Notes for EECS 475

Cryptography

Albon Wu

April 1, 2025

Contents

0 Introduction

Essential concepts in cryptography, precise attack models and security definitions, and constructions of real-world cryptosystems.

Professor: Mahdi Cheraghchi

1 The Cryptographic Methodology

Suppose Alice wants to communicate a message to Bob that she wants to be safe from an eavesdropper. She should do three things:

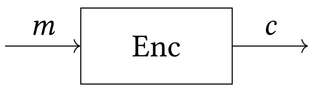
1. Form a realistic model of the scenario, adjusting as necessary to allow for possible solutions.
2. Precisely define the desired function and security properties of a potential solution.
3. Construct and analyze a solution, ideally proving it satisfies the desired properties.

Step 3 is the most mathematical and requires careful attention. But obviously your system is still vulnerable if you formally prove everything but your model does not fully capture reality. In cryptography, we often prove things (e.g., RSA) given well-established assumptions.

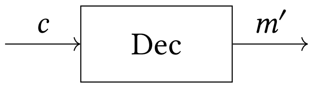
1.1 Modeling encryption

Here's a specific case of encryption:

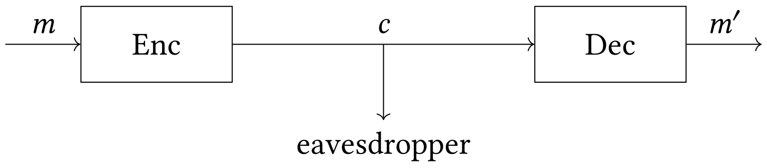
Sender A is represented by an algorithm Enc(·) that takes a "plaintext" message m from a (finite) set of possible messages $\mathcal{M}$ ("message space") and outputs a "ciphertext" c from a finite set $\mathcal{C}$ ("ciphertext space").



Receiver B is represented by an algorithm Dec(·) that takes some $c \in \mathcal{C}$ and outputs some $m' \in \mathcal{M}$.

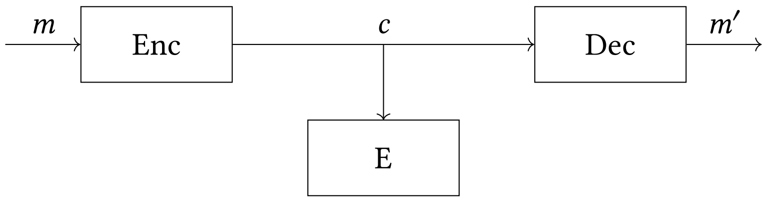


So here is our model so far:



The eavesdropper is represented by an algorithm E(·) that takes $c \in \mathcal{C}$ and outputs...

...what does it output?



At the very least, the system should be "correct" in that if there is no adversary, legitimate users should be able to communicate. That is,

$$\forall m \in \mathcal{M} \qquad \text{Dec}(\text{Enc}(m)) = m.$$

The tricky part is formulating in a mathematical sense what "security" means. At a minimum, E shouldn't be able to always recover the message m from c.

But what prevents E := Dec? It brings up an interesting problem in our model - there is no way to distinguish a legitimate party from an attacker.

The correct way is to use a secret that only the legitimate parties know. What is not correct is to keep the decryption algorithm itself secret - secrets don't last if you can't change them quickly.

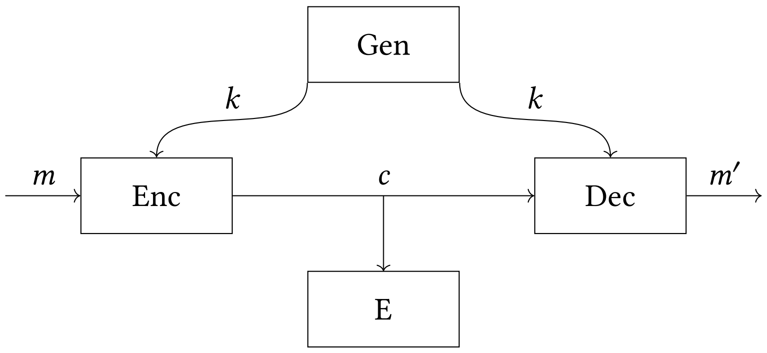
"Not only is it bad, it is terrible." (Madhi)

In cryptography, everything is open source. There are no secrets in knowledge; security by obscurity is to be avoided. This is called Kerckhoffs's principle: the system should remain secure even if all its algorithms are known to the public.

"The enemy knows the system." (Shannon)

Instead, we give the Dec box an extra (secret) input key. Making the key itself secret is a problem for later.

So some algorithm Gen(·) (the "key generator") is responsible for producing and sharing that key. Note that Gen is not secret, even if its output is.



This is called "private (or symmetric) key encryption". Our updated definition of correctness is then:

$$\forall m \in \mathcal{M}, \forall k \in \mathcal{K} \qquad \mathrm{Dec}_k(\mathrm{Enc}_k(m)) = m$$

where $\mathcal{K}$ is the "key space."

Security should mean:

1. E won't be able to recover m from c
2. E won't be able to recover k
3. E won't recover *any part of* or anything about m

1.2 Shannon and perfect secrecy

Definition 1.2.1. *An encryption scheme (Gen, Enc, Dec) is* ***Shannon secret*** *if for any message distribution D over message space $\mathcal{M}$, any fixed $\overline{m} \in \mathcal{M}$ and fixed ciphertext $\overline{c} \in C$,*

$$\Pr_{\substack{m \leftarrow D \\ k \leftarrow \mathrm{Gen}()}}(m = \overline{m} \mid \mathrm{Enc}_k(m) = \overline{c}) = \Pr_{m \leftarrow D}(m = \overline{m}).$$

If you are an eavesdropper on a Shannon secret communication, you are no more certain about the message contents after eavesdropping than before.

The RHS is also called *a priori*, while the LHS is *a posteriori*.

This condition provides a very strong security guarantee but is very cumbersome and requires that we deal with every possible D.

$$\Pr_{m,k}(m = \overline{m} \mid \mathrm{Enc}_k(m) = \overline{c}) = \frac{\Pr_{m,k}(m = \overline{m} \wedge \mathrm{Enc}_k(m) = \overline{c})}{\Pr_{m,k}(\mathrm{Enc}_k(m) = \overline{c})}$$
$$= \frac{\Pr_{m,k}(m = \overline{m} \wedge \mathrm{Enc}_k(\overline{m}) = \overline{c})}{\Pr_{m,k}(\mathrm{Enc}_k(m) = \overline{c})}$$

where in the last equality we replace m by $\overline{m}$. By independence of m, k:

$$= \Pr_m(m = \overline{m}) \cdot \frac{\Pr_k(\mathrm{Enc}_k(\overline{m}) = \overline{c})}{\Pr_{m,k}(\mathrm{Enc}_k(m) = \overline{c})}.$$

So a scheme is Shannon secret if and only if the right term is 1 for all D and all $\overline{m} \in \mathrm{supp}(D) \subseteq \mathcal{M}$. Note that the message distribution is not relevant here.

The fraction equals 1 if:

$$\Pr_k(\mathrm{Enc}_k(\overline{m}) = \overline{c}) = \Pr_{m,k}(\mathrm{Enc}_k(m) = \overline{c}) \qquad \forall \overline{m} \in \mathrm{supp}(D), \forall \overline{c} \in C. \tag{$*$}$$

Let D be the uniform distribution over $\mathcal{M}$. This means the LHS is constant for all $\overline{m}$. This means that $\forall m_0, m_1 \in \mathcal{M}, \forall \overline{c} \in C$:

$$\Pr_k(\mathsf{Enc}_k(m_0) = \overline{c}) = \Pr_k(\mathsf{Enc}_k(m_1) = \overline{c}).$$

This means that if we encrypt distinct messages, they are equally likely to be a given ciphertext; that is, the distribution of the ciphertext is the same. This is called **perfect secrecy**; it is implied by Shannon secrecy.

Conversely, if perfect secrecy holds, then ($*$) holds.

$$\Pr_{\substack{m \leftarrow D \\ k \leftarrow \mathrm{Gen}}}(\mathsf{Enc}_k(m) = \overline{c}) = \sum_{m' \in \mathcal{M}} \Pr_k(\mathsf{Enc}_k(m') = \overline{c}) \cdot \Pr_m(m = m')$$

conditioning over all possible messages and applying total probability. By perfect secrecy, this equals

$$\sum_{m \in \mathcal{M}} \Pr_k(\mathsf{Enc}_k(\overline{m}) = \overline{c}) \cdot \Pr_m(m = m')$$

for arbitrary $m' \in \mathcal{M}$, so we can pull out the first sum term factor:

$$\Pr_k(\mathsf{Enc}_k(\overline{m}) = \overline{c}) \cdot \sum_{m' \in \mathcal{M}} \Pr(m = m')$$

$$= \Pr_k(\mathsf{Enc}_k(\overline{m}) = \overline{c})$$

as desired. So we have just shown:

Theorem 1.2.1. *Shannon secrecy is equivalent to perfect secrecy.*

1.3 One-time pad

The one-time pad, also known as the Vernam cipher, was patented in 1917. For OTP, we have

$$\mathcal{K} = \{0,1\}^\ell \qquad \mathcal{M} = \{0,1\}^\ell \qquad \mathcal{C} = \{0,1\}^\ell$$

with

- $\mathsf{Gen}()$: choose $k \longleftarrow \{0,1\}^\ell$ uniformly randomly.
- $\mathsf{Enc}_k(m \in \{0,1\}^\ell)$: output $c = m \oplus k \in \{0,1\}^\ell$.
- $\mathsf{Dec}_k(c \in \{0,1\}^\ell)$: output $m = c \oplus k$.

Correctness is fairly immediate:

Claim. One-time pad is correct.

Proof. Let $m, k \in \{0,1\}^\ell$ be arbitrary. Then

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = (m \oplus k) \oplus k = m$$

by associativity. □

The secrecy proof is a few more lines:

Theorem 1.3.1. *OTP is perfectly secret.*

Proof. We need that for any $\overline{c} \in \{0,1\}^\ell$, it holds that $\Pr_k(\mathsf{Enc}_k(m) = \overline{c})$ is constant over $m \in \{0,1\}^\ell$. By routine bit manipulation:

$$\Pr_k(\mathsf{Enc}_k(m) = \overline{c}) = \Pr_k(m \oplus k = \overline{c}) = \Pr_k(k = m \oplus \overline{c}) = \frac{1}{2^\ell}$$

by uniformity of k, as desired. □

> "Sometimes you read a complicated math paper, and at the end it says 'as desired' just because they wanted to be done with the proof, not because the proof was desired." (Mahdi)

There are some caveats of one-time pad, though.

1. The key must be very long—as long as the message.
2. It is *one-time*; if you encrypt two messages with the same key and then XOR the ciphertexts, you get information about the messages.

So it seems like we got baited by the cryptographic methodology, because we followed it and ended up with a solution that is still vulnerable to an attack (point 2).

But not really, because this attack is outside the threat model that the security guarantee tries to cover. We formulated the kind of secrecy we want—Shannon secrecy—and then came up with an attack that is outside this model, one that requires you to encrypt multiple times.

The question we now want to ask is:

Question. Is there a perfectly secret scheme with shorter keys?

Sadly, the answer is no.

Theorem 1.3.2 (Shannon)**.** *In any perfect encryption, we must have* $|\mathcal{K}| \geq |\mathcal{M}|$.

Proof. Suppose toward a contradiction that $|\mathcal{K}| < |\mathcal{M}|$.

Fix $\overline{c} \in C$ and pick any $m_0 \in \mathcal{M}$ such that m_0 encrypts to $\overline{c}$. Denote the set of possible decryptions of $\overline{c}$ by $\mathcal{D}$ and note that $\mathcal{D} = \mathcal{K}$ because every decryption of a ciphertext corresponds to exactly one key.

Then $|\mathcal{D}| < |\mathcal{K}|$, so we can pick $m_1 \in \mathcal{M} \backslash \mathcal{D}$. By construction,

$$\Pr_k(\mathsf{Enc}_k(m_0) = \overline{c}) = 1 \neq 0 = \Pr_k(\mathsf{Enc}_k(m_1) = \overline{c}),$$

contradicting perfect secrecy. □

We conclude that an attack does exist that gives an attacker some information about the ciphertext. But what kind of attacker would actually do this? It would require picking some m_0, encrypting it, then decrypting that $\overline{c}$ with all possible $k \in \mathcal{K}$ to find a message outside $\mathcal{D}$.

And this is exponential in the length of the key—infeasible for basically any attacker. So the good news is that even though an attack can exist for non-perfectly secret cryptosystems, we can make them computationally infeasible.

But that's for the next chapter.

2 Computational Security

2.1 The computational approach to security

In our analysis of computational security, we only care about security against attacks that are limited to a feasible (whatever that means) amount of computation. We must also necessarily allow a tiny chance that the attacker can violate security (e.g., guess the key).

The template is "No feasible attacker can get more than a tiny advantage against system X under attack model Z."

Now what should "feasible computation" mean?

- Is $2^{30} \approx 10^9$ operations feasible? Sure. This is like just over the upper bound of a competitive programming solution.
- $2^{60} \approx 10^{18}$ operations? Yeah, that's a few minutes on a supercomputer.
- 2^{80}, which is 1-2 years on a supercomputer? Yes.
- 2^{100}, one million years on a supercomputer? We'll still say yes, since who knows how supercomputers will evolve in the next decade.
- 2^{128}, one trillion years even after 200 more years of Moore's law? That's 2^{70} operations every second since the start of the universe. We can consider this infeasible.
- 2^{256} is more than the number of atoms in the universe. The rule of thumb for quantum computers is that they can achieve square-root speedup on numerical computations, so the cryptographic answer is to double your key length.

What should "tiny" mean? In practice it means "much much smaller than the chance of something else going wrong."

- 2^{-60} probability is smaller than the chance of both sender and receiver being hit by lightning in a given year.

Runtime divided by probability of success is the quantity we should care about as the "cost" of an attack. This is called **concrete security** and is important for practice, but cumbersome to keep track of.

For simplicity, we use the **asymptotic approach**. Instead of taking specific numbers, we parameterize every scheme by a security parameter n that somehow indicates how much security you need.

The good parties should use algorithms whose complexity grows slowly in n, such as $O(n \log n)$ or $O(n^3)$. A "feasible" attacker is one that uses a polynomial-time algorithm, or $\mathsf{poly}(n)$ for short.

In the asymptotic approach, the attacker must have a tiny advantage that we define by a negligible function that decays faster than any inverse polynomial.

Definition 2.1.1. *A function $\epsilon(n)$ is **negligible**, written $\epsilon(n) = \mathsf{negl}(n)$, if $\epsilon(n) = o(n^{-c})$ for* *every constant $c > 0$. Equivalently,*

$$\lim_{n \to \infty} n^c \cdot \epsilon(n) = 0$$

for every constant $c > 0$.

Example 2.1.1. *$\epsilon(n) = \frac{1}{n^5}$ is not negligible, but $\epsilon(n) = \frac{1}{2^n}$ is.*

So a more formal template for computational security is:

> Every polynomial-time attacker has only negligible advantage when attacking system X.

Now we will reformulate OTP with these mathematical foundations. Recall that the problem is that the key must be at least as long as the message. If we shorten the key, we need some way to expand it to something that 1) can be XOR'd with the message and 2) is functionally equivalent to the original key.

Pseudorandom generators achieve this goal.

2.2 Pseudorandom generators

Python and C++ have pseudorandom generator implementations, but the PRGs we use in cryptography are even more secure (because they must be robust to adversaries).

Definition 2.2.1. *A **pseudorandom generator** with expansion $\ell(n)$ is a deterministic,* poly(n) *time algorithm G with two properties:*

1. *$|G(s)| = \ell(n) > n$ for all $s \in \{0,1\}^n$; that is, G stretches an input of length n to $\ell(n)$ bits.*
2. ***pseudorandomness**: (informally) for a uniformly random seed $s \in \{0,1\}^n$, $G(s) \in \{0,1\}^{\ell(n)}$ "looks like" a uniformly random string to all feasible observers.*

   *More precisely, let $D : \{0,1\}^n \longrightarrow \{0,1\}$ be a **distinguisher** which is a polytime algorithm.*

   *D's **advantage** is given by*

   $$\mathrm{Adv}_G(D) = \left| \Pr_{s \leftarrow \{0,1\}^n}(D(1^n, G(s)) = 1) - \Pr_{y \leftarrow \{0,1\}^{\ell(n)}}(D(1^n, y) = 1) \right|$$

   *where 1^n is the number n written in unary; that is, the number with n ones. G is **pseudorandom** if for all (possibly randomized) polytime distinguishers D, $\mathrm{Adv}_G(D) =$ negl(n).*

Intuitively, there are two possible worlds:

- "Real world:" D is given $y = G(s)$ for some uniformly random secret seed $s \in \{0,1\}^n$.
- "Ideal world:" D is given a uniformly random $y \longleftarrow \{0,1\}^{\ell(n)}$

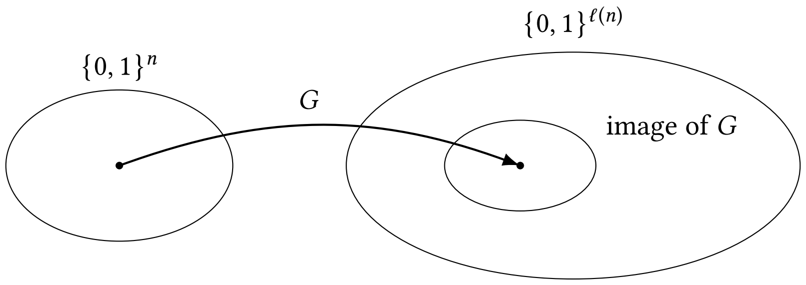
The advantage of D measures how much its predictions differ between worlds. If that difference is small; that is, it always guesses 1 or always guesses 0, then intuitively it's not very good at distinguishing between the worlds. Indeed, the advantage of D is low in this case.

Now suppose D always gets it right. The difference in terms is maximal and so is the advantage, which checks out. If D is always wrong, note that advantage is still maximized.

Also, note that we constrain D to be polytime. Otherwise, it could iterate through the size-2^n image of G and have perfect accuracy. In this case, it has advantage

$$\mathrm{Adv}_G(D) = \left|\Pr_{\text{real}}(D = 1) - \Pr_{\text{ideal}}(D = 1)\right| = \left|1 - \frac{2^n}{2^{\ell(n)}}\right| \geq \frac{1}{2},$$

where the bound follows from the fact that $\ell(n) > n$. $\Pr_{\text{ideal}}(D = 1) = \frac{2^n}{2^{\ell(n)}}$ follows from this picture:



and the fact that $D = 1$ only when considering an element in $\mathrm{im}(G)$. This is the best an adversary can hope to achieve.

Now it is probably prudent to ask:

Question. Do PRGs exist?

Well, yesn't. We assume they do for the sake of cryptography, but no one has shown that rigorously. This is because of the following:

$$\begin{aligned} &\text{PRGs exist} \\ \implies &\text{One-way functions exist} \\ \implies &\mathsf{P} \neq \mathsf{NP} \end{aligned}$$

since the decision problem "is an element in the image of a one-way-function" is in NP (by construction of PRGs) but not in P.

2.3 EAV-secrecy

Definition 2.3.1. *A **stream cipher** is a pair of polytime, deterministic algorithms* (Init, NextBit) *where:*

- Init(s) *outputs a state* st_0
- NextBit(st_i) *outputs a bit* $y_{i+1} \in \{0,1\}$ *and next state* st_{i+1}

We can define $G_\ell(s)$ *as the repeated composition of* NextBit *with itself:*

$$G_\ell(s) = (\mathrm{NextBit} \circ \cdots \circ \mathrm{NextBit})(\mathrm{Init}(s)).$$

*The stream cipher is **pseudorandom** if for every* $\ell(n) = \mathrm{poly}(n)$, *the function* G_ℓ *is pseudo-*

random.

Note that a stream cipher allows you to generate bits sequentially, while a PRG gives you a block of bits whose length you must know in advance. A stream cipher is like a cassette tape, while a PRG is like a CD.

Stream ciphers exist if and only if PRGs exist. The forward direction is easy to see; just compose NextBit a fixed number of times. The other direction is harder, but can be constructed using a PRG with $\ell(n) \geq n + 1$.

Now back to perfect secrecy...

Recall that our goal is to perform OTP by extending a short key. We have already shown this will not be perfectly secret, but we don't even need that much. We will use a revised notion of secrecy, EAV-**secrecy**.

Definition 2.3.2. *Let* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a cryptosystem and* $\mathcal{A}$ *be a polytime adversary.* $\mathcal{A}$ *plays the* EAV***-security game*** *as follows:*

1. *Get the security parameter n.*
2. *Output two messages* m_0, m_1 *where* $|m_0| = |m_1| = \ell(n)$.
3. *The game generates* $k \longleftarrow \mathsf{Gen}(1^n)$ *"behind the scenes" and a ciphertext* $c \longleftarrow \mathsf{Enc}_k(m_b)$ *for either* $b = 0$ *or* $b = 1$ *(fixed).*
4. *Receive c and output a decision bit.*

We can break down the EAV-security game in English as follows:

1. Get the security parameter.
2. Using it, produce two equal-length messages that you claim will expose the cryptosystem's secrecy.
3. Have the cryptosystem discreetly generate a key, use it to encrypt one of the messages, and send the ciphertext to you.
4. Output which message you think was encrypted.

So bit b in some sense tells us which "world" we're in—which message's ciphertext the adversary is looking at.

There is also a notion of advantage in the EAV-security game:

Definition 2.3.3. *The* ***advantage*** *of an adversary* $\mathcal{A}$ *against* Π *in the* EAV*-security game is*

$$\mathrm{Adv}_{\Pi}^{\mathsf{EAV}}(\mathcal{A}) = \left| \Pr_{k \leftarrow \mathsf{Gen}}(\mathcal{A} \textit{ accepts in world 1}) - \Pr_{k \leftarrow \mathsf{Gen}}(\mathcal{A} \textit{ accepts in world 0}) \right|$$

$$= \left| \Pr_{k \leftarrow \mathsf{Gen}}(\mathcal{A}(1^n, \mathsf{Enc}_k(m_1)) = 1) - \Pr_{k \leftarrow \mathsf{Gen}}(\mathcal{A}(1^n, \mathsf{Enc}_k(m_0)) = 1) \right|.$$

Note that this is more of a property of the adversary than the ciphertext distributions. It doesn't matter if a ciphertext was equally likely encrypted from m_0 or m_1; it only matters if the adversary thinks so.

Remark. We talk a lot about the degree of randomness an adversary can perceive. To make this more concrete, consider the fact that a coin flip isn't uniformly distributed.

If you have a supercomputer and sensors and compute time, you can probably compute the exact distribution of a particular coin toss. But if you're just a human, it's no different to you than uniform randomness.

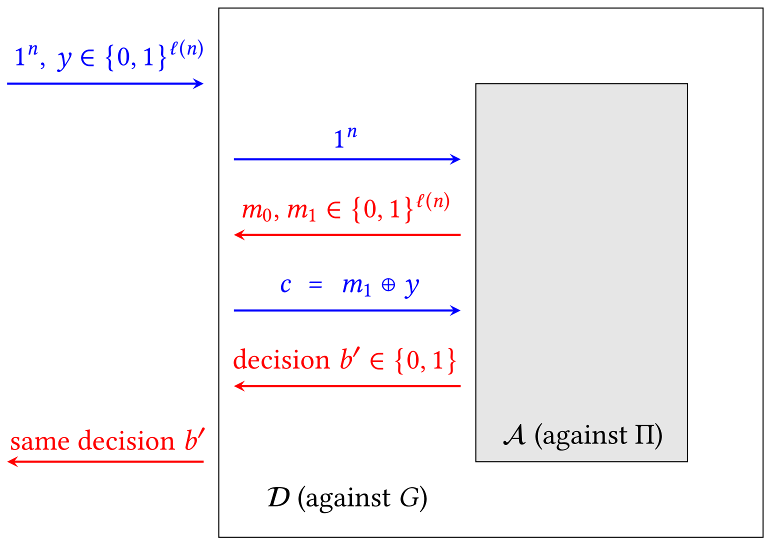
We can construct an EAV-secure cryptosystem as follows:

- Gen: choose a uniform key $k \leftarrow \{0,1\}^n$.
- Enc: for a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^{\ell(n)}$, output $c = G(k) \oplus m$ for some PRG G.
- Dec: for a key $k \in \{0,1\}^n$ and ciphertext c, output $G(k) \oplus c$.

Correctness comes from the OTP argument. EAV-security is a little trickier.

Theorem 2.3.1. *If G is a PRG, then the above cryptosystem is* EAV*-secure.*

Proof. Take some polytime adversary $\mathcal{A}$ against Π and consider this construction:


$1^n,\ y \in \{0,1\}^{\ell(n)}$
1^n
$m_0, m_1 \in \{0,1\}^{\ell(n)}$
$c = m_1 \oplus y$
decision $b' \in \{0,1\}$
same decision b'
$\mathcal{A}$ (against Π)
$\mathcal{D}$ (against G)

In steps, our construction is:

1. $\mathcal{D}$ takes in n and some length-$\ell(n)$ string y that may or may not be pseudorandomly generated by G.
2. $\mathcal{D}$ runs $\mathcal{A}$ with security parameter n.
3. $\mathcal{A}$ gives two equal-length strings m_0, m_1.
4. $\mathcal{D}$ encrypts m_1 (WLOG) with y as the key and sends it to $\mathcal{A}$.
5. $\mathcal{A}$ identifies whether it is in world 0 or 1.
6. $\mathcal{D}$ returns $\mathcal{A}$'s decision as its own.

We claim this distinguishes G, a contradiction. First, we write

$$\mathrm{Adv}_G^{\mathrm{PRG}}(\mathcal{D}) = \left| \Pr_{k \leftarrow \{0,1\}^n} (\mathcal{D}(1^n, G(k)) = 1) - \Pr_{y \leftarrow \{0,1\}^{\ell(n)}} (\mathcal{D}(1^n, y) = 1) \right|$$

$$= |\Pr(\mathcal{A} \text{ given encryption of } m_1 \text{ accepts}) - \Pr(\mathcal{A} \text{ given complete randomness accepts})|$$

$$= |\Pr(\mathcal{A} \text{ accepts in world 1}) - \Pr(\mathcal{A} \text{ accepts in "hybrid" world})|.$$

By "hybrid world" we mean the world in which $\mathcal{A}$ is fed some completely random string $c = m_1 \oplus y$. c has this property because by definition, y is not pseudorandom, and XOR with a truly random string produces another truly random string.

So this is problematic. We know the above term, the advantage of $\mathcal{D}$ against G, is negligible because G is pseudorandom, but what we are really after is that

$$\mathrm{Adv}_\Pi(\mathcal{A}) = |\Pr(\mathcal{A} \text{ in world 1 accepts}) - \Pr(\mathcal{A} \text{ in world 0 accepts})|$$

is negligible. Our earlier expression for $\mathrm{Adv}_G^{\mathrm{PRG}}(\mathcal{D})$ correctly includes a term for acceptance in world 1, but nothing for world 0.

To reconcile this, recall that the "acceptance in world 1" term comes about because in our construction we feed $c = m_1 \oplus y$ into $\mathcal{A}$. If we make another construction $\mathcal{D}'$ where we feed in $c = m_0 \oplus y$, its advantage is

$$\mathrm{Adv}_G^{\mathrm{PRG}}(\mathcal{D}') = \left| \Pr_{k \leftarrow \{0,1\}^n} (\mathcal{D}'(1^n, G(k)) = 1) - \Pr_{y \leftarrow \{0,1\}^{\ell(n)}} (\mathcal{D}'(1^n, y) = 1) \right|$$

$$= |\Pr(\mathcal{A} \text{ given encryption of } m_0 \text{ accepts}) - \Pr(\mathcal{A} \text{ given complete randomness accepts})|$$

$$= |\Pr(\mathcal{A} \text{ accepts in world 0}) - \Pr(\mathcal{A} \text{ accepts in "hybrid" world})|.$$

Now the triangle inequality gives

$$\begin{aligned}\mathrm{Adv}_\Pi(\mathcal{A}) &= |\Pr(\mathcal{A} \text{ accepts in world 1}) - \Pr(\mathcal{A} \text{ accepts in world 0})| \\ &\leq |\Pr(\mathcal{A} \text{ accepts in world 1}) - \Pr(\mathcal{A} \text{ accepts in "hybrid" world})| \\ &\quad + |\Pr(\mathcal{A} \text{ accepts in world 0}) - \Pr(\mathcal{A} \text{ accepts in "hybrid" world})|.\end{aligned}$$

And we conclude by noting

$$\mathrm{Adv}_\Pi(\mathcal{A}) \leq \mathrm{Adv}_G^{\mathrm{PRG}}(\mathcal{D}) + \mathrm{Adv}_G^{\mathrm{PRG}}(\mathcal{D}') \leq \mathrm{negl}(n) + \mathrm{negl}(n) = \mathrm{negl}(n).$$

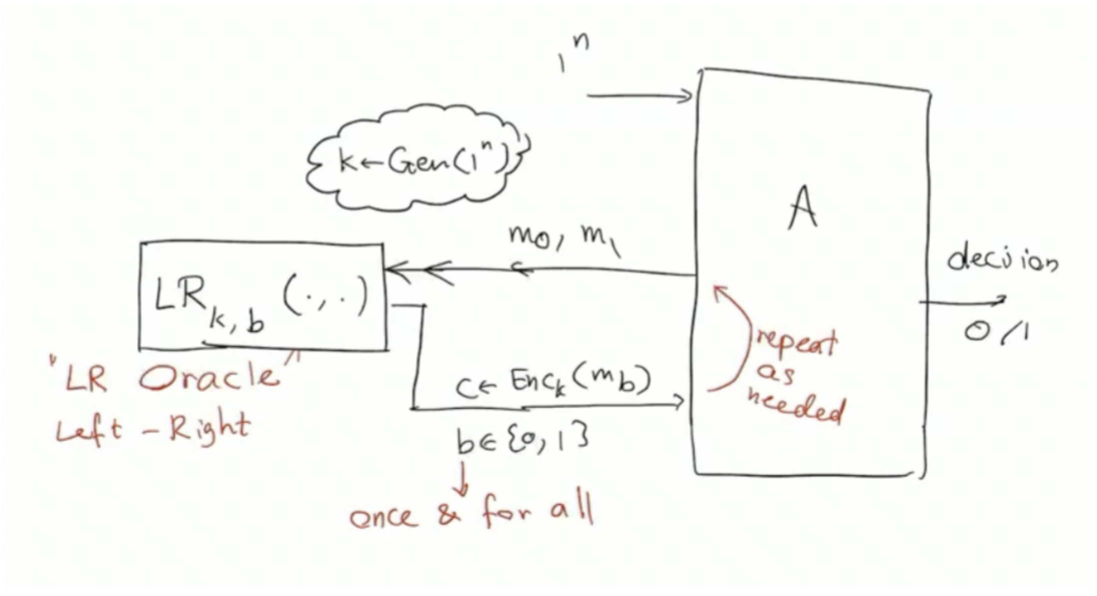
$\square$

This is called a proof by reduction, where we take an arbitrary adversary against Π and convert it into a distinguisher for G. The existence of an efficient $\mathcal{A}$ therefore implies an efficient distinguisher for G, contradicting its pseudorandomness.

2.4 Chosen-plaintext attack

Recall the other problem with one-time pad, which is the "one-time" part. We haven't fixed this with the eavesdropping game since it's still a one-shot model.

The stronger attack model we will look at now is called **chosen-plaintext attack**.



The adversary $\mathcal{A}$ will send two messages to the **left-right oracle**, which encrypts one and sends it back to $\mathcal{A}$. Afterward, $\mathcal{A}$ may repeat this as much as it wants before giving a binary decision.

Concretely, the CPA game is parameterized by a bit $b \in \{0,1\}$ representing left or right.

1. Attacker $\mathcal{A}$ is given a security parameter n and a key is chosen $k \longleftarrow \mathsf{Gen}(1^n)$ by the game.
2. $\mathcal{A}$ can make several (possibly adaptive) queries to an oracle $\mathsf{LR}_{k,b}(\ldots)$, defined as

$$\mathsf{LR}_{k,b}(m_0, m_1) = \begin{cases} c \longleftarrow \mathsf{Enc}_k(m_b) & \text{if } |m_0| = |m_1| \\ \text{error} & \text{otherwise} \end{cases}$$

3. $\mathcal{A}$ outputs a decision in $\{0,1\}$; that is, accept/request.

There is also a notion of advantage in the CPA game:

Definition 2.4.1. *The **advantage** of $\mathcal{A}$ in the CPA game against a cryptosystem $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is*

$$\mathrm{Adv}_{\Pi}^{CPA}(\mathcal{A}) = \left|\Pr(\mathcal{A}^{\mathsf{LR}_{k,1}(\cdot,\cdot)} \text{ accepts}) - \Pr(\mathcal{A}^{\mathsf{LR}_{k,0}(\cdot,\cdot)} \text{ accepts})\right|.$$

Definition 2.4.2. *A cryptosystem Π is **CPA-secure** if every p.p.t. $\mathcal{A}$ has only negligible advantage in the CPA game against Π; that is,*

$$\mathrm{Adv}_{\Pi}^{\mathrm{CPA}}(\mathcal{A}) = \mathsf{negl}(n).$$

Remark. There is a variant of this game in the book: $\mathcal{A}$ gets unlimited access to an Enc_k oracle, so it can encrypt arbitrary messages, but only one call to the LR oracle.

Obviously, our definition has this as a special case; just query LR with both arguments as m, but it turns out that the definitions are equivalent. We will skip the proof of this statement, but there will be a similar proof when we talk about public key cryptography.

Now if you are particularly shrewd, you have probably noticed the following:

Theorem 2.4.1 (Questionable...). *There are no CPA-secure systems.*

"Proof". Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be any cryptosystem. Then attack it as follows:

1. Choose $m_0 \neq m_1$ from the message space.
2. Query LR on (m_0, m_0). This is simply $c \longleftarrow \mathsf{Enc}_k(m_0)$.
3. Since we have infinite access to LR, query it again on (m_0, m_1). If the result equals c, output 0; otherwise, output 1.

WLOG, if in world 0, then $\mathsf{LR}_{k,0}(m_0, m_1) = \mathsf{Enc}_k(m_0) = c$, and we output 0, as desired. □

So what's wrong with this?...

We assumed Enc_k *was deterministic.*

Removing this assumption creates some complications. Firstly, the ciphertext length must be greater than the message length, since each message must correspond to more than one ciphertext. Also, the encryption function must have internal state to remember which outputs it has given.

So the correct theorem is:

Theorem 2.4.2. *There exists no CPA-secure encryption scheme with deterministic and stateless* Enc_k.

Alright. Then how do you make a non-deterministic encryption function? Recall stream ciphers that produce constant "streams of randomness." If we need a length n key, we can generate that many bits, and generate n more anytime we need to keep encrypting.

The only thing is that the legitimate parties must agree on some "sequence number," or how far into the "generate n bits at a time" sequence they should start in order to get the right key.

What we seem to need is a random-access version of a stream cipher; we call this a **pseudo-random function**.

2.5 Pseudorandom functions

Following the intuition of a "random access stream cipher," the object we need for CPA security is a pseudorandom function (PRF). This is a function with two arguments: key and address/input $F : \mathcal{K} \times \mathcal{X} \longrightarrow \mathcal{Y}$. $\mathcal{K}$ is the key space, $\mathcal{X}$ is the input/address space, and $\mathcal{Y}$ is the output space.

F should act like a "truly random function" $\mathcal{U}$, which is a function that has a random lookup table; $\mathcal{U}(x)$ is uniformly random and fully independent for each x.

The model for a PRF, then, is a keyed function $F : \mathcal{K} \times \mathcal{X} \longrightarrow \mathcal{Y}$, where $\mathcal{K} = \mathcal{X} = \mathcal{Y} = \{0,1\}^n$ typically, that is efficient to compute.

For short, we write $F_k(x) = F(k,x)$, where k is random but F is deterministic given the k. We want, for random k, that $F_k(\cdot)$ is indistinguishable from $\mathcal{U}(\cdot)$, a uniformly random function.

Remark. Given k, the description of F needs $|\mathcal{X}| \cdot \log |\mathcal{Y}|$ bits, which is $2^n \cdot n$ bits for $\mathcal{X} = \mathcal{Y} = \{0,1\}^n$. That's n bits for each of the 2^n possible inputs.

This is enormous! As a result, we can't afford to define the "PRF game" the same way as the PRG game, because the thing we're trying to distinguish from pure randomness is so big we can't even feasibly write it down.

In other words, the lookup table is so massive that we can't just store it in memory and read off of it to serve PRF requests; we need a device (i.e., oracle) that can efficiently serve a single query at a time.

The PRF game, played by a p.p.t. distinguisher D against F allows D to repeatedly query its oracle $\mathcal{O}$ on its choices of $x \in \mathcal{X} = \{0,1\}^n$ and receives $y = \mathcal{O}(x) \in \mathcal{Y} = \{0,1\}^n$.

Definition 2.5.1. *A keyed function F is a* ***pseudorandom function (PRF)*** *if every p.p.t. distinguisher D has negligible advantage in the PRF game against F:*

$$\mathsf{Adv}_F^{\mathrm{PRF}}(D) = \left| \Pr_{k \leftarrow \mathcal{K}}(D^{F_k}(1^n) = 1) - \Pr_{\text{random } \mathcal{U}}(D^{\mathcal{U}(\cdot)}(1^n) = 1) \right| = \mathsf{negl}(n).$$

The existence of PRFs is equivalent to the existence of PRGs, so we don't know if they exist. In practice, people use heruistics:

1. DES (1976): $\mathcal{K} = \{0,1\}^{56}$ and $\mathcal{X} = \mathcal{Y} = \{0,1\}^{64}$. Not concretely secure because not enough keys; D can enumerate all keys to figure out if $\mathcal{O}$ is some $F_k(\cdot)$ or not.
2. AES (2001): $\mathcal{X} = \mathcal{Y} = \{0,1\}^{120}$ and $\mathcal{K} = \{0,1\}^n$ for $n = 128, 192, 256$. Concretely secure in that no better attack than brute force has proven successful so far.

Example 2.5.1. *Define $F_k(x) = k \oplus x$. The following is a distinguisher:*

1. *Query $y_0 = \mathcal{O}(0^\ell)$ and $y_1 = \mathcal{O}(1^\ell)$ for some length ℓ.*
2. *If $y_0 \oplus y_1 = 0^\ell \oplus 1^\ell = 1^\ell$, accept; otherwise, reject.*

If $\mathcal{O} = F_k$, then D always accepts. If $\mathcal{O} = \mathcal{U}$, then y_0 and y_1 are uniform and independent. Therefore,

$$\Pr(y_0 \oplus y_1 = 1^\ell) = \frac{1}{2^n} = \mathsf{negl}(n).$$

So the advantage of D is $1 - \mathsf{negl}(n) \approx 1$, *so F is not a PRF.*

Let's go back to CPA security. Intuitively, we come up with the following scheme:

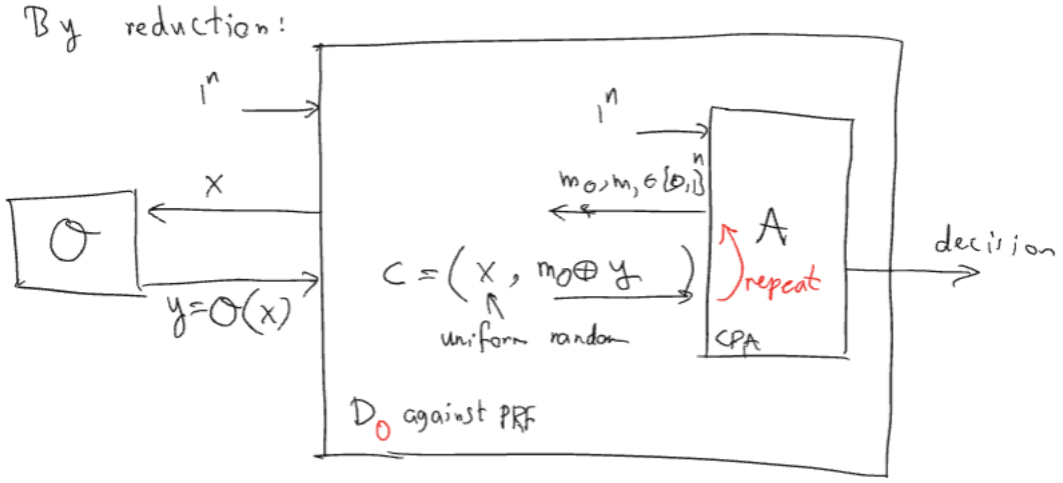
- $\mathsf{Gen}(1^n)$: Choose a random key $k \longleftarrow \{0,1\}^n$.
- $\mathsf{Enc}(m)$: Choose random $x \longleftarrow \{0,1\}^n$. Output $c = (x, m \oplus F_k(x))$.

- Dec(c) for $c := (x, c')$: Output $c' \oplus F_k(x)$.

Correctness follows directly from the fact that F_k is deterministic.

Theorem 2.5.1. *If F is a secure PRF, then this construction is a CPA-secure encryption scheme for messages of length n.*

Proof. Consider an adversary $\mathcal{A}$. We do the following reduction:



Out goal is to show the advantage of $\mathcal{A}$ is negligible, which is the contrapositive of the familiar "if $\mathcal{A}$ has non-negligible advantage, than so does D."

We first analyze the advantage of D_0, which is the construction where $c = (x, m_0 \oplus y)$. If D_0 is in its real world ($\mathcal{O} = F_k$ for random k), then it perfectly simulates the left world of $\mathcal{A}$. Therefore,

$$\Pr(D_0 \text{ in the real world accepts}) = \Pr(\mathcal{A} \text{ in the left world accepts}).$$

If D_0 is in its ideal world ($\mathcal{O} = \mathcal{U}$), then x and $m_0 \oplus y$ are both uniformly random, so c is too. In this case, D_0 simulates some "hybrid" world.

$$\Pr(D_0 \text{ in the ideal world accepts}) = \Pr(\mathcal{A} \text{ in the hybrid world accepts}) \pm \mathsf{negl}(n).$$

The negligible term comes from the birthday paradox: we choose x at random to feed to $\mathcal{O}$, but if we randomly pick two identical values, then y will be constant if $\mathcal{O} = F_x$. The probability of this happening is roughly quadratic in the number of queries, hence

$$\Pr(D_0 \text{ chooses some } x \text{ more than once}) \approx \frac{(\#\text{ queries})^2}{2 \cdot 2^n} = \frac{\mathsf{poly}(n)}{2^{n+1}} = \mathsf{negl}(n).$$

The extra factor of 2 in the denominator comes from the approximation: the number of ways to pick two identical queries is $\binom{q}{2} \approx \frac{q^2}{2}$.

In any case, we can now reason about the advantage of $\mathcal{A}$ using the advantage of D. We write

$$\begin{aligned} \mathrm{Adv}(\mathcal{A}) &= |\Pr(\mathcal{A} \text{ accepts in left world}) - \Pr(\mathcal{A} \text{ accepts in right world})| \\ &\le |\Pr(\mathcal{A} \text{ accepts in left world} - \Pr(\mathcal{A} \text{ accepts in hybrid world}))| \quad (1) \\ &+ |\Pr(\mathcal{A} \text{ accepts in right world}) - \Pr(\mathcal{A} \text{ accepts in hybrid world})| \quad (2) \end{aligned}$$

Line (1) reduces to:

$$\begin{aligned} &|\Pr(\mathcal{A} \text{ accepts in left world} - \Pr(\mathcal{A} \text{ accepts in hybrid world}))| \\ = &|\Pr(D_0 \text{ in real world accepts}) - \Pr(D_0 \text{ in ideal world accepts}) + \mathsf{negl}(n)| \\ &\mathrm{Adv}_F^{\mathrm{PRF}}(D_0) + \mathsf{negl}(n). \end{aligned}$$

Analogously, line (2) reduces to $\mathrm{Adv}_F^{\mathrm{PRF}}(D_1) + \mathsf{negl}(n)$, where D_1 is the construction that feeds $c = (x, m_1 \oplus y)$ into $\mathcal{A}$. In all,

$$\mathrm{Adv}(\mathcal{A}) = \mathrm{Adv}_F^{\mathrm{PRF}}(D_0) + \mathsf{negl}(n) + \mathrm{Adv}_F^{\mathrm{PRF}}(D_1) + \mathsf{negl}(n) = \mathsf{negl}(n),$$

where the last equality follows from the assumption of pseudorandomness of F. $\square$

Epic. We have patched the two big problems with OTP: PRGs solve the key length problem and PRFs solve the "one-time" problem. That's it for the course!

Just kidding. Note that in the computation of the ciphertext c, we XOR m with $F_k(x)$. This implicitly constrains the length of the message.

That's not too much of a problem, though. CPA security is designed for safe reuse of keys, so we can just break up the message into components that can be XOR'ed with $F_k(x)$. But doing so naively will blow up the length, since if $\mathcal{X} = \mathcal{Y}$, then c is double the length of m.

A much better idea is to use so-called **modes of operation**.

2.6 Modes of operation

A mode of operation is a way to handle long messages given a cryptographic primitive that only supports fixed-length or short messages.

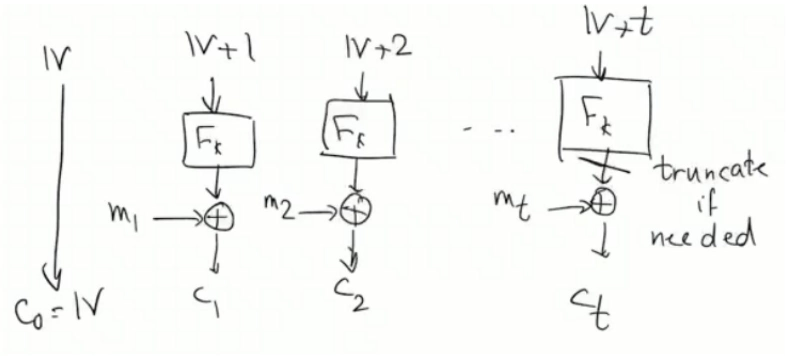
Suppose F is a PRF with output length n. We can define the following encryption scheme:

- $\mathsf{Enc}_k(m)$: break m into chunks $m = m_1 \| m_2 \| \cdots \| m_t$ where $|m_i| = n$.

  Choose random initialization vector $\mathrm{IV} \leftarrow \{0,1\}^n$ and output $c = c_0 \| c_1 \| \cdots \| c_t$ where

  $$c_0 = \mathrm{IV} \qquad c_i = m_i \oplus F_k(\mathrm{IV} + i).$$

  See the following diagram:



  By $\mathrm{IV} + i$ we mean treating IV as an integer, adding i, and modding by 2^n; that is, taking the least significant n bits.

- Dec: Just do the reverse.

Theorem 2.6.1. *If F is a PRF, then CTR with F is CPA-secure.*

Proof. Same idea as last time; observe the proof works as long as different encryptions don't call F_k on the same input more than once. □

CTR mode is great because

1. It's simple
2. It can run in parallel
3. No need to pad the last block

One natural drawback is that you need to make sure IV is really random. Otherwise, after running CTR multiple times with F, you may end up with collisions where you get two IV's that are too close.

For some modes of operation, you need F_k to be invertible.

Definition 2.6.1. *A* ***block cipher*** *is a keyed function $F : \mathcal{K} \times \{0,1\}^n \longrightarrow \{0,1\}^n$ where every $F_k(\cdot)$ is bijective and both $F_k(\cdot)$ and $F_k^{-1}(\cdot)$ are efficiently computable given k.*

Definition 2.6.2. *F is a* ***pseudorandom permutation*** *(PRP) if $F_k(\cdot)$ is a block cipher and for $k \longleftarrow \mathcal{K}$, $F_k(\cdot)$ is indistinguishable from a random bijection. That is, for all p.p.t. $\mathcal{A}$:*

$$\left| \Pr_{k \leftarrow \mathcal{K}}(\mathcal{A}^{F_k(\cdot)} \text{ accepts}) - \Pr_{P \leftarrow \mathcal{P}_n}(\mathcal{A}^{\mathcal{P}(\cdot)} \text{ accepts}) \right| = \mathsf{negl}(n)$$

where $\mathcal{P}$ denotes the set of bijections on $\{0,1\}^n$.

Remark. Sometimes we consider giving $\mathcal{A}$ access to F_k^{-1}, resulting in a "strong PRP."

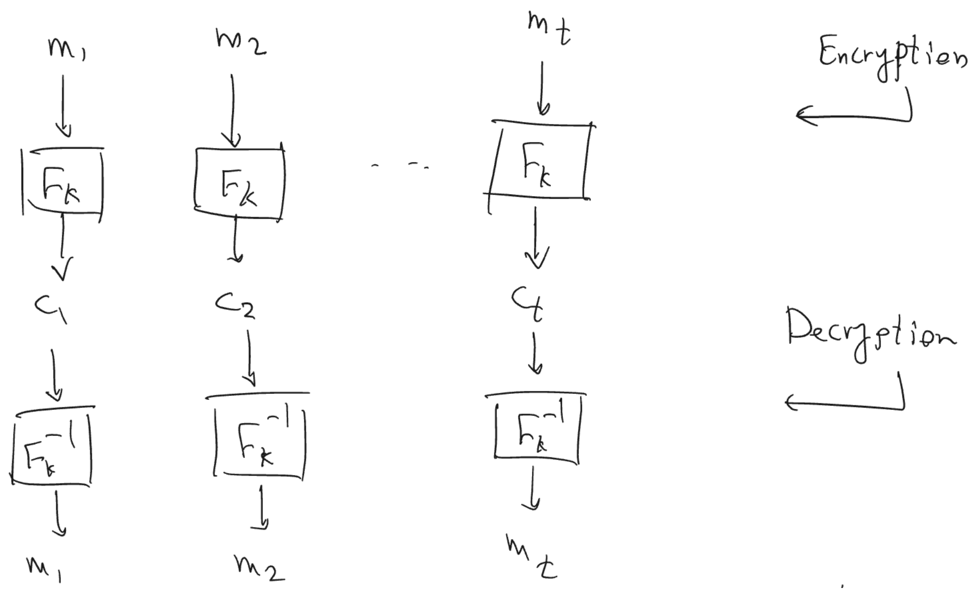
In some sense, a PRP does more than a PRF, which leads us to the following:

Theorem 2.6.2. *Every PRP is a PRF.*

Proof sketch. Given oracle access, a random function and permutation are statistically indistinguishable, provided that different queries to the function don't return the same output. The chance of this is negligible by the Birthday Paradox. □

There are some other modes of operation we will look at:

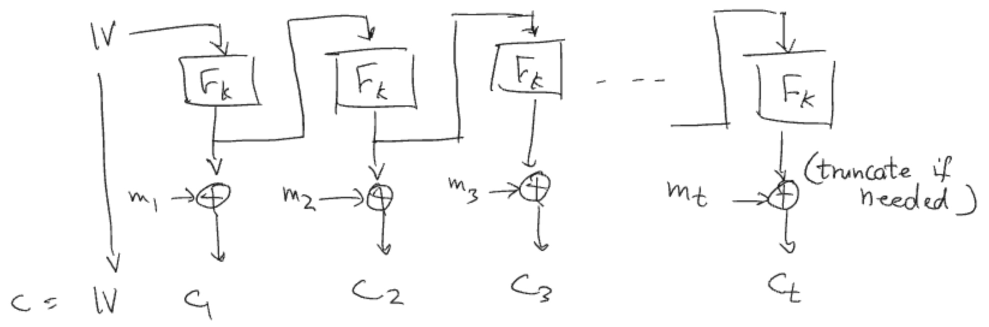
- Electronic Codebook (ECB):



This one sucks. Not only is it not CPA secure, it's not even EAV secure (exercise). Never use ECB in practice.

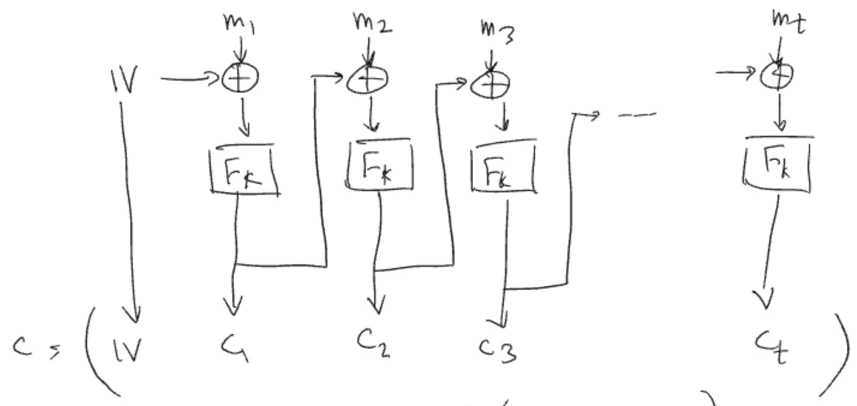
> "I should kill it with fire." - Mahdi

- Ouput Feedback (OFB):



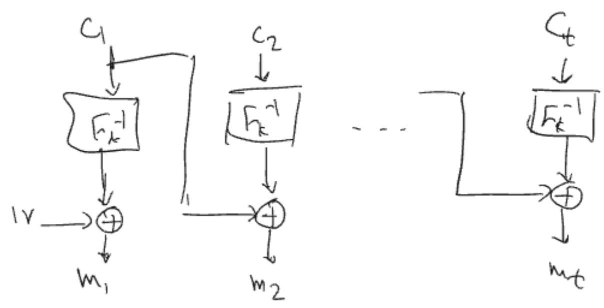
We're taking "chunks of randomness" from F_k; this is, in effect, a stream cipher. We're unlikely to see the same input twice, so it's CPA secure.

- Cipher Block Chain (CBC):



$c_0 = IV, \quad c_i = F_k(c_{i-1} \oplus m_i).$

Decrypt: $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$



Theorem 2.6.3. *If F is a PRP, then CBC is CPA-secure.*

The pros of CBC include:

- If IV has weak randomness, the scheme could still potentially survive in practice.

The cons include:

- Not too simple.
- Encryption can't be parallel (although decryption can).
- You need to pad the last block to length n.
- Can be broken in "streaming models."

Another major con of CBC is the **padding oracle attack**.

Padding is often performed with PKCS#7 padding. If the end of your message requires b bytes to fill the last block, PKCS#7 will append b bytes of value b to make everything whole.

This means that the receiver of a ciphertext coming from a CBC mode can throw an "invalid padding" error if it looks at the ciphertext's last byte b and sees some byte in the last b that doesn't also equal b.

The padding oracle attack exploits **malleability**: the ability of an attacker to modify the ciphertext to a cryptosystem in a way that meaningfully changes the decrypted plaintext. CBC is malleable because changing c_{i-1} changes $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$.

As a result, a receiver that throws an invalid padding error (serves as a "padding oracle") in a CBC is prone to the padding oracle attack as follows:

Part 1: *Recover the length of the padding.*

Let $c = c_1 \| c_2 \| \cdots \| c_l$. Change the first byte of c_{l-1}, which changes the first byte of m_l, and send the modified c to the oracle $\mathcal{O}$.

If an error occurs, we know we changed a byte (the first in the last block) that was part of the padding. We conclude that m_l consists entirely of padding. If no error occurs, the padding stayed intact despite our change, so change the next byte until we get an error. Once that happens, we know we have modified (hence, found) the first byte of padding.

Denote the distance of this byte to the end of the message by b.

Part 2: *Recover the last block.*

We know the last b bytes have value b; we'll try to guess the last character before the padding. Pick a "trial byte" t', a value used to determine the plaintext byte t through oracle feedback. Let's query $\mathcal{O}$ on c, replacing c_{l-1} with c'_{l-1}, where

$$c'_{l-1} = c_{l-1} \oplus (00 \ldots 0t'\tilde{b}\tilde{b} \ldots \tilde{b}) \qquad \tilde{b} = b \oplus (b+1).$$

So we replace the last non-padding byte with our guess t' and each padding byte with $b \oplus (b + 1)$.

Why? Decrypting the modified c gives

$$m'_l := m_l \oplus (00 \ldots 0t'\tilde{b}\tilde{b} \ldots \tilde{b}) = F_k^{-1}(c_i) \oplus c'_{l-1}.$$

Recall that the last b bytes of m_l are b. They evaluate to $b \oplus b \oplus (b+1) = b+1$ by definition of $\tilde{b}$. Every other byte of m_l, except the last non-padding byte (call it t), remains the same. That one becomes $t \oplus t'$.

Now recall the PKCS#7 invariant: the last byte's value is the number of suffix bytes with that value. So if m'_l has valid padding, its last $b+1$ bytes must be $b+1$, including $t \oplus t'$. Therefore, we can validate our guess t' based on whether $\mathcal{O}$ gives a padding error. If it doesn't, then

$$t \oplus t' = b + 1 \Longrightarrow t = (b+1) \oplus t'.$$

So we recover t. Of course, this strategy is not specific to $b+1$. We continue with $b+2, b+3, \ldots$, which gives us the bytes before t and, therefore, the rest of block l.

Part 3: *Recover the rest of the blocks.*

Recall that b is at most the length of one block. To recover block $l-1$, then, drop the last ciphertext block and change the last byte of c_{l-1} until there is no padding error.

Now that our ciphertext properly decrypts, repeat Part 1 to find the new padding of m_{l-1}, and repeat Part 2 to recover its bytes.

Repeat this until every block of the ciphertext is recovered.

3 Integrity

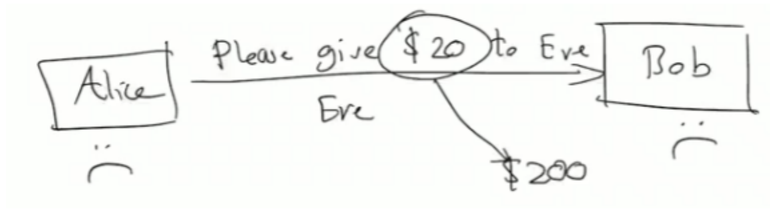
3.1 Message authentication codes

Now we'll turn our attention away from confidentiality and focus on the problem that enables the padding oracle attack: integrity.

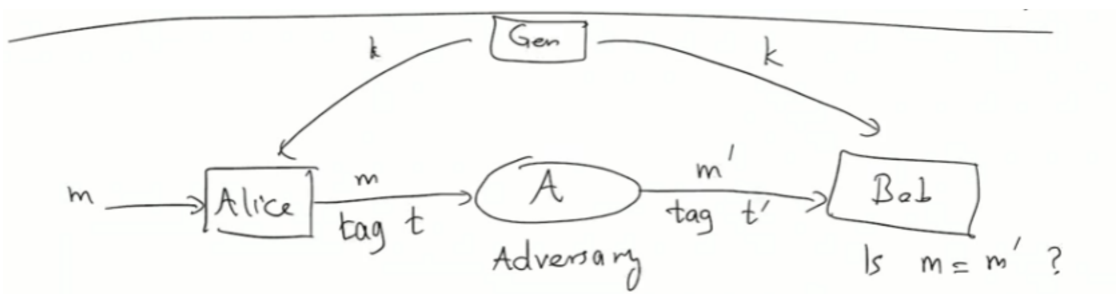
In confidentiality, our concern is to make sure an adversary who can read our communication doesn't learn about secrets.

In **integrity**, our concern is to make sure an adversary who manipulates our communication won't cause us to unknowingly decrypt the wrong message.

There's a pretty immediate example of the importance of integrity:



CBC is overkill to show that CPA does not guarantee integrity. Even OTP is enough: if we change c to $c' := c \oplus 00\ldots010\ldots0$, then $k \oplus c'$ has the corresponding bit flipped. Needless to say, integrity is important in applications.



Here's the attack model we're working with now. The adversary can intercept the message m and replace it with m', but we would like to make it impossible to replicate the tag t without k. In this system, Bob would know whether m has been tampered with.

This is called a **Message Authentication Code** (MAC). We write $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ for message space M, key space K, and tag space T, where

- $\mathsf{Gen}(1^n)$: Outputs a key $k \longleftarrow K$.
- $\mathsf{Tag}_k(m)$: Given a message m and key k, outputs a tag $t \longleftarrow T$.
- $\mathsf{Ver}_k(m', t')$: Accept if t' is a valid key for m' given key k; reject otherwise.

Like CPA-security, a MAC must satisfy correctness and security:

- For $m \in M$ and $k \in K$, we have $\mathsf{Ver}_k(m, \mathsf{Tag}_k(m))$ accepts.
- Ver should reject if m is manipulated.

To formalize security, we introduce the chosen message attack (CMA) game against a MAC given by $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$. The adversary is called a forger, denoted $\mathcal{F}$, and does the following:

1. $\mathcal{F}$ receives the security parameter 1^n.
2. The game generates a key $k \longleftarrow \mathsf{Gen}(1^n)$.
3. $\mathcal{F}$ queries the oracle $\mathsf{Tag}_k(\cdot)$ on at most $\text{poly}(n)$ messages to get tags for them.
4. $\mathcal{F}$ attempts a forgery by outputting (m^*, t^*).

$\mathcal{F}$ wins if $\mathsf{Ver}_k(m^*, t^*)$ accepts and m^* was not previously queried.

Definition 3.1.1. *A MAC* $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ *is* ***unforgeable under chosen message attack*** *(UF-CMA) if for every p.p.t. forger* $\mathcal{F}$*,*

$$\Pr_{k \leftarrow \mathsf{Gen}}(\mathcal{F}^{\mathsf{Tag}_k(\cdot)} \text{ wins}) = \mathsf{negl}(n).$$

We might want to propose the following MAC using a PRF:

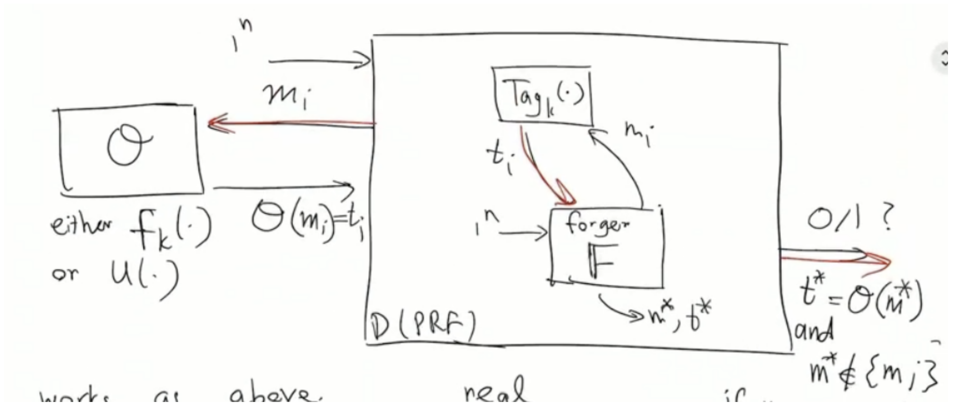
Take a PRF $f : \mathcal{K} \times \{0,1\}^\ell \longrightarrow \{0,1\}^n$. Then define

- $\mathsf{Gen}(1^n)$: Choose $k \longleftarrow \mathcal{K}$.
- $\mathsf{Tag}_k(m \in \{0,1\}^\ell)$: Output $f_k(m)$.
- $\mathsf{Ver}_k(m', t')$: Accept if $t' = f_k(m')$.

If Tag is a deterministic function of (k, m), then there exists a "canonical verifier" Ver that is trivial: just check if $t' = \mathsf{Tag}_k(m')$.

Theorem 3.1.1. *This is UF-CMA if f is a PRF.*

Proof. Consider the following reduction:



D accepts if the two conditions in the bottom right hold, and rejects rejects otherwise. To analyze the advantage of D, first consider the probability it accepts in the real world. Then, by construction, we have simulated the CMA game, against which $\mathcal{F}$ has non-negligible advantage.

Conversely, if we are in the ideal world, $\mathcal{F}$ cannot do better than random guessing. So it has a $1/|T| - \mathsf{negl}(n)$ chance of guessing it correctly as $\mathcal{U}(m^*)$. We conclude that

$$\text{Adv}(\mathcal{F}) = \text{Adv}(D) \pm \mathsf{negl}(n).$$

□

The notion of CMA security does not protect against repetition attacks. If Alice sends \$20 to Bob and Eve sees that, there's nothing stopping her from repeating that communication and causing the transaction to happen again.

To solve this, we add some source of "freshness" to the message (e.g., timestamp, sequence number, etc.).

Another issue with our definition of CMA security is that there is nothing stopping an adversary from making a new tag for an old message. To address this, we redefine the winning criteria for the CMA game as:

- $\mathsf{Ver}_k(m^*, t^*)$ accepts
- (m^*, t^*) has never been seen

Note that this definition of security does not prevent m^* from being a previous message. This is not a problem if the tag function is deterministic.

One final remark is that we can also give the forger access to the verification function, but again this is not a concern for deterministic Tag because then Tag and Ver are essentially the same thing.

Now how do we handle arbitrarily long messages?

We can start with a familiar approach where we split up the message and generate a tag for each length-ℓ block. But we have to be careful.

Given $m = m_1 \| m_2 \| \cdots \| m_s$, say we define $\mathsf{Tag}_k(m)$ as $t = (F_k(m_1), F_k(m_2), \dots, F_k(m_s))$. This is vulnerable to the so-called reordering attack, where you can create a valid tag for a new message by scrambling the blocks. For instance, an adversary that sees $t = (F_k(m_1), F_k(m_2))$ for $m = m_1 \| m_2$ can create a valid tag $t' = (F_k(m_2), F_k(m_1))$ for $m' = m_2 \| m_1$.

To prevent the ordering attack, bind each block to its position in the sequence. That is, write $t_i = F_k(m_i \| i)$. Unfortunately, this is also insecure for many reasons:

- Truncation attack: query $m_1 \| m_2 \implies t_1 \| t_2$ and forge $m_1 \implies t_1$.
- Mix-and-match idea: query $m_1 \| m_2 \implies t_1 \| t_2$ and $m_1' \| m_2' \implies t_1' \| t_2'$, and forge $m_1 \| m_2' \implies t_1 \| t_2'$.

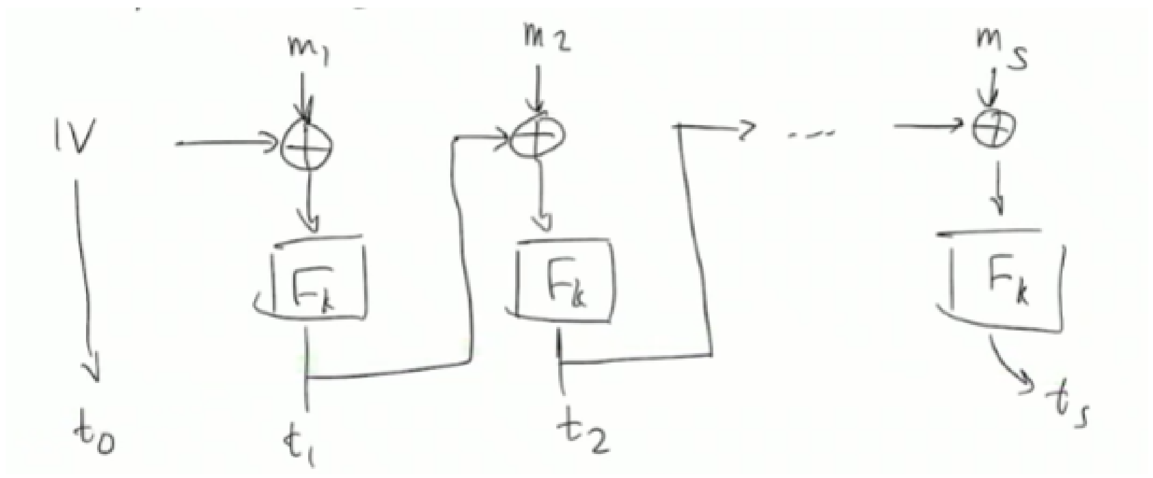
To prevent truncation, we can also include the number of blocks or the total bit length $|m|$. This doesn't do anything about mix-and-match, however.

We can then try to give each request a unique identifier $r \longleftarrow \{0,1\}^{\ell/4}$. So our tags look like:

$$t_i = F_k(m_i \| r \| i \| |m|).$$

And this is, in fact, CMA-secure. The proof in the book is basically a big case analysis.

Why not use modes of operation? The above is extremely wasteful—the tag length is huge. Inspired by CBC, we come up with CBC-MAC. We won't use CTR because it processes each block individually, and that just runs into our construction from earlier.

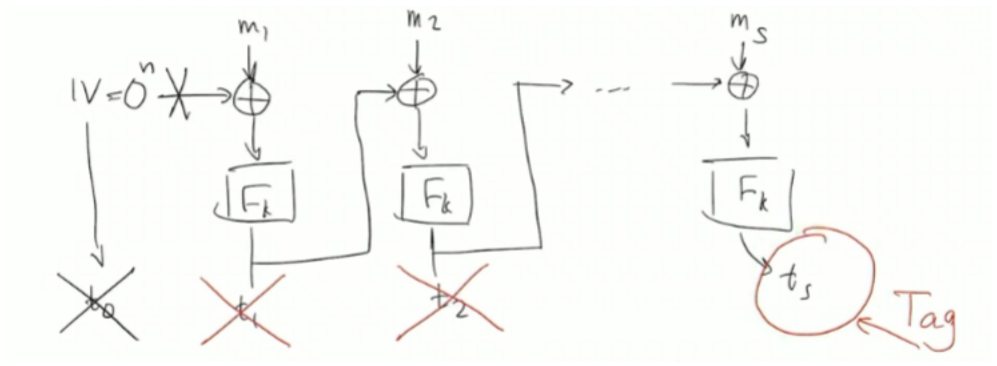


Sadly, this is really easy to forge. An adversary can pick an existing t_0 and XOR it with some string t'. Then compute $m_1 \oplus t'$ and all the remaining tag blocks stay the same, so we obtain a new tag for this new message with modified m_1.

But recall that MAC tags don't need to be deterministic. So let's fix $IV = 0^n$. This does indeed fix that problem.

Except somehow this is worse, because now we know that t_1 is precisely $F_k(m_1)$, so we essentially have complete access to F_k and we can generate any tag we want. To add insult to injury, this approach is still vulnerable to the truncation attack, *and the tag is still linear in the message size.*

The following design fixes all of these problems at once:



Theorem 3.1.2. *If F is a PRF, then CBC-MAC is UF-CMA-secure for messages of length exactly s blocks.*

Proof. See book (it's messy). □

Note that if s isn't fixed-length, the adversary can perform length extension:

- Query one block consisting of m to get $t = F_k(m)$ (this was a problem earlier, too)
- Query $m' = t \oplus \hat{m}$ and get $t' = F_k(t \oplus \hat{m})$
- Forge $m^* = m \| \hat{m}$ and get tag $t^* = t'$

We're taking advantage of the fact that the tag from the prefix of a string (XORed with the next message block) is precisely the input to the next tag block's computation.

In other words, the forger is able to do this when a message is a prefix of another. To fix this, we can use **prefix-free encoding**, which is a way to represent messages so that none are prefixes of each other.

How do we construct prefix-free encoding? Simply encode m as $(|m| \| m)$.