With a pseudorandom generator in hand, we have overcome the main drawback of the one-time pad. Alice and Bob now have an effectively interminable stream of pseudorandom bits at the cost of having shared a few hundred bits of key. How should they use this stream to encrypt data?

One possibility is for Alice and Bob to keep a counter that records where the fresh portion of the pseudorandom stream begins. For example, if the counter has value 56, it means that bits 0 to 55 were already used. Suppose she now wants to send message `love`. Here we'll represent each letter with its ASCII encoding

$$m = \texttt{love} = \texttt{01101100 01101111 01110110 01100101}.$$

The ciphertext c should then equal m plus with the portion of the pseudorandom stream between positions 56 and $56 + 32 = 88$. After sending c she would reset her counter to 88. Bob would perform the same sequence on his side to recover the message.

Is this scheme secure? If the stream is pseudorandom, Eve cannot distinguish an encryption of `love` that starts at position 56 from an encryption of `hate` that starts at the same position. She can, however, distinguish an encryption of `buy` from one of `sell`. The silly reason is that the former's ciphertext is 24 bits long, while the latter's is 32 bits long.

If there is no a priori limit on the message length, some two ciphertexts will always be of differing length, and Eve can distinguish the two. A practical solution to this problem is to insist that messages are always encrypted in blocks of the same length. This can be ensured either by padding the data (e.g., replacing `buy` with `buy*`) or waiting for enough data to become available to fill in a block.

One drawback of stateful encryption is the need to synchronize. Alice has to ensure that her state accurately reflects the unused portion of the stream at all times. If Alice represents a distributed entity like a collection of sensors that share the same secret key, coordinating among these sensors might be technically challenging.

Another drawback of this mode of encryption is that encrypted data can only be accessed in sequence. Suppose that Alice wants to store her data on a server run by Bob that she has privacy concerns about. Symmetric-key encryption provides a solution to this problem: Bob receives an encryption of the data, but Alice never gives away her secret key. What happens when Alice wants to retrieve an item? She can certainly ask Bob to send her back the data and decrypt it with her secret key. This is, however, wasteful: If Alice wants to retrieve the thousandth entry in the database, she has to run her PRG for 1000 steps to make sense of it.

The source of these problems is that when a pseudorandom generator is applied to produce a stream of bits, it has to be invoked sequentially. There is a different type of sampler that provides chosen access[1] to the pseudorandom sequence. The 938-th entry of the sequence can be calculated quickly from the key without having to go over the previous 937 items.

1 Pseudorandom functions

A pseudorandom generator was a sampler that produced an output indistinguishable from a pure random sequence. A pseudorandom function is an "implicit" sampler that computes a function that is indistinguishable from a pure random function.

What is a pure random function? Let's start with a function that maps a 3-bit input into a 1-bit output. Such a function can be described by listing its $2^3 = 8$ possible values as in Table 1.

[1] This is commonly referred to as *random access* in computer architecture. To avoid confusion in the context of cryptography I prefer to call it "chosen access" or "query access".

TABLE 1: A function from 3 bits to 1 bit.

x	000	001	010	011	100	101	110	111
$f(x)$	1	0	1	1	1	0	1	0

We can think of this list of 8 values, i.e., the sequence 10111010, as a random variable. There are $2^8 = 2^{2^3}$ possible such sequences, each describing a different function. In general, a function that maps an n-bit input into a 1-bit output can be described by listing its 2^n values. There are 2^{2^n} such functions. A random function R is a random variable that is chosen uniformly among these 2^{2^n} possibilities.

A pure random function R can be sampled by picking the 2^3 values $R(000), R(001), \ldots, R(111)$ as uniform and independent random bits. This gives a natural implementation of R by an *implicit sampler*.

An implicit sampler is one that works in two stages. In the setup stage, it takes in some pure randomness, and stores it. From then on, the only randomness that is available to it is whatever it has stored. All remaining calculations have to be deterministic. In the evaluation stage, the sampler takes an input sequence x and outputs a function value $F(x)$.

Here is an implicit sampler for a pure random function on 3 bits:

Setup stage: Sample and store eight random bits $r_0, \ldots, r_7$.
Evaluation stage: On input x, output the value $r_{\text{int}(x)}$, where $\text{int}(x)$ is the integer representation of x.

For example, if $r_0 \cdots r_7$ is the sequence 10111010 then this program samples the function in Table 1.

This type of implementation scales exponentially in the length n of the input sequence. An implementation of a function with n-bit inputs entails storing 2^n values. Could there be a more clever implementation that avoids storing so many values? This is not possible. Storing even one fewer bit results in a function that is distinguishable from random. A pseudorandom function is a function F that is indistinguishable from random, but admits an efficient implicit sampler with reasonable storage.

What does it mean for a function to be indistinguishable from random? Since a function is completely described by listing its values, we would like to say that the bit sequences

$$(F(00\cdots0), F(00\cdots1), \ldots, F(11\cdots1)) \quad \text{and} \quad (R(00\cdots0), R(00\cdots1), \ldots, R(11\cdots1))$$

are indistinguishable with reasonable effort. These lists are exponentially long, so merely reading them would take uneasonable effort.

To make sense of this idea we should allow Eve *query access* to the function. She should be able to submit any input of her choice and observe the corresponding output. For example, she could ask to see $F(001)$ and see the value 1. She could then ask to see $F(010)$ and see the value 0. In a pseudorandom function, these values should be indistinguishable from $R(001)$ and $R(010)$. That is, they should look like a pair of independent random bits.

Implementing a random function on n input bits by an implicit sampler entailed storing 2^n random values. The aim is to implement a pseudorandom function by storing a much shorter random *key* in the setup stage, yet maintaining the appearance that any values Eve chooses to observe look as if they were the values of a truly random function, i.e., independent random values.

At this point it is helpful to see some non-examples. Before we do that let us say a few words about the *output* length of a (pseudo)random function. A random function requires storage exponential the length of its input. In contrast, the length of its output is largely irrelevant. The reason is that ℓ bits of input can be "traded off" for 2^ℓ bits of output. For example, the function in Table 1 can be converted into one with 2 input bits and 2 output bits by forgetting the last input bit and concatenating the corresponding outputs:

x	00	01	10	11
$g(x)$	10	01	10	10

For these reasons, from now on we won't worry about the output length of a pseudorandom function.

In the following examples both the inputs and the outputs are 3-bits long.

Example 1. $F(x) = \text{int}(x)^{\text{int}(x)} \bmod 8$ in binary representation. F is not pseudorandom because it is deterministic. To distinguish it from random, Eve asks to see the value at 000, and accepts if this value equals $0^0 \bmod 8 = 0$. This happens with probability 1 for F, but only with probability 1/8 for a random function. Thus Eve can distinguish the two with advantage 7/8.

The reason F is not pseudorandom has nothing to do with its specification. A pseudorandom function must be randomized. Every deterministic function is distinguishable from a random function.

Example 2. The function F implemented as:

> **Setup stage:** Sample and store a random 3-bit key k.
> **Evaluation stage:** On input x, ignore x and output k.

The value $F(000)$ equals the random key k. It is perfectly indistinguishable from $R(000)$. For the same reason, $F(x)$ and $R(x)$ are indistinguishable at every input x. Yet F is not pseudorandom. To understand why let's expand it out:

x	000	001	010	011	100	101	110	111
$F(x)$	k	k	k	k	k	k	k	k

The values of F are not independent. Eve can distinguish F from a random function by checking that the values at, say, inputs 000 and 001 match. This happens with probability 1 for F but with probability only 1/8 for a random function. The two can be distinguished with advantage at least 7/8.

Example 3. The function F implemented as:

> **Setup stage:** Sample and store a 3-bit random key k.
> **Evaluation stage:** On input x, output the bitwise sum $x + k$ modulo 2.

The value $F(x)$ is again uniformly random at every input k. Yet F is again not pseudorandom. One reason is that the modulo 2 sum $F(000) + F(001)$ equals $(000 + k) + (001 + k) = 001$. Eve can test and accept if this is the case. F passes the test with probability one. A random function passes it with probability 1/8 only.

2 How to construct a pseudorandom function

There are infinitely many ways in which a function can fail to be pseudorandom. So how should one go about building one? Remarkably, there is a systematic construction that guarantees pseudorandomness. All we need is a *length-doubling* pseudorandom generator to get started.

A pseudorandom generator G that takes ℓ bits of seed and produces 2ℓ bits of output is called length-doubling. The starting point is that G can be viewed as a rudimentary pseudorandom function F_1—one that takes a single bit of input and produces ℓ bits of output.

Setup stage: Sample and store an ℓ-bit seed k for G.
Evaluation stage: On input x,
1 If $x = 0$, output the first ℓ bits of $G(k)$.
2 If $x = 1$, output the last ℓ bits of $G(k)$.

To be concrete, suppose $\ell = 3$ and G is the length-doubling function:

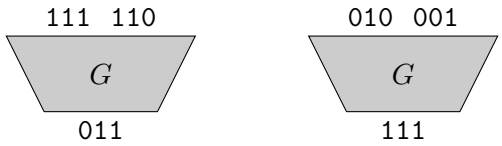
k	000	001	010	011	100	101	110	111
$G(k)$	000011	111110	110000	010010	111110	110011	011111	010001

G is clearly not a pseudorandom generator as ℓ is far too small. It will be useful to pretend that it is. If the key happened to be $k = \texttt{110}$, F_1 would be the function

x	0	1
$F_1(x)$	011	111

The reason F_1 is pseudorandom is that the output of G is indistinguishable from a pure random 6-bit string. Therefore its left and right halves are indistinguishable from independent random 3-bit strings. So will be the two values of F_1.

Now comes the main idea of the construction. The values $F_1(\texttt{0})$ and $F_1(\texttt{1})$ can be used as seeds for another application of G. As $F_1(0)$ and $F_1(1)$ are indistinguishable from independent random sequences, the outputs of G should not introduce any apparent dependencies:



We now have four values that are indistinguishable from independent random strings. The two on the left arose from applying G on seed $F_1(\texttt{0}) = \texttt{011}$. Let's name those $F_2(\texttt{0}\underline{\texttt{0}}) = \texttt{111}$ and $F_2(\texttt{0}\underline{\texttt{1}}) = \texttt{110}$. Similarly, the two values on the right arose by applying G on $F_1(\texttt{1}) = \texttt{111}$. We'll name them $F_2(\texttt{1}\underline{\texttt{1}}) = \texttt{010}$ and $F_2(\texttt{1}\underline{\texttt{1}}) = \texttt{001}$. F_2 is now a pseudorandom function that takes two bits of input.

We can iterate to extend the input as much as we want, one bit at a time. Suppose we have managed to construct a pseudorandom function F_{n-1} that takes $n-1$ input bits $x_1 \dots x_{n-1}$. The function

$$F_n(x_1 \dots x_n) = \begin{cases} \text{first } \ell \text{ bits of } G(F_{n-1}(x_1 \dots x_{n-1})), & \text{if } x_n = 0 \\ \text{last } \ell \text{ bits of } G(F_{n-1}(x_1 \dots x_{n-1})), & \text{if } x_n = 1 \end{cases} \tag{1}$$

is now a pseudorandom function that takes n input bits.

A judicious piece of notation helps greatly demystify the calculation of F_n: Call G_0 the first ℓ bits of G and G_1 the last ℓ bits of G. Thus $G(k)$ equals the pair of values $(G_0(k), G_1(k))$. As G is length-doubling, G_0 and G_1 are length-preserving. In this notation, formula (1) becomes

$$F_n(x_1 \dots x_n) = G_{x_n}(F_{n-1}(x_1 \dots x_{n-1})).$$

We can now "unwind" F_{n-1} to obtain

$$F_n(x_1 \dots x_n) = G_{x_n}(G_{x_{n-1}}(F_{n-2}(x_1 \dots x_{n-2})))$$

all the way down to

$$F_n(x_1 \dots x_n) = G_{x_n}(G_{x_{n-1}}(\cdots G_{x_2}(F_1(x_1) \cdots)).$$

but $F_1(x_1)$ is simply the correct half of G evaluated on the seed k, or $G_{x_1}(k)$. Finally,

The Goldreich-Goldwasser-Micali (GGM) pseudorandom function:
Setup stage: Sample and store a key k for pseudorandom *generator* G.
Evaluation stage: On input $x = x_1 x_2 \cdots x_n$, output the value

$$\boxed{F(x_1 x_2 \dots x_n) = G_{x_n}(G_{x_{n-1}}(\cdots G_{x_1}(k) \cdots))}$$

For instance, if $k = \texttt{110}$ as in our running example, then

$$F(\texttt{0010}) = G_0(G_1(G_0(G_0(\texttt{110})))) = G_0(G_1(G_0(\texttt{011}))) = G_0(G_1(\texttt{010})) = G_0(\texttt{000}) = \texttt{000}.$$

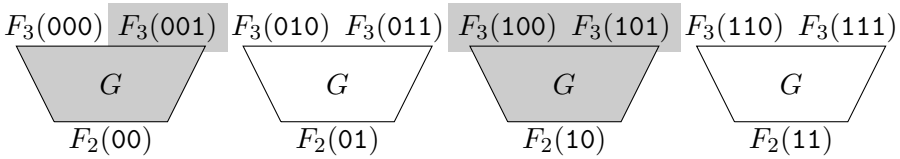


FIGURE 1: Security of GGM. Eve queries F_3 at inputs 001, 100, and 101.

To understand the security of the GGM pseudorandom function, it is helpful to expand the two sides of (1) as in Figure 1. The table of values that F_3 takes is derived from that of F_2 by applying the PRG G to each output. As F_2 is indistinguishable from random, the PRG seeds are indistinguishable from independent random strings. If Eve queries, say, the values of F_3 at inputs 001, 100, and 101, she would be effectively observing evaluations of two independent copies of G (highlighted in grey). Eve's advantage in distinguishing two outputs of G from random is at most double that for a single copy. It is still extremely small. You will investigate the distinguishing advantage of single and multiple pseudorandom generator outputs in Homework 2.

FIGURE 2: Oded Goldreich. (From *CityNews*.)

3 The pseudorandom function pad

With the help of a pseudorandom function (PRF), Alice and Bob can encrypt and decrypt without having to synchronize a counter.

The Pseudorandom Function Pad (PRF Pad):

Shared key: The key k used to set up PRF F.

Encryption: To encrypt a r-bit message m, Alice samples a purely random n-bit pointer x. She sends the *pair* $(x, m + F(x))$ as the ciphertext. (As always, addition is modulo 2.)

Decryption: Upon receiving ciphertext pair (x, c), Bob recovers $F(x) + c$ as the message.

In a typical application, n might be on the order of 1000 and r might be on the order of several thousands. To understand how the scheme works, however, let's take $n = 4$ and $r = 3$. Suppose F is the GGM PRF that we just saw and the shared key is $k = 110$.

If Alice wants to encrypt the message $m = 101$, she samples a pointer x, say $x = 0010$, and sends out the pair

$$(x, c) = (x, m + F(x)) = (0010, 101 + F(0010)) = (0010, 101 + 000) = (0010, 101).$$

(In this instantiation, m and c are equal, but that is just because $F(x)$ happened to evaluate to zero by chance.) To decrpypt the pair $(x = 0010, c = 101)$, Bob undoes the operations to recover the message $m = 101$.

What did Eve learn by observing this interaction? Eve saw a random value $x = \texttt{0010}$ of the pointer, together with $c = m + F(x)$. From Eve's perspective, $F(\texttt{0010})$ is indistinguishable from value of a random function evaluated at input `0010`. This is a random 3-bit sequence that is independent of x. Shifting the message m by a random 3-bit sequence yields a c that is not only purely random, but also independent of x.

Eve sees a pair (x, c) that is indistinguishable from seven purely random bits. Had Alice encrypted $m' = \texttt{100}$, the effect would have been the same. Encryptions of m and m' are indistinguishable to Eve.

What happens when Alice now encrypts another message m^*, say $m^* = \texttt{001}$? Its encryption may look like

$$(x^*, c^* = m^* + F(x^*)) = (\texttt{1011}, \texttt{001} + F(\texttt{1011})) = (\texttt{1011}, \texttt{001} + \texttt{111}) = (\texttt{1011}, \texttt{110}).$$

By the same logic, the sequence (x^*, c^*) looks like seven independent random bits to Eve. Not only that, but these seven bits also appear independent from the encryption (x, c) of m. The reason is that the pointers x and x^* are not only independent but also happen to be *distinct*. If F were a truly random function, $F(x)$ and $F(x^*)$ would uniform and independent random values (when conditioned on x and x^*). Even after shifting them by the corresponding messages m and m', the uniformity and independence of these values is preserved.

Now F is not truly random but merely pseudorandom. From Eve's perspective, the joint sequences $(x, c), (x^*, c^*)$ still look indistinguishable from 14 pure random bits.

What would have happened if, by chance, the pointers x and x^* happened to not be distinct? In such a scenario Eve would gain considerable advantage. She would know that m and m^* were shifted by the same phase $F(x)$ and would be able to recover their sum by adding up the ciphertexts:

$$c + c^* = (m + F(x)) + (m^* + F(x^*)) = (m + F(x)) + (m^* + F(x^*)) = m + m^*.$$

Leaking the sum of two messages does not meet our stringent standards of cryptographic security.

But what are the chances that x and x^* collide? They were both chosen at random among all 8-possible 3-bit pointers. The probability that x equals x^* is $1/8$. This probability of failure is rather high, but it has to do with our PRF F taking in only a meagre 3 bits of input. If x and x^* were 256-bits long, the probability of a collision $x = x^*$ drops to a tiny $1/2^{256}$. This is an acceptable security risk.

More generally, suppose Alice uses the PRF Pad with pointers of length n to send t messages to Bob. All t pointers will then be distinct except with probability at most $\binom{t}{2} \cdot 2^{-n}$. (We will explain why later in the course.) When, e.g., $n = 256$ and t equals a million, this exceptional event is still extremely unlikely. Assuming it doesn't happen, the Pseudoradom Function Pad encryption is as secure as the underlying PRF.

Now that we understand why the PRF Pad is secure, it is a good time to reflect on its compatibility with the requirements of secure encryption. The PRF Pad is randomized as necessary. Unlike in the one-time pad, however, the randomness here is used for two purposes. The random choice of a shared key k prevents Eve from telling F apart from a random function. The random choice of a pointer x, on the other hand, ensures that different instantiations of encryption use non-overlapping (pseudo)random masks, except in the very unlikely event of a collision.

One consequence of randomizing every instantiation of the encryption algorithm is that even when the same message is encrypted twice, the ciphertexts look independent! This is not a "bug" of the pseudorandom pad. It is a necessity if we want encryptions of multiple messages to be distinguishable. Eve should not be able to tell if Alice sent out two encrypted `loves` from encrypted `love` followed by encrypted `hate`.

Randomness is but one mechanism that ensures the joint indistinguishability of multiple encryptions. Any nonrepeating bit sequence, or *nonce*, will do. Suppose Alice's data A consists of a large list of items. To store her x-th item $A(x)$ securely on Bob's server, Alice encrypts it as $F(x) + A(x)$. She can then retreive any item of her choice without having to look up the others.

4 Chosen plaintext attacks

In our modelling of security so far we have thought of Eve as a *passive* adversary. Eve can observe all communication from Alice to Bob, but she cannot influence it. There are situations in which Eve may be able to affect in part what Alice wants to tell Bob next.

The Enigma machine was a formidable mechanical device that the Germans used to encrypt their communications in World War II. Despite its intricate design, Polish and British cryptographers eventually managed to crack it. One of their strategies was to create situations that would trigger a particular encryption:[2]

> Occasionally, when there was a particularly urgent need to break German naval Enigma, such as when an Arctic convoy was about to depart, mines would be laid by the RAF [Royal Air Force] in a defined position, whose grid reference in the German naval system did not contain any of the words (such as sechs or sieben) for which abbreviations or alternatives were sometimes used. The warning message about the mines and then the "all clear" message, would be transmitted both using the dockyard cipher and the U-boat Enigma network. *(From Wikipedia)*

The British knew, with reasonable certainty, what the Germans would be encrypting in response to their action. For instance, if their mine was laid at coordinates `43.27N 31.12W`, this information would likely be featured in the plaintext. Eve has the power to choose which plaintext Alice will encrypt next!

The *Chosen Plaintext Attack* is a security model that allows Eve to completely choose the messages encrypted by Alice. After observing enough encryptions to her satisfaction, Eve then asks Alice one of two challenge encryptions of her choice, just in the message indistinguishability game. If Eve can tell the two apart with high enough advantage, she wins.

The PRF Pad is secure against a Chosen Plantext Attack. Even if Eve gets to choose the messages encrypted by Alice, all she ends up observing are shifts of these messages by values of the PRF F evaluated at random pointers, together with those pointers. As long as all pointers are distinct, Eve has learned nothing by observing the encryptions. To her they are indistinguishable from independent random bit sequences.

5 Woman-in-the-middle attacks

There are other ways in which Eve can actively interfere with communications from Alice to Bob. A particularly nefarious one is the woman-in-the-middle attack.

Imagine that Alice controls the internet router that directs the messages from Alice to Bob. If the messages are encrypted, Eve cannot read them. She can, however *maul* them: take a ciphertext c that Alice intended for Bob and turn it into another ciphertext c', all without seeing the message, or having any idea about what the secret key is (see Figure 3).

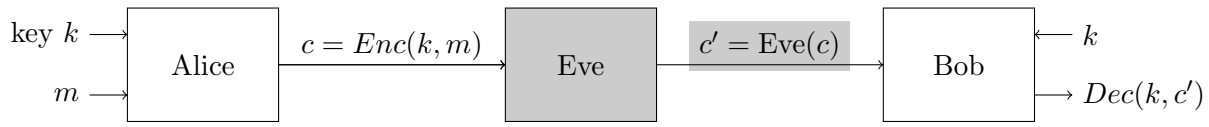


FIGURE 3: Woman-in-the-middle attack.

The PRF Pad is susceptible to woman-in-the-middle attacks. Here is how one would play out.

Suppose Eve knows that Alice is encrypting either `love` or `hate` to Bob. Alice thinks she has intercepted an encryption a declaration of $m =$ `love`. She wants to turn into `hate`.

$$(x, c) = (x, m + F(x)) = (\texttt{010010}, \texttt{love} + F(\texttt{010010})).$$

[2]The Katz-Lindell textbook has another example of this kind from the Pacific Theatre.

To turn `love` into `hate`, all she has to do is to subtract `love` and then add `hate` to c. The modulo 2 difference `love` + `hate` is the string

$$\begin{aligned}\Delta &= \texttt{love} + \texttt{hate} \\ &= \texttt{01101100 01101111 01110110 01100101} + \texttt{01101000 01100001 01110100 01100101} \\ &= \texttt{00000100 00001110 00000010 00000000}.\end{aligned}$$

To mount her woman-in-the middle attack, Eve intercepts the ciphertext (x, c), replaces it with $(x, c') = (x, c + \Delta)$, and forwards the result to Bob. The new ciphertext is

$$(x, c') = (x, c + \Delta) = (x, (\texttt{love} + F(x)) + \Delta) = (x, (\texttt{love} + \Delta) + F(x)) = (x, \texttt{hate} + F(x)).$$

Eve has turned an encryption of `love` into an encryption of `hate`.

This specific attack, and malleability of ciphertexts in general, can be prevented using *message authentication codes*. This will be the topic of Lecture 7.

6 Summary

Pseudorandom generators solve the problem of private key encryption and allow us to worry about other luxuries: How about stateless encryption? What if Eve can influence the messages being encrypted? Both of these features can be realized with a pseudorandom function: A program with a short random key that, when evaluated on arbitrary inputs of Eve's choosing, appears to be producing random and independent outputs.

Pseudorandom functions (PRFs) are tricky to build from scratch. Goldreich, Goldwasser, and Micali showed a principled construction using a length-doubling pseudorandom generator as the only building block.

The PRF Pad is a randomized encryption scheme that is stateless and secure even if Eve observes multiple messages. The only effective attack available to Eve is to hope for a collision between the random pointers. The PRF Pad remains secure even if Eve can choose the messages whose encryptions she gets to observe. It is, however, not immune to mauling ciphertexts: An Eve-in-the-middle can turn en encryption of `love` into an encryption of `hate` without knowing the key.