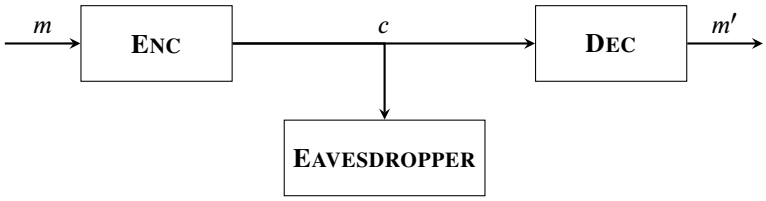
1 One Time Pad

1.1 What is [NOT] CRYPTOGRAPHY

1.1.1 Introduction

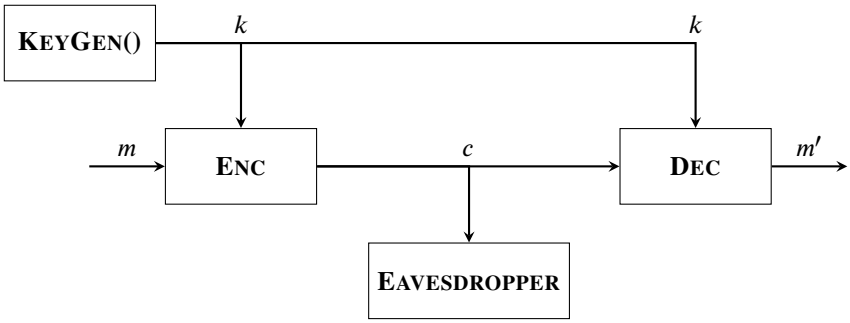
In the idealized model we assume that Alice wants to send a message m (*privately*) to Bob. Alice will modify the message m, also called ***plaintext***, with any method to create a ***ciphertext*** c which will be actually sent to Bob. This transformation is also called encryption (***Enc***). After receiving the ciphertext c, Bob will reverse the step of transforming by using a decription 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 λ. The plain- and ciphertexts are also λbit-strings.
The construction of such an one-time pad looks as follows:

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

Recapture that $k \leftarrow \{0,1\}^\lambda$ means that k is sampled uniformly from the set of λbit-strings.
Further we claim that for all $k, m \in \{0,1\}^\lambda$ it is true, that $Dec(k, Enc(k, m)) = m$.
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 leran 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 EAVESDROP(m) is the **uniform distribution** of $\{0,1\}^\lambda$, explicitly for every $m, m' \in \{0,1\}^\lambda$ the distributions EAVESDROP(m) and 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** consissts 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 ouputs 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 Σ. Each component is then referred to as Σ.KEYGEN, Σ.ENC, Σ.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 ∈ {0,1}^λ):
  k ← {0,1}^λ
  c := k ⊕ 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 Σ:

```
EAVESDROP(m ∈ Σ.𝕄):
  k ← Σ.𝕂
  c := Σ.ENC(k, m)
  return c
```

Such an encryption scheme Σ 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 ∈ Σ.𝕄):
  k ← Σ.𝕂
  c := Σ.ENC(k, m)
  return c
```

```
EAVESDROP(m ∈ Σ.𝕄):
  c ← Σ.ℂ
  return c
```

Σ 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 too 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** betwenn 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:

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

$$=$$

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

2.2.1.1 Example of a *non-secure* Σ

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

$\mathbb{K} = \{0,1\}^\lambda$ $\mathbb{M} = \{0,1\}^\lambda$ $\mathbb{C} = \{0,1\}^{2\lambda}$	KEYGEN: $k \leftarrow \mathbb{K}$ **return** k	ENC$(k, m \in \mathbb{M})$ $c' = k \oplus m$ $c := c' \| c'$ **return** c	DEC$(k, c \in \mathbb{C})$ $c' = first\ \lambda\ bits\ of\ c$ **return** $c' \oplus k$

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:

ADVERSARY $\mathbb{A}$
c = EAVESDROP(0^λ)
L = first half of c
R = second half of c
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

In general the "doubling-OTP" is *secure* over the distributione of 2λ-bit-strings with the same first and second half. We required EAVESDROP to be uniform, while the more useful definition is that EAVESDROP is uniform for all m. Therefore we got another approach for the security definition:
Σ is *secure* if, for all calling programs $\mathbb{A}$, the distribution of the following two subroutines does not differ:

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

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

A calling program chooses two (different) plaintexts and give them to the subroutine. Each EAVESDROP ignores one of the inputs and only encrypts one plaintext. If this subroutine is *secure* an adversary cannot distinguish between the two implementations because they will have the same input/output behavior. Otherwise $\mathbb{A}$ can see a difference which contriutes against the security definition.
These sort of "attacks" are called **chosen-plaintext** attacks. This security definition is based on that you cannot get any information from the ciphertext about the plaintext - even if you know it was only one of two options - which you have chosen.

2.3 Provable Security Fundamentals

2.3.1 Libraries & Interfaces

A ***library*** $\mathbb{L}$ is a collection of subroutines and *private/static* variables. A library is often represented in its ***interface*** form, in which the names, argument types and, output types of all its subroutines are stated.
When a calling program $\mathbb{A}$ is linked to a distinct library we write: $\mathbb{A} \diamond \mathbb{L}$, which means that the implementation of a subroutine stated in the library $\mathbb{L}$ is used. The event that $\mathbb{A} \diamond \mathbb{L}$ outputs the value z is written as $\mathbb{A} \diamond \mathbb{L} \Rightarrow z$.

2.3.1.1 Example for linking adversary and interfaces

We consider the following libraries $\mathbb{L}_1$ and $\mathbb{L}_2$ (which we have seen before):

Library $\mathbb{L}_1$
EAVESDROP(m):
$k \leftarrow \{0,1\}^\lambda$
$c' := k \oplus m$
return $c' \| c'$

Library $\mathbb{L}_2$
EAVESDROP(m):
$c \leftarrow \{0,1\}^{2\lambda}$
return c

And the following adversary $\mathbb{A}$:

Adversary $\mathbb{A}$
$c :=$ EAVESDROP(0^λ)
$L :=$ first half of c
$R :=$ second half of c
return $L \stackrel{?}{=} R$

The outputs of the previously discussed form are then:

$$Pr[\mathbb{A} \diamond \mathbb{L}_1 \Rightarrow \text{TRUE}] = 1$$
$$Pr[\mathbb{A} \diamond \mathbb{L}_2 \Rightarrow \text{TRUE}] = \frac{1}{2^\lambda}$$

2.3.1.2 Example of library interfaces

A library can also contain couple of subroutines:

Library $\mathbb{L}$
$s \leftarrow \{0,1\}^\lambda$
RESET():
$s \leftarrow \{0,1\}^\lambda$
GUESS$(x \in \{0,1\}^\lambda)$
return $x \stackrel{?}{=} s$

Code outside of a subroutine is run one in the initialization. Variables defined in this initialization time (as s in this library) are visible in all subroutine scopes.

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 Demonstarte Insecurity with Attacks

Filler