



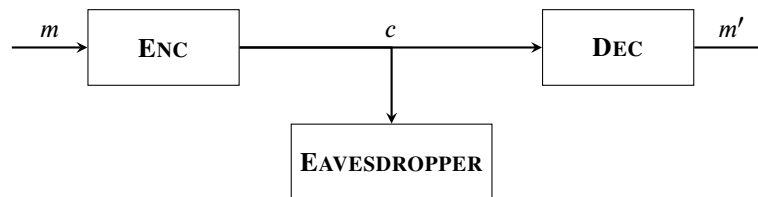
# 1 One Time Pad

## 1.1 What is [NOT] CRYPTOGRAPHY

### 1.1.1 Introduction

In the idealized model we assume that Alice wants to send a message  $m$  (*privately*) to Bob. Alice will modify the message  $m$ , also called **plaintext**, with any method to create a **ciphertext**  $c$  which will be actually sent to Bob. This transformation is also called encryption (**Enc**). After receiving the ciphertext  $c$ , Bob will reverse the step of transforming by using a decryption algorithm (**Dec**) to (*hopefully*) get the original plaintext  $m$ .

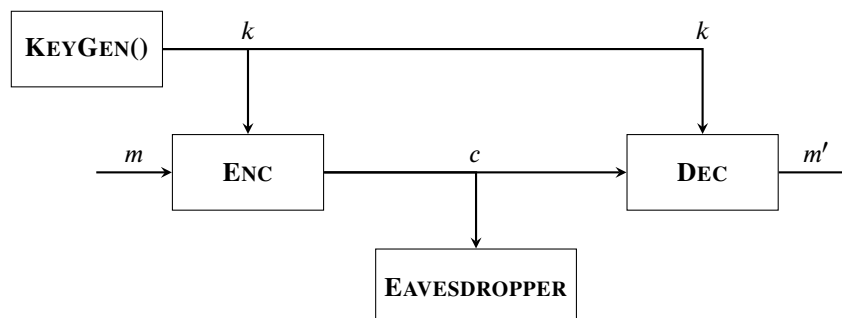
**NOTE:** We are not trying to hide that a message is sent, so an **EAVESDROPPER** (an attacker between Alice and Bob) can obtain the ciphertext  $c$  at any instance. Hiding the existence of communication is called *steganography*.



### 1.1.2 Kerckhoff's principle

*The method must not be required to be secret, and it must be able to fall into the enemy's hands without causing any inconvenience.*

So if the algorithm do not need to be secret, there must be additional information in the system, which is kept secret from any **EAVESDROPPER**. This information is called a (**secret**) **key**  $k$ .





## 1.2 Specifics of ONE-TIME PAD

A *one-time pad* often uses a secret key in the form of a bit string of length  $\lambda$ . The plain- and ciphertexts are also  $\lambda$ bit-strings.

The construction of such an one-time pad looks as follows:

|   |   |   |
|---|---|---|
| $\begin{array}{l} \text{KEYGEN}() \\ k \leftarrow \{0, 1\}^\lambda \\ \text{return } k \end{array}$ | $\begin{array}{l} \text{ENC}(k, m \in \{0, 1\}^\lambda) \\ c := k \oplus m \\ \text{return } c \end{array}$ | $\begin{array}{l} \text{DEC}(k, c \in \{0, 1\}^\lambda) \\ m' := k \oplus c \\ \text{return } m' \end{array}$ |
|---|---|---|

Recapture that  $k \leftarrow \{0, 1\}^\lambda$  means that  $k$  is sampled uniformly from the set of  $\lambda$ bit-strings.

Further we claim that for all  $k, m \in \{0, 1\}^\lambda$  it is true, that  $\text{Dec}(k, \text{Enc}(k, m)) = m$ .

Otherwise the usage of one-time pad would be silly.

For security reasons we want to say about the encryption scheme that an EAVESDROPPER (who does not know  $k$ ) cannot learn anything about the message  $m$ .

In the end we need to claim that an encryption algorithm is secure if for every  $m \in \{0, 1\}^\lambda$  the distribution  $\text{EAVESDROP}(m)$  is the **uniform distribution** of  $\{0, 1\}^\lambda$ , explicitly for every  $m, m' \in \{0, 1\}^\lambda$  the distributions  $\text{EAVESDROP}(m)$  and  $\text{EAVESDROP}(m')$  are identical.

## 2 The Basics of Proveable Security

### 2.1 Generalizing and Abstracting One-Time Pad

#### 2.1.1 Syntax & Correctness

A **symmetric key encryption (SKE) scheme** consists of the following algorithms:

- ▷ **KEYGEN**: a randomized algorithm that outputs a **key**  $k \in \mathbb{K}$
- ▷ **ENC**: a (*possibly randomized*) algorithm that takes a key  $k \in \mathbb{K}$  and **plaintext**  $m \in \mathbb{M}$  as input, and outputs a **ciphertext**  $c \in \mathbb{C}$
- ▷ **DEC**: a deterministic algorithm that takes a key  $k \in \mathbb{K}$  and a ciphertexts  $c \in \mathbb{C}$  as input, and outputs a plaintext  $m \in \mathbb{M}$

$\mathbb{K}$  is called the **key space**,  $\mathbb{M}$  the **message space**, and  $\mathbb{C}$  the **ciphertext space** of the scheme. Often the entire scheme (with all its algorithms) is referred to as  $\Sigma$ . Each component is then referred to as  $\Sigma.\text{KEYGEN}$ ,  $\Sigma.\text{ENC}$ ,  $\Sigma.\text{DEC}$ ,  $\Sigma.\mathbb{K}$ ,  $\Sigma.\mathbb{M}$ , and  $\Sigma.\mathbb{C}$ .

Furthermore we define an encryption scheme to be **correct** if for all  $k \in \mathbb{K}$  and all  $m \in \mathbb{M}$ :

$$\Pr[\Sigma.\text{DEC}(k, \Sigma.\text{ENC}(k, m)) = m] = 1$$

In other words, decrypting a ciphertext, using the same key used for encryption, **always** results in the original plaintext.



## 2.2 Towards an Abstract Security Definition

With the properties of one-time pad shown in the first chapter a first attempt can be made to define security:

For all  $m \in \{0, 1\}^\lambda$ , the output of the following subroutine is uniformly distributed over  $\{0, 1\}^\lambda$ :

EAVESDROP( $m \in \{0, 1\}^\lambda$ ):  
 $k \leftarrow \{0, 1\}^\lambda$   
 $c := k \oplus m$   
**return**  $c$

This property is too specific to one-time pad. To get a more general-purpose security definition, we can write the subroutine using the totally generic encryption scheme  $\Sigma$ :

EAVESDROP( $m \in \Sigma.\mathbb{M}$ ):  
 $k \leftarrow \Sigma.\mathbb{K}$   
 $c := \Sigma.\text{ENC}(k, m)$   
**return**  $c$

Such an encryption scheme  $\Sigma$  is secure if, for all  $m \in \Sigma.\mathbb{M}$ , the output of this subroutine is uniformly distributed over  $\Sigma.\mathbb{C}$ .

### 2.2.1 Adversary as Distinguishers

For a better conceptualization of the security definition we consider the following two libraries:

EAVESDROP( $m \in \Sigma.\mathbb{M}$ ):  
 $k \leftarrow \Sigma.\mathbb{K}$   
 $c := \Sigma.\text{ENC}(k, m)$   
**return**  $c$

EAVESDROP( $m \in \Sigma.\mathbb{M}$ ):  
 $c \leftarrow \Sigma.\mathbb{C}$   
**return**  $c$

$\Sigma$  is secure if both implementations of the subroutine EAVESDROP have the same *input-output behavior* (both subroutines generate the same output distribution, on every input).

Furthermore for every calling program  $\mathbb{A}$  the connection with the right or left version of the subroutine EAVESDROP should not change the output distribution of  $\mathbb{A}$ . Such an  $\mathbb{A}$  is called an **adversary** which gets to choose plaintexts to send to an EAVESDROP subroutine, but does not know which one of the left or right version is used. Its only goal is to **distinguish** between the left and right implementation. This means to detect differences in the behavior of the two subroutines. To distinguish whether our implementation is secure, the following must hold:

$$Pr[\text{the adversary outputs } \mathbf{1} \text{ if connected to the left implementation}]$$

$$=$$

$$Pr[\text{the adversary outputs } \mathbf{1} \text{ if connected to the right implementation}]$$



### 2.2.1.1 Example of a non-secure $\Sigma$

In order to add some redundancy to the data, the following OTP is used:

|                                    |                           |  |   |
|------------------------------------|---------------------------|--|---|
| $\mathbb{K} = \{0, 1\}^\lambda$    | <u>KEYGEN:</u>            | <u>ENC(<math>k, m \in \mathbb{M}</math>)</u> | <u>DEC(<math>k, c \in \mathbb{C}</math>)</u>    |
| $\mathbb{M} = \{0, 1\}^\lambda$    | $k \leftarrow \mathbb{K}$ | $c' = k \oplus m$                            | $c' = \text{first } \lambda \text{ bits of } c$ |
| $\mathbb{C} = \{0, 1\}^{2\lambda}$ | <b>return</b> $k$         | $c := c' \  c'$                              | <b>return</b> $c' \oplus k$                     |
|                                    |                           | <b>return</b> $c$                            |   |

Intuitively this version of OTP should be also *secure*. However doubling the output ciphertext does not meet the security definition from above.

We can write an adversary which can distinguish between this scheme and a totally random subroutine version:

|                                      |
|--------------------------------------|
| $\mathbb{A}$                         |
| $c = \text{EAVESDROP}(0^\lambda)$    |
| $L = \text{first half of } c$        |
| $R = \text{second half of } c$       |
| $\text{return } L \stackrel{?}{=} R$ |

For the doubling version of the subroutine this distinguisher will **always** return 1. The probability that a fully random subroutine returns a ciphertext with equal first and second halves is  $\frac{1}{2^\lambda}$ . Therefore the probabilities are not equal and the adversary can successfully distinguish between both implementations

### 2.2.2 Chosen Plaintext Attack Template

Filler

## 2.3 Provable Security Fundamentals

### 2.3.1 Libraries & Interfaces

Filler

### 2.3.2 Semantics & Scopes

Filler

### 2.3.3 Interchangeability

Filler

### 2.3.4 Security Definition, Using New Terminology

Filler



## **2.4 How to Prove Security with the Hybrid Technique**

### **2.4.1 Chaining Several Components**

Filler

### **2.4.2 One-Time Secrecy of One-Time Pad**

Filler

## **2.5 How to Demonstrate Insecurity with Attacks**

Filler