# CS 255: Introduction to Cryptography

Samuel Wong
Department of Statistics
Stanford University

**Abstract**

The course starts with a history of cryptography including ciphers such as the Caesar cipher and the one time pad. PRGs are discussed in the context of creating stream ciphers. The security of PRFs and PRPs are examined, as well as how they are used in encryption schemes that use these block ciphers and transform them into stream ciphers. We review confidentiality as well as message integrity through the use of hash functions and MACs. The last portion of symmetric encryption is put together by combining these two concepts into authenticated encryption. The second part of the course deals with public key encryption and its use in key exchange, digital signatures, and certificates. Lastly, these cryptographic primitives are used in the discussion of identification protocols, authenticated key exchange, and zero knowledge protocols.

## Contents

# 1 Introduction

Cryptography is everywhere is our lives today. It ranges from secure communication (HTTPS, WiFi) to encrypting files on disk to user authentication and much more. The three steps in cryptography are

- Precisely specify threat model
- Propose a construction
- Prove that breaking construction under threat model will solve an underlying hard problem

In cryptography, we want to send a message between two or more parties without allowing an adversary to be able to read the message. We can do so using a cipher. Informally, a cipher is an encryption scheme that allows us to send this message using a secret key that protects us from the adversary.

Some historic examples of ciphers that have been used (now all badly broken) include the Substitution cipher, the Caesar cipher, and the Vigener cipher.

In the Substitution cipher, each letter of the alphabet is substituted with a letter or the alphabet. For example, maybe the letter "a" maps to "c", "b" maps to "w", "c" maps to "n" and "z" maps to "a". Under this mapping, if we encoded the message "bcza" under the substitution cipher, we would get "wnac." When we decode "wnac," we obtain the original message again, "bcza."

If we denote one key as one set of mappings, then the size of the key space in the substitution cipher is $26! \approx 2^{88}$. There are 26! different possible keys that can be used in the substitution cipher. Given this size of the key space, is the substitution cipher a good cipher? The answer is no!

This cipher can be broken by looking at the frequency of the occurrence of the letters. We know that statistically the most common letters in the English alphabet are "e", "t", and "a", so we can figure out what those map to just be looking at the frequency of the letters in the ciphertext (the text after the substitution). Then we use frequency of pairs of letters (digrams) to figure out more mappings, etc. Thus, the substitution cipher is susceptible to a ciphertext (CT) only attack, meaning that an adversary can decode the message using only the information in the ciphertext - this is bad!

The Caesar cipher is an even worse variation of the substitution cipher, where each letter maps to the letter 3 to the right of it. For example, "a" maps to "d", "b" maps to "e", and "z" maps to "c". The key space in the Caesar cipher is of size 1, and it can be easily broken.

## 1.1 XOR Review

In cryptography, the XOR function is commonly used. The XOR of two strings in $\{0,1\}^n$ is their bit-wise addition mod 2. The symbol that will be used for XOR is $\oplus$. For example, if $x = 0110111$ and $y = 1011010$, then $x \oplus y = 1101101$. The following are some important properties of XOR:

- Commutative: $A \oplus B = B \oplus A$
- Associative: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- Identity: $A \oplus 0 = A$
- Self-inverse: $A \oplus A = 0$

One very important property of XOR particularly used in cryptography is the following theorem. Let $Y$ be a random variable over $\{0,1\}^n$ coming from any probability distribution. Let $X$ be a random variable that is independent from $Y$ and that $X$ is uniformly distributed on $\{0,1\}^n$. Then, $Z := Y \oplus X$ is a uniform variable on $\{0,1\}^n$.

The proof of this theorem is as follows (for $n = 1$). For $Y$, the denote the probability of 0 as $p_0$ and the probability of 1 as $p_1$. We know that $p_0 + p_1 = 1$ since $Y$ can only take on those two values. For $X$, the probability of 0 is 0.5 as is the probability of 1. Thus, since $X$ and $Y$ are independent, the probability under the joint distribution of seeing $X = 0, Y = 0$ is $p_0 * 0.5$, the probability of $X = 0, Y = 1$ is $p_1 * 0.5$, the probability of $X = 1, Y = 0$ is $p_0 * 0.5$, and the probability of $X = 1, Y = 1$ is $p_1 * 0.5$.

Now we want to find the probability that $Z = 0$. $P(Z = 0) = P((X,Y) = (0,0) \cup (X,Y) = (1,1)) = P((X,Y) = (0,0)) + P((X,Y) = (1,1))$ (since $X, Y$ are disjoint), which equals $p_0 * 0.5 + p_1 * 0.5 = 0.5$. Thus the probability of $Z = 0$ is 0.5 and similarly, the probability of $Z = 1$ is 0.5. Thus $Z$ is a uniform random variable over $\{0,1\}$. This proof can be extended to $n > 1$.

## 1.2 Definitions

The basic mechanism for encrypting a messaged using a shared secret key is called a cipher (or encryption scheme). A more simplified notion of a cipher, the Shannon cipher, is a pair $\mathcal{E} = (E, D)$. The function $E$, the encryption function, takes as input a key $k$ and a message $m$ (plaintext) and produces as output as ciphertext $c$. Thus, we can write $c = E(k, m)$.

The function $D$, the decryption function, takes as input a key $k$ and a ciphertext $c$ and produces a message $m$. Thus, we can write $m = D(k, c)$. Note that the key $k$ need not be the same in encryption and decryption, but when they are the same, it is called a symmetric cipher.

A more formal definition of a symmetric cipher is as follows:

A symmetric cipher defined over the space $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ is a pair of "efficient" algorithms $(E, D)$ where $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ and $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$, such that for all $m \in \mathcal{M}$ and $k \in \mathcal{K}$, that $D(k, E(k, m)) = m$.

This definition above requires that our decryption "undoes" the encryption, and therefore is satisfies the correctness property. Note that $E$ can be randomized (it often is randomized), but $D$ is always deterministic.

We have defined what a cipher (and more specifically what a symmetric cipher) is, but we have not yet defined if the cipher is secure or not (how good it is). One definition of security comes from Information Theoretic Security (Shannon 1949).

A cipher $(E, D)$ over the space $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ has perfect secrecy if for any $m_0, m_1 \in \mathcal{M}$ where $len(m_0) = len(m_1)$ and $c \in \mathcal{C}$, then $P[E(k, m_0) = c] = P[E(k, m_1) = c]$ where $k$ is uniform in $\mathcal{K}$.

In other words, given the ciphertext (CT) only, the adversary would not be able to tell at all if the plaintext (PT) message is $m_0$ or $m_1$ for any $m_0, m_1$. It doesn't matter how powerful the adversary is, they cannot perform a CT only attack (however, other attacks are possible).

## 1.3 One Time Pad (OTP)

The One Time Pad is a Shannon cipher where $\mathcal{K} := \mathcal{M} := \mathcal{C} = \{0, 1\}^n$. Thus the secret key is a random bit string of length $n$, and the message is also of length $n$. In one time pad, $E(k, m) = k \oplus m$, and $D(k, c) = k \oplus c$. For example, if the message was 0100110 and the key was 1101100, then $c = E(k, m) = 1101100 \oplus 0100110 = 1001010$. To decode this, we would have $m = D(k, c) = 1101100 \oplus 1001010 = 0100110$.

We can prove that the one time pad satisfies the correctness property in the general case.

$D(k, E(k, m)) = D(k, k \oplus m)$
$= k \oplus (k \oplus m)$
$= (k \oplus k) \oplus m$
$= 0 \oplus m$
$= m$

A nice property of the one time pad is that it is very fast to encode and decode, all we need to do is XOR. However, the problem with the one time pad is that it requires a very long key (as long as the message itself). Imagine encrypting a 4GB movie. The person decoding this would need to be sent a 4GB key just to decode the cipher text. This is not practical as if we had a secure way of sending the key to the other party, we might as well have just sent the movie itself.

Typically, the goal is to keep a key to 128 bits (16 bytes) or 256 bits (32 bytes). This is because an exhaustive search of all keys of this length would take longer than the age of the universe. For standard computers, running through $2^{128}$ possibilities for keys would take longer than the age of the universe, and for quantum computers, we need to bump that up to $2^{256}$.

The one time pad also satisfies perfect secrecy. We can prove this by the following. $P(E(k, m) = c) = \frac{\#keys\ k \in \mathcal{K}\ st\ E(k,m)=c}{|\mathcal{K}|} = \frac{\#keys\ k \in \mathcal{K}\ st\ k \oplus m = c}{|\mathcal{K}|} = \frac{1}{|\mathcal{K}|}$. The reason we know that there is only 1 key such that $k \oplus m = c$ is because $k \oplus m = c$ implies that $k = m \oplus c$ (we can see this by XOR-ing m on both sides). Therefore, there is one unique value of $k$ and thus one key.

Therefore, $P(E(k, m) = c) = \frac{1}{|\mathcal{K}|} = \frac{1}{2^n}$. So no matter which $m_0, m_1 \in \mathcal{M}$ and $c \in \mathcal{C}$, the probability is the same, so the adversary learns nothing about the message or the key.

Again, although one time pad is secure, its downside is that the key needs to be the same length as the message, which is impractical. Thus, the logical question is to find out if there is another cipher that satisfies perfect secrecy with shorter keys? Sadly, the answer is no. Shannon proved that to achieve perfect secrecy, the key must be at least as long as the length of the message. More specifically, $|\mathcal{K}| \geq |\mathcal{M}|$, which implies that the length of the key must greater or equal to the length of the messages.

The intuition behind the proof is as follows. For a fixed message length, denote the different messages as $m_1, m_2, ..., m_{2^n}$. The adversary sees the ciphertext $c$. For perfect secrecy to hold, we know that there must exist a key that encrypts the message $m_i$ to the ciphertext $c$. This is because suppose there does not exist a key that encrypts one of the messages $m_i$ to $c$. Then an adversary has learned something about the plaintext when they intercept $c$, i.e. that it cannot be message $m_i$. Therefore, we conclude that for every message, there must exist a key that maps it to $c$. Now, these keys must be distinct, so there must be at least $2^n$ keys. Thus, $|\mathcal{K}| \geq |\mathcal{M}|$.

# 2 Stream Ciphers

A stream cipher modifies the one-time pad using a function $G$, called a pseudorandom generator (PRG). Since in the one-time pad, the key is as long as the message, the goal is to use a shorter key. In doing so, in stream ciphers, we replace this "random" key with a "pseudorandom" key, which is created by $G$. The process is as follows. We use a short, $s$-bit "seed", $k$, as the encryption key, and stretch this to the length of the message by using $G$. We then can use $G(k)$ as our key for the one-time pad. Therefore, a stream cipher has $\mathcal{K} = \{0,1\}^s$, $\mathcal{M} = C = \{0,1\}^n$. In a stream cipher, $E(k,m) = G(k) \oplus m$, and $D(k,c) = G(k) \oplus c$. Note that $G$ is an "efficient", deterministic algorithm. Since by the lemma we saw before, the stream cipher cannot achieve perfect secrecy since $s < n$, but we can have a different definition of security called semantic security that can be satisfied.

## 2.1 Pseudorandom Generators (PRG)

Since our PRG function $G$ maps the shorter key $k$ to the $n$-bit long $G(k)$ which is used to encrypt the message, it is logical to reason that the security of this cipher depends on properties of the function $G$. As stated before, $G$ must be an "efficient", deterministic algorithm.

For a stream cipher to be secure, the PRG $G$ must be unpredictable. The definition of unpredictable is that for all $i = 1,..,n$, for all "efficient" algorithms $\mathcal{A}$, if $k \overset{R}{\leftarrow} \{0,1\}^s$, then $\mathcal{A}$ cannot compute $G(k)[i]$ just given $G(k)[0,...,i-1]$. In other words, if we select a key $k$ uniformly at random from the key space, if an adversary obtains the first $i-1$ bits of $G(k)$, using any efficient algorithm, the adversary cannot predict any of the next bits of $G(k)$. Mathematically, this means that

$|P[\mathcal{A}(G(k)[0,...,i-1]) = G(k)[i]] - \frac{1}{2}| \leq$ "negligible" ($2^{-80}$ in practice)

This means that the probability that an adversary can guess the next bit given the previous bits is not more than $\frac{1}{2}$ (random guessing).

Why this is required is as follows. Suppose $G$ is predictable, so that given the first $i-1$ bits of $G(k)$, an adversary can predict $G(k)[i]$, therefore recover the entire $G(k)$, then it is not secure. This is because an adversary can often guess the first few characters of a message based on prior knowledge (for example in an SMTP message, the first 5 characters are "FROM:"). Then, we know that for a stream cipher $m \oplus G(k) = c$, that is equivalent to $c \oplus m = G(k)$. Thus, if the adversary can guess the first few characters of the message, then they can XOR the message with the ciphertext $c$, and obtain the first few bits of $G(k)$. If $G$ is predictable, then the adversary can obtain the rest of $G(k)$ given the first few bits of $G(k)$. Then the adversary can recover the full message by doing $c \oplus G(k)$, thus it is not secure.

Some examples of PRGs that should never be used for crypto because they are linear congruential generators (and thus predictable)

- glibc: rand()
- Java: math.random()
- Microsoft Visual Basic: RND()

The formal definition of a secure PRG is as follows. A PRG $G : S \to R$ (where $S$ is the seed space and $R$ is an output space) is secure if $\{k \overset{R}{\leftarrow} S : G(k)\}$ is indistinguishable from $\{r \overset{R}{\leftarrow} R\}$. In other words, if we select a seed uniformly at random, and calculate $G(k)$, it will be indistinguishable from just selecting an $n$-bit string uniformly as random. How do we define "indistinguishable?" The distributions $P_1, P_2$ over a space $\Omega$ are defined to be indistinguishable ($P_1 \overset{P}{\approx} P_2$) if for all efficient algorithms $\mathcal{A}$ (where it outputs either 0 or 1), that

$|P[x \overset{R}{\leftarrow} P_1 : \mathcal{A}(x) = 1] - P[x \overset{R}{\leftarrow} P_2 : \mathcal{A}(x) = 1]| <$ "negligible" ($2^{-80}$ in practice)

This means that every efficient algorithm $\mathcal{A}$ behaves the same on $P_1$ as on $P_2$. Why is it important to define security? It is because it can be proven that a secure PRG implies that a PRG is unpredictable. Intuitively this is because if a PRG is secure, then we cannot distinguish it from a truly random string. In a truly random string, given the first $i-1$ bits, we have no better way than random guessing to predict the $i$ bit, and thus it is unpredictable. Thus, whenever a PRG is secure, we can treat $G(k)$ as if it were uniform over all of $R$. Thus, we can use a secure PRG to generate $G(k)$ as a key, and since it behaves as though we had chosen a random $n$-bit string, we can use it as the key for the one-time pad. Hence, for stream ciphers, using a secure PRG, we are protected against one-time CT only attacks.

## 2.2 Real World PRGs

The first PRG we'll look at is RC4, which was used in the 802.11b WEP. The problem with this PRG is that the first 256 bytes of the output are biased. In fact, the if it were truly secure, then the probability of each byte taking on a particular value in the range of 0 to 255 is $\frac{1}{256}$. As it turns out, under RC4, the probability of the second byte of output of $RC4(k) = 0$ is $\frac{2}{256}$.

What happens is that we take a message $m_1$ (could be a password or cookie) and XOR it with $RC4(k_1)$ to get $c_1$. The next time we use this, we take the same message $m_1$ and XOR it with $RC4(k_2)$ to get $c_2$. If we send this many times, the adversary can just look at the most common second byte value, and that will be the second byte of $m_1$. As it turns out, after about 10,000 samples, the probability that the most common second byte of $c_i$'s is the most common second byte of $m_1$ is over 90%! In addition, it turns out that other bytes are also biased using $RC4$.

The second PRG, which is a secure PRG, is called ChaCha20. It is quite fast on 64-bit architecture, and today is widely used on Android. ChaCha20 uses a fixed one-to-one (permutation) function $\Pi : \{0,1\}^{512} \to \{0,1\}^{512}$. We also have a seed, $s \in \{0,1\}^{256}$ as well as as counter, $j \in \{0, 1, ..., 2^{64} - 1\}$. There is a padding function that takes as input $s$ and $j$, that outputs $\{0,1\}^{512}$. We want to create many 512-bit blocks. We create a 512-bit block by running the padding function with $s$ and counter $j_i$, call this $m_i$. We then XOR $m_i$ with $\Pi(m_i)$ to obtain the $i$th 512 bit block. This block is the first 512 bits that we will use to encrypt (using XOR) the first 512 bits of our message. If our message is longer, we can do this again. Analysis shows that ChaCha20 is a secure PRG when $\Pi$ is a random permutation.

The third PRG, which is not a secure PRG, is called CSS and was used to encrypt CDs. Previously this was used in hardware as well in software, but it was been broken and should not be used anymore.

## 2.3 Attacks on OTP and Stream Ciphers

The first attack is the two-time pad. This attack demonstrates that the one-time pad key or the stream cipher key $G(k)$ should only be used one time to encrypt a message. Say we send two different messages with the same key $k$. Then we have that $m_1 \oplus k = c_1$ and that $m_2 \oplus k = c_2$. If an adversary obtains both ciphertexts, they can XOR them together and get $c_1 \oplus c_2 = m_1 \oplus m_2$. Given this result and domain knowledge of the English language, the adversary can recover both $m_1$ and $m_2$. Thus, the stream cipher key should ONLY be used once!

There are real life mistakes that have allowed adversaries to use the two time pad attack. For example, it used to be that when a client connected with a server, the client would send messages to the server encrypted with a key $k$ and the server would send messages back to the client encrypted using the same key $k$. This allowed for the two time pad attack.

The second attack is called a no integrity attack, and in this attack, the adversary is assumed to be an active adversary. For example, if Alice is sending a message to the bank, the adversary can intercept the CT of the message, change the message, and then continue sending it to the bank.

We can see this as follows. Alice sends a message encrypted with $G(k)$, thus the attacker sees $c = m \oplus G(k)$. The attacked then XORs this result with his own message, call it $\Delta$. Then when the bank decrypts this tampered-with CT, it gets $[(m \oplus G(k)) \oplus \Delta] \oplus G(k) = m \oplus \Delta$. One example of how this would be effective is if the adversary knows that Alice is sending a message to the bank that says "Pay Bob \$100". The adversary wants to change the message to say "Pay Eve \$100". Thus the adversary can calculate that "Bob" is $0x426F62$ and "Eve" is $0x457665$. XORing these two together the adversary gets $0x071907$, which would be $\Delta$. Thus if the adversary intercepts this message and XORs it with $\Delta$ before sending it to the bank, the bank will send Eve 100 dollars instead of Bob.

## 3 Block Ciphers

Another type of cipher instead of the stream cipher is called a block cipher. A block cipher differs from a stream cipher in that the cipher takes in a fixed length ($n$-bits) input and outputs a fixed length output (same length as input). More formally, a block cipher is a deterministic cipher $\mathcal{E} = (E, D)$ whose message space and ciphertext space are the same finite set $\mathcal{X}$. Here we use the notation $\mathcal{X}$ instead of message space $\mathcal{M}$ since the input may not be the entire message. As usual, the block ciphers encryption and decryption algorithms must be able to be computed "efficiently." Two examples of block ciphers are AES and 3DES. In AES, $n = 128$ bits, and depending on the version of AES, the length of the key $k = 128$, 192, or 256 bits. For example in AES128, we have that $\mathcal{K} \times \mathcal{X} \rightarrow \mathcal{X}$, where $\mathcal{K} = \mathcal{X} = \{0,1\}^{128}$. For 3DES, we have that $\mathcal{K} \times \mathcal{X} \rightarrow \mathcal{X}$, where $\mathcal{K} = \{0,1\}^{128}$ and $\mathcal{X} = \{0,1\}^{64}$.

Block ciphers are built by iteration. How this works is that the key $k$ is transformed into many different keys $k_1, k_2, ..., k_n$ using key expansion. The initial $n$-bit message is fed into a round function $R(k, m)$ along with $k_1$. This is considered one round. The output of this first round is then fed into the same round function but this time with $k_2$, which is the second round etc. This continues until the output of the last round, which is the ciphertext. 3DES has 48 rounds, AES128 has 10 rounds, AES192 has 12 rounds, and AES256 has 14 rounds. Note that to decrypt, we can just apply the inverted round functions in the opposite order, starting with $R^{-1}(k_n, c)$ and the output at the end will be the plaintext.

Let's look into how AES works more closely. For AES, since it has 10 rounds, it takes the initial key, and through key expansion, creates $k_0, k_1, k_2, ..., k_{10}$. It then takes the input and XORs it with $k_0$. This is then fed into the round function, which is a permutation (which is invertible by definition), $\Pi$. The output is then XORed with $k_1$ and then fed into $\Pi$. This is repeated until the the last $\Pi$ and then that output is XOred with $k_{10}$, and that result is the ciphertext. Each $\Pi$ consists of 3 steps, a ByteSub, a Shiftrow, and a MixColumn, except for the last $\Pi$, which does not have a MixColumn but has the other two. The input message is 16-bits in AES, which is stored in a 4 by 4 block, and same for the key (and keys generated from the key expansion). In ByteSub, certain bytes are just substitued with other bytes according to a lookup table. In ShiftRows, for each of the 4 rows (row = 0 to row = 3), the elements in that row are rotated left by the row number of spaces. So the first row remains the same, in the second row all elements are shifted left by 1 space (where the leftmost element wraps around and becomes the rightmost element) and etc for the other 2 rows. In MixColumns, a linear transformation is applied independently for each one of the columns (a matrix is multiplied to each column). It is important to once again note that these moves are all deterministic.

It is difficult to implement AES so one should not try and implement it by oneself, but rather call on existing crypto libraries. These days, some libraries that take advantage of parallelism and pipelining can compute AES very quickly.

### 3.1 PRFs and PRPs

It helps to take a step back and understand more abstractly what PRFs and PRPs are before continuing studying block ciphers. A PRF (pseudo random function) is a function defined over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$ where $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ such that there is an efficient algorithm to evaluate $F(k, x)$. Note that the PRF definition is loose, it does not require that $F$ be invertible, nor require that the input and output space be of the same length.

A PRP (pseudo random permutation) is a function defined $(\mathcal{K}, \mathcal{X})$ where $E : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{X}$ such that

- There exists an "efficient" algorithm to evaluate $E(k, x)$
- The function $E(k, \cdot)$ is one-to-one
- There exists an "efficient" inversion algorithm $D(k, x)$

Therefore a PRP is a type (subset) of PRF. PRPs are also called block ciphers. Therefore, in the previous discussion about block ciphers: 3DES, AES, those are all PRPs (note how they are all one-to-one and invertible).

#### 3.1.1 Security of PRFs and PRPs

Now that we have defined PRFs and PRPs, we want to have a notion of what is a secure PRF and a secure PRP. We want to formalize a definition for this. With PRF, we create a set called Funs[$\mathcal{X}$, $\mathcal{Y}$], which is the set of all functions that map from $\mathcal{X}$ to $\mathcal{Y}$. This size of this set is $|\mathcal{Y}|^{|\mathcal{X}|}$, since that is how many mappings there are. We then define another set using our PRF. We define this set as $S_F = \{F(k, \cdot) \text{ such that } k \in K\}$, which is clearly a subset of Funs[$\mathcal{X}$, $\mathcal{Y}$]. The size of this set is much smaller, it is $|\mathcal{K}|$ since we have one set of mappings for our PRF for each key $k$.

The intuition is as follows: a PRF is secure if a random function in Funs[$\mathcal{X}$, $\mathcal{Y}$] is indistinguishable from a random function in $S_F$. We make this rigorous by using an experiment.

In an experiment, there are two possible scenarios, we are either in experiment 0 or experiment 1 ($b = 0, 1$). If we are in experiment 0, the challenger selects $k \overset{R}{\leftarrow} K, f \leftarrow F(k, \cdot)$. If we are in experiment 1, the challenger selects $f \overset{R}{\leftarrow}$ Funs[$\mathcal{X}$, $\mathcal{Y}$]. The adversary $\mathcal{A}$ will try to figure out which experiment we are in. The adversary will send queries $x_i \in \mathcal{X}$ to the challenger and receive $f(x_i)$ as a response. The adversary can send as many queries as they want, and after some time, they will try to guess if $b = 0$ or $b = 1$. The definition is as follows: $F$ is a secure PRF if for all "efficient" $\mathcal{A}$, $Adv_{PRF}[A, F] = |P(EXP(0) = 1) - P(EXP(1) = 1)|$ is "negligible." This means that the adversary cannot tell if we are in experiment 0 or experiment 1.

For example, let $F(k, x) = k \oplus x$. If adversary $\mathcal{A}$ chooses $x_0$ and $x_1$ as its queries ($x_0 \neq x_1$), it will obtain $y_0 = k \oplus x_0$ and $y_1 = k \oplus x_1$ from the challenger. The adversary can XOR these two results, so it will have $y_0 \oplus y_1$. The adversary can then do $x_0 \oplus x_1$, and if the two match, then the adversary will output 0, otherwise 1. We can calculate the advantage as $Adv_{PRF}[A, F] = |P(EXP(0) = 1) - P(EXP(1) = 1)| = |0 - (1 - \frac{1}{2^n})| = 1 - \frac{1}{2^n}$, which is non-negligible. Thus, this PRF is insecure. The reason why $P(EXP(1) = 1)$ has a very small probability and it not 0 is because even if we drew from the set Funs[$\mathcal{X}$, $\mathcal{Y}$], there is a very small probability that we could have been very lucky and drew values that matched.

Defining a secure PRP is essenitally the same as a secure PRF except instead of using the set of all functions: Funs[$\mathcal{X}$, $\mathcal{Y}$], we use the set of all permutations: Perms[$\mathcal{X}$] (a subset of all functions, only the one-to-one invertible ones of the same length). Thus, the advantage in a PRP setting is similarly calculated as $Adv_{PRP}[A, E] = |P(EXP(0) = 1) - P(EXP(1) = 1)|$.

The PRPs that were previously discussed, 3DES and AES fulfill the definition and are secure PRPs.

## 3.2 Semantic Security

Now that we have defined a PRF and a PRP, which are functions, we want to find out how to use these functions in an encryption scheme. For example, we could have a secure PRP such as AES, but it takes in 128 bits. So if we have a long message, the encryption scheme involves how to use AES and apply it to this long message.

For example, one bad way to do it would be to encrypt each 128 bit message with AES in sequence. This would be bad because if the attacker saw two ciphertexts that were the same, they would know that those two 128 bit blocks had the same message. Therefore the attacker learns something about the plaintext. This would be an example of a secure PRP that is used insecurely in the encryption scheme.

Therefore, to determine whether or not an encryption scheme (cipher) is secure, we define the notion of semantic security. We make this definition rigorous by using an experiment. In this experiment, we have a challenger and an attacker and a cipher $(E, D)$. The attacker is allowed to send (one-time) two messages $m_0$ and $m_1$ of the same length to the challenger.

In this experiment, there are two possible scenarios, we are either in experiment 0 or experiment 1 ($b = 0, 1$). If we are in experiment 0, the challenger encrypts $m_0$ and returns $E(k, m_0)$. If we are in experiment 1, the challenger encrypts $m_1$ and returns $E(k, m_1)$ to the adversary. The adversary $\mathcal{A}$ will try to figure out which experiment we are in ($b = 0$ or $b = 1$ from receiving this output). The definition is as follows: $E$ is semantically secure if for all "efficient" $\mathcal{A}$, $Adv_{SS}[A, E] = |P(EXP(0) = 1) - P(EXP(1) = 1)|$ is "negligible." This means that the adversary cannot tell if we are in experiment 0 or experiment 1.

In semantic security, no "efficient" adversary can learn any information about the plaintext from a SINGLE ciphertext.

## 3.3 Secure Constructions

### 3.3.1 Deterministic Counter Mode

From the previous section, there was an example where one bad way to use a PRF/PRP is to encrypt each $n$-bit message in sequence with the same PRF/PRP. This was bad because if the attacker saw two ciphertexts that were the same, they would know that those two $n$-bit blocks had the same original message.

One way to avoid this is to turn a PRF into a PRG. For example, instead of using the PRF, $F$, for each $n$-bit block of the message, we instead create $F(k, 0), F(k, 1), ...$ etc. and then XOR these blocks with the message. Essentially, we are constructing a one-time pad using a counter from 0 to however many we need to generate a pseudorandom key to XOR with the message. We are building a stream cipher from a PRF (eg. AES, 3DES). This methodology is called deterministic counter mode.

There is a theorem that states the security of deterministic counter mode (abbreviated DETCTR). This theorem states that for any number of $n$-bit blocks of the message, if $F$ is a secure PRF over $(\mathcal{K}, \mathcal{X}, \mathcal{X})$, then $E_{DETCTR}$ is a semantically secure cipher over $(\mathcal{K}, \mathcal{X}^L, \mathcal{X}^L)$. In particular, for any adversary $\mathcal{A}$ attacking $E_{DETCTR}$, there exists a PRF adversary $\mathcal{B}$ such that $Adv_{SS}[A, E_{DETCTR}] = 2 * Adv_{PRF}[B, F]$. Since we know that $Adv_{PRF}[B, F]$ is negligible for functions such as 3DES and AES, then $Adv_{SS}[A, E_{DETCTR}]$ must be negligible and therefore is a secure cipher.

### 3.3.2 Many Time Key

Although deterministic counter mode works for one time, we know that by converting the block cipher into a stream cipher and doing a one-time pad, it is no longer secure for multiple uses. We may want to reuse the same key many times, and therefore want to be able to use an algorithm that still maintains security when a key is used multiple times.

Therefore, we want to define a notion of security for a many-time key. We will extend our notion of semantic security (adversary can only query one time) to CPA security (adversary can query as many times as they like). All aspects of the semantic security game are the same, except now the adversary can query as many times as they would like. For a cipher to be CPA secure, the adversary needs to have a negligible advantage even after all these queries.

We have seen that stream ciphers are insecure under CPA. More generally, if $E(k, m)$ always produces the same ciphertext, then the cipher is insecure under CPA. Therefore, if a secret key is to be used multiple times, to be secure, when given the same plaintext message twice, the encryption algorithm must produce different outputs.

## 3.4 Nonce Based Encryption

We saw that if the desire is to have a secure cipher that uses the same key more than one time, that given the same plaintext message twice, the encryption algorithm produces two different ciphertexts. But how can we do this if the encryption algorithm is deterministic? The answer is to use a third argument for the encryption, called a nonce. Therefore, now our encryption algorithm looks like $E(k, m, n) = c$ and our decryption algorithm looks like $E(k, m, n) = m$, where $n$ is the nonce.

A nonce is a value that changes from message to message. The key, nonce pair should never be used more than once, otherwise we will be in the same situation as the one time pad, which we know is insecure when used more than once. However, the nonce can be public. Even if the attacker knows the nonce, if they do not know the secret key, they cannot break the cipher. Two common ways of choosing the nonce are as follows. One way is that the encryptor chooses a random nonce $n \overset{R}{\leftarrow} \mathcal{N}$. This nonce would then be sent to the decryptor along with the ciphertext. The second way is that the nonce is a counter that increases by 1 after every encryption. Sometimes when using the counter this way, the nonce doesn't even need to be explicitly sent to the decryptor because it may be implicit. For example, in TLS, both computers on the network are aware what the packet number is, which can be used as the nonce.

### 3.4.1 CBC with Nonce

This example is one that should not be used in practice (because it is non parallelizable). How it works is that the message is broken up in $n$-bit lengths, $m[0], m[1], ..., m[n]$. Then a nonce is chosen at random and the nonce is XORed with $m[0]$ and the result is put into an encryption algorithm, $E(k_1, \cdot)$. The output is $c[0]$. This $c[0]$ then gets XORed with $m[1]$ and that is put into the encryption algorithm to get $c[1]$. This process continues until all the plaintext has been converted into ciphertext.

For CBC, it can be proven that for any number of $n$-bit blocks of the message, denote it $L$, then if $E$ is a secure PRP over $(\mathcal{K}, \mathcal{X})$, then $E_{CBC}$ is CPA secure over $(\mathcal{K}, \mathcal{X}^L, \mathcal{X}^{L+1})$. In particular, for a $q$-query adversary $\mathcal{A}$ attacking $E_{CBC}$, there exists a PRP adversary $\mathcal{B}$ such that $Adv_{CPA}[A, E_{CBC}] \leq 2 * Adv_{PRP}[B, E] + 2\frac{q^2 L^2}{|X|}$. Therefore, if we use a secure PRP, then CBC is secure as long as $q^2 L^2 << |X|$. The above methodology is called CBC with random nonce.

CBC can be done with a counter as well. In this case, rather than starting off by XORing $m[0]$ with the nonce, we would first put the nonce into $E(k_2, n)$, and then take that output and XOR with $m[0]$. The rest of the process is the same. Note that the key used to encrypt the nonce must be the different key $k_2$ that the key used to encrypt the rest of steps, $k_1$. It can be shown that if the same key is used for both, it is insecure.

One disadvantage of CBC is also that if the last message block is not exact $n$-bits, we need to pad it with additional bytes, which can cause overhead.

### 3.4.2 Nonce Counter Mode

A much better way to encrypt rather than CBC is to use a nonce counter mode. How this works is the following. Let's say the message we want to encrypt is in 128-bit blocks. Then we create an IV, which is a 128-bit block, where the first 96 bits are a nonce and the last 32 bits are a counter. For each document that we want to encrypt, we use a different nonce. For each block within the message, we will increment the counter by 1. Notice that every time we encrypt, our IV will be different. Thus our IV is our unique "nonce" in this example. Then we use our secure PRF to encrypt these IVs, one for each block of message that we have and this output will be XORed with the message to create the ciphertext. For example, the first 128-bits of the message will be the XOR of $m[0]$ and $F(k, IV)$. The second 128-bits of the message will be the XOR of $m[1]$ and $F(k, IV+1)$.

It can be shown that for any number of $n$-bit blocks of the message, denote it $L$, then if $F$ is a secure PRF over $(\mathcal{K}, \mathcal{X}, \mathcal{X})$, then $E_{RCTRMOD}$ is CPA secure over $(\mathcal{K}, \mathcal{X}^L, \mathcal{X}^{L+1})$. In particular, for a $q$-query adversary $\mathcal{A}$ attacking $E_{RCTRMOD}$, there exists a PRF adversary $\mathcal{B}$ such that $Adv_{CPA}[A, E_{RCTRMOD}] \leq 2 * Adv_{PRF}[B, F] + 2\frac{q^2 L}{|X|}$. Therefore, assuming we are using a secure PRF, counter mode is secure as long as $q^2 L << |X|$. Note that counter mode doesn't actually require a PRP and only a PRF because the function does not need to be invertible. All the happens is an XOR, so the decryptor can calculate $F$ on their own and just XOR it to get the original plaintext back again. Note also that Counter Mode has an advantage over CBC in terms of parallel processing, and the number of blocks before a key change is required.

It is important to note that semantic security, or even CPA security provides confidentiality, but does not protect data integrity. In other words, it ensures that the attacker does not gain any information, but does not prevent them from changing the bits.

Some categories of attacks that can be done:

- Linear attacks
- Side channel attacks
- Quantum attacks

# 4 Message Integrity

So far, the discussion around security has been only about confidentiality, so whether or not the adversary knows the plaintext from the ciphertext. However, another major issue is the issue of message integrity. Message integrity means that when A is sending something to B, the item that B receives is what A has intended. If an adversary intercepts the message halfway and changes it, we want B to able to figure out that the message was tampered with and is not the original. For simplicity, let's assume that the message A is sending to B is not secret (Linux distribution, an advertisement, etc.). So not worrying about confidentiality, we still want to know of B did receive the intended ad or correct Linux distribution (without a virus).

To achieve message integrity, we will use what is a called a tag $t$. The sender will use a signing algorithm $S(k, m)$ to produce a tag and append it to the end of the message. The receiver (who has the same secret key $k$) will use the verification algorithm $V(k, m, t)$, to determine if the message has integrity. This way, if the attacker intercepts the message and wants to replace the tag with his own, he needs to be able to calculate $t'$ from his new message $m'$, but will be unable to do so because the attacker has no access to the key $k$ and thus cannot create $S(k, m') = t'$.

## 4.1 MAC (Message Authentication Code)

A MAC is a pair of algorithms $(S, V)$ (where $S$ is a signing algorithm and $V$ is a verification algorithm) that is defined over $(\mathcal{K}, \mathcal{M}, \mathcal{T})$. Both $S$ and $V$ are required to be "efficient" and $S(k, m) = t \in \mathcal{T}$ and $V(k, m, t)$ returns either "yes" or "no". We require that the MAC must follow the correctness property: $V(k, m, S(k, m))$ is "yes". There are deterministic and randomized MAC systems. In a deterministic MAC system, given a key $k$ and message $m$, the tag $t$ will always be the same, but in a randomized MAC system, the output may be one of many possible tags.

So what is a secure MAC? To define security, we allow the attacker's power to be a "chosen message attack." This means that the attacker can choose the messages $m_1, ..., m_q$ and as a result he gets the output tags for those messages. To define security, we are going to play a game with a challenger and an adversary (similar to semantic security definition). In this game, the adversary gets to choose as many messages $m_i \in \mathcal{M}$, $i = 1, 2, ..., q$ and send them to the challenger. The challenger has secret key $k$ and then signs these messages $S(k, m_i)$ to produce tags $t_i$, $i = 1, 2, ..., q$ which are sent back to the adversary. The adversary wins the game if they can produce a valid $(m, t)$ pair where the message $m$ was not one of the ones that they had sent the challenger. This goal is called existential forgery. If the adversary is able to do so, that means that they are able to produce a tag for an unseen message, and therefore the MAC system is insecure. On the flipside, if the adversary is unable to do so, we say the MAC system is secure, because an adversary cannot produce a valid tag for any message $m \in \mathcal{M}$. More formally, $(S, V)$ is a secure MAC if for all efficient adversary $\mathcal{A}$, $Adv_{MAC}[\mathcal{A}, (S, V)] = \text{P(Adv wins)}$ is negligible. Note that for a non-deterministic MAC, the adversary wins the game if they can generate a different valid tag from one that they had previously seen even with one of the messages that they had previously sent.

We can construct secure MACs from secure PRFs. If $F$ is a secure PRF, then we can define our secure MAC as $S(k, m) = F(k, m)$ and $V(k, m, t)$ as "yes" if $t = F(k, m)$ and "no" otherwise. There is a theorem that states that if $F$ is a secure PRF over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$, and $\frac{1}{|Y|}$ is negligible, then $(S, V)$ is a secure MAC. In particular, for every MAC adversary $\mathcal{A}$, there is a PRF adversary $\mathcal{B}$ such that $Adv_{MAC}[\mathcal{A}, (S, V)] \leq Adv_{PRF}[B, F] + \frac{1}{|Y|}$. Therefore, if $Y$ is negligible and the PRF is secure, then the MAC is secure. Intuitively, the reason that $\frac{1}{|Y|}$ must be small is that if it is not, the adversary can randomly guess (or exhaustive search) through the options and get lucky.

Therefore, we know AES is a secure PRF, and thus AES gives a secure MAC for 16-byte messages (AES takes as input and returns as output $n = 128$ bits). We want to know if we can enhance this to create a secure MAC for messages longer than 16-bytes. To do this, we will use a small secure PRF to construct a larger secure PRF, and then we know that this larger PRF will then be able to be used as a larger MAC.

One remark is that truncating longer secure MACs to shorter lengths preserves the security as long as the length is is not too short. Taking a larger MAC and truncating the MAC to $w$ bits is OK as long as $\frac{1}{2^w}$ is considered negligible. So in the AES case, where it outputs 128 bits, we can truncate it to say 80 bits and it would still be secure.

### 4.1.1 CBC-MAC

To construct the CBC-MAC, we do the following. Let $F$ be a secure PRF such as AES (which takes in as input and outputs 128 bits). Our goal is now to create a new PRF $F_{CBC}$. How this works is assume we have a long message and we divide it into 128 bits blocks. We take the first block $m[0]$ and encrypt it using $F(k_1, m[0])$. We take this output and XOR it with $m[1]$. We then feed this through the PRF again, so $F(k_1, \cdot)$. This gets XORed with $m[2]$ and the process continues for the whole message. After the last message block has been XORed and run through $F(k_1, \cdot)$, then one of two things happens. In Raw CBC-MAC, this output is the tag $t$. In encrypted CBC-MAC, this output is the put into $F(k_2, \cdot)$ and then the output of that is the tag $t$. Raw CBC-MAC is not secure if we are considered messages of different lengths (only is secure for fixed size messages). Encrypted CBC-MAC is always secure.

In CBC-MAC, we have to deal with padding. If the message is not a multiple of a block size, what happens to the last block? We pad the last block with a 1 and then 0s after. If the message is a perfect multiple of the block size, then we append an extra block with 1 and then 0's after.

### 4.1.2 PMAC

One disadvantage of CBC-MAC is that it is sequential and thus will compute slowly. The natural next step is to see if it is possible to have a parallelizable PRF instead. PMAC (Parallel MAC) is a way to do this.

How PMAC works is we start again by taking a large message and dividing it up into smaller fixed size blocks $m[0], m[1], ....$ We also have a function $p$. Each message block is then XOR'ed with $p(k', counter)$. The output of each of these is applied to the PRF $F(k, \cdot)$. All of these outputs are then XORed together. This quantity is then fed into $F(k'', \cdot)$ and the output is the tag $t$. We can see how this is parallel because all of the $p$ functions can be computed simultaneously, as well as feeding the output of all of these into the $F$ function.

## 4.2 Collision Resistant Hashing

Let us define a hash function as $H : \mathcal{M} \to \mathcal{T}$ where the size of the message space is much larger than the size of the tag space. Then a collision of defined to be a pair of $m_0 \neq m_1$ such that $H(m_0) = H(m_1)$. Although since the tag space is much smaller than the message space, there will be lots of collisions, we don't want anyone to easily be able to find where these collisions occur.

A function $H : \mathcal{M} \to \mathcal{T}$ is CRH (collision resistant hash) if for all explicit "efficient" algorithms $\mathcal{A}$, the probability of finding a collision is negligible. Some examples of collision resistant hash functions are SHA256, SHA384, SHA512, or the more recent family, SHA3-256, SHA3-384, and SHA3-512.

An immediate application of CRH is to use it for MACs. Let's say we had a secure MAC $(S, V)$, but only takes in messages of a small size. Then if we want to apply this secure MAC to a large message, we can first hash the large message to turn it into a smaller size, and then feed this into the secure MAC that works for the smaller size input.

There is a theorem that states that if $(S, V)$ is a secure MAC, and $H$ is a CRH, then $S'(k, m) = S(k, H(m))$ and $V'(k, m, t) = V(k, H(m), t)$ is a secure MAC. Why is CRH necessary? Well, if the adversary could find two messages $m_0, m_1$ where $H(m_0) = H(m_1)$, then they could just request a tag on $m_0$, and then create an existential forgery using $m_1$ and the same tag received for $m_0$.

### 4.2.1 The Birthday Paradox

The birthday paradox states the following. Let $r_0, ..., r_n \in \{1, ..., B\}$ be independent uniform random variables. Then when $n \geq 1.2\sqrt{B}$, then $P(\exists i \neq j : r_i = r_j) \geq \frac{1}{2}$.

The reason the Birthday Paradox is important is because it is a generic attack on CRH. An adversary can choose random messages $m_0, m_1, ..., m_{2^l} \overset{R}{\leftarrow} \mathcal{M}$ and then compute the hashes of all these. Thus for time $2^{\frac{l}{2}}$, the Birthday Paradox tells us that the adversary will find a collision with probability $\geq \frac{1}{2}$. Therefore, for 128-bit hashes, the adversary has a $2^{64}$ time collision finder, which is computationally feasible. This is why the hashes need to be at least 256 bits. Note that the Birthday Paradox assumes a uniform distribution, which is actually the worst case scenario for the adversary. If the distribution is not uniform, the probability of a collision is even higher.

### 4.2.2 Merkle-Damgard

To construct CRH, we start with a Merkle-Damgard construction. How this works is that we have a compression function $h : \{0,1\}^b \times \mathcal{T} \to \mathcal{T}$. We break our message up into blocks of size $b$. We also randomized an initial value in the tag space $\mathcal{T}$. We then put the initial value and $m[0]$ into $h$. The output is then the tag, which along with $m[1]$ is put into $h$ again. This continues until the last block of the message, and the output from the last $h$ function is our final tag $t$. We will again pad the last message block with 1 and then 0's after, along with the message length as well.

There is an important theorem that states that if $h$ is a CRH, then $H_{MD}$ is a collision resistant hash. Therefore, if we choose a CRH as $h$, our whole Merkle-Damgard construction is a collision resistant hash as well, so we have now found a way to create a CRH that takes in larger input from a smaller CRH.

### 4.2.3 Davies-Meyer

So how do we build a compression function $h$ that is a CRH? This is created from a block cipher. Let $E(k, x)$ be a block cipher over $(\mathcal{K}, \mathcal{X})$ where $x = \{0, 1\}^n$. Then $h(m, c) := E(m, c) \oplus c$. The compression function is the block cipher that uses the message $m$ as a key and then the output of this block cipher is XORed with the "message" $c$ to produce the output. There is a theorem that states that if $E$ is an "ideal" block cipher, then finding a collision on $h$ takes time $\geq 2^{\frac{n}{2}}$.

SHA256 uses Merkle-Damgard with Davies-Meyer with a block cipher called SHACAL2 and it operates on a 512-bit key (meaning that the message is broken up into 512-bit blocks).

## 4.3 HMAC

What our goal is, is to build a PRF (and a MAC) from a CRH $H : \mathcal{M} \to \mathcal{T}$. Since a PRF takes in a key $k$, but a CRH does not have a key, how do we do this? One way that is INSECURE is to define $F(k, m) = H(k||m)$. Concatenation of the key and message is insecure of extension attacks on Merkle-Damgard hash functions. This is because Merkle-Damgard is sequential, so if we had $H(m)$, we can then take the MD and run it through some more iterations of $h$, using the concatenated part as the new message, and we would be able to create a valid tag for that new concatenated message.

The secure way to create a PRF from a CRH is with HMAC (Hash MAC). We define $F_{HMAC}(k, m) = H((k \oplus opad)||H((k \oplus ipad)||m))$.

There is a theorem that states that if a compression function $h(m, c)$ is a secure PRF (with either input as key), then HMAC is a secure PRF. Thus, we have successfully gone from a small PRF to a large PRF. With HMAC, we can now create tags from inputs of any size. Note that HMAC is very popular since crypto libraries already implement SHA256, so then HMAC implementation is just a few more lines of code.

We can compare the two different data integrity options: MAC or CRH. When using MAC, each file must be stored with it's corresponding MAC. The downside is that for a writer/verifier to read the tag, they need to have a copy of the key $k$. If the file is tampered with, the receiver will be able to tell that the tag is different when computing the verification algorithm with the key $k$. On the other hand, with CRH, no keys are necessary, so this would be good for a distribution of software. Anyone downloading it can view the tag and see if it matches to tag on the software distributor's website. However, what the distributor needs to do it protect the tag on their own website to make sure no attacker is writing the different tag to the website.

# 5 Authenticated Encryption

So far, we have examined ways to provide confidentiality by fulfulling CPA-security as well as ways to provide data integrity (through MACs). Authenticated encryption allows us to combine confidentiality and integrity.

How do we define if a cipher is AE-secure? We define this using a game. An adversary $\mathcal{A}$ can repeatedly send messages $m_i$ to a challenger. The challenger returns $c_i = E(k, m_i)$ back to the adversary. The adversary wins the game if they are able to successfully send the challenger a new $c$ which was not one of the previously returned $c_i$'s that is not malformed. In other words, the $c$ that the adversary sent to the challenger isn't "rejected" (it makes sense).

## 5.1 Constructions for AE using MAC and Cipher

Since we want authenticated encryption (AE), we can combine a MAC and a cipher to accomplish this. However, there are many ways of combining these two, and not all of these ways are secure. Let $k = (k_e, k_m)$, where $k_e$ is the key for the cipher and $k_m$ is the key for the MAC. One possible way to combine the cipher and the MAC (used in SSH) is in the encryption step, produce $t = S(k_m, m)$ and $c = E(k_e, m)$, and then send $c||t$. This does not fulfill AE, as the tag $t$ is sent over and there is no guarantee of CPA security for a MAC.

Another way (used in TLS 1.0) is during the encryption step, $t = S(k_m, m)$ and $c = E(k_e, m||t)$ and then send $c$. This has been shown to be vulnerable to the Poodle attack and is not secure.

One way that is secure (used in IPsec) is during the encryption step, to create $c = E(k_e, m)$ and $t = S(k_m, c)$ and then send $c||t$. It can be proven that this strategy provides authenticated encryption whenever $(E, D)$ is CPA-secure and $(S, V)$ is a secure MAC. One thing to remember is that $k_e$ and $k_m$ must be independent keys. One way this can be done is to start with a secret key $k$ and generated $k_e = PRF(k, "enc")$ and $k_m = PRF(k, "mac")$. That way, two parties only need to find a way to exchange one secret key $k$ and they can both generate two secret keys out of this one key.

The standard way to do authenticated encryption is to use GCM (Galois Counter Mode). How this works is first we use nonce-based counter mode and then use a MAC on the result. The typical MAC that GCM uses is called the Carter-Wegman MAC. Note that GCM is a requirement for TLS 1.3.

## 5.2 Authenticated Encryption with Associated Data (AEAD)

Often, we want to provide authenticated encryption, but there is metadata (associated data) that we do not need to have confidentiality, only integrity. In other words, we do not mind if an adversary sees this metadata, we only require that they do not change it.

Therefore, how AEAD works is that during encryption, we create $c = E(k_e, m)$ and we create that tag as $t = S(k_m, ad||c)$, where $ad$ is the associated data. We then send over $c||t$.

What is this metadata (associated data)? For example, in a filesystem, these could be fields such as the file name. In routing, these fields could be IP source address and destination address.

## 6 Basic Key Exchange

The previous sections have all addressed symmetric encryption. In what has been detailed above, we assume that the two parties, the encryptor and decryptor, have a shared secret key $k$. Assuming that they both had a secret key $k$ that no one else had access to, we discussed confidentiality, ciphertext integrity, and authenticated encryption. But how do the two parties obtain the key $k$ in the first place? In this section, we want to talk about how to safely allow two parties to obtain a key $k$, so that we can then use the methods that were discussed previously.

A first attempt at basic key exchange involves a trusted third party (TTP). Assume there are multiple parties that all want to send messages to each other. What this framework entails is that the TTP is the middleman between all the different parties. The different parties share a secret key with the TTP (for example, party A has $k_a$, party B has $k_b$ etc.) and the TTP has all the keys (one per party) $(k_a, k_b, ...)$. How these initial keys would be shared may be through an offline medium, such as writing it on a piece of paper. One these keys are established, then the parties can send messages to each other. For example if party A wants to send a message to party C, then they would encrypt with $k_a$ and send to the TTP. The TTP would decrypt with $k_a$ and encrypt the message with $k_c$ and send it to party C, would would be able to read the message by decrypting it with $k_c$. The problem with this is that the TTP has access to all the messages (and can read them) since it has all the keys.

The natural question then is to ask if we can perform key exchange without a TTP? It was shown by Merkle in 1974 that key exchange could be performed without a TTP using only symmetric ciphers. The problem, however, was that the two parties needed to exchange a large amount of data for this work, and this is totally impractical. Therefore, more practically, this is done not using symmetric ciphers, but with other methods. Most common in today's world are Diffie-Hellman, RSA, and Elliptic Curve Crypto (which is a variant of Diffie-Hellman). To understand how these work, we first need to understand some facts about number theory and group theory.

# 7 Number/Group Theory

## 7.1 Modular Arithmetic Rules

$a \equiv b(modn)$ can be rewritten as $a = kn + b$

- Reflexivity: $a \equiv a(mod)n$
- Symmetry: $a \equiv b(modn)$ if $b \equiv a(modn)$ for all $a, b, n$
- Transitivity: If $a \equiv b(modn)$ and $b \equiv c(modn)$, thne $a \equiv c(modn)$

Properties:

- $(a + b)modn = [amodn + bmodn]modn$
- $(a - b)modn = [amodn - bmodn]modn$
- $(a * b)modn = [amodn * bmodn]modn$
- $(g^a modn)^b modn = (g^b modn)^a modn = g^{ab} modn$ (used in Diffie-Hellman)

GCD:

- $gcd(a, b) = gcd(b, r)$; where $a = q * b + r$
- Euclid's algorithm: repeated iteration of the fact in the prior bullet to efficiently solve $gcd(a, b)$ even for large $a, b$
- Extended Euclidean algorithm: for all integers $a, b$, there exists some integers (denote them as $x, y$) such that $a*x+b*y = gcd(a, b)$. These integers $x, y$ can be found in time roughly quadratic in the logarithms of $a, b$

Denote the integers mod n as $\mathbb{Z}_n$.

Modular inversion:

- Definition: the inverse of $x \in \mathbb{Z}_n$ is $y \in \mathbb{Z}_n$ such that $x * y = 1$ in $\mathbb{Z}_n$. The inverse of $x$ is denoted $x^{-1}$
- Not all elements in $\mathbb{Z}_n$ must have an inverse. We know that $x \in \mathbb{Z}_n$ has an inverse iff $gcd(x, n) = 1$ ($x$ and $n$ are coprime)
- We can solve a linear equation in $\mathbb{Z}_n$ in the form $a * x + b = 0$ in quadratic log time since $x = -b * (a)^{-1}$ in $\mathbb{Z}_n$

Denote $\mathbb{Z}_n^*$ as the set of invertible elements in $\mathbb{Z}_n$. For any prime number $p$, $\mathbb{Z}_p^* = \mathbb{Z}_p \backslash \{0\} = \{1, 2, ..., p - 1\}$

Fermat's Little Theorem:

- Theorem: Let $p$ be a prime. Then for all $x \in \mathbb{Z}_p^*$, $x^{p-1} = 1$ in $\mathbb{Z}_p$
- Can be used to calculate inverses, since $x * x^{p-2} = x^{p-1} = 1$. Thus, $x$ and $x^{p-2}$ are inverses. However, this method of inversion is rarely used as it is slower than Extended Euclidean algorithm.
- Can be used to generate large primes with high probability. By guessing a large number, if $2^{p-1} = 1$ for that large number, there is an extraordinarily high probability that it is a prime number.
- Can solve problems such as $3^{2021} mod 11$. We know $3^{10} mod 11 = 1$ by Fermat's Little Theorem. Thus, $3^{2021} mod 11 = (((3^{10})^{202} * 3^1) mod 11) = 1^{202} * 3^1 = 3$

## 7.2 Group Theory

Definition: a Group is a set of elements $G$ and a binary operation $*$ such that:

- It is closed under the operation $*$: for all $x, y \in G$, $x * y \in G$
- For all $x \in G$, an inverse $x^{-1}$ exists in the group. $x * x^{-1} = e$, where $e$ is the unique identity element in the group
- $y * e = e * y = y$ for all $y \in G$
- Associativity holds: $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$

If a group also satisfies the commutative property, that $x * y = y * x$, then it is called an Abelian group.

A group $G$ is cyclic if there exists an element in the group that can generate the entire group $G$. Such a $g$ is called the generator of the group. The group $G$ is a finite cyclic group if the group is cyclic and there are a finite number of elements in the group. For example, in a multiplicative group, this means that there exists an element $g$ such that $\{1, g, g^2, g^3, ...\}$ is the entire group.

The order of an element $h$ in group $G$ is the size of the set than can be generated by $h$.

Lagrange's Theorem: for any finite group $G$, the order of every subgroup of $G$ divides the order of $G$.

### 7.2.1 Application to $\mathbb{Z}_p^*$

For a prime number $p$, we have that $\mathbb{Z}_p^*$ is a finite cyclic group. That means that there exists a generator $g$ such that $\{1, g, g^2, ..., g^{p-2}\} = \mathbb{Z}_p^*$. For example, when $p = 7$, $g = 3$ since $\{1, 3, 3^2, 3^3, 3^4, 3^5\} = \{1, 3, 2, 6, 4, 5\}$. Not all elements are generators. For example, $h = 2$ generates the set $\{1, 2, 4\}$. For all elements $h \in G$, $h^{order(h)} = 1$. This can easily be seen since after raising $h$ to the number of times of its order, it moves back to the front and becomes 1 again.

We can apply Lagrange's Theorem in this way. If we take any element $h \in G$ and generate a set, it will be a subgroup of $G$. For example, using $h = 2$ gave us $\{1, 2, 4\}$, which is a subgroup of $\mathbb{Z}_p^*$. According to Lagrange's Theorem, the order of this subgroup must divide the order of the group. We see that 3 divides 6.

It is actually easy to prove Fermat's Little Theorem (a special case of) using Lagrange's Theorem. If we let $h \in \mathbb{Z}_p^*$, then $h^{p-1} = h^{|\mathbb{Z}_p^*|} = [h^{order(h)}]^{\frac{|\mathbb{Z}_p^*|}{order(h)}} = 1^{\frac{|\mathbb{Z}_p^*|}{order(h)}} = 1$. This works out because we know from Lagrange's Theorem that $order(h)$ divides $|\mathbb{Z}_p^*|$, so the exponent is an integer.

Note that $\mathbb{Z}_p^*$ has order $p - 1$, which is even (if $p \neq 2$). To obtain a prime order cyclic group, typically we would use an element $h \in \mathbb{Z}_p^*$ whose order is a prime number $q$ (which results in a subgroup of $G$).

### 7.2.2 Modular eth Roots

Let $G$ be a finite cyclic group of known prime order ($|G| = q$). For example, we might have taken $\mathbb{Z}_p^*$, then taken an element $g$ in this set, and generated a subgroup $G \subset \mathbb{Z}_p^*$. This subgroup is of prime order $q$, We want to solve the following problem.

Given an element $h \in G$ and and exponent $e$, where $1 < e < q$, we want to find $y = h^{\frac{1}{e}}$, called the eth root. The algorithm is:

1. Compute $\alpha = e^{-1}$ in $\mathbb{Z}_q^*$
2. Output $y = h^\alpha$ in $G$

Why does this work? Well we know that $\alpha * e = 1$ in $\mathbb{Z}_q^*$, and since this is mod q, we can write this as $\alpha * e = 1 + k * q$. Thus, we have

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

We know that $h^q = 1$ since this equals $h^{order(h)} = 1$.

This methodology works when $q$ is known and $gcd(e, q) = 1$ (so that the inverse exists). Thus, we can compute the eth root (square root, cube root, etc.) using $O(logq)$ multiplications in $G$.

## 7.3 Diffie-Hellman in a Finite Cyclic Group

In Diffie-Hellman, we have two parties, call them Alice and Bob, that want to do a key exchange. The following is how the Diffie-Hellman protocol works in any finite cyclic group. We have a finite cyclic group is $\mathbb{Z}_p^*$, where $p$ is a prime. Therefore, the order of $\mathbb{Z}_p^*$ is $p - 1$. Let $G$ be a finite cyclic group, which is a subgroup of $\mathbb{Z}_p^*$ with order $q$ with generator $g$, where $q$ is a prime that divides $p - 1$. Then, $(G, g, q)$ are public parameters. How this works is as follows.

The first party, Alice, picks a random $\alpha$ from $\mathbb{Z}_q$. The second party, Bob, independently picks a random $\beta$ from $\mathbb{Z}_q$. Alice computes $A = g^\alpha modp$ and sends this over to Bob. Bob computes $B = g^\beta modp$ and sends this over to Alice. Alice can then compute $B^\alpha modp = (g^\beta)^\alpha modp = g^{\alpha\beta} modp$. Bob can then compute $A^\beta modp = (g^\alpha)^\beta modp = g^{\alpha\beta} modq$. Since both parties have $g^{\alpha\beta}$, they can use this as their shared key $k$.

Assuming an adversary can only eavesdrop and not tamper with messages, the eavesdropped can intercept and see $g, A = g^\alpha$, and $B = g^\beta$, but it should be difficult for the adversary to compute $g^{\alpha\beta}$ from just this information. Therefore, we say that $(G, g)$ satisfies the computational D-H assumption (CDH) if for all efficient adversaries $\mathcal{A}$, the $P[\mathcal{A}(g, g^\alpha, g^\beta) = g^{\alpha\beta}]$ is negligible. In other words, the adversary cannot construct the key $k$ even if they intercept the messages.

Note that if the adversary had the power to tamper with the message, then this would be insecure due to a Man in the Middle attack.

The best known algorithm for solving CDH runs in e to the cubic root of the log time. This is much faster than $2^n$ (random guessing), and thus the primes used for Diffie Hellman must be very large in order to secure. In elliptic curve crypto, we use the same methodology but the best known algorithm to solve that one is quite slow, so we don't need as large of a prime. Thus elliptic curve crypto over $\mathbb{Z}_p$ is the most commonly used methodology today.

### 7.3.1 Discrete Log Problem

A related problem to CDH is the discrete log problem for a group $G$. The problem is as follows. Given $h \in G$, we want to find $\alpha \in \mathbb{Z}_q$ such that $h = g^\alpha$.

The discrete log assumption holds for $(G, g)$ if for every efficient algorithm $\mathcal{A}$, the $P[\mathcal{A}(g, g^\alpha) = \alpha]$ is negligible. Both $\mathbb{Z}_p^*$ and elliptic curve have this property. It is a fact that if solving the discrete log if easy, then solving the CDH is easy. Thus, for both these methodologies, both are hard.

# 8 Public Key Encryption (PKE)

In the Diffie-Hellman key exchange example we had examined, both parties Alice and Bob selected a value, exponentiated that, and then sent it over. For practical use, this could work in an "interactive" setting such as TLS, where both Alice and Bob are online at the same time. When they want to send messages to each other, they could use key exchange to do so.

However, there are many situations that are "non-interactive", where Alice and Bob may not both be online at the same time. For example, Alice may send Bob a text message that Bob will only receive at a later time when he logs on. In this case, there is an intermediary, the "cloud" which receives Alice's message, waits until Bob is online, then sends over the message to Bob. How can we work in this "non-interactive" setting effectively? The answer is public key encryption (PKE).

In PKE, let's assume Alice wants to send Bob a message. Bob first generates a secret key $SK_{bob}$, and from this he calculated a public key $PK_{bob}$ and sends this over to Alice. When Alice wants to send a message to Bob, she encrypts using $E(PK_{bob}, m) = c$ and sends this over to Bob. Bob decrypts this as $D(SK_{bob}, c) = m$ to retrieve the message. Note that only Bob has access to his secret key and neither Alice, nor an adversary can get it. Also note that the public key is public, so the fact that $PK_{bob}$ might be intercepted by an eavesdropping adversary does not compromise the security (assuming a good cipher).

More specifically, a public key encryption scheme (PKE) over $(\mathcal{M}, \mathcal{C})$ is a triple of algorithms $(Gen, E, D)$ such that:

- $Gen() \to (PK, SK)$
- $E(PK, m) \to c$
- $D(SK, c) \to m$

such that $P(D(SK, E(PK, m)) = m) = 1$ (satisfies the correctness property). Note that the generator is a randomized algorithm so that the adversary cannot construct the SK from the PK. Note that the encryption algorithm E is also a randomized algorithm for security. We need this to fulfill the semantic security (security against eavesdropping) property to deem a PKE secure (against eavesdropping only).

Semantic security for a PKE is defined as follows. Assume we are either in experiment $b = 0$ or $b = 1$. The challenger generates a $(SK, PK)$ pair. The adversary sends two messages of the same length $m_0, m_1 \in \mathcal{M}$ to the challenger. The challenger, depending on which experiment we are in, sends back $c = E(PK, m_b)$. The adversary then needs to guess whether $b = 0$ or $b = 1$. We say $\mathcal{E} = (G, E, D)$ is semantically secure if for all efficient adversaries $\mathcal{A}$, that $Adv_{SS}[\mathcal{A}, \mathcal{E}] = |P[EXP(0) = 1] - P[EXP(1) = 1]|$ is negligible.

Note that in order to fulfill this definition of semantic security, the algorithm $E$ must be randomized. Otherwise the adversary could calculate $E(PK, m_0)$ and $E(PK, m_1)$ on his own and then see which one matches the ciphertext the challenger sends back to tell which experiment we are in.

The notion of security to defend against active attacks is called CCA security. CCA security is defined with the following game. We are either in $b = 0$ or $b = 1$. The challenger generates a SK, PK pair and sends the PK over to the adversary. The adversary sends two messages $m_0, m_1$ to the challenger and receives back the ciphertext $c = E(PK, m_b)$ for one of the messages. So far, this is the same as semantic security, but now we allow the adversary even more power. The adversary can send ciphertexts $c_i \neq c$ to the challenger and receive the decryption $D(SK, c_i)$ back from the challenger. The adversary can send as many $c_i$ to the challenger as it wants and receive the corresponding decryptions. The adversary then guesses if we are in $b = 0$ or $b = 1$. If the advantage is non-negligible, then the system is non CCA secure. If the advantage is negligible, then the system is CCA secure.

## 8.1 Applications of PKE

The first application of PKE is a toy example of key exchange. Alice can first generate $(PK, SK)$ pair and send $PK$ over to Bob. Bob then randomly chooses a key $k$, encrypts it using $E(PK, k) = c$ and then send it back to Alice. Alice can decrypt this using $D(SK, c) = k$. Now both Alice and Bob have the key $k$. If the system is semantically secure, then even if an adversary intercepts $PK$ and/or $E(PK, k) = c$, they would not be able to distinguish this from $E(PK, 0)$, thus there is no attack. Therefore, the adversay would learn nothing about $k$.

Note that this is not secure against Man in the Middle attacks, only eavesdropping attacks.

Another application of PKE is for file sharing in an encrypted file system. Let $(E_S, D_S)$ be a symmetric encryption system over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$. Let $(G, E, D)$ be a PKE where the message space is $\mathcal{K}$. Let's say Alice has a secret key $k_a$ for the symmetric system. Alice then uploads an encrypted file $F_1$ to the cloud. She does this by choosing a random file encryption key $k_1$. Then she computes $E_S(k_a, k_1)$. She then computes $E_S(k_1, F_1)$. She uploads these to the cloud, thus the cloud is storing $E_S(k_a, k_1)$ and $E_S(k_1, F_1)$. Whenever Alice wants to open/modify the file, she can do so because she has the keys.

Now if she wants to give read/write access to Bob for a file, she can do the following. She obtains the $PK_{bob}$ from Bob which is sent over the cloud. She then encrypts $E(PK_{bob}, k_1)$ and uploads that to the cloud. Now Bob can decrypt this using $SK_{bob}$ and obtain $k_1$, which he can then use to decrypt the file $F_1$ to read/write. Note that this requires no interaction with Bob!

## 8.2 ElGamal Encryption

ElGamal Encryption is PKE created from Diffie-Hellman. How it works is as follows. We have a finite cyclic group $G$, such as $\mathbb{Z}_p^*$ of order $p-1$ (since $p$ is prime). We have a subgroup $G$ of order $q$ generated by a generator $g$. Note that $q$ divides $p-1$. Assume there are two parties Alice and Bob. Alice chooses a random $\alpha$, which is the secret key, and computes $A = g^\alpha$, which is the public key. She sends over $A$ to Bob. Note that since this is PKE, it works in a "non-interactive" setting, and Bob need not be online at the time. When Bob gets online, he selects a random $\beta$ as a secret key and constructs $B = g^\beta$. He also computes $A^\beta$, which allows him to derive a symmetric key $k$ using a hash function $H$. He uses $k$ to encrypt a message $m$ and then sends a tuple of $B$ and the encrypted message over to Alice. When Alice wants to decrypt, she can compute $B^\alpha$, and then use this along with $H$ to obtain the key $k$. Using the key $k$, she can decrypt and read the message. Typically in ElGamal encryption, a new $\alpha$ and $\beta$ is chosen for each new message sent.

Let's reiterate the ElGamal encryption in more detail. We have:

- A finite cyclic group $\mathbb{Z}_p^*$ of order $p$ and a subgroup of order $q$ with generator $g \in G$
- $(E_S, D_S)$, which is a symmetric AE defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$
- $H : G^2 \to \mathcal{K}$, a hash function that takes as input two elements from the group $G$ and returns an output in the key space

Alice constructs a PKE $(G, E, D)$ by choosing a random generator $g \in G$. She chooses a random $\alpha \in \mathbb{Z}_q$ as the secret key and computes $A = g^\alpha (modp)$ as the public key, which is sent over to Bob. When Bob wants to send a message to Alice, he chooses a random $\beta \in \mathbb{Z}_q$, and computes $g^\beta (modp) = u$. He then takes $A$ which Alice had sent over and computes $A^\beta (modp) = v$. Note that $u$ is Bob's public key and $\beta$ is his secret key. Then he uses the hash function $H(u, v)$, which outputs a key $k$. He uses this key $k$ to encrypt his message by $E_S(k, m) = c$. He sends over $(u, c)$ to Alice.

When Alice wants to decrypt, she calculates $v = u^a (modp)$ (since $(g^\alpha)^\beta (modp) = (g^\beta)^\alpha (modp)$). Using $H$, she runs $H(u, v)$ to obtain the key $k$. She then decrypts the ciphertext by $D_S(k, c) = m$ and now has access to the message.

The performance of ElGamal is that the encryption step takes two exponentiations, but the decryption step only takes one exponentiation.

In practice, the ElGamal system is implemented as ECIES (Elliptic Curve Integrated Encryption System).

When is ElGamal secure? There is a theorem that states that this is semantically secure when

- CDH holds in $(G, g)$
- $(E_S, D_S)$ is semantically secure
- $H$ is a secure key derivation (it maintains the entropy of the inputs)

When is ElGamal CCA-secure (secure against active attackers)? There is a theorem that states that this is CCA-secure when

- Interactive Diffie-Hellman assumption holds in $(G, g)$
- $(E_S, D_S)$ provides authenticated encryption
- $H$ is a "random oracle"

## 8.3 RSA

RSA (Rivest, Shamir, Adelman) is an alternative public key encryption to ElGamal encryption. It uses a different construction than ElGamal, namely the trapdoor function, to do the encryption/decryption.

A trapdoor function $\mathcal{X} \to \mathcal{Y}$ is a triple of efficient algorithms $(G, F, F^{-1})$ where

- $G()$: randomized algorithm that outputs a key pair $(PK, SK)$
- $F(PK, \cdot)$: deterministic algorithm that defines a function $\mathcal{X} \to \mathcal{Y}$
- $F^{-1}(SK, \cdot)$: defines a function $\mathcal{Y} \to \mathcal{X}$ that inverts $F(PK, \cdot)$

This triple satisfies the correctness property. For all $(PK, SK)$ pairs outputted by $G$, for any $x \in \mathcal{X}$, $F^{-1}(SK, F(PK, x)) = x$. When is a trapdoor function secure? It is secure when $F(PK, \cdot)$ is a "one-way" function. Anyone can compute $F(PK, \cdot)$ in the forward direction, since the $PK$ is public, but only the person with the $SK$ can invert this function. More formally, a trapdoor function is secure if any efficient adversary $\mathcal{A}$ is given the $PK$ as well as $y = F(PK, x)$ for all $x \in \mathcal{X}$, and the adversary cannot find $x$.

To use trapdoor functions as a PKE, we do the following. We have a secure trapdoor function, an symmetric authenticated encryption $(E_S, D_S)$ defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ and a hash function $H$. To encrypt, we select an $x$ at random from $\mathcal{X}$ and use the trapdoor function to produce $y = F(PK, x)$. We create a symmetric key $k = H(x)$. We encrypt our message $c = E_S(k, m)$ and send over $(y, c)$ to the other party. To decrypt, the other party first obtains $x$ by $x = F^{-1}(SK, y)$. Then then obtain the key by $k = H(x)$. They can then decrypt the message $m = D_S(k, c)$. The security theorem states that if the trapdoor function is secure and $(E_S, D_S)$ provides authenticated encryption, and $H$ is a random oracle, then this PKE is CCA secure.

An incorrect use of a trapdoor function is to directly apply $F$ to the plaintext. If we just output $c = F(PK, m)$, this is insecure! $F$ is a deterministic function. For example, if two of the same message is sent, both the ciphertexts are exactly the same and the adversary learns that information.

### 8.3.1 RSA Trapdoor Permutation

How do we build a Trapdoor function? RSA, in fact, uses a trapdoor permutation (it is one-to-one where the domain equals the range).

A quick review of arithmetic mod composites that we'll need. Let $N = p * q$ where $p, q$ are prime. Then let us denote $\mathbb{Z}_N = \{0, 1, 2, ..., N-1\}$. We know that $x \in \mathbb{Z}_N$ is invertible iff $gcd(x, N) = 1$. Therefore, the number of elements (Euler's totient function) in $(\mathbb{Z}_N)^*$ is $\varphi(N) = (p-1)(q-1) = N - p - q + 1$. By Euler's theorem, for all $x \in (\mathbb{Z}_N)^*$, $x^{\varphi(N)} = 1 mod(N)$.

The RSA trapdoor permutation works like this. In the generation algorithm $G$, we choose random primes $p, q$ that are approximately 1024 bits and set $N = p * q$. We then compute $\varphi(N) = N - p - q + 1$. Note that $p, q, \varphi(N)$ are secret. $\varphi(N)$ cannot be calculated efficiently without knowledge of $p, q$. We then choose integers $e, d$ such that $e * d = 1 mod(\varphi(N))$. In other words, $e$ and $d$ are inverses of each other. Our $PK = (N, e)$ and our $SK = (N, d)$. The function for encrypting is $F(PK, x) = x^e mod(N)$. To invert this with the secret key, we have $F^{-1}(SK, y) = y^d (modN)$. This works because $y^d = (x^e)^d = x^{ed} = x^{k\varphi(N)=1} = (x^{\varphi(N)})^k * x = 1 * x = x$.

Is the RSA really a one way function? We need to be assured of this to be assured that we have security when using RSA. It turns out that this is an outstanding problem in cryptography, but for many years, no one has been able to efficiently invert RSA without knowledge of the secret key. This boils down to the following. It is a hard problem to factor $N$ into primes $p$ and $q$. Without $p$ and $q$, it is hard to calculate $\varphi(N)$. Therefore, given the public key $e$, an adversary cannot find the inverse $d$ since they do not know $\varphi(N)$. Note that if $\varphi(N)$ ever became exposed, an adversary could find $d$ using the extended Euclidean algorithm to find inverses. Therefore, $p, q, \varphi(N), d$ need to be kept secret.

Using the same format as earlier to create a PKE from a trapdoor function, RSA public key encryption is as follows. Given a symmetric encryption scheme with AE, $(E_S, D_S)$ and a hash function $H$, we have

- $G()$: generate the RSA parameters: $PK = (N, e)$, $SK = (N, d)$
- $E(PK, m)$
  - Hash message $H(m)$ in $\mathbb{Z}_N^*$
  - Calculate $H(m)^e (modN) = c$ and output

- $D(SK, c)$
  - Calculate $[H(m)^e (modN)]^d (modN) = H(m)$

This encryption scheme is secure. Note again that applying RSA trapdoor directly to the message is insecure. This can be broken using a Meet in the Middle attack.

### 8.3.2 RSA in Practice

The RSA encryption scheme described above is the ISO standard, unfortunately, it is not always used in practice because RSA was implemented in many systems before the ISO standard was formalized. Typically in practice, the message key first does a preprocessing step, and the key is expanded, and then RSA encrypts this expansion into ciphertext. However, depending on how the key is "expanded," there are attacks that can be done. One famous attack on an old RSA method is the Bleichenbacher attack.

Since in RSA, the $d$ and $e$ could be quite large, computationally RSA can be slow. As it turns out, we cannot choose a $d$ that is too small because it is insecure. However, it turns out to be ok to use a small $e$. The minimum value of $e$ that can be used is 3. This speeds up RSA.

Note that RSA is asymmetric in the fact that is it fast to encrypt and slow to decrypt since $e$ can be small and $d$ must be large. This is contrasted with ElGamal where the time to encrypt and decrypt is approximately the same.

## 9 Digital Signatures

In the digital world, instead of in the physical world, a signature such as handwriting is easy to copy and useless. Instead, a digital signature is a function of the document being signed. A signature scheme is a triple of algorithms $(G, S, V)$, where $G$ generates a $(PK, SK)$ pair, $S(SK, m) = \sigma$ and $V(PK, m, \sigma)$ returns accept or reject. A signature scheme must follow the correctness property, that for any message $m \in \mathcal{M}$, $V(PK, m, S(SK, m))$ returns accept. Note that the idea is that there is one signer that knows the secret key, but that anyone (many people) can verify with the public key. In general, digital signatures are useful in the applications where there need to be many verifiers and one signer. If there only needs to be one verifier, it would be more advisable to use a MAC, since it is faster to compute.

A secure signature scheme is defined as follows. An attacker has the power for a chosen message attack (CMA). An attacker can send a challenger $m_1, ..., m_q \in \mathcal{M}$ and the challenger returns $\sigma_i = S(SK, m_i)$ to the attacker. The attacker's goal is to produce an existential forgery. He wants to produce some new valid pair $(m, \sigma)$ where $m \neq m_1, ..., m_q$. If all efficient attackers $\mathcal{A}$ cannot produce an existential forgery (probability of winning is negligible) then the signature algorithm is secure.

Some real world applications of digital signatures:

- Android devices ship to customers with a built in PK. Every software update is signed by Google and whenever there is a software update, the Android devices use the PK to verify the signature on the update. Google is the only one that has access to the SK.
- In credit card payments, the PoS terminal will send transaction details and a nonce to the chip card (which has a SK). The chip card will return the signature and the card certification to the PoS terminal, which then forwards this to the bank, payment processor, etc.

Three general approaches to data integrity are CRH, digital signature, and MAC. CRH requires a read-only public space for hashes, and allows for many verifiers. Digital signatures similarly allow one signer and many verifiers, but do not require the read-only public space. Instead, the requirement is that the the verifiers need the correct PK. Lastly, a MAC is suitable when there is one signer and one verifier.

In terms of a length of the message input into the signature scheme, we don't have to worry about that. The reason is that if we have a secure signature scheme and a CRH $H$, we can hash a message of any length down into the smaller message space that the signature scheme requires, and then run the signature scheme. There is a theorem that states that if the signature scheme is secure and $H$ is a CRH, then the entire signature scheme is secure for any length message if we hash it first then put it into the signature scheme. Thus, we can focus on building secure signature schemes for a certain size message space, and know that this will generalize.

What primitive should be use for a signature scheme? It is possible to use a general one-way function. However, it has been shown that the signature generated from this is quite long (4-40KB). The benefit is that it is quantum resistant. Therefore, this method is suitable for software updates, but not as much for constant signing.

We can also use a discrete log method, such as in DH. Common uses of this are ECDSA, Schnorr, and BLS. The signature size is 48 or 64 bytes, which is quite short! Lastly, we can use a trapdoor permutation such as in RSA to do this.

Assume we have a trapdoor permutation $(G, F, F^{-1})$ and a hash function $H$. Then our signature scheme is

- Signing: $\sigma = F^{-1}(SK, H(m))$
- Verifying: $F(PK, \sigma) = H(m)$

If $(G, F, F^{-1})$ is a secure trapdoor permutation and $H$ is a random oracle, then this is a secure signature scheme. Note that we use $F^{-1}$ to sign since only the signer can invert $F$ at $H(m)$, whereas we use $F$ to verify since anyone with the PK can verify.

More concretely, this is how RSA-FDH (full domain hash) works. For our generating algorithm $G$, we will choose $N = p * q$ where $p, q$ are primes, calculate $\varphi(N) = N - p - q + 1$ and choose $e, d$ such that $e * d = 1(mod \varphi(N))$. Our CRH is $H$, which takes a message and hashes it into an element in the group $(\mathcal{M} \rightarrow \mathbb{Z}_N^*)$. The public key is $PK = (N, e, H)$ and the secret key is $SK = (N, d, H)$. To sign a message, we have $\sigma = [H(m)^d mod N]$. To verify a message, we return accept iff $\sigma^e mod N = H(m)$.

One small note is that the range of $H$ is dependent on the $N$ that is chosen, since $H$ needs to output something in $\mathbb{Z}_N^*$. However, this can be solved by just hashing into a larger domain, and then reduce that result by running $mod N$ on that output.

Since $e$ is small and $d$ is large for RSA, verification is extremely fast, even though generating the signature may be slow.

Why do we need the hash function $H$ and why do we not just apply $F^{-1}$ to the message $m$ directly? It turns out that there is an attack if we do this called the blinding attack on RSA. How it works is as follows. Assume the adversary $\mathcal{A}$ wants a signature on $m \in \mathcal{M}$. Then, the adversary does the following:

1. Choose $r \overset{R}{\leftarrow} \mathbb{Z}_N^*$ and compute $\hat{m} = r^e * m (mod N)$
2. Request the signature on $\hat{m} \in \mathbb{Z}_N^*$ and gets back $\hat{\sigma}$ from the challenger. Note that $\hat{\sigma} = \hat{m}^d$ since we are signing the message directly. Note that $\hat{\sigma}^e = [\hat{m}^d]^e = \hat{m}^{de} = \hat{m}$.
3. Set $\sigma = \frac{\hat{\sigma}}{r} (mod N)$. $\sigma$ is a valid signature on $m$.

We can easily see that $\sigma$ is a valid signature for $m$ since $\sigma^e (mod N) = (\frac{\hat{\sigma}}{r})^e = \frac{\hat{m}}{r^e} = \frac{m * r^e}{r^e} = m$. What this means is that an adversary can ask for a signature on a random $\hat{m}$ and get back a signature on $m$. Although this is an attack, it turns out be a very useful property of RSA for problems relating to secrecy such as anonymized voting or anonymized digital cash. For example, a bank can sign a check for some amount of money without knowing who it is to.

## 10 Certificates

In this section, we discuss public key management (public key infrastructure (PKI)). When Bob wants to send a message to Alice, or verify Alice's signature, he needs Alice's public key. We want to make sure that Bob obtains the correct public key. If Bob instead is fooled into using an adversary's public key, thinking it is Alice's public key, then the adversary can perform a Man in the Middle attack and/or signature forgery.

To solve this problem, we involve the use of a third party, the certificate authority (CA). How this works is as follows. We assume that all parties have the correct public key of the CA, $PK_{CA}$. Alice generates a $(PK_A, SK_A)$ pair and sends her $PK_A$ over to the CA. The CA signs this message containing the $PK_A$ with their own secret key $SK_{CA}$, and sends the certificate back to Alice. Now, when Alice wants to talk to Bob, she sends over this certificate to Bob. Bob can verify that this certificate is valid because he also has $PK_{CA}$. Therefore, he now trusts that he has the correct $PK_A$ and now he can send a message to Alice $E(PK_A, m) = c$ knowing that this is truly Alice and not an adversary.

Some notes about this process:

- Everyone needs to know the authentic $PK_{CA}$. This is possible because these are usually distributed with the installation of the OS or the browser.
- The CA is mostly offline: Alice only needs to talk to the CA as part of the key generation process. After that, she can communicate freely with the other party without consulting the CA
- Unlike the problems shown previously with using a TTP, in this case, the CA does not know Alice's SK and therefore cannot read any of her messages or breach her privacy in any way.

In the real world, there is not just one CA, but rather many of them (approx 1200). There is a certification hierarchy. Around 60 of these are deemed root CAs, and these are the CAs whose public keys are shipped with everyone's OS and/or browser. The other CAs are called intermediate CAs, and they can offer certificates as well, but they also need to have their public key signed by the root CA. Therefore, there is a CA tree, and one can obtain a certificate from any CA in this tree. However, if we obtain a certificate from an intermediate CA, the verification process is slightly different. For example, let's say that $CA_1$ is an intermediate CA that is certified by the root CA. Let's also say that $CA_2$ is an intermediate CA certified by $CA_1$. Let's say Amazon obtained a certificate from $CA_2$. Amazon's certificate would look like this: $[amazon.com|PK_A|CA_2sig]; [CA_2ID|PK_{CA_2}|CA_1sig]; [CA_1ID|PK_{CA_1}|rootCAsig]$. When someone is trying to connect to Amazon, they verify Amazon's $PK_A$ as follows. Since everyone only has the $PK_{rootCA}$, they would first verify that $rootCAsig$ is valid and then know that $PK_{CA_1}$ is valid. Then then use $PK_{CA_1}$ to verify $CA_1sig$ and then know that $PK_{CA_2}$ is valid. They then use $PK_{CA_2}$ to verify $CA_2sig$ and then know that $PK_A$ is valid. This is called a certification chain.

### 10.1 Complexities of Certificates

The first difficulty with using certificates in the real world is with certification revocation. Often, someone's private key gets stolen when they get hacked, we want to then invalidate the certificate associated with this private key. However, the attacker has a valid certificate with this private key. How do we revoke this certificate? A few solutions are below:

- Expiration Date: Each certificate comes with an expiration date, after which it is no longer valid. Typically these are 3 months or 1 year (but there is progress towards making these shorter). Therefore, the attacker will only have the valid certificate for a certain period of time before it expires.
- CRLSet (Certificate Revocation List Set): This is a list of revoked certificates, which is signed by the CA. The browsers then download this CRLSet every so often (maybe once a day) and whenever receiving a certificate, they first check to see if a certificate is in the list, then reject it if it is. The downside of this method is that everyone needs to be in sync with the latest CRLSet.
- OCSP (Online Certificate Status Protocol): Every time a browser receives a certificate, it sends a message to an online OCSP responder server to check if a certificate is valid. The downside of this method is that it severely increase the traffic over the Internet and it is unclear who would run these OCSP responders. Also, the OCSP responders would be able to know exactly which sites everyone is visiting, which is a privacy issue.
- Short Lived Certifications: Change the expiration date to 4 days.

The solutions above describe how to revoke certificates when someone's private key has been compromised. What about when a CA is compromised? Then it could issue "fake" certificates to domains. There are currently a few solutions:

- Certificate Pinning: Certain domains are "pinned" to some allowable CAs. In other words, some certificates are only valid when signed by a certain CA.

- Certificate Transparency: Require the CAs to publish logs of issued certificates. These logs are called CT logs. The benefit of having these logs is that anyone can check these logs to see if their own certificates are valid or if there is an adversary who has created a fake certificate on behalf of a domain.

# 11 Identification Protocols

Now that we have reviewed cryptographic primitives such as symmetric encryption, signatures, certificates, etc. we can examine how these all come together in the real world to build protocols. The first type of protocols we want to examine are called identification protocols. These are used in the context of two parties, a prover and a verifier. The prover wants to prove to the verifier that they are who they say they are. The prover will have a secret key $SK$ and a verifier will have a verifier key $VK$, and through an exchange, the verifier wants to be able to determine if the prover really is who they say they are. For example, this could be on a website, where a user needs to login with a username and password, or a bank ATM where a user needs to authenticate in order to withdraw money, etc.

To examine the security of these protocols, there are three types of attackers with increasing power that we will examine:

- Direct Attacker: impersonates prover with no additional information (other than $VK$). Example: opening a door lock
- Eavesdropping Attacker: impersonates prover after eavesdropping on a few conversations between prover and verifier. Example: wireless car entry system
- Active attacker: interrogates prover and then attempts to impersonate prover. Example: fake ATM in shopping mall

## 11.1 Protocols Against Direct Attacks

The most basic and weakest protocols are the ones that only protect against direct attacks (since these are the weakest attackers). We have a prover and a verifier that perform an exchange, and the only power a direct attacker can do is to hack into the verifier to obtain the $VK$.

In the basic password protocol, the prover (user) chooses a password $SK$ and the verifier (server) stores a copy of that password in plaintext. So $SK = VK =$ password. When the user wants to authenticate, they send over the $SK$ and the server matches this against the $VK$ and if it matches, they know that this is the user. Clearly this protocol is insecure against direct attacks since if an attacker hacks into the server, they obtain (in plaintext) $VK = SK$ and they can thereafter impersonate the user and login whenever they want. Therefore, clearly, servers (verifiers) should never store passwords in plaintext.

The next logical thing to do is that given a hash function $H$, the server stores $VK = H(SK)$. Whenever the user wants to authenticate, they send over $SK$, the server runs $H(SK)$, and if $H(SK) = VK$ then they know this is the user. It appears that if an attacker breaks in to the server, they can only recover $H(SK)$ and cannot get $SK$ from that. However, this method is insecure mainly because users select poor passwords. The most common passwords include: "123456", "password", "qwerty", "iloveyou", etc. Statistics have shown that a dictionary of 360,000,000 words covers about 25% of user passwords. This dictionary of 360,000,000 words is well known among the world.

Therefore, what the attacker can do is hash each of the words in this dictionary once to obtain the hashed outputs of these words. Then they hack into the server to obtain all the $H(SK)$, and then match it to the list of hashed outputs that they have. They will then be able to obtain on average 25% of the passwords! This is what happened to LinkedIn in 2012, where the attackers ended up recovering 90% of the passwords. Note that the attacker only need to hash the words in the dictionary 1 time, and then they can just scan the hacked $H(SK)$ and try to match them. This is too fast. We want to slow the attacker down, and force them to at least have to do the hash computation per user/pw combination.

Therefore, to prevent this kind of attack, we can use salts. How this works is that when a user $A$ sets a password, the server generates a random $n$-bit salt associated with that user, denote it $S_A$. The server then saves $S_A$ and $H(pw_A, S_A)$. When user $B$ sets a password, the server saves $S_B$ and $H(pw_B, S_B)$. Therefore, even if user $A$ and user $B$ have the same password ($pw_A = pw_B$), if an attacker compute $H(pw_A, S_A)$, then cannot automatically obtain $H(pw_B, S_B)$. Instead, they need to recompute the hash using $S_B$ to obtain this. What this ultimately means is that if an attacker has the 360,000,000 word dictionary, and they hack into the server and obtain all the $S_i$ and $H(pw_i, S_i)$, they need to hash all words in the dictionary per user (instead of just once) in order to do the dictionary attack (since each salt is different), which will take much more time. Given that we have $F$ users, what we have done is convert the attack on the entire server's passwords from $O(|Dict| + |F|)$ to $O(|Dict| * |F|)$. The typical size of the salt is 64 or 128 bits.

To slow the attacker down, we can also use slower hash functions. A hash function such as SHA256 is quite fast. We also know that we can create hash functions by repeatedly iterating a hash function. For example,

$H(pw) = SHA256(SHA256(...SHA256(pw, S_A)...))$.

The goal is to perform enough iterations that to log in once (computing this composition of hash functions once) is fast enough that it is unnoticeable to the user, but slows the attacker down enough that if they were to do this in a large batch, it would take much longer. Hash functions such as PBKDF2 and bcrypt use this method. The problem however, is that custom hardware (ASICs) can evaluate hash functions up to 50,000X faster than a commodity CPU. This is because these ASICs are built specifically to only compute hash functions (not have cache, remove everything else on the processor besides the ability to compute these hashes).

Therefore, hash functions such as scrypt and Argon2i take an approach where they require lots of memory in order to evaluate the hash function. Thus, the ASICs (that don't have caches) cannot compute these hash functions and the attacker must have caches/memory store in order to compute these hashes. Therefore, the attacker can only go at the speed of a commodity CPU to evaluate these hash functions.

## 11.2 Protocols Against Eavesdropping Attacks

The protocols against eavesdropping attacks use one-time password systems. This is the 2-factor authentication that is commonly used today in practice. The idea is that along with the usual password system, the prover and verifier agree upon another password as well that is only good for one login, and after that it changes.

More formally, the setup for the one-time password is as follows. A random secret key $k$ is generated and the prover has $SK = (k, 0)$ and the verifier also has $VK = (k, 0)$. Given a PRF $F$, the first time the user wants to login, they send $r_0 = F(k, 0)$ and the server computes $F(k, 0)$ on their end and checks if it matches. If it matches, the user has successfully authenticated. Both the user and server increase their state to 1. Next time the user wants to authenticate, then send over $r_1 = F(k, 1)$. Therefore, each time the user authenticates, they send over something different and this protects against eavesdroppers. Note that instead of $0, 1, 2, ...etc$, any value that is agreed upon by the user and the server can be used here. One common value that is used here is time. These methods are called TOTP (timed one-time passwords).

One downside to this methodology is that the server stores $VK = SK$, so it they get hacked, this is exposed. Is there a way to store $VK$ so that it is a public key and exposing it does no harm? There is a way called S/Key which does this. Let $H^{(n)}(x) = H(H(...H(x)...))$ where the hash function is applied $n$ times. When generating the $SK, VK$ pair, we have $SK = (k, n)$ and $VK = H^{(n+1)}(k)$.

Whenever the prover wants to authenticate for the $i$th time, their current $SK = (k, i)$. They send over $r = H^{(i+1)}(k)$ and update $SK = (k, i-1)$. The verifier has $H^{(i+1)}(k)$ and verifies by running $H(r)$ and seeing if it matches $H^{(i+1)}(k)$. If it does then the prover has successfully authenticated and the verifier then sets $VK = r$.

This S/Key protocol is secure against eavesdropping and has a public $VK$, so it is secure against hacking. However, the drawbacks are that it requires $t$ to be quite long (e.g. 80 bits). This makes it difficult for a human to type in 80 bits every time they need to do a 2 factor authentication, so more common is the method where the server does store the $VK = SK$.

## 11.3 Protocols Against Active Attacks

All protocols discussed thus far are vulnerable against active attacks. To protect against these attacks, the verifier needs to challenge the prover and view their response. How this works is as follows. It is called a signature-based challenge response.

A $SK, VK$ key pair is generated using an algorithm such as RSA. The user stores $SK$ and the server stores $VK$. Note that $VK$ is allowed to public in this scenario (without compromising security). When the user wants to authenticate, the server sends over a random $m$. The user computes $t = Sign(k, m)$ and sends this back to the server. The server than verifies this and if the verifying algorithm returns "accept", then the user authenticates. Note that this uses the primitive of signatures, where only the person with the $SK$ can sign the message, but anyone with the public key $VK$ can verify. Since the challenge $m$ is generated at random each time the user logs in, an attacker cannot log in at a later time even if they intercepted previous $\sigma$. If the server gets hacked, it only stores the $VK$, which is the public key, so no security is compromised. This protocol, however, requires $t$ to be long and thus is hard for a human to type in each time they want to authenticate.

The most common use of this signature-based challenge response is with U2F. A user has a U2F token (such as YubiKey) that they insert into the computer. Even if the browser or the computer has malware and can intercept $t$, they cannot log in on their own at a later time, because the $m$ is random and will change. The user doesn't need to manually type in this long $t$, since the U2F token usually has a method of signing such as fingerprint touch, so when a user wants to sign in, they just touch the key and the U2F token will create the signature and send it over to the verifier.

# 12 Authenticated Key Exchange

Previously, we had looked at how to do key exchange using PKE (public key encryption), but the methodology that was discussed was only good for protecting against eavesdropping adversaries. We also want to protect against active adversaries that may modify, inject, and delete packets such as Man In the Middle attacks. Therefore, we need to come up with better protocols that allow for secure key exchange against not just eavesdropping adversaries, but also active adversaries. This type of secure key exchange is called Authenticated Key Exchange (AKE).

All AKE protocols required a TTP (trusted third party) to certify the user identities. TTPs can be offline (such as CAs) or online (such as Kerberos). Here we discuss offline TTPs using the framework of the CAs that had been previously discussed.

Assume the two parties wanting to do an AKE are Alice and the Bank. If the AKE is successfully completed, Alice will obtain $(k, Bank)$, where $k$ is the symmetric key. For an AKE to be secure, we want the following to hold:

- Authenticity: If Alice's key $k$ is shared with anyone, it is only shared with Bank. The same needs to hold for the Bank as well.
- Secrecy: To the adversary, Alice's key $k$ is indistinguishable from random (even if adversary sees keys from other instances of Alice or Bank). The same needs to hold for the Bank as well.
- Consistency: If Bank completes the AKE then it obtains $(k, Alice)$.

If the above holds, we say that the AKE has static security. This is the weakest form of AKE security. If we have static security, and also the property that if the adversary learns $SK_{bank}$ at time $T$, and all the sessions with the bank prior to time $T$ still remain secret, then we have forward secrecy. The strongest level of AKE security is called HSM security. In HSM security, we have forward secrecy, but also the property that if an adversary queries an HSM holding $SK_{bank}$ $n$ times, then at most $n$ sessions are compromised.

AKEs can be one-sided or two sided. On the internet, more common are one-sided AKE. This happens when only one side has a certificate. Most users don't have a certificate, but servers do, and so when a user goes to a website, this is often a one-sided AKE. Usually, then this one-sided AKE is then followed by an identification protocol (such as a password), so the user then has to prove who they are. As a note, we should not design an AKE protocol ourselves but rather just use the latest version of TLS.

Assume again we have two parties Alice and the Bank. In a one-sided AKE, the bank has $SK_{bank}$ and $cert_{bank}$ and both Alice and the Bank have the $PK_{CA}$. In a two-sided AKE, Alice also has $SK_{alice}$ and $cert_{alice}$.

## 12.1 AKEs with Static Security

The simplest AKE protects only has static security. Let's examine the one-sided AKE protocol first. Assuming our two parties are Alice and the Bank, here are the steps. We also have algorithms $E$, which is chosen-ciphertext secure, and $S$, which is a secure signing algorithm.

1. Bank chooses a nonce $r$ at random, and sends over $r$ and $cert_{bank}$ to Alice
2. Alice verifies $cert_{bank}$ is valid using $PK_{CA}$.
3. Alice generates a shared key $k$.
4. Alice encrypts $E(PK_{bank}, (k, r)) = c$ and sends it over to the Bank.
5. The Bank decrypts $c$ using $SK_{bank}$ to obtain $(k, r)$. It checks to see if $r$ matches the one it sent over and if it does, then it knows that $k$ is valid.

This protocol is a statically secure one-sided AKE. Note that Alice must send $r$ inside of the encryption. Alice cannot send $c = (E(PK_{bank}, k), r)$ to the Bank that is not secure against a replay attack. The attack is as follows. The attacker waits for Alice and the Bank to perform the key exchange protocol and share the key $k$. Now, since both Alice and the Bank have key $k$, then let's say later on Alice sends $c_1$ to the Bank using symmetric encryption under key $k$ asking the Bank to "Pay Bob \$30." The adversary records $c$ and $c_1$ and also knows $r$. The adversary then runs its own key exchange with the bank. The bank sends over $r'$ to the adversary, and the adversary responds with $c, r'$. Note that this $c$ is the $c$ from Alice, but under a different $r'$ since the $r'$ is not encrypted. The Bank will accept this $c$ and assume the adversary is using key $k$. The adversary then can replay $c_1$ by sending it to the bank as many times as it wants, and each time the bank will "Pay Bob \$30." Hence, to be secure, we require that the protocol be to send $E(PK_{bank}, (k, r)) = c$, NOT $(E(PK_{bank}, k), r) = c$.

In a two-sided AKE, the protocol is as follows:

1. Bank chooses a nonce $r$ at random, and sends over $r$ and $cert_{bank}$ to Alice
2. Alice verifies $cert_{bank}$ is valid using $PK_{CA}$.
3. Alice generates a shared key $k$.
4. Alice encrypts $E(PK_{bank}, (k, "Alice")) = c$ and sends it over to the Bank. Alice also signs $\sigma = S(SK_{alice}, (r, c, "bank"))$ and sends over $\sigma$ and $cert_{alice}$ to the Bank
5. The Bank uses $PK_{CA}$ to verify that $cert_{alice}$ is valid. Then checks $\sigma$ using $PK_{alice}$ from $cert_{alice}$ to check for the correct id "*bank*" and the correct nonce $r$. The Bank decrypts $c$ using $PK_{bank}$ and checks to see if "*Alice*" matches the certificate name and if it does, then it knows that $k$ is valid.

Note that any minor changes to this protocol will results in insecurity. For example, if Alice sends over $E(PK_{bank}, (k, r)) = c$ instead of $E(PK_{bank}, (k, "Alice")) = c$, then this is susceptible to a misbinding attack. What an adversary can do is intercept/block the $c$ and $\sigma$ going from Alice to the Bank, and they themselves create $\sigma'$ and send that over along with Alice's $c$ to the Bank. The result is that Alice thinks she is talking to the Bank, but the Bank thinks it is talking to the adversary. This is called a misbinding attack.

## 12.2 AKEs with Forward Secrecy

So far, the AKE protocols that have been examined have static security, but we want to strengthen them in order to also have forward secrecy. This is because say Alice and the Bank performed the AKE one year ago, and a year later the adversary obtains $SK_{bank}$. If the adversary during the previous year had recorded all traffic between Alice and the Bank, they could then use the $SK_{bank}$ to obtain the $k$ from $c$ (from the original AKE) and then decrypt all communications (which are symmetrically encrypted) between Alice and the Bank using the $k$ it just derived. Clearly there is no forward secrecy.

We now visit a one-sided AKE with forward secrecy. The protocol is as follows:

1. In addition to $SK_{bank}$ and $cert_{bank}$ that the Bank has permanently, the Bank generates a new $(PK, SK)$ pair the first time Alice wants to connect.
2. The Bank sends $PK$, $cert_{bank}$, and $\sigma = S(SK_{bank}, PK)$ over to Alice.
3. Alice validates $cert_{bank}$ with $PK_{CA}$ and then uses $PK_{bank}$ to verify $\sigma$, from which she obtains $PK$.
4. Alice generates a shared key $k$.
5. Alice encrypts $c = E(PK, k)$ and sends it over to the Bank.
6. The Bank decrypts $c$ using $SK$ to obtain the key $k$.
7. The Bank permanently deletes $SK$ (and $PK$ too).

Note that the last step, where the Bank deletes the $SK$ is the reason why this has forward secrecy. The $(PK, SK)$ pair is only used for the AKE and then is deleted afterwards. The $SK_{bank}$ is only used for the signature. Thus, if the $SK_{bank}$ is compromised in the future, the attacker cannot read past sessions because they cannot obtain $SK$. No one has $SK$ anymore after the AKE. Therefore, this has forward secrecy.

Although this AKE has forward secrecy, it is not HSM secure. This is because if an attacker breaks into a Bank and queries the HSM once, it has complete key exposure forever. What the attacker can do is generate its own $(PK', SK')$. If the attacker breaks into the Bank and queries the HSM once using $PK'$, then the HSM will return $\sigma' = S(SK_{bank}, PK')$. Now the attacker has a valid signature $\sigma'$. The attacker can then send $PK'$ and $\sigma'$ along with $cert_{bank}$ to Alice. Alice will then check $\sigma'$, but $\sigma'$ is a valid signature, so Alice will think she's talking to the bank. She then generates the shared key $k$ and sends $c = E(PK', k)$, which the attacker can intercept and decrypt to obtain the key $k$. Then the attacker from then on can decrypt all messages from Alice to the Bank forever.

## 12.3 AKEs with HSM Security

We do not want the attack described above to happen, and thus we want to construct an AKE that is HSM secure. This is strongest level of security. In creating this AKE protocol, we may also want to preserve privacy to ensure that an eavesdropper doesn't learn to which service Alice is connecting to. This is called end-point privacy. Therefore, the $cert_{bank}$ must not be sent in plaintext if we want to have end-point privacy. This AKE protocol is as follows:

1. Alice generates a $(PK, SK)$ pair and sends $PK$ over to the Bank

2. The Bank generates two shared keys, $k, k'$.

3. The Bank encrypts $c = E(PK, (k, k'))$. The Bank creates a signature $\sigma = S(SK_{bank}, (PK, c))$ using the $SK_{bank}$ that is stored in the HSM. The Bank also uses symmetric encryption to encrypt $c' = E(k', (cert_{bank}, \sigma))$. The Bank sends $c, c'$ to Alice.

4. Alice uses $SK$ to decrypt $c$ and obtain $k, k'$. She then uses $k'$ to decrypt $c'$ to obtain $cert_{bank}, \sigma$.

5. Alice verifies $cert_{bank}$ using $PK_{CA}$. She then uses the $PK_{bank}$ obtained from this certificate to verify $\sigma$. Then she knows that she is truly talking to the bank and not an adversary. The key $k$ is now verified by both Alice and the Bank and they can use that as their symmetric encryption key for future messages.

For the PKE that is used in this AKE, we can use Diffie-Hellman. This is called DHAKE. There are many more AKE variants that are possible as well. In typical TLS 1.3, the TCP handshake is first performed, and then the AKE is performed.

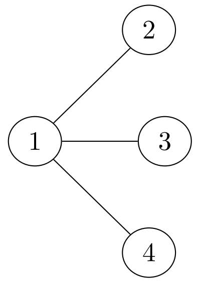
# 13 Zero Knowledge Protocols

In a zero knowledge protocol, we have two parties, a prover and a verifier. The goal is to create a system such that the prover can prove to the verifier that they know something, without revealing to the verifier any information about the thing that they know. The verifier needs to be able to distinguish whether or not the prover knows that thing without learning anything about that that thing. For example, in a zero knowledge protocol with the game "Where's Waldo," one person claims to have found Waldo. In a zero knowledge protocol, that person can prove to the other person that they know where Waldo is without revealing the location of Waldo. (This is easily done with a pair of scissors and another piece of paper).

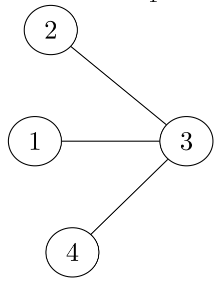
## 13.1 The Class NP

To understand zero knowledge, we first have to understand the class NP. The definition of NP is as follows. A language $L \subseteq \{0,1\}^*$ is in NP if there exists a polynomial time algorithm $M$ such that $x \in L$ iff there exists a $w \in \{0,1\}^*$ such that $M(x, w) = 1$. In this definition, $x$ is called a statement, and $w$ is a witness. What this definition means is that for a language to be in NP, for all elements in the language, there exists a witness such that when the witness and statement are provided a polynomial time algorithm $M$, that the algorithm accepts. If $x$ is not in the language, then $M$ rejects (returns 0). What is means is that for a language to be in NP, we can verify if an element in the in the language in polynomial time.

A concrete example of this is the problem of graph isomorphism. In graph isomorphism, we have given two different graphs: $G_0 = (V, E_0)$ and $G_1 = (V, E_1)$ and we want to figure out if they are isomorphic. We say they are isomorphic if there exists a one-to-one function $\phi : V \to V$ such that $(u, v) \in E_0$ maps to $(\phi(u), \phi(v)) \in E_1$. For example, say we have $G_0$ as below:



And we have $G_1$ as follows:



$G_0$ and $G_1$ are isomorphic because there exits a $\phi$ that maps between them one-to-one. We define $\phi$ where $1 \to 3, 2 \to 2, 3 \to 1, 4 \to 4$, and it is easy to see that this fulfills the definition of isomorphism.

In this example, the language is $L_{G_{iso}} = \{(G_0, G_1 | \exists\phi : V \to V : G_1 = G_0^{\phi}\}$. The statement $x$ is the pair $(G_0, G_1)$. Then, $\phi$ is a witness that $x \in L_{G_{iso}}$. We can clearly see that this is in NP since verifying $\phi$ (using $M$) is in polynomial time. Note that this problem is not NP-complete, but is not known to be in P.

## 13.2 Zero Knowledge Proof System

A zero knowledge proof system for a language $L \in$ NP is a pair of efficient probabilistic algorithms $(P, V)$ such that the prover $P(x, w)$ interacts with the verifier $V(x)$ and the verifier will return "yes" or "no". We denote the interaction between the prover and verifier $P(x, w) \leftrightarrow V(x)$. This proof system must fulfill the following properties:

1. Completeness: For all $x, w$, if $x \in L$, then P$[(P(x, w) \leftrightarrow V(x)) =$ "yes"$] = 1$. This means that if $x \in L$, then verifier will always return "yes".
2. Soundness: For all $x \notin L$, then for a false prover $\hat{P}$, P$[(\hat{P}(x) \leftrightarrow V(x)) =$ "yes"$] \leq \frac{1}{2}$. We can repeat this process $n$ times to decrease the accept probability to a negligible one of $\frac{1}{2^n}$. Note that the prover is this case does not have $w$ since $x \notin L$. This means that when $x \notin L$, the verifier will say "yes" with a negligible probability and say "no" otherwise.

3. Zero Knowledge: This protocol reveals "nothing" to to $V$, other than that $x \in L$, even if $V$ is malicious. More formally, for $x, w$, let transcript($(P(x, w) \leftrightarrow V(x))$ be the sequence of messages between $P(x, w)$ and $V(x)$. $(P, V)$ is zero knowledge for $L$ if for all efficient $\hat{V}$, there exists an efficient simulator $S$ such that for all $x \in L$, $S(x) \approx$ transcript($(P(x, w) \leftrightarrow \hat{V}(x))$. This means that there exists a simulator $S$ that only has the statement $x$, and can generate a distribution indistinguishable from the transcript. Therefore, clearly if anyone $(\hat{V}(x))$ can generate this simulator using only $x$, then nothing is revealed through this interaction between the prover and the verifier.

There is a theorem that proves that every $L$ in NP has an efficient ZK proof system. This can be proved by finding an efficient ZK proof system for an NP complete problem, and then since all NP problems can be transformed into that NP complete problem, then we have an efficient ZK proof system for all NP problems.

Let's examine the zero knowledge proof system of $L_{G_{iso}}$, the graph isomorphism problem. In this problem, the prover has $G_0, G_1$ and $\phi$ and the verifier have $G_0, G_1$. The prover wants to prove to the verifier that there is $\phi$ without revealing to the verifier what $\phi$ is. The protocol works as follows.

The prover chooses a random $\tau : V \to V$, which is a random one-to-one function. The prover also chooses a random $b$ that takes on the value either 0 or 1. Depending on which $b$ the prover chose, they will permute the corresponding graph with $\tau$. In other words, they will create $G = (V, E) = G_b^{\tau}$ and send $G$ over to the verifier. The verifier will pick a challenge $c$ at random that is either 0 or 1 and send it to the prover.

Note the following. Assume the prover had chosen $b = 1$. Then they permuted $G_1$ using $\tau$ to get $G$ and thus there is an isomorphism between $G_1$ and $G$. If there truly is a witness $\phi$, then to create an isomorphism from $G_0$ to $G$, the prover can easily just do that by $\tau \circ \phi$ (transforming $G_0$ to $G_1$ and then $G_1$ to $G$. If there is no isomorphism between $G_0$ and $G_1$, then the prover also cannot construct an isomorphism between $G_0$ and $G$. The reverse holds true if the prover randomly had chosen $b = 0$ instead. In summary, if the isomorphism exists, the prover has an isomorphic transformation from $G_0$ and $G_1$ to $G$ and if it does not than the prover only has a transformation from either $G_0$ or $G_1$ to $G$, but not the other.

Resuming the protocol, the prover has received $c$. If $c = b$, then the prover returns $\pi = \tau$ to the verifier. If $c = 1 - b$, then the prover returns $\pi = \tau \circ \phi$ to the verifier. The verifier checks to make sure that $G_c^{\pi} = G$. If it is, then the verifier "accepts," otherwise they "reject."

We can see that this fulfills the 3 properties. It satisfies completeness because if if $G_0$ and $G_1$ are isomorphic, then the prover will always produce a $\pi$ that is valid to the verifier. It satisfies soundness because if $G_0$ and $G_1$ are not isomorphic, then the prover will not be able to produce the correct $\pi$ half of the time. It satisfies zero knowledge because the verifier learns nothing about $\phi$ from this interaction. In fact, it is possible to construct a simulator $S$ that is indistinguishable from the transcript, $(G, c, \pi)$.

## 13.3 Zero Knowledge Proof of Knowledge System

So far, the zero knowledge proof system only proves to the verifier that $x \in L$. What if the verifier wants to be convinced that the prover actually "knows" the witness $w$. The idea is that if the prover "knows" $w$ then $w$ can be extracted from the prover. Therefore, zero knowledge proof of knowledge system (ZKPK) is stronger than a zero knowledge proof system and fulfills the properties:

1. Completeness: same as in zero knowledge proof system

2. Zero Knowledge: same as in zero knowledge proof system

3. Can extract witness from the prover (stronger property than soundness)

An example of a ZKPK system can be illustrated using the discrete log problem. Let $G$ be a finite cyclic group of order $q$ with generator $g \in G$. The prover has $\alpha$ and can construct $h = g^{\alpha}$. The verifier has only $h$. The prover selects a random $r$ from $\mathbb{Z}_q$, constructs $y = g^r$ and sends that over to the verifier. The verifier selects a random challenge $c$ from $\mathbb{Z}_q$ and sends that to the prover. The prover computes $z = r + c * \alpha \in \mathbb{Z}_q$ and returns that to the verifier. The verifier outputs "accept" if $g^z = h^c * y$.

So far we can see the similarity between this problem and the graph isomorphism problem, so this is a zero knowledge proof system. But does it satisfy the property that the witness can be extracted? We can show that it is possible to extract the $\alpha$ from a prover. First the prover selects a random $r$ to create $y$ and commits to that $y$ by sending it to the verifier. Then the verifier sends the challenge $c_1$ to the pover and the prover responds with $z_1$. If it were possible to rewind the prover and send it a different challenge $c_2$ and obtain $z_2$, we can see the following.

If the verifier "accepts," then $g^{z_1} = h^{c_1} * y$. Also, then $g^{z_2} = h^{c_2} * y$. Dividing the two equations, we get that $g^{z_1 - z_2} = h^{c_1 - c_2}$. Solving for this, we get that $h = g^{\frac{z_1 - z_2}{c_1 - c_2}}$. We can see that $\alpha$ can be extracted then by $\alpha = \frac{z_1 - z_2}{c_1 - c_2} \in \mathbb{Z}_q$. Therefore, since $\alpha$ can be extracted, a prover cannot convince the verifier with non-negligible probability unless it knows $\alpha$. Thus, this is a ZKPK.

## 13.4 Schnorr Signatures

Schnorr signatures are a popular signature scheme that is built on this problem of the discrete log ZKPK. In a Schnorr signature, the generator algorithm generates the secret key $\alpha$ and the public key $h = g^{\alpha}$. It then selects a random $r$ from $\mathbb{Z}_q$ and calculates $y = g^r$. Since there is no prover or verifier in this signature scheme, to receive a challenge, the signer simply calculates $H(m, y) = c$ given a certain hash function $H$ and the message they want to sign $m$. Then they compute $z = r = c * \alpha$ and output $\sigma = (c, z)$. The signature is usually 64 bytes, which is much shorter than an RSA signature of 256 bytes.

Someone verifying the signature can simply accept if $H(m, \frac{g^z}{h^c}) = c$, since $g^z = h^c * y$ if the signature is valid.

Schnorr signatures are become more and more popular due to their shorter length and ease of computation.