1 One-Time Pad & Kerckhoffs' Principle

You can't learn about cryptography without meeting Alice, Bob, and Eve. This chapter is about the classic problem of **private communication**, in which Alice has a message that she wants to convey to Bob, while also keeping the contents of the message hidden from an eavesdropper[1] Eve. As you will learn, there is more to cryptography than just private communication, but it is the logical place to start. After all, the word *cryptography* means "hidden writing" in Greek.

1.1 What Is [Not] Cryptography?

There's more to security than just cryptography. Every security problem you care about probably involves both computers and humans, both of which are relatively complex systems. Having a security mindset means worrying about adversarial behavior in *all parts* of those complex systems.

Let's take a closer look at our motivating scenario featuring Alice, Bob, and Eve, and establish some clear boundaries about what part of the problem we are actually trying to solve in this course. This discussion is going to look like a lot of excuses about why certain important things are not our problem! That's the price we pay for not solving all the world's problems, and being intellectually honest about it.

Encryption Basics & Terminology

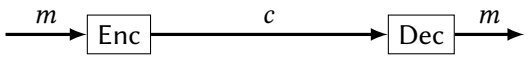
In our idealized model, Alice has a message m that she wants to send (privately) to Bob. We call m the **plaintext**. We assume she will process that plaintext somehow to obtain a value c (called the **ciphertext**) that she will actually send to Bob. The process of transforming m into c is called encryption, and we will use Enc to refer to the encryption algorithm.

We assume that the ciphertext may be observed by the eavesdropper Eve. Note that we are not trying to hide *the fact that Alice is sending something* to Bob,[2] we only want to hide the *contents* of that message. We also are not concerned with how c reliably gets from Alice to Bob. For now, we will not worry about an attacker that tampers with c (causing Bob to receive a different value), but this scenario will be discussed later.

When Bob receives c, he runs a decryption algorithm Dec to (hopefully) obtain the original plaintext m.

[1] "Eavesdropper" refers to someone who secretly listens in on a conversation between others. The term originated as a reference to someone who literally hung from the eaves of a building in order to hear conversations happening inside.

[2] Hiding the existence of a communication channel is called steganography



Secrets & Kerckhoffs' Principle

Something important is missing from this picture. If we want Bob to be able to decrypt c, but Eve to *not* be able to decrypt c, then Bob must have some information that Eve doesn't have (do you see why?). Something has to be kept secret from Eve.

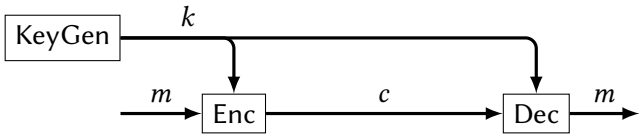
You might suggest to make the details of the Dec algorithm secret. This is how cryptography was done throughout most of the last 2000 years, but it has major drawbacks. If the attacker does eventually learn the details of Enc and Dec, then the only way to recover security is to *invent new algorithms.* If you have a system with many users, then the only way to prevent everyone from reading everyone else's messages is to *invent new algorithms* for each pair of users. Inventing even one good encryption method is already hard enough!

The first person to articulate this problem was Auguste Kerckhoffs. In 1883 he formulated a set of cryptographic design principles, the most famous of which is now known as **Kerckhoffs' principle**:

> **Kerckhoffs' Principle:**
>
> *"Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."*
>
> **Literal translation:** [The method] must not be required to be secret, and it must be able to fall into the enemy's hands without causing inconvenience.
>
> **Bottom line:** Design your system to be secure even if the attacker has complete knowledge of all its algorithms.

If the algorithms themselves are not secret, then there must be some other secret information in the system. That information is called the **(secret) key**. The key is just an extra piece of information given to both the Enc and Dec algorithms. Another way to interpret Kerckhoffs' principle is that *all of the security of the system should be concentrated in the secrecy of the key*, not the secrecy of the algorithms. If a secret key gets compromised, you only need to choose a new one, not reinvent an entirely new encryption algorithm. Multiple users can all safely use the same encryption algorithm but with independently chosen secret keys.

The process of choosing a secret key is called **key generation**, and we write KeyGen to refer to the (randomized) key generation algorithm. We call the collection of three algorithms (Enc, Dec, KeyGen) an **encryption scheme.** Remember that Kerckhoffs' principle says that we should assume that an attacker knows the details of the KeyGen algorithm. But also remember that knowing the details (i.e., source code) of a randomized algorithm doesn't mean you know the *specific output* it gave when the algorithm was executed.



Excuses, Assumptions

Here are a few things that we will blatantly ignore.

We won't discuss *how* Alice and Bob actually obtain a common secret key. This is obviously an incredibly important problem (known as **key distribution**) in the real world. We will discuss some clever approaches to this problem much later in the book. In our defense, the problem we are solving is already quite non-trivial: once two users have established a shared secret key, how can they use that key to communicate privately?[3]

Throughout this course we also just assume that the users have the ability to uniformly sample random strings. Indeed, without randomness there is no cryptography. In the real world, obtaining uniformly random bits from deterministic computers is extremely non-trivial. John von Neumann famously said, *"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."* Again, even when we take uniform randomness for granted, we still face the non-trivial question of how to *use* that randomness for private communication (and other applications), and also how to use only a *manageable amount* of randomness.

For now, we will assume that both Alice and Bob each have a copy of the *same* key. Much later in the course we will also consider situations where Alice & Bob actually have different secret information.

Not Cryptography

People use many techniques to try to hide information, but many are "non-cryptographic" since they violate Kerckhoffs' principle:

- Encoding/decoding methods like base64 . . .

  `joy of cryptography` ↔ `aGFoYSwgSSBmb29sZWQgeW91IQ==`

  . . . are **public** algorithms that involve no secrets. base64 is useful for incorporating arbitrary binary data into a structured file format. But it adds nothing in terms of *security*.

- Sometimes the simplest way to describe an encryption scheme is with operations on binary strings (i.e., `0`s and `1`s) data. As we will see, one-time pad is defined in terms of plaintexts represented as strings of bits. (Future schemes will require inputs to be represented a bitstring of a specific length, or an element of $\mathbb{Z}_n$, etc.)

  In order to make sense of some algorithms in this course, it may be necessary to think about data being converted into binary representation. As above, representing things in binary has no effect on security since it does not involve the secret key.

[3] One of my favorite descriptions of cryptography is: "cryptography is a tool for turning lots of different problems into key management problems." I first heard this quote in a talk by Lea Kissner, principal security engineer at Google.

1.2 Specifics of One-Time Pad

People have been trying to send secret messages for roughly 2000 years. Unfortunately, there are really only 2 useful ideas from before 1900 that have any relevance to modern cryptography. The first idea is Kerckhoffs' principle, which you have already seen. The other idea is **one-time pad**, which illustrates several important concepts, and can even still be found hiding deep inside many modern encryption schemes.

One-time pad is sometimes called "Vernam's cipher" after Gilbert Vernam, a telegraph engineer who patented the scheme in 1919. However, an earlier description of one-time pad was rather recently discovered in an 1882 text by Frank Miller on telegraph encryption.[4]

In most of this book, secret keys will be strings of bits. We generally use the variable λ to refer to the length (# of bits) of the secret key in a scheme, so that keys are elements of the set $\{0,1\}^\lambda$. In the case of one-time pad, the choice of λ doesn't affect security ($\lambda = 10$ is "just as secure" as $\lambda = 1000$). However, in future chapters, increasing λ has the effect of making the scheme harder to break. For that reason, λ is often called the **security parameter** of the scheme.

In one-time pad, not only are the keys λ-bit strings, but plaintexts and ciphertexts are too. You should consider this to be just a simple coincidence, because we will soon encounter schemes in which keys, plaintexts, and ciphertexts are strings of different sizes.

The specific KeyGen, Enc, and Dec algorithms for one-time pad are given below:

Construction 1.1 (One-time pad)

KeyGen:	$\mathsf{Enc}(k, m \in \{0,1\}^\lambda)$:	$\mathsf{Dec}(k, c \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$ return k	return $k \oplus m$	return $k \oplus c$

Recall that "$k \leftarrow \{0,1\}^\lambda$" means to sample k uniformly from the set of λ-bit strings. The way we have defined one-time pad states that the key should be chosen in exactly this way.

Example *Encrypting the following 20-bit plaintext m under the 20-bit key k using OTP results in the ciphertext c below:*

$$\begin{array}{cll} & 11101111101111100011 & (m) \\ \oplus & 00011001110000111101 & (k) \\ \hline & 11110110011111011110 & (c = \mathsf{Enc}(k, m)) \end{array}$$

Decrypting the following ciphertext c using the key k results in the plaintext m below:

$$\begin{array}{cll} & 00001001011110010000 & (c) \\ \oplus & 10010011101011100010 & (k) \\ \hline & 10011010110101110010 & (m = \mathsf{Dec}(k, c)) \end{array}$$

[4] See the article Steven M. Bellovin: "Frank Miller: Inventor of the One-Time Pad." *Cryptologia* 35 (3), 2011.

Note that Enc and Dec are essentially the same algorithm (return the XOR of the two arguments). This results in some small level of convenience and symmetry when implementing one-time pad, but it is more of a coincidence than something truly fundamental about encryption (see Exercises 1.12 & 2.5). Certainly you should expect to soon start seeing encryption schemes whose encryption & decryption algorithms look very different.

Correctness

The first property of one-time pad that we should confirm is that the receiver does indeed recover the intended plaintext when decrypting the ciphertext. Without this property, the thought of using one-time pad for communication seems silly. Written mathematically:

Claim 1.2 *For all* $k, m \in \{0,1\}^\lambda$*, it is true that* $\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$.

Proof This follows by substituting the definitions of OTP Enc and Dec, along with the properties of XOR listed in Chapter 0.2. For all $k, m \in \{0,1\}^\lambda$, we have:

$$\begin{aligned}\mathsf{Dec}(k, \mathsf{Enc}(k, m)) &= \mathsf{Dec}(k, k \oplus m)\\ &= k \oplus (k \oplus m)\\ &= (k \oplus k) \oplus m\\ &= 0^\lambda \oplus m\\ &= m.\end{aligned}$$

Example *Encrypting the following plaintext* m *under the key* k *results in ciphertext* c*, as follows:*

$$\begin{array}{cll} & 00110100110110001111 & (m)\\ \oplus & 11101010011010001101 & (k)\\ \hline & 11011110101100000010 & (c)\end{array}$$

Decrypting c *using the same key* k *results in the original* m*:*

$$\begin{array}{cll} & 11011110101100000010 & (c)\\ \oplus & 11101010011010001101 & (k)\\ \hline & 00110100110110001111 & (m)\end{array}$$

Security

Suppose Alice encrypts a plaintext m and an eavesdropper eventually sees the resulting ciphertext c. We want to say something like "the eavesdropper (who doesn't know k) doesn't learn about m." First, we need to be as precise as possible about what exactly the eavesdropper sees. We can represent what the eavesdropper sees as an output of the following algorithm:

```
EAVESDROP(m ∈ {0,1}^λ):
  k ← {0,1}^λ
  c := k ⊕ m
  return c
```

.

This algorithm describes how the sender computes the ciphertext using secret values (choosing a key k in a specific way, and using the one-time-pad encryption procedure). It also describes that the eavesdropper sees *only* the ciphertext (but not the key).

This is a *randomized* algorithm, which you can see from the random choice of k. Even after fixing the input m, the output is not fixed. Instead of thinking of "$\text{EAVESDROP}(m)$" as a fixed output, you should think of it as a *probability distribution* over output values. More precisely, we can say that an eavesdropper sees a **sample** from the distribution $\text{EAVESDROP}(m)$.

Example *Let's take $\lambda = 3$ and work out by hand the distributions $\text{EAVESDROP}(\texttt{010})$ and $\text{EAVESDROP}(\texttt{111})$. In each case EAVESDROP chooses a value of k uniformly in $\{\texttt{0},\texttt{1}\}^3$ — each of the possible values with probability 1/8. For each possible choice of k, we can compute what the output of EAVESDROP (c) will be:*

$\text{EAVESDROP}(\texttt{010})$:

Pr	k	output $c = k \oplus \texttt{010}$
1/8	000	010
1/8	001	011
1/8	010	000
1/8	011	001
1/8	100	110
1/8	101	111
1/8	110	100
1/8	111	101

$\text{EAVESDROP}(\texttt{111})$:

Pr	k	output $c = k \oplus \texttt{111}$
1/8	000	111
1/8	001	110
1/8	010	101
1/8	011	100
1/8	100	011
1/8	101	010
1/8	110	001
1/8	111	000

So the distribution $\text{EAVESDROP}(\texttt{010})$ assigns probabilty 1/8 to 010, probability 1/8 to 011, and so on.

In this example, notice how every string in $\{\texttt{0},\texttt{1}\}^3$ appears *exactly once* in the c column of $\text{EAVESDROP}(\texttt{010})$. This means that EAVESDROP assigns probability 1/8 to *every* string in $\{\texttt{0},\texttt{1}\}^3$, which is just another way of saying that the distribution is the *uniform distribution* on $\{\texttt{0},\texttt{1}\}^3$. The same can be said about the distribution $\text{EAVESDROP}(\texttt{111})$, too. Both distributions are just the uniform distribution in disguise!

There is nothing special about 010 or 111 in these examples. For any λ and any $m \in \{\texttt{0},\texttt{1}\}^\lambda$, the distribution $\text{EAVESDROP}(m)$ is the uniform distribution (over $\{\texttt{0},\texttt{1}\}^\lambda$). Before we prove it, let's first think about why you should care. From the eavesdropper's point of view, someone chooses a plaintext m and shows you a sample from the distribution $\text{EAVESDROP}(m)$. But this is a distribution that you can sample from yourself, even if you don't know m. Indeed, you could have chosen an arbitrary m' and run $\text{EAVESDROP}(m')$, and you would have induced the same distribution as $\text{EAVESDROP}(m)$. Truly, the ciphertext that you see (a sample from some distribution) can carry *no information* about m if it is possible to sample from the same distribution without even knowing m!

Claim 1.3 *For every $m \in \{\texttt{0},\texttt{1}\}^\lambda$, the distribution $\text{EAVESDROP}(m)$ is the **uniform distribution** on $\{\texttt{0},\texttt{1}\}^\lambda$. Hence, for all $m, m' \in \{\texttt{0},\texttt{1}\}^\lambda$, the distributions $\text{EAVESDROP}(m)$ and $\text{EAVESDROP}(m')$ are identical.*

Proof Arbitrarily fix $m, c \in \{0,1\}^\lambda$. We will calculate the probability that EAVESDROP(m) produces output c. That event happens only when

$$c = k \oplus m \iff k = m \oplus c.$$

The equivalence follows from the properties of XOR given in Section 0.2. That is,

$$\Pr[\text{EAVESDROP}(m) = c] = \Pr[k = m \oplus c],$$

where the probability is over uniform choice of $k \leftarrow \{0,1\}^\lambda$.

We have fixed specific m and c, so there is *only one* value of k that makes $k = m \oplus c$ true (encrypts m to c), and that value is exactly $m \oplus c$. Since k is chosen *uniformly* from $\{0,1\}^\lambda$, the probability of choosing the particular value $k = m \oplus c$ is $1/2^\lambda$. ■

Discussion

- **Isn't there a paradox?** Claim 1.2 says that c can always be decrypted to get m, but Claim 1.3 says that c contains no information about m! The answer to this riddle is that Claim 1.2 talks about what can be done with knowledge of the key k. Claim 1.3 is about the output distribution of the EAVESDROP algorithm, which doesn't include k (see Exercise 1.9). In short, if you know k, then you can decrypt c to obtain m; if you don't know k, then c carries no information about m (in fact, it looks uniformly distributed). This is because m, c, k are all *correlated* in a delicate way.[5]

- **Isn't there another paradox?** Claim 1.3 says that the output of EAVESDROP(m) doesn't depend on m, but the EAVESDROP algorithm uses its argument m right there in the last line! The answer to this riddle is perhaps best illustrated by the previous examples of EAVESDROP(010) and EAVESDROP(111). The two tables of values are indeed different (so the choice of $m \in \{010, 111\}$ clearly has some effect), but they represent the *same probability distribution* (since order doesn't matter). Claim 1.3 considers only the resulting probability distribution.

Limitations

The keys in one-time pad are as long as the plaintexts they encrypt. This is more or less unavoidable (see Exercise 2.11) and leads to a kind of chicken-and-egg dilemma: If two users want to privately convey a λ-bit message, they first need to privately agree on a λ-bit string.

Additionally, one-time pad keys can be used to encrypt only one plaintext (hence, "one-time" pad); see Exercise 1.6. Indeed, we can see that the EAVESDROP subroutine in Claim 1.3 provides no way for a caller to guarantee that two plaintexts are encrypted with the same key, so it is not clear how to use Claim 1.3 to argue about what happens in one-time pad when keys are reused in this way.

Despite these limitations, one-time pad illustrates fundamental ideas that appear in most forms of encryption in this course.

[5] This correlation is explored further in Chapter 3.

Exercises

1.1. The one-time pad encryption of plaintext `mario` (when converted from ASCII to binary in the standard way) under key k is:

$$\texttt{100001000000011101010100000111000001110 1}.$$

What is the one-time pad encryption of `luigi` under the same key?

1.2. Alice is using one-time pad and notices that when her key is the all-zeroes string $k = \texttt{0}^\lambda$, then $\mathsf{Enc}(k, m) = m$ and her message is sent in the clear! To avoid this problem, she decides to modify KeyGen to exclude the all-zeroes key. She modifies KeyGen to choose a key uniformly from $\{\texttt{0},\texttt{1}\}^\lambda \setminus \{\texttt{0}^\lambda\}$, the set of all λ-bit strings except $\texttt{0}^\lambda$. In this way, she guarantees that her plaintext is never sent in the clear.

Is it still true that the eavesdropper's ciphertext distribution is uniform? Prove or disprove.

1.3. When Alice encrypts the key k itself using one-time pad, the ciphertext will always be the all-zeroes string! So if an eavesdropper sees the all-zeroes ciphertext, she learns that Alice encrypted the key itself. Does this contradict Claim 1.3? Why or why not?

1.4. What is so special about defining OTP using the XOR operation? Suppose we use the bitwise-AND operation (which we will write as '&') and define a variant of OTP as follows:

KeyGen:	$\mathsf{Enc}(k, m \in \{\texttt{0},\texttt{1}\}^\lambda)$:
$k \leftarrow \{\texttt{0},\texttt{1}\}^\lambda$	return $k \,\&\, m$
return k	

Is this still a good choice for encryption? Why / why not?

1.5. Describe the flaw in this argument:

> Consider the following attack against one-time pad: upon seeing a ciphertext c, the eavesdropper tries every candidate key $k \in \{\texttt{0},\texttt{1}\}^\lambda$ until she has found the one that was used, at which point she outputs the plaintext m. This contradicts the argument in Section 1.2 that the eavesdropper can obtain no information about m by seeing the ciphertext.

1.6. Suppose Alice encrypts two plaintexts m and m' using one-time pad with the same key k. What information about m and m' is leaked to an eavesdropper by doing this (assume the eavesdropper knows that Alice has reused k)? Be as specific as you can!

1.7. You (Eve) have intercepted two ciphertexts:

$$c_1 = \texttt{1111100101111001110011000001011110000110}$$
$$c_2 = \texttt{1111101001100111110111010000100110001000}$$

You know that both are OTP ciphertexts, encrypted with the *same key*. You know that **either** c_1 is an encryption of `alpha` and c_2 is an encryption of `bravo` **or** c_1 is an encryption

of `delta` and c_2 is an encryption of `gamma` (all converted to binary from ASCII in the standard way).

Which of these two possibilities is correct, and why? What was the key k?

1.8. A **known-plaintext attack** refers to a situation where an eavesdropper sees a ciphertext $c = \mathrm{Enc}(k, m)$ and also learns/knows what plaintext m was used to generate c.

(a) Show that a known-plaintext attack on OTP results in the attacker learning the key k.

(b) Can OTP be secure if it allows an attacker to recover the encryption key? Is this a contradiction to the security we showed for OTP? Explain.

1.9. Suppose we modify the subroutine discussed in Claim 1.3 so that it also returns k:

```
EAVESDROP'(m ∈ {0,1}^λ):
  k ← {0,1}^λ
  c := k ⊕ m
  return ( k , c)
```

.

Is it still true that for every m, the output of EAVESDROP$'(m)$ is distributed uniformly in $(\{0,1\}^\lambda)^2$? Or is the output distribution different for different choice of m?

1.10. In this problem we discuss the security of performing one-time pad encryption twice:

(a) Consider the following subroutine that models the result of applying one-time pad encryption with two *independent* keys:

```
EAVESDROP'(m ∈ {0,1}^λ):
  k_1 ← {0,1}^λ
  k_2 ← {0,1}^λ
  c := k_2 ⊕ (k_1 ⊕ m)
  return c
```

.

Show that the output of this subroutine is uniformly distributed in $\{0,1\}^\lambda$.

(b) What security is provided by performing one-time pad encryption twice with *the same key*?

1.11. We mentioned that one-time pad keys can be used to encrypt only one plaintext, and how this was reflected in the EAVESDROP subroutine of Claim 1.3. Is there a similar restriction about re-using *plaintexts* in OTP (but with independenty random keys for different ciphertexts)? If an eavesdropper *knows* that the same plaintext is encrypted twice (but doesn't know what the plaintext is), can she learn anything? Does Claim 1.3 have anything to say about a situation where the same plaintext is encrypted more than once?

1.12. In this problem we consider a variant of one-time pad, in which the keys, plaintexts, and ciphertexts are all elements of $\mathbb{Z}_n$ instead of $\{0,1\}^\lambda$.

(a) What is the decryption algorithm that corresponds to the following encryption algorithm?

$$\boxed{\begin{array}{l}\underline{\mathsf{Enc}(k, m \in \mathbb{Z}_n):} \\ \quad \text{return } (k + m) \,\%\, n\end{array}}.$$

(b) Show that the output of the following subroutine is uniformly distributed in $\mathbb{Z}_n$:

$$\boxed{\begin{array}{l}\underline{\textsc{eavesdrop}'(m \in \mathbb{Z}_n):} \\ \quad k \leftarrow \mathbb{Z}_n \\ \quad c := (k + m) \,\%\, n \\ \quad \text{return } c\end{array}}.$$