CS 255 (INTRODUCTION TO CRYPTOGRAPHY)

DAVID WU

Abstract. Notes taken in Professor Boneh's Introduction to Cryptography course (CS 255) in Winter, 2012. There may be errors! Be warned!

Contents

1. 1/11: Introduction and Stream Ciphers

1.1. **Introduction.** The goal of secure communication is that as data travels across a network, an attacker cannot (i) eavesdrop on the communication and (ii) cannot tamper with the data. This is done via Secure Sockets Layer (SSL)/TLS. SSL consists of two main parts: (i) a handshake protocol to establish a shared secret key (public key cryptography) and (ii) a record layer where data is transmitted using a shared secret key to ensure confidentiality and integrity. Another application of cryptography is to protect files on disk (no eavesdropping - confidentiality and no tampering - integrity). This is analogous to secure communication.

Building block of secure traffic is **symmetric encryption**. This consists of two algorithms an encryption algorithm E and a decryption algorithm D and a secret key k. Note that E and D should be *public* algorithms (should not use proprietary ciphers!). Given plaintext message m with key k, $E(k, m)$ yields ciphertext c and correspondingly, given ciphertext c and the key k, the decryption algorithm yields $D(k, c) = m$. There are two use cases of symmetric encryption. Single use keys (one time key) are keys that are only used to encrypt one message (i.e. encrypted email).

Multi-use keys (many time key) where one key is used to encrypt many messages (i.e. encrypted files). In general, we need more machinery to ensure security of multi-use keys.

Cornerstone of cryptography consists of establishment of shared secret key and secured communication. Other applications of cryptography include

- Digital signatures: creating signatures that cannot be forged.
- Anonymous communication: mix networks that allow user to anonymously communicate with server.
- Anonymous digital cash: spend a "digital coin" without merchant knowing who spent the money as well as prevent double spending. Idea here is that if the coin is spent once, anonymity is preserved, but if done more than once, anonymity is destroyed.

Another application is with multi-party protocols: several inputs try to jointly compute single function without revealing individual inputs.

- Elections: computing result of election / who has majority of votes without revealing individual vote. Voters send encrypted votes to election center so election center does not know individual vote, but is able to use encrypted votes to determine winner.
- Private auctions: bidders need to compute the 2nd highest bid without revealing actual bids.

One possible approach to secure multi-party computation is to rely on trusted authority (who computes the desired function without revealing inputs). This is not interesting from a cryptographic standpoint however. In fact,

Theorem 1.1. *Anything that can be done with a trusted authority can also be done without.*

In particular, in the case of multi-party computation, it is possible for the parties to communicate with one another such that at the end of the exchange, each party knows the value of the function, but not the value of any individual party.

Cryptography can also be used for privacy. In private information retrieval (PIR), Alice wants to look up a record in a database, but does not want anyone to know that she is interested in the particular record. There is trivial solution where database returns all the data to Alice, but using crypto, length of the returned value can be comparable to length of actual record.

There are also some magical application of cryptography. One is **privately outsourcing computation**. Suppose Alice issues a search query to Google; using crypto, it is possible for Alice to send a query to Google and get the result without the server knowing what the query is! Note that this particular example is not practical. Another application is **zero knowledge**. Suppose that Alice has a very large graph and is able to find a large clique in the graph. Alice can then communicate to Bob that there is a clique in the graph with certainty but Bob will have no idea what the clique is!

1.2. **History of Cryptography.** To communicate securely, Alice and Bob will share a secret key k that the attacker does not know. To communicate with Bob, Alice sends an encrypted message $c := E(k, m)$ to Bob who then decrypts it using $m := D(k, m)$. Two simple ciphers:

- Substitution cipher: the key defines a mapping from a character onto its substitute: $k := \{a \mapsto c, b \mapsto w, \ldots\}$
- Caesar cipher: substitution cipher where each character is shifted by 3 (not really a cipher since key is fixed): $\{a \mapsto d, b \mapsto e, \ldots\}$

For a substitution cipher, there are $26! \approx 2^{88}$ possible keys. Despite its large key-space, we can use frequency analysis of English letters to break it. First, we use frequency of English letters (i.e. most frequent character in the ciphertext is likely to be most frequent character in English and so on). We can then use frequency of pairs of letters (digrams) and trigrams, and so on, to recover the key. This is an example of a *ciphertext-only* attack, which is no more secure than just sending plaintext!

In the Vigener cipher, we take the key to be a common word. We repeat the key multiple times to span the length of the plaintext. To encrypt, we just add the letters in the plaintext to the letter in the key, taken modulo 26. To break this cipher, suppose first that we know the length ℓ of the key. This reduces to ℓ substitution ciphers, which we can break as above. Even if we do not know the length of the key, we can simply run the algorithm for $\ell = 1, 2, \ldots$ until we recover a sensible plaintext.

Rotor machines encode the substitution table in a electric rotor. Each time key is pressed, disk rotates and substitution table is shifted each time. Can still be broken using frequency analysis. Introducing additional rotors render it more difficult to break (more rotor positions), culminating in Enigma. Still broken using frequency analysis.

In digital age, encryption shift to computers. Government issue proposal for Data Encryption Standard (DES) in 1974. Has a key space of 2^{56} with a block size of 64 bits (encrypt 64 bits / 8 characters) at a time. Relatively small keyspace by modern standards and has been replaced by Advanced Encryption Standard (AES) (2001) and Salsa20 (2008).

1.3. **Stream Ciphers.**

Definition 1.2. A **cipher** is defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ where $\mathcal{K}$ is the set of all possible keys, $\mathcal{M}$ is the set of all possible messages and $\mathcal{C}$ is the set of all possible ciphertexts and consists of a pair of "efficient" algorithms $(\boldsymbol{E}, \boldsymbol{D})$ such that $\boldsymbol{E} : \mathcal{K} \times \mathcal{M} \mapsto \mathcal{C}$ and $\boldsymbol{D} : \mathcal{K} \times \mathcal{C} \mapsto \mathcal{M}$ which must satisfy the *correctness property* (consistency requirement):

$$\forall m \in \mathcal{M}, k \in \mathcal{K} : \boldsymbol{D}\left(k, \boldsymbol{E}(k, m)\right) = m$$

Here "efficient" runs in polynomial time (theoretical) or in a specific time period (practical). Oftentimes, $\boldsymbol{E}$ is randomized (for a randomized algorithm $\mathcal{A}$, the output $\mathcal{A}(x)$ should be taken to be a random variable), but $\boldsymbol{D}$ is deterministic.

Example of a "secure" cipher is One Time Pad (OTP). Here, $\mathcal{M} = \mathcal{C} = \{0,1\}^n$. Additionally, $\mathcal{K} = \{0,1\}^n$ so the key is as long as the message. In particular, we have

$$c := E(k, m) = k \oplus m \qquad D(k, c) = k \oplus c$$

We check the consistency requirement:

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

The One Time Pad is fast to decrypt and encrypt, but the keys are long, so impractical. We now consider possible (flawed) definitions of a secure cipher:

- If the attacker sees a single CT, he cannot recover the secret key. Under this notion of security, the substitution cipher is insecure (can recover secret key using frequency analysis). This is a poor definition since the identity cipher would be considered "secure:" $E(k, m) = m$ (impossible to retrieve key in this case!)
- If the attacker sees a single CT, he cannot recover the message. Consider the encryption algorithm $E(k, m_0 \| m_1) = (m_0 \| m_1 \oplus k)$ where $\|$ denotes concatenation. It is not possible to recover the original message given a single CT, but only part of the message has been encrypted!

The basic idea of the security of a cipher is that the ciphertext should reveal no information about the plaintext. More precisely,

Definition 1.3. A cipher $(\boldsymbol{E}, \boldsymbol{D})$ over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ has **perfect secrecy** if

$$\forall m_0, m_1 \in \mathcal{M} : \mathrm{len}(m_0) = \mathrm{len}(m_1) \text{ and } \forall c \in \mathcal{C} : \Pr\left[\boldsymbol{E}(k, m_0) = c\right] = \Pr\left[\boldsymbol{E}(k, m_1) = c\right]$$

where k in uniform in $\mathcal{K}$ $\left(k \xleftarrow{R} \mathcal{K}\right)$. In words, given a particular ciphertext, you cannot tell if message is m_0 or m_1 for all m_0 and m_1. In particular, the most powerful adversary cannot learn anything about the PT from the CT. There is no CT-only attack, though other attacks may be possible. More formal (and equivalent) definition: for all algorithms $\mathcal{A}$ and messages $m \in \mathcal{M}$, consider the following experiment:

$$\mathrm{Exp}\left[\mathcal{A}, m\right] := \begin{array}{l} k \xleftarrow{R} \mathcal{K} \\ c \xleftarrow{R} \boldsymbol{E}(k, m) \\ \text{output } \mathcal{A}(c) \end{array}$$

The cipher has perfect secrecy if

$$\forall \mathcal{A} \text{ and } \forall m_0, m_1, |m_0| = |m_1| : \mathrm{Exp}\left[\mathcal{A}, m_0\right] = \mathrm{Exp}\left[\mathcal{A}, m_1\right]$$

where $\mathrm{Exp}[\mathcal{A}, m]$ denotes a distribution over the outputs of the experiment. Thus, the two underlying distributions must be the same, and so no information about the message is leaked.

Lemma 1.4. *The One Time Pad has perfect secrecy.*

Proof. First, we see that

$$\forall m, c : \Pr_k\left[E(k, m) = c\right] = \frac{\text{number of keys } k \in \mathcal{K} \text{ such that } E(k, m) = c}{|\mathcal{K}|}$$

Thus, suppose we have a cipher such that

$$\forall m, c : \left|\{k \in \mathcal{K} : E(k, m) = c\}\right| = \text{constant}$$

In this case, the above probability is constant, and so the cipher will have perfect secrecy. In the case of the OTP, if $E(k,m) = c$, then $k \oplus m = c \Rightarrow k = m \oplus c$. Thus, for the OTP, $|\{k \in \mathcal{K} : E(k,m) = c\}| = 1 \ \forall m, c$ and so the OTP has perfect secrecy. There is no CT-only attack on the OTP! □

Theorem 1.5. *In order to have perfect secrecy, we must have* $|\mathcal{K}| \geq |\mathcal{M}|$.

1.4. **Pseudorandom Generators (PRGs).** To have perfect secrecy, the length of the key must be at least as long as the message, and thus, such ciphers can be difficult to use in practice. To address this, we replace the random key with a pseudorandom key. First, we define a **psuedorandom generator** (PRG) $G : \{0,1\}^s \mapsto \{0,1\}^n$ with $n \gg s$. Here, the input is denoted the seed of the PRG and $\{0,1\}^s$ is the seed space. Now, in our encryption algorithm, we let the key be the seed and use the PRG to expand the key to the length of the message. In particular, in a **stream cipher** we have

$$c = E(k,m) := m \oplus G(k) \qquad D(k,c) = c \oplus G(k)$$

Note that since the key is shorter than the message, the stream cipher cannot have perfect secrecy.

Definition 1.6. To ensure security, the PRG must be unpredictable. A PRG is **predictable** if

$$\exists \text{ efficient algorithm } \mathcal{A} \text{ and } \exists 1 \leq i \leq n-1 : \Pr_{k \xleftarrow{R} \mathcal{K}} \left[\mathcal{A}\left(G(k)|_{1,\ldots i} \right) = G(k)|_{i+1} \right] \geq \frac{1}{2} + \varepsilon$$

for some non-negligible ε. Here, $G(k)|_{1,\ldots,i}$ denotes the first i characters of $G(k)$. Conversely, a PRG is **unpredictable** if it is not predictable:

$$\forall i : \text{ no efficient algorithm can predict the } i+1 \text{ bit for non-neglegible } \varepsilon$$

In practice, ε is non-negligible if $\varepsilon \geq 2^{-30}$ and negligible if $\varepsilon \leq 2^{-80}$. More rigorously, in theory, we take ε as a function $\varepsilon : \mathbb{Z}^{\geq 0} \to \mathbb{R}^{\geq 0}$. Then, ε is non-negligible if $\exists d : \varepsilon(\lambda) \geq \lambda^{-d}$ infinitely often and ε is negligible, $\forall d, \lambda \geq \lambda_d : \varepsilon(\lambda) \leq \lambda^{-d}$.

Example 1.7. Examples of predictable and unpredictable PRGs:

- $\varepsilon(\lambda) = 2^{-\lambda}$ is negligible.
- $\varepsilon(\lambda) = \lambda^{-1000}$ is non-negligible (consider taking $d = 10000$).
- $\varepsilon(\lambda) = \begin{cases} 2^{-\lambda} & \lambda \text{ odd} \\ \lambda^{-1000} & \lambda \text{ even} \end{cases}$ is non-negligible since $\varepsilon(\lambda) \geq \lambda^{-d}$ infinitely often for $d \geq 1000$.

Linear congruential generators with parameters a, b, p are insecure! More precisely, we let $r[0] =$ seed and use the update $r[i] \leftarrow a \cdot r[i-1] + b \bmod p$. Each step, we output a few bits of $r[i]$. The `random()` function in `glibc` is also insecure!

Now we consider the notion of the security of PRGs. The goal in designing a secure PRG is that the output $G(k)$ of a PRG with seed $k \in \{0,1\}^s$ should be indistinguishable from a random string r taken from $\{0,1\}^n$. We begin by defining a statistical test on $\{0,1\}^n$.

Definition 1.8. A **statistical test** on $\{0,1\}^n$ is an algorithm $\mathcal{A}$ that takes $x \in \{0,1\}^n$ as input and outputs one bit $\in \{0,1\}$ where 0 denotes pseudorandom and 1 denotes random. Examples:

- $\mathcal{A}(x) = 1$ if and only if $|\#0(x) - \#1(x)| < 3 \cdot \sqrt{n}$ where $\#0$ denotes the number of 0s and $\#1$ denotes the number of 1s. Note that the standard deviation of the difference is $\sqrt{n}$.
- $\mathcal{A}(x) = 1$ if and only if $\frac{1}{3}\log_2 n \leq$ max-run-of-0$(x) \leq 3 \cdot \log_2 n$. Note that the expected length of the maximum run is $\log_2 n$.

Definition 1.9. The advantage of a statistical test $\mathcal{A}$ is given by

$$\text{Adv}_{\text{PRG}}[\mathcal{A}, G] := \left| \Pr_{k \xleftarrow{R} \{0,1\}^s} [\mathcal{A}(G(k)) = 1] - \Pr_{r \xleftarrow{R} \{0,1\}^n} [\mathcal{A}(r) = 1] \right|$$

If the advantage is close to 0, then $\mathcal{A}$ cannot distinguish the output of the generator G from random (PRG is good) and if the advantage is close to 1, then $\mathcal{A}$ can distinguish G from random (PRG is broken).

Example 1.10. Suppose $G(x)$ satisfies $G(k)|_1 \oplus G(k)|_2 \oplus \cdots \oplus G(k)|_n = 0$ for $\frac{2}{3}$ of the seeds. We define a statistical test as follows

$$\mathcal{A}(x) = 1 \iff x|_1 \oplus \cdots \oplus x|_n = 0$$

The advantage of this test is then

$$\text{Adv}\left[\mathcal{A}, G\right] = \left|\frac{2}{3} - \frac{1}{2}\right| = \frac{1}{6}$$

Definition 1.11. A PRG is **secure** if for all efficient statistical tests $\mathcal{A}$ if $\text{Adv}_{\text{PRG}}\left[\mathcal{A}, G\right]$ is "negligible." Note that the "efficiency" requirement is necessary for a meaningful definition.

We do not know whether there are any provably secure PRGs: this is the $P \neq NP$ problem (proving that a PRG is secure would imply that $P \neq NP$). Thus, we will assume that some problems are hard and use those to construct PRGs.

1.5. **Attacks on Stream Ciphers and OTP.** The two time pad is insecure! In particular, if we encrypt two messages $c_1 \leftarrow m_1 \oplus \text{PRG}(k)$ and $c_2 \leftarrow m_2 \oplus \text{PRG}(k)$ using the same pad. Then, an eavesdropper can simply compute $c_1 \oplus c_2 = m_1 \oplus m_2$. There is enough redundancy in English and ASCII encodings that given $m_1 \oplus m_2$, we can recover m_1 and m_2. Thus, never use stream cipher key more than once! Some real world examples:

- Project Venona: Use of two-time pads by Russians broken by US cryptoanalysis of CT.
- MS-PPTP (Windows NT): Messages from client and server both encrypted using the same key! Need to have key for client to server communication and key for server to client communication. In particular, we should treat $k = (k_{C\to S}, k_{S\to C})$ where $k_{C\to S}$ is for client to server communication and $k_{S\to C}$ is for server to client communication.
- 802.11b WEP: Key used to encrypt each frame consists of concatenation of IV (incremented after each message) with key. Length of IV is 24 bits and repeats after $2^{24} \approx 16$M frames. On some cards, IV resets to 0 after power cycle! Keys also share same suffix (very closely related) and for the PRG (RC4) used, key may be recovered after observing a small number of frames.
- Disk encryption: A small local change in a file can reveal where the modification takes place (unchanged portions will not change since same key is used).

Stream ciphers do not provide integrity (OTP is **malleable**). Given a message m encrypted with key k, the cipher text is $m \oplus k$. An attacker can then modify the encrypted message with $c \oplus p = (m \oplus k) \oplus p$. Upon decryption, we are left with $((m \oplus k) \oplus p) \oplus k = m \oplus p$, which has a very predictable effect on the plaintext: attacker can change content of message!

1.6. **Stream Ciphers in Practice.** An old example (1987) of a stream cipher is RC4. First, given a seed of length 128 bits, it first expands the seed to 2048 bits which is the internal state of the generator. Then, each iteration of a loop, 1 byte is generated. This is used in HTTPS and WEP. RC4 has several weaknesses (not truly random):

- Bias in initial output: $\Pr\left[2^{\text{nd}} \text{ byte} = 0\right] = \frac{2}{256} > \frac{1}{256}$. (Recommendation is to ignore first 256 bytes due to bias).
- Probability of sequence $(0,0)$ is given by $\frac{1}{256^2} + \frac{1}{256^3} > \frac{1}{256^2}$.
- Related key attacks as discussed above.

Another example is content scrambling system (CSS) used in old DVD systems. Used in hardware based on linear feedback shift register (LFSR). Register consists of sequence of cells that have certain taps used for xor operator. After each iteration, one bit shifted off and first bit of result shifted in. In CSS, the seed is 5 bytes (40 bits) consisting of a 17-bit LFSR and a 25-bit LFSR. The first LFSR is initialized with 1 concatenated with the first 2 bytes of key and the second with 1 concatenated with the last 3 bytes of key. Each LFSR generates 8 bits of output and the results are added together modulo 256 along with the carry from the previous block. This yields one byte per round. This is easy to break in time $\approx 2^{17}$. Given an encrypted DVD which has a common format, we know a prefix of the plaintext, in particular the first 20 bytes. If we xor the first 20 bytes of the encrypted segment with this prefix, we obtain the first 20 bytes of the output of the PRG. Now, for each of the 2^{17} possible initializations of the first LFSR, we run it for 20 bytes. Given this 20-byte output, we subtract it from the output of the CSS to obtain what would be the output of the 2nd LFSR. Given a 20-byte sequence, we can easily tell whether it originated from an LFSR or not. In doing so, we can recover the initialization state of the 17-bit LFSR, and correspondingly of the 25-bit LFSR. Given the initializations, we can predict the PRG and decrypt the DVD.

Modern stream cipher is eStream (2008). Stream cipher also incorporate a nonce (a non-repeating value for a given key). Now, the PRG is given by $\{0,1\}^s \times \mathbb{R} \to \{0,1\}^n$ and so

$$E(k, m; r) = m \oplus \text{PRG}(k; r)$$

where $r \in \mathbb{R}$ is a nonce. Note that the pair (k, r) is never used more than once (allows for reuse of keys k). Particular example of eStream is Salsa20: $\{0,1\}^{128 \text{ or } 256} \times \{0,1\}^{64} \to \{0,1\}^n$ given by

$$\text{Salsa20}(k; r) := H\left(k, (r, 0)\right) \| H\left(k, (r, 1)\right) \| \ldots$$

Note that $s_1 \| s_2$ denotes the concatenation of string s_1 and s_2 . The function H takes in 3 parameters: the key k, the nonce r and a counter i that increments. To evaluate H, we expand the states to 64-bytes:

τ_0	k	τ_1	r	i	τ_2	k	τ_3
4 bytes	16 bytes	4 bytes	8 bytes	8 bytes	4 bytes	16 bytes	4 bytes

Here, τ_i $0 \le i \le 3$ are constants defined in the specification. This expanded input is used as the argument to an invertible function $h : \{0,1\}^{512} \to \{0,1\}^{512}$. We apply h 10 times to the input and evaluate its sum with the input. There are no known attacks on this and has security on the order of 2^{128}. This is a very fast stream cipher on both hardware and software!

2. 1/18: PRGs and Semantic Security

2.1. **Secure PRGs.**

Proposition 2.1. *A secure PRG is unpredictable.*

Proof. We will show the contrapositive. Suppose that the PRG is predictable. In particular, consider an efficient algorithm $\mathcal{A}$ such that

$$\Pr_{k \xleftarrow{R} \mathcal{K}} \left[\mathcal{A}(G(k)|_{1,\ldots,i}) = G(k)|_{i+1}\right] = \frac{1}{2} + \varepsilon$$

Now, define a statistical test $B(x)$ where

$$B(x) = \begin{cases} 1 & \mathcal{A}\left(x|_{1,\ldots,i}\right) = x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Given a truly random string $r \xleftarrow{R} \{0,1\}^n$, $\Pr\left[B(r) = 1\right] = \frac{1}{2}$. Given, $k \xleftarrow{R} \mathcal{K}$, however, $\Pr\left[B(G(k)) = 1\right] = \frac{1}{2} + \varepsilon$ and so the advantage of the statistical test is given by

$$\text{Adv}_{\text{PRG}}\left[B, G\right] > \varepsilon$$

and so we have a statistical test that can distinguish the output of the PRG with a non-negligible probability ε, and so, the PRG is not secure. □

Theorem 2.2. *An unpredictable PRG is secure. More precisely, if $G : K \to \{0,1\}^n$ is a PRG, then for all $i \in \{0, \ldots, n-1\}$, G is unpredictable at position i, then G is a secure PRG.*

In other words, if next-bit predictors cannot distinguish G from random, then no statistical test can!

Example 2.3. Let $G : K \to \{0,1\}^n$ be a PRG such that from the last $\frac{n}{2}$ bits of $G(k)$, it is easy to compute the first $\frac{n}{2}$ bits. We show that G is predictable for some $i \in \{0, \ldots, n-1\}$.

Proof. Clearly, G is not secure since we can predict the first $\frac{n}{2}$ bits using the last $\frac{n}{2}$ bits. But then, by Theorem 2.2, G is predictable. □

Definition 2.4. Let P_1 and P_2 be two distributions over $\{0,1\}^n$. Then, P_1 and P_2 are **computationally indistinguishable** (denoted $P_1 \approx_P P_2$) if for all efficient statistical tests $\mathcal{A}$, the quantity

$$\left|\Pr_{x \xleftarrow{R} P_1}\left[\mathcal{A}(x) = 1\right] - \Pr_{x \xleftarrow{R} P_2}\left[\mathcal{A}(x) = 1\right]\right|$$

is negligible.

Using this notation, we may write that a PRG is secure if $\left\{k \xleftarrow{R} \mathcal{K} : G(k)\right\} \approx_P \text{Uniform}\left(\{0,1\}^n\right)$.

2.2. **Semantic Security.** Our goal is to define a notion of security that is more practical than perfect secrecy. In the case of perfect secrecy, we may write our condition using the above notation

$$\{E(k, m_0)\} = \{E(k, m_1)\} \text{ for } k \overset{R}{\leftarrow} \mathcal{K}$$

In the case of semantic security, we relax this constraint and take

$$\{E(k, m_0)\} \approx_P \{E(k, m_1)\} \text{ for } k \overset{R}{\leftarrow} \mathcal{K}$$

for all explicit messages m_0 and m_1. We now define semantic security for a one-time key.

Definition 2.5. Consider an adversary $\mathcal{A}$ that selects two messages $m_0, m_1 \in \mathcal{M}$ where $|m_0| = |m_1|$. The challenger than encrypts one of the messages $E(k, m_b)$ using a $k \overset{R}{\leftarrow} \mathcal{K}$ and sends it back to the adversary. The adversary now guesses the message $b' \in \{0, 1\}$ that yielded the particular encryption. Let W_0 be the event where $b = 0$ and the adversary says $b' = 1$ and let W_1 be the event where $b = 1$ and the adversary says $b' = 1$. Then, a cipher (E, D) is **semantically secure** if the advantage

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

is negligible for all efficient $\mathcal{A}$.

Example 2.6. If a cipher is semantically secure, then the adversary cannot learn any bit of the PT from the CT.

Proof. We consider the contrapositive. Suppose $\mathcal{A}$ is an efficient algorithm such that $\mathcal{A}(E(k, m)) = \text{lsb}(m)$ where lsb denotes the least significant bit of the message. Let the adversary output two messages m_0, m_1 where $\text{lsb}(m_0) = 0$ and $\text{lsb}(m_1) = 1$. The encrypted message is then $c = E(k, m_b)$ for $b \in \{0, 1\}$ with $k \overset{R}{\leftarrow} K$. Consider the advantage of $\mathcal{A}$

$$\text{Adv}_{\text{SS}}[\mathcal{A}, E] = |\Pr(W_0) - \Pr(W_1)| = \left| \underbrace{\Pr(b = 0, b' = 1)}_{0} - \underbrace{\Pr(b = 1, b' = 1)}_{1} \right| = 1$$

and so the cipher is not semantically secure. Note that we can easily extend this construction to apply to any system that leaks any bit of the plaintext. Thus, any semantically secure encryption scheme cannot leak any bit of information. □

Example 2.7. The OTP is semantically secure.

Proof. Consider the advantage $\text{Adv}_{\text{SS}}(\mathcal{A}, \text{OTP})$ for any efficient algorithm $\mathcal{A}$. Note that in the case of the OTP, for two messages m_0 and m_1, the distributions $k \oplus m_0$ and $k \oplus m_1$ are identically distributed (no algorithm can distinguish them). Then, W_0 and W_1 are identical events and so

$$\text{Adv}_{\text{SS}}(\mathcal{A}, \text{OTP}) = |\Pr(W_0) - \Pr(W_1)| = 0$$

which is negligible for all $\mathcal{A}$. □

Theorem 2.8. *Given a secure PRG G, the derived stream cipher is semantically secure. Equivalently (contrapositive), given a semantically secure adversary $\mathcal{A}$, then there exists a PRG adversary $\mathcal{B}$ such that*

$$Adv_{SS}[\mathcal{A}, E] \leq 2 \cdot Adv_{PRG}[\mathcal{B}, G]$$

Proof. Let $\mathcal{A}$ be a semantically secure adversary. Let the challenger select a random $r \overset{R}{\leftarrow} \{0, 1\}^n$ and instead of sending the encrypted string $c = m_b \oplus G(k)$, the challenger sends the string $c = m_b \oplus r$. However, if the PRG is secure, $G(k)$ should be indistinguishable from r. But now this is the OTP and we know that the OTP is semantically secure, so the stream cipher with a secure PRG is corresponding secure. More precisely, let W_b denote the event that the adversary outputs $b' = 1$ for message m_b encrypted with the PRG and let R_b denote the event that the adversary outputs $b' = 1$ for message m_b encrypted with a random string r (OTP). Since R_0 and R_1 correspond to a semantic security game against the OTP, we have that

$$|\Pr(R_0) - \Pr(R_1)| = \text{Adv}_{\text{SS}}(\mathcal{A}, \text{OTP}) = 0$$

Thus, we see that $\Pr(R_0) = \Pr(R_b) = \Pr(R_1)$. Furthermore, we claim that there exists a $\mathcal{B}$ such that

$$|\Pr(W_b) - \Pr(R_b)| = \text{Adv}_{\text{PRG}}(B, G)$$

To see this, we explicitly construct the PRG adversary $\mathcal{B}$. Given an input string $y \in \{0,1\}^n$, run adversary $\mathcal{A}$ to produce two messages m_0 and m_1. We encrypt $m_0 \oplus y$ and send it to $\mathcal{A}$. The output of $\mathcal{B}$ will just be the output b' of $\mathcal{A}$. Now, the advantage of $\mathcal{B}$ will be given by

$$\text{Adv}_{\text{PRG}}(\mathcal{B}, G) = \left| \Pr_{r \xleftarrow{R} \{0,1\}^n} [\mathcal{B}(r) = 1] - \Pr_{k \xleftarrow{R} \mathcal{K}} [\mathcal{B}(r) = 1] \right| = |\Pr(R_0) - \Pr(W_0)|$$

which concludes the claim. Given this we now have,

$$-\text{Adv}_{\text{PRG}}(\mathcal{B}, G) \leq \Pr(W_b) - \Pr(R_b) \leq \text{Adv}_{\text{PRG}}(\mathcal{B}, G)$$

and so

$$\text{Adv}_{\text{SS}}[\mathcal{A}, E] = |\Pr(W_0) - \Pr(W_1)| \leq 2 \cdot \text{Adv}_{\text{PRG}}[\mathcal{B}, G]$$

Thus, if a PRG is secure, then the advantage of any efficient adversary $\mathcal{B}$ will be negligible, which correspondingly implies that the advantage of any adversary $\mathcal{A}$ against the stream cipher is negligible. □

2.3. **Generating Random Bits in Practice.** RFC 4086 provides ways of collecting entropy on a computer. In general, we measure interrupt timings (i.e. disk access times, mouse events, packet timings) and use them as the input to a random number generator (RNG). The RNG outputs a random bit whenever sufficient entropy has been collected. On Linux, use `/dev/random` and on Windows, *only* use `BCryptGenRandom` in the CNG API. We can also use inherent randomness in hardware for generating random numbers. For instance, in the Intel processors (Ivy Bridge) contains a processor command `RdRand` to generate true randomness. Note that some postprocessing is used to extract true entropy. This is very fast: 3 `Gb/s`!

Suppose you have an entropy source that generates a sequence of bits. While the generated bits are independent, they are *biased*; in particular, $\Pr(0) = p$ and $\Pr(1) = 1 - p$. Our goal is to generate random bits with *uniform* probability. To do this, we can consider pairs of bits. If we see a $(0,0)$ or a $(1,1)$, we ignore the pair. If we instead see $(0,1)$, we output a 1 and if we see $(1,0)$, we output a 0. Since the bits are independent, $\Pr(1) = p(1-p) = \Pr(0)$ as desired. To correct for dependent bits, we often use hashing.

2.4. **Block Ciphers.** Using a PRG with a stream cipher, we can only use a key once. To remedy this problem, we consider a block cipher, which takes in a block of n bits and also outputs a block of n bits. A block cipher still consists of an encryption algorithm E and a decryption algorithm D. Canonical examples include 3DES where $n = 64$ and $k = 168$ bits and AES where $n = 128$ bits and $k = 128,\ 192,\ 256$ bits. Here, k denotes the length of the key. A block cipher works via repeated iteration. In particular, given the initial key k, we expand the key using a PRG to obtain a set of keys $\{k_1, \dots, k_n\}$. Then, given a message m, we iteratively apply a round function $R(k, m)$ using keys $k_1, \dots, k_n$ in succession. Note that in each round, we take m to be the output of the previous round (for the first round, we take our input to be the message m). For 3DES, the round function is applied 48 times and in AES-128, it is applied 10 times. In terms of running time, stream ciphers outperform block ciphers.

3. 1/23: Block Ciphers

3.1. **Pseudorandom Functions (PRF) .**

Definition 3.1. A **pseudorandom function** (PRF) defined over $(\mathcal{K}, X, Y)$ where $\mathcal{K}$ denotes the key space, X denotes the input space and Y denotes the output space is a function $F : \mathcal{K} \times X \to Y$ such that there exists an "efficient" algorithm to evaluate $F(k, x)$. A **pseudorandom permutation** (PRP) defined over $(\mathcal{K}, X)$ is a function $E : \mathcal{K} \times X \to X$ such that there exists an "efficient" *deterministic* algorithm to evaluate $E(k, x)$, and $E(k, \cdot)$ is one-to-one, and there exists an "efficient" inversion algorithm $D(x, y)$. Note that this is not a permutation of the bits, but a permutation of the *space* over which the function operates.

Two examples of PRPs is 3DES ($\mathcal{K} = X = \{0,1\}^{128}$) and AES ($X = \{0,1\}^{64}$ and $\mathcal{K} = \{0,1\}^{168}$). Furthermore, note that a PRP is a PRF where $\mathcal{X} = \mathcal{Y}$ that is efficiently invertible. Now, consider the notion of a secure PRF. In particular, let $F : \mathcal{K} \times X \to Y$ be a PRF. Define the set Funs$[X, Y]$ is the set of all functions from X to Y. This set has size $|Y|^{|X|}$. Let the set $S_F = \{F(k, \cdot) \mid k \in \mathcal{K}\} \subseteq \text{Funs}[X, Y]$ denote the set of all functions specified by the PRF when we specify the key. Intuitively then, a PRF is secure if a random function in Funs$[X, Y]$ is indistinguishable from a random function in S_F. More precisely, for $b = 0, 1$, we define an experiment Exp(b) where in the case where $b = 0$, the challenger takes $k \leftarrow \mathcal{K}$ and $f \leftarrow F(k, \cdot)$ and for $b = 1$, the challenger takes $f \leftarrow \text{Funs}[X, Y]$. The adversary may now submit queries $x_i \in X$ to the challenger and the challenger responds with the value $f(x_i)$.

Finally, the attacker decides whether the PRF is pseudorandom or truly random. As usual, we take the advantage of the adversary to be

$$\text{Adv}_{\text{PRF}}[\mathcal{A}, F] = |\Pr[\text{Exp}[0] = 1] - \Pr[\text{Exp}[1] = 1]|$$

A PRF is secure if this advantage is "negligible." In that case, the adversary cannot distinguish a pseduorandom function from a truly random function. We can define a similar definition for PRPs. For $b = 0, 1$, define experiment Exp(b) where in the case $b = 0$, the challenger takes $k \leftarrow \mathcal{K}$ and $f \leftarrow E(k, \cdot)$ and for $b = 1$, the challenger takes $f \leftarrow \text{Perms}[X]$. The adversary now submits queries x_i to the challenger and the challenger responds with $f(x_i)$. The adversary then decides whether the output is due to a pseudorandom or a truly random PRP. In particular, the advantage of the PRP is given by

$$\text{Adv}_{\text{PRP}}[\mathcal{A}, E] = |\Pr[\text{Exp}[0] = 1] - \Pr[\text{Exp}[1] = 1]|$$

and a PRP is secure if for all "efficient" $\mathcal{A}$, the advantage is "negligible." Note that a block cipher is secure if it is a secure PRP.

Example 3.2. Let $F : \mathcal{K} \times C \to \{0,1\}^{128}$ be a secure PRF, and consider

$$G(k, x) = \begin{cases} 0^{128} & \text{if } x = 0 \\ F(k, x) & \text{otherwise} \end{cases}$$

It is easy to see that this is not a secure PRF. The adversary would just have to provide the input $x = 0$ and if the response is 0, the adversary can simply say "not random" (would only occur x^{-128} by chance) This shows that as long as there is one input for which the output is predictable, then the PRF is not secure.

Example 3.3. It is easy to construct a PRG from a PRF. In particular, given $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a secure PRF. Then, the following $G : \mathcal{K} \to \{0,1\}^{\text{nt}}$ is a secure PRG:

$$G(k) = F(k, 0) \| F(k, 1) \| \cdots \| F(k, t)$$

The key property is that this is nicely parallelizable (can compute $F(k, \cdot)$ in parallel). Note that the security follows from the PRF property: $F(k, \cdot)$ is indistinguishable from a random function $f(\cdot)$. Had we used a truly random function F to construct $G(k) = F(0) \| F(1) \| \cdots \| F(k)$, the output would certainly not be predictable, and thus would yield a truly random string. But if the PRF is secure, then its output would be indistinguishable from a truly random function, and so its output would be indistinguishable from a truly random string. Hence, the resultant PRG is necessarily secure.

Example 3.4. All 2^{80}-time algorithms $\mathcal{A}$ will have $\text{Adv}[\mathcal{A}, \text{AES}] < 2^{-40}$.

Lemma 3.5 (PRF Switching Lemma). *Let E be a PRP over $(\mathcal{K}, X)$. Then, for any q-query adversary $\mathcal{A}$, we have*

$$|\text{Adv}_{\text{PRF}}[\mathcal{A}, E] - \text{Adv}_{\text{PRP}}[\mathcal{A}, E]| \le \frac{q^2}{2|X|}$$

Thus, if $|X|$ is large so that $\frac{q^2}{2|X|}$ is "negligible," then $\text{Adv}_{\text{PRP}}[\mathcal{A}, E]$ is negligible implies that $\text{Adv}_{\text{PRF}}[\mathcal{A}, E]$ is also "negligible." In other words, if we have a secure PRP, we also have a secure PRF. Thus, AES is both a PRF and a PRP.

3.2. **Data Encryption Standard (DES).** The core idea of DES is a Feistel network . Given functions $f_1, \ldots, f_d$ that map from $\{0,1\}^n \to \{0,1\}^n$. Our goal is to build an invertible function $F : \{0,1\}^{2n} \to \{0,1\}^{2n}$. Given our input with $2n$ bits, we divide it into two blocks consisting of n bits each. Denote them R_0 and L_0. Then, we compute $L_1 = R_0$ and $R_1 = f_1(R_0) \oplus L_0$. This constitutes one round of the Feistel network. In general, the round function of a Feistel network operates as follows:

$$\begin{cases} R_i = f_i(R_{i-1}) \oplus L_{i-1} \\ L_i = R_{i-1} \end{cases}$$

for $i = 1, 2, \ldots, d$.

Claim 3.6. For all $f_1, \ldots, f_d$, the Feistel network $F : \{0,1\}^{2n} \to \{0,1\}^{2n}$ is invertible.

Proof. We explicitly construct the inverse. In particular, we take

$$\begin{cases} R_i = L_{i+1} \\ L_i = f_{i+1}(R_i) \oplus R_{i+1} = f_{i+1}(L_{i+1}) \oplus R_{i+1} \end{cases}$$

□

Thus, we see that the inversion process is analogous to the encryption process, only that $f_1, \ldots, f_d$ are applied in reverse order. This is attractive for hardware developers since the encryption and decryption mechanisms are identical. This is a general method for building invertible functions (block ciphers).

Theorem 3.7 (Luby-Rackoff)**.** *If $f : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ is a secure PRF, then a 3-round Feistel network $F : \mathcal{K}^3 \times \{0,1\}^{2n} \to \{0,1\}^{2n}$ is a secure PRP (secure block cipher). In particular, the PRF is used in every round of the Feistel network (the function f computed with a independent secret key in each round of the Feistel network).*

DES is a 16-round Feistel network with $f_1, \ldots, f_{16} : \{0,1\}^{32} \to \{0,1\} \to 64$, where $f_i(x) = \mathbf{F}(k_i, x)$ where k_i is the round key derived from the main key k. Then, the input to the DES cipher consists of 64 bits. The input is first permuted (not for security reasons, just specified in the standard). The permuted bits are used as input to the 16-round Feistel network. The output goes into a reverse permutation function (again, not necessarily for security regions). Note that the initial key k is expanded to 16 round keys $k_1, \ldots, k_{16}$. To invert the cipher, we simply use the 16 round keys in reverse order. Now, we describe the specifics of the function $\mathbf{F}(k_i, x)$ where $x \in \{0,1\}^{32}$ and $k_i \in \{0,1\}^{48}$. First, the input x is used as the input to an expansion box that replicates certain bits to construct a 48-bit input. This result is xored with the round key k_i. The resultant 48 bits are divided into 8 groups of 6 bits each and used as inputs to S-boxes $S_1, \ldots, S_8$ where $S_i : \{0,1\}^6 \to \{0,1\}^4$. The resultant 32 bits (8 blocks of 4 bits from the S-box) is then permuted (P-box) and these permuted bits comprise the output of $\mathbf{F}$. Note that the S-boxes are implemented as simple lookup tables that specify the 4-bit mapping for each possible 6-bit input.

Example 3.8. Suppose $S_i(x_1, \ldots, x_6) = (x_2 \oplus x_3, x_1 \oplus x_4 \oplus x_5, x_1 \oplus x_6, x_2 \oplus x_3 \oplus x_6)$ or written equivalently $S_i(x) = A_i \cdot x \bmod 2$. In this case,

$$A_i = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

In this case, S_i becomes a linear function of x. In this case then, the DES cipher would be fully linear (just computing xor and bit permutations). Then, there will exist a matrix B such that

$$\mathrm{DES}(k, m) = B \cdot \begin{pmatrix} m \\ k_1 \\ k_2 \\ \vdots \\ k_{16} \end{pmatrix} = c \bmod 2$$

In this case, then

$$\begin{aligned} \mathrm{DES}(k, m_1) \oplus \mathrm{DES}(k, m_2) \oplus \mathrm{DES}(k, m_3) &= B \begin{pmatrix} m_1 \\ k \end{pmatrix} \oplus B \begin{pmatrix} m_2 \\ k \end{pmatrix} \oplus B \begin{pmatrix} m_3 \\ k \end{pmatrix} \\ &= B \begin{pmatrix} m_1 \oplus m_2 \oplus m_3 \\ k \oplus k \oplus k \end{pmatrix} \\ &= \mathrm{DES}(k, m_1 \oplus m_2 \oplus m_3) \end{aligned}$$

which is certainly not a random function.

Thus, the S-boxes must be carefully chosen to ensure the security of DES. In fact, even choosing the S-boxes and the P-box at random would yield an insecure block cipher (key recovery may be possible after $\approx 2^{24}$ outputs). Thus, in designing S and P boxes, we have that the output bit should not be close to a linear function of the input bits. Some other considerations are also made to ensure security.

3.3. **Advanced Encryption Standard (AES) .** AES is build as a substitution-permutation networks and not a Feistel network, so all the bits are changed in *every* round. In particular, each round, we xor the input with the round key k_i, then send the result through a substitution phase (bits mapped based upon substitution table), and finally send the result through a permutation layer. Note that all layers in AES are reversible (applying each step in reverse order). In the case of AES-128, we begin with a 4-by-4 matrix of bytes (128 bits in total). Each round, we xor the block with the key k_i, then apply `ByteSub`, `ShiftRow`, and `MixColumn` to the xored result, in that order. Each key k_i is also a 4-by-4 array of bytes. In total there are 10 rounds. We now describe each of the three operations:

- `ByteSub`: This is a 1-byte S-box consisting of 256 bytes. There is one 1 S-box for each entry in the 4-by-4 matrix. Then, $\forall i, j : A[i, j] \leftarrow S[A[i, j]]$ where $A[i, j]$ denotes the (i, j) element in the input 4-by-4 block and $S[A[i, j]]$ denotes the entry in the S box (lookup table) that corresponds to $A[i, j]$
- `ShiftRows`: Permutation: cyclic shift of the rows in the matrix. Second row shifted by one position, third row shifted by two positions, and fourth row shifted by three positions.
- `MixColumns`: Switch columns around in the 4-by-4 matrix.

It is possible to precompute a lot of the operations to improve performance at the expense of code size. For instance, an AES implementation in Javascript may not include the precomputed tables so as to minimize network traffic. However, on the client side, the Javascript can precompute the tables so as to improve performance. There are also hardware implementations of AES. For instance, in Intel Westmere, `aesenc` and `aesenclast` perform one round of AES using 128-bit registers on the processor.

There are only two known attacks on AES at this time. Currently, the best key recovery attack is only four times better than exhaustive search (key size effective 2^{126}). Related key attack of AES-256: given 2^{99} input and output pairs from four *related* keys in AES-256 can recover keys in time approximately 2^{99} (while still impractical, this is a huge improvement over exhaustive search which would take 2^{256} time in this key-space). In practice though, AES keys will not be related, so such an attack is difficult to carry out.

3.4. **Exhaustive Search Attacks.** Our goal is given a few input/output pairs $(m_i, c_i = E(k, m_i))$, $i = 1, \ldots, 3$, recover key k.

Lemma 3.9. *Suppose DES is an **ideal cipher** (each key $k \in \mathcal{K}$, DES implements a random invertible function, so DES is a collection of 2^{56} random invertible functions $\{0,1\}^{64} \to \{0,1\}^{64}$. Note that the DES keyspace is $\{0,1\}^{56}$). Then, for all m, c there is at most one key such that $c = \mathrm{DES}(k, m)$ with probability at least $1 - \frac{1}{256}$.*

Proof. Consider the probability that there is a key $k' \neq k$ such that $\mathrm{DES}(k, m) = \mathrm{DES}(k', m)$

$$\Pr\left[\exists k' \neq k : c = \mathrm{DES}(k, m) = \mathrm{DES}(k', m)\right] \leq \sum_{k' \in \{0,1\}^{56}} \Pr\left[\mathrm{DES}(k, m) = \mathrm{DES}(k', m)\right]$$

using the union bound. For a fixed $\mathrm{DES}(k, m)$, this is the probability that a random permutation of bits is precisely $\mathrm{DES}(k, m)$. Since there are 64-bits in $\mathrm{DES}(k, m)$, this is just $\frac{1}{2^{64}}$. Since there are 2^{56} keys in DES, we have then

$$\Pr\left[\exists k' \neq k : c = \mathrm{DES}(k, m) = \mathrm{DES}(k', m)\right] \leq \frac{2^{56}}{2^{64}} = \frac{1}{2^8} = \frac{1}{256}$$

Thus, the probability that the key is unique is at most $1 - \frac{1}{256}$, as desired. □

Now, if we have two DES pairs $(m_1, c_1 = \mathrm{DES}(k, m_1))$ and $(m_2, c_2 = \mathrm{DES}(k, m_2))$, the probability that there is only one key is $1 - 2^{-71}$ which is effectively 1. Thus, given two plaintext/ciphertext pairs, we can perform an exhaustive search over the keyspace for the key. By 1999, a DES key could be recovered in 22 hours. Thus, the scheme is completely insecure. Thus, 56-bit ciphers should not be used!

We can strengthen DES against exhaustive search by expanding the keyspace. In particular, consider the Triple-DES (3DES) scheme as described below. Let $E : \mathcal{K} \times M \to M$ be a block cipher. Then, take $3E : \mathcal{K}^3 \times M \to M$ where

$$3E\left((k_1, k_2, k_3), m\right) = E(k_1, D(k_2, E(k_3, m)))$$

Note that if $k_1 = k_2 = k_3$, 3DES is equivalent to DES (hardware implementation of Triple-DES may be used for normal DES). For 3DES, the key-size is now 168 bits, but three times slower than DES.

As an alternate construction, consider instead double DES. In particular, define $2E((k_1, k_2), m) = E(k_1, E(k_2, m))$. Suppose we have plaintext messages $M = (m_1, \ldots, m_{10})$ with ciphertexts $C = (c_1, \ldots, c_{10})$. Our goal is to find (k_1, k_2) such that $E(k_1, E(k_2, m)) = C$. In particular, we note that we have $E(k_2, m) = D(k_1, C)$. Thus, we consider a meet-in-the-middle attack. First, we construct a table where for each possible key (all 2^{56} DES keys) and compute the encryption of $E(k,m)$ and sort the table on $E(k,m)$. This takes time in $2^{56}\cdot\log(2^{56})$ (construction + sorting). Now, for each key k, decrypt the ciphertext c by computing $D(k,c)$ and check if the entry exists in the table. If we have found a pair of keys $E(k^i,M)=D(k,C)$, then $(k^i,k)=(k_1,k_2)$ is just the pair that we are interested in. The attack runs in time

$$\underbrace{2^{56}\log 2^{56}}_{\text{build table}}+\underbrace{2^{56}\log 2^{56}}_{\text{test decryption and search in table}}<2^{63}\ll 2^{112}$$

so this is not secure. Note that we can perform a similar attack on 3DES, which will require time 2^{118}, which is considered sufficiently secure. In both cases, we will need space to store the table, which has order 2^{56}.

Another method for strengthening DES is DESX. In particular, if $E:\mathcal{K}\times\{0,1\}^n\to\{0,1\}^n$ be a block cipher. Then, define $EX((k_1,k_2,k_3),m)=k_1\oplus E(k_2,m\oplus k_3)$. For DESX, the key length will be $64+56+64=184$ bits. Note that there is an easy attack in time $2^{64+56}=2^{120}$ time (left as exercise). Note that if we just perform one of the xor operations, $k_1\oplus E(k_2,m)$ and $E(k_2,m\oplus k_1)$ does nothing (equally vulnerable to exhaustive searches!)

3.5. **More Attacks on Block Ciphers.** We can attack a block cipher based upon its implementation details. For instance, a side channel attack may be to measure the time to perform encryption/decryption or measure the power needed for encryption/decryption. In doing so, we can recover the length of the secret key. Another possible attack is to exploit computation errors in the last round of the block cipher (fault attack). Do not implement primitives by hand: use established libraries!

Given *many* input and output pairs, we can recover the key in time less than 2^{56} (time better than exhaustive search) using linear cryptanalysis. Let $c=\text{DES}(k,m)$ and suppose for a random k,m, we have

$$\Pr\left[\underbrace{m[i_1]\oplus\cdots\oplus m[i_r]}_{\text{subset of message bits}}\oplus c\underbrace{[j_1]\oplus\cdots\oplus c[j_v]}_{\text{subset of CT bits}}=\underbrace{k[l_1]\oplus\cdots\oplus k[l_u]}_{\text{subset of key bits}}\right]=\frac{1}{2}+\varepsilon \tag{3.1}$$

Theorem 3.10. *Suppose our cipher satisfies (3.1). Then, given $\frac{1}{\varepsilon^2}$ random $(m,\ c=\text{DES}(k,m))$ pairs, then*

$$k[l_1,\dots,l_u]=\text{majority}\left[m[i_1,\dots,i_r]\oplus c[j_1,\dots,c_v]\right]$$

with probability of at least 97.7%.

For DES, there is a weak linear relation in the 5th S-box, and so (3.1) is satisfied with $\varepsilon=2^{-21}$. Thus, with, 2^{42} input/output pairs, we can find 14 bits of the key in time 2^{42}. We can now brute force the remaining $56-14=42$ bits in time 2^{42}. Thus, we have a 2^{43} method to break DES, assuming we are given 2^{42} random input/output pairs. Thus, even a small amount of linearity will greatly weaken the security of the cipher.

Finally, we consider quantum attacks. First, we define the generic search problem. Let $f:X\to\{0,1\}$ be a function and our goal is to find $x\in X$ such that $f(x)=1$. For a classical computer, the best generic algorithm time will be $O(|X|)$. Using a quantum computer, however, we may now solve the problem in time $O\left(|X|^{1/2}\right)$. Now, consider a quantum exhaustive search. Given $m, c=E(k,m)$, let us define

$$f(k)=\begin{cases}1 & \text{if } E(k,m)=c\\ 0 & \text{otherwise}\end{cases}$$

If we have a quantum computer, then we can recover key k in time $O\left(|K|^{1/2}\right)$ using an exhaustive search. For DES, we can break the cipher with time 2^{28} while for AES, we can break it with time 2^{64}, both of which would be considered insecure. Using AES-256 (256 bit keys), the cipher would still be secure against a quantum exhaustive search.

3.6. **Block Cipher Modes of Operation.** An insecure mode of block-cipher operation is electronic code book (ECB). Here, each block in the plaintext is encrypted in the same way (independently of the others). In particular, if $m_1=m_2$, then $c_1=c_2$. Thus, ECB leaks information about the plaintext! More precisely, we can prove that ECB is not semantically secure. Consider the semantic security game, where the adversary sends two messages $m_0=$ `Hello World` and $m_1=$ `Hello Hello`, each consisting of two blocks. The challenger then replies with the encryption of m_b for $b=0,1$, which will consist of two blocks $(c_1,c_2)\leftarrow E(k,m_b)$. The adversary then outputs 0 if $c_1=c_2$ and 1 otherwise. The advantage here will be close to 1 and so, ECB mode is not secure.

A secure construction is deterministic counter mode from a PRF F. Here,

$$E_{\text{DETCTR}}(k,m) = [m[0], m[1], \ldots, m[L]] \oplus [F(k,0), F(k,1), \ldots, F(k,L)] = [c[0], c[1], \ldots, c[L]]$$

Note that this a stream cipher built from a PRF (such as AES, 3DES). Note also that a secure PRF implies a secure PRG.

Theorem 3.11. *For any $L > 0$, if F is a secure PRF over $(\mathcal{K}, X, X)$, then E_{DETCTR} is a semantically secure cipher over $\left(\mathcal{K}, X^L, X^L\right)$. In particular, for any adversary $\mathcal{A}$ attacking E_{DETCTR}, there exists a PRF adversary B such that*

$$\text{Adv}_{\text{SS}}\left[A, E_{\text{DETCTR}}\right] = 2 \cdot \text{Adv}_{\text{PRF}}\left[B, F\right]$$

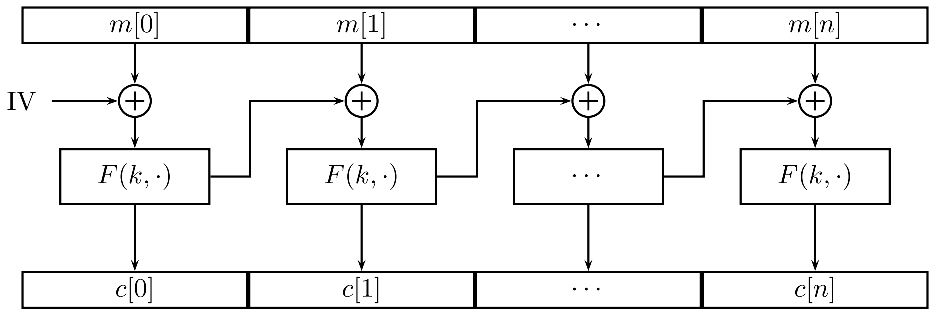
Proof. Consider two experiments. The adversary supplies two messages m_0 and m_1. Define Exp$[b]$ where the challenger returns $m_b \oplus [F(k,0), \ldots, F(k,L)]$. Consider now an alternate experiment where we xor with a truly random key. But this will be an attack on the OTP, which is impossible. Since a secure PRF is indistinguishable from a truly random key, E_{DETCTR} is indistinguishable from encryption with the OTP, which is semantically secure. ☐

Now, we define semantic security for many-time keys. In particular, we relax the assumption that each key is used only once. In particular, the key is now used more than once and the adversary sees many ciphertexts with the same key. Furthermore, we now define a chosen-plaintext attack (CPA). To define semantic security for many-time keys, we define Exp$[b]$ as:

(1) The adversary chooses any $m \in \mathcal{M}$ and receives the encryption $E(k,m)$.
(2) Then, the adversary plays the standard semantic security game.

Given this definition, we see that deterministic ciphers are not CPA secure. Thus, if the same secret key is to be used more than once, the encryption must produce different outputs. To allow for this, we define nonce-based encryption. In particular, the nonce n is a value that changes from message to message and the pair (k,n) is always unique. There are two methods for nonce-based encryption: either pick the nonce at random ($n \leftarrow N$) (would have to include nonce with the ciphertext) or use a counter. If we pick the nonce at random, there is a chance that we pick the same nonce twice (for 128 bit keys like with AES, this will likely happen after 2^{64} nonces have been generated)

One nonce-based mode of operation is ciphertext chaining (CBC). We begin by choosing a random, unpredictable IV (nonce). To encrypt, we have $c[0] = E(k, m[0] \oplus IV)$ and $c[1] = E(k, m[1] \oplus c[0])$ and so on where $c[i] = E(k, m[i] \oplus c[i-1])$. The ciphertext then consists of $[IV, c[0], c[1], \ldots, c[n]]$. Graphically, this is shown below:



where F is our block cipher encryption function.

Theorem 3.12. *For any $L > 0$, if E is a secure PRP over $(\mathcal{K}, X)$, then E_{CBC} is semantically secure under CPA over $\left(\mathcal{K}, X^L, X^{L+1}\right)$. In particular, for a q-query adversary $\mathcal{A}$ attacking E_{CBC}, there exists a PRP adversary $\mathcal{B}$ such that*

$$\text{Adv}_{\text{CPA}}\left[\mathcal{A}, E_{\text{CBC}}\right] \leq 2 \cdot \text{PRP}[\mathcal{B}, E] + 2 \cdot \frac{q^2 L^2}{|X|}$$

Thus, CBC is only secure as long as $q^2L^2 \ll |X|$ (repetition in nonces due to birthday paradox). The first construction we consider is CBC with unique nonce. We may compute a unique nonce by taking a possibly nonrandom IV and encrypting it with the block cipher to compute a IV$'$. We then use IV$'$ as the IV in CBC mode. Now, since the block cipher is a PRF, IV$'$ is effectively random, and so we can achieve CPA security using a possibly nonrandom nonce. In AES, we also have to pad the message so that each block has 16 bytes. In TLS, we introduce an n-byte pad where each entry in the pad has value n. Note that if no padding is needed, we need to introduce a dummy block at the end (otherwise, we cannot distinguish payload from padding).

In the randomized counter mode of operation for block ciphers, we begin by choosing a random IV. Then, we encrypt the message by encrypting each plaintext block i with $F(k, IV+i)$: $m[i] \oplus F(k, IV+i)$. Note that randomized counter-mode can be parallelized: each block can be encrypted independent of the previous ones. In nonce-based counter mode, we take the IV to be nonce$\|$counter where the nonce and the counter are each 2^{64} bits. Note that what is important is that any particular IV is used exactly once! Counter mode is preferable to CBC mode of encryption. In particular, a counter mode only requires a PRF (counter mode does not require that we be able to invert the function since we do not require the inverse to decrypt), but CBC mode requires a PRF (we have to run the block cipher in "decryption" mode which requires a PRF). Counter mode can also run in parallel, has better security guarantees ($q^2 L < 2^{96}$ as opposed to $q^2 L^2 < 2^{96}$ - stated below), does not require dummy padding (simply truncate the key for the last block).

Theorem 3.13. *For any $L > 0$, if F is a secure PRF over $(\mathcal{K}, X, X)$ then E_{CTR} is a semantically secure under CPA over $\left(\mathcal{K}, X^L, K^{L+1}\right)$. In particular, for a q-query adversary $\mathcal{A}$ attacking E_{CTR}, there exists a PRF adversary $\mathcal{B}$ such that*

$$\text{Adv}_{\text{CPA}}\left[\mathcal{A}, E_{\text{CTR}}\right] \leq 2 \cdot \text{Adv}_{\text{PRF}}\left[\mathcal{B}, F\right] + 2 \cdot \frac{q^2 L}{|X|}$$

Note: it is possible to use a stream cipher multiple times by using a block cipher to generate a unique key for each message. This makes sense because stream ciphers are much faster to use than block ciphers; thus, use them to encrypt the bulk of the message and block cipher for the shorter key.

4. 1/25: Message Integrity

4.1. **Message Integrity.** Encryption without integrity is useless! Encryption provides security against eavesdropping, but does not protect from the attacker modifying or corrupting the ciphertext. Examples where integrity is important include the delivery of ads or software patches (fine for the data to be public, but we want to prevent an attacker from modifying it). In particular, for a given message m, the sender will include a tag $S(k, m)$ computed as a function of the message m and some secret key k. The verifier is then a function $V(k, m, t)$ that outputs "yes" or "no" to indicate whether the message is properly signed or not. To achieve integrity, we always need a secret key! An example of a keyless integrity check is CRC (cyclic redundancy check). The problem is that the attacker can compute the tag for a given message and thus, alter the message and compute a new tag for the message. Such signatures are suitable for checking for *random* errors, not malicious errors.

Definition 4.1. A **message authentication code** (MAC) I consists of two algorithms (S, V) defined over $(\mathcal{K}, \mathcal{M}, \mathcal{T})$ where $\mathcal{K}$ is the key space, $\mathcal{M}$ is the message space, and $\mathcal{T}$ is the tag space. Then, $S(k, m) = t \in \mathcal{T}$ and $V(k, m, t)$ outputs "yes" or "no." A MAC must satisfy the consistency requirement: $\forall k, m : V(k, m, S(k, m)) =$ "yes".

Now, we define the notion of a **secure MAC**. We consider a **chosen message attack** in which the attacker can submit messages $m_1, \ldots, m_q$ and will receive $t_i \leftarrow S(k, m_i)$. The goal of the attacker is an **existential forgery** where he produces some valid (m, t) where $(m, t) \notin \{(m_1, t_1), \ldots, (m_q, t_q)\}$. In other words, the attacker should not be able to produce a valid tag for *any* message. Formally, we consider the following "MAC game." Consider a MAC $I = (S, V)$ and an adversary $\mathcal{A}$. Now, the challenger begins by generating a random key k. The adversary then sends messages $m_1, \ldots, m_q$ and receives $t_1 \leftarrow S(k, m_1), \ldots, t_q \leftarrow S(k, m_q)$. The adversary then submits a message (m, t). The challenger then outputs 1 if $V(k, m, t) = 1$ and $(m, t) \notin \{(m_i, t_i)\}$ and 0 otherwise.

Definition 4.2. Formally, a MAC $I = (S, V)$ is secure if for all efficient adversaries $\mathcal{A}$, the advantage

$$\text{ADV}_{\text{MAC}}[\mathcal{A}, I] = \Pr\left[\text{challenger outputs 1}\right]$$

is negligible.

Note that we distinguish between an adaptive vs. non-adaptive adversary. The non-adaptive adversary must send all queries simultaneously, while the adaptive adversary can select his messages based upon the previous response.

Example 4.3. Consider a MAC where S has a particular cyclical property: $S(k, 0) = s_1$ and $S(k, s_1) = s_2$ and $S(k, s_2) = 0$. To an non-adaptive attacker, this MAC would be secure (the attacker would have no way of knowing

s_2 ahead of time). To an adaptive attacker however, he can send the message 0 and get back s_1. From s_1, he can get s_2. Now, with s_2, the attacker can forge the message $(s_2, 0)$ which defeats the MAC (wins the "MAC game").

Example 4.4. Let $I = (S, V)$ be a MAC and suppose an attacker is able to find $m_0 \neq m_1$ such that $S(k, m_0) = S(k, m_1)$ with probability $\frac{1}{2}$. This is clearly not secure since the attacker can just request $S(k, m_0)$ and produce $S(k, m_1)$ which is an existential forgery. The advantage of this attack will then be $\frac{1}{2}$.

Example 4.5. Let $I = (S, V)$ be a MAC and suppose that $S(k, m)$ is always 5-bits long. This is also not secure since the attacker can just guess the tag for the message. The attacked will be correct with probability $\frac{1}{2^5} = \frac{1}{32}$ which is not negligible. Thus, this example shows that MACs cannot be too short - typical MAC lengths are 64-bits or 96-bits.

One application of MACs is to protect system files. For example, suppose at install time, the system computes $t_i = S(k, F_i)$ where F_i is a file on the OS and k is some key derived from the user's password. Later, a virus infects a system and modifies some of those files. The user can verify file integrity by rebooting into a clean OS, supply his password and compute the MAC on each of the files. If the MAC is secure, the virus cannot alter the file and generate a new checksum for it (existential forgery), and so the tampering can be detected. Note however that this method does not protect against *swapping* of the files: the virus can still switch F_1 with F_2, which can also cause harm to the OS. To protect against this attack, we can include the filename as part of the MAC for the file as well.

Note that any secure PRF is already a secure MAC. In particular, let F be a PRF over $(\mathcal{K}, X, Y)$. Then, take $I_F(S, V)$ to be

$$S(k, m) = F(k, m)$$
$$V(k, m, t) = \begin{cases} 1 & \text{if } t = K(k, m) \\ 0 & \text{otherwise} \end{cases}$$

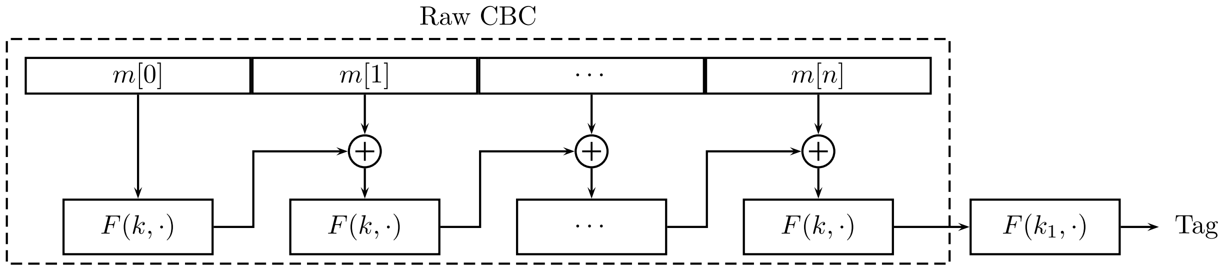
Lemma 4.6. *For all MAC adversaries $\mathcal{A}$, there exists a PRF adversary $\mathcal{B}$ such that*

$$\text{Adv}_{\text{MAC}}[\mathcal{A}, I] \leq \text{MAC}_{\text{PRF}}[\mathcal{B}, F] + \frac{1}{|Y|}$$

Proof. Suppose F is random in Funs$[X, Y]$. Then, the adversary sees $F(m_1), \ldots, F(m_q)$ and needs to guess $F(m)$ where $m \notin \{m_1, \ldots, m_q\}$. For a function, $F(m)$ is independent of everything else, so the best the adversary can do is guess a random value, which will occur with probability $\frac{1}{|Y|}$. □

To be secure, we note that $\frac{1}{|Y|}$ must be negligible in order for the MAC to be secure. From the above lemma, we see that AES is a MAC for 16-byte messages. The goal is to use this MAC to construct a MAC for larger messages. There are several possible constructions such as CBC-MAC and HMAC. Suppose a MAC is built from a PRF that outputs n bits ($n = 128$ in the case of AES). It is OK to truncate the MAC to w bits as long as $\frac{1}{2^w}$ is negligible. This hinges on the fact that a truncated PRF is still a PRF.

We consider the encrypted CBC-MAC (ECBC) . Let F be a PRF from $\mathcal{K} \times X \to X$ such as AES. Consider a new PRF F_{CBC} defined by the following construction:



The secret key then consists of (k, k_1). Thus, F_{CBC} is defined over $\left(\mathcal{K}^2, X^{\leq L}, X\right)$. Note that if we do not xor with $F(k', \cdot)$, we have raw-CBC mode, which is *not* secure.

5. 1/27: Proofs in Cryptography

5.1. **Time/Space Tradeoff.** We consider a time-space attack against 3DES with 2 keys. Then, the keyspace has size 2^{56} and there are 2^{64} possible plaintexts. In the case of 3DES with 2 keys, we take the message, encrypt it first with k_1, then with k_2, and finally again with k_1 (note not k_3 here):

$$m \xrightarrow{k_1} s \xrightarrow{k_2} m_2 \xrightarrow{k_1} c$$

We now consider the CPA semantic security game. The attacker employs a standard meet-in-the-middle attack. Fix the string s to be the output of the first encryption step $E(k_1, m)$. Then, the attacker computes $D(k_1, s) = m$ and submits a chosen-plaintext query with m to get the corresponding ciphertext c. Now, given c, we can compute $D(k_1, c) = m_2$. The attacker then builds a table consisting of (k_1, m_2), requiring 2^{56} space. Given this table, we start brute forcing k_2. For each candidate k_2, we encrypt $E(k_2, s)$ and see if the result is in the table of (k_1, m_2). If so, then we have found a pair of keys (k_1, k_2) such that $m' = E(k_2, s) = m$:

$$m \xrightarrow{k_1} s \xrightarrow{k_2} m_2' = m_2 \xrightarrow{k_1} c$$

Note however that the probability that there exists keys k_1, k_2 such that this is satisfied is on the order 2^{-64}, but there are 2^{112} possible keys in the keyspace. Hence, there is a high likelihood that our particular choice of k_1 and k_2 are not the actual pairs. We can repeat this with multiple blocks to decrease this probability. More precisely, we construct a table using $E(k_1, s_1, \ldots s_\ell) = (m_{2,1}', \ldots, m_{2,\ell}')$. We then build up a table for $(k_1, m_{2,1}', \ldots, m_{2,\ell}')$ and repeat the above attack. The probability that a pair of keys will yield messages $(m_{2,1}, \ldots, m_{2,\ell}) = (m_{2,1}', \ldots, m_{2,\ell}')$ will now have probability $2^{-64\cdot\ell}$. With $\ell = 3$, the probability will be sufficiently low. Note that a standard brute-force against this will require 2^{112} time, while the meet-in-the-middle attack will require 2^{57} time (brute force k_1 and k_2 independently) and 2^{56} space.

5.2. **Proofs in Cryptography.** In cryptography, we often want to prove statements of the form if A is secure, then B is secure. Oftentimes, it is much easier to prove the contrapositive (show that an attack on our scheme will lead to an attack on a system known to be secure).

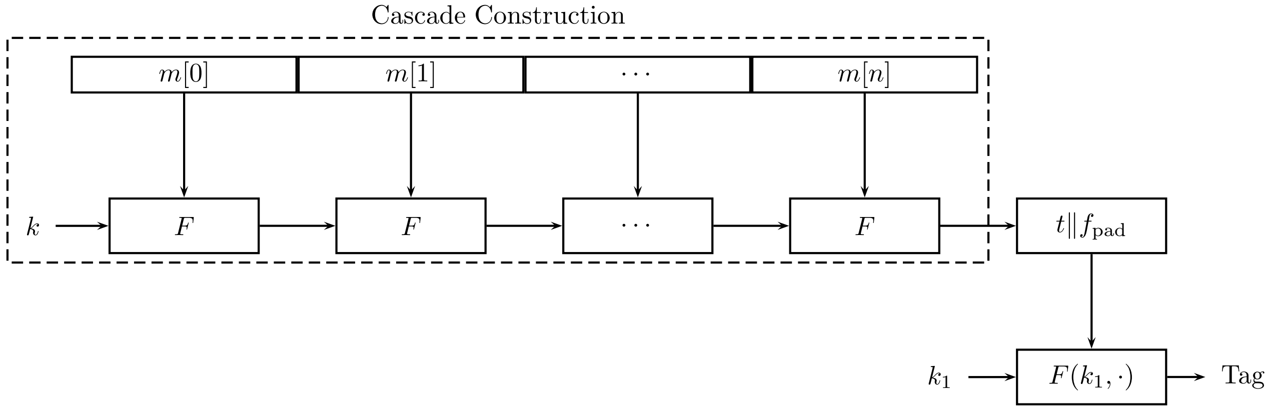
Example 5.1. If $G : X \to X^\ell$ is a secure PRG, then $F : X \times [\ell] \to X$ where $F = F(k, i) = [G(k)]_i$ is a secure PRF. Note that $[\ell] = \{0, \ldots, \ell - 1\}$.

Proof. To show this, we consider the contrapositive. In particular, we will construct an adversary against G by using an adversary against F. Consider the following PRG challenger. We consider two experiments $b = 0, 1$. If $b = 0$, then the challenger takes $y \leftarrow X^\ell$ and if $b = 1$, the challenger takes $s \leftarrow X$ and $y = G(s)$. Consider a PRF adversary that submits a query i to the challenger. The challenger responds with $F(k, i) = y_i$ (the i^{th} block of the PRG output). Now if $b = 0$, then y is truly random, and correspondingly, all sub-blocks of y is truly random. In particular, y_i will also be truly random. On the other hand, if $b = 1$, then y is the output of a PRG so by definition, y_i is just $F(s, i)$ where s is the key. Therefore, if $b = 0$, then the result is truly random and if $b = 1$, it appears that we are interacting with the PRF. But if we could now distinguish between these two cases, we have broken the PRG (determined the value of b with non-negligible advantage). Since the PRG is secure, the PRF must correspondingly be secure. □

The above example shows that it is possible to construct a PRF from a PRG. In particular, if $G : K \to K^2$ be a secure PRG, we can define a 1-bit PRF as $F : K \times \{0, 1\} \to K$ where $F(k, x \in \{0, 1\}) = G(k)[x]$ where $G(k)[x]$ denotes block x in the output of the PRG. We can then extend this PRG by defining $G_1 : K \to K^4$ with $G_1(k) = G(G(k)[0]) \| G(G(k)[1])$. We now prove that G_1 is secure. To do so, consider a construction where we take $G_1'(k) = G(r_0) \| G(r_1)$ where r_0 and r_1 are truly random. The result of this PRG should be indistinguishable from the original PRG; otherwise, we would be able to distinguish the output of $G(k)$ from random, which is impossible for a secure G. Now, given this, we consider a game where we output $r_{00} \| r_{01} \| G(r_1)$ where r_{00} and r_{01} are truly random bits. Now, if the adversary can distinguish this from being truly random, then he can distinguish $G(r_1)$ from being truly random, which is again impossible. Thus, the PRF game is approximately equivalent to the truly random game, and so if G is secure, G_1 is secure. This is an example of a *hybrid argument* since our construction has both a truly random component as well as a pseudo-random component. Note that these two constructions are only approximately equal. Such a construction is known as the Goldreich-Goldwasser-Micali (GGM) construction of a PRG.

6. 1/30: MAC Functions

6.1. **Message Integrity.** An alternate construction to raw CBC is NMAC where we have a PRF $F : \mathcal{K} \times X^{\leq L} \to \mathcal{K}$ which uses a cascade structure. Here, given a message m partitioned into n blocks, we compute



The above construction is $F_{\text{NMAC}} : \mathcal{K}^2 \times X^n \to K$. Note that the output of the cascade construction is a value $t \in \mathcal{K}$ and since the length of the values in the key space is generally smaller than the message space X, we pad the messages accordingly.

Theorem 6.1. *For all $L > 0$, and for every PRF adversary $\mathcal{A}$ that makes at most q queries, there exists an adversary $\mathcal{B}$,*

$$\text{Adv}_{\text{PRP}}\left[\mathcal{A}, F_{\text{ECBC}}\right] \leq \text{Adv}_{\text{PRP}}\left[\mathcal{B}, F\right] + \frac{2q^2}{|x|}$$
$$\text{Adv}_{\text{PRF}}\left[\mathcal{A}, F_{\text{NMAC}}\right] \leq \text{Adv}_{\text{PRF}}\left[\mathcal{B}, F\right] + \frac{q^2}{2\,|x|}$$

Thus, a MAC is secure as long as $q \ll \sqrt{|x|}$.

Example 6.2. Consider ECBC MAC with AES. If we want $\text{Adv}_{\text{PRF}}\left[\mathcal{A}, F_{\text{ECBC}}\right] \leq \frac{1}{2^{32}}$, then

$$\frac{q^2}{|x|} \leq \frac{1}{2^{32}} \Rightarrow q \leq \frac{\sqrt{|x|}}{2^{16}} = \frac{\sqrt{2^{128}}}{2^{16}} = 2^{48}$$

and so if we use AES-CBC-MAC, we need to change keys every 2^{48} messages! For 3-DES, the corresponding bound is only 2^{16}, which is relatively small.

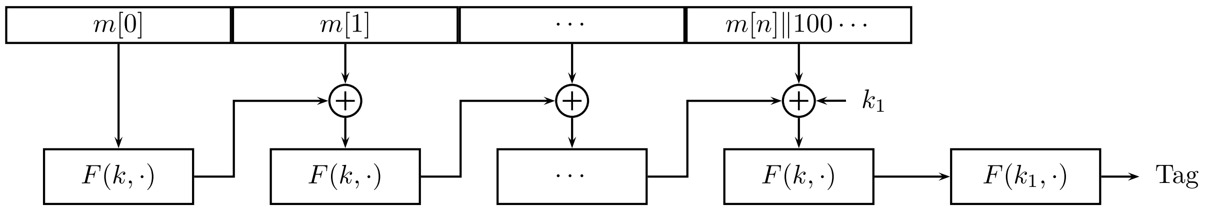
Now, we consider why it is necessary to xor with a different key in each of the MAC constructions. Without this step, the cascade construction of HMAC will be vulnerable to an **extension attack**. Suppose we have the cascade of (k, m), then it is possible to compute the cascade of $(k, m\|m_1)$. Given the cascade of (k, m), we have enough information to compute the next block. The last step of NMAC prevents these extensions (if we use the same key, a chosen-message attack can break the MAC). Now, we note that raw CBC mode is not vulnerable to the same extension attack since without knowledge of the key, we cannot evaluate the PRF. However, consider the following adversary. Take a message consisting of a single block m and request its tag $t \leftarrow F(k, m)$. Then, the adversary then submits the forgery $m' = (m, t \oplus m)$. Now,

$$\text{RawCBC}(k, (m, t \oplus m)) = F(k, F(k, m) \oplus (t \oplus m)) = F(k, t \oplus (t \oplus m)) = F(k, m) = t$$

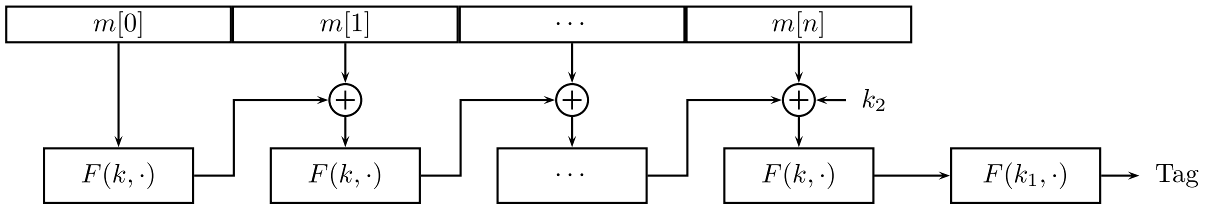
which completes the forgery. Note that if the length of the blocks is fixed, then RawCBC will be secure: RawCBC is a secure PRF on $\left(\mathcal{K}, X^L, X\right)$.

6.2. **MAC Padding.** We consider the problem where our message length is not an integer multiple of the block size. Thus, we have to introduce padding to the message. One possibility is to append a block of all 0s. However, this is not secure! To see this, consider an attacker that begins by choosing a message m that is not a multiple of 16 bytes. Then, request tag of m to get a tag t. The attacker then submits the forgery $(m\|0, t)$. The primary problem here is that there are many messages m_0 and m_1 where $\text{pad}(m_0) = \text{pad}(m_1)$. Thus, to ensure security, the padding must be invertible (one-to-one). One possibility is to use the same TLS mapping as before (append the number of bytes with the value of each byte equal to the length of the byte). In the ISO standard, we add a block consisting of a 1 followed by all 0s. Note that in both cases, we must introduce a dummy block if no padding is required. This is necessary in order that the pad be invertible (the space of messages with length equal to the block size is much smaller than the space of all the possible messages, so any invertible pad must introduce a dummy block in some case).

Another possibility is to use the key to generate a pad (CMAC). We consider two cases: when the last block of the message is a multiple of the block length and one when it is not. In the first case, we pad the message with 1 followed by all 0s, xor the result with some secret key k_1 and then encrypt one more time with F to yield a tag. In the second case, we xor the result with k_2, encrypt with F to obtain the tag. Note that this is a *3-key construction* (one for the CBC, and k_1, k_2). More precisely, in the case m needs to be padded, we have

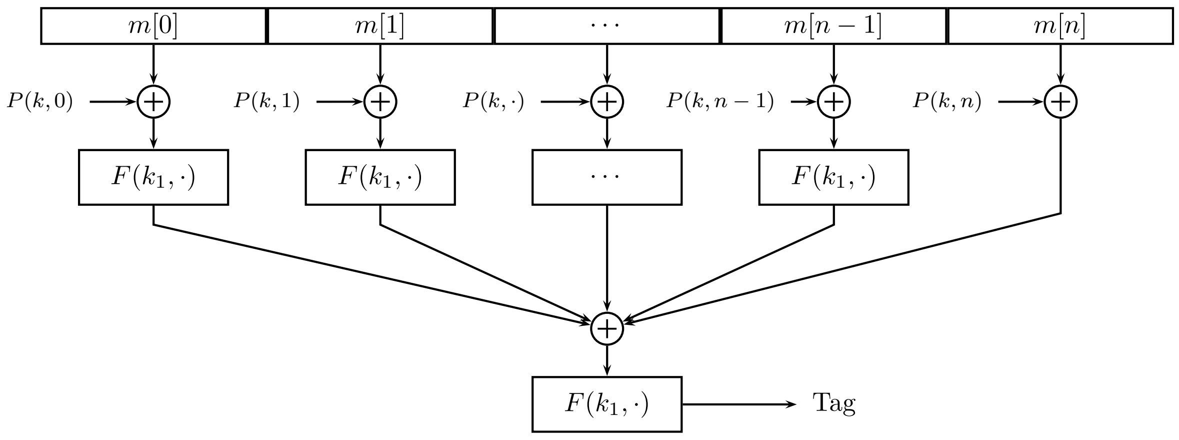


and in the case where no padding is necessary, we have



Note that extension attacks are thwarted by the last keyed xor. Note also that we do not need a dummy block in this case (ambiguity is resolved by the two keys). We can prove that CMAC is a secure PRF.

6.3. **Parallel MAC (PMAC) .** The MAC constructions we have so far are sequential MACs (cannot be nicely parallelized). Let $F : \mathcal{K} \times X \to X$ be a secure PRF. Then, we may define a new PRF $F_{\text{PMAC}} : \mathcal{K}^2 \times X^{\leq L} \to X$:



Note that the functions P is essential for the security of the scheme. Otherwise, we can simply permute the blocks without affecting the tag which leads to very simple existential forgeries. Thus P enforces a notion of order in the block.

$$P(k,i) = \left[i \times k \text{ in } \mathrm{GF}(2^{128})\right]$$

where $\mathrm{GF}(2^{128})$ denotes the general field of order 2^{128}.

Theorem 6.3. *For any $L > 0$, if F is a secure PRF over $(\mathcal{K}, X, X)$, then F_{PMAC} is a secure PRF over $\left(K, X^{\leq L}, X\right)$. In particular, for every efficient q-query PRF adversary $\mathcal{A}$ attacking F_{PMAC}, there exists an efficient PRF adversary $\mathcal{B}$ such that*

$$\mathrm{Adv}_{\mathrm{PRF}}\left[\mathcal{A}, F_{\mathrm{PMAC}}\right] \leq \mathrm{Adv}_{\mathrm{PRF}}\left[\mathcal{B}, F\right] + \frac{2q^2L^2}{|X|}$$

Thus, PMAC is secure as long as $qL \ll \sqrt{|X|}$.

Example 6.4. PMAC is incremental. In particular if we change one block $m[i]$ to $m'[i]$, we can easily update the tag by computing

$$F^{-1}(k_1, \text{tag}) \oplus F(k_1, m[1] \oplus P(k,1)) \oplus F(k_1, m'[1] \oplus P(k,1))$$

and then apply $F(k_1, \cdot)$.

6.4. **One-time MAC .** We consider a MAC that is only used for one message (analog of one-time pad for integrity). Then, for a MAC $I = (S, V)$ and an adversary $\mathcal{A}$, we can define a MAC game as follows:

(1) $\mathcal{A}$ submits a message $m_1 \in \mathcal{M}$ and receives $t_1 \leftarrow S(k, m_1)$.
(2) $\mathcal{A}$ submits a message-tag pair (m, t). The challenger outputs $b = 1$ if $V(k, m, t) = \text{yes}$ and 0 otherwise.

Then, I is a **secure MAC** if for all efficient $\mathcal{A}$,

$$\mathrm{Adv}_{\mathrm{MAC}}\left[\mathcal{A}, I\right] = \Pr\left[b = 1\right]$$

is negligible. Such MACs can be much faster than PRF-based MACs.

Example 6.5. Let q be a large prime. Then, take the key to be a pair $(k, a) \in \{1, \dots, q\}^2$ and take a message $m = (m[1], \dots, m[L])$. Then,

$$S(k, m) = P_m(k) + a \bmod q$$

where $P_m(x) = m[L] \cdot x^L + \cdots + m[1] \cdot x$ is a polynomial of degree L. Then, given only $S(k, m_1)$, the adversary has no information about $S(k, m_2)$ where m_2 is some other message. Note that this is a one-time MAC and not secure if the key is reused!

It is possible to convert a one-time MAC into a many-time MAC. Let (S, V) be a secure one-time MAC over $(\mathcal{K}_l, \mathcal{M}, \{0,1\}^n)$. Let $F : \mathcal{K}_F \times \{0,1\}^n \to \{0,1\}^n$ be a secure PRF. Then the **Carter-Wegman MAC** is defined as follows

$$\mathrm{CW}\left((k_1, k_2), m\right) = \left(r, F(k_1, r) \oplus S(k_2, m)\right)$$

for a random $r \leftarrow \{0,1\}^n$. Notice that the fast one-time MAC is applied to the long input (m) and the slower, PRF is applied to the shorter input (r). Verification in this scheme is very simple:

$$V\left(r, t\right) = V_2\left(k_2, m, F(k_1, r) \oplus t\right)$$

where V_2 denotes the verification algorithm for the one-time MAC.

Theorem 6.6. *If (S, V) is a secure one-time MAC and F a secure PRF, then CW is a secure MAC outputting tags in $\{0,1\}^{2n}$.*

Note that the Carter-Wegman MAC is an example of a *randomized MAC*. The same message may map onto several different MACs due to the use of the nonce r.

6.5. **Collision Resistance.** HMAC is built on top of a collision-resistant hash function. First, we define the notion of collision-resistance. Consider a hash function H from $\mathcal{M} \to T$ where $\mathcal{M}$ is the message space and T is the tag space where $|T| \ll |\mathcal{M}|$. Then, a **collision** for H is a pair such that $m_0 \neq m_1 \in \mathcal{M}$ such that $H(m_0) = H(m_1)$.

Definition 6.7. A function H is **collision resistant** if for all explicit, efficient algorithms $\mathcal{A}$, the advantage

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

is negligible.

An example of this is SHA-256 which has an output size of 2^{256}. We consider an application of collision resistant hash functions.

Example 6.8. Consider a function S where $S(k,m) = \mathrm{AES}(k, H(m))$ which is a secure MAC if AES is a secure PRF and H is collision resistant. Note that since AES is a 128-bit cipher, we can apply AES in raw-CBC mode (note that raw-CBC is acceptable because the block has *fixed* length (always 2 blocks since $H(m) \in \{0,1\}^{256}$).

7. 2/1: Collision Resistance

7.1. **Collision Resistant Hash Functions.** One application of collision resistant hash functions is the distribution of software packages. Suppose we have software packages F_1, F_2, F_3 and in a public read-only space, we write $H(F_1), H(F_2), H(F_3)$. Then, when you download one of the packages, you can compute $H(F_1)$ to verify the integrity of the package F_1. Note that an attacker cannot change the contents of the file without being detected. Note that in this example, we do not need a key to verify integrity; it is *universally verifiable*, but we need a public read-only space to ensure security.

Now, we consider an attack on a collision resistant hash function. Given a hash function $H : \mathcal{M} \to \{0,1\}^n$ with $|\mathcal{M}| \gg 2^n$, our goal is to find a collision in time $2^{n/2}$.

Theorem 7.1 (Birthday Paradox). *Let $r_1, \ldots, r_n \in \{1, \ldots, B\}$ be i.i.d random variables. Then, for $n = 1.2\sqrt{B}$, we have*

$$\Pr[\exists i \neq j : r_i = r_j] \geq \frac{1}{2}$$

Proof. We shall prove this for the uniform distribution. Then,

$$\Pr[\exists i \neq j : r_i = r_j] = 1 - \Pr[\forall i \neq j : r_i \neq r_j]$$

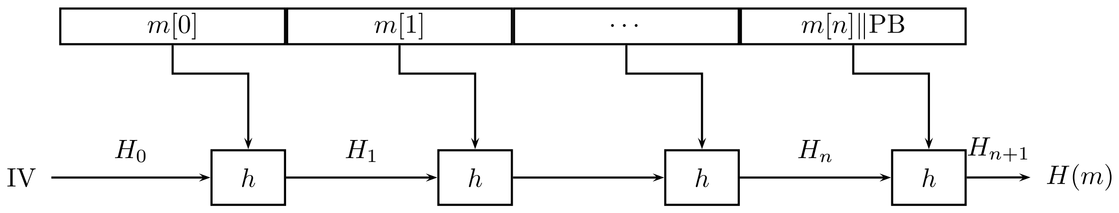
Consider the first choice: r_1. Clearly, it cannot collide with anything. For the second choice, r_2, the probability that it does not collide with anyone (r_1) will just be $\frac{B-1}{B}$. Extending this, we have

$$\begin{aligned}\Pr[\exists i \neq j : r_i = r_j] &= 1 - \frac{B-1}{B} \cdot \frac{B-2}{B} \cdots \frac{B-k+1}{B} \\ &= 1 - \prod_{i=1}^{n-1}\left(1 - \frac{i}{B}\right) \geq 1 - \prod_{i=1}^{n-1} e^{-i/B} = 1 - \exp\left(-\frac{1}{B}\sum_{i=1}^{n-1} i\right) \\ &= 1 - \exp\left(-\frac{1}{B} \cdot \frac{n(n-1)}{2}\right) \geq 1 - \exp\left(-\frac{1}{B} \cdot \frac{n^2}{2}\right)\end{aligned}$$

where we have used the fact that $1 - x \leq e^{-x} = 1 - x + \frac{x^2}{2} + \ldots$ (Taylor expansion). Taking $n = 1.2\sqrt{B}$, $\Pr[\exists i \neq j : r_i = r_j] \approx 0.53 > \frac{1}{2}$, as desired. □

Using the birthday paradox, we can construct a generic attack on hash functions. Basically, choose $\sqrt{2^n} = 2^{n/2}$ random messages $m_1, \ldots, m_{\sqrt{2^n}} \in \mathcal{M}$ with each m_i distinct. Then, for all i, compute $r_i \leftarrow H(m_i)$. Look for $i \neq j$ such that $r_i \neq r_j$. If no collision is found, repeat this process and otherwise output a collision. Thus, we can find a collision in roughly $O\left(\sqrt{2^n}\right)$ time. Thus, in order to be secure, a hash function requires fairly large output spaces (128 bit outputs is not secure!). Note that using a quantum computer, we can improve upon this to $O\left(\sqrt[3]{n}\right)$, so the key length needs to be three times as long as the desired security requirements. Standard hash functions include SHA-1 (160 bits), SHA-256 (256 bits), and SHA-512 (512 bits). SHA-1 is probably not secure (no collisions found yet, but there is an attack that runs in 2^{51} hash evaluations).

7.2. **Construction of Collision Resistant Hash Functions.** All hash functions use the **Merkle-Dangard (MD) construction**. Consider a message $m : [m[0], \ldots, m[n]]$. The Merkle-Damgard construction is then:



Given a fixed IV (hard coded into the standard), at each round, we send the IV through a compression function h. We assume that $h : T \times C \to T$ is a collision-resistant function. Each iteration (application of compression function to a message block), we obtain a chaining variable

$$H_i = \begin{cases} \mathrm{IV} & i = 0 \\ h(H_{i-1}, m[i-1]) & i > 0 \end{cases}$$

For the last block, we introduce a padding block (PB) and treat it as a single block. In particular, we take

$$\mathrm{PB} : 100 \cdots 0 \| \text{message length}$$

In particular, the compression function is a mapping $T \times X \to T$ and; by using this construction then, we obtain a hash function $H : X^{\leq L} \to T$. Now, we prove that if h is collision resistant, then H is also collision resistant:

Theorem 7.2 (Merkle-Dangard construction)**.** *If h is collision resistant, then H is collision resistant.*

Proof. We will show the contrapositive: if H is not collision resistant, h is not collision resistant. Suppose we have m, m' such that $H(m) = H(m')$. Let the chaining variables for each the message be

$$[\mathrm{IV}, H_0, H_1, \ldots, H_{t-1}, H_{t+1}] = H(m) = H(m') = \left[\mathrm{IV}, H'_0, H'_1, \ldots, H'_{r-1}, H'_{r+1}\right]$$

Now, by construction, we have

$$H(m) = H_{t+1} = h(H_t, m[t] \| \mathrm{PB}) = h(H'_r, m'[r] \| \mathrm{PB}') = H(m')$$

Now, if any $H_t \neq H'_r$, $m[t] \neq m'[r]$, or $\mathrm{PB} \neq \mathrm{PB}'$, then we have found a collision for h and we are done. Thus, consider the case where $H_t = H'_r$, $m[t] = m'[r]$, and $\mathrm{PB} = \mathrm{PB}'$. But since the padding blocks are equal, the messages must be of the same length so $t = r$. Now, we consider the second-to-last block:

$$h(H_{t-1}, m[t-1]) = H_t = H'_t = h(H'_{t-1}, m'[t-1])$$

Now, if $H_{t-1} \neq H'_{t-1}$ or $m[t-1] \neq m'[t-1]$, then we have found a collision for h and we are done. Otherwise, it must be the case that $H_{t-1} = H'_{t-1}$ and $m[t-1] = m'[t-1]$. Notice now that we can apply this argument iteratively to each block in succession to either find a collision for h or to show that $m[i] = m'[i]$ for each $i = 0, \ldots, t+1$. If we do not find a collision at any point in this process, then we have that all blocks of m and m' must be equal and hence $m = m'$ which is a contradiction of the fact that m, m' is a collision for H. Thus, it must be the case that at some point in this process, we must find a collision for h, as desired. □

The above theorem indicates that it is sufficient to construct a compression functions that are collision resistant to construct collision-resistant hash functions. Thus, our next goal is to build a collision-resistant compression function $h : T \times X \to T$ where T is the chaining variable and X is the message block. One method is to build the compression function for block ciphers. The first construction we consider is the **Davies-Mayer construction**. Take an encryption function $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ and take

$$h(H, m) = E(m, H) \oplus H$$

Note that m is the key in this block cipher and H is the data (chaining variable).

Theorem 7.3. *Suppose E is an ideal cipher (collection of $|\mathcal{K}|$ random permutation on $\{0,1\}^n$). Then, finding a collision takes $O\left(2^{n/2}\right)$ evaluations of E or D. Note that by the birthday paradox, $2^{n/2}$ is an upper bound for the time needed to find a collision. Thus, we cannot do any better.*

Example 7.4. The Davies-Mayer construction is very particular. Consider a simple modification $h(H, m) = E(m, H)$, which is *not* collision resistant! To see this, we construct a collision (H, m) and (H', m'). For any given (H, m, m') where $m' = h(H, m) = E(m, H)$, consider $H' = D(m', E(m, H))$ where D is the decryption function. Then, consider

$$h(H', m') = E(m', H') = E(m', D(m', E(m, H))) = E(m, H) = m'$$

which is a collision for h.

There are many other block-cipher constructions that are collision resistant. For instance, the Miyaguchi-Preneel construction is given as

$$h(H, m) = E(m, H) \oplus H \oplus m$$

which is used in Whirlpool. There are other variants similar to this such as $h(H, m) = E(H \oplus m, m) \oplus m$ which are secure as well as ones that are *not* secure $h(H, m) = E(m, H) \oplus m$.

SHA-256 is a MD construction using Davies-Mayer construction with a SHACAL-2 block cipher. The SHACAL-2 block cipher has a key size of 512 bits and a message size of 256 bits.

7.3. **Provably Secure Compression Functions.** There are also provably secure compression functions based upon difficult number theoretic problems. To do so, choose a random 2000-bit prime p and take two random values $1 \le u, v \le p$. Then,for $m, h \in \{0, \dots, p-1\}$, define

$$h(H, m) = u^H \cdot v^m \bmod p$$

Fact 7.5. *If we can find a collision for h, then we can compute the discrete log modulo p, which is a very difficult number theoretic problem.*

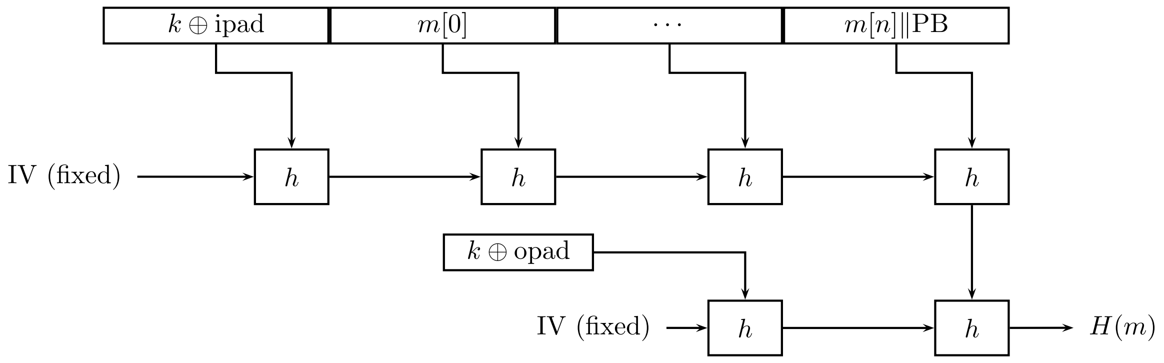
In practice, we do not use the above construct due to efficiency. Block-ciphers perform much better.

8. 2/6: HMAC And Timing Attacks

8.1. **HMAC.** Our goal is to leverage SHA-256 to construct a secure MAC. Consider a collision-resistant Merkle-Damgard hash function $H : X^{\le L} \to T$. Take $S(k, m) = H(k \| m)$. This scheme is insecure and vulnerable to an extension attack. In particular, given $H(k\|m)$, the adversary can compute $H(k\|m\|\text{PB}\|w)$ for any w where PB denotes the padding block. This is very straightforward, simply concatenate the padding block, followed by any block w. Then, evaluate the compression function h one more time to derive the new tag, which will yield the desired forgery. The proper way to convert a collision resistant hash function to a MAC is HMAC:

$$S(k, m) = H\left(\underbrace{k \oplus \text{opad}}_{512 \text{ bits}}, H\left(\underbrace{k \oplus \text{ipad}}_{512 \text{ bits}} \| m\right)\right)$$

where ipad denotes the *inner* pad and opad denotes the *outer* pad. A diagram of this is given below:



Both the ipad and the opad are embedded in the implementation of HMAC. HMAC is assumed to be a secure PRF (can be proven to be secure under certain PRF assumptions about $h(\cdot)$ with security bounds similar to NMAC: $\frac{q^2}{|T|}$ is negligible).

8.2. **Timing Attacks .**

Example 8.1. Consider the following function from the Keyczar crypto library (Python):

```
def Verify(key, msg, sig_bytes):
  return HMAC(key, msg) == sig_bytes
```

The == operator is implemented as a byte-to-byte comparison, which immediately admits an attack. In particular, to compute a tag for a target message m, we can

(1) Query server for a random tag.
(2) Loop over all possible first bytes and query server. Stop when verification takes a little longer than in step 1.
(3) Repeat for each byte in the tag until a valid tag has been found.

This is an example of a verification timing attack. Notice that though we have a security theorem, we can still construct an attack. The key difference in this case is the fact that we have additional information from a *side channel* (timing differences). There are several possible defenses to this attack. One possibility is to make all string comparators take the same amount of time:

```
return false if sig_bytes has wrong length
result = 0
for x,y in zip(HMAC(key,msg), sig_bytes):
  result |= ord(x) ^ ord(y)
return result == 0
```

This is difficult to implement properly due to compiler optimizations (hard to ensure that all operations take equal time). Note that adding a random amount of time will not work well. A better approach is

```
def Verify(key, msg, sig_bytes):
  mac = HMAC(key, msg)
  return HMAC(key, mac) == HMAC(key, sig_bytes)
```

Notice that the adversary no longer knows what bytes are currently being compared, and thus, the scheme is not vulnerable to the above attack.

8.3. **Authenticated Encryption.**

Our goal is to provide both CPA security (eavesdropping security for many-time keys) and ciphertext integrity (CI) . First, we play the ciphertext integrity game:

(1) The challenger chooses a random key $k \xleftarrow{R} \mathcal{K}$.
(2) The adversary can now subject chosen-plaintext queries. In particular, he submits a message m_i and receives the encryption $E(k, m_i)$.
(3) The adversary submits a ciphertext c.

The adversary wins if $D(k, c)$ is not rejected and $c \notin \{c_1, \dots, c_q\}$.

Definition 8.2. (E, D) has ciphertext integrity if for all efficient algorithms $\mathcal{A}$, $\mathcal{A}$ wins the ciphertext integrity game with negligible security.

Definition 8.3. (E, D) is authentic encryption if (E, D) is CPA-secure and has ciphertext integrity.

Note that in real life, *only* use authenticated encryption or integrity only methods. We now construct a few authenticated encryption schemes. First, we consider a scheme using a MAC and a cipher. In particular, let $I = (S, V)$ be a secure MAC and (E, D) be a CPA-secure cipher. In practice, we use *distinct* keys for encryption and MAC. There are various ways of combining encryption and MAC:

(1) MAC-then-Encrypt (SSL): First, we compute a tag $t \leftarrow S(k_m, m)$ for the message and then compute $c \leftarrow E(k_c, m\|t)$. There are artificial CPA-secure encryption schemes where this is not secure. However, if we use a block cipher in random-counter-mode, then this is secure even if the MAC is only one-time-secure (a one-time-MAC is a MAC that is secure if the attacker can make at most one query).
(2) Encrypt-then-MAC (IPSec): First, we encrypt the message $c \leftarrow E(k_c, m)$ and then $t \leftarrow S(k_m, c)$. The output is then $c\|t$.
(3) Encrypt-and-MAC (SSH). First, we compute the message $c \leftarrow E(k_m, m)$ and $t \leftarrow S(k_m, m)$ and output $c\|t$. This is not secure since MACs do not guarantee secrecy (take $\hat{S}(k, m) = m\|S(k, m)$). For deterministic MACs, this scheme will also not provide CPA security (tags for identical messages are identical).

Theorem 8.4. *For all CPA-secure encryption algorithms and secure MAC algorithms, using Encrypt-then-MAC will provide authenticated encryption.*

We consider a few examples of authenticated encryption.

8.4. **TLS.** We consider the TLS record protocol. In particular, suppose the browser and the server have a shared key $k = (k_{b \to s}, k_{s \to b})$. Note that use TLS uses *stateful* encryption (each side maintains two 64-bit counters, one for each direction: $\text{ctr}_{b \to s}$ and $\text{ctr}_{s \to b}$ in order to prevent replay attacks. The counters are initially 0 at setup and incremented for each message.) First, we consider 3DES+HAC+SHA196. Consider the browser to server communication. The TLS header consists of a header (type, version, length), followed by payload, and finally a padding. Everything except the header is encrypted as follows:

(1) Compute a MAC on the message $\text{ctr}_{b \to s} \| \text{header} \| \text{payload}$. Notice that $\text{ctr}_{b \to s}$ is *not* sent with the message, but because the connection is stateful, both sides know the counter. Note that this requires that the packets arrive in order (true for SSL over TCP/IP).
(2) Then, pad the packet to 3DES block length.
(3) Encrypt with $(\text{IV}_{b \to s}, k_{b \to s})$.
(4) Prepend the packet with the header.

There were various problems with the TLS standard. For example, if after decryption and the pad is invalid, then TLS sends a "decryption failed" message. However, if the pad is valid, but the MAC is invalid, then TLS sends a "bad record MAC" message. But this is vulnerable to a padding attack! The take-away is that when decryption fails, *never* explain why!

8.5. **Chosen Ciphertext Attacks .** Suppose an adversary has a certain ciphertext c that he wants to decrypt. In real life, it is often possible for an adversary to fool the recipient/server into decrypting certain ciphertexts, though not necessarily c. For instance, in the TCP/IP Protocol, the adversary can send a TCP/IP packet and the server will response with an ACK if a certain checksum is correct. By sending such repeated queries, the adversary can learn some information about the plaintext. Thus, we consider a notion of chosen ciphertext security. In particular, the adversary now have the power to obtain the encryption of any message of his choice (CPA) and decrypt any ciphertext of his choice that is not the challenge (CCA). The adversary's goal is again to break semantic security. We formalize the chosen ciphertext security model. Take a cipher $\mathbb{E} = (E, D)$ defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$. For $b = \{0, 1\}$, define $\text{Exp}(b)$ as follows:

(1) The challenger chooses a $k \xleftarrow{R} \mathcal{K}$
(2) For $i = 1, \ldots, q$, the adversary can submit either a CPA query or a CCA query:
    - CPA query: The adversary sends $(m_{i,0}, m_{i,1}) \in \mathcal{M}$ such that $|m_{i,0}| = |m_{i,1}|$ and receives $c_i \leftarrow E(k, m_{i,b})$
    - CCA query: The adversary submits $c_i \in \mathcal{C}$ where $c_i \notin \{c_1, \ldots, c_{i-1}\}$ and receives $m_i \leftarrow D(k, c_i)$
(3) The adversary then outputs $b' = \{0, 1\}$.

The cipher is CCA secure if for all efficient $\mathcal{A}$, the advantage

$$\text{Adv}_{\text{CCA}}[\mathcal{A}, \mathbb{E}] = |\Pr[\text{Exp}(0) = 1] - \Pr[\text{Exp}(1) = 1]|$$

is negligible.

Example 8.5. CBC with randomized IV is not CCA-secure. This is easy to see. Let the adversary submit two messages m_0 and m_1 with $|m_0| = |m_1| = 1$. The challenger replies with $c \leftarrow E(k, m_b) = \text{IV} \| c[0]$. The adversary can then submit a chosen ciphertext query $c' = (\text{IV} \oplus 1) \| c[0]$ and receive the decryption $D(k, c') = m_b \oplus 1$. Now, it is trivially easy for the adversary to determine the value of b, and thus win the semantic security game with advantage close to 1.

Theorem 8.6. *Let (E, D) be a cipher that provides authenticated encryption. Then, (E, D) is CCA secure. In particular, for any q-query efficient algorithm $\mathcal{A}$, there exists efficient algorithms $\mathcal{B}_1$ and $\mathcal{B}_2$ such that*

$$\text{Adv}_{\text{CCA}}[\mathcal{A}, E] \leq 2q \cdot \underbrace{\text{Adv}_{\text{CI}}[\mathcal{B}_1, E]}_{\textit{ciphertext integrity}} + \underbrace{\text{Adv}_{\text{CPA}}[\mathcal{B}_2, E]}_{\textit{CPA security}}$$

Proof. The CCA game is approximately equivalent to a game where the challenger rejects every ciphertext. Because the scheme provides ciphertext integrity, the challenger will not be able to forge a ciphertext that is accepted. But if the challenger always rejects, then the adversary does not gain any information through CCA queries. Thus, it is sufficient to simply eliminate CCA queries. Then, we are simply left with the original CPA game, which concludes the proof. □

9. 2/8: Authenticated Encryption and Key Management

9.1. **Authenticated Encryption.** Recall that the goal of authenticated encryption is to provide both confidentiality and integrity (attacker can forge a ciphertext that will be decrypted). Confidentiality without integrity is *worthless.* Consider IPsec, which provides packet-level encryption over the IP protocol using a shared key between the server and the browser. Upon receipt of the encrypted packet, the TCP/IP stack on the server decrypts the packet, determines the packet destination, and forwards the decrypted packet in the clear to the process on the specified port. Notice that if the packet is *not* authenticated and sent using AES in CBC mode with random IV, the attacker can simply change the IV so that the decrypted packet goes to the attacker's server instead (i.e. change the destination port on the server). In the first step of CBC decryption, we take $m_0 = D(k, c[0]) \oplus \text{IV}$, so my changing the IV, we can change the IV to modify m_0 as desired.

Example 9.1. Suppose we use the same key for both CBC-encryption and CBC-MAC. This may seem like a reasonable choice since we only have to perform one pass of CBC with the key. However, given additional blocks in the CBC-chain, CBC-MAC is no longer secure: we can alter a bit in one of the previous blocks of the CBC-chain without affecting the MAC. In that way, it become easy to forge a message.

There are three standards of authenticated encryption:

- CCM: CBC-MAC, then CTR mode encryption (used in 802.11i). Note that this is MAC-then-Encrypt, which is OK since we are using counter mode.
- GCM: CTR mode encryption then CW-MAC (special MAC that does not rely on AES and similar primitives)
- EAX: CTR mode encryption then CMAC

All of the above modes supported authenticated encryption with associated data (AEAD) where part of the message needs to be encrypted and MACed and another part of the message should only be MACed. For instance, the header (containing the destination address) of a packet should certainly not be encrypted, but should be authenticated, while the payload should be encrypted and authenticated.

In the modes of authenticated encryption that we have discussed so far, we generally need to make two passes through the data with the block cipher, once for encryption, and once for MAC. OCB is an example of a more efficient authenticated encryption which only requires one encryption operation per block and is highly parallelizable.

Theorem 9.2. *If E is a PRP, then OCB is an authenticated encryption scheme.*

Note that if a scheme is authenticated encryption

Example 9.3. In WEP, a message m is concatenated with a cyclic redundancy attack: $m \| \text{CRC}(m)$. This message is xored with $\text{RC4}(\text{IV}\|k)$ to compute the ciphertext. We then send $\text{IV}\|c$ where c is the ciphertext. Now, CRC is a linear function:

$$\text{CRC}(m \oplus b) = \text{CRC}(m) \oplus F(b)$$

for some known function F. Because, F is known, we can tamper with the ciphertext block and using the knowledge of F, compute a new checksum. Thus, it is very easy to tamper with and so WEP packets are very vulnerable to tampering. Thus, never use CRC for integrity!

9.2. **Key Management.** Up to this point, if there are n parties that need to communicate, the number of keys that would be required for them to communicate with each other will be $O(n^2)$ which is neither efficient nor scalable. The first improvement was the development of the key distribution center (KDC). In particular, each party has a shared secret key k_i with the KDC. Then, when two parties i and j want to communicate, they securely negotiate a shared key k_{ij} via the KDC (trusted). We consider one example protocol. Take two parties P and Q that want to set up a shared secret key. Further, suppose that each has an encryption key and a MAC key.

(1) P begins by sending a random nonce and some unique identifier id_p to q: r_p, id_p. The nonces prevents replay attacks.
(2) Q sends $r_p, r_q, \text{id}_p, \text{id}_q$ to the KDC where r_q is another random nonce and id_q is some unique identifier for Q.
(3) The KDC chooses a random key k and sends $c_q \leftarrow \text{Enc}_q(k)$ and $t_q \leftarrow \text{MAC}_q(\text{id}_p, \text{id}_q, r_p, r_q)$ to Q. Q then verifies the MAC and decrypts k.

(4) The KDC sends the ticket $c_p \leftarrow \mathrm{Enc}_p(k)$ and $t_p \leftarrow \mathrm{MAC}_p(\mathrm{id}_q, \mathrm{id}_p, r_p, r_q, c_p)$ to P. P then verifies the MAC and decrypts k.

Note that removing or altering any step in this protocol will compromise its security. The above protocol forms the basis of Kerberos. At the end, P and Q thinks that they are talking to each other. The KDC also knows the shared key K. The question, however, is whether this can be done without the KDC. This is possible and constitutes the basis of public key cryptography.

Example 9.4 (Merkle Puzzles)**.** Suppose Alice wants to communicate with Bob. She sets up a puzzle $\mathrm{Puzzle}_i = E\left(p_i, \mathrm{puzzle}\, X_i \| k_i\right)$ where p_i is a 40-bit key and k_i is a 128-bit key, which may be solvable by an exhaustive search. Alice then sends Bob $\mathrm{Puzzle}_1, \dots, \mathrm{Puzzle}_{2^{40}}$. Bob then picks a random puzzle Puzzle_j, solves it, and finds p_j to recover X_j and k_j. Bob then sends X_j to Alice in the clear. Alice then looks up puzzle with id X_j and recovers k_j. To break this scheme, the adversary would have to solve all puzzles, which would take time proportional to $\left(2^{40}\right)^2 = 2^{80}$. Thus, the adversary requires quadratic time to recover the key while Alice and Bob only require linear time to set up the channel. Note that a quadratic gap is the best you can do if you assume a "black box" cipher.

10. 2/13: Number theory

10.1. **Diffie-Hellman Example.** We consider first an example of a key exchange system secure against eavesdropping (no active attackers for now).

Example 10.1 (Diffie-Hellman)**.** Suppose Alice and Bob want to set up a shared key k. Take a fixed prime p and a value $g \in \{1, \dots, p-1\}$. Then Alice selects a value $a \xleftarrow{R} \{1, \dots, p-1\}$ as does Bob $b \xleftarrow{R} \{1, \dots, p-1\}$. Then, Alice sends $g^a \bmod p$ to Bob and Bob sends $g^b \bmod p$ to Alice. Then, Alice and Bob communicates

$$k = \left(g^b\right)^a \bmod p = g^{ab} \bmod p = \left(g^a\right)^b \quad \bmod p = k$$

Now the adversary sees the values of p, g, g^a, g^b and must compute $g^{ab} \bmod p$, which is a difficult problem in number theory.

10.2. **Arithmetic Modulo Primes.** We begin our discussion with arithmetic modulo primes. Take p to be a huge prime (on the order of 2048-bits). Then, let us denote $\mathbb{Z}_p = \{0, 1, \dots, p-1\}$ (standard notation in number theory is $\mathbb{Z}/p\mathbb{Z}$). Addition, multiplication in $\mathbb{Z}_p$ behave as we would expect (are commutative, associative, distributive).

Theorem 10.2 (Fermat's Little Theorem)**.** *For all $a \neq 0$ and prime p, we have $a^{p+1} = 1 \bmod p$.*

Example 10.3. Consider $3^4 \bmod 5$. Then

$$3^4 = 81 = 1 \bmod 5$$

Definition 10.4. We define the inverse of $x \in \mathbb{Z}_p$ is an element $a \in \mathbb{Z}_p$ such that $ax = 1 \in \mathbb{Z}_p$. Then, we denote this by x^{-1}.

Example 10.5. The inverse of 2 modulo any odd prime p is given by $\frac{p+1}{2}$ since

$$2 \cdot \frac{p+1}{2} = p + 1 = 1 \bmod p$$

Lemma 10.6. *If $x \neq 0 \in \mathbb{Z}_p$, then x is invertible.*

Proof. The inverse of $x^{-1} = x^{p-2} \in \mathbb{Z}_p$. We verify this

$$x \cdot x^{p-2} = x^{p-1} = 1 \bmod p$$

by Fermat's Theorem. Thus, $x^{p-2} = x^{-1}$, as desired. $\square$

Definition 10.7. $\mathbb{Z}_p^*$ denotes the set of invertible elements $\{1, \dots, p-1\} \subseteq \mathbb{Z}_p$.

Given this, we may now solve linear equations modulo p. For instance, given $ax + b = 0 \bmod p$, we have $x = -b \cdot a^{p-2} \in \mathbb{Z}_p$.

Now, we consider the structure of $\mathbb{Z}_p^*$.

Theorem 10.8 (Euler)**.** *$\mathbb{Z}_p^*$ is a cyclic group. In particular,*

$$\exists g \in \mathbb{Z}_p^* : \left\{1, g^1, \dots, g^{p-2}\right\} = \mathbb{Z}_p^*$$

In this case, g is a generator of $\mathbb{Z}_p^*$.

Example 10.9. We work modulo 7. Then

$$\langle 3\rangle = \{1,3,2,6,4,5\} = \mathbb{Z}_p^*$$

where $\langle g\rangle = \left\{1, g^1, \ldots, g^{p-2}\right\}$. Note that not every element of $\mathbb{Z}_p$ is a generator. In the above example, $\langle 2\rangle = \{1,2,4\}$. In general, $|\langle x\rangle| \mid p-1$.

Definition 10.10. The order of $g \in \mathbb{Z}_p^*$, denoted $\mathrm{ord}_p(g)$ is $|\langle g\rangle|$.

Fact 10.11. *For $g \in \mathbb{Z}_p^*$, we see that $g^{\mathrm{ord}_p(g)} = 1$. In particular, $\mathrm{ord}_p(g)$ is the smallest positive power x of g such that $g^x = 1$.*

Theorem 10.12 (Lagrange)**.** *For all $g \in \mathbb{Z}_p^*$, $\mathrm{ord}_p(g) \mid p-1$.*

Note that Fermat's theorem is just a corollary of this:

$$g^{p-1} = \left[g^{\mathrm{ord}_p(g)}\right]^{\frac{p-1}{\mathrm{ord}_p(g)}} = 1$$

10.3. **Quadratic Equations in** $\mathbb{Z}_p$**.** Our goal is to find x such that $x^2 - a = 0$. This is equivalent to asking whether a has a square root in $\mathbb{Z}_p$.

Definition 10.13. A square root of $x \in \mathbb{Z}_p$ is a number $y \in \mathbb{Z}_p$ such that $y^2 = x \bmod p$. If x has a root in $\mathbb{Z}_p$, then x is a quadratic residue.

Example 10.14. If we work in $\mathbb{Z}_7$, $\sqrt{2} = 3 \bmod 7$ and $\sqrt{3} \bmod 7$ does not exist.

If we have that $x^2 = y^2 \bmod p$, then $(x+y)(x-y) = 0 \bmod p$. Note that $x = y$ or $x = -y$ in this case (holds because p is prime). Thus, for $p \neq 2$ and $x \neq 0$, either x has no roots or it has two roots.

Theorem 10.15. *x is a quadratic residue if and only if $x^{\frac{p-1}{2}} = 1 \bmod p$*

Proof. If x is a quadratic residue, then there exists y such that $y^2 = x \bmod p$. Then,

$$x^{\frac{p-1}{2}} = \left(y^2\right)^{\frac{p-1}{2}} = y^{p-1} = 1 \bmod p$$

by Fermat's theorem. Now, for the reverse direction, suppose that $x^{\frac{p-1}{2}} = 1 \bmod p$. Take g to be a generator for $\mathbb{Z}_p^*$. Thus, there exists some α such that $x = g^\alpha$. Then,

$$x^{\frac{p-1}{2}} = g^{\alpha \cdot \frac{p-1}{2}} = 1$$

Since g is a generator, $\mathrm{ord}_p g = p-1$ and so $p-1 \mid \alpha \cdot \frac{p-1}{2}$. But this means that α must be even and so $\sqrt{x} = g^{\frac{\alpha}{2}}$ so x is a quadratic residue, as desired. ☐

Fact 10.16. *If $x \in \mathbb{Z}_p^*$, then $x^{\frac{p-1}{2}} = \sqrt{x^{p-1}} = \sqrt{1} \in \{1,-1\}$.*

The Legendre symbol of $x \in \mathbb{Z}_p$ is defined as

$$\left(\frac{x}{p}\right) = \begin{cases} +1 & x \neq 0 \text{ is a quadratic residue} \\ -1 & x \neq 0 \text{ is not a quadratic residue} \\ 0 & x = 0 \end{cases}$$

Thus, we see that

$$\left(\frac{x}{p}\right) = x^{\frac{p-1}{2}}$$

Then, the number of quadratic residues in $\mathbb{Z}_p^*$ is precisely $\frac{p-1}{2}$ (squaring is a two-to-one map).

Now, we consider the computation of square roots. In particular, if $p = 3 \bmod 4$, then it is very easy to compute square roots. In particular, if $x \in \mathbb{Z}_p^*$, $\sqrt{x} = x^{\frac{p+1}{4}} \in \mathbb{Z}_p$. To see this, consider,

$$\left[x^{\frac{p+1}{4}}\right]^2 = x^{\frac{p+1}{2}} = x \cdot x^{\frac{p-1}{2}} = x \in \mathbb{Z}_p$$

We do not know of any deterministic algorithms to compute square roots modulo p. There does exist a simple randomized algorithm with expected polynomial time run-time. The general idea is that we try to factor the polynomial $x^2 - a \in \mathbb{Z}_p$. First, consider

$$\gcd\left(x^2 - a, x^{\frac{p-1}{2}} - 1\right)$$

Now, the roots of $x^{\frac{p-1}{2}} - 1$ will be all the quadratic residues of p. However, $\gcd\left(x^2 - a, x^{\frac{p-1}{2}} - 1\right) = x^2 - a$, so this will not work. However, we can select a randomized $r \in \mathbb{Z}_p$ and compute

$$\gcd\left((x+r)^2 - a, (x+r)^{\frac{p-1}{2}} - 1\right)$$

The common roots are them $-\sqrt{a} - r$ and $\sqrt{a} - r$. If we get lucky, then $-\sqrt{a} - r$ will be a quadratic residue and $\sqrt{a} - r$ will not. In this case, the gcd evaluates to

$$\gcd\left((x+r)^2 - a, (x+r)^{\frac{p-1}{2}} - 1\right) = x - \sqrt{a} + r$$

so we can easily read off the value of $\sqrt{a}$. It is provable that we will get "lucky" for $\frac{1}{2}$ of the values of r, so this algorithm will run in expected polynomial time.

11. 2/15: Number Theory (Continued)

11.1. **Multi-precision Arithmetic.** We begin by noting some facts about multi-precision arithmetic. To represent a large number $x \in \mathbb{Z}$, we use a series of buckets. Addition then takes $O(\log p)$ time (require $\log p$ buckets (32-bit integers) to store the bits of p). A naive implementation of multiplication (taking each digit and multiplying by the other number) requires $O(\log^2 p)$ time. If we exchange some multiplications with additions, we can get $O(\log^{1.71} p)$ performance (Karat-2uba). The best known multiplication algorithm can run in $O(\log p \cdot \log\log p \cdot \log\log\log p)$, but this is not used in practice. In general, division modulo p may be done in $O(\log^2 p)$.

We can compute $g^x \bmod p$ in $O(\log_2 x \cdot T_m)$ time where T_m denotes the time needed for multiplication. To do so, we use repeated squaring. Let $x = x_n x_{n-1} \ldots x_1 x_0 \in \{0,1\}^{n+1}$. Then, to compute $g^x \bmod p$, we do

```
(1) Set $z \leftarrow 1$ and $y \leftarrow g$
(2) For $i = 0, 1, \ldots, n$
    (a) If $x_i = 1$, set $z \leftarrow z \cdot y \bmod p$
    (b) $y \leftarrow y^2 \bmod p$
(3) Output $z$
```

Note that squaring can not be faster than multiplication by more than a factor of 3. Otherwise, we can multiply any two numbers by computing

$$xy = \frac{(x+y)^2 - x^2 - y^2}{2}$$

Thus, using the naive multiplication algorithm, squaring will be $O(\log^2 p)$, and so exponentiation will be $O(\log^3 p)$

11.2. **Diffie-Hellman Protocol.** Recall the Diffie-Hellman Protocol discussed above. We now generalize this. Consider a cyclic group G and let g be a generator of G of order q. In other words, $g^q = 1 \in G$. Now, when Alice and Bob want to communicate, Alice and Bob selects a random number $a \xleftarrow{R} \{1, \ldots, q\}$, $b \xleftarrow{R} \{1, \ldots, q\}$. Then, Alice sends $g^a \in G$ to Bob and Bob sends $g^b \in G$ to Alice. Then both Alice and Bob can compute the key $\left(g^b\right)^a = g^{ab} = (g^a)^b \in G$. To an adversary Eve, she observes G, g, g^a, g^b. Her goal is to compute g^{ab}.

Definition 11.1. The **Diffie-Hellman** function $\mathrm{DH}_g : G \times G \to G$ is defined as

$$\mathrm{DH}_g(g^a, g^b) = g^{ab}$$

Definition 11.2. The **Computational Diffie-Hellman Assumption** states that for all polynomial time algorithms $\mathcal{A}$, we have that

$$\Pr\left[\mathcal{A}(X,Y) = \mathrm{DH}_g(X,Y)\right] \leq \varepsilon$$

for some negligible ε and for $g \xleftarrow{R} G$.

This is closely related to the discrete log function:

Definition 11.3. Given a finite group G and generator g, the **discrete log function** is given by

$$\mathrm{Dlog}_g(x) = [\text{smallest } c \geq 0 \text{ such that } g^c = x]$$

Lemma 11.4. *If* $\mathrm{Dlog}_g(\cdot)$ *is each, then* $\mathrm{DH}_g(X,Y)$ *is also easy.*

Proof. Note that

$$\mathrm{DH}_g(X,Y) = X^{\mathrm{Dlog}_g(Y)}$$

so if the discrete log is easy, then the computational Diffie-Hellman problem is also easy. □

Example 11.5. There are many groups where computing the discrete log is easy. For instance, consider over the group $G = \{0, 1, \dots, p-1\}$ with operation addition modulo p. In this case, the discrete-log operation is effectively equivalent to division.

For a 2048-bit prime p, the discrete log in $\mathbb{Z}_p^*$ is believed to be as hard as breaking AES-128. The best known algorithm for solving the discrete log problem is the General Number Field Sieve (GNFS) which runs in $e^{\sqrt[3]{\log p}}$ which is a sub-exponential time. Notice that an exhaustive search over k bits will require $O\left(e^{\ln\left(2^k\right)}\right)$ time, which is much slower.

Remark 11.6. The discrete log is easy whenever $p-1$ only has small factors. Additionally, for a general group, the best algorithm will run in $\sqrt{|G|} = e^{\frac{1}{2}\log p}$. In the case of elliptic curve groups, the best known algorithm is the generic algorithm.

To formally prove the security of Diffie-Hellman assumption, we would like to show that if Diffie-Hellman is easy, then the discrete log should also be easy. However, this is an open problem, though believed to be true.

11.3. **Public Key Cryptography.**

Definition 11.7. A public key encryption scheme (PKE) is a tuple (G, E, D) where G outputs $(\mathrm{pk}, \mathrm{sk})$ where pk denotes the private key and sk denotes the secret key. Then $E(\mathrm{pk}, m)$ outputs c and $D(\mathrm{sk}, c)$ outputs m. We require consistency: for all outputs $(\mathrm{pk}, \mathrm{sk})$ of G and all m,

$$D(\mathrm{sk}, E(\mathrm{pk}, m)) = m$$

We can now solve key exchange very easily using a PKE. We now define semantic security for public key cryptography. As usual, we define experiments Exp(0) and Exp(1). In particular,

1. The challenger begins by generating $(\mathrm{pk}, \mathrm{sk})$ using G and sends pk to the adversary.
2. The adversary then submits two messages $m_0 \neq m_1$ to the challenger.
3. The challenger responds with $E(\mathrm{pk}, m_b)$ and the adversary outputs $b' \in \{0, 1\}$ to indicate which message was encrypted.

Definition 11.8. (G, E, D) is semantically secure if for all efficient algorithms $\mathcal{A}$,

$$\mathrm{Adv}_{\mathrm{SS}}(\mathcal{A}, E) = |\Pr(\mathrm{Exp}(0) = 1) - \Pr(\mathrm{Exp}(1) = 1)| < \varepsilon$$

for some negligible ε.

Using the Diffie-Hellman protocol, we can construct a public key encryption system. In particular, take G to be a finite cyclic group of order g where $g \in G$. Let H be a hash function $G \to \{0,1\}^n$ such as SHA-256. Now, we define (G, E, D):

- G: Take $\alpha \xleftarrow{R} \{1, \dots, q\}$ and $h \leftarrow g^\alpha \in G$. Then, $\mathrm{pk} = h$ and $\mathrm{sk} = \alpha$.
- $E(\mathrm{pk}, m)$: To encrypt a message $m \in \{0,1\}^n$ using the public key, take $\beta \xleftarrow{R} \{1, \dots, q\}$ where $u \leftarrow g^\beta \in G$ and $v \leftarrow h^\beta \in G$. Then, compute
$$c \leftarrow \underbrace{H(u\|v)}_{\text{OTP key}} \oplus m$$
The output is then (u, c).
- $D(\mathrm{sk}, c)$: We compute $m \leftarrow H(u, u^\alpha) \oplus c = H(u, v) \oplus c$. This is the El Gamal cryptosystem.

12. 2/22: Public Key Cryptography and RSA

12.1. **El Gamal Cryptosystem.**
Consider the El Gamal cryptosystem from above. First, we define the decisional Diffie-Hellman (DDH) assumption. We consider Exp_b with a group G and generator g. Then,

1. Take $\alpha \xleftarrow{R} \{1, \dots, q\}$ and $\beta \xleftarrow{R} \{1, \dots, q\}$ and $A = g^\alpha$ and $B = g^\beta$.
2. If $b = 0$, the challenger outputs $C = g^{\alpha\beta}$ and if $b = 1$, the challenger outputs $C = g^r$ where $r \xleftarrow{R} \{1, \dots, q\}$.
3. The adversary sees (g, A, B, C) and must decide whether $b = 0$ or $b = 1$.

Note that the CDH assumption is at least as hard as DDH (if you can break CDH, then you can simply compute $g^{\alpha\beta}$ and break DDH). It is believed that DDH is also very hard.

Theorem 12.1. *For a suitable H, given the DDH assumption, El Gamal is semantically secure.*

Proof. Suppose there is an adversary $\mathcal{A}$ that breaks El Gamal. We show that this adversary can also break DDH. In particular, the adversary sees $(g, g^\alpha, g^\beta, g^\gamma) = (g, A, B, C)$. Consider an El Gamal system with $u \leftarrow g^\beta$ and $v \leftarrow g^\gamma$. We now consider the El Gamal semantic security challenge. The adversary provides two messages m_0 and m_1. The challenger picks $b \xleftarrow{R} \{0,1\}$. Now, consider $\left(g^\beta, H(g^\beta, g^\gamma) \oplus m_b\right)$. Now, if g^γ is random, then $H(g^\beta, g^\gamma)$ is also random and the adversary is effectively interacting with the OTP, and the adversary will output m_0 or m_1 with probability $\frac{1}{2}$. Now, if $g^\gamma = g^{\alpha\beta}$, this is a valid encryption and so the adversary $b' = b$ with probability $\frac{1}{2} + \text{constant}$. Thus, the adversary can guess b with some constant, non-negligible advantage. ☐

12.2. **RSA Trapdoor Permutation.** We begin with a brief review of arithmetic modulo composites. Let $N = pq$ where p, q are primes. Then, $\mathbb{Z}_n = \{0, 1, \dots, N-1\}$ and let $Z_N^* = \{\text{invertible elements in } \mathbb{Z}_N\}$. Then,

$$x \in \mathbb{Z}_n^* \iff \gcd(x, N) = 1$$

and

$$|\mathbb{Z}_n^*| = \varphi(N) = (p-1)(q-1)$$

where φ is the Euler totient function. Now, Euler's theorem states that

$$\forall x \in \mathbb{Z}_n^* : x^{\varphi(n)} = 1 \in \mathbb{Z}_n^*$$

Definition 12.2. A **trapdoor permutation** consists of three algorithms $\left(G, F, F^{-1}\right)$ where G outputs a private key pk and a secret key sk. In particular, pk defines a function $F(\text{pk}, \cdot) : X \to X$. Note that $F(\text{pk}, \cdot)$ evaluates the function at x and $F^{-1}(\text{sk}, y)$ inverts the function at y using sk. A secure trapdoor permutation is a function $F(\text{pk}, \cdot)$ that is one-way without the trapdoor sk.

Now, we define the RSA trapdoor permutation:

- G: Take $N = pq$ where N is 1024 bits and p, q are each 512 bits. Take e to be an encryption exponent where $\gcd(e, \varphi(N)) = 1$
- F: $\text{RSA}(M) = M^e \in \mathbb{Z}_n^*$ where $M \in \mathbb{Z}_n^*$
- F^{-1}: $\text{RSA}(M)^d = M^{ed} = M^{k\varphi(N)+1} = \left(M^{\varphi(N)}\right)^k \cdot M = M$
- The trapdoor in this case is the decryption exponent d where $e \cdot d = 1 \bmod \varphi(N)$

The (n, e, t, ε)-**RSA assumption** is thus: for all t-time algorithms $\mathcal{A}$,

$$\Pr\left[\mathcal{A}(N, e, x) = x^{1/e}(N) : p, q \xleftarrow{R} \text{n-bit primes}, N \leftarrow pq,\ x \xleftarrow{R} \mathbb{Z}_N^*\right] < \varepsilon$$

Note that textbook RSA is insecure. In the textbook RSA scheme, we let the public key be (N, e) and the private key be d. Then to encrypt a message, we take $C = M^e \bmod N$ and to decrypt, we take $C^d = M \bmod N$ for $M \in \mathbb{Z}_N^*$.

Example 12.3. We consider a simple attack on textbook RSA. Suppose a client is communicating with a server. The browser sends a `CLIENT HELLO` to the server and the server replies with a `SERVER HELLO` along with (N, e). Then, the browser selects a random session key K (64 bits long) and sends to the server $C = \text{RSA}(K)$. The server then decrypts C using d to obtain the shared session key. Now, consider the perspective of an eavesdropper. Here, the eavesdropper sees $C = K^e \bmod N$. Suppose $K = K_1 \cdot K_2$ where $K_1, K_2 < 2^{34}$ (occurs with probability ≈ 0.20). Then, $C/K_1^e = K_2^e \bmod N$. Now, the adversary can build a table $C/1^e, \dots C/2^{34e}$ which requires time 2^{34}. Now, for each $K_2 = 0, \dots, 2^{34}$, test if K_2^e is in the table (requires $\log 2^{34} = 34$ times). This is a standard meet-in-the-middle attack and requires $34 \cdot 2^{34} \approx 2^{40} \ll 2^{64}$ time.

To address this problem, we take an authenticated encryption scheme (E_s, D_s) and some hash function $H : \mathbb{Z}_n \to \mathcal{K}$ where $\mathcal{K}$ is the key space of (E_s, D_s). Then, we define the following:

- G: Generate RSA parameters $\text{pk} = (N, e)$ and $\text{sk} = (N, d)$
- $E(\text{pk}, m)$: Choose random $x \in \mathbb{Z}_n$. Take $u \leftarrow \text{RSA}(x) = x^e$ and $k \leftarrow H(x)$. Then, output $(u, E_s(k, m))$.
- $D(\text{sk}, (u, c))$: Output $D_s(H(\text{RSA}^{-1}(u)), c)$

The above scheme is the ISO standard RSA scheme, which is not actually used widely in practice. What is used in practice is the PKCS1 V1.5 scheme. Given a message m, we construct the following

$$\underbrace{02}_{\text{16 bits}} \|\text{random pad}\|\text{FF}\|m$$

and encrypt with RSA. In early implementations, the web server will respond with an error message if the first two bytes (16 bits) is not 02. By leaking this information, the scheme is no longer secure against a chosen CT attack!

Basically, the attacker now has an oracle that tells him if the first two bytes of the plaintext is "02." Then, to decrypt a given ciphertext C, the attacker does the following:

- Pick $r \in \mathbb{Z}_N$. Compute $C' = r^E \cdot c = (r \cdot \text{PKCS1}(M))^e$
- Send C' to the web server and check the response.

To fix the scheme, a new preprocessing function, OAEP, was introduced in PKCS1 V2.0.

13. 2/27: Digital Signatures

13.1. **One-Way Functions.**

Definition 13.1. A **one-way function** is a function $f : A \mapsto B$ where given any $x \in A$, there is an efficient algorithm to compute $f(x)$, but given $y \in \text{Im}(B)$, there is no efficient algorithm to find $x \in A$ such that $f(x) = y$. More formally, we say that f is (t, ε)-one-way, if for all efficient algorithms $\mathcal{A}$ running in time t,

$$\Pr\left[f\left(\mathcal{A}(f(x))\right) = f(x)\right] < \varepsilon$$

for all $x \in A$.

Example 13.2. A block cipher $E(k, m)$ may be used to construct a one-way function. More precisely, take $f^E(k) = E(k, m_0)$ for some fixed m_0.

Example 13.3. If p is a large prime and $g \in \mathbb{Z}_p^*$ is a generator, then $f^{\text{dlog}}(x) = g^x \bmod p$ is also an one-way function. This is the discrete-log one-way function. The fastest inversion algorithm for a general prime p requires time $\exp \sqrt[e]{\log p}$ which is a *sub-exponential* function in the number of bits $(\log p)$. Note that $e^{\log \log p}$ is polynomial while $e^{\log p}$ is exponential in $\log p$. Note also the following properties of f^{dlog}:

- $f^{\text{dlog}}(x + y) = f^{\text{dlog}}(x) \cdot f^{\text{dlog}}(y)$
- $f^{\text{dlog}}(ax) = \left[f^{\text{dlog}}(x)\right]^a$

This function is used in Diffie-Hellman key exchange protocol as well as El Gamal encryption. Note that the first property allows public key cryptography.

Example 13.4. Consider $N = pq$ with p, q prime. Take e such that $\gcd(e, \varphi(N)) = 1$. The RSA one-way function is then given as

$$f^{\text{RSA}}(x) = x^e \bmod N$$

which is permutation on $\mathbb{Z}_N$. To invert f^{RSA}, we effectively want to find the e^{th} root modulo N. This is equivalent to finding

$$\left\{ \begin{array}{l} e^{\text{th}} \text{ root } \bmod p \\ e^{\text{th}} \text{ root } \bmod q \end{array} \right\} + \text{CRT}$$

where CRT is the Chinese Remainder Theorem. The fastest inversion algorithm is then to factor N. The best known algorithm to factor N has complexity $\exp \sqrt[3]{\log N}$ which is again subexponential. The RSA one-way function satisfies the property that

$$f^{\text{RSA}}(xy) = f^{\text{RSA}}(x) \cdot f^{\text{RSA}}(y)$$

Note that this function has a trapdoor. In particular, given d such that $e \cdot d \bmod \varphi(N)$, we can invert f^{RSA} by computing $\left(y^d\right)^e = y \bmod N$. In other words, given e and d, it is easy to factor N.

13.2. **Digital Signatures.**

Now, we consider a **digital signature** scheme. Such a scheme should satisfy the following properties:

- The signature should convince anyone that the message is indeed from the sender.
- Only the sender can produce a convincing signature.

Suppose Alice wants to send a message to Bob. Then, Alice has a key pair $(\text{pk}_A, \text{sk}_A)$ and Bob has her public key pk_A. Then, when Alice wants to send a message m to Bob, she includes a signature $\sigma = \text{Sign}\left(\text{sk}_A, m\right)$. Then the verifier computes $\text{Verify}(\text{pk}_A, m, \sigma)$, which accepts if σ is a valid signature for m and rejects otherwise. There are many applications of digital signatures:

- Software updates. When your computer downloads a software update from Microsoft, the update will include a signature signed by Microsoft's private key. Then, before an update is installed, the computer verifies that the update is properly signed by Microsoft.

- Certificates. We want to solve the problem of verifying that pk_A really belongs to Alice. We introduce a certificate authority with has its own key pair $(\text{pk}_{\text{CA}}, \text{sk}_{\text{CA}})$. Then, to identify Alice, Alice provides her public key pk_A to the CA and obtains a signature $\sigma = \text{Sign}(\text{sk}_{\text{CA}}, \text{pk}_A)$. To verify Alice's identity, Bob can then compute $\text{Verify}_{\text{pk}_{\text{CA}}}(\text{pk}_A, \sigma)$. Thus, the CAs provide authentication for Alice's public key. This begs the question of who authenticates the CA's public key. In many cases, the browser has a hard-coded list of CA public keys. There are only a few CAs.

Note the parallel with MAC integrity schemes. Both systems convince the recipient that the message was successfully transmitted. However, note that MACs require a shared secret key for signing and verification. Digital signatures do not require any kind of secret key. Furthermore, the digital signature is also verifiable by a third party (no need to share secret keys). We now formally define a signature system:

Definition 13.5. A **signature system** is a triple consisting of three algorithms (G, S, V) where

- G: Generates a key pair (pk, sk). Required to be random or else everyone would have the same secret key.
- $S(\text{sk}, m)$: Produces a signature for message m: $S(\text{pk}, m) \to \sigma$.
- $V(\text{pk}, m, \sigma)$: Accepts if σ if a valid signature for m, otherwise rejects.

The signature scheme must satisfy the correctness property. For all $(\text{pk}, \text{sk}) \leftarrow G$ and all messages m,

$$\Pr\left[V(\text{pk}, m, S(\text{sk}, m)) = \text{accept}\right] = 1$$

Note that there are trivial scheme that satisfy these properties. For example, if V always accepts, the above condition is trivially satisfied. Thus, to have security, we need stronger conditions. As usual, we consider a security game:

(1) The challenger generates a public-secret key pair $(\text{pk}, \text{sk}) \leftarrow G$. The challenger sends pk to the adversary.
(2) The adversary can make signature queries. In particular, the adversary sends messages m_i to the challenger and receives $\sigma_i \leftarrow S(\text{sk}, m_i)$.
(3) The adversary outputs a forgery (m^*, σ^*). The adversary wins if $V(\text{pk}, m^*, \sigma^*) = \text{accept}$ and $m^* \neq m_i$ for all i. In other words, the adversary wins if he can forge a signature for a unseen message.

Definition 13.6. A signature scheme is (t, ε)-secure if for all algorithms $\mathcal{A}$ that runs in time at most t,

$$\Pr\left[\mathcal{A} \text{ wins}\right] \leq \varepsilon$$

This means that the signature scheme is existentially unforgeable under a chosen message attack.

13.3. **RSA Signatures.** We consider a naive RSA signature scheme:

- G: Generate $N = pq$ with p, q prime. Then, generate e, d such that $ed = 1 \bmod \varphi(N)$. The public key is (N, e) and the secret key is d.
- $S(\text{sk}, m)$: $S(\text{sk}, m) \to m^d \bmod N$
- $V(\text{pk}, m, \sigma)$: Accept if $\sigma^e = m$, reject otherwise.

If $\sigma = m^d$, then $\sigma^e = m^{ed} = m \bmod N$ since $ed = 1 \bmod \varphi(N)$. Notice though that using the homomorphic properties of RSA, we can easily generate a forgery. For example, given a signature σ on a message m, we can construct the forgery σ^a for message m^a. To fix this problem, we use a "hash-and-sign" approach. In particular, given a *collision-resistant* hash function $H : \{0,1\}^n \to \mathbb{Z}_N^*$, we define

- G: Generate $N = pq$ with $ed = 1 \bmod \varphi(N)$. Take $\text{pk} = (N, e, H)$ and $\text{sk} = d$.
- $S(\text{sk}, m) : S(\text{sk}, m) \to H(m)^d \bmod N$
- $V(\text{pk}, m, \sigma)$: Accept if $\sigma^e = H(m) \bmod N$ and reject otherwise.

Naturally, if the adversary can factor N or equivalently, compute $\varphi(N)$, the scheme is broken. Similarly, if the hash function is not collision-resistant, the adversary can again construct a simple existential forgery. For a given message m with signature σ, if the adversary can easily find a collision m', then he can just output (m', σ) as his forgery. Note that in the above, H is a *full-domain hash* and so the above scheme is referred to as RSA-FDH. To prove that this scheme is secure, we make use of the RSA assumption.

Definition 13.7. For all efficient algorithms $\mathcal{A}$ and fixed e,

$$\Pr\left[\mathcal{A}(N, y) = x : N \leftarrow G,\ x \xleftarrow{R} \mathbb{Z}_n^*,\ y = x^e \bmod N\right] \leq \varepsilon$$

This is the **RSA assumption**.

Theorem 13.8. *If the RSA assumption holds, then RSA-FDH is secure in the "random-oracle" model. The "random oracle" model is the assumption that the output H is indistinguishable from uniform in $\mathbb{Z}_n^*$.*

Proof. We outline a proof. Suppose that $\mathcal{A}$ wins the security game. Then, using $\mathcal{A}$, we can invert the RSA function, which would violate the RSA assumption (presumed to be true). □

13.4. **Digital Signatures in Practice.** In practice, we use PKCS v.1.5 mode 1 (public key cryptography standards). In this mode of operation, given a collision-resistant hash function H, we compute a digest $D(m)$ of a message m. Then, $S(\text{sk}, m) = D(m)^d \bmod N$. We have no proof of whether this scheme is secure or not.

14. 2/29: Digital Signatures (Continued)

14.1. **One-Time Signatures.** Recall that block ciphers $E(k, m)$ may also be used as one-way functions. Thus, we can use them to construct a digital signature scheme. To see this, we describe a general signature scheme using just a one-way function. Take a one-way function $H : X \to Y$ where $X = Y = \{0,1\}^{256}$. We will begin by constructing a one-time signature scheme (signature is existentially unforgeable under a 1-time chosen message attack).

- G: Choose random values $x_{0,0}, \ldots, x_{0,v-1}$ and $x_{1,0}, \ldots, x_{1,v-1}$ where $v = 256$. Then take $y_{i,j} \leftarrow H(x_{i,j}) \in Y$. Then, take the secret key to be $\{x_{i,j}\}$ and the public key to be $\{y_{i,j}\}$.
- $S(\text{sk}, m = m[0], \ldots, m[v] \in \{0,1\}^v)$: The signature is given by
$$\sigma = \left(x_{m[0],0}, x_{m[1],1}, \ldots, x_{m[v-1],v-1}\right)$$
- $V(\text{pk}, m, \sigma)$: Accept only if $H(\sigma[i]) = y_{m[i],i}$ for all $i = 0, \ldots, v-1$.

Theorem 14.1. *The above signature scheme is one-time secure.*

Clearly, the above scheme is not two-time secure. Using just two messages 0^v and 1^v, we can recover the entire secret key and the adversary can now forge the signature for any message. The length of the above signature is $v \cdot \log_2 |X|$. We consider some applications of one-time signatures:

- Online/offline signatures. First, in the offline setting, we use RSA to sign the one-time signature key. In the online setting, we sign using the one-time signature. Then, the signature on the message comprises of both the RSA signature as well as the one-time signature. This way, the bulk of the work may be done offline (RSA signing).
- Singing video streams. Consider an authenticated video stream. To authenticate, we can sign the entire video with RSA, but then the recipient would have to download the full video in order to verify integrity, which is inefficient. Alternatively, we can sign each individual packet with RSA, but that is very slow and impractical. To fix this problem, we can sign the first packet using RSA, and then use one-time signatures on each of the subsequent packets (each message includes the public key for the next packet).

Now, we show that it is possible to construct a many-time signature using a one-time signature. For a two-time, *stateful* signature scheme, we can just generate two public keys $(\text{pk}_1, \text{pk}_2)$. The first time, we sign with pk_1 and the second time we sign with pk_2. To construct a many-time signature scheme, we use a tree-based construction based on a one-time signature scheme. Given a published key pk_1, we use it to sign $\text{pk}_2 \| \text{pk}_3$. Using pk_2, we sign public keys $\text{pk}_4 \| \text{pk}_5$. Similarly, pk_i signs $\text{pk}_{2i} \| \text{pk}_{2i+1}$. Then, for a message m_i, we sign the message with every signature along the path from pk_i to the root. For a tree with depth n, this scheme can sign 2^n messages. Note further that as we sign more messages, we can also grow the depth of the tree accordingly. The signatures generated by this scheme tend to be long. Note that to verify, Bob verifies each signature along the path, and uses pk_0 to verify the root.

14.2. **RSA.** Recall the RSA encryption scheme where we have public keys $N = pq$ and e with $\gcd(e, \varphi(n)) = 1$. To invert, we use the RSA trapdoor: $d \cdot e = 1 \bmod \varphi(n)$. Then, $\text{RSA}(x) = x^e \bmod N$. The problem of inverting the RSA functions is essentially equivalent to the problem of computing the e^{th} root of a number modulo N. In particular, given $N, e, y = x^e \bmod N$, we want to compute $\sqrt[e]{y}$. Currently, we do not know how hard this problem is. The best known algorithm to invert the RSA algorithm leverages the Chinese Remainder Theorem.

(1) Factor $N = pq$.
(2) Find $\sqrt[e]{y} \bmod p$ and $\sqrt[e]{y} \bmod q$. We can do this easily by evaluating $y^{d_p} \bmod q$ and $y^{d_q} \bmod q$ where $d_p \cdot e = 1 \bmod \varphi(p)$ and $d_q \cdot e = 1 \bmod \varphi(q)$. This is the same problem as RSA decryption! Now, $\varphi(p) = p - 1$ and $\varphi(q) = q - 1$ since p and q are prime. Thus, $d_p = e^{-1} \bmod p - 1$ and $d_q = e^{-1} \bmod q - 1$.
(3) Use the Chinese Remainder Theorem to compute $x = y^d = \sqrt[e]{y} \bmod N$.

Note that because the exponents are half the size as are the modulos, these computations may be done 8 times faster. Since we now perform this operation twice, the overall computation is 4 times faster. This is known as

RSA-CRT decryption and is widely used. An open problem is whether computing e^{th} roots of a number equally hard as factoring.

Consider what happens when the value of d in RSA is small. For instance, consider $d \sim 2^{128}$, which would lead to a 16x speedup (compared to normal $d \sim 2^{1024}$) in RSA decryption. However, it has been shown that if $d < N^{1/4}$, it is easy to break RSA. The best known bound is given by $d < N^{0.292}$. We consider the first result. Given (N, e) and the fact that $e^{-1} \bmod \varphi(N) < N^{1/4}$, our goal is to factor N. Given that there is a d such that $e \cdot d = 1 \bmod \varphi(n)$, we know there there exists some $k \in \mathbb{Z}$ such that

$$e \cdot d = k \cdot \varphi(n) + 1 \implies \left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| \leq \frac{1}{d \cdot \varphi(N)}$$

Note that the above also holds with equality. Also, note that $\varphi(N) = (p-1)(q-1) = N - p - q + 1$ so $|\varphi(N) - N| \leq 3\sqrt{N}$. Then, if $d < \frac{N^{1/4}}{3}$, we have that

$$\left| \frac{e}{N} - \frac{k}{d} \right| \leq \frac{1}{2d^2}$$

Now, $\frac{e}{N}$ is known. By the continued fraction theorem, we can recover $\frac{k}{d}$ using the continued fraction approximation to $\frac{e}{N}$. Given $\frac{k}{d}$, we can now recover d. Thus, the private exponent d in RSA cannot be small. However, the public exponent e can be small (taken to be 3 in many cases, and more commonly, $65537 = 2^{16} + 1$ - evaluating RSA requires 17 multiplications in this case) . Therefore, evaluation of RSA is fast, but inversion is slow. In the context of signature schemes, this means that signing is slow, but verification is fast.

To achieve comparable security to block ciphers, the RSA modulus needs to be very large. For instance, to achieve the same security as an 80-bit block cipher, we would need a 1024-bit RSA modulus.

15. 3/2: Number Theory (Composites)

15.1. **Structure of $\mathbb{Z}_N^*$.** First, we give the **extended Euclidean algorithm**. In particular, we rely on the fact $\gcd(a, b) = \gcd(b, a \bmod b)$. By the division algorithm, we have that $a = bq + r$ for some $q, r \in \mathbb{Z}$. Then, we have $r = a \bmod b$ and take $d = \gcd(a, b)$. Now, if $d|a$ and $d|b$, we must have $d|r$, so the construction is complete. We can iterative apply this until we have $\gcd(x', 0) = x'$.

Theorem 15.1 (Bezout). *For any $a, b, c \in \mathbb{Z}$ there exists a solution to $ax + by = c$ if and only if* $\gcd(a, b)|c$.

Proof. Consider the forward direction. We have that $\gcd(a, b)|a$ and $\gcd(a, b)|b$ by definition. Thus, $\gcd(a, b)|c$, as desired. For the reverse direction, we note that it is sufficient to solve $ax + by = \gcd(a, b) = d$. Then, $d|c$ so $c = kd$ for some $k \in \mathbb{Z}$. We can solve this linear system using the extended Euclidean algorithm and trace through the steps in reverse. □

The above theorem demonstrates that $a^{-1} \bmod N$ exists if and only if $\gcd(a, N) = 1$. We can compute a^{-1} by applying the extended Euclidean algorithm. Then, $\mathbb{Z}_N^*$ forms a group under multiplication (inverses exist, other group properties easily checked). Note that $\mathbb{Z}_n$ is not a group since multiplicative inverses do not necessarily exist. In particular, $|\mathbb{Z}_N^*| = \varphi(N)$ and $a^{\varphi(N)} = 1 \bmod N$ for all $\gcd(a, N) = 1$. This is effectively a generalized version of Fermat's theorem ($\varphi(p) = p - 1$ for prime p). Furthermore, note that the order of any subgroup of G must divide $\text{ord}(G)$.

The problem of computing e^{th} roots $\bmod N$ is the problem of computing $a^{1/e}$. In particular, this is equivalent to computing $e^{-1} \in \mathbb{Z}_{\varphi(N)}$. This is only possible if $\gcd(e, \varphi(N)) = 1$. Notice that the exponents constitute a group $\mathbb{Z}_{\varphi(N)}$. Now, we consider some properties of the Euler totient function:

- If p is prime, then $\varphi(p) = p - 1$ and $\varphi(p^2) = p^2 - p = p(p-1) = p^2 \left(1 - \frac{1}{p}\right)$.
- For a prime p, $\varphi(p^\alpha) = p^\alpha - p^{\alpha-1} = p^\alpha \left(1 - \frac{1}{p}\right)$.
- Now, for $N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_\ell^{\alpha_\ell}$ for primes $p_1, \ldots, p_\ell$, then

$$\varphi(N) = N \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_\ell}\right)$$

Using the last fact, if $N = pq$ for primes p and q, then $\varphi(N) = (p-1)(q-1) = N - p - q + 1 \approx N$ for very large N. This is used in RSA.

15.2. **Chinese Remainder Theorem.** Given $n_1, \ldots, n_\ell$ such that $\gcd(n_i, n_j) = 1$ for $i \neq j$. The goal is to find $x \in \mathbb{Z}$ such that $x = a_1 \bmod n_1, \ldots, x = a_\ell \bmod n_\ell$ for given $a_1, \ldots, a_\ell$. The Chinese Remainder Theorem then states that for any two solutions x, y, it must be the case that $x = y \bmod N$ for $N = a_1 a_2 \cdots a_\ell$.

We consider the RSA cryptosystem. Here, $n_1 = p$ and $n_2 = q$ with $pq = N$. Consider the system

$$x = a \bmod p = b \mod q$$

with $x \in \mathbb{Z}_N$. We want to find $\hat{e}_p$ and $\hat{e}_q \in \mathbb{Z}_n$ where

$$\begin{aligned} \hat{e}_p &= 1 \bmod p & \hat{e}_p &= 0 \bmod q \\ \hat{e}_q &= 0 \bmod p & \hat{e}_q &= 1 \bmod q \end{aligned}$$

Then, the solution to the above system is given by

$$x = a\hat{e}_p + b\hat{e}_q \in \mathbb{Z}_N$$

since $x \bmod p = a\hat{e}_p \bmod p = a \bmod p$ and similar mod$q$. Our goal is to find $\hat{e}_p$ and $\hat{e}_q$. Consider the equation

$$px + qy = 1$$

We can solve this equation for x and y using the extended Euclidean algorithm since p, q are coprime. Furthermore, from this equation, notice that $px = 0 \bmod p$ and $px = 1 \bmod q$. Thus, $\hat{e}_q$ is precisely px and similarly, $\hat{e}_p = qy$. Thus, by caching the values of $\hat{e}_p$ and $\hat{e}_q$, we can compute x very easily by forming $a\hat{e}_p + b\hat{e}_q$. Using these observations, we are able to speed up RSA inversion (decryption).

16. 3/5: Authenticated Key Exchange

16.1. **Public Key Management.** The question we want to address is the process by which Bob obtains Alice's public key. This is necessary for Bob to encrypt a message to Alice and for Bob to verify Alice's signatures. We begin by discussing a single-domain certificate authority (CA). In this scheme, when Alice wants to communicate with Bob, she sends her public key to the CA. The CA then verifies Alice's identity and issues her a certificate signed by the CA. Alice then forwards the certificate to Bob. Bob verifies the certificate using the public key of the CA (hard-coded into the browser). Certificates generally have the following format (X509 v3):

- Subject (entity being authenticated): (X500 Object) Name, address, organization name
- Public key: Public key of the subject
- CA: (X500 Object) Name, address, organization name
- Validity Dates
- CA's digital signature on certificate

To reiterate, a certificate binds a public key to its owner. We compare and contrast the KDC model (does not require public-key cryptography) and the CA model (relies upon public key cryptography) for key exchange:

- The CA provides an offline key-exchange system (in contrast to KDCs which must be online). Alice contacts CA once and never has to again (once she has the certificate).
- The CA does not know the actual session keys (again in contrast to KDC which knows the session keys). However, the CA can do active attacks (impersonate sites by signing appropriate certificates).
- If the CA is compromised, only future sessions are compromised. If a KDC is compromised, the keys from the past and present may be recovered and past sessions may be recovered.
- Bob must have the CA public key in order to verify signatures from the CA.
- The CA is taken to be a trusted party.

16.2. **Cross-Domain Certification.** Suppose Alice and Bob are under different authorities CA$_1$ and CA$_2$ (Alice only knows the public key of CA$_1$ and Bob only knows the public key of CA$_2$). If Alice and Bob want to communicate, then we can use a hierarchical setup, in which we introduce CA$_3$ that verifies CA$_1$ and CA$_2$. Then, when Alice communicates with Bob, she presents a certificate containing CA$_1$'s signature on Alice's public key as well as CA$_3$'s signature on CA$_1$'s certificate. In this scheme, CA$_3$ is a root certificate and CA$_1$ and CA$_2$ are intermediate CAs.

Another solution is the web-of-trust. In this case, suppose there are users u_1, u_2, u_3 where $(A, u_1), (u_1, u_2), (u_2, u_3), (u_3, B)$ are pairs of people that mutually trust each other. Then, Alice's public key to Bob contains u_1's certificate on A's public key, u_2's certificate on u_1's public key, and u_3's certificate on u_2's public key. This is not widely deployed since trust relations do not tend to be transitive. B may trust u_3 and u_3 may trust u_2, but B does not necessarily truest u_2.

16.3. **Certificate Revocation.** When a CA is compromised, it is necessary to revoke the certificates issued by the CA. There are several ways to address this:

- Wait until the certificate expires.
- Certificate Revocation List (CRL): List of certificate serial numbers that have been revoked and signed by the CA. The primary problem with this scheme is that it is slow. Furthermore, browsers have to periodically (i.e. once per week) check for certificate revocation.
- Online Certificate Status Protocol (OCSP): Browser sends certificate serial number to OCSP responder. The OCSP sends back a signed response specifying whether the certificate is valid or not. The response is cached for 4-7 days. Note that if the browser cannot connect to the OCSP, it will fail *open* (the connection is allowed). By blocking access to the OCSP, the browser cannot tell whether the certificate has been revoked or not.

17. 3/7: SSL/TLS

17.1. **Authenticated Key Exchange.** There are two goals in TLS:

- One-sided authentication: The client/browser knows who he is talking to (via certificate on server side).
- Mutual authentication: Both the browser and the server know who they are talking to. In particular, this requires both a certificate on the client-side and on the server side.

The wrong approach to key exchange is to start with anonymous key exchange and follow with authentication (for instance, anonymous Diffie-Hellman). This scheme is easily defeated via a man-in-the-middle attack (attacker can set up secure connection with browser and with the server). To make the scheme non-anonymous, we include a certificate. We now describe the TLS protocol:

1. Browser begins by sending the client hello along with a session ID, client randomness (28 bytes), and a set of supported cipher-specs.
2. Server replies with server hello which consists of the session ID, server randomness (28 bytes), the selected cipher-spec (there must exist at least one functional one), and a server certificate.
3. The browser validates the server's certificate. The browser and server go through a key-exchange algorithm to derive a 48-byte premaster secret (PMS).
4. Using a key derivation algorithm, derive session keys ($K_{\text{b}\to\text{s,enc}}, K_{\text{b}\to\text{s,mac}}, K_{\text{s}\to\text{b,enc}}, K_{\text{b}\to\text{s,mac}}$) from the PMS, the client randomness, and the server randomness. Note that the client and server randomness are necessary to prevent replay attacks.
5. Browser and server then send "Finished" messages along with a hash of the messages sent. This ensures that tampering has not occurred (that both sides agree on the messages that has been sent).

Note that the first GET request should only happen after certificate validation! We now discuss the key-exchange algorithm. One possibility is to use RSA encryption (PKCS1 v1.5). The browser chooses a random 48-byte PMS to the server and the server decrypts using the browser's public key. However, this scheme does not provide **forward secrecy** (if adversary can obtain the PMS in the future, then he can obtain the session keys for the past session and decrypt the transmission). We consider a scheme that provides forward secrecy.

17.2. **Ephemeral Diffie-Hellman Protocol (EDH).** The following protocol provides forward secrecy:

1. The server has a large prime p and a generator g.
2. To initiate the key exchange, the server chooses a random a and compute $z_1 \leftarrow g^a \bmod p$ and signs (p, g, z_1) with his RSA key. The server sends p, g, z_1 along with the signature.
3. The browser verifies the signature. Then, he chooses a random b and computes $z_2 \leftarrow g^b \bmod p$. The browser then sends z_2 to the server.
4. The browser and server then agree on a PMS by computing $z_2^a = z_1^b$ (Diffie-Hellman).

Note that EDH is 3 times slower than plain RSA.

17.3. **TLS-Resume.** It is inefficient to re-negotiate a new TLS key for each request. The TLS-resume mechanism allows for reusing of the TLS key. In particular, the browser begins by sending the previous session's session ID. If the session ID is in the server's cache, then the previous key would be used; otherwise, a new key would be negotiated. This is much faster than renegotiating a key.

It is very difficult to integrate TLS into HTTP (HTTPS):

- Proxies: Browser would typically send GET request to proxy and proxy would forward to remote host. But if the request is encrypted, then the proxy cannot forward the request. To get around this, the browser would first send a connect request to the proxy. The proxy would then relay the SSL packets.
- Virtual hosting. Multiple websites can be hosted at the same IP address. Without HTTPS, this is resolved via the host header. But under HTTPS, this is encrypted! To resolve this, we can add a domain field to the client hello to specify which host is desired (IE6 does not support this!).
- SSL-Strip: In the SSL-Strip attack, the browser sends a request to the web server. The server responds with a 302 request to the https version of the page. The attacker strips out the "s" so the original requests are still sent over http. The attacker establishes an SSL connection to the browser, completing the man-in-the-middle attack. To the client, the only difference is that there is no lock on the browser. The solution to this is to use STS headers (on first connect, a page sets a cookie specifying that the browser must always use SSL to connect to the webpage). There is a cost since all pages on the website would require SSL in this case.

18. 3/12: Authentication Protocols

18.1. **Authentication Protocols.** In many cases, a user P (prover) wants to prove his identity to some server V (verifier). In many cases, we have an algorithm G that provides a secret key sk to the user and a verification key vk to a verifier. The verification key is sometimes public and sometimes private. Some applications include keyless entries to systems, ATMs, or login to a remote web site once key-exchange completes. Note an ID protocol alone *cannot* establish a secure session (not even when combined with anonymous key exchange!). As discussed previously, this is vulnerable to the man-in-the-middle attack.

There are three security models that we will consider:

- Direct attacker: impersonates prover with no additional information (other than vk) - example: door lock
- Eavesdropping attacker: Impersonates prover after eavesdropping on a few conversation 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

To protect against a direct attacker, we can use the basic password protocol. In particular, the algorithm G just selects a password $\mathrm{pw} \leftarrow \mathrm{PWD}$ where PWD is the set of passwords. Algorithm G then just outputs $\mathrm{sk} = \mathrm{vk} = \mathrm{pw}$. The problem in this scheme is that vk must be kept secret. Compromising the server will compromise all the passwords! Therefore, we should only store a one way hash of the password. In particular, the verifier V now computes $\mathrm{vk} = H(\mathrm{sk})$ where H is a secure hash function. In practice however, people often choose passwords from a small set (i.e. a dictionary). Thus, the password scheme is vulnerable to a dictionary attack. **Online dictionary attacks** can be defeated by doubling the response time after every failure, but are difficult to block when the attacker commands a bot-net. We can only consider **offline dictionary attacks** . In particular, suppose the attacker broke into the web server and obtained all the hashed passwords. Then, he can essentially hash all the words in the dictionary until a word w is found such that $H(w) = \mathrm{vk}$. This is a simple attack that may be done in time $O(|\mathcal{D}|)$ where $\mathcal{D}$ is the dictionary. Note that in this case, the attacker can perform a batch offline dictionary attack. In particular, if the attacker obtains a password file $\mathcal{F}$ with the hashed passwords for all users, he can then build a list L containing $(w, H(w))$ for all $w \in \mathcal{D}$. Then, to mount the attack, he just computes the intersection of L and $\mathcal{F}$. Using a total runtime of $O\left(|\mathcal{D}| + |\mathcal{F}|\right)$, the attacker is able to mount a dictionary attack on each password. By using a unique salt for each password, we hash the password along with the salt (can be stored in plaintext). Then, for each password, the attacker needs to mount a full dictionary attack, and the full runtime now becomes $O\left(|\mathcal{D}| \times |\mathcal{F}|\right)$. Another defense against the attack is to use a slow hash function H. One candidate is to use

$$H(\mathrm{pw}) = \mathrm{SHA1}\left(\mathrm{SHA1}\left(\cdots \mathrm{SHA1}(\mathrm{pw}) \cdots\right)\right)$$

so that the overall hash chain requires roughly 0.50 seconds to evaluate. In this case, there is a negligible impact to the user, but makes the dictionary attack much harder to mount in practice. Another possible defense is to use a secret salt (**pepper**). When setting the password, select a short random r. When verifying, simply try all values of r (on average, we would need to try 128 values for r), but for an attacker, there is a 256 time slow down (for a wrong password, need to try all values).

In Unix, a 12-bit public salt was used: they took the password and a salt and a DES key k and iterative applied DES 25 times (changed one of the bits in the DES tables in order to protect against hardware-accelerated attacks). In Windows, MD4 is used (no salts).

In many cases, we often use two-factor authentication (another method in addition to the password) for added security. One example is biometric signatures such as fingerprints, retina, or facial recognition. While these are hard to forget, they generally are not secret and cannot be changed (easily...). Thus, they are fine as a second factor authentication, but should *never* be used a primary factor authentication.

In many cases, users tend to use the same password at many sites. As a result, a break-in at a low security site compromises the security at a high security site. The standard solution is software that converts a common password pw into a unique password:

$$\text{pw}' = H\left(\text{pw}, \text{user-id}, \text{server-id}\right)$$

and pw′ is what is eventually sent to the server.

18.2. **ID Protocols Secure Against Eavesdropping Attacks.** In the eavesdropping security model, the adversary is given the vk and the transcript of several interaction between honest prover and verifier. The adversary's goal is to then impersonate the prover. A protocol is "secure against eavesdropping" if no efficient adversary can win this game. Clearly, the password protocol is insecure since once the attacker learns the password (from one valid exchange), the system has been compromised. We consider one-time passwords. For instance, in the SecurID system, the algorithm G selects a random key $k \leftarrow \mathcal{K}$. Then G outputs $\text{sk} = (k, 0) = \text{vk}$. Then, when the value wants to authenticate, he sends $r_n \leftarrow F(k, 0)$ where F is a secure PRF. The verifier then checks that $r = F(k, 0)$. Then, both prover and verifier increment k. Note that if an adversary was able to break the verifier, then they can impersonate any user (secure information is stored on the server).

Another system that provides security against eavesdropping is the S/Key system. Let $H^{(n)}(x)$ denote n iterative applications of H on x. Then algorithm G begins by selecting a random key $k \leftarrow \mathcal{K}$ and outputs $\text{sk} = (k, n)$ and $\text{vk} = H^{(n+1)}(k)$. For the first verification, the user presents $H^{(n)}(k)$ and the server verifies by computing $H(H^{(n)}(k)) = H^{(n+1)}(k)$. Then, the verifier remembers $H^{(n)}(k)$ for the next authentication (set $\text{sk} \leftarrow H(k)$). The prover then submits $H^{(n-1)}(k)$ for the next authentication. This process continues for at most n iterations (new hash chain needs to be computed in that case). Note that in this case, vk can be made public. However, we can only support a *limited* number of authentications using this scheme.

18.3. **ID Protocols Secure Against Active Attacks.** We now consider security against active attacks. In particular, the attacker can first interact with the prover by sending queries and receiving responses. Then, the adversary tries to impersonate the prover to the verifier. The defense comes in the form of a challenge-response protocol. In the MAC-based challenge response system, G issues a $k \leftarrow \mathcal{K}$ to both the prover ($\text{sk} = k$) and the verifier ($\text{vk} = k$). Then the verifier begins by issuing a random challenge $m \leftarrow \mathcal{M}$ and the prover response with $t \leftarrow S_{\text{MAC}}(k, m)$. Then, this protocol is secure against active attacks provided that $(S_{\text{MAC}}, V_{\text{MAC}})$ is a secure MAC. This method again requires a secret on the server (the MAC verification key).

To not rely on server-side secrets, we can use signature-based challenge response system. In this case, G outputs a secret key sk for the prover and a verification key vk for the server. Then, the server issues a challenge $m \leftarrow \mathcal{M}$ and the prover replies with a signature on m: $t \leftarrow \text{Sign}(k, m)$. Then, the protocol is secure against active attacks provided the signature scheme used is secure. However, this is not widely used in practice since *secure* digital signatures tend to be long (at least 20 bytes - otherwise, the attacker can do an exhaustive search over the possible signatures).

Index