CMSC456 Course Material

Published with the permission of Professor Jonathan Katz

Randomness

Pseudorandom Functions (PRFs)

$\text{Func}_n =$ all functions that map from $\{0,1\}^n$ to $\{0,1\}^n$

How large is this set of functions? For any bit in the input string, we can apply an Identity function, a NOT function ($\neg$), a ONE function (1), or a ZERO function (0). Therefore a function in this space is just a sequence of these smaller function components such that we accumulate n of them, one for each bit. This unique ordering and arrangement of the sub functions is what comprises the function table.

Input	Output
$0...000$	2^n possibilities
$0...001$	2^n possibilities
$0...010$	2^n possibilities
$0...100$	2^n possibilities
$...$	$...$

Input	Output
$1...111$	2^n possibilities

Because there are 2^n ways to configure an n bit string, and each of these can be transformed into any other n bit string, the size of the function space is as follows:

$|\mathrm{Func}_n| = (2^n)^{2^n} = 2^{n2^n}$

F is a pseudorandom function if F_k for uniform key $k \in \{0,1\}^n$ is indistinguishable from a uniform function $f \in \mathrm{Func}_n$.

In this mode of thinking, an attacker can probe however much they like, but they should not be able to distinguish whether they are given a box with a uniformly chosen key *or* an actual function f; both of which are chosen uniformly and at random.

▼ Pseudorandom Permutations (PRPs)

Let $f \in \mathrm{Func}_n$. f is a PRP if it is a bijection, meaning that the inverse (f^{-1}) exists.

Let $\mathrm{Perm}_n \subset \mathrm{Func}_n$ be the set of permutations. What is $|\mathrm{Perm}_n|$?

For something to be a 'permutation', we disallow duplicate values in out output function. Therefore the number of possibilities for each entry in the list of all possible bit strings actually dwindles over time, leading to 2^n!possibilities rather than the much larger set of possibilities in the set of *all* functions.

▼ Pseudorandom Generators (PRGs)

Access to a PRF F immediately implies the access to a PRG G, simply because the only requirement for a PRG is that it is indistinguishable from a PRF. In practice though, we don't have usually use PRFs, only PRGs that are indistinguishable from them. The ontology of *true* PRGs is still a contested topic. If PRGs do actually exist, this would imply that $P \neq NP$. In practice we either simply *assume* that PRGs exist, or construct valid PRGs on the backs of weaker assumptions.

▼ Probabilities

▼ Events

An ***Event*** is a particular occurrence in some experiment. e.g., the event that random variable X takes values x.

$Pr[E]$: probability of event E

Conditional Probability

Probability that one event A occurs, given that some other event Boccurred.

$Pr[A|B] = \frac{Pr[A \wedge B]}{Pr[B]}$

Bayes Theorem

Relatedly, Bayes Theorem shows us how to rearrange such probabilities:

$Pr[A|B] = Pr[B|A] * \frac{Pr[A]}{Pr[B]}$

Independence

Two random variables X, Y are independent *iff*

for all $x, y : Pr[X = x|Y = y] = Pr[X = x]$

In other words, the probability of X having value x is the same, regardless of Y's value

Law of Total Probability

Say that $\mathrm{E}_1, ...\mathrm{E}_n$ are a partition of all possibilities. Then for any A:

$Pr[A] = \sum_i Pr[A \wedge \mathrm{E}_i] = \sum_i Pr[A|\mathrm{E}_i] * Pr[\mathrm{E}_i]$

Negligible Probabilities (negls)

$\mathrm{function}\ f : \mathbb{N} \to \mathbb{R}$ is **negligible** $\iff$ for every polynomial p there is an N such that for all $n < N$ it holds that $f(n) < \frac{1}{p(n)}$. Typically, we denote arbitrary negligible functions as 'negl's.

Let $\mathrm{negl}_1$and negl_2 be negligible functions. It follows that

- Defining $\mathrm{negl}_3 \leftarrow \mathrm{negl}_1 + \mathrm{negl}_2$ implies a third negligible function.
- Similarly, $\mathrm{negl}_4 \leftarrow p(n) * \mathrm{negl}_1$ also implies a negligible function for any Polynomial $p(n)$

▼ Group Theory and Related Math

▼ Abelian Groups

An Abelian Group is a set G and a binary operation $\circ$ defined on G such that the following properties hold.

- **Closure**: Given a pair of elements $g, h \in G$; $g \circ h$ is also in G
- **Identity**: There is an element $e \in G$ such that for all other $g \in G$, $e \circ g = g$.
- **Inverse**: Every element $g \in G$ has an inverse $h \in G$ such that $h \circ g = g \circ h = e$, meaning they produce the Identity.
- **Associativity** for all $f, g, h \in G$; $f \circ (g \circ h) = (f \circ g) \circ h$
- **Communativity** for all $g, h, \in G$; $g \circ h = h \circ g$

▼ Cyclic Groups

Let G be a finite group of order m and let g be some element of G. We can now define a cyclic group by using g as a "generator" and checking if it is able to populate G.

We write the generated group as $< g >= \{g^0, g^1, g^2, \dots, g^{m-1}\}$. We expect that g^m will simply produce g^0 since we are generating $\mod m$. Values *may* begin to loop before each power is represented. It's only when each power produces a distinct value that $< g >$ produces a set of the expected order, which then tells us that g is a generator and produces a cyclic group. Cyclic groups can have more than one group.

This operation can be performed both in groups under addition modulo N and groups under multiplication modulo N.

When an expression regarding a given group G is written $\log_g i$, we define g as the generator and i as the index within the group.

- Example 1: Given $\mathbb{Z}_{12}$, find $\log_5 5$:
  - $< 5 >= \{0, 5, 10, 3, 8, 1, 6, 11, 4, 9, 2, 7\} \therefore \log_5 5 = 1$
- Example: Given $\mathbb{Z}^*_{11}$, find $\log_8 9$:
  - $< 8 >= \{1, 8, 9, 6, 4, 10, 3, 2, 5, 7\} \therefore \log_8 9 = 2$

Relevant Theorems:

- Any group of *prime order* is cyclic, and every non-identity element is a generator
- If p is prime, then $\mathbb{Z}_p^*$ is cyclic, noting that the order of said group is $p-1$

Diffie-Hellman Problems

Based on our understanding of cyclic groups and associated notation, we describe two interesting problems. It can be demonstrated that a proof of one version's ease or difficulty extends to the other's.

Given a cyclic group G and a generator g, define:

$$\mathrm{DH}_g(h_1, h_2) = \mathrm{DH}_g(g^x, g^y) = g^{xy} = h_1^y = h_2^x$$

Computational Diffie-Hellman Problem (CDH):

Given $\mathrm{DH}_g(h_1, h_2)$, find the result

Decisional Diffie-Hellman Problem (DDH):

Given the result $\mathrm{DH}_g(g^x, g^y)$, find originating values

When selecting a group G to compute on, we generally search for groups that are hard to solve for in cryptographic contexts. For example, $\mathbb{Z}_N$ is easy for any N and any selected generator g. Therefore for our purposes we select *prime-order groups*. These have the added benefit that every element except for the identity is a generator!

Example:

Select a prime-order subgroup of $\mathbb{Z}_p^*$ where p is prime

- E.g. let $p = kq + 1$ for p, q prime
- So $\mathbb{Z}_p^*$ has order $p - 1 = kq$
- Take the subgroup of k^{th}powers ie
  - $G = \{[x^k \mod p] | x \in \mathbb{Z}_p^*\} \subset \mathbb{Z}_p^*$
  - G is a group

    - Can show that it has order $(p-1)/k = q$
    - Since q is prime, G must be cyclic

Group Order

The order of an Abelian Group is the number of elements contained within it. Note that this means each element tallied must conform to the properties of Abelian Groups.

- For groups where $\circ = + \mod N$, their order is simply $|\mathbb{Z}_N| = N$.
- For groups where $\circ = * \mod N$, it is less clear. For these, we denote their order as $|\mathbb{Z}_N^*| = \phi(N)$.
    - If N is prime, then we know all elements of $\mathbb{Z}_N^*$ are Invertible $\mod N$, therefore $\phi(N) = N - 1$.
    - If $N = pq$ for p, q distinct primes, then the invertible elements are the integers from 1 to $N - 1$ that are not multiples of p or q. Therefore $\phi(N) = (p-1)(q-1)$.
    - Note that assuming we have an efficient algorithm for Invertibility and access to all potential members of the group, we can simply traverse each element in the generated group and ask if it is Invertible, which is a required property of a true Abelian Group. Tallying the invertible elements reveals the true Order.

Group Isomorphisms

Let $\mathbb{G}, \mathbb{H}$ be groups with respect to the operations $\circ_{\mathbb{G}}, \circ_{\mathbb{H}}$, respectively. A function $f : \mathbb{G} \to \mathbb{H}$ is an isomorphism from $\mathbb{G}$ to $\mathbb{H}$ if:

- f is a bijection / permutation and
- For all $g_1, g_2 \in \mathbb{G}$ we have $f(g_1 \circ_{\mathbb{G}} g_2) = f(g_1) \circ_{\mathbb{H}} f(g_2)$

If there exists an isomorphism that satisfies these conditions we state that the two groups are isomorphic and write $\mathbb{G} \simeq \mathbb{H}$.

In essence, all we are doing is *renaming elements.*

Chinese Remainder Theorem

The Chinese Remainder Theorem states that for some $N = pq$ where $p, q > 1$ are relatively prime, we can conclude that

$$\begin{gathered}\mathbb{Z}_N \simeq \mathbb{Z}_p \times \mathbb{Z}_q \\ \text{and} \\ \mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*\end{gathered}$$

Moreover, let f be the function that maps elements $x \in \{0, \ldots, N-1\}$ to pairs (x_p, x_q) where $x_p \in \{0, \ldots, p-1\}$ and $x_q \in \{0, \ldots, q-1\}$ defined by

$$f(x) \overset{def}{=} ([x \mod p], [x \mod q])$$

By this definition, f serves as an isomorphism in both the cases of $\mathbb{Z}_N^*$ and $\mathbb{Z}_N$.

▼ Greatest Common Divisor (GCD)

Given two numbers $a, b \in \mathbb{Z}$ where $a \geq b$, we determine the largest value that evenly divides both a and b and denote it d. In the process of solving for d, we can also find the factors of a and b that produce d such that $d = Xa + Yb$. These values will always exist.

Euclidean Algorithm:

If a is 0, we know that we are done. Otherwise, we recurse, using $b \mod a$ in place of a and a in place of b. When a reaches 0, b will by definition be the GCD.

```
let EuclideanGCD (a: int) (b: int) =
  (* This is the base case *)
  if a == 0 then
    (b, 0, 1)
  (* Otherwise *)
  else
    (* Use the remainder to compute recursively *)
    let (d, X, Y) = EuclideanGCD (b % a) a in
    (* Rewrite X and Y *)
    (d, Y - floor(b / a) * X, X)
```

▼ Modular Invertibility

An integer b is Invertible modulo N if there exists an integer a such that $ab \mod N = 1$. Therefore to determine if integer b is Invertible, we simply need to find a value a which satisfies this condition. The resulting value a is written $a = [b^{-1} \mod N]$.

We can test for Invertibility succinctly by asserting that $[b^{-1} \mod N]$ is valid $\iff \text{GCD}(b, N) = 1$, meaning that b and N share no common factors.

▼ Division Mod N

If an expression is written $[c/b \mod N]$, it is only valid so long as we know b is Invertible $\mod N$. To evaluate, first solve $x = [b^{-1} \mod N]$ then plug into $[cx \mod N]$.

$$[c/b \mod N] \begin{cases} [c * [b^{-1} \mod N] \mod N] & \text{GCD}(b, N) = 1 \\ \text{Error} & \text{GCD}(b, N) \neq 1 \end{cases}$$

▼ Modular Exponentiation

If an expression is written $[a^b \mod N]$ for some base $a \in \mathbb{Z}_N$ and an integer exponent $b > 0$, we must solve it in a clever way, as we expect b to be a large value. By solving it recursively and applying a modulo N at each step of the function, we can keep the values we are working with comparatively small.

By Hand Algorithm:

Note that if we can solve $\phi(N)$ for the modulo N, the exponent b can be rewritten as $[b \mod \phi(N)]$, meaning we can rewrite our full expression as $[a^{[b \mod \phi(N)]} \mod N]$. If this produces an exponent that is easier to work with it can make the process way easier!

Naïve Algorithm:

```
let rec modExp_naive (a: int) (b: int) (n: int) =
  (* This is the base case *)
  if b == 0 then
    (* a^0 = 1 in all cases *)
    1
  else
    (* a * [a^(b-1) mod N] mod N *)
    (a * (modExp_naive a (b-1) n)) mod n
```

This code is based on the following recurrence:

$$[a^b \mod N] = a * a^{b-1} \mod N$$

Intelligent Algorithm:

```
let rec modExp_smart (a: int) (b: int) (n: int) =
  (* This is the base case *)
  if b == 0 then
    (* a^0 = 1 in all cases *)
    1
  else
    (* If b is even *)
    if b mod 2 == 0 then
      (* (a^{b/2})^2 mod N *)
      let sub = modExp_smart a (b / 2) n in
      (sub * sub) mod n
    else
      (* a * (a^{(b-1)/2})^2 mod N *)
      let sub = modExp_smart a ((b - 1) / 2) n in
      (a * sub * sub) mod n
```

This code is based on the following recurrence:

$$[a^b \mod N] = \begin{cases} (a^{\frac{b}{2}})^2 \mod N & b \mod 2 = 0 \\ a * (a^{\frac{b-1}{2}})^2 \mod N & b \mod 2 = 1 \end{cases}$$

▼ Security Metrics

▼ A Note on Evaluating Attackers

When running an attacker, the exact output for a given set of messages is not entirely relevant. Instead, we care about the probability that running the attacker A produces a 1 (successful attack) rather than a 0 (failed attack). For an attacker to be viable, we must showcase that this probability is greater than $Pr[\frac{1}{2}]$ (random) for selecting the bit b of the encoded message m_b, plus a negl. Formally, an attacker is successful *iff* $Pr[A = 1] > Pr[\frac{1}{2} + \text{negl}]$.

▼ Perfect Secrecy

For an encryption scheme to be 'perfectly secret', observing a given ciphertext should reveal no information at all about the plaintext. Formally, this means that the likelihood of guessing the correct plaintext is *unchanged* by observing the ciphertext. $P[M = m \mid C = c] = P[M = m]$

One major limitation of this definition is that it requires the key k for any given scheme attempting perfect secrecy to contain the same quantity of information as the message m, that is, to have the same length. $|k| = |m|$. To showcase that a scheme *does not* comply with perfect security, one simply needs to showcase that observing the ciphertext changes the likelihood with which you are able to guess the plaintext.

▼ Malleability

Malleability refers to a scheme's ability to be resistant to ciphertext modifications and the ability of adversaries to produce expected results in a decrypted plaintext. An example of this is the One-Time Pad, which is extremely malleable, allowing for attackers to make deterministic alterations to what will eventually become the plaintext, even if perfect secrecy is maintained.

▼ Integrity

Integrity refers to a scheme's ability to verify that a given ciphertext was actually sent by who we believe it was. Violating this principle only requires that an adversary gets a recipient to accept a message that was never actually sent by an honest party.

▼ EAV-Security

Highly related to Perfect Secrecy, EAV-Security specifies a standard of security for protecting against *Eavesdroppers* who are able to listen in on the communication channel and observe transmitted ciphertexts. To showcase that a scheme is *not* EAV-Secure, one needs only demonstrate that it is not perfectly secret across an arbitrary quantity of messages. For example, a single use One-Time Pad is perfectly EAV-Secure, but this security is broken if the key is ever reused.

▼ CPA-Security

By definition *stronger* in definition than EAV-Security, CPA-Security allows for the attacker to choose their own plaintexts, sending these plaintexts through the encryption algorithm and witnessing the ciphertexts they produce. We evaluate the security of the scheme by asking if this extra information gives the attacker an upper hand in decrypting the messages.

Attacker:

$\mathrm{PrivK}_{A,\Pi}^{CPA}(n):$

1. $k \leftarrow \mathrm{Gen}(1^n)$
2. $A(1^n)$ interacts with an *encryption oracle* $\mathrm{Enc}_k()$, outputting m_0, m_1 where $|m_0| = |m_1|$.
3. $b \leftarrow \{0,1\},\ c \leftarrow \mathrm{Enc}_k(m_b)$. give c to A
4. A can continue to interact with $\mathrm{Enc}_k()$
5. A outputs b'; A succeeds if $b = b$' and experiment evaluates to 1 in this case

▼ CCA-Security

By definition *stronger* in definition than both EAV-Security *and* CPA-Security, CCA-Security allows the attack to choose their own plaintexts, witness the resulting ciphertexts, and decrypt ciphertexts of their chosing, witnessing the resulting decodings. The one exception occurs with the challenge ciphertext being presented to the attacker A, as being allowed to decrypt it outright would defeat the point of the exercise.

Attacker:

$\mathrm{PrivK}_{A,\Pi}^{CCA}(n):$

1. $k \leftarrow \mathrm{Gen}(1^n)$
2. $A(1^n)$ interacts with an *encryption oracle* $\mathrm{Enc}_k()$ and *decryption oracle* $\mathrm{Dec}_k()$, outputting m_0, m_1 where $|m_0| = |m_1|$.
3. $b \leftarrow \{0,1\},\ c \leftarrow \mathrm{Enc}_k(m_b)$. give c to A.
4. A can continue to interact with $\mathrm{Enc}_k()$ and $\mathrm{Dec}_k()$

5. A outputs b'; A succeeds if $b = b$' and experiment evaluates to 1 in this case

▼ Unforgeability

We define unforgeability only in a formal context, one that is decided by the evaluation of this experiment. Essentially, we are checking if the attacker $\mathcal{A}$ is able to, given access to the *encryption oracle* Enc_k, create a new ciphertext that is successfully decrypted without error and that we never fed the *encryption oracle*.

$\text{Enc\_Forge}_{A,\Pi}(n)$:

1. $k \leftarrow \text{Gen}(1^n)$
2. $A(1^n)$ interacts with an *encryption oracle* $\text{Enc}_k()$, eventually outputting ciphertext c.
3. $m := \text{Dec}(c)$ and let Q denote the set of all queries that A submitted to the encryption oracle.
4. iff $m \neq \perp$ and $m \notin Q$ output 1, otherwise output 0.

A private-key encryption scheme Π is unforgeable if for all PPT adversaries A, there is a negligible function negl such that: $Pr[\text{Enc\_Forge}_{A,\Pi}(n) = 1] \leq \text{negl}(n).$

▼ Authenticated Encryption

A scheme is said to possess Authenticated Encryption (AE) if it posessses both CCA-Security and Unforgeability.

▼ Probabilistic Polynomial Time (PPT)

A scheme is *secure* in the context of PPT if an attacker succeeds in breaking the scheme with at most negligible probability. By manipulating the security parameter n, the scheme designer can alter the probability of breaking the scheme and the duration of computation required to achieve that probability. Because increasing n increases the compute time for encryption and decryption, the value is to be minimized while still protecting against potential threats. Approximately, we are looking for probabilities of 2^{-60}(ε) for cracking the scheme within 200 years (t).

- $\text{function } f : \mathbb{N} \to \mathbb{R} \iff \exists c \text{ where } f(n) < n^c \text{ for all } n$
- $\text{algorithm } A$: Runs in polynomial time $\iff$ there exists a polynomial p such that for all input $x \in \{0,1\}^*$ the computation of $A(x)$ terminates within at most $p(|x|)$ steps.

Cryptographic Utilities

Stream Ciphers

Linear Feedback Shift Registers (LFSRs)

Historically used for random number generation, LFSRs are comprised of an array of $n$single-bit registers $s_{n-1}, ..., s_0$, along with a set of n boolean *feedback coefficients* $c_{n-1}, ..., c_0$. The size n of the array is referred to as the *degree* of the LFSR. This specification makes LFSRs extremely hardware efficient. After each 'clock tick', the value of s_0 is output as y_i for the ith clock tick and the states of each register are updated, register s_{i+1} passing its value on to register s_i. The initial register s_{n-1} is overwritten using the *feedback coefficients*.

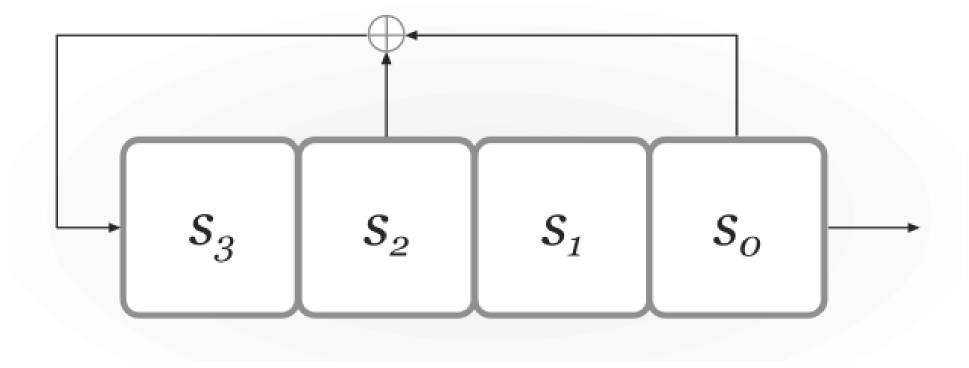
Note that the superscript t in s_i^t represents the clock tick. To describe the way each register updates on each clock tick, we write

$$\begin{cases} s_i^{(t+1)} = s_i^{(t)} & i = 0, \ldots, n-2 \\ s_{n-1}^{(t+1)} = \bigoplus_{i=0}^{n-1} c_i s_i^{(t)} \end{cases}$$

To denote the outputs of an LFSR as $y_0, y_1, \ldots,$ where $y_i = s_0^{(i)}$, we write

$$\begin{cases} y_i = s_i^{(0)} & i = 0, \ldots, n-1 \\ y_i = \bigoplus_{j=0}^{n-1} c_j y_{i-n+j} & i > n-1 \end{cases}$$

One way to conceptualize this is to realize the first n output bits *are* the initial state of the registers $s_0 \ldots s_{n-1}$.



An example of a simple LFSR

LFSRs can be used to construct stream ciphers of the form $(\text{Init}, \text{Next})$ in an organic way- give them an initial state and produce a new value by executing a single clock tick.

Note that a degree-n LFSR has at most 2^n possible states that correspond to the possible values of each register. A degree-n LFSR will eventually repeat a previous state. Once it does, it will then repeatedly cycle through this same set of states and their corresponding outputs.

An LFSR is said to have *maximum length* if it cycles through all 2^{n-1} nonzero states before repeating. Note that this is the ideal case.

Key-recovery attacks:

The output of a maximum-length LFSR has good properties, and typically far too many possible states to traverse. Despite this, LFSRs are not secure stream ciphers. If we assume the *feedback coefficients* are known to the attacker, then the first n bits of output from a degree-n LFSR will reveal the initial state of the LFSR, which functions as the key. Once known, all future output can be predicted. See formal definition above if confused.

Even if the *feedback coefficients* are hidden to an attacker, the initial state can still be deduced in only $2n$ output bits. As before, the first n output bits reveal the initial state. Given the next n output bits, we can construct a system of equations for which there is only one solution, revealing each *feedback coefficient*.

$$\begin{aligned} y_n &= c_{n-1}y_{n-1} \oplus \cdots \oplus c_0 y_0 \\ &\vdots \\ y_{2n-1} &= c_{n-1}y_{2n-2} \oplus \cdots \oplus c_0 y_{n-1} \end{aligned}$$

▼ Feedback Shift Registers (FSRs)

FSRs are the product of introducing Nonlinearity to LFSRs. This makes the systems less predictable, and more secure. There are multiple approaches for introducing this nonlinearity:

- **Nonlinear feedback**: The most apparent way to implement nonlinearity is to make the feedback loop nonlinear. Everything is identical to an LFSR, except that the new value of the leftmost register s_{n-1} is now determined nonlinearly.

  Given some arbitrary nonlinear function g that operates on the collection of registers, the FSR is able to generate a new state for s_{n-1}. For security, we posit that for g to be "balanced", $\Pr[g(s_{n-1}, \ldots, s_0) = 1] \approx 1/2$ where the probability is computed over uniformly selected states. In practice, we often see the use of ANDs and ORs.

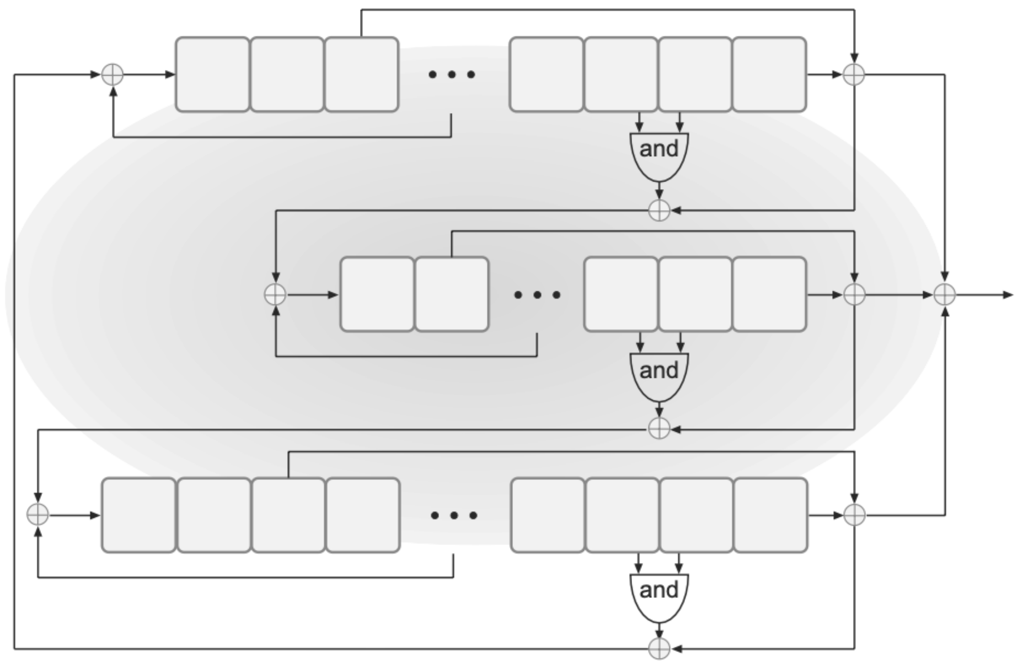
  Formally:

  $$\begin{cases} s_i^{(t+1)} := s_{i+1}^{(t)} & i = 0, \ldots, n-2 \\ s_{n-1}^{(t+1)} := g(s_{n-1}^{(t)}, \ldots, s_0^{(t)}) \end{cases}$$

- **Nonlinear output**: Rather than updating our new state nonlinearly, we can simply determine our output nonlinearly. Here, we denote the function that computes the output bit to be g and refer to it as the *filter*. As before, we expect it to be balanced to avoid obvious bias.

- **Combination generators**: By using more than one LFSR, we can generate the final output stream by combining the outputs of the individual LFSRs in some nonlinear way. The individual LFSRs do not have to be of the same degree, and their having this property is actually advantageous in achieving maximization. Care must be taken to ensure the combined output is balanced, as before.

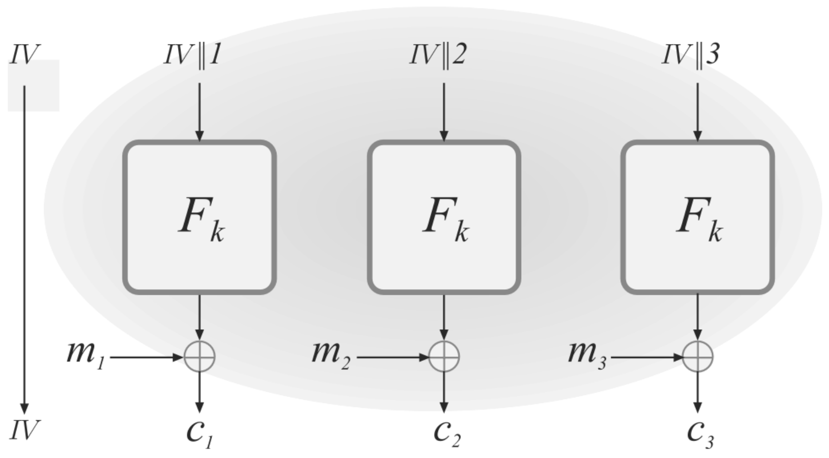
Trivium is the best example of an FSR used in practice as a stream cipher, and is an implementation of a combination generator. Within, we can see three couples, nonlinear FSRs that have decrees 93, 84, and 111 respectively. The state is the concatenation of the states of these sub-FSRs. In the Init function, Trivium is run for 4 * 288 clock ticks with discarded output to ensure its initial state cannot be deduced.



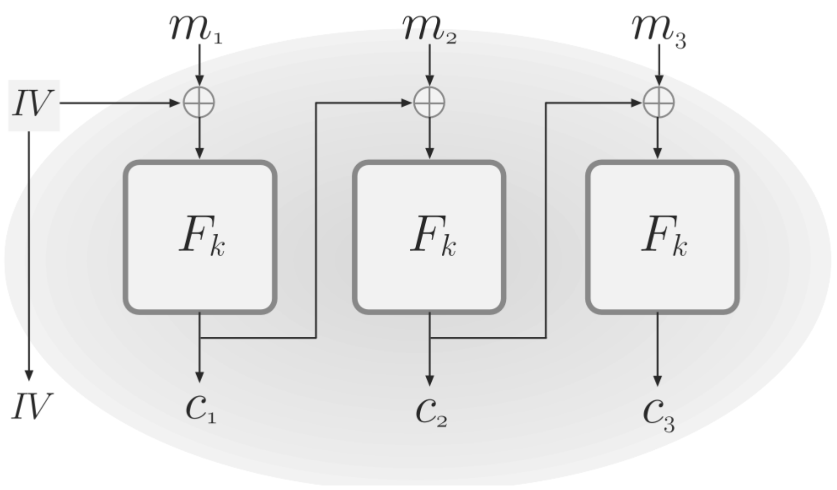
Block Ciphers

Block Ciphers Formal Definition

When a block scheme is in CTR Mode (Counter), XORing takes place *after* encryption via the PRF. The outputs of each block are independent from one another. Thus, modifiying the m_n block of a message m will not modify any blocks of the ciphertext seldom c_n.



By contrast when in CBC Mode(Cipher Block Chaining), XORing takes place *prior* to encryption via the PRF. The outputs of each block are XORed with the input of the subsequent block, meaning that modification to m_n will modify all subsequent blocks m_x where $x > n$.



Both of these are CPA-Secure.

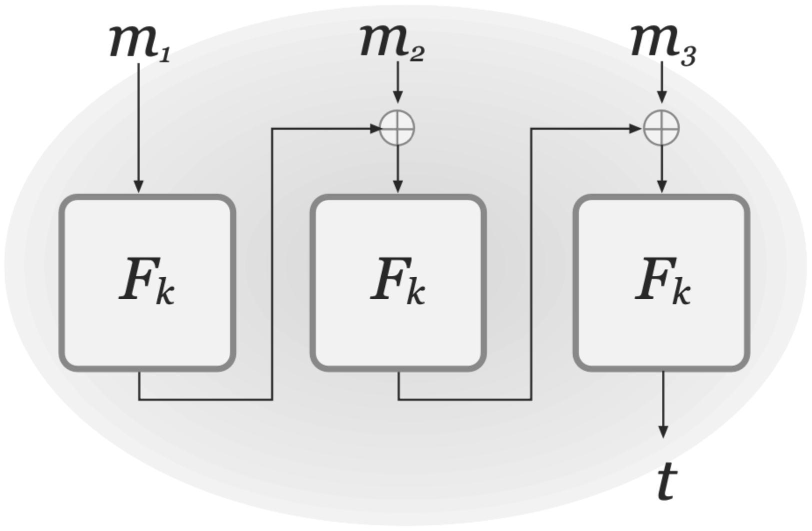
Neither of these are CCA-Secure because they are malleable. By changing bits in the ciphertext, certain blocks of the decrypted message will become random through the inverse function F_k^{-1}, but others will become predictably modified because they are also being $\oplus$'d with information *outside* of the inverse function.

▼ Cipher Block Chaining Message Authentication Code (CBC-MAC)

A message, m_1 is input into a function F_k, its ouput is $\oplus$'d with m_2 before m_2 is run through F_k. Once all messages have been run through F_k, its final output is t, which should prove integrity.

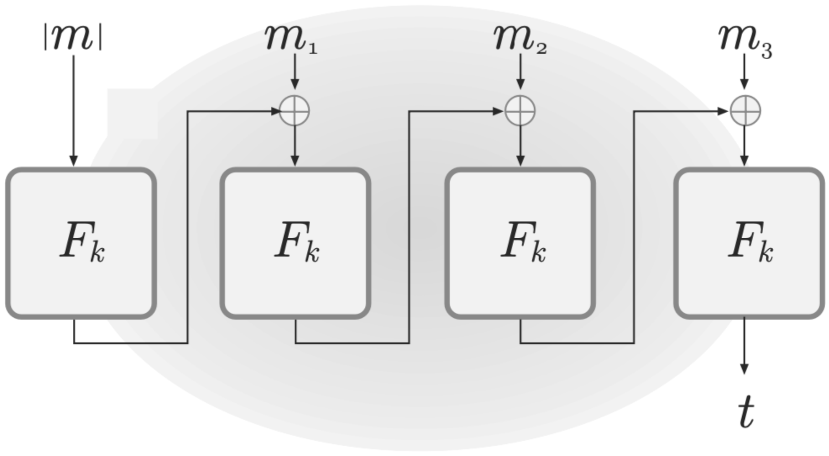
This methodology *does not require an IV*, unlike CBC-Mode encryption. It is *deterministic* for this reason. CBC-MACs do not need to be probabilistic to be secure! Verification can simply be done by re-computing the result.

Note that we explicitly avoid using an IV because its use reveals too much information to adversaries.



The methodology of $\mathrm{Vrfy}_k(m, t)$ is simply to compare the output of this scheme with t. As long as k remains secret this is valid.

To ensure the utility of CBC-MAC for *arbitrary length* messages, we must also encode the length of the message (in blocks) as the first block of our CBC. This is done at the start of the blocks rather than at the end to ensure unforgeability.



▼ Confusion-Diffusion Paradigm

The CDP is, simply, a paradigm for constructing concise and random-looking permutations. The goal is to construct a PRP F with a large block length from many smaller PRPs $\{f_i\}$ with a smaller block length.

Through the use of numerous PRPs $\{f_i\}$, the scheme is said to introduce *confusion* into F.

On its own, this definition is not enough. Assuming the input on $F_k(x)$ is split into the n components that will be fed to n sub functions, changes to a single component in x will produce predictable changes in $F_k(x)$.

Therefore we require an additional step to perform *diffusion*. In this *diffusion* step, the bits of the output are permuted using a *mixing permutation*. This spreads local change (e.g. change in a single byte) throughout the entirety of the output.

In practice, it is the application of the *confusion* and *diffusion* steps in sequence that is referred to as a 'round'. As the rounds are applied, the output of one becomes the input to the next. Through the application of sufficient rounds, F becomes indistinguishable from a PRP.

▼ Substitution-Permutation Networks (SPNs)

SPNs are used to directly implement the Confusion-Diffusion Paradigm.

For each round:

- *Key mixing* (Confusion): Set $x := x \oplus k$, where k is the current round's sub-key.
- *Substitution* (f_i / S-box): Set $x := S_1(x_1)||...||S_n(x_n)$ where x_i is the i th byte of x.
- *Permutation* (Diffusion): Permute the bits of x to obtain the output of the round.

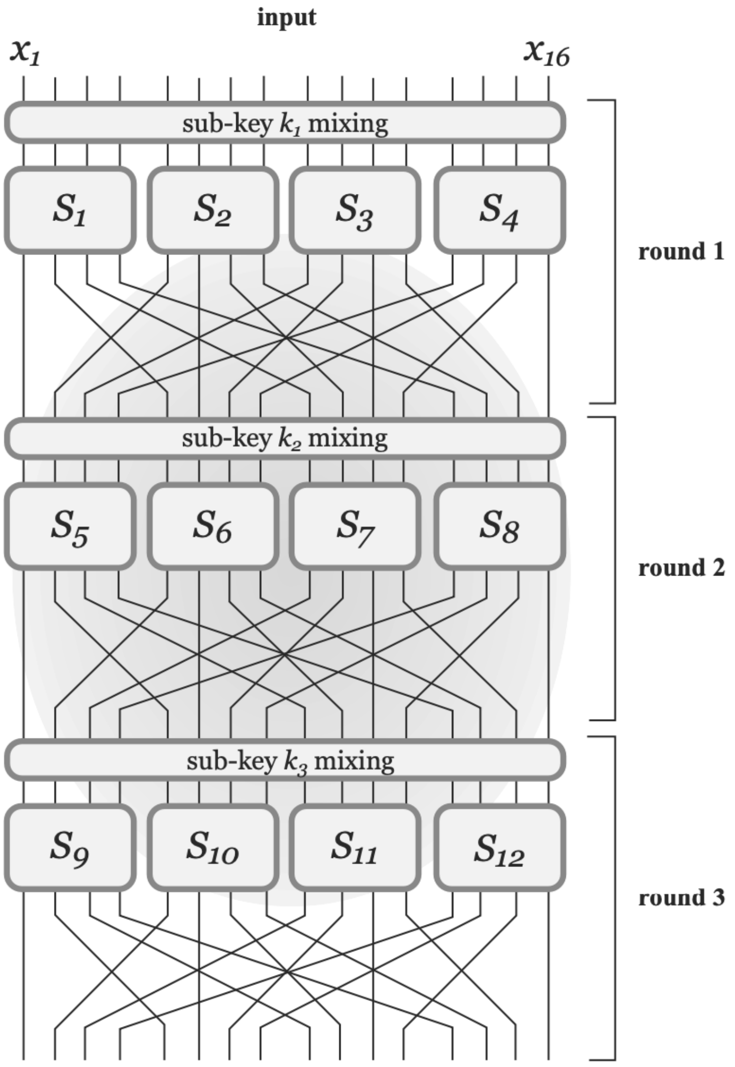
Finally, perform another *Key mixing* step such that the final *Substitution* and *Permutation* processes are not wasted.

There are certain requirements placed on the SPN that are required to achieve the desired effect:

1. Changing a single input bit for any given S-box changes at least two of the resulting output bits.
2. Mixing permutations are designed such that the bits output by any given S -box affect the input of multiple S-boxes in the next round.

The presence of these requirements causes an **Avalanche Effect**: a single permuted bit in the input can be expected to alter half of the output bits.

An example of a properly constructed 3-round SPN.



▼ Feistel Networks

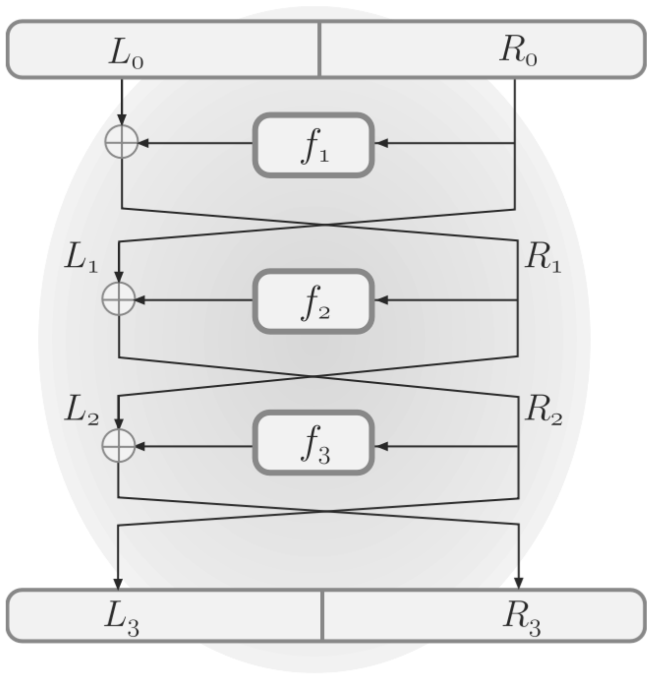
Unlike the S-Boxes used in SPNs, the sub-functions of Feistel Networks do not need to be invertible. By consequence, even Hash functions can be used in each of the blocks. The goal here, then, is to compose an invertible function from non-invertible components.

For a balanced Feistel Network with ℓ-bit block length, the ith round function $\hat{f}_i$ takes as input a sub-key k_i and an $\ell/2$-bit string and generates an $\ell/2$-bit output. A single master key k can be used to derive each sub-key.

- $f_i : \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2}$ via $f_i(R) \overset{\text{def}}{=} \hat{f}_i(k_i, R)$

- Note that the functions $\hat{f}_i$ are public and known- it is only the keys used for them that are kept a secret.
- At each stage of the network, we solve each half of the network L_i and R_i.
  - $L_i := R_{i-1}$
  - $R_i := L_{i-1} \oplus f_i(R_{i-1})$
- To invert / decrypt, we simply run the output of the network back through- but reverse the order of each sub-function f_i.

This is an example of a 3-round Feistel Network



Attacking Feistel Networks:

- In the one round case:
  - $F_k(L_0, R_0) = (R_0, f_1(R_0) \oplus L_0)$ where f_1 is in some way dependent on k.
  - The left half of the output is always the right half of the input, and is being XORed with the left half of the input. Simply using the all zero

string can allow the attacker to make inferences about f_1.

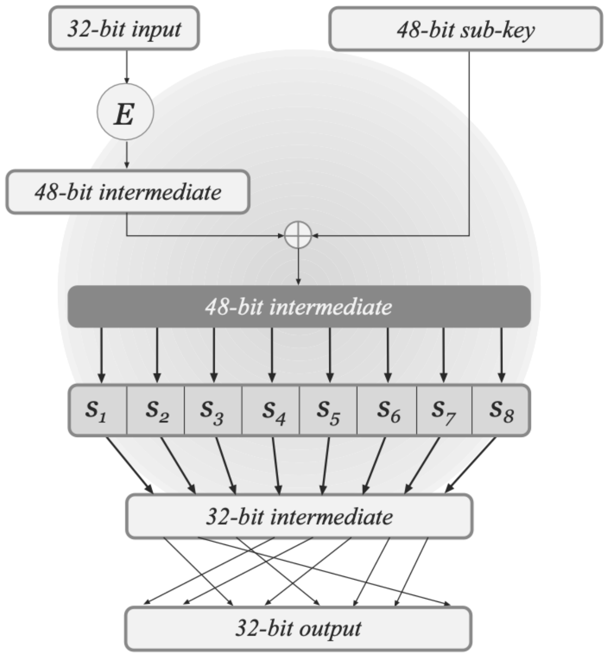
- In the two-round case:
  - $F_k(L_0, R_0) = (f_1(R_0) \oplus L_0, R_0 \oplus f_2(f_1(R_0) \oplus L_0))$ where f_1, f_2 are in some way dependent on k.
  - There are still correlations between the outputs of F_k on related inputs that can be used to distinguish F_k from a random permutation. In this way, even a two round Feistel Network does not uphold Perfect Secrecy.
- With more than two rounds and certain conditions being satisfied, Feistel Networks become indistinguishable from PRPs.

▼ Data Encryption Standard (DES)

The DES block cipher is a 16-round Feistel Network with a block length of 64 bits and a key length of 56 bits. The same round function $\hat{f}$ is used in each of the 16 rounds. The round function takes a 48-bit sub-key and a 32-input (half a block). The *key schedule* of DES is used to derive a sequence of 48-bit sub-keys $k_1, \ldots, k_{16}$ from the 56-bit master key.

By the DES definition, each of the sub-keys k_i is simply a permuted subset of 48 bits of the master key.

The DES round function $\hat{f}$, sometimes called the *mangler function*, is a practical implementation of an SPN with the slight variation that the S-boxes in this case do not need to be invertible since they are being utilized in the greater context of a Feistel Network. Indeed, the entire round function $\hat{f}$ does not need to be invertible. The entirety of this scheme is publicly known, it is only the master key that is kept secret.



A graphical representation of the DES mangler.

S-box properties under DES:

- Each S-box is a 4-to-1 function (4 inputs are mapped to each possible output)
- Each row in the table contains each of the 16 possible 4-bit strings exactly once
- Changing one bit of any input to an S-box always changes at least two bits of the output

This produces the same *avalanche effect* as described in SPNs.

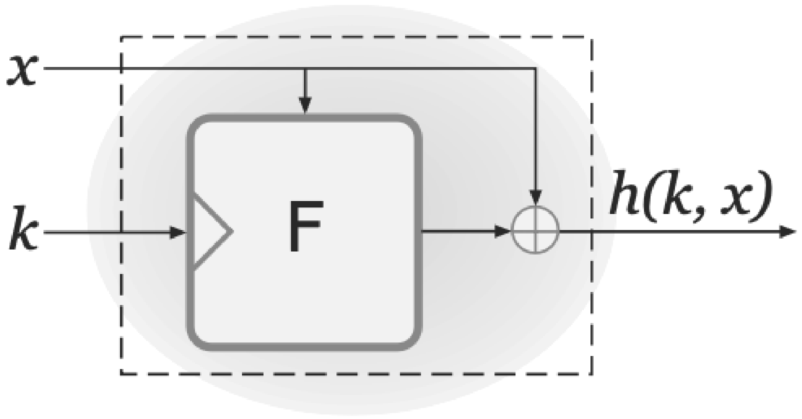
Attacks on DES can only practically carried out if the number of rounds is greatly reduced from 16 to a number ≤ 3. It's in these cases we can showcase the *avalanche effect* is not complete, allowing us to make inferences about the scheme and plaintext we should not be able to.

▼ Davies-Meyer Construction

The purpose of this construction is to construct a collision-resistant compression function from any given block cipher that satisfies strong security properties.

The Davies-Meyer Construction defines the compression function

$h : \{0,1\}^{\ell}$ by $h(k,x) \overset{def}{=} F_k(x) \oplus x$.



A graphical representation of the equation for constructing the compression function.

Proving collision resistance based on the assumption that F is a strong PRP is not possible as far as we know. If, however, we model F as an *ideal cipher*, such a proof does become possible. In this case, the scheme is collision resistant so long as ℓ is sufficiently large.

▼ Hash Functions

▼ Hash Functions Formal Definition

Put simply, hash functions transform arbitrary length inputs ($\{0,1\}^*$) into fixed length outputs ($\{0,1\}^{\ell(n)}$). Because of the implied utility of hash functions though, we expect them to not cause *collisions*, or at least to do so with a vanishing probability. Collisions occur when two distinct inputs x, x' produce the same output, meaning $H(x) = H(x')$.

Collision Resistance:

Formally, we look at *keyed* hash functions.

$H^s(x) \; = H(s,x)$. It must be difficult to find a collision for a randomly generated key s. In the case of hash functions, the key is generally not kept secret, unlike the keyed functions we've seen prior. An attacker wishes simply to find a collision.

Attacker:

- $s \leftarrow \mathrm{Gen}(1^n)$
- The adversary A is given key s and outputs two messages x, x'
- Both hash messages are run through the hash function producing hashes in the form $\{0,1\}^{\ell' n}$
- *iff* $H^s(x) = H^s(x')$ the attacker A outputs 1, otherwise outputs 0.

To insist that a hash function is collision resistant is to show that the probability of this attacker outputting 1 is negligible, or $P[A = 1] \leq \mathrm{negl}$.

Weaker Notions: Strict Collision Resistance is not always necessary.

Second-preimage resistance: A hash function is *collision-resistant* if it is infeasible for any PPT adversary to find a collision in H for two inputs x, x' where $x \neq x'$.

Preimage resistance: For an attacker given s, y where $y = H^s(x)$ for a uniform x, it is infeasible for a PPT adversary to find a value x' such that $x' \neq x$ and $y = H^s(x')$.

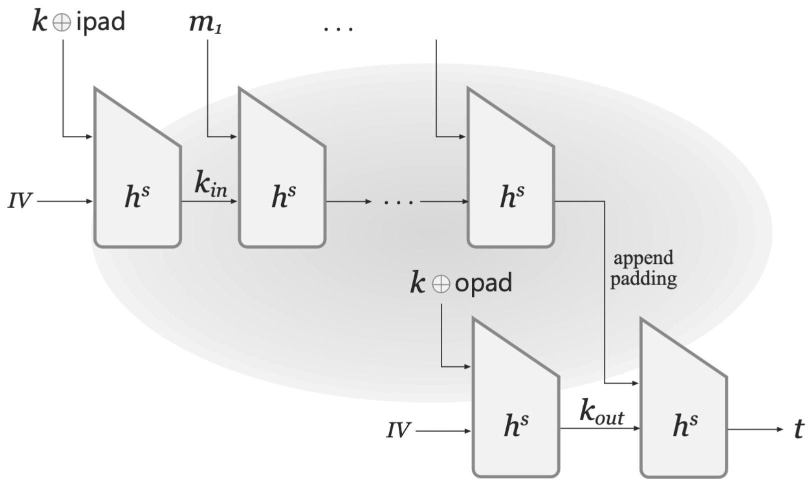
▼ Hash-and-MAC

By applying a hash function H to some message m, we can effortlessly generate a MAC tag t for the given message. This is secure because the attacker A cannot output any new hash values, and H being collision resistant ensures the attacker will be unable to find new messages with a matching hash. The MAC utilized here is based on CBC-MAC.

- Gen' : Choose k uniformly from $\{0,1\}^n$; $s \leftarrow \mathrm{Gen}_H(1^n)$
- Mac' : Given k, s, for some message $m \in \{0,1\}^*$, output $t \leftarrow \mathrm{Mac}_k(H^s(m))$
- Vrfy' : Given k, s, for some message $m \in \{0,1\}^*$, and some tag t, output 1 iff $\mathrm{Vrfy}_k(H^s(m), t) = 1$.

▼ HMAC

In practice, the Hash-and-MAC methodology is rarely used because it requires instantiating *two* cryptographic primitives, the Hash function and a block cipher. Utilizing the Merkle-Damgård Transform to hash our arbitrarily sized message, we can perform our MAC and Hash in synchrony.



- $\mathrm{Gen}(1^n)$: Choose k uniformly from $\{0,1\}^{n'}$; $s \leftarrow \mathrm{Gen}_H(1^n)$
- Mac' : Given k, s, for some message $m \in \{0,1\}^*$, output
  $t := H^s((k \oplus \mathrm{opad})||H^s((k \oplus \mathrm{ipad})||m))$
- Vrfy' : Given k, s, for some message $m \in \{0,1\}^*$, and some tag t, output 1 iff
  $t = H^s((k \oplus \mathrm{opad})||H^s((k \oplus \mathrm{ipad})||m))$.

▼ Merkle-Damgård Transform

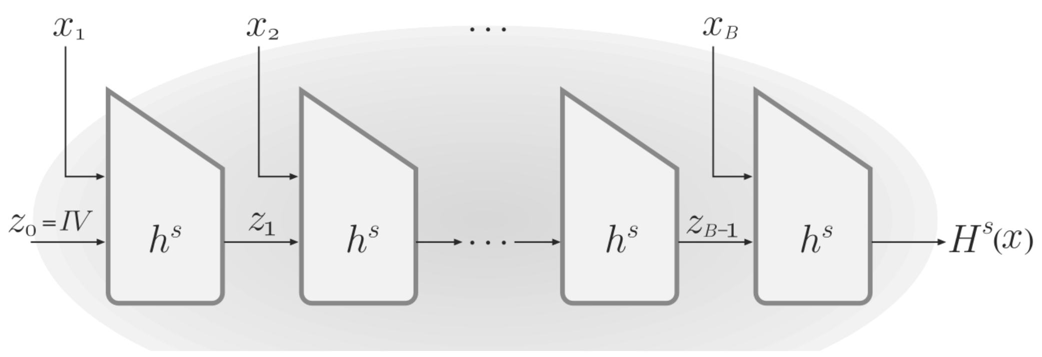
The purpose of this scheme is to transform fixed length hash functions into *arbitrary length* hash functions.

Let (Gen, h) be a compression function for inputs of length $n + n\prime \geq 2n$ with output length n. Fix $\ell \leq n'$ and $IV \in \{0,1\}^n$. Construct hash function (Gen, H) as follows:

- Gen : remains unchanged
- H : receiving input of key s and string $x \in \{0,1\}^*$ of length $L < 2^\ell$, do:

1. Append a 1 to x, followed by enough zeros that the length of the resulting string is ℓ less than a multiple of n'. Then append L, encoded as an ℓ-bit string. Parse the resulting string as the sequence of n'-bit blocks $x_1, ..., x_B$.
2. Set $z_0 \coloneqq IV$.
3. For $i = 1, ..., B$, compute $z_i \coloneqq h^s(z_{i-1}||x_i)$.
4. Output z_B.

If (Gen, h) is collision-resistant, then so is (Gen, H). Explained simply, we are breaking the input up into respective chunks and running each of these chunks through the fixed length hash function, using a variation of chained CBC-Mode.



▼ Likelihood of Collision

Given a positive integer N, say that some $q \leq \sqrt{2N}$ elements $y_1, \ldots, y_q$ are chosen uniformly an d independently from a set of size N. We can then assert that

$$\frac{q*(q-1)}{4N} \leq 1 - e^{-q(q-1)/2N} \leq \mathrm{coll}(q, N) \leq \frac{q*(q-1)}{2N}$$

▼ PKCS #7 Encoding

The purpose of this encoding is to ensure that messages encoded using a block cipher encryption scheme are less succeptible to tampering or forgery, boosting the integrity and decreasing the malleability of the scheme.

Sender:

- Assume message is an integral # of bytes
- Let L be the block length in bytes of the cipher
- Let $b > 0$ be the number of bytes that need to be appended to the message to get length as multiple of L
    - $1 \leq b \leq L$; note $b \neq 0$
- Append b (encoded in 1 byte), b times
    - I.e. if 3 bytes of padding are needed, append $0\text{x}030303$
    - I.e. if 4 bytes of padding are needed, append $0\text{x}04040404$

Receiver:

- Extract the final number from the the Encoded Data. This number will be interpreted as b. This is why b must always be ≥ 1, even if our message already evenly fits into the block length. The receiver must be able to unambiguously read the amount of padding. If it expects to read the padding amount and we instead see the unpadded ciphertext, things would go wrong.
- Use CBC-Mode decryption to obtain the encoded data
- Say the final byte of encoded data has value b
    - If $b = 0$ or $b > L$ return 'error'
    - If final b bytes of the encoded data are not all equal to b, return 'error'
    - Otherwise, strop off final b bytes of the encoded data, and output what remains as the message

▼ Padding Oracles

By understanding the definition of PKCS #7 Encoding, we can systematically approach the violation of a scheme through what at first appears to be a trivial leakage of information. So long as the 'Padding Oracle' tells us whether a given ciphertext c has proper formatting, we can actually decode the entirety of the message. Knowing that neither method we've discussed for block ciphers are CCA-Secure, this should make sense. In both cases, we can devise a methodology for producing predictable changes to the final block of decrypted data, thus giving us the opportunity to modify the padding content. By finding the number of padding bytes through trial and error, we can then create modifications that increase this number, moving on to the previous bit as the bit to modify. Through some clever XORing, we can uncover the entirety of the plaintext this way.

Formal Assumptions

The Factoring Assumption

Let $\mathrm{GenModulus}$ be a PPT algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$ and p, q are n-bit prime numbers. This is allowed to fail with negl probability.

Here we define an experiment in which an attacker $\mathcal{A}$ is given an integer N and is asked to deduce the prime factorization that produced the number.

$\mathrm{Factor}_{\mathcal{A},\mathrm{GenModulus}}(n)$:

1. $(N, p, q) \leftarrow \mathrm{GenModulus}(1^n)$
2. $\mathcal{A}$ is given N and outputs $p', q' > 1$
3. The output of the experiment is defined to be $1 \iff p' * q' = N$, returning 0 otherwise.

Note that if the experiment evaluates to 1, $\{p', q'\} = \{p, q\}$.

The Factoring Assumption states that for all PPTs $\mathcal{A}$ there exists a negl such that

$$\Pr[\mathrm{Factor}_{\mathcal{A},\mathrm{GenModulus}}(n) = 1] \leq \mathrm{negl}(n)$$

RSA Assumption

Given a modulus N and an integer $e > 2$ relatively prime to $\phi(N)$, we know that exponentiation to the eth power modulo N is a *permutation*. Therefore for any $y \in \mathbb{Z}_N^*$ we can define $[y^{1/e} \mod N]$ to be the unique element of $\mathbb{Z}_N^*$ that yields y when raised to the eth power modulo N.
$x = [y^{1/e} \mod N] \iff x^e = [y \mod N]$

Informally, our task is to compute x for a modulus N of unknown factorization.

Let GenRSA be a PPT algorithm that, given input 1^n outputs a modulus N that is the product of two n-bit primes, as well as two integers e, d where $e, d > 1$ and $\text{GCD}(e, \phi(N)) = 1$ and $ed = [1 \mod \phi(N)]$. This is allowed to fail with negl probability.

Here we define an experiment in which the attacker $\mathcal{A}$ is given modulo N, exponent e, and a uniformly chosen $y \in \mathbb{Z}_N^*$, and asked to produce the value x which, when raised to the eth power produces $[y \mod N]$.

$\text{RSAinv}_{\mathcal{A},\text{GenRSA}}(n)$:

1. $(N, e, d) \leftarrow \text{GenRSA}(1^n)$
2. Choose a uniform $y \in \mathbb{Z}_N^*$
3. $\mathcal{A}$ is given (N, e, y) and outputs $x \in \mathbb{Z}_n^*$
4. The output of the experiment is defined to be $1 \iff x^e = [y \mod N]$, returning 0 otherwise.

The RSA Assumption states that for all PPTs $\mathcal{A}$ there exists a negl such that

$$\Pr[\text{RSAinv}_{\mathcal{A},\text{GenRSA}}(n) = 1] \leq \text{negl}(n)$$

▼ Discrete-Logarithm / Diffie-Hellman Assumptions

Let $\mathcal{G}$ denote a generic PPT algorithm for group generation. Given input of size 1^n, $\mathcal{G}$ outputs a description of a cyclic group $\mathbb{G}$ as well as its order q where $||q|| = n$ and a generator $g \in \mathbb{G}$. Assume that we can perform group operations and check if a given bit-string is an element of $\mathbb{G}$ in PPT. Despite all cyclic groups of the same order being isomorphic (comprised of the same elements), their representation will determine the complexity of the operations.

Here we define an experiment in which the attacker $\mathcal{A}$ is given the cyclic group $\mathbb{G}$, its order q, its generator g, and a uniformly selected member of $h \in \mathbb{G}$. The attacker's goal is to find a value x such that $g^x = h$. In other words, to what power is the generator g raised such that it produces the uniformly chosen value h? The value x is guaranteed to exist since all values in $\mathbb{G}$ are found by raising g to some power $\mod q$.

$\text{DLog}_{\mathcal{A},\mathcal{G}}(n)$:

1. $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^n)$
2. Choose a uniform $h \in \mathbb{G}$
3. $\mathcal{A}$ is given $(\mathbb{G}, q, g, h)$ and outputs some $x \in \mathbb{Z}_q$
4. The output of the experiment is defined to be $1 \iff g^x = h$, returning 0 otherwise.

The DLog Assumption states that for all PPTs $\mathcal{A}$ there exists a negl such that

$$\Pr[\mathrm{DLog}_{\mathcal{A},\mathcal{G}}(n) = 1] \leq \mathrm{negl}(n)$$

Encryption Schemes

Private-Key Encryption

Formal Definition for Private Key Encryption

A private-key encryption scheme contains three probabilistic functions, $\mathrm{Gen},\ \mathrm{Enc}, \&\ \mathrm{Dec}$ such that:

- $k \leftarrow \mathrm{Gen}(1^n)$ we assume that this key satisfies $|k| \geq n$.
- $c \leftarrow \mathrm{Enc}_k(m)$ where $m \in \{0,1\}^*$
- $m \leftarrow \mathrm{Dec}_k(c)$ where $m \in \{0,1\}^*$ or an error, denoted $\perp$

We require, simply, that $m = \mathrm{Dec}_k(\mathrm{Enc}_k(m))$

If, for some reason, we specify the Message Space ($\mathcal{M}$) to be $m \in \{0,1\}^{\ell(n)}$, we then say that the set of these functions can be referred to as a fixed length private key encryption scheme of length $\ell(n)$.

We also make a few assumptions moving forwards:

- Dec is actually deterministic
- Calls to Enc are probabilistically independent from one another
- We assume all encryption schemes are **stateless**, like the one above, unless specified otherwise

One-Time Pad

Only to be used with a given key *one time*, the One-Time Pad specifies a means to achieve perfect secrecy in communication at the cost of the requirement that $|k| = |m|$.

- Let $\mathcal{M} = \{0,1\}^n$
- Gen: choose a uniform key $k \in \{0,1\}^n$
- $\text{Enc}_k(m) = k \oplus m$
- $\text{Dec}_k(c) = k \oplus c$

▼ Public-Key Encryption

▼ Plain Rivest-Shamir-Adleman (RSA) Scheme

Before fortifying it, we first consider an insecure "plain" RSA encryption scheme.

Here, we define GenRSA as it is referenced in the RSA Assumption.

$\text{GenRSA}:$

1. $(N, p, q) \leftarrow \text{GenModulus}(1^n)$
2. $\phi(n) := (p-1) * (q-1)$
3. Choose some $e > 1$ such that $\text{GCD}(\text{e}, \phi(\text{N})) = 1$
4. Compute $d := [e^{-1} \mod \phi(N)]$
5. Return (N, e, d)

Now, with GenRSA in hand, we define the public key encryption scheme as follows:

- Gen : on input 1^n run $(N, e, d) \leftarrow \text{GenRSA}(1^n)$. $\langle N, e\rangle$ is the public key. $\langle N, d\rangle$ is the private key.
- Enc : on input a public key $pk = \langle N, e\rangle$ and a message $m \in \mathbb{Z}_N^*$, compute the ciphertext $c := [m^e \mod N]$.
- Dec : on input a private key $sk = \langle N, d\rangle$ and a ciphertext $c \in \mathbb{Z}_N^*$, compute the message $m := [c^d \mod N]$.

If the message is not chosen *uniformly* from $\mathbb{Z}_N^*$, we risk exposing information about the message. Moreover, this algorithm is *deterministic* and so by definition *cannot* be CPA-Secure.

▼ Diffie-Hellman key-exchange protocol

The Diffie-Hellman key-exchange protocol is radical in that it allows for the exchange of secret keys over an *insecure* channel, such as the public internet.

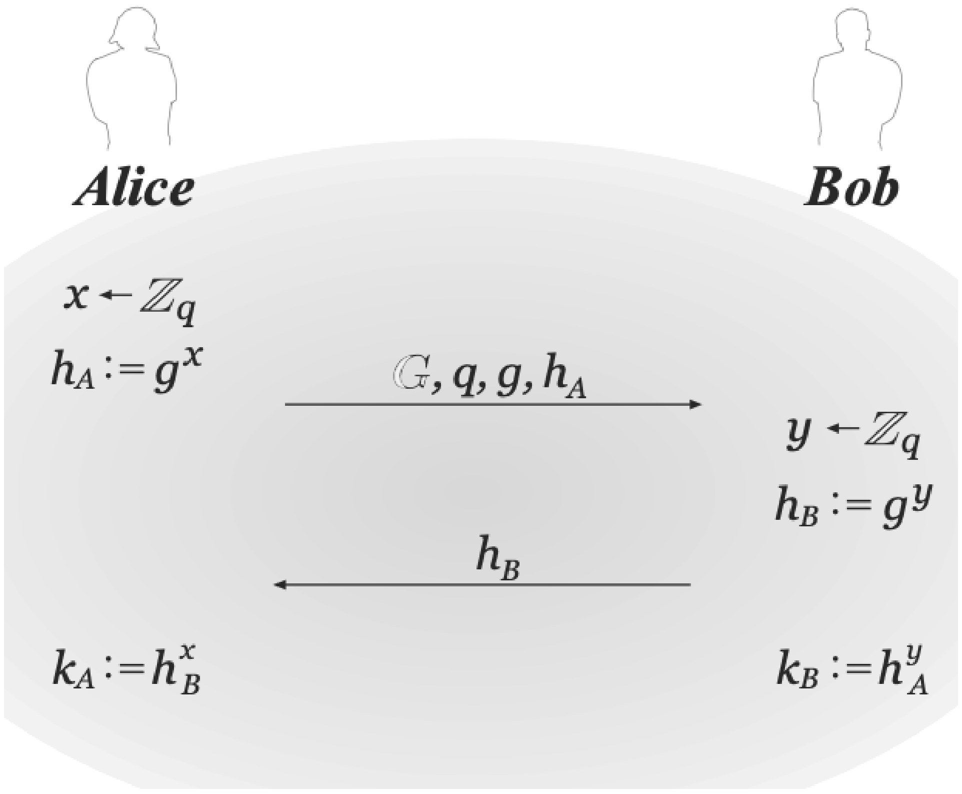
We assume we are given $\mathcal{G}$, a PPT algorithm that when given input 1^n outputs a description of a cyclic group $\mathbb{G}$, its order q (where $||q|| = n$), and a generator $g \in \mathbb{G}$.

The protocol is defined formally as follows:

1. Alice performs $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^n)$
2. Alice chooses a uniform $x \in \mathbb{Z}_q$ and computes $h_A := g^x$.
3. Alice sends $(\mathbb{G}, q, g, h_A)$ to Bob.
4. Receiving $(\mathbb{G}, q, g, h_A)$, Bob chooses a uniform $y \in \mathbb{Z}_q$ and computes $h_B := g^y$. Bob sends h_B to Alice and outputs the key $k_B := h_A^y$.
5. Alice receives h_B and outputs the key $k_A := h_B^x$.

We can see the protocol is correct and that both parties arrive at the same value via the following relation:

$$\begin{aligned} k_B &= h_A^y = (g^x)^y = g^{xy} \\ k_A &= h_B^x = (g^y)^x = g^{xy} \end{aligned}$$



A graphical representation of the formalized protocol

Attacking the protocol:

Here, we define an experiment in which keys are exchanged. An attacker A is given a either a uniformly chosen bit-string of length n, or the real key k that was generated during execution. If the attacker is unable to distinguish the two with better than random probability, the scheme is secure.

$\mathrm{KE}^{\mathrm{eav}}_{\mathcal{A},\Pi}(n)$:

1. Both parties holding 1^n execute the protocol Π. This produces a transcript trans containing all messages sent by both parties as well as a key k output by each of the parties
2. A uniform bit $b \in \{0, 1\}$ is chosen. If $b = 0$, set $\hat{k} := k$, otherwise choose a uniform $\hat{k} \in \{0, 1\}^n$.
3. $\mathcal{A}$ is given trans and $\hat{k}$ and outputs a bit b'.

4. The output of the experiment is defined to be $1 \iff b' = b$, returning 0 otherwise.

The exchange protocol is considered secure in the presence of an eavesdropper if for all PPTs $\mathcal{A}$ there exists a negl such that

$$\Pr[\mathrm{KE}^{\mathrm{eav}}_{\mathcal{A},\Pi}(n) := 1] \leq \frac{1}{2} + \mathrm{negl}(n)$$

▼ El Gamal Encryption

Taher El Gamal realized that the Diffie-Hellman key-exchange protocol could be repurposed as a full public key encryption scheme, rather than for simply choosing a key k. To do this, we perform our operations as before. Once a key k is agreed upon, Bob is able to encrypt a message $m \in \mathbb{G}$ simply by sending $k * m$ to Alice, who can easily recover m using her knowledge of k. By extension, we argue that the eavesdropper learned nothing about m.

This scheme is used in a symmetric key exchange.

- Gen : On input 1^n run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$. Then choose a uniform $x \in \mathbb{Z}_q$ and compute $h := g^x$. The public key is $\langle \mathbb{G}, q, g, h \rangle$ and the private key is $\langle \mathbb{G}, q, g, x \rangle$. The message space is $\mathbb{G}$.
- Enc : Given a public key $pk = \langle \mathbb{G}, q, g, h \rangle$ and a message $m \in \mathbb{G}$, choose a uniform $y \in \mathbb{Z}_q$ and output the ciphertext $\langle g^y, h^y * m \rangle$.
- Dec : Given a private key $sk = \langle \mathbb{G}, q, g, x \rangle$ and a ciphertext $\langle c_1, c_2 \rangle$, output $\hat{m} := c_2 / c_1^x$.

To showcase that this is a successful scheme, let $\langle c_1, c_2 \rangle = \langle g^y, h^y * m \rangle$ where $h = g^x$. Then consider the equivalence

$$\hat{m} = \frac{c_2}{c_1^x} = \frac{h^y * m}{(g^y)^x} = \frac{(g^x)^y * m}{g^{xy}} = \frac{g^{xy} * m}{g^{xy}} = m$$