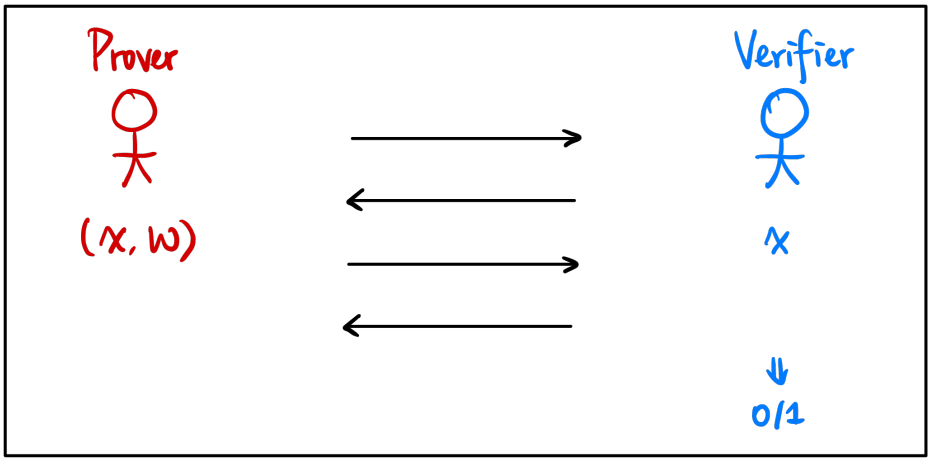
This has been an open problem for many years—it was recently proved that this is *not* zero-knowledge. However, it is still not clear whether we can extract more information from this protocol running in parallel.

§10 March 2, 2023

§10.1 Zero Knowledge Proofs, *continued*

§10.1.1 Recap

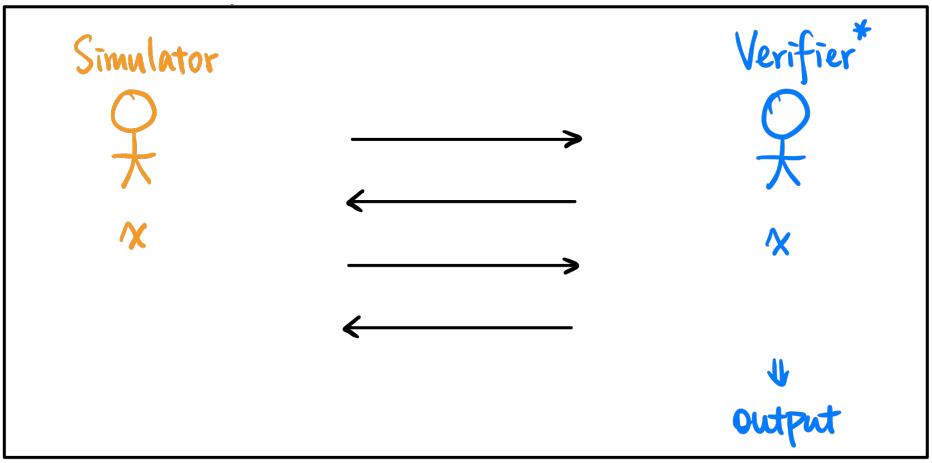
Recall that a zero-knowledge proof is an interactive proof system between two parties: a *prover* and a *verifier*. We say that (P, V) a pair of probablistic poly-time (PPT) interactive machines is a *zero-knowledge proof system* for a language L with associated relation R_L if we have



Completeness. If $\forall (x, w) \in R_L$ (x and witness w), the prover will always be able to prove that this is true.

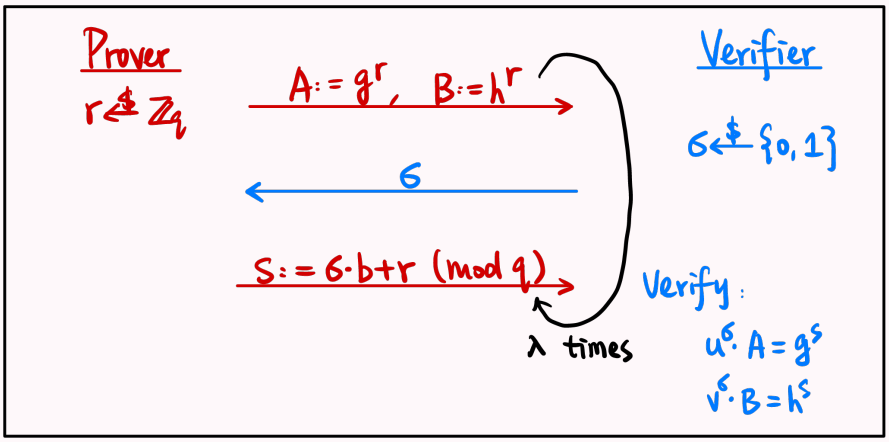
Soundness. If $\forall x \notin L$, the prover cannot convince the verifier that $x \in L$.

Furthermore, we have a zero-knowledge definition, which is a bit counterintuitive. To guarantee that the verifier doesn't learn anything from the system, it means that a simulator that doesn't know the witness w can simulate the entire transcript (all transactions between the prover and verifier). That it, since a simulator can simulate this, the verifier had better not learn any additional information.



Example 10.1

We saw a quick example of a zero-knowledge proof for the Diffie-Hellman tuple. Our witness b is the Diffie-Hellman exponent, and inputs are g^a, g^b, g^{ab}.



The prover will generate a mask $r \xleftarrow{\$} \mathbb{Z}_q$. The prover sends $A := g^r, B := h^r$ to the verifier. The verifier issues a challenge bit, and the prover either provides the mask $s := r$ if $\sigma = 0$, otherwise the prover will provide b masked with r $s := b + r$.

We'll detail some of the desired properties of this zero-knowledge proof:

Completeness. Completeness is straightforward, since the prover will always convince the verifier.

Formally, $\forall (x, w) \in R_L$ (any x with witness w in language), $\Pr[P(x, w) \leftrightarrow V(x) \text{ outputs } 1] = 1$.

Soundness. If the statement is not true, the prover cannot convince to the verifier. The prover

can only convince the verifier with probability $\frac{1}{2}$ on every iteration, repeating λ iterations makes this probability $\frac{1}{2^\lambda}$ (we want this! we want negligible probability that the prover succeeds).

Formally, $\forall x \notin L$, $\forall$PPT P^* (a malicious prover, who is trying to prove $x \in L$), $\Pr[P^*(x) \leftrightarrow V(x) \text{ outputs } 1] \simeq 0$.

Zero Knowledge. Since the simulator can 'rewind time', it will guess the challenge bit σ. If it is correct, it'll proceed, otherwise it will rewind until the correct σ is chosen. This takes at worst[37] $O(\lambda^2)$ attempts over λ.

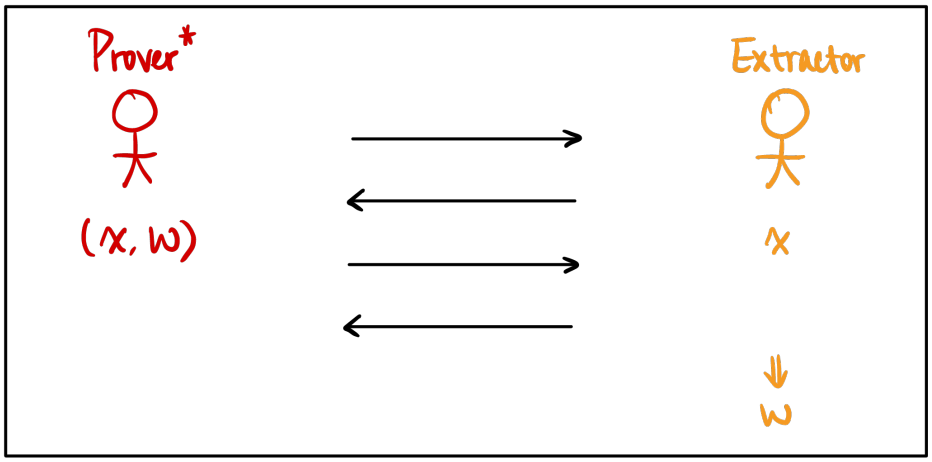
Formally, $\forall$PPT V^* (for any verifier, even one acting maliciously), $\exists$PPT S (exists a simulator) such that $\forall (x, w) \in R_L$, the simulator's output is computationally indistinguishable from the output $\text{Output}_{V^*}[P(x, w) \leftrightarrow V^*(x)]$ (whatever the verifier outputs in the real world).

§10.1.2 Proof of Knowledge

We missed something very subtle in our soundness guarantee. Consider a case where *every* $x \in L$. Our soundness guarantee is moot here—the prover will never be attempting to prove something false. Yet, our model doesn't require the prover to actually know the witness w.

There is an even stronger assumption we can make, *Proof of Knowledge* that describes protocols where the prover needs to actively know a witness, not just that a certain element is in the language.

We define Proof of Knowledge similarly to Zero Knowledge property.

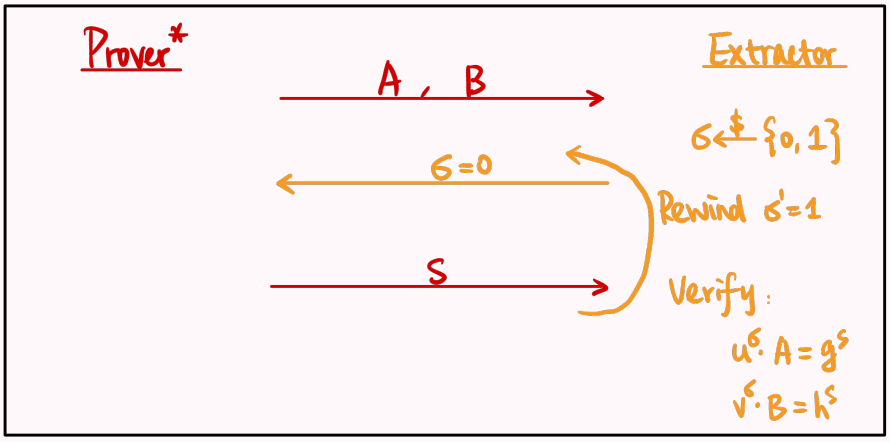


[37] That is, the probability of reaching a $O(\lambda^2)$ runtime is negligible in λ.

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.

Example 10.2

How might an extractor get witness b in the Diffie-Hellman example?



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.

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 $x \in R_L$.

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.

§10.2 Schnorr's Identification Protocol

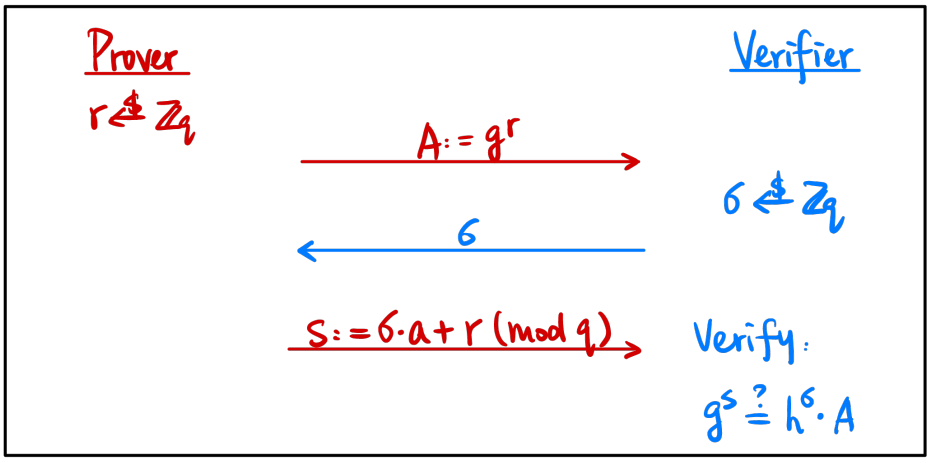
> **Remark.** *The following is a hodgepodge of (arguably) clearer notes taken for Peihan's previous cryptography seminar, as well as lecture and slides from the current offering of Applied Cryptography. The notation might not correspond to lecture 1-to-1, but they are the same protocols albeit with different symbols.*

We saw a variant earlier with the Diffie-Hellman triple proof. This, the Schnorr's Identification Protocol, is the general form.

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

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

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.

§10.2.1 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

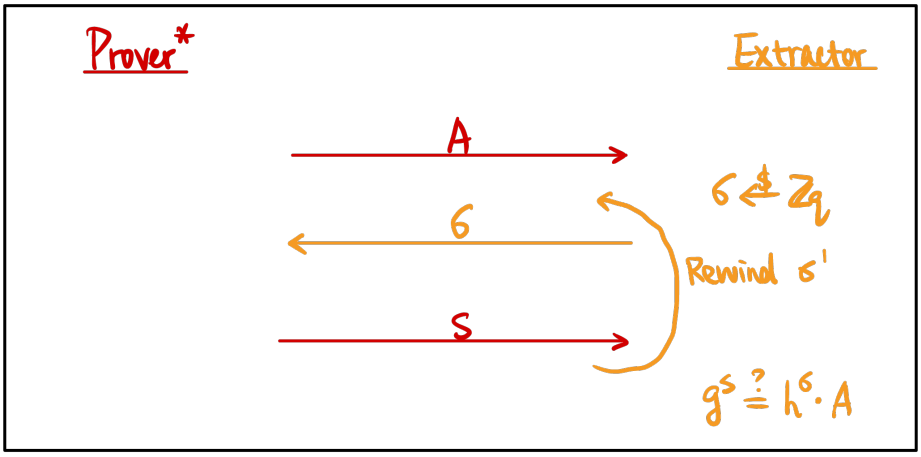
$$g^s = h^\sigma \cdot u$$
$$g^{s'} = h^{\sigma'} \cdot u$$

Given these two equations, we can

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

Then we have

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



§10.2.2 Honest-Verifier Zero-Knowledge

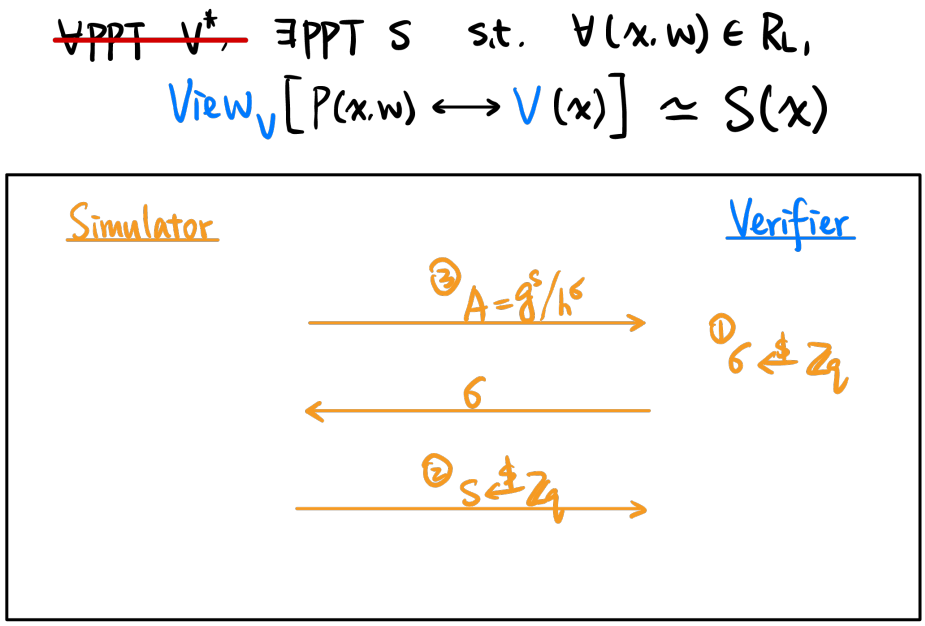
Can we also construct Zero-Knowledge for this protocol?

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

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 $g^s \cdot h^{-\sigma} = A$.

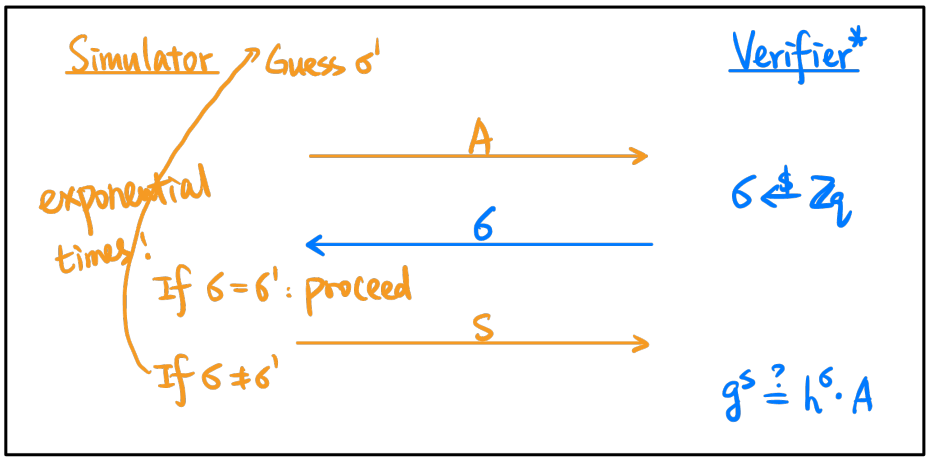
§10.2.3 Zero-Knowledge

What about Zero-Knowledge? This interaction happens in order so we cannot do the same neat trick we performed earlier. We have to commit to a u first before we get c. *It turns out that this is an open question.*

However, given that it is Honest-Verifier Zero-Knowledge, can we make it Malicious-Verifier ZK? We can construct another protocol that is resistant against a malicious verifier.

We can have the challenger commit to σ before A is transmitted.

How can we make a simulator that talks to a malicious verifier? After the commitment is opened, the simulator can rewind to a state after the commitment (but before the opening), and the simulator can generate u and s such that they are consistent in the same way we did in the Honest-Verifier Zero-Knowledge case.



§10.3 Sigma Protocols

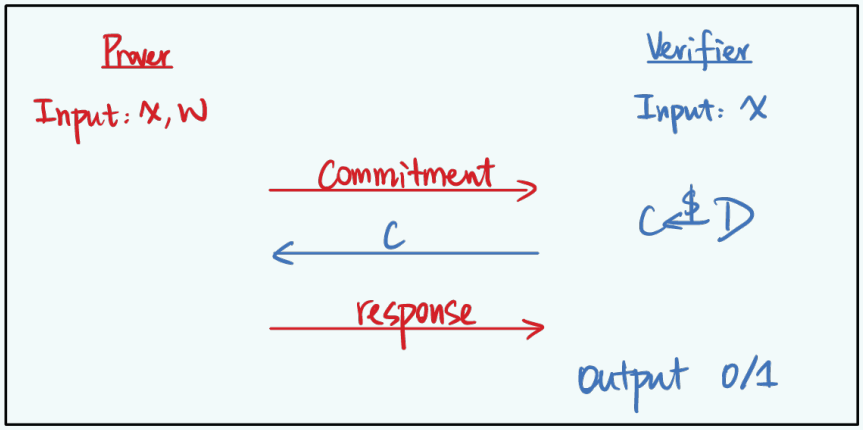
Definition (Sigma Protocols)

Sigma Protocols are generally 3-round protocols:

Round 1. The prover has some input x, w, and commits to these inputs.

Round 2. *Public coin.* The verifier samples $c \xleftarrow{\$} D$.

Round 3. The prover gives a response and the verifier checks if this is consistent.



It's called a *sigma protocol* because the shape looks like a capital sigma (Σ). It's a special zero-knowledge proof-of-knowledge.

§10.3.1 Motivation

Why do we care about Sigma Protocols?

Theoretically, a ZK proof can be given for any NP language: any NP language can be reduced to three-colorings which has a ZK proof. However, these are very complicated and expensive. The NP reduction is also expensive in practice.

To make Zero-Knowledge proofs more accssible and usable, we have differnt techniques for different languages and scenarios. Sigma protocols are a special case of ZK proofs that are very efficient to perform, especially for languages that are useful in crypto.

For example, the case of discrete-log is very useful in cryptography. We'll see some other examples of languages that can be proved through a Sigma protocol.

§10.4 Chaum-Pedersen Protocol for Diffie-Hellman Tuple

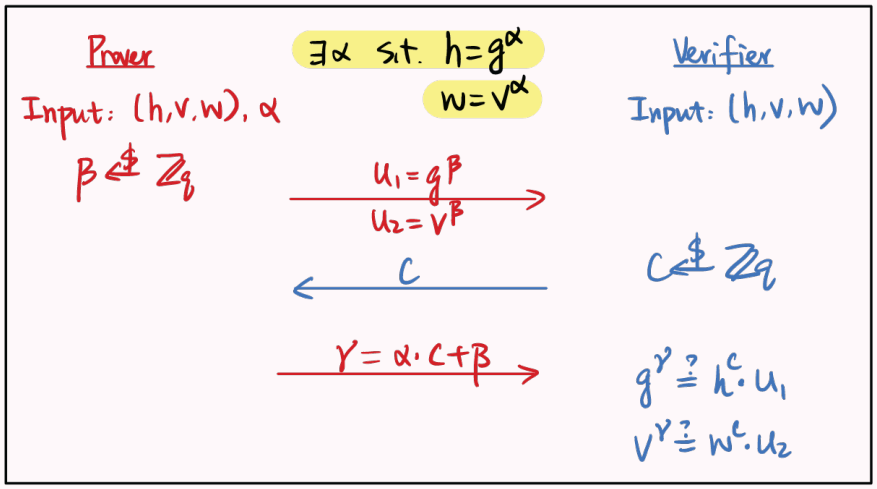
Example 10.3 (Diffie-Hellman Tuple)

We want to prove that h, v, w is a valid Diffie-Hellman Tuple (that is, $h = g^\alpha, v = g^\beta, w = g^{\alpha\beta}$) without exposing their exponents.

Round 1. The prover has some (h, v, w, α). The prover sends $u_1 = g^\beta$ and $u_2 = v^\beta$ with $\beta \xleftarrow{\$} \mathbb{Z}_q$.

Round 2. The verifier samples the public coin $c \xleftarrow{\$} \mathbb{Z}_q$ randomly.

Round 3. The prover sends $\gamma = \alpha \cdot c + \beta$ and the verifier checks that $g^\gamma \stackrel{?}{=} h^c \cdot u_1$ and $v^\gamma \stackrel{?}{=} w^c \cdot u_2$



§10.4.1 Proof-of-Knowledge

How do we construct Proof-of-Knowledge? How do we construct an extractor that talks to a (potentially malicious) prover to try to extract the knowledge from the protocol.

We do the same thing: run it for two iterations. We have $g^{\gamma} = h^{c} \cdot u_1$ and $v^{\gamma} = w^{c} \cdot u_2$, $g^{\gamma'} = h^{c'} \cdot u_1$ and $v^{\gamma'} = w^{c'} \cdot u_2$. Then we have

$$g^{\gamma-\gamma'} = h^{c-c'}$$
$$v^{\gamma-\gamma'} = w^{c-c'}$$

Then we have

$$\alpha = (\gamma - \gamma') \cdot (c - c')^{-1}.$$

§10.4.2 Honest-Verifier Zero-Knowledge

If we have a simulator, can we simulate the transcript that is indistinguishable from the real world transcript?

We can do the same thing again. We fix $c \xleftarrow{\$} \mathbb{Z}_q, \gamma \xleftarrow{\$} \mathbb{Z}_q$ first and then derive u_1 and u_2:

$$u_1 = g^{\gamma}/h^{c}$$
$$u_2 = v^{\gamma}/w^{c}$$

§10.5 Okamoto's Protocol for Representation

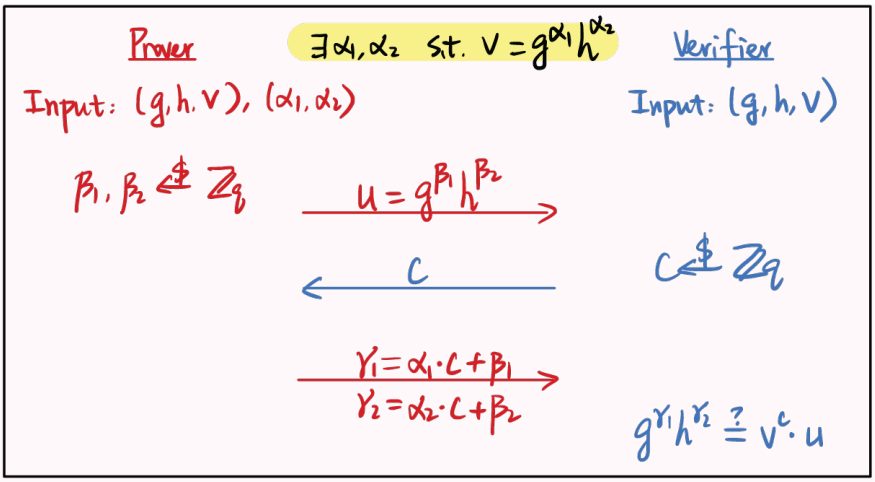
Example 10.4

We have 2 elements g, h. We want to prove we can show existence of α_1, α_2 such that $v = g^{\alpha_1} h^{\alpha_2}$.

Round 1. We sample $\beta_1, \beta_2 \xleftarrow{\$} \mathbb{Z}_q$ and send $u = g^{\beta_1} h^{\beta_2}$, in the same form as v.

Round 2. Verifier samples public coin challenge $c \xleftarrow{\$} \mathbb{Z}_q$.

Round 3. Prover computes 2 exponents, $\gamma_1 = \alpha_1 c + \beta_1, \gamma_2 = \alpha_2 c + \beta_2$. Verifier checks $g^{\gamma_1} h^{\gamma_2} \stackrel{?}{=} v^{c} \cdot u$.



We can prove proof-of-knowledge and honest-verifier zero-knowledge in the same way.

§10.5.1 Proof-of-Knowledge

We get two sets

$$g^{\gamma_1} h^{\gamma_2} = v^c \cdot u$$
$$g^{\gamma_1'} h^{\gamma_2'} = v^{c'} \cdot u$$

so

$$g^{\gamma_1 - \gamma_1'} \cdot h^{\gamma_2 - \gamma_2'} = v^{c - c'}$$
$$g^{\frac{\gamma_1 - \gamma_1'}{c - c'}} \cdot h^{\frac{\gamma_2 - \gamma_2'}{c - c'}} = v$$

where $\alpha_1 = \frac{\gamma_1 - \gamma_1'}{c - c'}, \alpha_2 = \frac{\gamma_2 - \gamma_2'}{c - c'}$.

§10.5.2 Honest-Verifier Zero-Knowledge

Can be done in the same way.

§11 March 7, 2023

§11.1 Zero-Knowledge Proof, *continued again*

§11.1.1 Recap

We'll review some definitions from last time, pertaining to the *Zero-Knowledge Proof of Knowledge*.

Completeness. If x is in the language, the prover is able to prove it.

Soundness. If x is not in the language, the probability that *any* prover can prove $x \in R_L$ is negligible.

Proof of Knowledge. Note in the soundness guarantee, if everything is in the language, we don't have the guarantee that P knows the witness. This notion gives us a guarantee that the prover can only prove it if it has access to the witness. There should be an extractor that, through interacting with the prover, can extract the witness.

Honest-Verifier Zero-Knowledge. $\exists$PPT S such that $\forall (x, w) \in R_L$,

$$\text{View}_V[P(x, w) \leftrightarrow V(x)] \simeq S(x).$$

That is, a simulator can simulate the view between an honest prover and verifier.

Zero-Knoweledge. The previous statement, but the verifier is allowed to be malicious—yet the simulator should still be able to simulate the output (with rewinding).

The following was review of protocols (with completeness, proof of knowledge, and HVZK/ZK properties) from last time. Refer to the Schnorr (section 10.2), Chaum-Pedersen (section 10.4), and Okamoto (section 10.5) ZKP protocols.

§11.1.2 Arbitrary Linear Equations

Example 11.1 (Arbitrary Linear Equations)

Arbitrary linear equations and do similarly as before. We're in cyclic group $\mathbb{G}$ of order q with generator g. Say we have some private a, b, c such that $u = g^a h^b$ and $v = u^a v^v g^c$.

Round 1. Sample 'masks' $r_1, r_2, r_3 \xleftarrow{\$} \mathbb{Z}_q$ for each r_i and send the exponentiated masks. Send

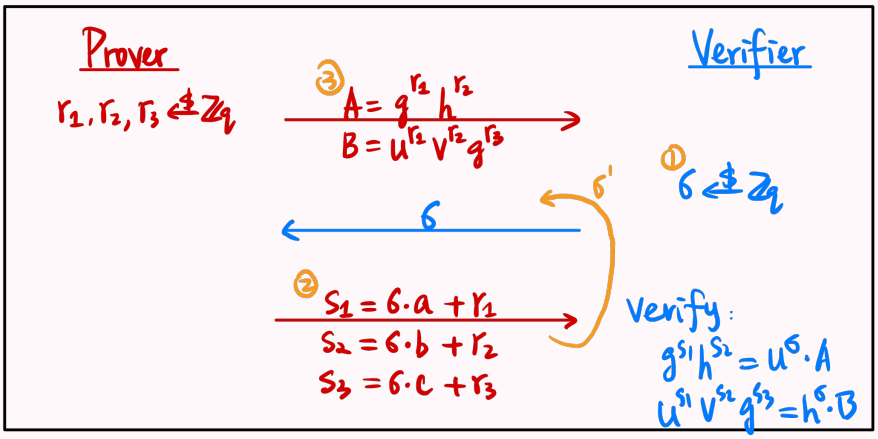
$$A = g^{r_1} h^{r_2}$$
$$B = u^{r_1} v^{r_2} g^{r_3}$$

Round 2. Public challenge $\sigma \xleftarrow{\$} \mathbb{Z}_q$.

Round 3. Send

$$s_1 = \sigma \cdot a + r_1$$
$$s_2 = \sigma \cdot b + r_2$$
$$s_3 = \sigma \cdot c + r_3$$

for each of a, b, c and verify.



Completeness, proof-of-knowledge and honest-verifier zero-knowledge done the exactly same way as we've been seeing before.

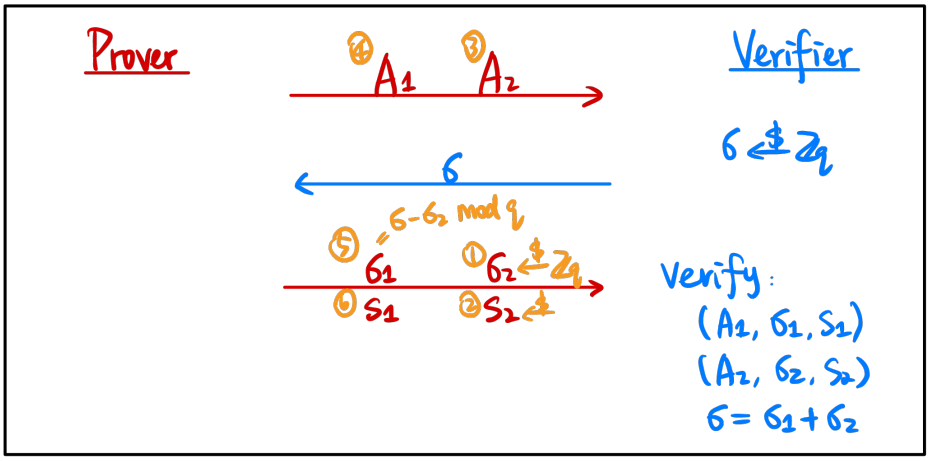
§11.1.3 AND and OR statements

Let's say we have some $(x_1, w_1) \in R_{L_1}$ and $(x_2, w_2) \in R_{L_2}$, how should we prove and/or relations between these?

The case of AND is simple: we can just run the protocol twice.

What about an OR statement? We have one witness for one statement, we want to prove that *at least one* is true, but without revealing to the verifier which they are proving is true exactly.

Here's what we'll do.



The prover will prove that *both* are true by running both protocols. The prover sends A_1, A_2 to the verifier. The verifier responds with a single challenge $\sigma \xleftarrow{\$} \mathbb{Z}_q$. Then, the prover will provide S_1, S_2 based on challenges σ_1, σ_2, and send all this to the verifier. The verifier checks $\sigma_1 + \sigma_2 = \sigma$. The intuition behind this is that the prover can fix one σ_i, and the other is a true challenge.

We rely on the fact that if the prover knows the challenge beforehand, they can use the simulator to generate a valid proof. WLOG say the prover knows x_1, then the prover will *fix* $\sigma_2 \xleftarrow{\$} \mathbb{Z}_q$ and generate a first round message A_2 knowing σ_2. Then, the prover will generate a valid first round message for the statement they know, A_1. When they receive challenge σ from the verifier, they compute $\sigma_1 = \sigma - \sigma_2$ and prove x_1 honestly.

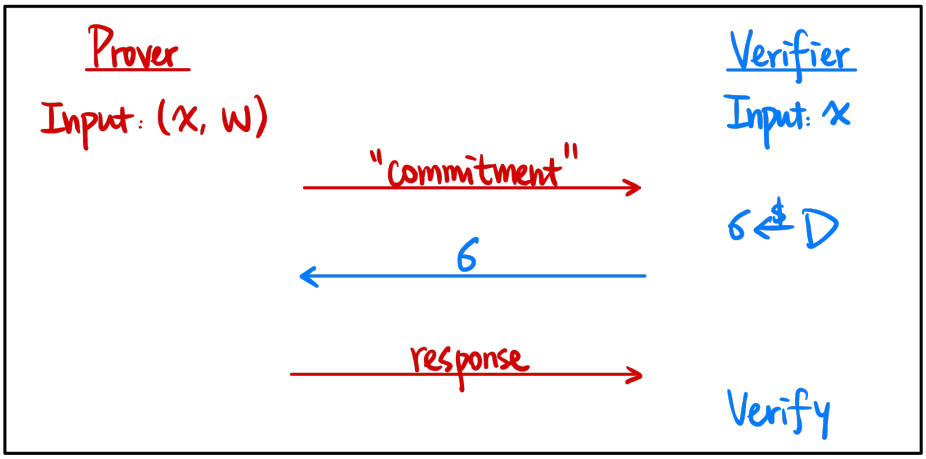
Completeness? This is straightforward, it follows from the completeness of the respective protocols.

Proof of Knowledge? How can we construct an extractor that can extract a valid witness, either w_1 for x_1 or w_2 for x_2? We'll use the previous extractors E_1 and E_2.

After challenge and response, we'll rewind to before sending challenge σ and send a different challenge σ'. If $\sigma_1 + \sigma_2 = \sigma$, and $\sigma_1' + \sigma_2' = \sigma' \neq \sigma$. Then at least $\sigma_1 \neq \sigma_1'$ or $\sigma_2 \neq \sigma_2'$. Say WLOG $\sigma_1 \neq \sigma_1'$, so we know (A_1, σ_1, s_1) and (A_1, σ_1', s_1'), and we use the corresponding extractor to extract w_1.

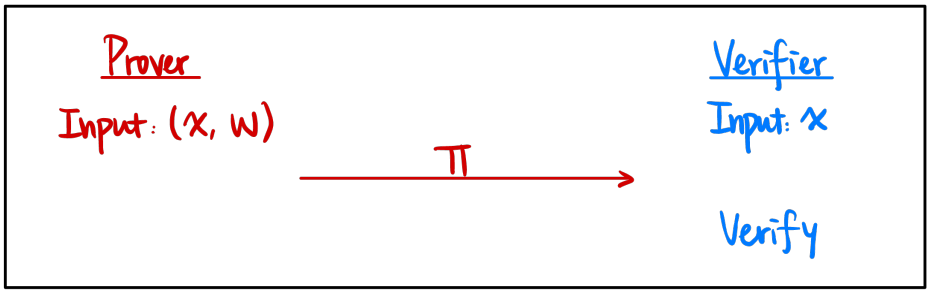
Honest Verifier Zero Knowledge? We sample $\sigma_1, \sigma_2, S_1, S_2 \xleftarrow{\$}$ randomly (then $\sigma = \sigma_1 + \sigma_2$ appears random), then compute our initial commitments A_1, A_2.

§11.1.4 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[38]. 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.

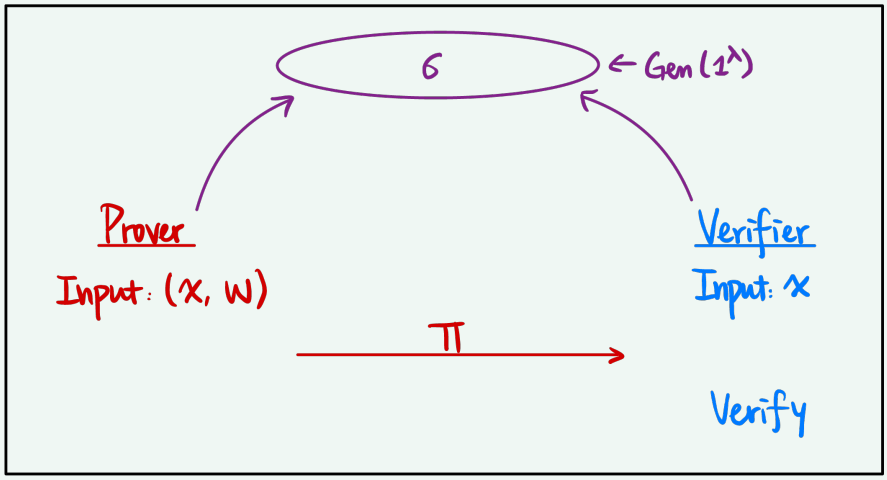
[38] This is generally impossible! For example, a NIZK for the DH tuple that satisfies Zero-Knowledge breaks the Decisional Diffie-Hellman assumption.

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.

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.

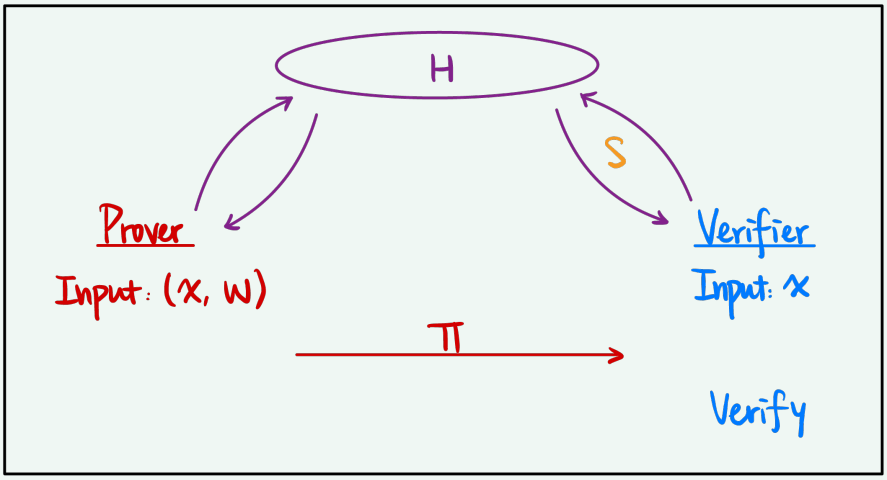
In reality, the CRS can be generated in a *key ceremony* between parties such that no party can interfere with the generated key.



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.



§11.1.5 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[39] 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 11.2

The Fiat-Shamir Heuristic can also transform Schnorr's Identification Protocol into Schnorr's Signature Scheme in the RO model.

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,

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

the random coin sample contains a hash of m as well,

Incomplete

§11.2 Anonymous Voting

We'll go over a very high level idea for anonymous online voting systems.

Say we have n voters, and each will vote 0 or 1 for some candidate. Each voter will generate an encryption of their vote, encrypting 0 or 1. We want to add the votes together, in a *homomorphic* way. We'll homomorphically (that is, the sum of ciphertexts is the sum of plaintexts) combine these votes altogether. We'll then decrypt the final encryption to learn the summation.

There are some problems we need to solve:

- Who is going to decrypt? Is there a single party who can decrypt the final ciphertext? If a single party can decrypt the final ciphertext, then one party can decrypt every single vote. We want the final decryption to be done jointly by a few parties.
- How can we combine these encryptions into an encryption of the summation based on the schemes we've learned so far?
- What if voters behave maliciously? What if a voter encrypts, say, 100? *We'll use zero-knowledge proofs for this.*

§12 March 9, 2023

§12.1 Intuition for Zero Knowledge Proofs

There has been feedback that the sections covering Zero Knowledge Proofs have been mathematical and unintuitive.

We'll try to provide some intuition for Zero Knowledge Proofs. There are two main components: *Zero Knowledge* and *Proof*.

A prover, in a proof, is trying to prove that they know some witness of an element in a language R_L. They need to be able to prove that they have this knowledge if they do (*completeness*), and they should not be able to prove this if they don't have this information (*soundness*).

For the zero-knowledge aspects: a verifier should not learn any information about the witness. One standard of security (honest-verifier zero knowledge) asks that the verifier acts honestly. Other standards allow all forms of verifiers.

One can review section 10.2 for an example of a zero-knowledge proof and components involved.

§12.2 Putting it Together: Anonymous Online Voting

We identified a few issues we would like resolved for our voting scheme:

§12.2.1 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 12.1 (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![40]

§12.2.2 Threshold Encryption

We've solved our first problem—we can add our votes together. We move on to figuring out which party will decrypt the vote. If one party could decrypt the vote, they would decrypt every vote.

Our solution is to use a threshold encryption scheme. Many parties will generate a *partial* public key, and only together can they form a public key that can decrypt the message. Even if *all but one* party colludes to try to decrypt, they will not be able to gain information about the ciphertext.

To decrypt, all parties will partially decrypt the ciphertext. They can use the partial keys to partially decrypt the ciphertext.

This is t out of t threshold encryption—all t parties will need to decrypt. You can have arbitrary ratios, like 3-out-of-t (you can only decrypt if you have 3 parties), or $\frac{t+1}{2}$-out-of-t (you need a majority to decrypt).

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

Example 12.2

This works quite well for ElGamal encryption. Each partial decryptor will generate their own ElGamal secret key and public key.

$$\begin{aligned} P_1 : sk_1 &\xleftarrow{\$} \mathbb{Z}_q, \qquad pk_2 = g^{sk_1} \\ P_2 : sk_2 &\xleftarrow{\$} \mathbb{Z}_q, \qquad pk_2 = g^{sk_2} \\ &\vdots \\ P_t : sk_t &\xleftarrow{\$} \mathbb{Z}_q, \qquad pk_t = g^{sk_t} \end{aligned}$$

The combined public key will be $pk = \prod pk_i = \prod g^{sk_i} = g^{\sum sk_i}$. The combined secret key will be $\sum sk_i \pmod q$. We have to figure out the summation of *all* the sk_i. If we only have $t-1$ secret keys, we're still missing one summand which acts as a one-time pad masking the entire secret key.

To encrypt, we'll just encrypt using pk. $\mathsf{Enc}_{pk}(m) = (g^r, pk^r, g^m)$.

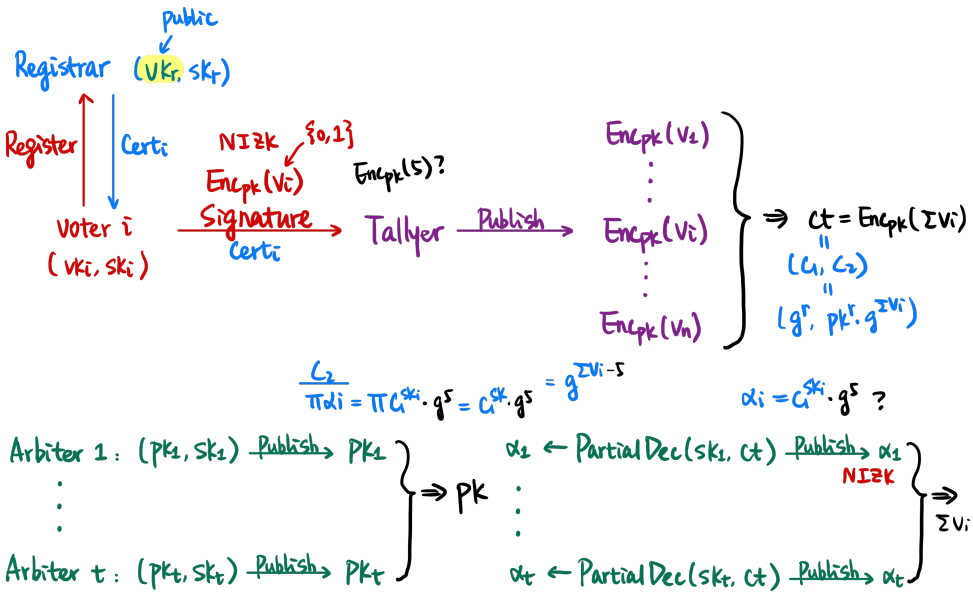
To decrypt, *every party* will need to decrypt using their partial key. Each party will unravel c_1,

$$\begin{aligned} P_1 : \alpha_1 = c_1^{sk_1} &\longrightarrow \alpha_1 \\ P_2 : \alpha_2 = c_2^{sk_2} &\longrightarrow \alpha_2 \\ &\vdots \\ P_t : \alpha_t = c_t^{sk_t} &\longrightarrow \alpha_t \end{aligned}$$

and $\frac{c_2}{\alpha_1 \cdot \alpha_2 \cdots \alpha_t} = g^m$ exactly noting $\prod \alpha_i = \prod c_1^{sk_i} = c_1^{\sum sk_i} = c_1^{sk}$. This ensures that everyone has to participate in the decryption.

Note that they decrypt to a specific message with a specific c_1, so you couldn't use this reconstituted decryption key to decrypt an individual vote.

§12.2.3 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[41]. Then, they will strip the signature and output $\mathsf{Enc}_{pk}(v_1), \ldots, \mathsf{Enc}_{pk}(v_i), \ldots, \mathsf{Enc}_{pk}(v_n)$.

At the end, arbiters will take the published votes and multiply them all together, then jointly decrypt them.

What if the voter is malicious? Nothing stops the voter from encrypting 5, or 100. This is guaranteed since votes are *hiding* so we won't know what is under a vote.

What if an arbiter is malicious? What if an arbiter doesn't partially decrypt honestly, and they can tweak the vote. If an arbiter computed $\alpha_i = c_1^{sk_i} g^k$, then decryption will give us $\frac{c_2}{\prod \alpha_i} = \frac{c_2}{\prod c_1^{sk_i} \cdot g^k} = \frac{c_2}{c_1^{sk_i} \cdot g^k} = g^{\sum v_i - k}$. This allows each arbiter to easily change the vote outcome.

[41] This way, the registrar cannot figure out who has voted and who hasn't.

We'll use zero knowledge proofs here: the voter will use a zero knowledge proof to prove that they only encrypted 0 or 1, and the arbiter will use a zero knowledge proof to prove that their α_i is correcet.

§12.2.4 Correctness of Encryption

We want voters to prove that their encryption is either of 0 or 1. This is perfect for a sigma OR protocol!

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[42] c is an encryption of 1."

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

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

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 ($c_2 = pk^r \cdot g$).

In both cases, we can use Chaum-Pedersen exactly (section 10.4) to prove Diffie-Hellman tuples. For the former, (c_1, pk, c_2) is a Diffie-Hellman tuple in the form (g^a, g^b, g^{ab}). For the latter, $(c_1, pk, c_2 \cdot g^{-1})$ is a Diffie-Hellman tuple in the same form.

Using the Fiat-Shamir heuristic (see section 11.1.5), we can convert this proof to a NIZK proof, and the voter can just publish their proof along with their vote.

§12.2.5 Correctness of Partial Decryption

Again, given cyclic group $\mathbb{G}$ of order q with generator g, we have partial public key $pk_i \in \mathbb{G}$ which produces partial decryption α_i. Our witness is the partial secret key sk_i.

Out language is

$$R_L = \{((pk_i, c_1, \alpha_i), \underbrace{sk_i}_{w}) : pk_i = g^{sk_i} \wedge \alpha_i = c_1^{sk_i}\}$$

and we prove that $pk_i = g^{sk_i}$ and $\alpha_i = c_1^{sk_i}$. How might we prove this?

[42] This is a Sigma OR statement, not just any plain 'or'. Recall in section 11.1.3 that a prover can prove arbitrary OR statements wihout revealing to the verifier which statement they are proving. So that, in this scenario, no verifier can know which statement the voter is attempting to prove, so nothing about their vote is revealed.

This is still the Diffie-Hellman tuple! $pk_i = g^{sk_i}, c_1 = g^r, \alpha_i = g^{r \cdot sk_i}$.

We'll use Chaum-Pedersen again, and condense this into a non-interactive proof using the Fiat-Shamir heuristic the same way. Each arbiter will publish a their NIZK proof that their partial decryption is correct.

§12.2.6 Generalizations?

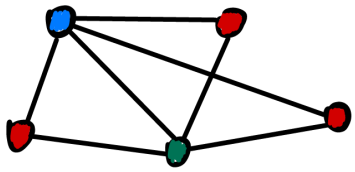
This is currently for 1 candidate. What if we have k candidates?

A naïve solution is to encrypt $v_n \in \{0, 1, \ldots, k-1\}$. But then adding the votes wouldn't give us any information of how people voted.

One solution is that each voter will vote once (provide some (c_1, c_2)) for each candidate. Then, the voter will prove in zero knowledge that each vote is $0, 1$ and that the sum of votes is 1 exactly (if we restrict every voter to vote for exactly 1, but we can adapt this scheme by composing our ZK proofs for arbitrary configurations). The arbiters run each election separately.

§12.3 Zero-Knowledge Proof for Graph 3-Coloring

We'll show a zero-knowledge proof for an NP-hard problem, the graph 3-coloring. To prove any arbitrary NP language, we can reduce it to graph 3-coloring (by a polynomial-time reduction), and prove that.[43]



Our NP language is $L = \{G : G \text{ has a 3-coloring}\}$. Our NP relation is $R_L = \{(G, \mathsf{3col})\}$ where $\mathsf{3col}$ is a 3-coloring for graph G (our witness).

We'll continue this next time.

[43] This is a... theoretical result. It signifies that any NP language can be proved in zero-knowledge, but is not used in practice that I know of. Most applications of ZK proofs will use sigma protocols.

§13 March 14, 2023

Comments are still open. There was a brief comment about questions in class—Peihan will still continue to ask questions (as she deems necessary) and students are very welcome to discuss amongst themselves.

There have been some comments about the mathematical nature of the course. Some students think it's a good balance, while some feel like it's too focused on theory. Going forward, Peihan will be taking more of an intuitive approach so that concepts make sense and are justified. Students also enjoyed having real-world applications of the theory.

§13.1 Zero-Knowledge Proofs for All NP

§13.1.1 Intuition

To give zero-knowledge proofs for all NP, we can just provide a ZK proof for *one* NP-complete problem. We'll use the graph 3-coloring. The idea is that since all NP problems *reduce* to an NP-complete problem, solving this gives us a feasible theoretical solution for proving all languages in NP.

The language will be $L = \{G : G \text{ has a 3-coloring}\}$, and our witness will be the precise 3-coloring of a given graph.

In a high level, the verifier will tell the prover two vertices, and the prover will reveal the colors of those two vertices. This follows from the protocols we've seen in the past where the prover will commit to something, be challenged by the verifier, and reveal their commitment to be valid.

What is the probability that the prover will be caught? If the graph has at least one edge that has vertices of the same color, then the probability of being caught will be $\geq \frac{1}{|E|}$.

Can we amplify this soundness? If we challenged 2 different edges, the verifier could learn *more* information about the graph and the coloring. How might we do this while still maintaining zero-knowledge for the verifier? Every time, the prover can create a new graph with different colors (that is, permute the colors) in it. This ensures that the colors from one proof will be independent of the colors of another proof.

Then, if the verifier tries to challenge another edge, the colors of the vertices with be independent. Note thaht the prover will *commit* to the colors, so it's not just a case of the prover proclaiming 'oh it was red and blue' on every challenge[44].

Over t iterations, this gives us probability $\left(1 - \frac{1}{|E|}\right)^t$. Setting $t = \lambda \cdot E$, using $\left(1 - \frac{1}{N}\right)^N \approx \frac{1}{e}$ gives $\left(1 - \frac{1}{|E|}\right)^{\lambda \cdot E} = \left(\frac{1}{e}\right)^{\lambda}$.

[44] Think of this as putting pieces of paper on each vertex, and being asked to show two adjacent vertices.

§13.1.2 Commitment Schemes

That was the intuition of this zero-knowledge proof. We want to hide some number of vertices while committing to the graph (we should not be able to change it to something else). This can be done using commitment schemes.

Definition 13.1 (Commitment Scheme)

A commitment scheme is a way for a sender to commit to a message without revealing the message to the receiver.

The sender, with message $\{0,1\}$, will generate randomness $r \xleftarrow{\$} \{0,1\}^\lambda$ and generate commitment $c := \text{Com}(m; r)$. c is sent to the receiver.

When the sender wants to reveal, the sender will send (m, r) (the randomness) to the receiver and the receiver can verify that $c = \text{Com}(m; r)$.

Example 13.2 (Pedersen Commitment)

Let's say we have a cyclic group $\mathbb{G}$ of order q with generator g. We randomly sample element $h \xleftarrow{\$} \mathbb{G}$.

To commit m with random $r \xleftarrow{\$} \mathbb{Z}_q$, the sender sends sends $c := \text{Com}(m; r) = g^m \cdot h^r$. This commitment is akin to the second component of the ElGamal encryption. From this group element, the receiver *cannot* infer any information from the message m since it is indistinguishable from uniform from $\mathbb{G}$. We can find r values such that $c = g^0 \cdot h^r$ as well as r' such that $c = g^1 \cdot h^{r'}$. This does us no good.

r essentially acts like a one-time pad to mask m.

What do we want from our commitment schemes?

Hiding. The receiver will not be able to figure out what message was committed to. $\text{Com}(0; r) \simeq \text{Com}(1; s)$. There is *perfectly hiding* which is that the distributions are the same $\text{Com}(0; r) \equiv \text{Com}(1; s)$. There is also *computationally hiding* which is that the distributions are computationally indistinguishable $\text{Com}(0; r) \overset{c}{\simeq} \text{Com}(1; s)$.

Binding. It is hard to find another r such that $\text{Com}(0; r) = \text{Com}(1; r)$. There is *perfectly binding* such that $\forall r, s$, $\text{Com}(0; r) \neq \text{Com}(1; s)$. There is also *computationally hiding* such that any PPT sender cannot find r, s such that $\text{Com}(0; r) = \text{Com}(1; s)$.

We need both these properties for the initial commitment to the colors.

What does our earlier example, the Pedersen commitment, satisfy? It is *perfectly hiding*, h^r is a random group element, which will act as a one-time-pad masking g^m.

We note that the Pedersen commitment cannot be perfectly binding if it is perfectly hiding—the sets must be disjoint for perfect binding, but then their distributions are not the same under the hiding property. In this case, the Pedersen commitment is computationally binding. Is it possible for a sender to find r, s such that $g^0 \cdot h^r = g^1 \cdot h^s$? This is equivalent to finding $g = h^{r-s}$, which is equivalent to solving discrete log of g base h (which is computationally hard).

So, the Pedersen commitment satisfies perfectly hiding and computationally binding properties.

Using this commitment scheme, the prover will commit to every vertex color, then only open the ones that the verifier chooses.

§13.1.3 Protocol

This gives us the protocol for the zero-knowledge proof for graph 3-coloring.

Out input is $G = (V, E)$, and some witness $\phi : v \to \{0, 1, 2\}$. With some computationally hiding[45] and perfectly binding commitment scheme, we can enact the following protocol.

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.

§13.2 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 $\mathrm{C}(x, w) = H(w) - x + 1$. 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.

[45] Slightly different from above, but also doable.

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.

§13.2.1 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 paramter), and the verifier also verifies in $O(|c| \cdot \lambda)$. This is the same for NIZK.

Can we make this proof system more succinct? Can we have communication and verification complexity to be *sublinear* in $|c|$ and $|w|$?

§13.3 Succinct Non-Interactive Argument (SNARG)

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

Definition 13.4 (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 13.5 (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.

Example 13.6 (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.

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

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 13.7 (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).

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.

§14 March 16, 2023

§14.1 Succinct Non-Interactive Argument (SNARG)

Definition 14.1

A non-interactive proof (unbounded prover) or argument (polybounded prover) system is succinct if

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

A SNARK is a Succinct Non-Interactive Argument of Knowledge. A zk-SNAR{K/G} is a SNAR{K/G} with zero-knowledge.

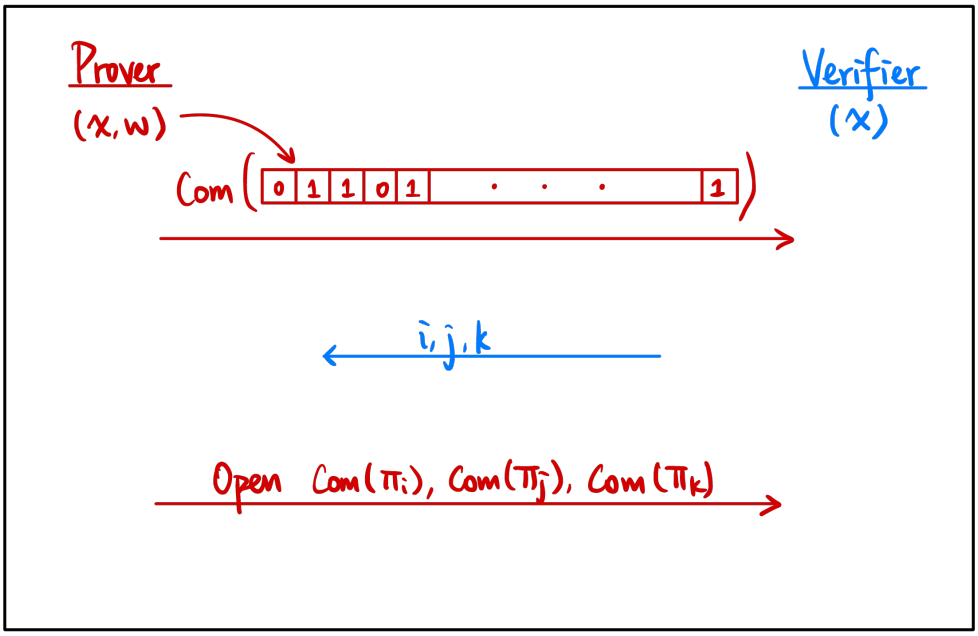
We rely on this theorem (without proof):

Theorem 14.2 (PCP Theorem)

Every NP language has a PCP where the verifier reads only a *constant* number of bits of the proof.

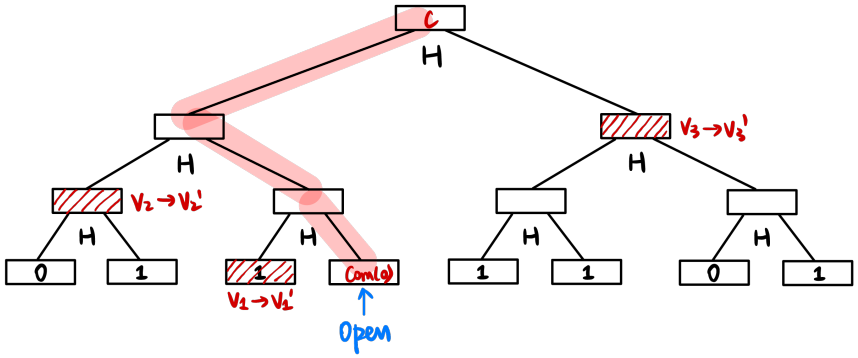
This theorem states that a verifier only needs to read a constant number of bits of the proof, which could condense a proof into some constant time.

Our first attempt could be to commit to the entire proof, and the verifier challenges with the bits it would like to observe, which the prover will open.



However, this entire message will still send the entire proof, which isn't what we want.

Our solution is to use a Merkle 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?

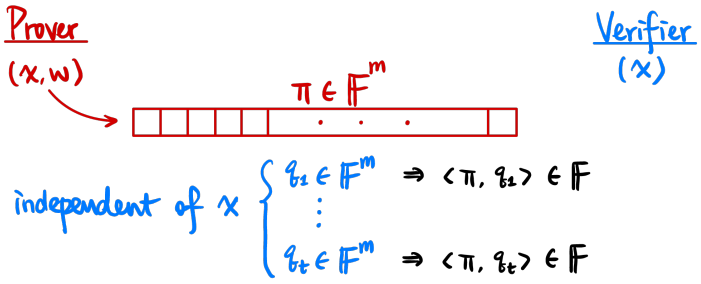
One solution would be to add a random string r as a sibling to every leaf. However, this would require us to reveal all siblings when we're verifying a certain leaf node. We can easily modify this to *salt every* leaf node. We can add some random r_i to the hash of *every* bit that hides those bits.

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 pathes 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. However, this is not practical.

§14.1.1 Linear PCP



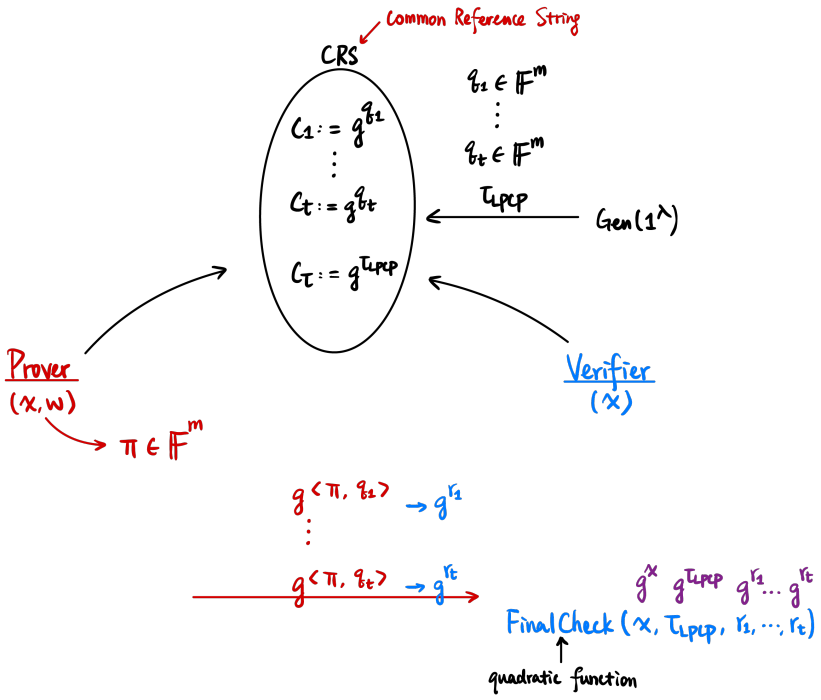
We can use a Linear PCP instead. The prover sends $\pi \in \mathbb{F}^m$, instead of checking a constant number of bits, we check a constant number of inner products $\langle \pi, q_i \rangle \in \mathbb{F}$. Constructions have size $m = O(|c|^2)$ for Walsh-Hadard codes or $m = O(|c|)$ for quadratic span programs.

The verifier will preprocess first by generating public and secret keys. The verifier encrypts challenges $q_i \in \mathbb{F}^m$ and sends $c_i \leftarrow \mathsf{Enc}_{pk}(q_i)$. The public and ciphertexts are published.

The prover will generate such $\pi \in \mathbb{F}^m$, and in an additively homomorphic encryption scheme, the prover can provide $\mathsf{Enc}_{pk}(\langle \pi, q_i \rangle) \to r_i$.

The verifier checks x, τ_{LPCP} and r_is in quadratic time.

However, this is *designated verifier*. Every verifier will need to generate their values q_i and c_i. This doesn't allow for us to publicly verify, and the preprocessing doesn't really allow for the random oracle model.



We can, however, use the common reference string (CRS) model. The assumption is that some trusted third party will generate a 'common reference string' $c_i = g^{q_t}$ and $c_\tau = g^{\tau_{\text{LPCP}}}$.

The prover can still multiply ciphertexts to get $g^{\langle \pi, q_i \rangle} = g^{r_i}$ (exponents are linear). Then the verifier can take $g^x, g^{\tau_{\text{LPCP}}}, g^{r_i}$ to verify.

But the verifier does not have the secret key! There are some nasty ways around this, but we'll use a piece of new technology in cryptography called *Bilinear Pairings*.

Definition 14.3 (Bilinear Pairing)

A Bilinear Pairing is a set of groups G_1, G_2, G_T with generators g_1, g_2, g_T and a function[46]

$$e : G_1 \times G_2 \to G_T$$

such that

$$e(g_1^a, g_2^b) = g_T^{ab}$$

Using this, we can verify without knowing secrets. Note that it's crucial that the CRS is generated correctly. This is usually done using MPC (multi-party computation) in a *key ceremony*[47]

[46] The target group should be different than the initial groups, but the first two groups can be the same. If all groups are the same then DDH is made easy.

[47] Zcash, at one point, had a bad vulnerability which exposed CRS privates.

§14.2 Secure Multi-Party Computation

§14.2.1 2-Party Computation

We've seen this before, but it is whensome parties want to compute the output of some function on their individual inputs, without revealing their own inputs.

Example 14.4

Alice and Bob just returned from a date, and want to figure out if they each want a second date. Alice has some choice bit x and Bob some choice bit y. They want to jointly compute $f(x, y) = x \wedge y$.

Example 14.5

Alice and Bob want to compare riches (who is richer?). They compute

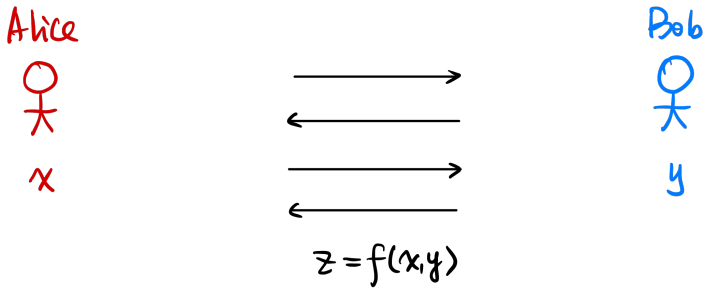
$$f(x, y) = \begin{cases} 0 & \text{if } x > y \\ 1 & \text{otherwise} \end{cases}$$

Example 14.6

Alice and Bob meet for the first time and want to see if they have friends in common. They have sets of friends X, Y, and compute

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

There are variants of this which only give cardinality of $X \cap Y$, etc.



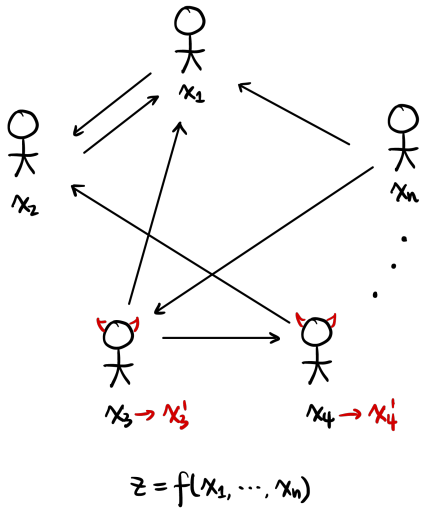
In general, this is when two parties have inputs x, y and want to compute some function $f(x, y)$ on them.

Use cases include:

- Password breach alert (Chrome/Firefox/Azure/iOS Keychain) runs a set intersection on your passwords and server leaked passwords.
- Privacy-preserving contact tracing for COVID-19 (Apple and Google). We want to know if we have contact but not who had contact with.
- Ads conversion measurements/personalized advertising (Google/Meta). We want to match conversions without either party knowing who converted.

§14.2.2 Multiple Parties!

The general case of this is Secure Multi-Party Computation (MPC)



This is when we have some parties P_i and want to compute input on $f(x_i, \cdots, x_n)$.

Here are some applications:

- A bunch missed
- Federated learning (used in Google Keyboard Search Suggestion). We want to run machine learning, federated amongst multiple devices. However, we don't want to leak the actual training data from users.
- Auctions (Danish sugar beet auction). Nobody should reveal their bid in the clear.
- Also deployed in Boston area to analyze the wage gap between genders without revealing the individual salaries.

Some applications are still in the works:

- Study/Analysis on Medical Data. Every institution has limited data, but they cannot openly share that data due to regulations. How could they jointly do analysis on this data without revealing the data.

- Fraud Detection (banks). Users might have cards at multiple banks, they want to jointly detect fraud but do not want to share their transactions.

When we normally talk about cryptography, we talk about 'slowing down' the system (crypto makes everything slower). In the case of MPC, though, we've enabled new features that were not otherwise possible without these tools.

§14.3 Definition

Our setting is that we have n parties $P_1, \ldots, P_n$ with private inputs $x_1, \ldots, x_n$. They want to jointly compute $f(x_1, \ldots, x_n)$.

In terms of communication infrastructure: we usually assume point-to-point channels between each pair (p_i, p_j). We know how to do this (key exchange, authenticated encryption, etc). Sometimes, we also assume a reliable broadcast channel where every other party gets information.

There is a single adversary that can "corrupt" a subset of the parties (at most t).

What properties do we want out of this system? Here are some common security properties we might want:

Correctness. The function is computed correctly.

Privacy. Only the output is revealed.

Independence of Inputs. Parties cannot choose their inputs depending on others' inputs.

Also with security guarantees:

Security with Abort. The adversary may "abort" the protocol. This prevents honest parties from receiving the output. This is the weakest model.

Fairness. If one party receives the output, then all parties will receive the output.

Guaranteed Output Delivery (GOD): Honest parties *always* receive output. Even if adversarial parties leave, the honest parties will simply continue the protocol.

We also have some characterizations of adversaries:

- Allowed adversarial behavior:

  - Semi-honest (or passive/honest-but-curious): They follow the protocol description honestly, but they try to extract more information by inspecting the transcript. This is the weaker model.

  - Malicious/active: These adversaries can deviate arbitrarily from protocol description.

- Adversary's computing power:

  - Unbounded computing power: this gives us information-theoretic (IT) security.

  - PPT bounded: this gives us computational security.

If you're interested, you can look into the literature of how to define security for MPCs. The idea is similar to that of ZK proofs—everything an adversary can do (see the transcript) can be simulated by a simulator who only has the input and output.

§15 March 21, 2023

Survey results: generally that course is well paced, some mentioned too slow or too fast. Seems to be a healthy in-between.

We'll also be having a guest speaker! They are from Google and have implemented MPC in real life.

Last time, we mentioned Bilinear pairings. It was left unresolved whether the target group can be the same group as the domain groups. We should have them be different. Specifically, this allows us to do a *single* multiplication in the exponent. If they are all the same group, we can do arbitrary polynomials in the exponent, which is not desired.

To do an arbitrary n number of equations is called a multilinear map. There are no known secure constructions of which.

§15.1 Secure Multi-Party Computation, *continued*

To quickly recap, a two-party computation is a computation where two parties want to jointly compute a function $f(x, y)$ on their private inputs x, y—but they do not reveal to each what their inputs are.

In the multi-party case, there will be n parties $P_1, P_2, \ldots, P_n$ with inputs $x_1, x_2, \ldots, x_n$ wishing to jointly compute $f(x_1, x_2, \ldots, x_n)$. We generally assume there are secure point-to-point channels, but some models assume broadcast channels. A single adversary can "corrupt" a subset of the parties, say t.

Here are properties we wish to attain in our protocol:

Correctness. The function is computed correctly.

Privacy. Only the output is revealed.

Independence of Inputs. Parties cannot choose their inputs depending on others' inputs.

Also with security guarantees:

Security with Abort. The adversary may "abort" the protocol. This prevents honest parties from receiving the output. This is the weakest model.

Fairness. If one party receives the output, then all parties will receive the output.

Guaranteed Output Delivery (GOD): Honest parties *always* receive output. Even if adversarial parties leave, the honest parties will simply continue the protocol.

§15.1.1 Feasibility Results

In the computational security setting, if we have a fundamental building block, a semi-honest oblivious transfer (OT), we can get semi-honest MPC for any function $t < n$. At a high level, using zero-knowledge proofs to enforce correctness of the protocol, we can convert any semi-honest MPC into a malicious MPC.

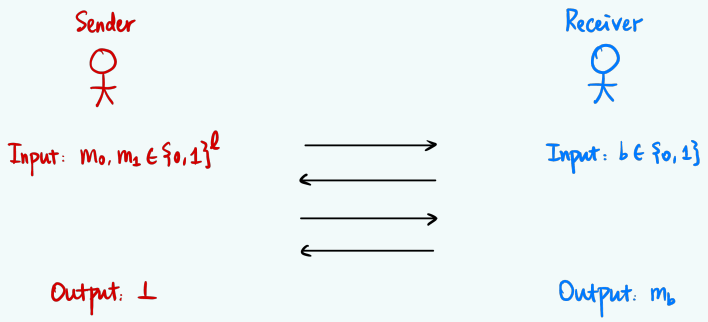
In terms of information-theoretic (IT) security. We can also get semi-honest and malicious MPC for any function with $t < \frac{n}{2}$. We call this an honest majority. This is a necessary bound, we cannot do any better than this.

§15.2 Oblivious Transfer

Definition 15.1 (Oblivious Transfer)

An oblivious transfer is a protocol in which a sender, with messages $m_0, m_1 \in \{0,1\}^l$ gives a choice to the receiver to receive either m_0, m_1.

Given a choice bit from the receiver $b \in \{0,1\}$, the receiver gets m_b and the sender also gets no information about the messeage transferred.



We'll learn about constructions of OT later, but we black-box its implementation until later.

Using a semi-honest OT, we can use Yao's Garbled Circuit to construct semi-honest 2PC for any function. We can also use the GMW compiler to compile this into a semi-honest MPC for any function. We'll focus on the first approach in this lecture, but we'll learn GMW in the following lectures.

§15.3 Yao's Garbled Circuit

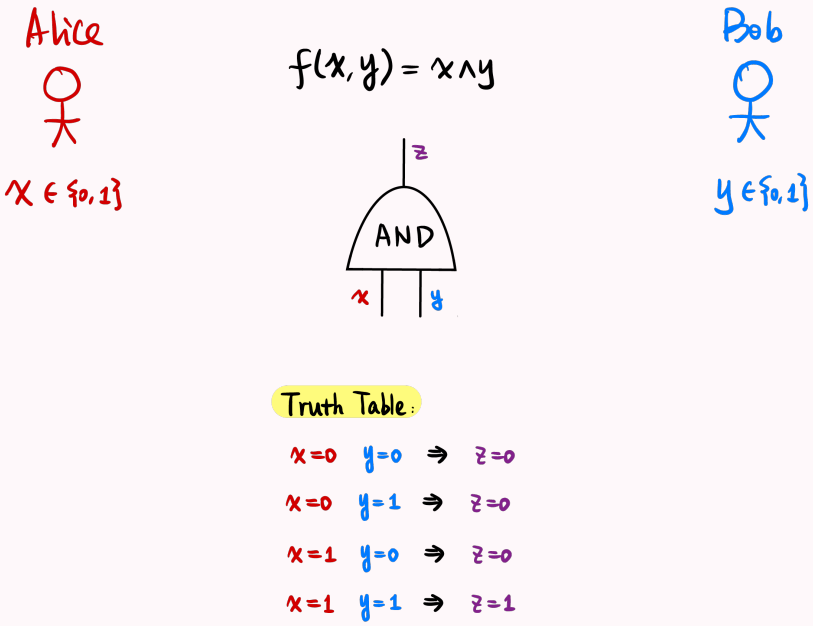
Example 15.2 (Private Dating/AND Gate)

Alice and Bob want to figure out whether they want to go on a second date. Alice has single

bit $x \in \{0,1\}$, and Bob also has single bit $y \in \{0,1\}$.

They want to compute a single AND gate.

Alice will *garble* circuit wires by generating some random l_0, l_1 for each wire corresponding to each bit possibility. We call these *labels*.



For each AND gate[48], she'll generate 4 ciphertexts,

$$\begin{aligned}
&\mathsf{Enc}_{\alpha_0}(\mathsf{Enc}_{\beta_0}(0))\\
&\mathsf{Enc}_{\alpha_0}(\mathsf{Enc}_{\beta_1}(0))\\
&\mathsf{Enc}_{\alpha_1}(\mathsf{Enc}_{\beta_0}(0))\\
&\mathsf{Enc}_{\alpha_1}(\mathsf{Enc}_{\beta_1}(1))
\end{aligned}$$

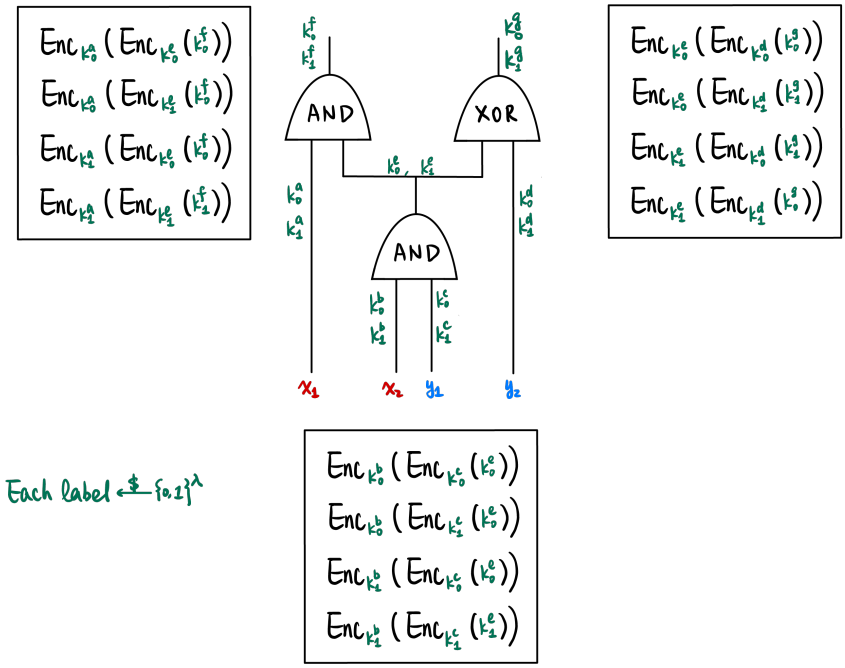
If we have some α_a, β_b, then we can decrypt $\mathsf{Enc}_{\alpha_a}(\mathsf{Enc}_{\beta_b}(\cdots))$ and all other ciphertexts will look like garbage (we gain no information). This is to say, we can only decrypt the ciphertext of the keys we know. The overarching idea is that we'll only know the right labels for our inputs.

Alice will send the circuit (the 4 encryptions) as well as the input label for x, α_a. Bob now needs to get the label correponding to his input wire, β_0, β_1. We can perform an oblivious transfer!

Bob has a choice bit and gets one of β_0, β_1 without Alice knowing his choice bit. Having attained β_b, Bob will try the encryption on all 4 ciphertexts with α_a, β_b, and sees which output is valid and returns that.

For Alice to learn this output, Bob will send the output back to Alice. In the semi-honest setting, Bob will honestly send the result back to Alice. In the malicious case, we might require Bob to provide some zero-knowledge proof in the end to prove that their plaintext result came from their circuit.

We'll generatlize this single-gate computation for arbitrary functions. We'll represent any arbitrary function as a boolean circuit consisting of only AND and XOR gates[49].



Every wire gets two labels, corresponding to a 0 bit or 1 bit. Each label is $\xleftarrow{\$} \{0,1\}^\lambda$. For each gate, we construct a 'mini' garbled table, where the encrypted message is the output 0 or 1 labels[50,51]. We vary the encryptions based on the gate we're trying to implement.

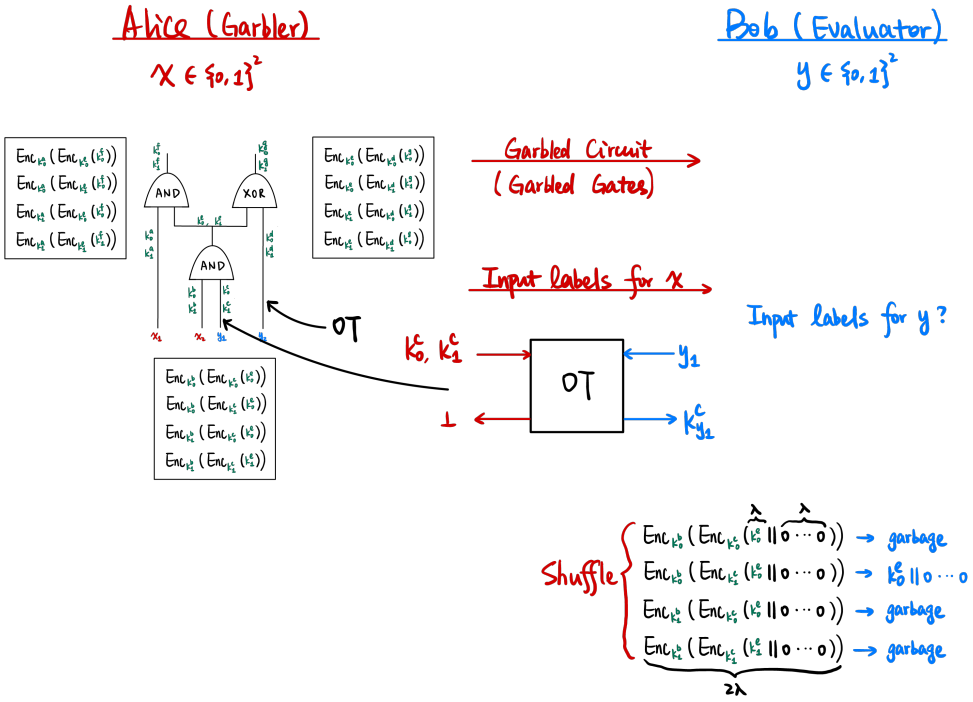
Using the garbled circuit, we can construct an arbitrary 2PC. We call the party who generates the circuit the 'garbler', and the other party the 'evaluator'.

[48] We can change the values depending on the different logic gates.

[49] Recall that any boolean circuit can be represented using only AND and XOR gates.

[50] *How will we know which is garbage?* Naïvely, we could just try every label. However, this is an exponential blowup for every gate we run the labels through. The solution is to attach a bitstring 'tag' (could just be a string of 0s) that indicates whether a decryption is indeed a label.

[51] One more subtle thing we should take care of! We show our ciphertexts in order of $00, 01, 10, 11$. This reveals information! We should take care to shuffle the ciphertexts everywhere.



Alice garbles the circuit, and sends it to Bob. Alice can easily send her own labels. For the labels corresponding to Bob's input, we run oblivious transfer for each input wire to get Bob's input bits without Alice knowing.

In the final output, we can encrypt plaintext $0, 1$. The other way is for Alice to send the final random labels to Bob along with their corresponding bits.

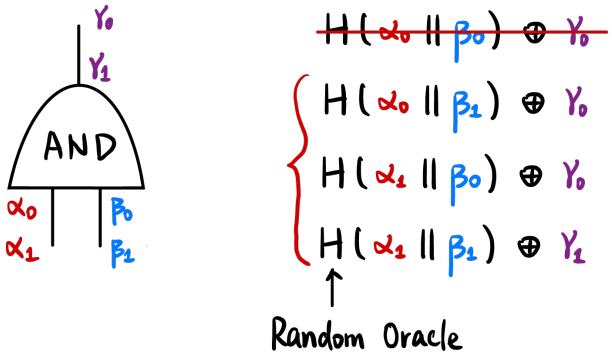
§15.3.1 Optimizations

There are some optimizations we can make:

Point-and-Permute. For each wire, we'll randomly sample signal bits $\sigma_\alpha, \sigma_\beta$, and flip it for the other input. (Note that this doesn't reveal anything about α, β). In the circuit, we can indicate using the signal bit which ciphertext to decrypt.

We reduce Bob's computation complexity by at least a constant of 4, and saves communication complexity by half (we don't need to expand our garbled circuit size anymore).

Row Reduction. In this construction, there are 4 ciphertexts per gate. We can just hash the labels and XOR with the corresponding output label (this is not CPA-secure, but that is fine). From the 4 ciphertexts, we can set γ_0 to exactly the hash $H(\alpha_0||\beta_0)$[52]. This is compatible with point-and-permute. We hide every row, which is fine. This gives us a $\frac{3}{4}$ space decrease.



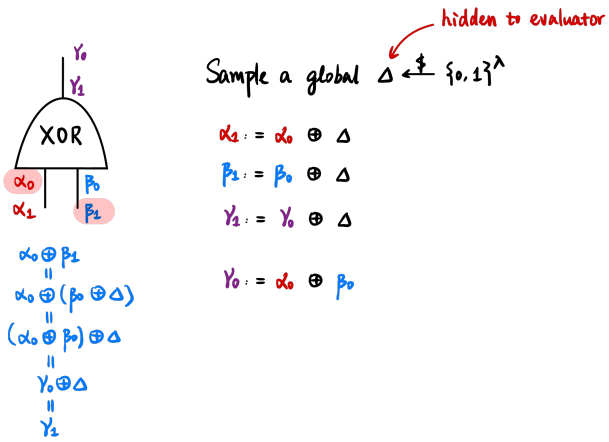
Free XOR. Sample a global $\Delta \xleftarrow{\$} \{0,1\}^\lambda$. Every pair of labels differ by Δ. That is,

$$\begin{aligned}\alpha_1 &:= \alpha_0 \oplus \Delta \\ \beta_1 &:= \beta_0 \oplus \Delta \\ \gamma_1 &:= \gamma_0 \oplus \Delta\end{aligned}$$

and $\gamma_0 = \alpha_0 \oplus \beta_0$. To compute the output label, you just perform the XOR plainly.

This is to say, XOR is free. We don't need to send labels and Bob doesn't need to encrypt/decrypt.

[52] Not really the $0, 0$ labels, but they can correspond to the signal bits.



We can also use *half-gates* which give us 2λ bits per AND gate + free XOR. A recent development, *slicing-and-dicing*, gives us around $\sim 1.5\lambda$ bits per AND gate + free XOR.

§16 March 23, 2023

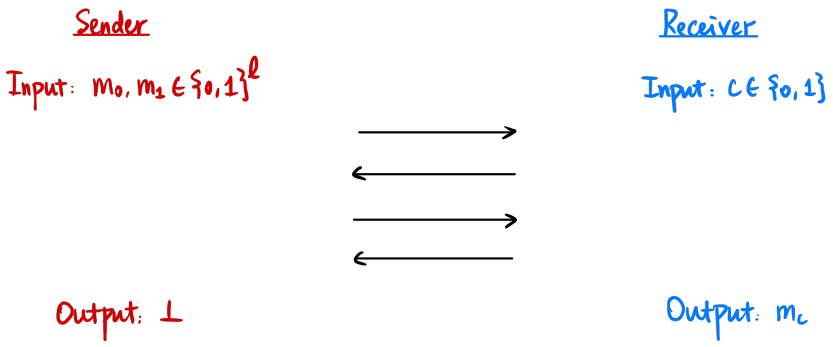
Some quick notes: if you submit homework after your alloted late allotment, we will still grade it, but it will not count toward your grade.

Additionally, we're seeing some responses that look like they were from chatbots like ChatGPT. They *will not* produce the correct responses and are definitely a violation of academic code. This has been placed in the syllabus.

§16.1 Oblivious Transfer

We saw last time how to construct Yao's garbled circuit using oblivious transfer, but we black boxed the implementation of OT.

We'll go over the implementation of semi-honest OT here. It will follow similarly to the Diffie-Hellman key exchange.

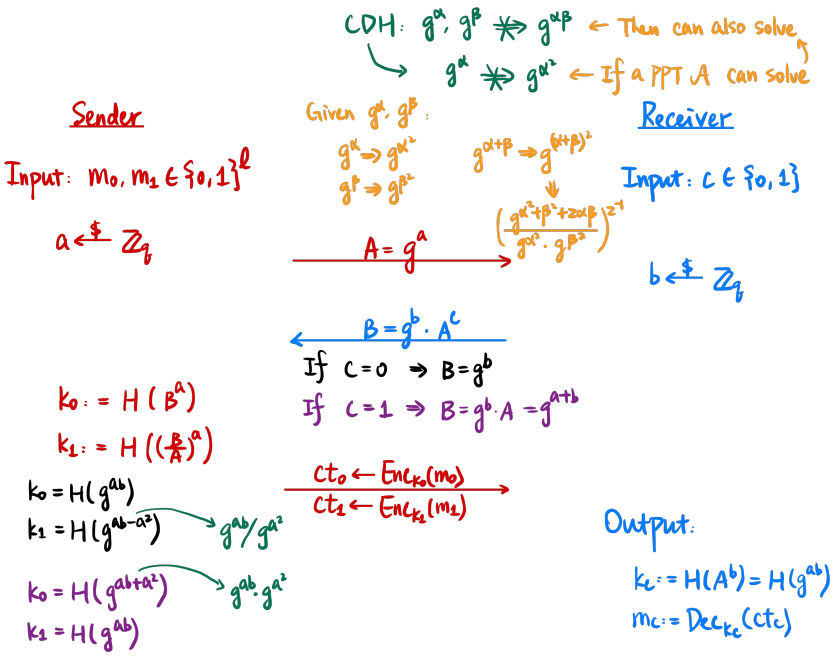


The sender will send $A = g^a$. The receiver will mask A^c with $c \in \{0, 1\}$ and $b \xleftarrow{\$} \mathbb{Z}_q$. a, b here are like Diffie-Hellman privates.

Then, the sender will compute $k_0 := H(B^a), k_1 := H\left(\left(\frac{B}{A}\right)^a\right)$. This means that k_c will be exactly $g^{ab} = A^b$ (whether $c = 0$ or $c = 1$). Then, k_0 and k_1 will be used to encrypt m_0, m_1 respectively.

Since only one will be the shared Diffie-Hellman key (and the other will require knowledge of a), the receiver will only be able to reveal one such message.

Doing out the algebra, we can conclude that the receiver can access the key.



Is this secure against a semi-honest receiver? If the key is $c = 0$, then the other key will be g^{ab-a^2}. g^{a^2} is difficult to compute, since the receiver only has $A = g^a$ and will need secret a to compute g^{a^2}.[53] If $c = 1$, then the other key will be g^{a^2+ab} which is hard again. So, for the receiver, it is computationally secure.

Is this secure against a semi-honest sender? g^b is a random mask on A^c, so the sender will not be able to distinguish between this.

§16.1.1 OT Extension

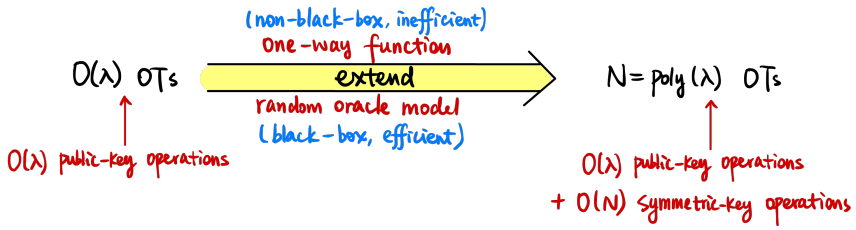
We used public-key operations to achieve our OT. Is it possible to construct OT only using symmetric-key primitives? Unlikely...

There are impossibility results that show that if we assume $\mathsf{P} \neq \mathsf{NP}$, it's not possible to construct an OT using symmetric-key primitives.

This makes OTs very difficult—since it takes an entire protocol (including expensive exponentiations) to transfer one bit. There has been current research in *extending* OT so we can use more bits.

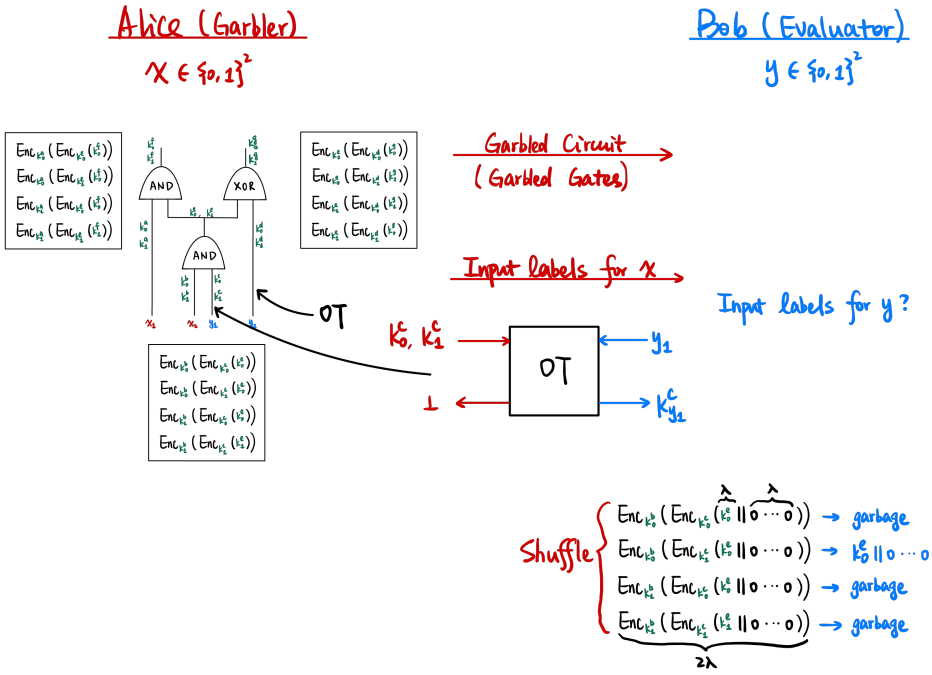
An OT extension can extend $O(\lambda)$ OTs (with $O(\lambda)$ public-key operations) into a $\text{poly}(\lambda)$ bit OTs.

[53] Formally, this security is guaranteed by the CDH assumption, that if we have g^α, g^β, it's computationally hard to determine $g^{\alpha\beta}$. If an adversary can derive g^{α^2} from g^α, they can also derive $g^{\alpha\beta}$. We can get $g^{\alpha^2}, g^{\beta^2}$, then we can get $g^{(\alpha+\beta)^2} = g^{\alpha^2+2\alpha\beta+\beta^2}$ and taking inverses we can peel off the α^2, β^2 exponents to get $g^{\alpha\beta}$.



§16.2 Putting it Together: Semi-Honest 2PC

We can now construct our 2PC protocol. Alice, the garbler, will create the circuit with garbled inputs and wires (shuffling order of ciphertexts). Alice sends this circuit to Bob, and Bob will use OTs with his input bits to get the wire labels that he should use. Then, Bob runs these labels on the garbled circuit.



In the semi-honest case, Alice will generate this circuit correctly and Bob will follow the protocol correctly. *What could go wrong against malicious adversaries?*

- Alice could garble an incorrect gate, or give an entirely incorrect circuit.
- Alice could refuse to send the result (translate output label to bits) back to Bob, or send an incorrect result to Bob. If the outputs are not garbled, then Bob could similarly refuse to send this back to Alice.
- Alice and Bob could both cheat about their inputs.

§16.3 GMW

We can convert this into a MPC for any function with $t \leq n-1$ (corrupted parties up to all but one).

Throughout the protocol, we keep the invariant that for each wire w, if the value of the wire is $v^w \in \{0,1\}$, then the parties hold an additive secret share of v^w. Each party P_i holds a random share $v_i^w \in \{0,1\}$ such that

$$\bigoplus_{i=1}^{n} v_i^w = v^w$$

and we keep this invariant throughout the entire circuit.

We need to be able to preserve this invariant throughout AND and XOR gates. The XOR case is easy, since XOR is completely commutative and associative, so each party can locally XOR their shares $c_i := a_i \oplus b_i$ for $c := a \oplus b$.

We'll wave our hands over the AND case, but we can do this. We'll proceede gate-by-gate for everyone to compute the result. Each party will publish their local shares, and everyone will XOR the result together to get the final result.

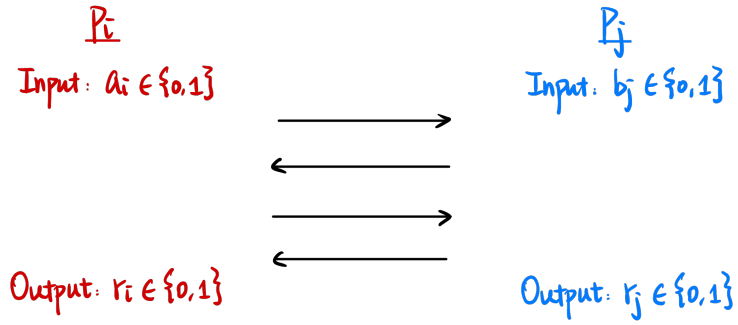
§16.3.1 AND Gates

We now finish addressing the AND gates. We have $\bigoplus_{i=1}^{n} a_i = a$ and $\bigoplus_{i=1}^{n} b_i = b$.

We want a set of $\{c_i\}$ s.t. $\bigoplus_{i=1}^{n} c_i = c = a \cdot b$ (multiplication of bits is AND). But

$$\begin{aligned} a \cdot b &= \left(\sum_{i=1}^{n} a_i\right) \cdot \left(\sum_{i=1}^{n} b_i\right) \pmod 2 \\ &= \left(\sum_{i=1}^{n} a_i \cdot b_i\right) \cdot \left(\sum_{i \neq j} a_i b_i\right) \pmod 2 \end{aligned}$$

The first sum is easy and computed locally, but the second sum requires parties to communicate. We do something called resharing.



Between P_i, P_j, we want random $r_i, r_j \in \{0,1\}$ such that $r_i + r_j = a_i \cdot b_j \pmod 2$. P_i will randomly sample $r_i \xleftarrow{\$} \{0,1\}$. We can use OT (!) to allow P_j to learn r_j such that $r_i + r_j = a_i \cdot b_j \pmod 2$ without revealing a_i or r_i.

P_i will be the sender, P_j is the receiver. P_j's choice bit is b_j. Then the messages will be

$$m_0 = (a_i \cdot 0) - r_i$$
$$m_1 = (a_i \cdot 1) - r_i$$

such that r_i, r_j are two shares of $a_i \cdot b_j$.

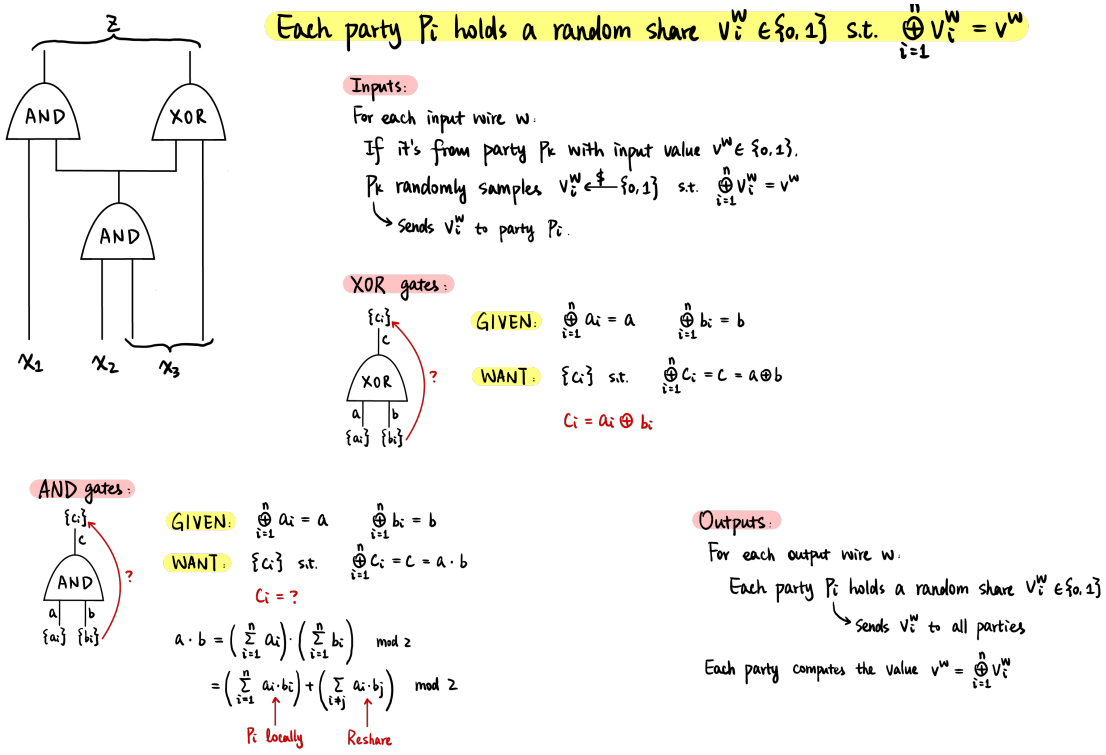
§16.3.2 Complexities

What is the computational complexity for each party? Computational complexity is $O(\#\mathsf{AND} \cdot n)$ for each party. And for communication, every pair of parties needs to communicate for every AND gate, so $O(n^2 \cdot \#\mathsf{AND})$.

The round complexity is the depth of the circuit, *only counting AND gates* (we can ignore XOR gates).

§16.3.3 Entire Protocol

Here's our entire protocol:



We now have a secure multi-party computation scheme. How might we compare them?

- Yao's Garbled Circuit
  - Malicious security lower overhead
- Goldreich-Micali-Wigderson (GMW)
  - The number of OTs is $\#\mathsf{AND} \cdot n^2$.

§17 April 4, 2023

§17.1 Secure Multiparty Computation, *continued*

§17.1.1 Comparing Yao's and GMW

We'll briefly recap what we covered before the break.

Recall we have the Yao's Garbled Circuit for semi-honest 2PC for any function (see section 16.2), and the GMW compiler for semi-honest MPC for any function (see section 16.3). They all rely on the semi-honest OT. We've been focusing on the semi-honest case, which is that parties will not deviate from protocol but can try to extract more information.

In the case of Yao's Garbled Circuit, one or the other party could behave maliciously, sending the wrong circuit, etc.

In the case of GMW, we discussed last class the complexity and flaws against malicious attackers. The complexity is generally the depth of AND gates, since XOR gates can be computed offline.

We started to compare Yao's with GMW last time. Malicious security for Yao's garbled circuit relies on security against Alice, since there is not much that Bob can do. We just need to protect against malicious Alice.

The advantages for Yao's is that it offeres malicious security with lower overhead (since we only need to protect against malicious Alice), and we only require OTs equal the number of inputs. On the other hand, the GMW protocol requires $\#\mathsf{AND} \cdot n^2$.

The computational complexity of Yao's and GMW is approximately the same. In GMW, we have $O(\#\mathsf{AND} \cdot n)$ for each party, and using free XOR we can also get $O(\#\text{input})\text{OTs} + O(\#\mathsf{AND})\text{AES}$ in Yao's. Communication complexity is also roughly the same at $O(\#\mathsf{AND} \cdot n^2)$ for GMW and $O(\#\text{input} + \#\mathsf{AND})$ for Yao's. However, the round complexity is constant in Yao's since it only requires a back-and-forth, while GMW requires resharing on depth of AND gates. This is a big advantage.

Is there any advantage for GMW? It supports any number of parties, while Yao's is solely for two parties. Additionally, GMW is extentable to arithmetic circuits (XOR becomes addition and AND becomes multiplication in some $\mathbb{Z}_p$), while Yao's is limited to bits.

> **Remark.** There's a fun story. The Yao's garbled circuit papers came out in 1982 and 1986 with about the same number of citations. Papers back in the days talked about these in high-level.
>
> The GMW paper came out in 1987.
>
> It becomes a question which Yao paper to cite—it was determined that it would be better to cite the 1986 paper since it then seems like GMW was developed just a year later, instead of having taken researchers 5 years to devise another protocol.

§17.1.2 GMW Malicious Security

We currently have semi-honest MPCs in both cases. We can apply a general compiler to convert our semi-honest protocol into a malicious protocol.

Given a semi-honest protocol (GMW), once inputs and randomness are fixed, then the protocol becomes entirely *deterministic*. We have two steps:

1. Each party P_i commits to its input x_i and randomness r_i to be used in the semi-hoenst protocol.

2. We run the semi-honest protocol. Along with every message, prove in zero-knowledge that the message has been computed correctly (based on its input, randomness, as well as transcript so far).

Are there any issues with this protocol? Parties can 'fake' their randomness that benefits them. Here are some proposed solutions:

1. While we could use a hash function (with the random oracle model) to generate the randomness, we want to hide the randomness instead of showing—but then we wouldn't be able to verify that the randomness was generated properly.

2. We could also use a common randomness model, but this exposes the same issue that the randomness is not shown.

3. We want to generate something that is not just contributed by the party themselves, but also by some external randomness. The idea is for every party to compute some randomness for you r_i^j (P_j providing shares for P_i), and your randomness is the XOR of everyone's randomness that they contributed to you.

This is a very expensive protocol. For every step, you need to be proving in zero-knowledge that the message is being computed correctly.

§17.1.3 Yao's Malicious Security

It's much easier to achieve malicious security for Yao's garbled circuit, since we only need to prevent against a malicious garbler. We'll employ a cut-and-choose model.

As an initial approach, Alice will garble λ circuits, and send those to Bob. Bob will randomly pick all but one for Alice to reveal, and will evaluate on the last garbled circuit.

If Alice generates 1 bad garbled circuit, then the probability that Alice is caught is $\frac{\lambda-1}{\lambda} = 1 - \frac{1}{\lambda}$. Which is a small probability, but it's not overwhelming.

Instead, Alice will generate 2λ garbled circuits, and Bob will pick λ circuits. Alice will reveal the

selected λ circuits, and Bob will evaluate on the remaining λ and take a majority. What's the probability that Bob will run a bad circuit. What, then, is the probability that Alice gets caught given that Alice generated $\leq \frac{\lambda}{2}$ garbled circuits incorrectly[54].

The probability that Alice passed is

$$\Pr[\text{Alice Passed}] = \frac{2\lambda - \frac{\lambda}{2}}{2\lambda} \cdot \frac{2\lambda - \frac{\lambda}{2} - 1}{2\lambda - 1} \cdots \frac{\frac{\lambda}{2} + 1}{\lambda + 1} \geq \left(\frac{3}{4}\right)^{\lambda}$$

which is negligible. Then, Alice will fail with high probability.

Cut-and-choose OT follows similarly, and should be used. The idea is that cut-and-choose will allow you to convert a semi-honest protocol into a malicious-secure one.

§17.2 Specialized MPC

We've so far only seen generalized MPC. We can construct efficient protocols for specific MPC. Recall the Private Set Intersection (PSI) problem from **??** .

Alice and Bob want to compute the intersection of their sets $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \cdots, y_n\}$. We extend Alice to have some weights $V = \{v_1, v_2, \ldots, v_n\}$. There are multiple problems under this class:

- PSI: $f(X, Y) = X \cap Y$.
- PSI-Cardinality: $f(X, Y) = |X \cap Y|$ which counts the number of items in the intersection without revealing the items.
- PSI-Sum: $f(X, Y) = |X \cap Y|, \sum_{i:x_i \in Y} v_i$ which adds up the weights from V of those elements that intersect. This is useful in ads conversion settings (and returns the cardinality too).

§17.2.1 Naïve Solution

Here's a naïve solution: Alice and Bob hash all their inputs, exchange hash values, and see which ones they have in common. You can learn the intersection, but could you learn *more* than that?

If the input space is a relatively small space, we can do a dictionary attack (for example, with names). This is not even semi-honest secure.

Can we even achieve 2PC/MPC with just a single round of communication (as we have done here, sending hashes one way)? Taking a step back, we receive a message from Alice and can derive the solution regardless of *any* y we have. This allows us to just test multiple inputs on the function received and we'll receive that output, so we can derive x from that input.

[54] If she generated less than this, then the majority will be correct anyways.

§17.2.2 DDH-Based PSI

We start with a cyclic group $\mathbb{G}$ of order q with generator g, where DDH holds. We also have a hash function $H : \{0,1\}^* \to \mathbb{G}$ (modeled as a random oracle).

Bob and Alice generate private exponents $k_B \xleftarrow{\$} \mathbb{Z}_q, k_A \xleftarrow{\$} \mathbb{Z}_q$ respectively. Bob will send

$$H(Y)^{k_B} := \left\{ H(y_1)^{k_B}, \cdots, H(y_n)^{k_B} \right\}$$

Alice does the same for X, $H(X)^{k_A}$, and sends $H(Y)^{k_A \cdots k_B}$. Bob then raises $H(X)^{k_A}$ to k_B and compares. In the semi-honest case, for Bob to perform the same dictionary attack, Bob will need to break DDH in order to raise arbitrary elements y' to $k_A \cdot k_B$.

We can modify this to count cardinality by randomly shuffling the returned $H(Y)^{k_A \cdot k_B}$ such that Bob cannot relate the re-encrypted hashes to the previous order.

We can also think about how we can achieve this for PSI-Sum.