CSCI 1515: Applied Cryptography

P. Miao

Spring 2026

These are lecture notes for CSCI 1515: Applied Cryptography taught at Brown University by Peihan Miao in the Spring of 2026.

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

Notes last updated February 23, 2026.

Contents

1 January 21, 2026 **4**
1.1 Introduction . . . 4
1.1.1 Staff . . . 4
1.1.2 Course Philosophy and Logistics . . . 4
1.2 What is cryptography? . . . 5
1.3 Secure Communication . . . 6
1.3.1 Message Secrecy . . . 7
1.3.2 Message Integrity . . . 9
1.3.3 Projects . . . 11
1.4 Zero-Knowledge Proofs . . . 11
1.5 Secure Multi-Party Computation . . . 13
1.6 Fully Homomorphic Encryption . . . 16
1.7 Further Topics . . . 17
1.8 Q & A . . . 18

2 January 28, 2026 **19**
2.1 Encryption Scheme Basics . . . 19
2.1.1 Syntax . . . 20
2.1.2 Symmetric-Key Encryption Schemes . . . 21
2.1.3 Public-Key Encryption Schemes . . . 27
2.1.4 RSA . . . 30

3 January 30, 2026 **32**
3.1 Basic Number Theory, *continued* . . . 32
3.2 RSA Encryption, *continued* . . . 32
3.3 Intro to Group Theory . . . 34
3.4 Computational Assumptions . . . 35
3.5 ElGamal Encryption . . . 36
3.6 Secure Key Exchange . . . 37
3.7 Prime Order Subgroups . . . 38

4 February 2, 2025 **39**
4.1 Message Integrity . . . 39
4.1.1 Message Authentication Code . . . 39
4.1.2 Digital Signature . . . 40
4.1.3 Syntax . . . 41
4.1.4 Chosen-Message Attack . . . 41
4.2 RSA Signatures . . . 43
4.2.1 Other Signature Schemes . . . 43
4.3 A Summary So Far . . . 44
4.4 Authenticated Encryption . . . 44
4.4.1 Encrypt-and-MAC? . . . 46
4.4.2 MAC-then-Encrypt? . . . 47
4.4.3 Chosen Ciphertext Attack Security . . . 47
4.4.4 Encrypt-then-MAC . . . 48

5 February 04, 2025 **50**
5.1 Hash Function . . . 50
5.2 Collision-Resistant Hash Function (CRHF) . . . 51
5.2.1 Random Oracle Model . . . 51
5.2.2 Constructions for Hash Function . . . 52
5.2.3 Applications . . . 53
5.3 Putting it Together: Secure Communication . . . 55
5.3.1 Diffie-Hellman Ratchet . . . 55
5.4 Block Cipher . . . 56
5.4.1 Pseudorandom Function (PRF), *continued* . . . 57
5.4.2 Pseudorandom Permutation (PRP) . . . 58

6 February 9, 2025 **60**
6.1 Recap . . . 60
6.2 Practical Constructions of Block Ciphers . . . 60
6.2.1 History of AES and DES . . . 60
6.2.2 Substitution-Permutation Network (SPN) . . . 60
6.2.3 Attacks on Reduced-Round SPN . . . 62
6.2.4 Feistel Network . . . 63
6.2.5 Data Encryption Standard (DES) . . . 63
6.3 Practical Constructions of Hash Functions . . . 64
6.4 Block Cipher Modes of Operation . . . 65

7 February 11, 2026 **66**
7.1 More Block Cipher Modes of Operation 66
7.1.1 CBC-MAC 70
7.1.2 Encrypt-last-block CBC-MAC (ECBC-MAC) 70
7.2 Putting it Together 72

8 February 18, 2026 **76**
8.1 One-Sided Secure Authentication 76
8.1.1 Certificate Chain 77
8.2 Password-Based Authentication 78
8.2.1 Salting 79
8.2.2 Two-Factor Authentication 80

§1 January 21, 2026

§1.1 Introduction

The course homepage is at https://cs.brown.edu/courses/csci1515/spring-2026/, where you can find information such as the syllabus, projects, homeworks, calendar, lectures and more.

The course is offered in-person in *Friedman 108*, as well as synchronously over Zoom and recorded asynchronously (lectures posted online). Lecture attendance and participation is highly encouraged!

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

§1.1.1 Staff

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

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

Our course staff all have strong backgrounds in cryptography and are excited to help you learn!

§1.1.2 Course Philosophy and Logistics

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

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

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

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

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

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

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

Capstone requirements are cancelled in FY25–26 for CS and joint-CS concentrators.

The following is the grading policy:

Type	*Percentage*
Introductions	1%
Project 0	4%
Projects 1 & 2 & 5	24% (8% each)
Projects 3 & 4	24% (12% each)
Code Review	8% (4% each)
Homeworks	25% (5% each)
Final Project	14%

You have a total of four (4) late days that you can use to extend the due dates of *Projects 0–5* free of charge, but at most two (2) late days may be applied to any one project. Beyond that, you will be penalized 40% of the project's value for each day it is late. There are **no** late days for *homeworks*, as we want to release the solution guide as soon as possible.

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

If you're sick, let Peihan know with a Dean's note. For a full description of the late policy, refer to the syllabus.

§1.2 What is cryptography?

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

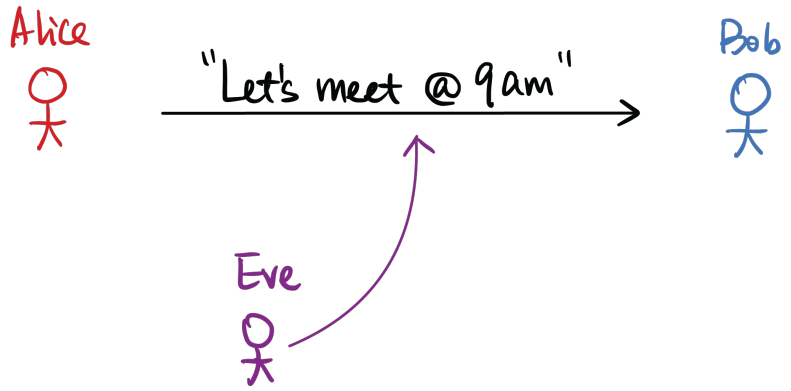
Question. Where is cryptography used in practice? What guarantees do we want in these scenarios?

- Online transactions

  - When you make a purchase, you might not want people to see your bank balance, what else you have purchased, etc.
  - You also want to ensure that it was really *you* who purchased the item and not somebody else i.e. authentication
- Secure messaging
  - End-to-end texting, iMessage
  - We don't want anyone else to see our messages
- Online voting
  - Privacy of votes, validity of votes
- Databases
  - Secure storage
- There is much more!

§1.3 Secure Communication

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



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

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

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

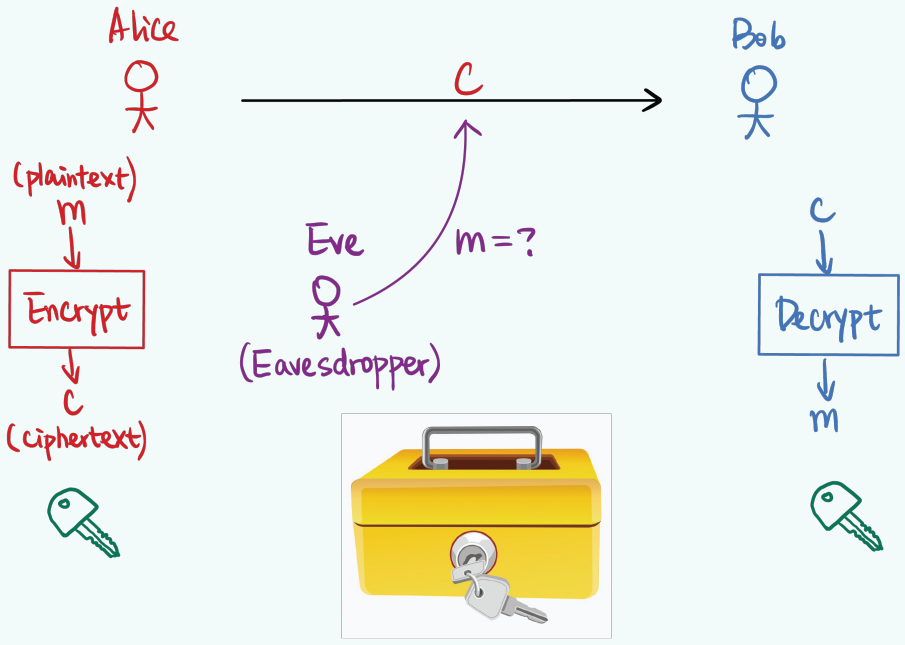
These two guarantees are the most important guarantees! The former is called message secrecy, the latter is called message integrity.

§1.3.1 Message Secrecy

Definition 1.1 (Message Secrecy)

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

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

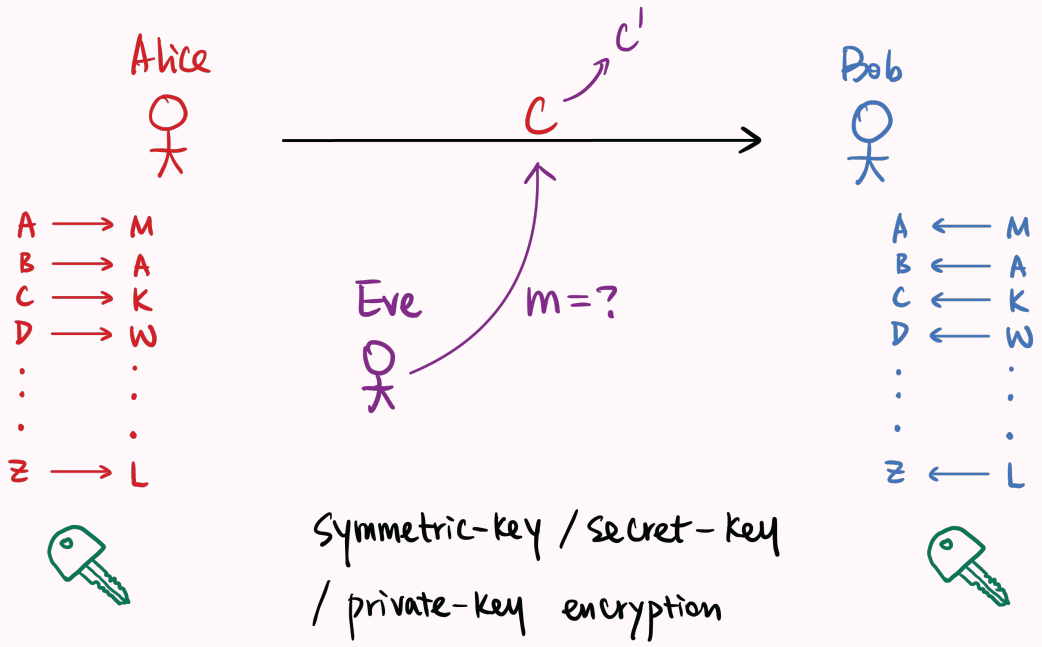


In this model, Eve is a weaker adversary, an *eavesdropper*. Eve can only see the message, not alter it.

Example 1.2 (Substitution Cipher)

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

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



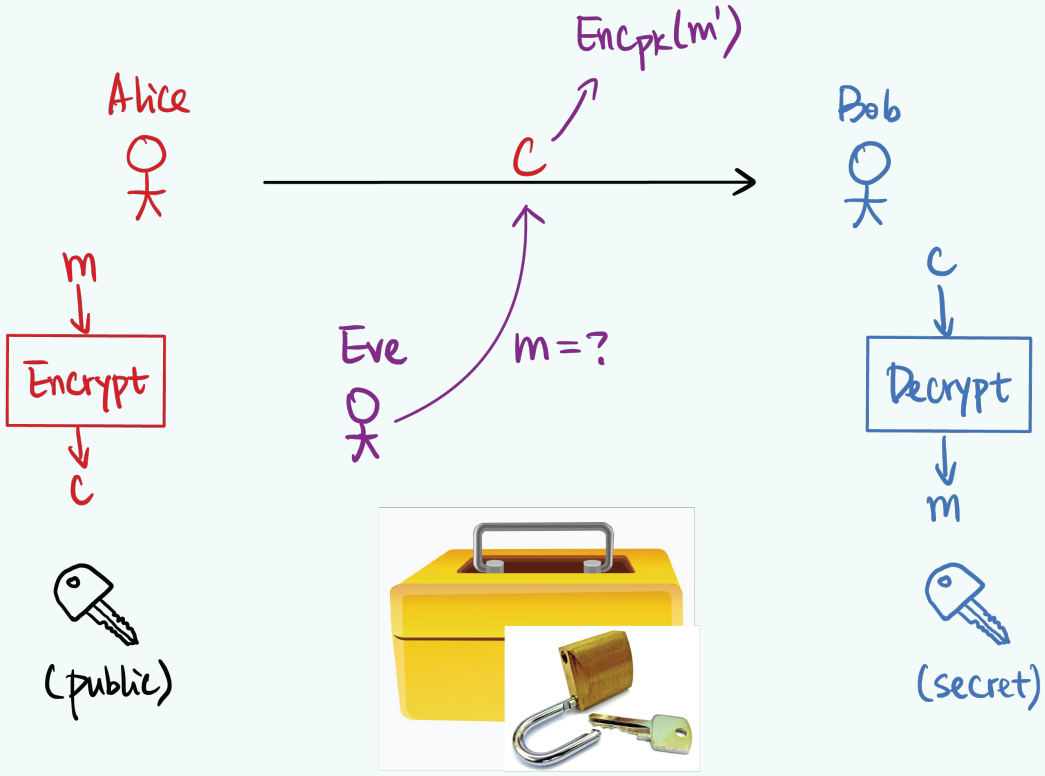
This scheme is not quite secure! *Why?*

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

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

Definition 1.3 (Public-key Encryption)

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

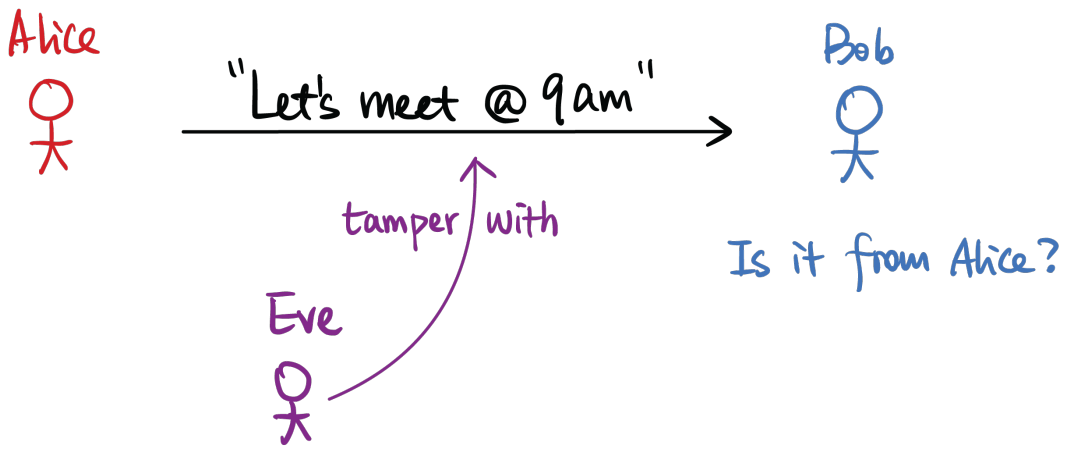


This is seemingly magic! Bob could publish a public key on his homepage, anyone can encrypt using a public key but only Bob can decrypt. *Stay tuned, we will see public-key encryption schemes next lecture!*

§1.3.2 Message Integrity

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

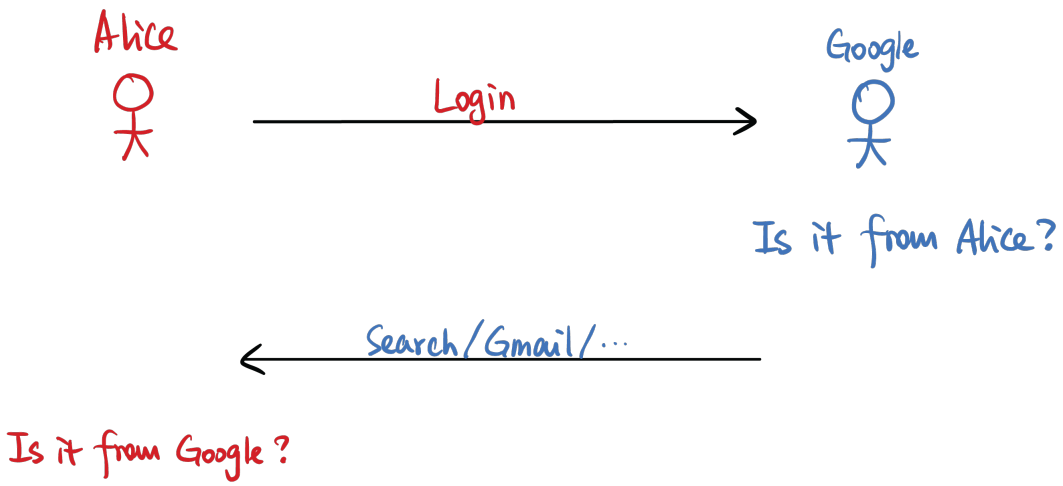
[1] You literally click it closed



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

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

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



This property that we want is called message integrity.

§1.3.3 Projects

Projects Overview

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

We'll now introduce the latter three projects!

§1.4 Zero-Knowledge Proofs

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

For example, Alice may want to

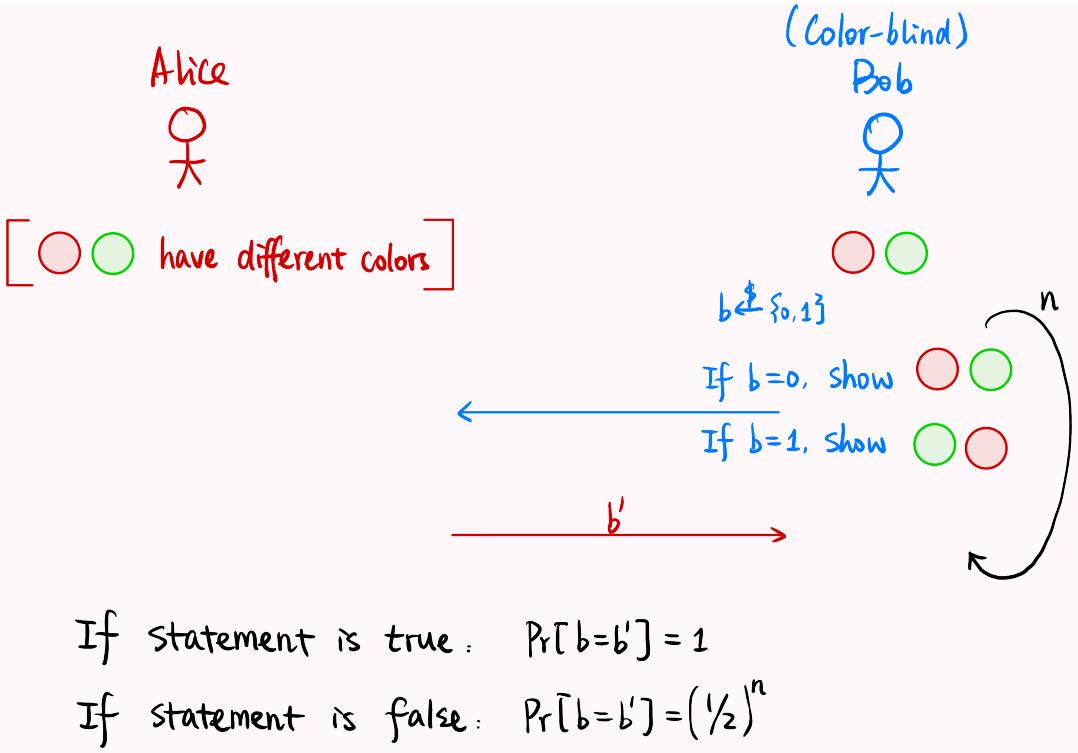
- Prove she knows the difference in taste between Coke and Pepsi without revealing how
- Prove that you have a bug in your code without revealing the bug
- She has the secret key for this ciphertext without revealing the plaintext
- Prove that she owns two different colored pens to her colorblind friend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

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

Example

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

Bob will randomly sample a bit $b \xleftarrow{\$} \{0,1\}$, with $b = 0$ being "Stay" and $b = 1$ being "Swap". Bob will then either swap the order of the markers and show them to Alice, or keep them the same. Alice will give a guess b' of whether Bob swapped or not.



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

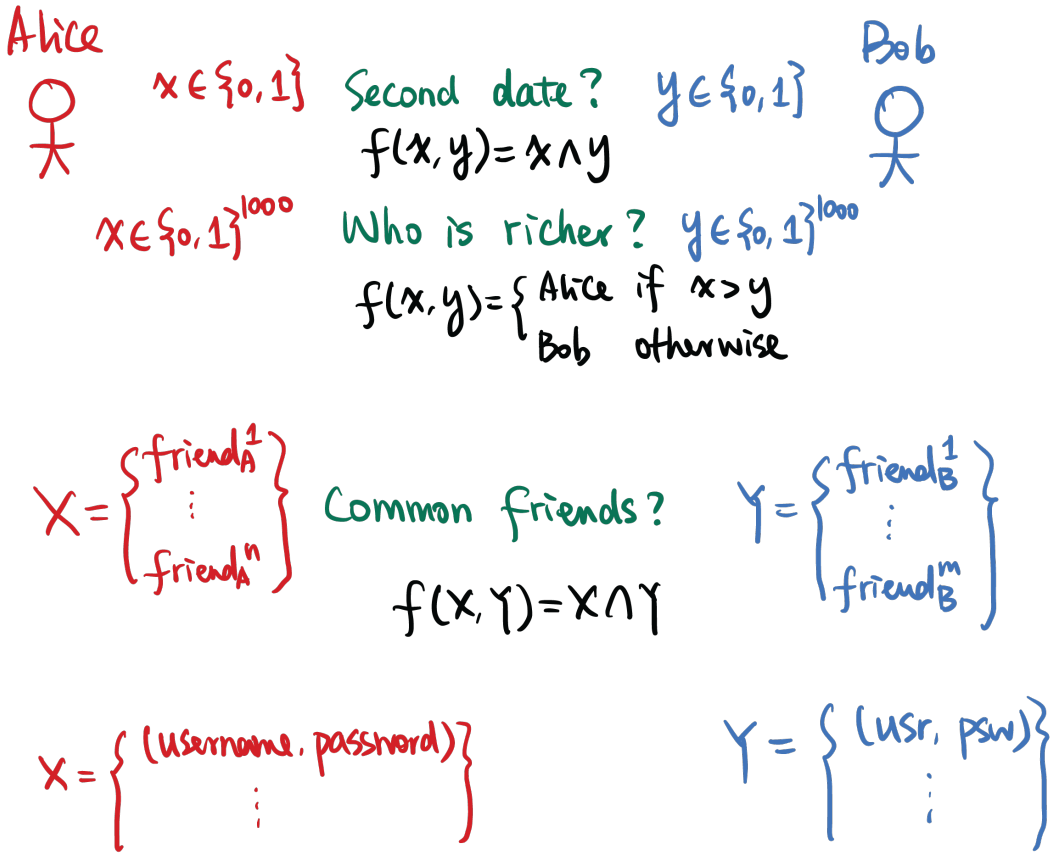
If the statement is false, $\Pr[b' = b] = \frac{1}{2}$ (Alice is guessing with 0.5 probability).

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

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

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

§1.5 Secure Multi-Party Computation



Example (Secure AND)

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

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

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

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

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

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

Example (Yao's Millionaires' Problem)

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

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

Example (Private Set Intersection)

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

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

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

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

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

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

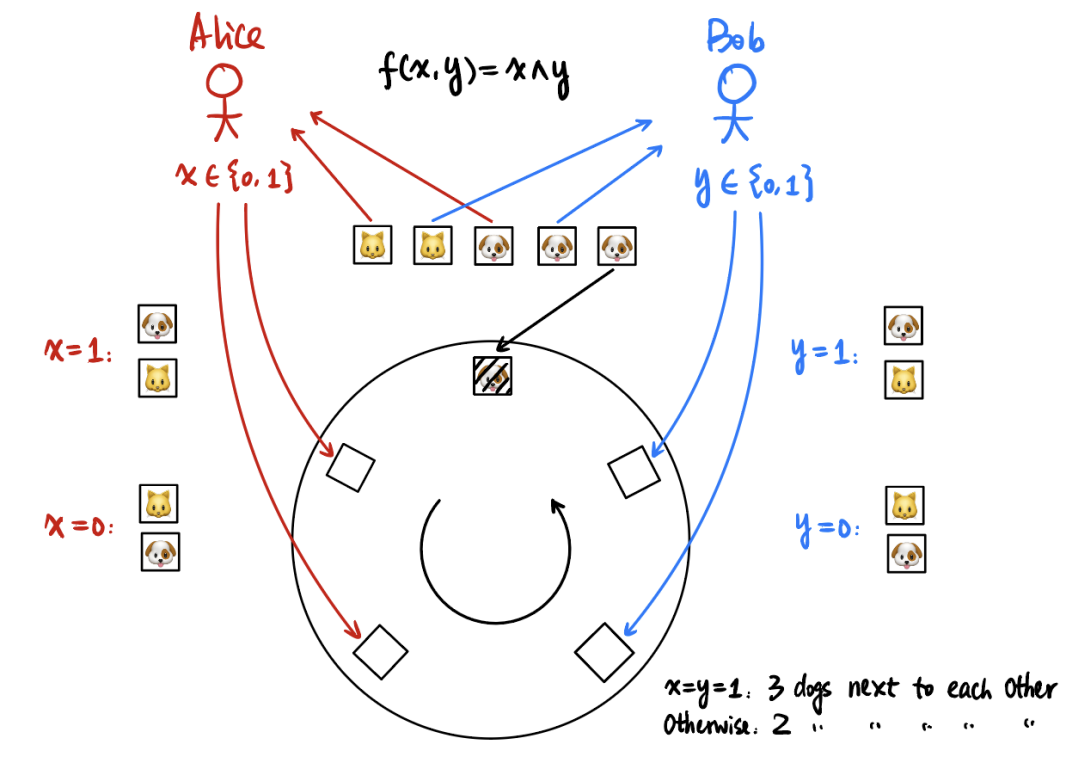
Yes, however in some cases (hospital health records), parties are legally obliged to keep data secure. We wish for security more than the secrecy of cardinality.

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

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

Example (Private Dating)

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



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

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

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

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

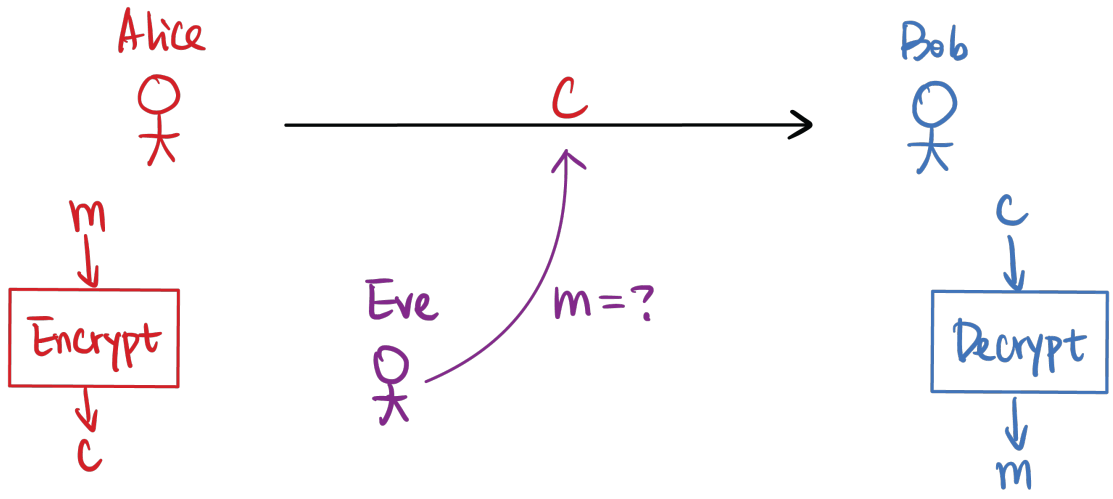
§1.6 Fully Homomorphic Encryption

We'll come back to the secure messaging example.

Alice wants to send Bob a message. She encrypts it somehow and sends a ciphertext $c_1 = \mathsf{Enc}(m_1)$. A nice feature for some encryption schemes is for Eve to do some computation homomorphically on the ciphertexts. Eve might possibly want to add ciphertexts (that leads to plaintext adding)

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

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

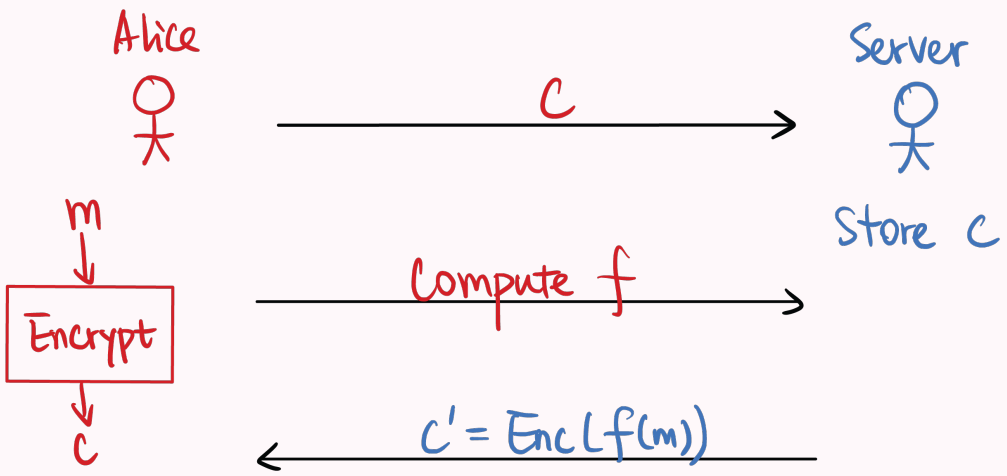


$$c_1 = \mathsf{Enc}(m_1)$$
$$c_2 = \mathsf{Enc}(m_2)$$
$$\Rightarrow \quad c' = \mathsf{Enc}(m_1 + m_2)$$
$$c'' = \mathsf{Enc}(m_1 \cdot m_2)$$

We want to hopefully compute any function in polynomial time!

Example (Outsourced Computation)

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



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

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

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

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

§1.7 Further Topics

We might cover some other topics:

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

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

- LLM Watermarking

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

Feel free to let us know on Ed!

§1.8 Q & A

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

  No!

- *Why C++*

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

- *Class Participation*

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

- *What is the difference between CSCI 1515 and CSCI 1510, MATH 1580, or CSCI 1040?*

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

  MATH 1580 considers crypto from the mathematical perspective. They try to understand some of the computational assumptions we assume from a mathematical standpoint. I.e. why is factoring hard to compute, and what is the best algorithm to compute it? In CS, we simply assume factoring is hard. MATH 1580 is more similar to number theory and group theory.

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

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

This course **will not** be offered in Spring 2027.

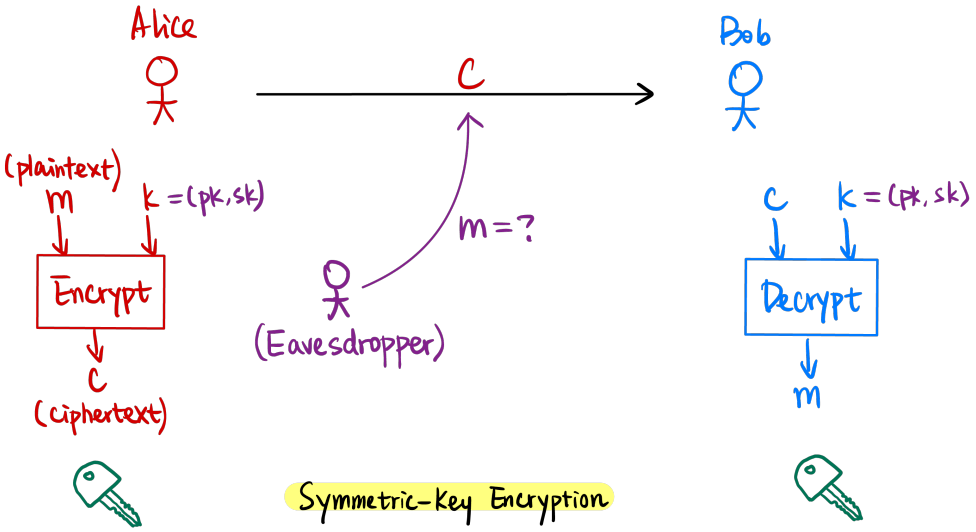
§2 January 28, 2026

Make-up lecture for our canceled Monday class will take place on Friday 3-4:20 pm in Friedman Hall 202 and on Zoom.

§2.1 Encryption Scheme Basics

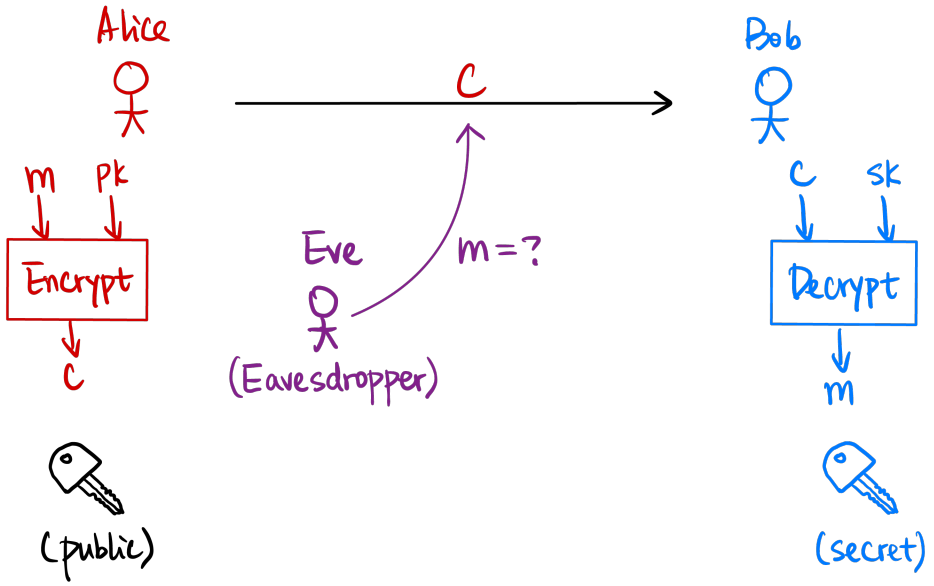
This lecture we'll cover encryption schemes. We briefly mentioned what encryption schemes were last class, we'll dive into the technical content: how we construct them, assumptions, RSA, ElGamal, etc.

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



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

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



§2.1.1 Syntax

Definition 2.1 (Symmetric-Key Encryption)

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

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

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

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

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

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

Definition 2.2 (Public-Key Encryption)

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

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

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

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

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

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

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

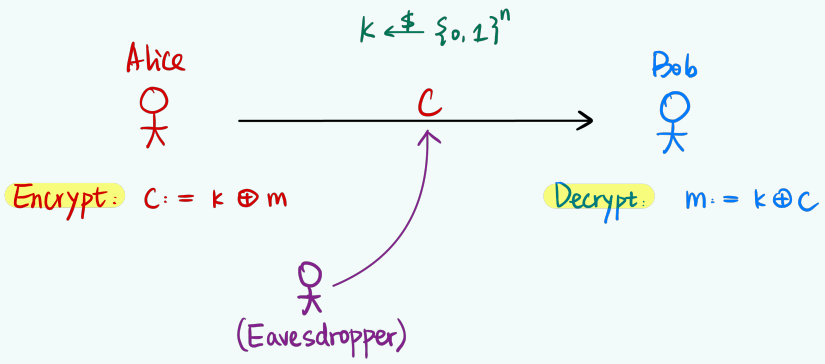
1. First of all, public-key encryption is almost always *more expensive*. Symmetric-key encryption schemes give us efficiency.

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

§2.1.2 Symmetric-Key Encryption Schemes

Definition 2.4 (One-Time Pad)

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



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

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

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

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

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

Question. Why is this correct?

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

Question. Why is this secure?

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

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

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

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

By XOR c and c', we get

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

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

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

Theorem 2.5 (Shannon's Theorem)

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

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

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

Question. How can we circumvent this issue?

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

We can think about computational security,

Definition 2.6 (Computational Security)

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

Definition 2.7 (Polynomial-Time Algorithm)

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

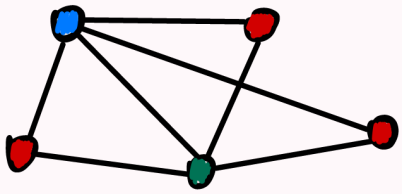
[3]That the distributions of ciphertexts are identical, that $\mathsf{Enc}_k(m_0) \equiv \mathsf{Enc}_k(m_1)$.

Definition 2.8 (NP Problem)

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

Example 2.9 (Graph 3-Coloring)

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



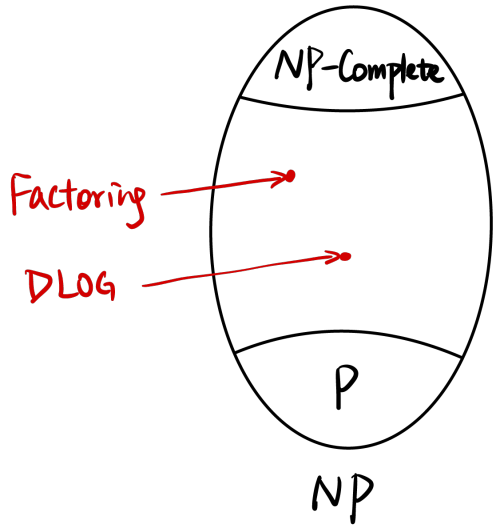
This can be *verified* in polynomial time (we can check if such a coloring is a valid 3-coloring), but it is computed in NP time.

Definition 2.10

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

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

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



Going back to our definition of computational security definition 2.6,

Definition (Computational Security)

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

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

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

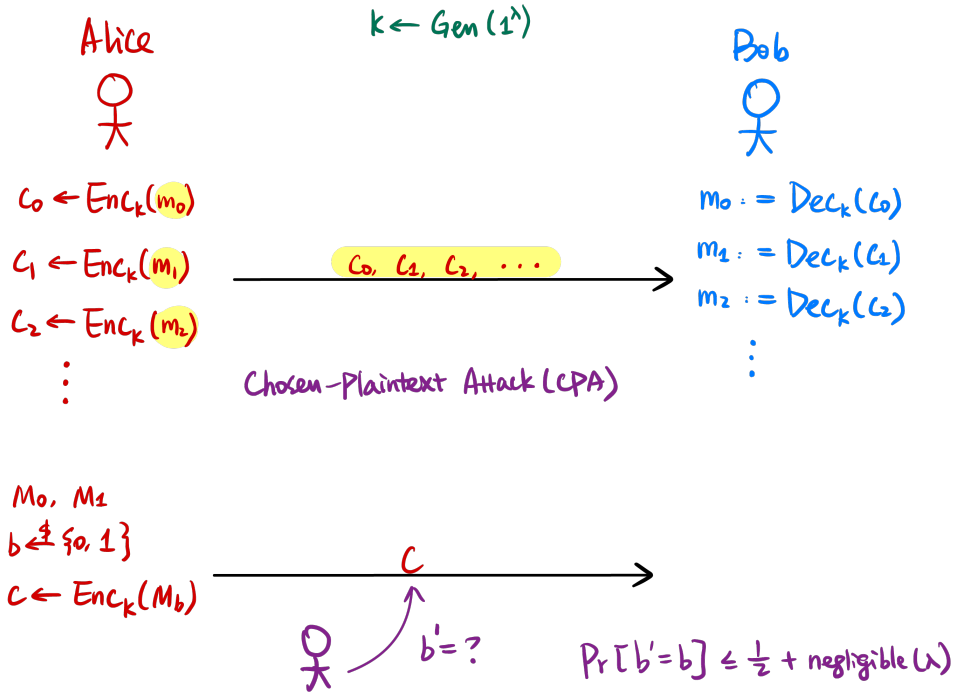
What does it mean for distributions to be "computationally indistinguishable"? Let's say Alice encrypts multiple messages $m_0, m_1, \ldots$ to Bob and produces $c_0, c_1, \ldots$. Even if Eve can see all plaintexts m_i in the open and ciphertexts c_i in the open, between known m_0, m_1 and randomly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

Definition 2.11 (Chosen-Plaintext Attack)

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

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

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



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

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

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

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

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

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

Then doing the math ...

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

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

Example 2.13

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

It can just be a λ bit string.

Example

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

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

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

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

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

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

§2.1.3 Public-Key Encryption Schemes

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

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

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

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

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

Theorem 2.14

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

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

Definition 2.15

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

Definition 2.16 (Primes)

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

Definition 2.17 (Mod)

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

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

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

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

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

Example

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

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

Definition 2.18

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

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

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

Example

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

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

or take $\gcd(12, 18)$

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

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

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

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

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

Question. How do we compute b?

We can use the Extended Euclidean Algorithm!

Example

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

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

We write this as

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

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

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

Definition 2.20 (Group of Units mod N)

We have set

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

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

Definition 2.21 (Euler's Phi Function)

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

Theorem 2.22 (Euler's Theorem)

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

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

With this, we can start talking about RSA.

§2.1.4 RSA

We first define the RSA assumption.

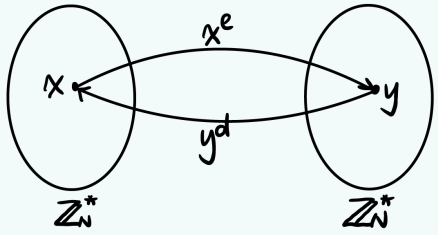
Definition 2.23 (Factoring Assumption)

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

Definition 2.24 (RSA Assumption)

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

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



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

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

Remaining questions:

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

We'll continue next class.

§3 January 30, 2026

§3.1 Basic Number Theory, *continued*

We recall a couple more definitions.

Definition 3.1

We first define the multiplicative group of integers modulo n as

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

Definition 3.2

We define the Euler's totient function as

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

Example 3.3

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

Theorem 3.4 (Euler's Theorem)

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

Corollary 3.5

If

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

, then

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

.

§3.2 RSA Encryption, *continued*

Recall: that the RSA encryption algorithm contains 3 components:

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

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

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

We have a few remaining questions:

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

2. How do we choose such an e?

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

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

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

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

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

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

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

Question. What happens if we can factor N?

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

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

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

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

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

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

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

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

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

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

§3.3 Intro to Group Theory

Definition 3.6 (Group)

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

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

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

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

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

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

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

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

Example 3.7

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

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

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

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

Definition 3.8 (Cyclic Group)

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

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

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

Example

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

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

Question. How do we find a generator?

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

§3.4 Computational Assumptions

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

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

Definition 3.9 (Discrete Lgoarithm (DLOG) Assumption)

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

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

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

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

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

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

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

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

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

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

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

§3.5 ElGamal Encryption

The ElGamal encryption scheme involves the following:

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

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

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

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

Notes about ElGamal:

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

§3.6 Secure Key Exchange

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

Definition 3.12 (Secure Key Exchange)

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

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

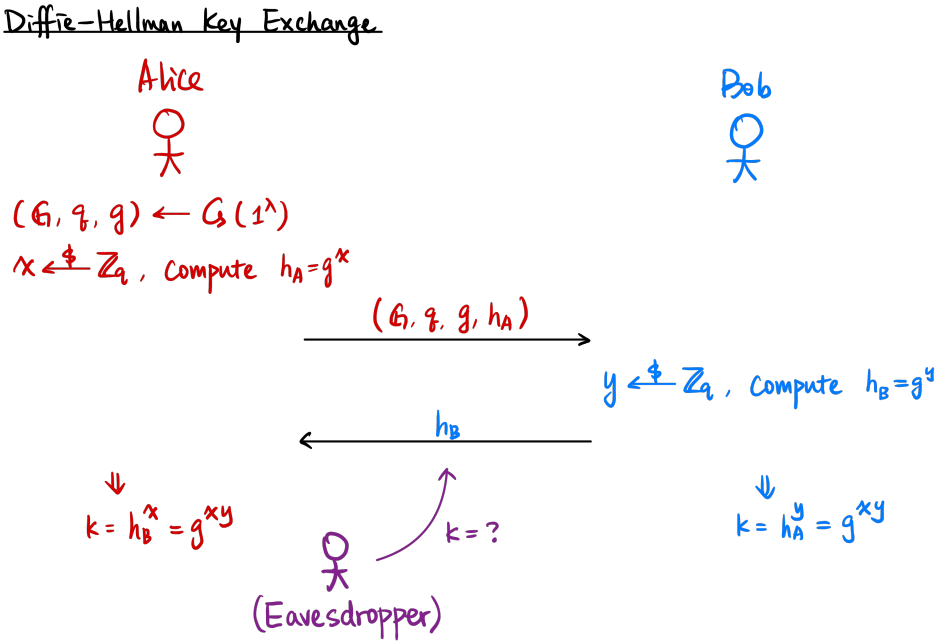
Theorem 3.13

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

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

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

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



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

§3.7 Prime Order Subgroups

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

Definition 3.14 (Subgroup)

A subgroup is some subset of a group that is also a group itself.

Definition 3.15 (Safe Prime)

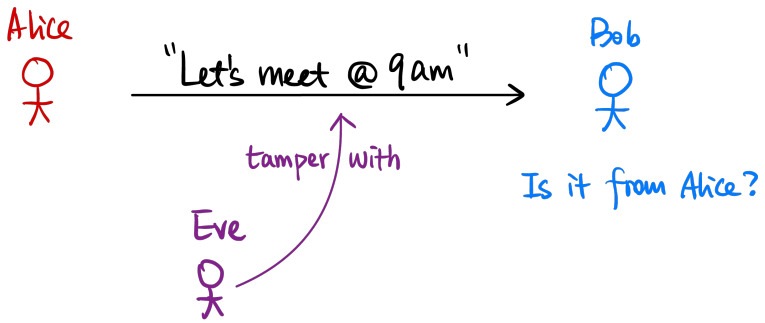
A prime p is a safe prime if $p = 2q + 1$, where q is prime. These are also known as Sophie Germain primes.

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

§4 February 2, 2025

§4.1 Message Integrity

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



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

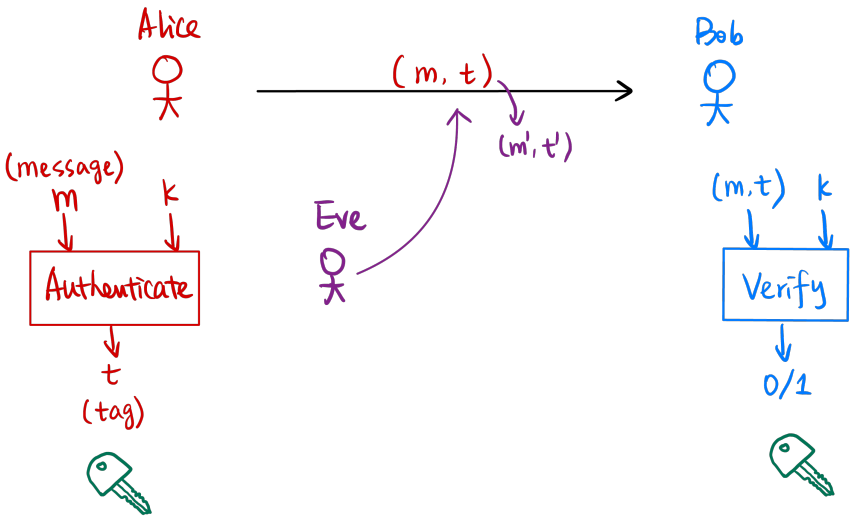
Using a message m, Alice can generate a *tag* or *signature*, and Bob can verify (m, t) is either valid or invalid.

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

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

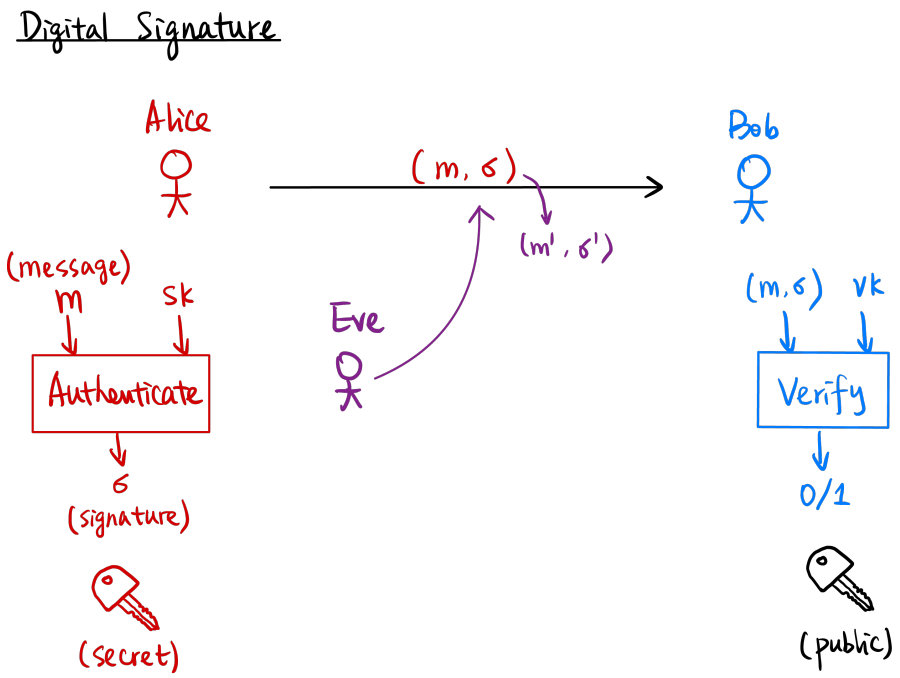
§4.1.1 Message Authentication Code

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.



§4.1.2 Digital Signature

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.



§4.1.3 Syntax

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

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

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

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

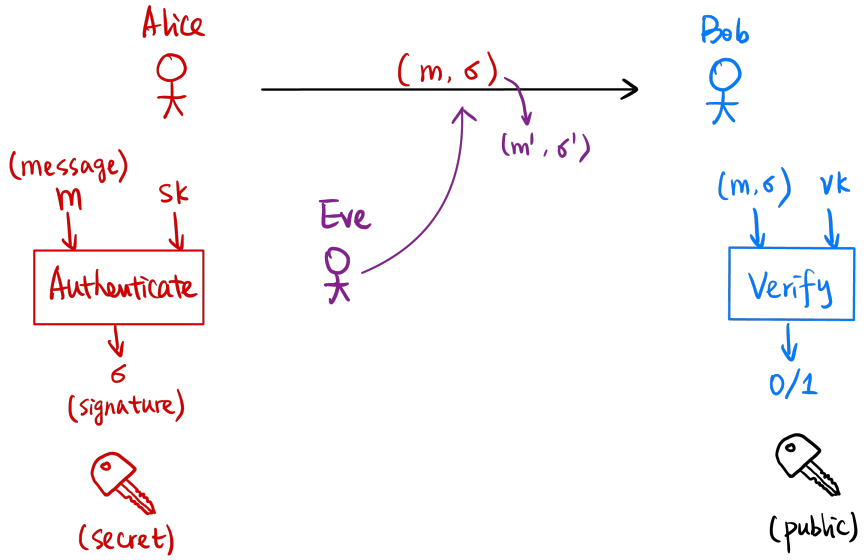
Verification $0/1 := \mathsf{Verify}_k(m, t)$.

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

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

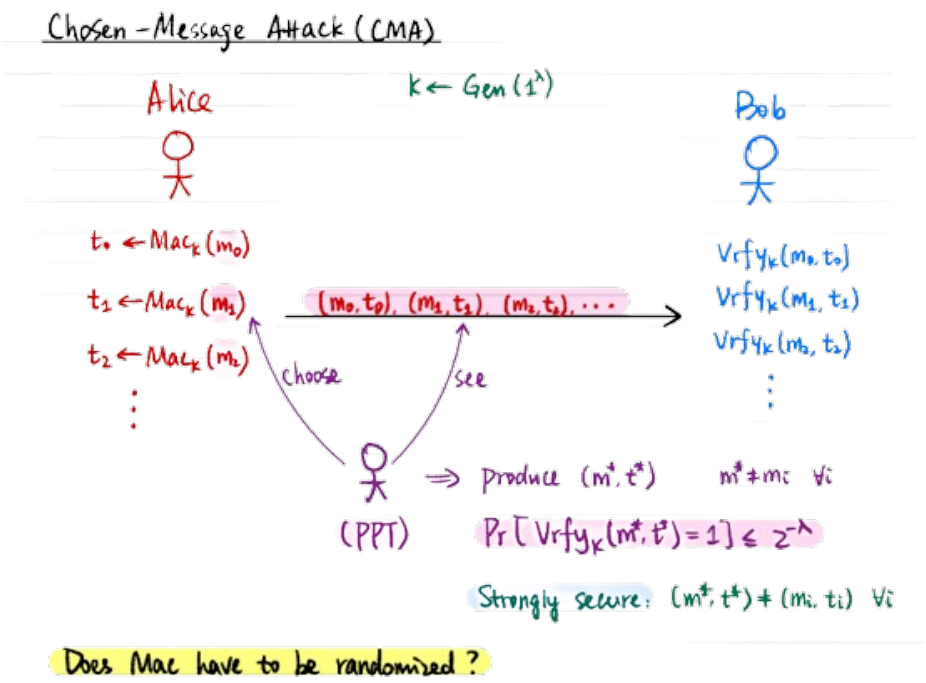
Authentication. $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.

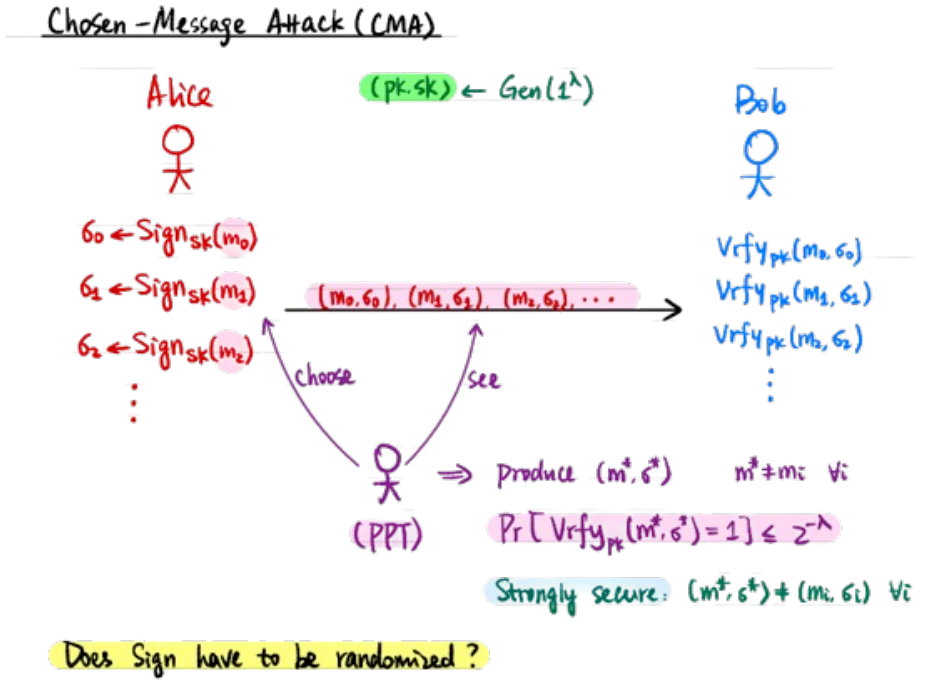
Verification $0/1 := \mathsf{Verify}_{vk}(m, \sigma)$.



§4.1.4 Chosen-Message Attack

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





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

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

§4.2 RSA Signatures

Our RSA signatures algorithm works very similarly to RSA encryption.

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

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

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

To verify, we compute

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

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

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

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

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

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

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

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

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

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

§4.2.1 Other Signature Schemes

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

- RSA signatures rely on the RSA assumption.

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

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

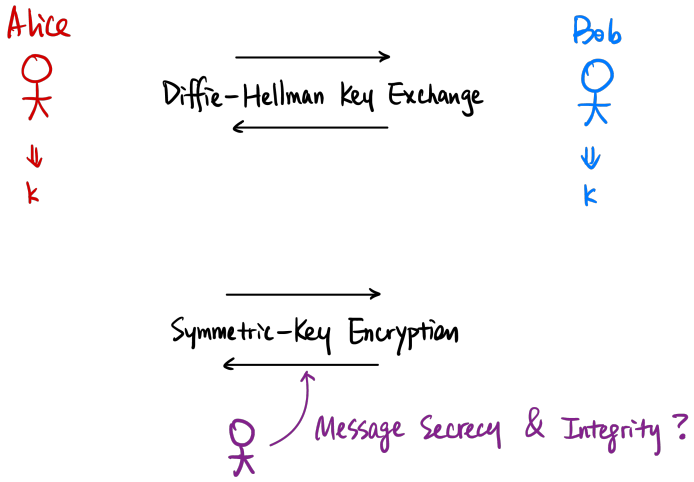
§4.3 A Summary So Far

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

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

§4.4 Authenticated Encryption

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



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

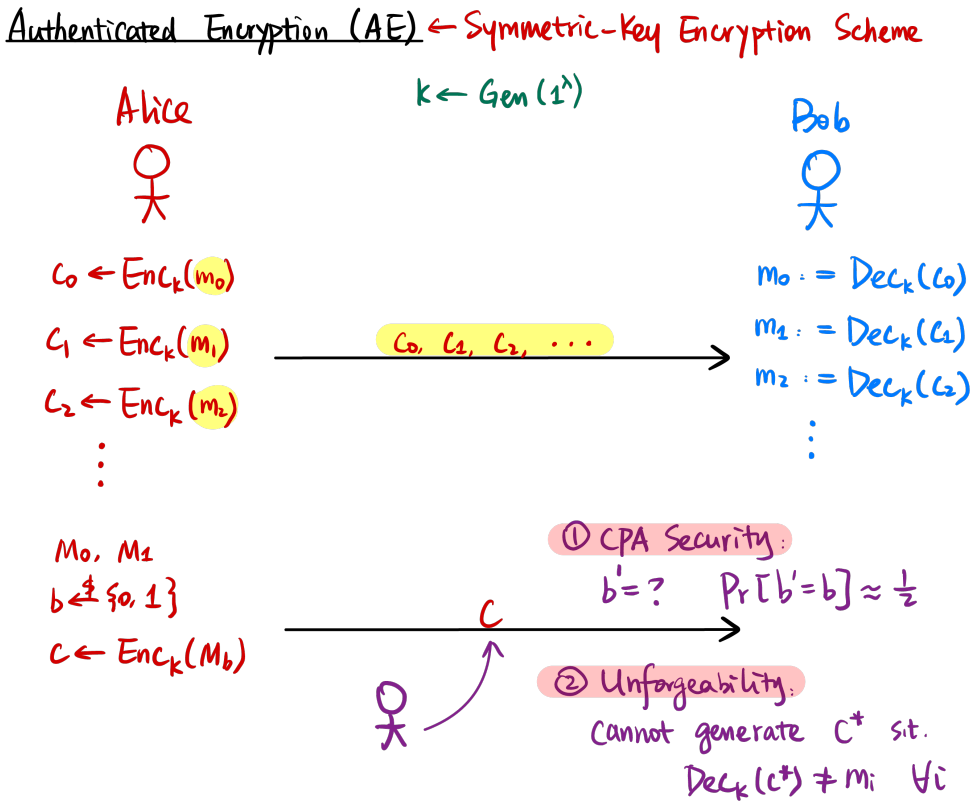
Our security definition is that our adversary can see the encryptions of many messages m_0, m_1, m_2. We want **CCA security**: that an adversary given previous ciphertexts and their decryptions cannot distinguish between the encryptions of a fresh pair of messages m_0 and m_1. *Additionally*, we want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

Definition 4.1 (Chosen Ciphertext Attack)

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

Definition 4.2 (Unforgeability)

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



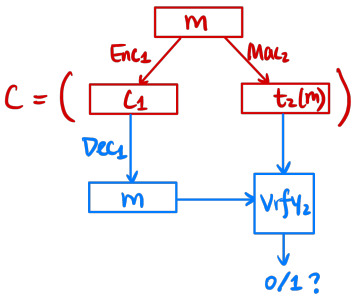
Now that we have two new primitives, we can construct Authenticated Encryption schemes.

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

§4.4.1 Encrypt-and-MAC?

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

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



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

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

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

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

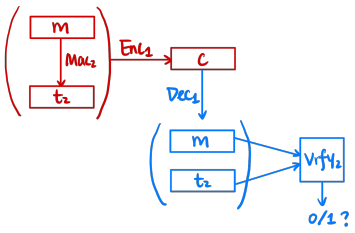
MAC gives you *unforgeability*—but it doesn't even try to hide the message at all. It's possible that the MAC scheme reveals the message in the clear. For example, we might have a MAC scheme that includes the message in the signature *in the clear* (which is still secure)!

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

We try something else...

§4.4.2 MAC-then-Encrypt?

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



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

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

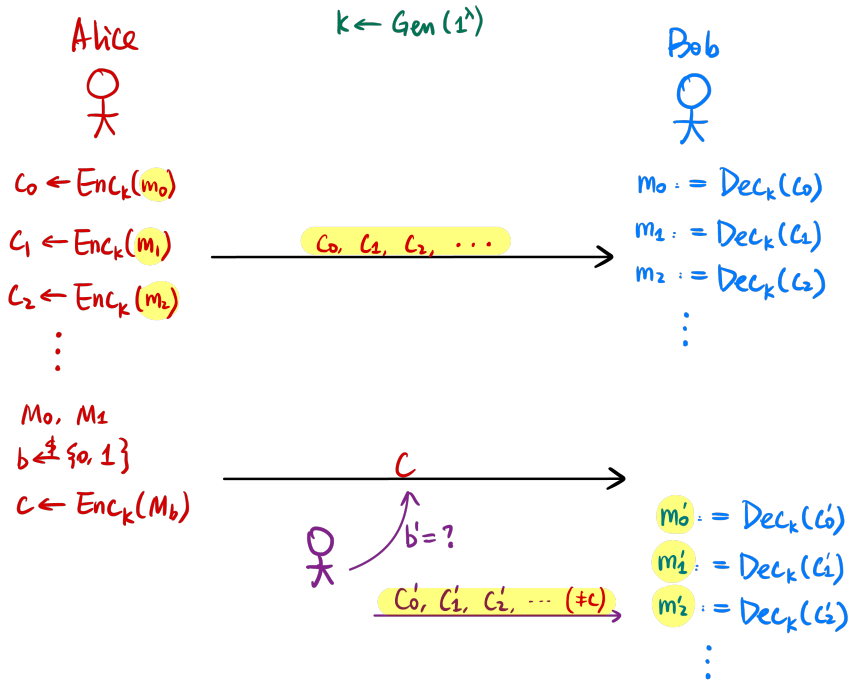
$\mathsf{Dec}(m)$: We have ciphertext c_1. Our message is $m||t_2 := \mathsf{Dec}_1(k_1, c_1)$, and our verification bit $b := \mathsf{Verify}_1(k_2, (m, t_2))$. If $b = 1$, output m, otherwise we output $\perp$.

Question. Is this secure?

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

§4.4.3 Chosen Ciphertext Attack Security

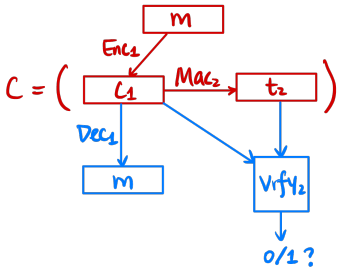
On top of CMA security, the adversary can now request Alice to decrypt ciphertexts $c_0, c_1, \ldots$.



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

§4.4.4 Encrypt-then-MAC

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



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

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

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

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

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

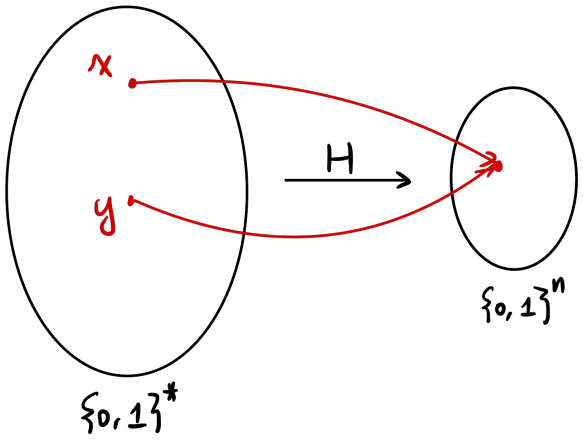
§5 February 04, 2025

§5.1 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?

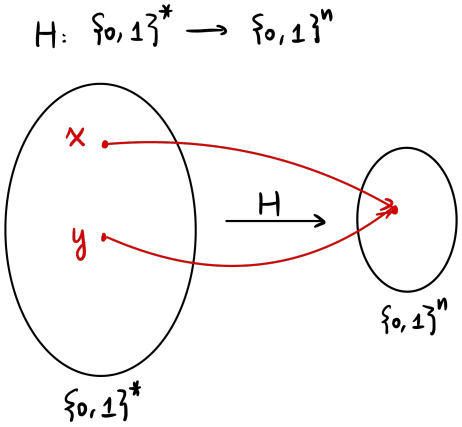
> **Remark.** This is related to the birthday problem. If there are q students in a class, assume each student's birthday is a random day $y_i \xleftarrow{\$} [365]$. What is the probability of a collision? $q = 366$ gives 1, $q = 23$ gives around 50%, and $q = 70$ gives roughly 99.9%.
>
> We can apply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

§5.2 Collision-Resistant Hash Function (CRHF)

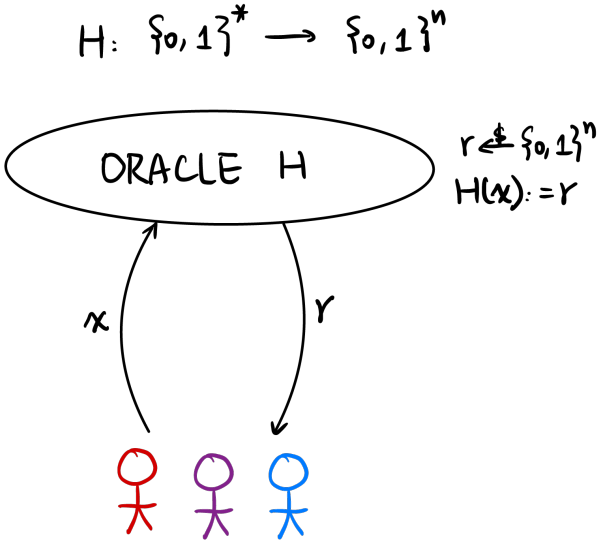
Recall that we defined a hash function to be a public and deterministic function for which it is computationally hard to find a collision. That is, finding two distinct strings x, y such that $H(x) = H(y)$ is computationally difficult.

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



§5.2.1 Random Oracle Model

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

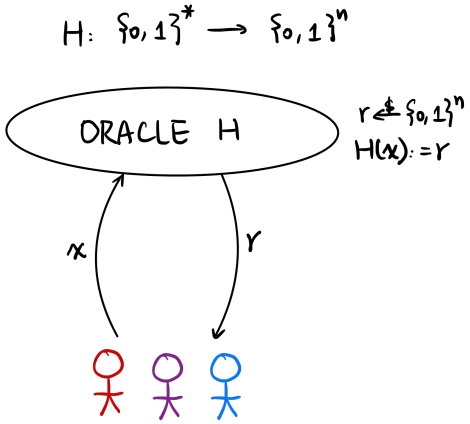


There are proofs that state that no hash functions can be a random oracle. There are schemes that can be secure in the random oracle model, but are not using hash functions[12].

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

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

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



§5.2.2 Constructions for Hash Function

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

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

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

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

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

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

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

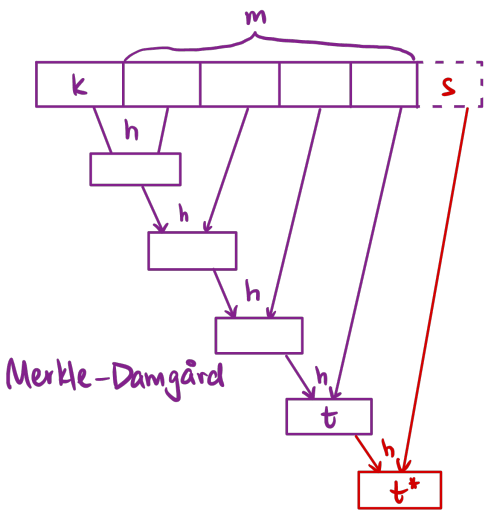
> **Remark.** The folklore is that during a session at a cryptography conference, a mathematician, Xiaoyun Wang, presented slide-after-slide of attacks on MD5 and SHA-0, astounding the audience.

§5.2.3 Applications

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

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

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



Therefore, in practice, we use a nested MAC like

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

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

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

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

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

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

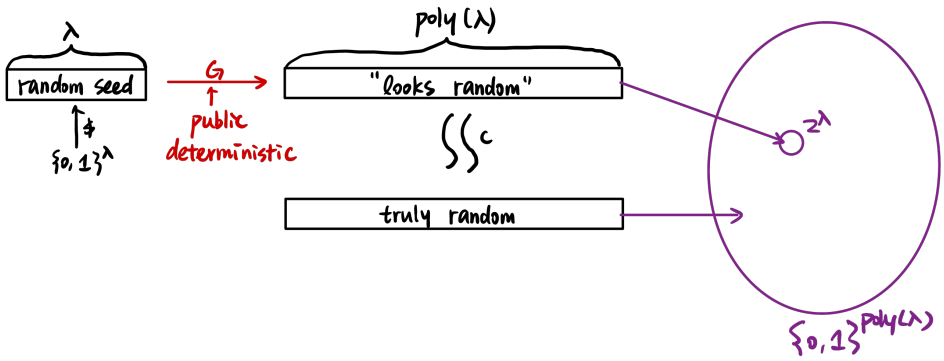
Deduplicate Files. We can also use hash functions to deduplicate files. We can hash two files to produce identifiers h_1 and h_2. If $h_1 \neq h_2$, this implies $D_1 \neq D_2$. If $h_1 = h_2$, it almost always[14] implies that $D_1 = D_2$.

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

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

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

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



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

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

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

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

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

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

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

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

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

§5.3 Putting it Together: Secure Communication

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

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

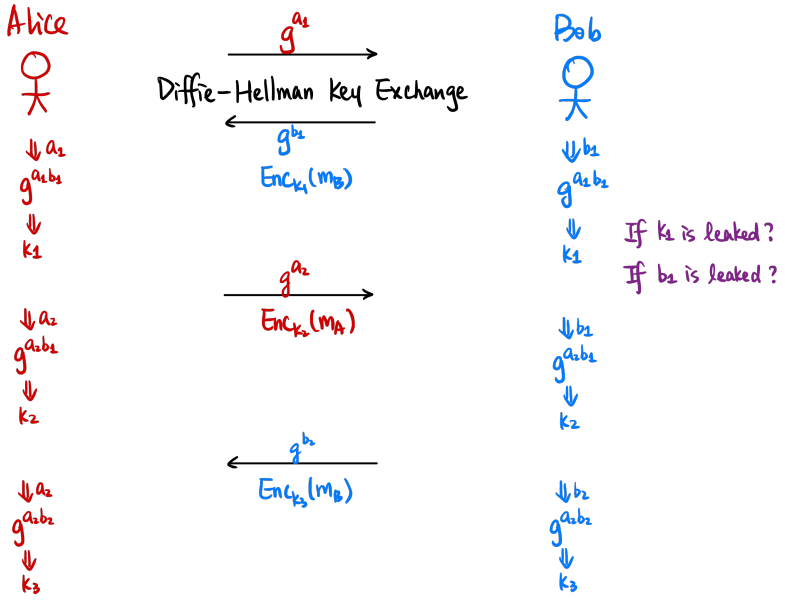
Question. Are there any issues with this scheme?

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

§5.3.1 Diffie-Hellman Ratchet

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

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



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

Question. What if k_1 is leaked?

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

Question. What if b_1 is leaked?

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

§5.4 Block Cipher

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

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

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

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

§5.4.1 Pseudorandom Function (PRF), *continued*

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

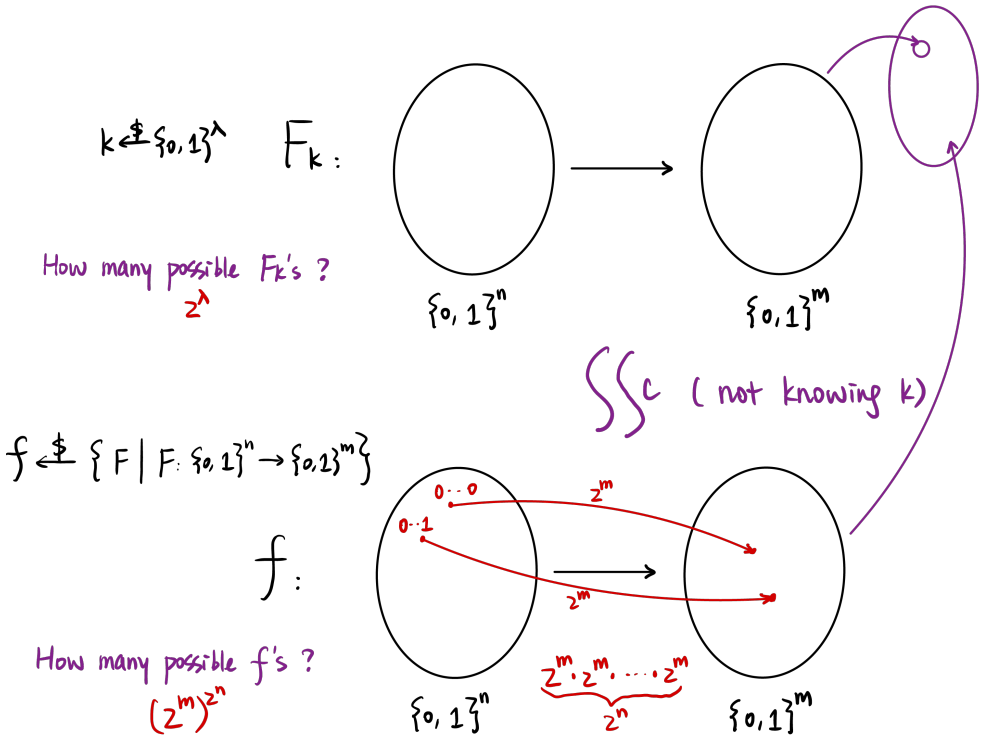
Recall last time that we talked about pseudorandom *generators* (PRG), which takes a seed and expands it into a long string of pseudorandom bits. This "random-looking" **string** is computationally indistinguishable from a truly random string.

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

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

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

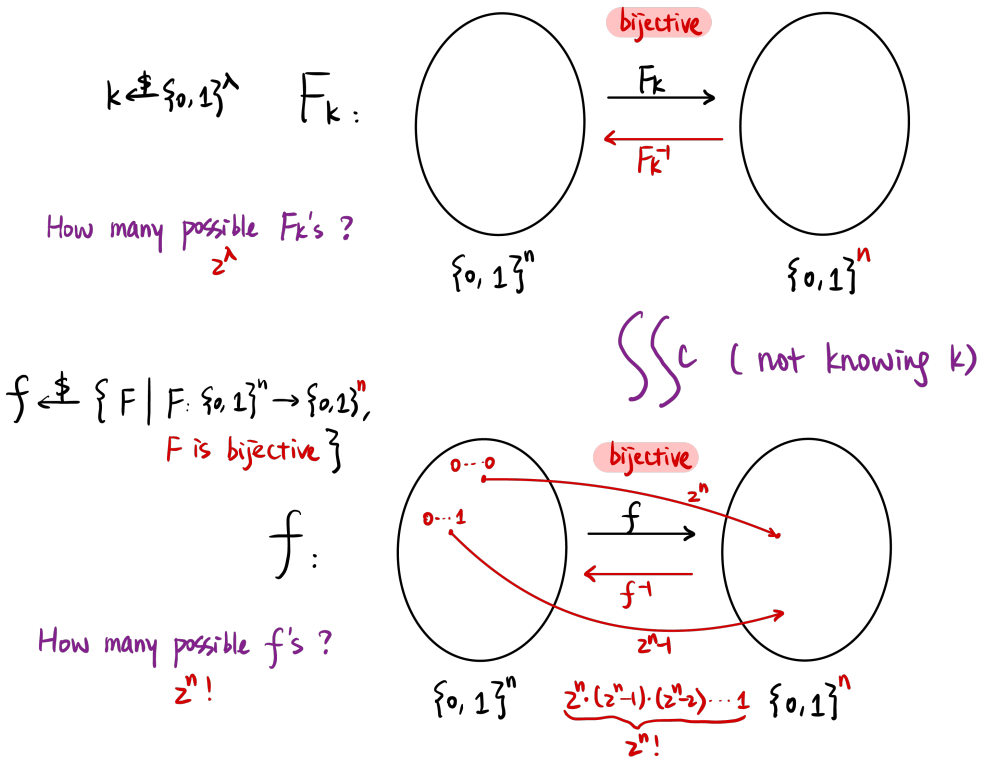
[16] In deterministic polynomial-time.



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

§5.4.2 Pseudorandom Permutation (PRP)

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



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

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

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

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

§6 February 9, 2025

§6.1 Recap

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

§6.2 Practical Constructions of Block Ciphers

Looking back on section 4.3, the last outstanding primitive was 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.

§6.2.1 History of AES and DES

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

We will see how to construct DES, which has similar ideas in constructing AES. To do so, we need to talk about Substitution-Permutation Networks (SPN) and Feistel Networks.

§6.2.2 Substitution-Permutation Network (SPN)

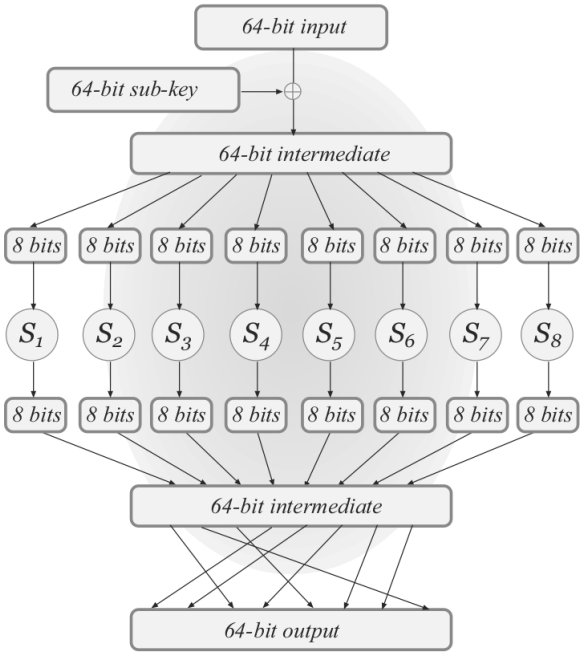
We want to incorporate a design principle known as the "Avalanche Effect", where a small change in the input should have an effect on every part of the output. In particular, even if one-bit is changed in the input, every bit in the output should be affected so that it looks completely different.

SPN proceeds as follows.

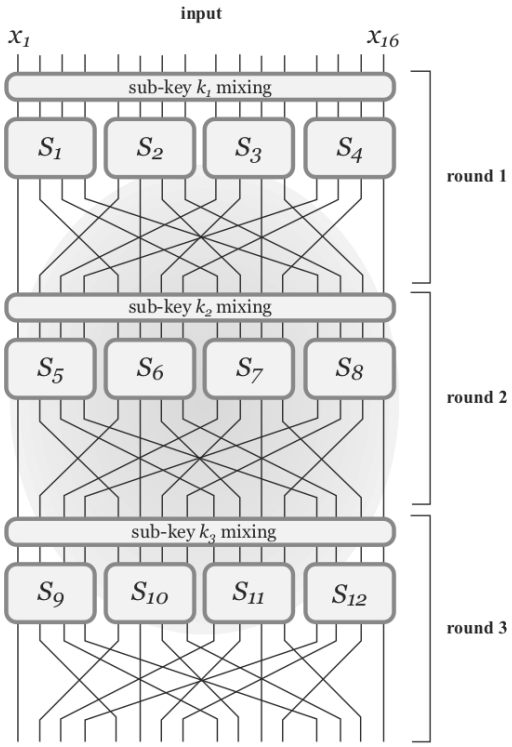
Step 1. **Key mixing.** Take input x and XOR it with sub-key k, i.e. $x := x \oplus k$. The result is 64-bit.

Step 2. **Confusion Step.** Split x into 8 parts of 8 bits each. On each part, apply an **S-box**, which is a public permutation of 8 bits. Furthermore, 1-bit change in the input gives a 2-bit change in the output.

Step 3. **Diffusion Step.** Take the output from each S-box and concatenate them together into a 64-bit result. Then use a public mixing permutation to shuffle the bits.



We can repeat this procedure for multiple rounds. For example, below is a 3-round SPN.



Different sub-keys are used in each round. These sub-keys are derived from a *master key* using a *key schedule*, for example, by taking different subsets from the master key.

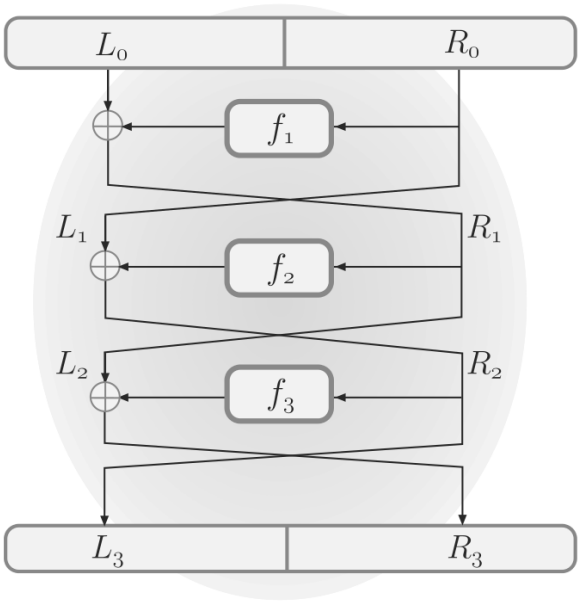
Given the master key, we can compute $F_k^{-1}(y)$. First, use the key schedule to find each sub-key, which we can use to XOR to get the inverse. Then, since each mixing permutation is public, we can compute its inverse for each round. Additionally, each S-box is public, so we can find its inverse too. Thus we have all the information we need to compute the inverse.

§6.2.3 Attacks on Reduced-Round SPN

1-round SPN without final key mixing. The adversary can begin with the output y, then invert it starting from the end to the beginning. Since the permutations are public, the adversary can compute $x \oplus k$, and since they know the input x, they can figure out k. Thus we have a complete break. This shows that we need a final key mixing step.

1-round SPN with final key mixing. Assume the key and input is 16-bit. The adversary can try to attack by enumerate the possible values of k_2, then using the SPN derive k_1 for each k_2. This takes $O(2^{16})$ time and gives 2^{16} possible values for the master key.

§6.2.4 Feistel Network



Let $x = L_0||R_0$ be the input split into two halves, left and right. The R_0 is fed into a *round function* f_1 and is XOR-ed with L_0 to get R_1. Then L_1 is defined to be R_0. This is repeated for several rounds (3 rounds in the figure above). Let $y = L_n||R_n$ be the output after n rounds.

To compute the inverse $F_k^{-1}(y)$, start from the output e.g. (L_3, R_3). We know $R_2 = L_3$, so we can compute $f_3(R_2)$. This satisfies $f_3(R_2) \oplus L_2 = R_3$, so we can find L_2. Thus we have found (L_2, R_2), and we can continue until we get x.

There are attacks on reduced-round Feistel Networks that we will not cover in lecture, but you can think about it on your own.

§6.2.5 Data Encryption Standard (DES)

Recall that the block length is $n = 64$ and master key length is $\lambda = 56$ for DES. Use the Feistel network on our 64-bit input x so that L_0, R_0 each get 32 bits. For the round functions, we use something called a *DES mangler function*, which is essentially a SPN.

However, unlike an SPN, the S-boxes are not permutation, but rather they reduce the size from 6-bit to 4-bit. Additionally, the follow the following properties

1. Maps $\{0,1\}^6 \to \{0,1\}^4$.

2. "4-to-1": Exactly 4 inputs map to the same output.

3. 1-bit change of input gives at least 2-bit change in the output.

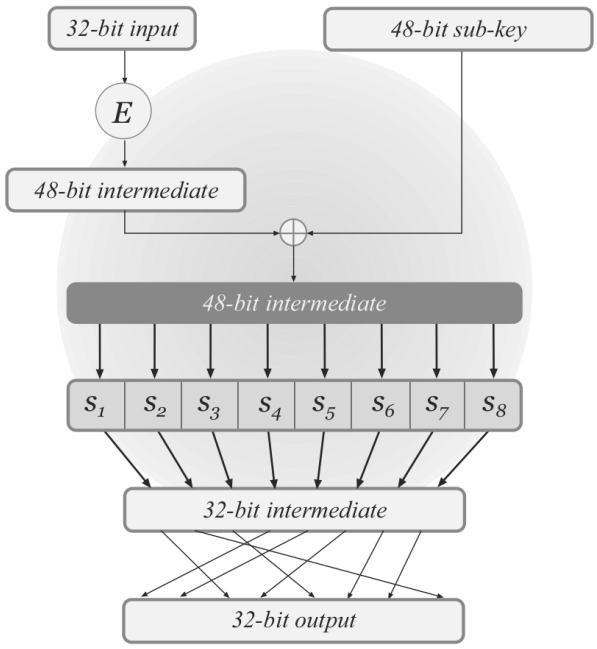


FIGURE 7.6: The DES mangler function.

E is an expansion function. Given a 32-bit string AB where A and B are 16-bit, the output is a 42-bit string ABA.

For the key schedule, we have a master key with length 56 but need a 48-bit sub-key. To do so, split the master key into two halves of 28-bits each, then take a random subset of 24-bits from each half. Then concatenate these two subsets to form a 48-bit sub-key.

> **Remark 6.1.** DES does seem very complicated. This is intentional to do so so that attacks are hard.
>
> Multiple rounds of DES does not necessarily improve the security guarantees. Once DES is applied multiple times, we lose our security guarantee, and we need to do additional cryptanalysis, so it is unclear if it is secure. This is the reason NIST developed a new standard known as AES.

§6.3 Practical Constructions of Hash Functions

Given a block cipher F, there is a way to construct a hash function.

First, let's construct a fixed-length hash function. We can split the input into two halves, k and x, and compute $F_k(x)$ as the output. This construction is known as the Davies-Mayer construction.

Theorem 6.2

If F is modeled as an "ideal cipher," then the Davies-Mayer construction is collision resistant.

Notice that this construction creates a compression function which takes an input of 2λ bits, and produces an output of λ bits. But hash functions can take an arbitrary length string!

To achieve this, we can use the Merkle-Damgård construction, which iteratively applies the compression function to blocks of the input message. At the end of the input message, we attach one more block which contains the length of the message in order to ensure collision-resistance.

Theorem 6.3

If the compression function is a collision-resistant hash function for fixed-length inputs, then Merkle-Damgård construction is a collision-resistant hash function for arbitrary-length inputs.

§6.4 Block Cipher Modes of Operation

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

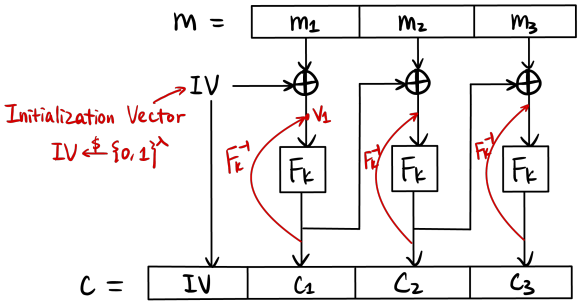
In summary: not CPA secure.

§7 February 11, 2026

§7.1 More Block Cipher Modes of Operation

Last time, we introduced the Electronic Code Book (ECB) mode, and demonstrated that it is not CPA secure.

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



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

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

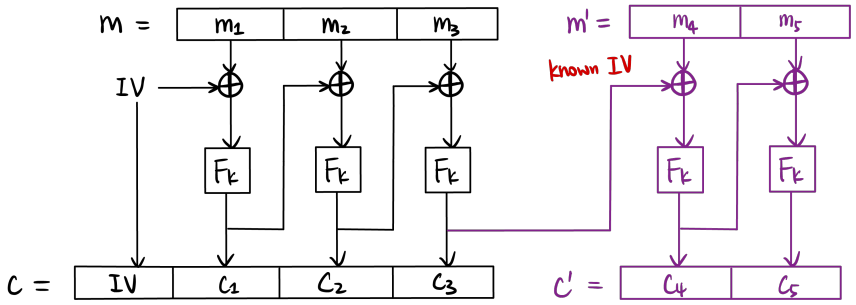
Unfortunately not. Say m_1 is XORed onto $0\cdots01$, an adversary under CPA can choose plaintext that is m_1 with its last bit flipped, such that v_1 is manipulated and the block cipher is again deterministic.

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

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

In summary: CPA secure, but non-parallel.

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



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

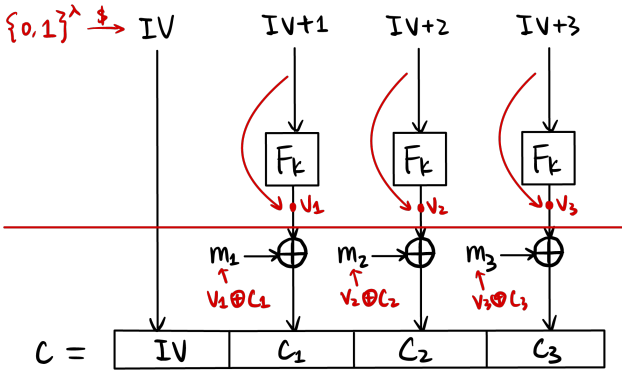
This makes chained-CBC *very subtly* different than CBC. If we squint our eyes enough, it just looks like sending a single message using CBC mode. The key difference is that between rounds of communication m and m', an adversary could influence m' given the knowledge of the previous round.

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

In summary: not CPA secure.

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

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



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

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

What about a "stateful CTR mode" which just increments IV every successive time? Instead of sending a new IV for the next message, we'll just increment the IV from before. Similar to Chained-CBC mode, the adversary will know the IV that is going into the next message. However, this doesn't *really* help the adversary. They've never seen those encrypted IV values before, and hence cannot modify the message given this information.

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

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

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

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

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

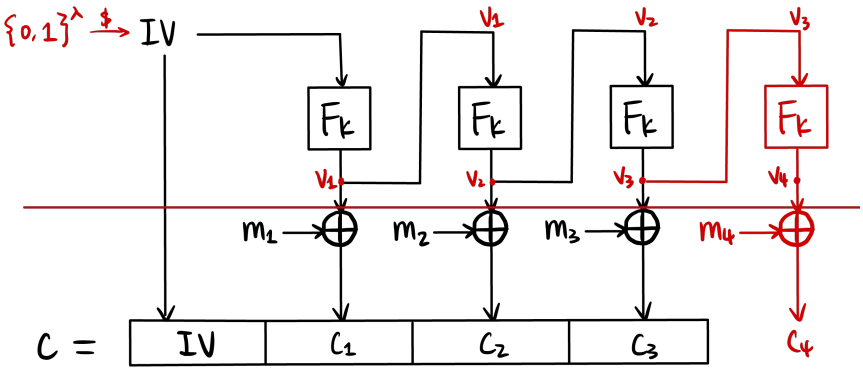
In fact, we can get rid of IV entirely and start at 0,

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

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

In summary: CPA secure, parallel.

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



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

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

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

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

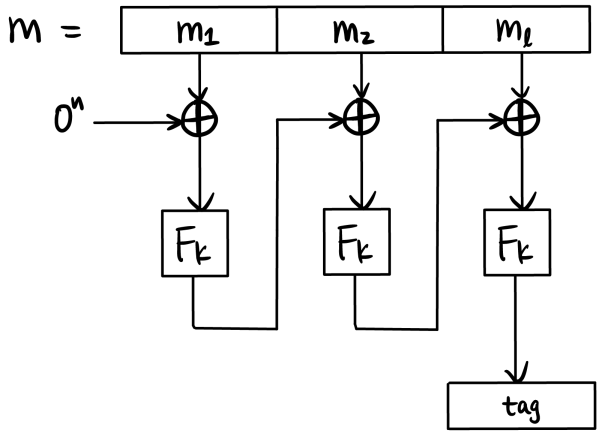
We've seen how things can go wrong catastrophically[17]. This is more true for counter mode than CBC mode. If our IV is reused in counter mode, our entire one-time pad has been exposed previously[18]. However, if our IV is reused in CBC mode, the worst that could happen is something akin to ECB mode,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

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

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

§7.1.1 CBC-MAC

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



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

Is this CMA secure?

- Fixed-length messages of length $l \cdot n$? Yes, since we can only query for fixed-length messages, this gives us no additional information.
- Arbitrary-length messages? This is where problems arise—the adversary could first query for a message of 1 block, then 2 blocks, then 3 blocks, etc. By combining this information, they could produce new valid signatures.

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

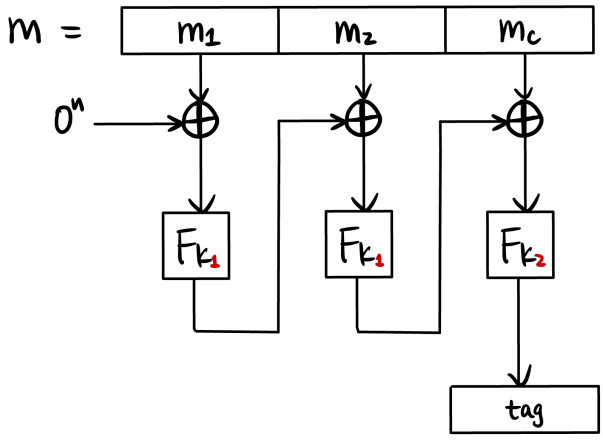
> **Remark.** Our constructions of authenticated encryption calls for an encryption scheme and MAC scheme. It's crucial that the two schemes have *different keys*. Using the same key k for both encryption and MAC can cuase issues (information from one could reveal something about the other).

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

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

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

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



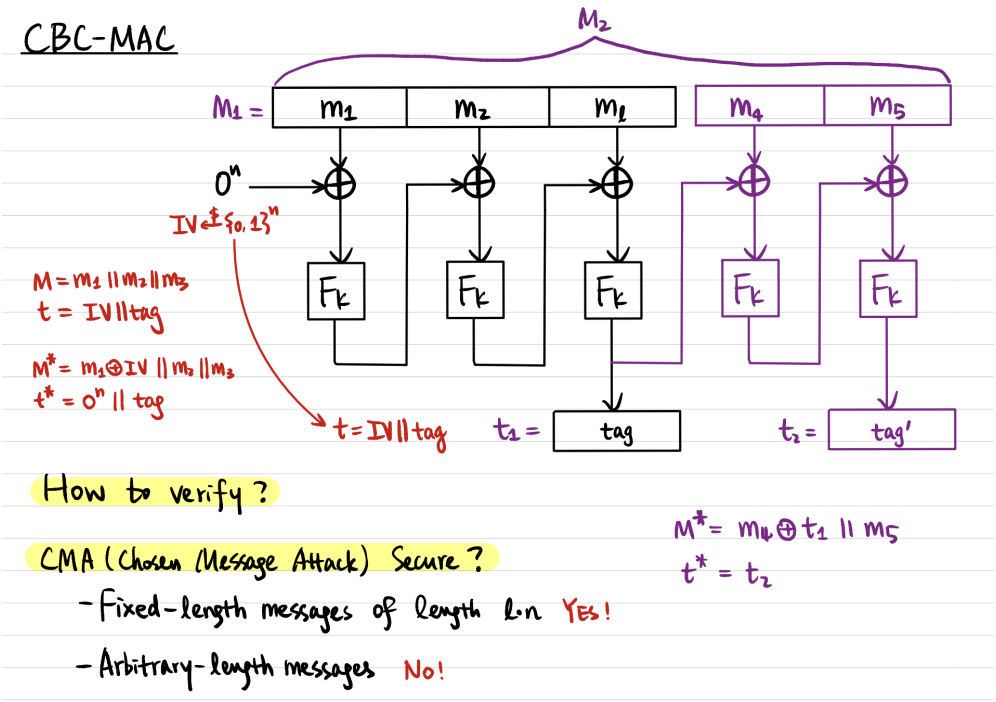
We could also attach length of messages to the first block, or other techniques.

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

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

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

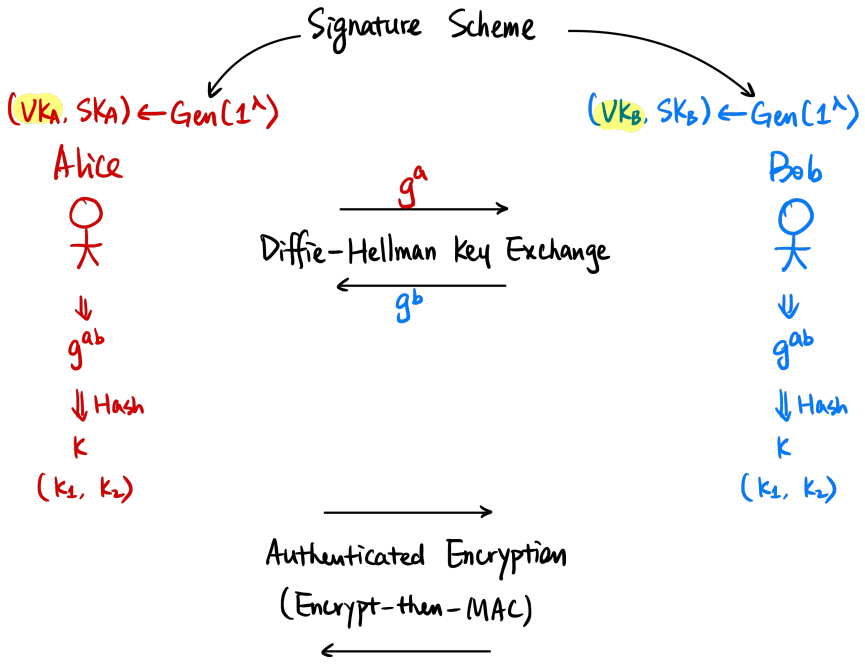
The adversary can generate a new valid tag for $m^* := m_1 \oplus \text{IV}||m_2||m_3$ and $t^* := (0^n, \text{tag})$.



§7.2 Putting it Together

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

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

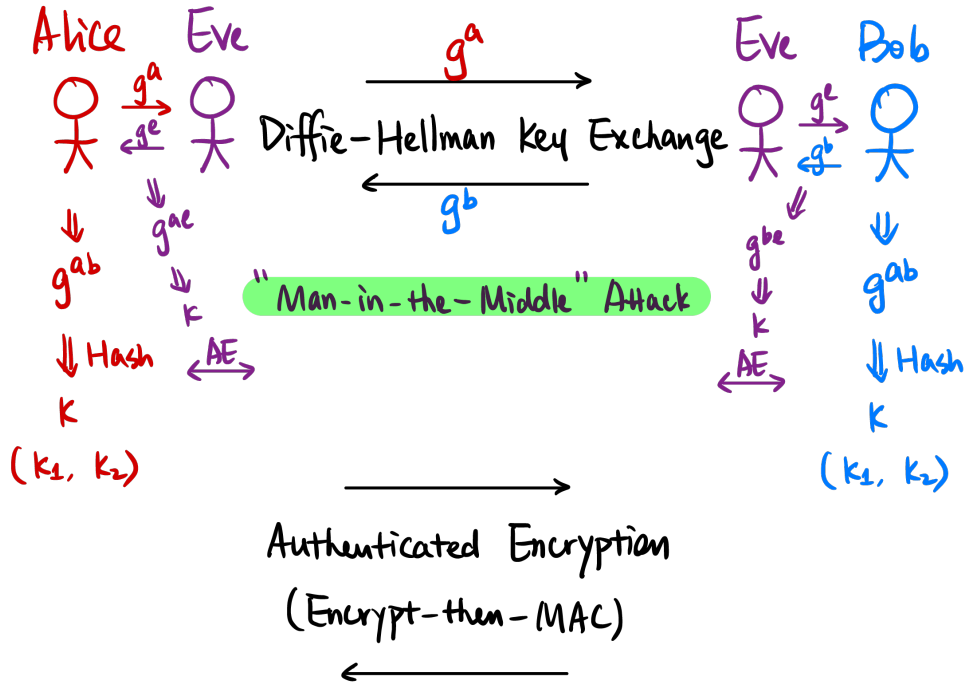


However, this still does not mitigate against a man-in-the-middle attack. Thus, before exchanging keys, Alice and Bob should publish verification keys (to a digital signature scheme, see section 4.1.2). Using this digital signature, Alice and Bob will each sign their Diffie-Hellman public values g^a,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

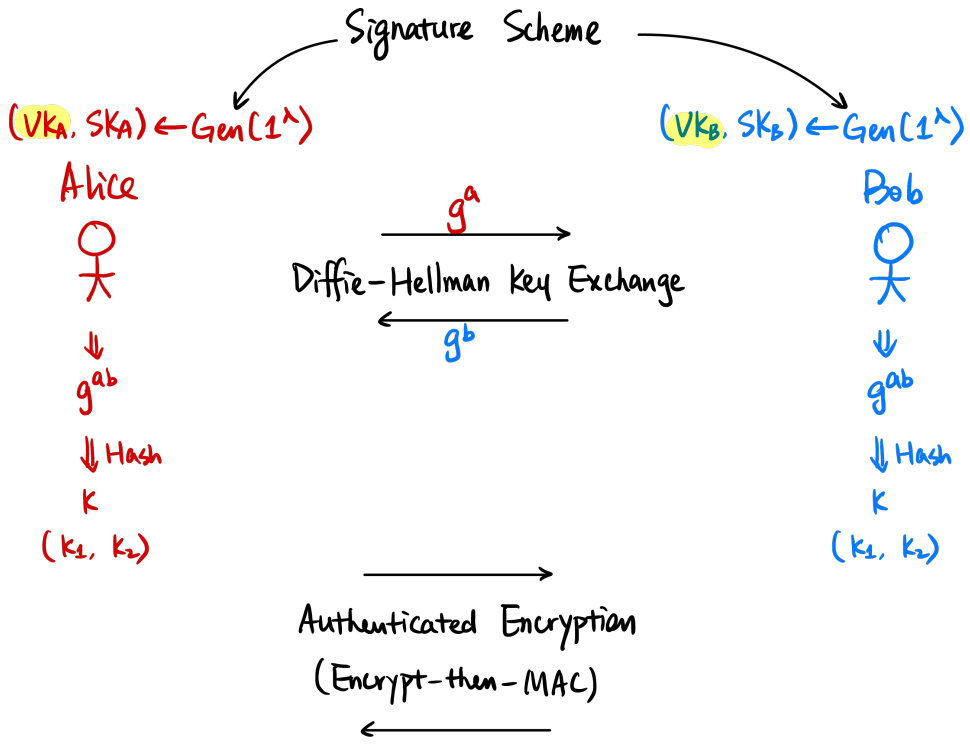
We will now go over the topics needed for the next project, Auth. Namely,

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

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



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



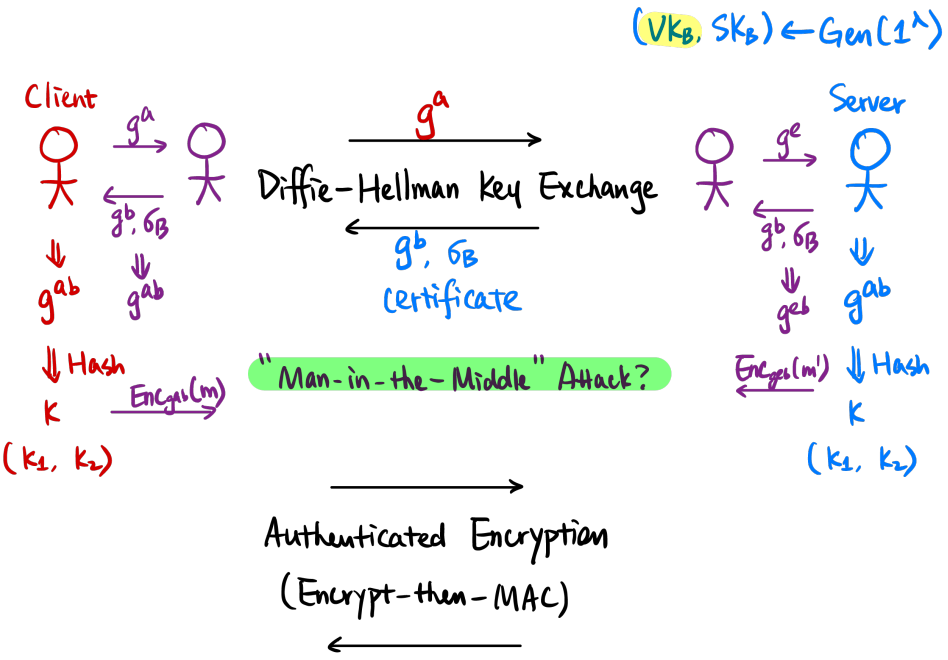
Is this now secure against an adversary in the middle? Yes, because the public values are guaranteed (via our digital signature scheme) by Alice and Bob's signing key. The man in the middle does not have access to the signing key, and cannot sign a phony public value.

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

§8 February 18, 2026

§8.1 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.



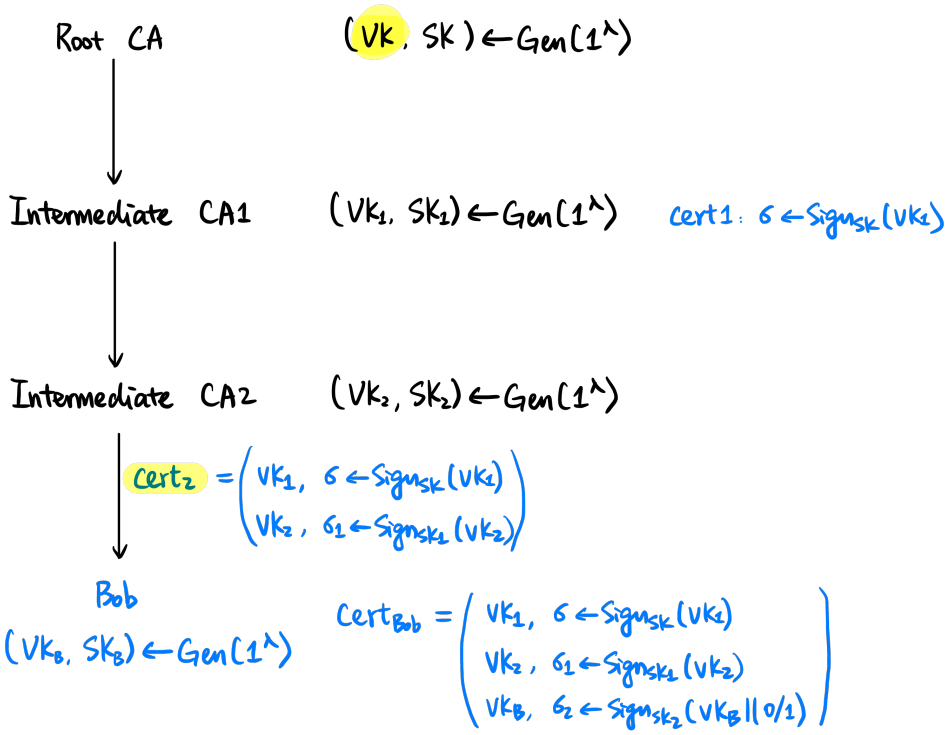
Whacould an adversary potentially do? The adversary could not pretend to be the server since they have no access to the server's signing key. The adversary *can* pretend to be the client and talk to the server. The adversary could forward all messages sent to the server, and can also communicate g^b,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

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

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

§8.1.1 Certificate Chain

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



A Root CA[20] 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

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

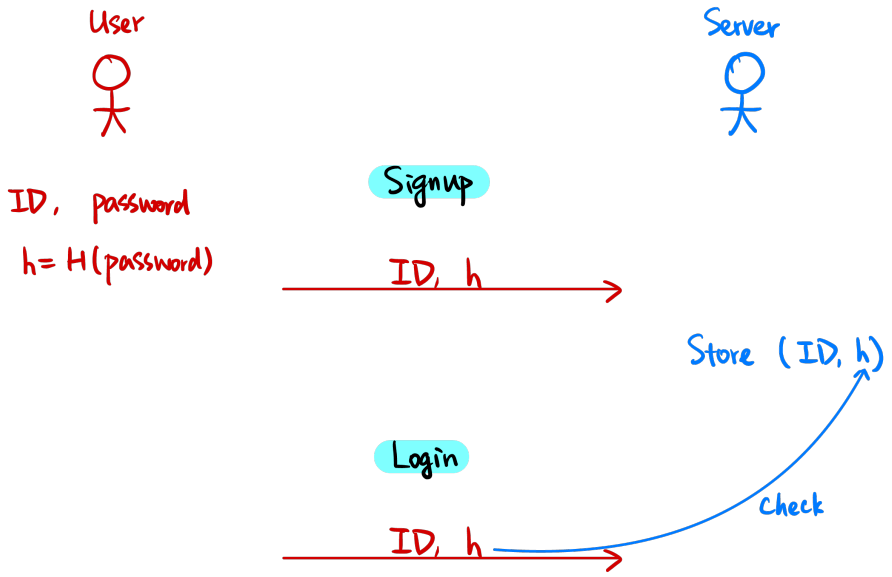
[20] We mentioned earlier that CAs are built into devices. For example, here is a list of all root certificates that are built-in for Apple devices. This can go wrong too! CAs have been misused which causes implications on the security of the internet.

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.

§8.2 Password-Based Authentication

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

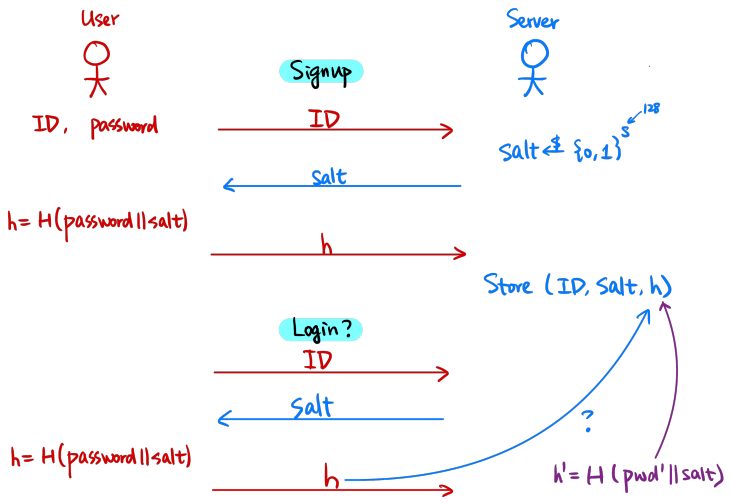


In this case, an adversary could launch an *Online Dictionary Attack* and try a lot of passwords with the server.

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

How can we prevent this?

§8.2.1 Salting



One way of ensuring that the hashing is non-deterministic is for servers to generate a salt $\xleftarrow{\$} \{0,1\}^s$ and send it to the user. The user will hash $H(\text{password}||\text{salt})$ and send that to the server. The server stores a database of $(\text{ID}, \text{salt}, \text{h})$.

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

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

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

Additionally, one strategy to make it *even harder* for an adversary is to make hashing more difficult (time-consuming). For example, we can compose SHA256 in certain ways[21]. 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.

[21] 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.

§8.2.2 Two-Factor Authentication

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

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

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