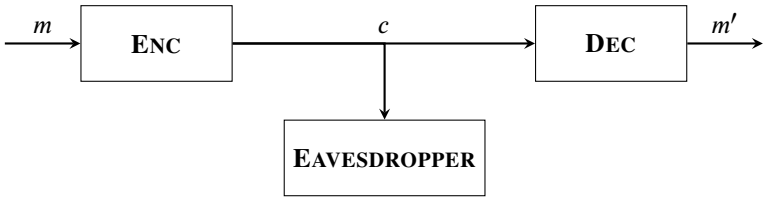
1 One Time Pad

1.1 What is [NOT] CRYPTOGRAPHY

1.1.1 Introduction

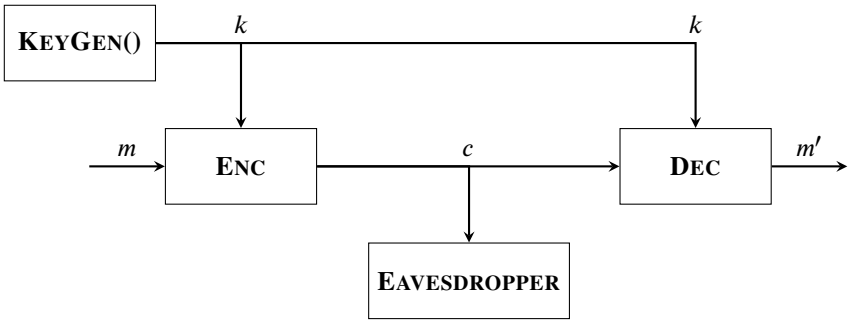
In the idealized model we assume that Alice wants to send a message m (*privately*) to Bob. Alice will modify the message m, also called ***plaintext***, with any method to create a ***ciphertext*** c which will be actually sent to Bob. This transformation is also called encryption (***Enc***). After 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 Interchangeability

We call two libraries $\mathbb{L}_{left}$ and $\mathbb{L}_{right}$ **interchangeable** and write $\mathbb{L}_{left} \equiv \mathbb{L}_{right}$, if for all calling pragrams $\mathbb{A}$ that output a single bit:

$$Pr[\mathbb{A} \diamond \mathbb{L}_{left} \Rightarrow 1] \;=\; Pr[\mathbb{A} \diamond \mathbb{L}_{right} \Rightarrow 1]$$

The interchangeability of libraries includes:
- Their only difference happens in an *unreachable block of code*
- Their only difference is the value they assign to a *variable that is never actually used*
- Their only difference is that one library *unrolls a loop* that occurs in the other library
- Their only difference is that one library *inlines a subroutine* that occurs in the other library

2.3.3 Security Definition, Using New Terminology

We say that Σ has **one-time uniform ciphertexts** if $\mathbb{L}^{\Sigma}_{ots\$-real} \equiv \mathbb{L}^{\Sigma}_{ots\$-rand}$, where:

Library $\mathbb{L}^{\Sigma}_{ots\$-real}$
CTXT($m \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m)$
return c

Library $\mathbb{L}^{\Sigma}_{ots\$-rand}$
CTXT($m \in \Sigma.\mathbb{M}$):
$c \leftarrow \Sigma.\mathbb{C}$
return c

We say that Σ has **one-time secrecy** if $\mathbb{L}^{\Sigma}_{ots-R} \equiv \mathbb{L}^{\Sigma}_{ots-L}$, where:

Library $\mathbb{L}^{\Sigma}_{ots-R}$
EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m_R)$
return c

Library $\mathbb{L}^{\Sigma}_{ots-L}$
EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m_L)$
return c

2.4 How to Prove Security with the Hybrid Technique

2.4.1 Chaining Several Components

If an adversary $\mathbb{A}$ is linked to more than one libraries (for example: $\mathbb{A} \diamond \mathbb{L}_1 \diamond \mathbb{L}_2$) there are several possible ways to interpret this:
- $(\mathbb{A} \diamond \mathbb{L}_1) \diamond \mathbb{L}_2$:
  A **compound calling program** linked to $\mathbb{L}_2$. After all, $\mathbb{A} \diamond \mathbb{L}_1$ is a program which calls subroutines of $\mathbb{L}_2$.
- $\mathbb{A} \diamond (\mathbb{L}_1 \diamond \mathbb{L}_2)$:
  $\mathbb{A}$ is linked to a **compound library**. After all, $\mathbb{A}$ is a program which calls subroutines of $\mathbb{L}_1 \diamond \mathbb{L}_2$.

Note that the placement of the parantheses does **NOT** affect the overall program, which will behave the same way in both options.

2.4.2 One-Time Secrecy of One-Time Pad

For any encryption scheme Σ which fulfills the characteristics of *one-time uniform ciphertexts* it holds that it also has *one-time secrecy*, also written as:

$$\mathbb{L}^{\Sigma}_{ots\$-real} \equiv \mathbb{L}^{\Sigma}_{ots\$-rand} \Rightarrow \mathbb{L}^{\Sigma}_{ots-L} \equiv \mathbb{L}^{\Sigma}_{ots-R}$$

We will prove this by using the ***hybrid technique***:
The scheme for this type of prove is the following:

$$\mathbb{L}^{\Sigma}_{ots-L} \equiv \mathbb{L}_{hyb-1} \equiv \mathbb{L}_{hyb-2} \equiv \mathbb{L}_{hyb-3} \equiv \ldots \equiv \mathbb{L}^{\Sigma}_{ots-R}$$

where $\mathbb{L}_{hyb-1}$, $\mathbb{L}_{hyb-2}$, … are so called ***hybrid*** libraries, which can be chosen nearly randomly. For this prove we will use that $\mathbb{L}^{\Sigma}_{ots\$-real} \equiv \mathbb{L}^{\Sigma}_{ots\$-rand}$ holds and show that $\mathbb{L}^{\Sigma}_{ots-L} \equiv \mathbb{L}^{\Sigma}_{ots-R}$ also holds:

$\mathbb{L}^{\Sigma}_{ots-L}$:

Library $\mathbb{L}^{\Sigma}_{ots-L}$
EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m_L)$
return c

We start with the library $\mathbb{L}^{\Sigma}_{ots-L}$.

$\mathbb{L}_{hyb-1}$:

EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$c := \text{CTXT}(m_L)$
return c

$\diamond$

Library $\mathbb{L}^{\Sigma}_{ots\$-real}$
CTXT($m \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m)$
return c

By factoring out the subroutine we do not change the external behaviour of this library. Also we can see that this new subroutine is equal to $\mathbb{L}^{\Sigma}_{ots\$-real}$ we have seen before.

$\mathbb{L}_{hyb-2}$:

EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$c := \text{CTXT}(m_L)$
return c

$\diamond$

Library $\mathbb{L}^{\Sigma}_{ots\$-rand}$
CTXT($m \in \Sigma.\mathbb{M}$):
$c \leftarrow \Sigma.\mathbb{C}$
return c

Because of the previously seen *chaining Lemma* we can change between $\mathbb{L}^{\Sigma}_{ots\$-real}$ and $\mathbb{L}^{\Sigma}_{ots\$-rand}$ which has no effect on its external behaviour (as shown before).

$\mathbb{L}_{hyb-3}$:

EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$c := \text{CTXT}(m_R)$
return c

$\diamond$

Library $\mathbb{L}^{\Sigma}_{ots\$-rand}$
CTXT($m \in \Sigma.\mathbb{M}$):
$c \leftarrow \Sigma.\mathbb{C}$
return c

Because CTXT does not depend on the given argument we can change it from m_L to m_R without any effects to its behaviour.

$\mathbb{L}_{hyb-4}$:

EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$c := \text{CTXT}(m_R)$
return c

$\diamond$

Library $\mathbb{L}^{\Sigma}_{ots\$-real}$
CTXT($m \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m)$
return c

We reverse the step where we changed the subroutine from $\mathbb{L}^{\Sigma}_{ots\$-real}$ to $\mathbb{L}^{\Sigma}_{ots\$-rand}$.

$\mathbb{L}^{\Sigma}_{ots-R}$:

Library $\mathbb{L}^{\Sigma}_{ots-R}$
EAVESDROP($m_R, m_L \in \Sigma.\mathbb{M}$):
$k \leftarrow \Sigma.\text{KEYGEN}$
$c \leftarrow \Sigma.\text{ENC}(k, m_R)$
return c

In the end we inline the subroutine and end with the library $\mathbb{L}^{\Sigma}_{ots-R}$.

2.5 How to Demonstrate Insecurity with Attacks

We will consider the following libraries (left and right version):

Library $\mathbb{L}^{\Sigma}_{ots-L}$
EAVESDROP(m_R, m_L):
$k \leftarrow \{permutations\ of\ \{1,\lambda\}\}$
for $i := 1$ to λ
$c_{k(i)} := (m_L)_i$
return $c_1 \dots c_\lambda$

Library $\mathbb{L}^{\Sigma}_{ots-R}$
EAVESDROP(m_R, m_L):
$k \leftarrow \{permutations\ of\ \{1,\lambda\}\}$
for $i := 1$ to λ
$c_{k(i)} := (m_R)_i$
return $c_1 \dots c_\lambda$

We define the following distinguisher:

Adversary $\mathbb{A}$
$c \leftarrow$ EAVESDROP$(0^\lambda, 1^\lambda)$
return $0^\lambda \stackrel{?}{=} c$

Linking $\mathbb{A}$ with either the *left* or *right* version of the library will result in a different behaviour:

$\mathbb{A} \diamond \mathbb{L}_{ots-\mathbb{L}}$

Linking $\mathbb{A}$ with the *left* library will return a c with each bit equal to 0. Therefore
$Pr[\mathbb{A} \diamond \mathbb{L}_{ots-\mathbb{L}} \Rightarrow 1] = 1$.

$\mathbb{A} \diamond \mathbb{L}_{ots-\mathbb{R}}$

Linking $\mathbb{A}$ with the *right* library will return a c with each bit equal to 1. Therefore
$Pr[\mathbb{A} \diamond \mathbb{L}_{ots-\mathbb{R}} \Rightarrow 1] = 0$.

Therefore the probabilities are different which leads to the conclusion that $\mathbb{A}$ can distinguish between the *left* and *right* library.

3 Basing Cryptography on Intractable Computations

3.1 What Qualifies as a "Computational Infeasible" Attack?

One-time pad cannot be broken, not even with a **brute-force** attack, trying all posible keys. However, all schemes that are coming next can be broken by such an attack. To make this more difficult for an *adversary* we use the λ parameter that we also call **security parameter** and choose it large enough to make the attack *impractical*. Ideally every attack (not just a **brute-force** attack) will have the difficulty roughly 2^{λ}.

3.1.1 Asymptotic Running Time

We will now defy where to draw a line between a "*feasible*" attack (which we want to protect against) and an "*infeasible*" one (which we do not need to care about).
We will consider the **asymptotic** cost of an attack. An algorithm which has a polynomial computation time scale reasonably well (especially if our security parameter is small) whereas one with exponential time do not. We also call a polynomial-time algorithm an "*efficient*" one.
Furthermore with the following **closure property** the consideration of polynomial-time algorithms is extremely useful: *repeating a polynomial-time algorithm a polynomial number of times result in a polynomial-time proces overall.*

3.1.2 Potential Pitfall: Numerical Algorithms

Because we will later discuss algorithms that operate over large numbers (e.g. thousand of bits long) we have to remember that representing this large number N on a computer only requires $\sim log_2 N$ bits. Therefore our security parameter will be $log_2 N$ instead of N which is a huge difference in the ranges of large N.
For reference, following are some numerical operations that will be used later:

Efficient algorithm known:
Computing GCDs
Arithmetic mod N
Inverses mod N
Exponentation mod N

No efficient algorithm known:
Factoring integers
Computing $\Phi(N)$ given N
Discrete logarithm
Square roots mod composite N

"*Efficient*" means again polynomial-time algorithms which run on standard, *classical* computers. There are known polynomial-time algorithms for the right-hand side which run on ***quantum*** computers.

3.2 What Qualifies as a "Negligible" Success Probability?

It is not enough to only consider the computation time but also the probability of an polynomial-runtime attacker to "*guess*" the right answer. Again, we do not want to consider expensive runtim algorithms and we also do not want to worry about algorithms with a success rate as low as a blind guess.
We will again use the **asymptotic** approach to draw a line between "*reasonable*" and "*unreasonable*" success.

In a scheme with λ bits, a blind guess attack succeeds with a probability of $1/2^{\lambda}$. For two, λ, λ^{42} guesses we have $2/2^{\lambda}$, $\lambda/2^{\lambda}$ and $\lambda^{42}/2^{\lambda}$ as success probability which are going towards zero for $\lambda \to \infty$. With this information we can conclude the following:

A function f is **negligible** if, for every polynomial p, we have $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.

In other words, if we independently repeat an algorithm with success rate f a polynomial number of times p the resulting success probability world be $pcdotf$.
Furthermore we will conclude that two function differ negligibly if:

If $f, g : \mathbb{N} \to \mathbb{R}$ are two functions, we write $f \approx g$ to mean that $| f(\lambda) - g(\lambda) |$ is a *negligible function*.

This terminology is exclusive for when we discuss probabilities, so the following will commonly occur:

$$Pr[X] \approx 0 \Leftrightarrow \textit{event X almost } \textbf{\textit{never}} \textit{ happens}$$
$$Pr[Y] \approx 1 \Leftrightarrow \textit{event Y almost } \textbf{\textit{always}} \textit{ happens}$$
$$Pr[A] \approx Pr[B] \Leftrightarrow \textit{events A and B happen with } \textbf{\textit{essentially the same}} \textit{ probability}$$

3.3 Indistinguishability

We will now consider libraries which we do not require to have the exactly same effect on the calling program, but only that the difference in effects is negligible. We will call two libraries $\mathbb{L}_{left}$ and $\mathbb{L}_{right}$ **indistinguishable** if, for all polynomial-time algorithms $\mathbb{A}$ that output a single bit, $Pr[A \diamond \mathbb{L}_{left} \Rightarrow 1] \approx Pr[A \diamond \mathbb{L}_{right} \Rightarrow 1]$.
The quantity $| Pr[A \diamond \mathbb{L}_{left} \Rightarrow 1] - Pr[A \diamond \mathbb{L}_{right} \Rightarrow 1] |$ **advantage** or **bias** of the algorithm $\mathbb{A}$ in distinguishing $\mathbb{L}_{left}$ and $\mathbb{L}_{right}$. Therefore two libraries are *indistingushable* if polynomial-time calling programs have negligible advantage in distinguishing them.

3.3.1 Bad Event Lemma

A common situation is that two libraries are expected to execute exactly the same statements, until some rare and exceptional condition happens. In that case, we can bound an adversary's distinguishing advantage by the probability of the exceptional state.
More formally:
Let $\mathbb{L}_{left}$ and $\mathbb{L}_{right}$ be libraries that each define a variable *bad* which is initialized to 0. If $\mathbb{L}_{left}$ and $\mathbb{L}_{right}$ have identical code, except for code blocks reachable only when *bad = 1* (e.g. guarded by an "*if bad = 1 then*" statement), then:

$$| Pr[A \diamond \mathbb{L}_{left} \Rightarrow 1] - Pr[A \diamond \mathbb{L}_{right} \Rightarrow 1] | \leq Pr[A \diamond \mathbb{L}_{left} \textit{ sets bad } = 1]$$

3.4 Birthday Probabilities & Sampling With/out Replacement

We consider the following libraries:

$\mathbb{L}_{samp-L}$
SAMP():
$r \leftarrow \{0,1\}^\lambda$
return r

$\mathbb{L}_{samp-R}$
$R = \emptyset$

SAMP():
$r \leftarrow \{0,1\}^\lambda \setminus R$
$R := R \cup \{r\}$
return r

Both libraries provide a SAMP method which will choose a random r out of the set $\{0,1\}^\lambda$. The right version will also keep track of what it will sample so it will not return the same value of r twice. The left version can return the same value twice.

The first thought about an algorithm that can distinguish between these two libraries is one that calls SAMP several times and searches for repeated values.
When the distinguisher tracks an repetition it is obviously linked to $\mathbb{L}_{samp-L}$. If you find no repeated value you may eventually stop and guess that you are linked to $\mathbb{L}_{samp-R}$.
We can know compute the advantage of such an adversary $\mathbb{A}_q$. Linked with the right version the algorithm's output will always be 0 because repeated value cannot be returned. The probability for 1 as an output when linked with $\mathbb{L}_{samp-L}$ is $BirthdayProb(q, 2^\lambda)$. If $\mathbb{A}_q$ is a polynomial-time algorithm q is therefore polynomial, too, and therefore the advantage is $BirthdayProb(q, 2^\lambda) = \Theta(q^2/2^\lambda)$ which is *negligible*!
But to show that both libraries are indistinguishable we have to show this *for all* calling programs. So we will modify our two libraries to get hybrid ones so we can use the *Bad Event Lemma*:

$\mathbb{L}_{hyb-L}$
$R := \emptyset$
$bad := 0$

SAMP():
$r \leftarrow \{0,1\}^\lambda$
if $r \in R$ *then*
bad = 1

$R = R \cup \{r\}$
return r

$\mathbb{L}_{hyb-R}$
$R := \emptyset$
$bad := 0$

SAMP():
$r \leftarrow \{0,1\}^\lambda$
if $r \in R$ *then*
bad = 1
$r \leftarrow \{0,1\}^\lambda \setminus R$
$R = R \cup \{r\}$
return r

We can see that $\mathbb{L}_{samp-L} \equiv \mathbb{L}_{hyb-L}$ because the added R and bad variables do not have an effect on the outside behaviour.
In $\mathbb{L}_{hyb-R}$ *rejection sampling* is used instead of sampling r directly out of $\{0,1\}^\lambda \setminus R$. This does not effect the outside behaviour and therefore it holds that $\mathbb{L}_{samp-R} \equiv \mathbb{L}_{hyb-R}$.

Because $\mathbb{L}_{hyb-L}$ and $\mathbb{L}_{hyb-R}$ only differ in a line of code which is only reachable when $bad = 1$ we can conclude:

$$\begin{aligned} |\ Pr[\mathbb{A} \diamond \mathbb{L}_{samp-L} \Rightarrow 1] \ -\ & Pr[\mathbb{A} \diamond \mathbb{L}_{samp-R} \Rightarrow 1]\ | \\ &= |\ Pr[\mathbb{A} \diamond \mathbb{L}_{hyb-L} \Rightarrow 1] - Pr[\mathbb{A} \diamond \mathbb{L}_{hyb-R} \Rightarrow 1]\ | \\ &\leq\ Pr[\mathbb{A} \diamond \mathbb{L}_{hyb-L} \textit{ sets } bad = 1] \end{aligned}$$

Finally we can conclude that $\mathbb{A} \diamond \mathbb{L}_{hyb-L}$ sets $bad = 1$ if it sees a repeated value which happens with a probability of $BirthdayProb(q, 2^\lambda)$.

4 Pseudorandom Generators

The idea is the Alice and Bob want to encrypt a message m of length 2λ but have agreed onto a key k of length λ. They know that for secrecy reasons the encryption key for one-time pad encryption must be the same length as the message (here 2λ). Also this key should be uniformly distributed but the generation of this key dependant of k should be *deterministic* so that sender and receiver compute the same value and decryption works correctly.
Let G denote the process that transforms k into this mystery value, so $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ and the encryption scheme is $Enc(k,m) = m \oplus G(k)$.
Because of the definition of G there will be 2^λ possible outputs of G, so G(k) cannot be uniform in $\{0,1\}^{2\lambda}$. Therefore this scheme is not secure in the same way as one-time pad.
The idea of **pseudorandomness** is therefore to get as close as possible to a uniform distribution so that no polynomial-time algorithm can distinguish the distribution of $G(k)$ values from the uniform distribution.

4.1 Definition

A **pseudorandom generator (PRG)** is a deterministic function G which transforms an input in a longer output. As can be seen above the distribution of the output values cannot be uniform over the whole output space. HOwever, the distribution is *pseudorandom* if it is indistinguishable from the uniform distribution. More formally:
Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+l}$ be a *deterministic* function with $l > 0$. We say that G is a **secure pseudorandom generator (PRG)** if $\mathbb{L}_{prg-real} \approx \mathbb{L}_{prg-rand}$, where:

$\mathbb{L}_{prg-real}$
QUERY():
$s \leftarrow \{0,1\}^\lambda$
return $G(s)$

$\mathbb{L}_{prg-rand}$
QUERY():
$r \leftarrow \{0,1\}^{\lambda+l}$
return r

The value l is called the ***stretch*** of the PRG. The input for the PRG is typically called a ***seed***.

4.2 Pseudorandom Generators in Practice

There are no real examples for **secure PRGs** because if you can prove this you would have solved the **P vs. NP Problem**. So there are only **candidate PRGs** which are *conjectured* to be secure. Furthermore PRGs as defined above are not really used but constructed from a *block cipher*.

4.2.1 How NOT to Build a PRG

The first idea for a PRG as described above would be the following length-goubling PRG (which is unfortunately insecure):

$G(s)$:
return $s \| s$

We will consider the following calling program $\mathbb{A}$:

Adversary $\mathbb{A}$
$x \| y = \text{QUERY}()$
return $x \overset{?}{=} y$

This adversary samples an output and checks whether the string has equal first and second halves.
When we link this adversary with $\mathbb{L}_{prg-real}$ it will always return 1. Linked to $\mathbb{L}_{prg-rand}$ the probability is $\frac{1}{2^\lambda}$. Therefore the advantage of this algorithm is non-zero and G is not a *secure* PRG.

4.3 Application: Shorter Keys in One-Time Secret Encryption

Going back to our initial problem, we will define the following encryption scheme:

$\mathbb{K} = \{0,1\}^\lambda$ $\mathbb{M} = \{0,1\}^{\lambda+l}$ $\mathbb{C} = \{0,1\}^{\lambda+l}$	KEYGEN(): $k \leftarrow \mathbb{K}$ **return** k	ENC(k, m): **return** $G(k) \oplus m$	DEC(k, c): **return** $G(k) \oplus c$

If this encryption scheme (called *pOTP*) is instantiated using a *secure PRG G* then this *pOTP* has **computational one-time secrecy**.
We have to show that $\mathbb{L}^{OTP}_{ots-L} \approx \mathbb{L}^{OTP}_{ots-R}$. To use that G is a secure PRG, we create hybrid libraries which use $\mathbb{L}^{G}_{prg-real}$ and replace it with $\mathbb{L}^{G}_{prg-rand}$ to show that both libraries are indistingushable (NOT interchangeable because of the definition of PRG security).

4.4 Extending the Stretch of PRG

We can use multiple PRGs which are connected in a row and use one half of an output string and uses it as a seed to generate a new key. Such an construction of PRGs which feed the following one with a seed is called a **stream cipher**.
A stream cipher PRG takes a seed s and a length l and outputs a string of this length l. Also if $i < j$ then $G(s, i)$ is a *prefix* of $G(s, j)$.

5 Pseudorandom Function & Block Ciphers

5.1 Definition

We imagine a large table T filled with shared random keys. Each key is therefore reference as $T[i]$. Seeing i as a binary string instead of an integer we can use an in-bit long i for a table with 2^{in} items. We can also see this process as a function taking an input from $\{0,1\}^{in}$ and giving an output from $\{0,1\}^{out}$.
A *pseudorandom function* (**PRF**) is emulating this huge table. Additionally it takes a second argument, (**seed**), which acts as a secret key.
The goal of an *PRF* is to behave like the following subroutine:

```
for x in {0,1}^in:
  T[x] ← {0,1}^out

LOOKUP(x ∈ {0,1}^in):
  return T[x]
```

This subroutine should be indistinguishable from the following PRF, when it is used with an *uniformly chosen* seed:

```
k ← {0,1}^λ

LOOKUP(x ∈ {0,1}^in):
  return F(k, x)
```

Because it can take a lot of time to instantiated an already filled table T, we can use a *lazy/on-demand* approach and only fill in its values when the calling programm requests them. This does not effect the outside behaviour of the table or how the values are sampled (*uniformly & independent*).
All in all we define:

Let $F : \{0,1\}^{\lambda} \times \{0,1\}^{in} \to \{0,1\}^{out}$ be a deterministic function. We call F a *secure* **pseudorandom function** (**PRF**) if $\mathbb{L}^{F}_{prf-real} \approx \mathbb{L}^{F}_{prf-rand}$, where:

$\mathbb{L}^{F}_{prf-real}$

```
k ← {0,1}^λ

LOOKUP(x ∈ {0,1}^in):
  return F(k, x)
```

$\mathbb{L}^{F}_{prf-rand}$

```
T := empty assoc. array

LOOKUP(x ∈ {0,1}^in):
  if T[x] undefined:
    T[x] ← {0,1}^out
  return T[x]
```

5.1.1 How NOT to Build a PRF

We will use the length doubling PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ and construct the following PRF:

$\underline{F(k,x)}$
return $G(k) \oplus x$

We will consider the following distinguisher $\mathbb{A}$:

$\mathbb{A}$
pick $x_1, x_2 \in \{0,1\}^{2\lambda}$ arbitrarily so that $x_1 \neq x_2$
$z_1 := \text{LOOKUP}(x_1)$
$z_2 := \text{LOOKUP}(x_2)$
return $z_1 \oplus z_2 \overset{?}{=} x_1 \oplus x_2$

Linking this distinguisher with $\mathbb{L}^F_{prf-real}$ will always result in a 1 as output, because on the left side we will compute $z_1 \oplus z_2 = G(k) \oplus x_1 \oplus G(k) \oplus x_2 = x_1 \oplus x_2$. Linking it with $\mathbb{L}^F_{prf-real}$ will result in a 1 as output with a probability of $1/2^{2\lambda}$. Therefore the advantage is non-zero and our chosen PRF is not *secure*.

5.2 Block Cipher (Pseudorandom Permutations)

A **pseudorandom permutation** or **block cipher** is a PRF with input from $\{0,1\}^{blen}$ and output from $\{0,1\}^{blen}$ that is guaranteed to be *invertible* for any seed. Because a pseudorandom function (**PRP**) is *invertible* it only have to be indistinguishable from a randomly chosen *invertible* function. Such a function is assigning the values while keeping track of which where already chosen such that it is sampling only value which were not assigned before.

Let $F : \{0,1\}^\lambda \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ be a determnisitic function. We refer to *blen* as the **blocklength** of F and any element of $\{0,1\}^{blen}$ as a **block**.
We call F a ***secure pseudorandom permutation (PRP) (block cipher)*** if the following two hold:

1. *(Invertible given k)*: There is a function $F^{-1} : \{0,1\}^\lambda \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ which satisfies:
   $F^{-1}(k, F(k,x)) = x \quad , \forall k \in \{0,1\}^\lambda \text{ and } \forall x \in \{0,1\}^{blen}$
2. *(Security)* $\mathbb{L}^F_{prp-real} \approx \mathbb{L}^F_{prp-rand}$, where:

$\mathbb{L}^F_{prp-real}$
$k \leftarrow \{0,1\}^\lambda$

$\underline{\text{LOOKUP}(x \in \{0,1\}^{blen}):}$
return $F(k,x)$

$\mathbb{L}^F_{prp-rand}$
$T :=$ empty assoc. array

$\underline{\text{LOOKUP}(x \in \{0,1\}^{blen}):}$
if $T[x]$ undefined:
$T[x] \leftarrow \{0,1\}^{blen} \setminus T.values$
return $T[x]$

5.3 Relating PRFs and Block Ciphers

5.3.1 Switching Lemma

Let $\mathbb{L}_{prp-rand}$ and $\mathbb{L}_{prf-rand}$ be defined as before with $in = out = blen = \lambda$ (so that the interfaces match up). Then $\mathbb{L}_{prp-rand} \approx \mathbb{L}_{prf-rand}$.

We know that the following two libraries are indistinguishable (replacement-sampling lemma):

$\mathbb{L}_{samp-L}$

SAMP():
$r \leftarrow \{0,1\}^\lambda$
return r

$\mathbb{L}_{samp-R}$

$R = \emptyset$

SAMP():
$r \leftarrow \{0,1\}^\lambda \setminus R$
$R = R \cup \{r\}$
return r

Using these with the follwing library $\mathbb{L}^\star$

$\mathbb{L}^\star$

T = empty assoc. array

LOOKUP()$x \in \{0,1\}^\lambda$:
if $T[x]$ is undefined:
$T[x] \leftarrow$ SAMP()
return $T[x]$

By linking $\mathbb{L}^\star$ with $\mathbb{L}_{samp-L}$ we obtain $\mathbb{L}_{prf-rand}$ and by linking $\mathbb{L}^\star$ with $\mathbb{L}_{samp-R}$ we obtain $\mathbb{L}_{prp-rand}$. Therefore with the replacement-sampling lemma we have shown that $\mathbb{L}_{prp-rand} \approx \mathbb{L}_{prf-rand}$.
Furthermore we can say that each *secure PRP* is also a *secure PRF*.

5.4 PRFs and Block Ciphers in Practice

5.5 Strong Pseudorandom Permutations

6 Security against Chosen Plaintext Attacks

Let Σ be an encryption scheme. We say that Σ has **security against chosen-plaintext attacks (CPA-security)** if $\mathbb{L}^{\Sigma}_{cpa-L} \approx \mathbb{L}^{\Sigma}_{cpa-R}$, where:

$\mathbb{L}^{\Sigma}_{cpa-L}$
$k \leftarrow \Sigma.\text{KEYGEN}()$
$\text{EAVESDROP}(m_L, m_R \in \Sigma.\mathbb{M})$:
$c := \Sigma.\text{ENC}(k, m_L)$
return c

$\mathbb{L}^{\Sigma}_{cpa-R}$
$k \leftarrow \Sigma.\text{KEYGEN}()$
$\text{EAVESDROP}(m_L, m_R \in \Sigma.\mathbb{M})$:
$c := \Sigma.\text{ENC}(k, m_R)$
return c

6.1 Limits of Deterministic Encryption

For any deterministic encryption we can use the following distinguisher to successfully attack this encryption:

Distinguisher $\mathbb{A}$
arbitrarily choose *distinct* plaintexts $x, y \in \Sigma.\mathbb{M}$
$c_1 = \text{EAVESDROP}(x, x)$
$c_2 = \text{EAVESDROP}(x, y)$
return $c_1 \stackrel{?}{=} c_2$

As encrypting the same plaintext with the same key using a deterministic encryption scheme will always result in the same result this distinguisher will always output 1 if it is linked to $\mathbb{L}^{\Sigma}_{cpa-L}$ and always 0 if it is linked ro $\mathbb{L}^{\Sigma}_{cpa-R}$. Therefore its advantage is 1 and every deterministic encryption scheme is therefore not *cpa-secure*.

6.1.1 Avoiding Deterministic Encryption

There are 3 general ways to design an encryption scheme that is not deterministic:

1. The encryption/decryption can be **stateful**, meaning that every call to ENC and DEC will actually *modify* the value of k.
2. The encryption can be **randomized** so that for each plaintext the encryption algorithm chooses fresh, independent randomness. This kind of encryption has the challenge that the decryption is difficult to implement.
3. Encryption can be **nonce-based**(nonce stands for *number used only once*). It refers to an extra argument which is passed to the ENC and DEC algorithm. It does not need to be random or secret but it have to be *distinct* for every call of ENC.

6.2 Pseudorandom Ciphertexts

For *CPA-security* we can modify our security definition by saying that the encryption of m looks uniform. This means that also multiple encryptions look *jointly* pseudorandom and independent (what was an issue with our *deterministic* PRFs and PRGs).

We say that an encryption scheme Σ has **pseudorandom ciphertexts in the presence of chosen plaintext attacks (CPA$-security)** if $\mathbb{L}^{\Sigma}_{cpa\$-real} \approx \mathbb{L}^{\Sigma}_{cpa\$-real}$, where:

$\mathbb{L}^{\Sigma}_{cpa\$-real}$

$k \leftarrow \Sigma.\text{KEYGEN}()$

$\underline{\text{CTXT}(m \in \Sigma.\mathbb{M}):}$
$c := \text{ENC}(k, m)$
return c

$\mathbb{L}^{\Sigma}_{cpa\$-real}$

$\underline{\text{CTXT}(m \in \Sigma.\mathbb{M}):}$
$c \leftarrow \Sigma.\mathbb{C}$
return c

If an encryption scheme is *CPA$-secure* it is also *CPA-secure*.

6.3 CPA-Secure Encryption Based on PRFs

We consider the previous example where Alice and Bob shared a huge table T initialized with uniform data. Alice can now encrypt a plaintext to Bob by saying something like "this is encrypted with one-time pad, using key #69875398" and sending $T[69875398] \oplus m$. Seeing this number an eavesdropper cannot get any information about $T[69875398]$. An PRF can be also used in allowing Alice and Bob encrypt messages with a short key r because $F(k, r)$ cannot be predicted by an adversary if k is secret. All in all an encryption of m using such a PRF will look like $(r, F(k, r) \oplus m)$.
In the end it has to be decided how r is chosen, whether *stateful, randomized or nonce-based*.

An encryption can therefore look as the following:

$\mathbb{K} = \{0,1\}^{\lambda}$
$\mathbb{M} = \{0,1\}^{out}$
$\mathbb{C} = \{0,1\}^{\lambda} \times \{0,1\}^{out}$

$\underline{\text{KEYGEN}():}$
$k \leftarrow \{0,1\}^{\lambda}$
return k

$\underline{\text{ENC}(k, m):}$
$r \leftarrow \{0,1\}^{\lambda}$
$x := F(k, r) \oplus m$
return x

$\underline{\text{DEC}(r, x)}$
$m := F(k, r) \oplus x$
return m

If an secure PRF F is used this encryption has *CPA$-security* (and therefore also CPA-security).

7 Block Cipher Modes of Operation

7.1 A Tour of Common Modes

7.1.1 ECB: Electronic Codebook

The most obvious way to use a block cipher is to encrypt every block independently from each other. This mode of block cipher should NEVER be used because it does not provide any security.

7.1.2 CBC: Cipher Block Chaining

Cipher Block Chaining (CBC) is the most common block cipher used today. The encryption of an l-block plaintext is $l + 1$ blocks long. The first ciphertext block is called an **initialization vector (IV)** which can be chosen randomly (but is not sufficient). This mode of block cipher is *CPA-secure*.

7.1.3 CTR: Counter

This mode also involves an IV block r which is chosen uniformly. The idea is then to use the sequence:

$$F(k,r)\ ;\ F(k,r+1)\ ;\ F(k,r+2)\ ...$$

as a long one-time pad to mask the plaintext.

7.1.4 OFB: Output Feedback

Output Feedback involves an IV r as well, but uses the sequence:

$$F(k,r)\ ;\ F(k,F(k,r)\ ;\ F(k,F(k,F(k,r)))\ ...$$

as a long one-time pad to mask the plaintext.

7.1.5 Compare & Contrast

CBC and CTR are the most commonly used block ciphers and provide the same security guarantees. So any conmparison between them must be based on factors outside of the CPA security definition:

1. CTR's decryption schmeme do not need F^{-1} and can therefore be instantiated with an PRF rather than a PRP (but it is rare to encounter a secure PRF that is not a PRP)
2. The encryption of CTR can be parallelized, which is impossible for CBC. Both have parallelizable decryption schemes.
3. Since only the IV affects the encryption of CTR it can be pre-computed which is not possible in CBC because also the plaintext affects the encryption.
4. It is relatively easy to modify a CTR to support plaintexts that are not a multiple of the block length. For CBC it is not so trivial.
5. If the IV is reused CBC can still be as secure as befor, because the encryption also depends on the plaintext itself. For CTR it could be quite devastating.

7.2 CPA Security for Variable-Length Plaintexts

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

Adversary $\mathbb{A}$
$c := \text{EAVESDROP}(0^\lambda, 0^{2\lambda})$
return $\mid c \mid \stackrel{?}{=} 2\lambda$

We can see that each encryption **leaks information about the number of blocks in the plaintext**. To hide also this little information we can modify our libraries just that they will disallow this kind of query and give an error message if $\mid m_L \mid \neq \mid m_R \mid$.

7.2.1 Don't Take Length-Leaking for Granted

We said that we don't really care about casually leaking some minor information about the plaintext (e.g. the length of it). Also with such little information some attacks can still be possible:

1. Using Google Maps many image tiles are received. Each of these tiles are compressed to save ressources (but all are not equally compressed). Every region has a specific "fingerprint" of image-tile length. So by leaking the information of length some attacker can determine which part of the world a user is requesting.
2. In audio/video encoding variable-bit-rate (VBR) encoding is used. If the data stream carries less information, it is encoded at a lower bit rate. So an attacker can determine whether a user is watching a Netflix or youtube video with a constant and predictable bit stream opposed to a video live stream.
3. VBR is also used in voice chat programs. An attacker can determine by observing the stream of ciphertext sizes who was speaking (from a set of candidates).

8 Diffie-Hellman Key Agreement

8.1 Cyclic Groups

Let $g \in \mathbb{Z}_n^\star$. Define $\langle g \rangle_n = \{g^i \% n \mid i \in \mathbb{Z}_n^\star$, the set of all powers of g reduced mod n. Then g is called a **generator** of $\langle g \rangle_n$, and $\langle g \rangle_n$ is called the **cyclic group generated by** g **mod** n.
If $\langle g \rangle_n = \mathbb{Z}_n^\star$, then we say g is a **primitive root mod** n.

8.1.1 Example

Taking $n = 13$, we have:

$$\begin{aligned}\langle 1 \rangle_{13} &= \{1\} \\ \langle 2 \rangle_{13} &= \{1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7\} = \mathbb{Z}_n^\star \\ \langle 3 \rangle_{13} &= \{1, 3, 9\}\end{aligned}$$

Therefore 2 is a *primitive root modulo 13*. Each of those groups $\{1\}, \mathbb{Z}_n^\star, \{1, 3, 9\}$ is a *cyclic group under multiplication mod 13*.
A cyclic group might have more than one generator:

$$\langle 3 \rangle_{13} = \langle 9 \rangle = \{1, 3, 9\}$$

Not every integer has a primitive root (for example $n = 15$), but if n is prime there will exist at least one.

Discrete Logarithm

The **discrete logarithm problem** is: given $X \in \langle g \rangle$, determine a number x such that $g^x = X$. Here the exponentiation is with respect to the multiplication operation in $\mathbb{G} = \langle g \rangle$.
The discrete logarithm problem is considered to be *hard* so there exists no polynomial-time algorithm for this problem.

8.2 Diffie-Hellman Key Agreement

Key agreement refers on the problem that Alice and Bob want to share a secret key by exchanging public messages.
The Diffie-Hellman protocol says the following:

Both parties agree (publicly) on a cyclic group $\mathbb{G}$ with generator g. Let $n = |\ \mathbb{G}\ |$. All exponentations are with respect to the group operations in $\mathbb{G}$.

1. Alice chooses $a \leftarrow \mathbb{Z}_n$. She sends $A = g^a$ to Bob.
2. Bob chooses $b \leftarrow \mathbb{Z}_n$. He send $B = g^b$ to Alice.
3. Bob locally outputs $K := A^b$. Alice locally ouptputs $K := B^a$. K is the secret key namely $g^{ab} \in \mathbb{G}$.

8.3 Decisional Diffie-Hellman Problem

The **decisional Diffie-Hellman (DDH) assumptions** in a cyclic group $\mathbb{G}$ is that $\mathbb{L}^{\mathbb{G}}_{dh-real} \approx \mathbb{L}^{\mathbb{G}}_{dh-rand}$, where:

$\mathbb{L}^{\mathbb{G}}_{dh-real}$
$a, b \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^{ab})

$\mathbb{L}^{\mathbb{G}}_{dh-rand}$
$a, b, c \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^c)

8.3.1 For Which Groups does the DDH Assumption Hold?

So far our only example for a cyclic group is $\mathbb{Z}^\star_p$ where p is prime. For this case the DDH example is false because of the following claim:

$$\text{If } p \text{ is prime and } X = g^x \in \mathbb{Z}^\star_p \text{ then } X^{\frac{p-1}{2}} \equiv_p (-1)^x.$$

So therefore we can easily proof whether x is even or odd.

Quadratic Residues

A number X is a **quadratic residue modulo** n if there exists some integer Y such that $Y^2 \equiv_n X$. That is, if X can be obtained by squaring a number mod n. Let $\mathbb{QR}^\star_p \subseteq \mathbb{Z}^\star_p$ denote the set of quadratic residues mod n.

If p is prime then $\mathbb{QR}^\star_p$ is a cyclic group with $(p-1)/2$ elements. When both p and $(p-1)/2$ are prime then we call p a **safe prime** (and $(p-1)/2$ a *Sophie-Germain prime*. If p is a safe prime the DDH assumption is true for $\mathbb{QR}^\star_p$.

9 Public-Key Encryption

So far all encryption schemes we have seen were **symmetric**. Now we will look at **public-key** (sometimes called **asymmetric**) encryptions.
Therefore we have to modify the syntax of our encryption/decryption schmes:

1. KEYGEN: Outputs a pair (pk, sk) where pk is a public key and sk is the private/secret key.
2. ENC: Takes the public key pk and a plaintext m and outputs a ciphertext c.
3. DEC: Takes the secret key sk and a ciphertext c and outputs a plaintext m.

The correctness condition is modified similarly (so that $\text{DEC}(sk, \text{ENC}(pk, m)) = m$).

9.1 Security Definitions

9.1.1 Chosen-Plaintext Attacks Security (CPA-Security

We now modify the definition of CPA security. Because we want pk to be public we have to inline a subroutine such that any attacker can learn about the public key. With this information we can define the following:

Let Σ be a *public-key encryption scheme*. Then Σ is **secure against chosen plaintext attack (CPA-secure)** if $\mathbb{L}^{\Sigma}_{pk-cpa-L} \approx \mathbb{L}^{\Sigma}_{pk-cpa-R}$, where:

$\mathbb{L}^{\Sigma}_{pk-cpa-L}$
$(pk, sk) \leftarrow \Sigma.\text{KEYGEN}()$

GETPK:
return pk

CHALLENGE$(m_L, m_R \in \Sigma.\mathbb{M})$:
return $\Sigma.\text{ENC}(pk, m_L)$

$\mathbb{L}^{\Sigma}_{pk-cpa-R}$
$(pk, sk) \leftarrow \Sigma.\text{KEYGEN}()$

GETPK:
return pk

CHALLENGE$(m_L, m_R \in \Sigma.\mathbb{M})$:
return $\Sigma.\text{ENC}(pk, m_R)$

9.1.2 Pseudorandom Ciphertexts

We can also modify/adapt the definition of pseudorandom ciphertexts to public-key encryption in a similar way:

Let Σ be a *public-key encryption scheme*. Then Σ has **pseudorandom ciphertexts in the presence of chosen-plaintext attacks (CPA$-secure)** if $\mathbb{L}^{\Sigma}_{pk-cpa\$-real} \approx \mathbb{L}^{\Sigma}_{pk-cpa\$-rand}$, where:

$\mathbb{L}^{\Sigma}_{pk-cpa\$-real}$
$(pk, sk) \leftarrow \Sigma.\text{KEYGEN}()$

GETPK:
return pk

CHALLENGE$(m \in \Sigma.\mathbb{M})$:
return $\Sigma.\text{ENC}(pk, m)$

$\mathbb{L}^{\Sigma}_{pk-cpa\$-rand}$
$(pk, sk) \leftarrow \Sigma.\text{KEYGEN}()$

GETPK:
return pk

CHALLENGE$(m_L, m_R \in \Sigma.\mathbb{M})$:
$c \leftarrow \Sigma.\mathbb{C}$
return c

As before CPA$-security implies CPA-security.

9.2 One-Time Security Implies Many-Time Security

We need to ensure that every public key is used to encrypt only one plaintext. Therefore we have to modify our libraries.

Let Σ be a public-key encryption scheme. We say that Σ has **one-time secrecy** if $\mathbb{L}^{\Sigma}_{pk-ots-L} \approx \mathbb{L}^{\Sigma}_{pk-ots-R}$, where:

$\mathbb{L}^{\Sigma}_{pk-ots-L}$

$(pk, sk) \leftarrow \Sigma.\text{KEYGEN}()$
$count := 0$

GETPK():
return pk

CHALLENGE($m_L, m_R \in \Sigma.\mathbb{M}$):
$count := count + 1$
if $count > 1$: **return** null
return $\Sigma.\text{ENC}(pk, m_L)$

$\mathbb{L}^{\Sigma}_{pk-ots-R}$

$(pk, sk) \leftarrow \Sigma.\text{KEYGEN}()$
$count := 0$

GETPK():
return pk

CHALLENGE($m_L, m_R \in \Sigma.\mathbb{M}$):
$count := count + 1$
if $count > 1$: **return** null
return $\Sigma.\text{ENC}(pk, m_R)$

9.3 ElGamal Encryption

The ElGamal encryption is a public-key encryption scheme that is based on DHKA:

$\mathbb{M} = \mathbb{G}$
$\mathbb{C} = \mathbb{G}^2$

KEYGEN():
$sk := a \leftarrow \mathbb{Z}_n$
$pk := A := g^a$
return (pk, sk)

ENC($A, M \in \mathbb{G}$):
$b \leftarrow \mathbb{Z}_n$
$B := g^b$
return $(B, M \cdot A^b)$

DEC($a, (B, X)$):
return $X(B^a)^{-1}$

9.3.1 Security

We imagine an adversary that sees the public key A and the encrypted message $(B, M \cdot A^b)$. The decisional Diffie-Hellman assumptions says that g^{ab} still looks random even if g^a and g^b are known. Therefore the message is masked with a pseudorandom group element (only difference is that we use the group operation of $\mathbb{G}$). Therefor if the DDH assumption holds the ElGamal encryption in group $\mathbb{G}$ is *CPA$-secure*.

10 RSA & Digital Signatures

10.1 "Dividing" Mod N

10.2 The RSA Function

The RSA function is defined as follows:

1. Let p and q be distinct primes and $N = p \cdot q$. N is called **RSA modulus**. Additionally $\Phi(N) = (p-1)(q-1)$.
2. Let e and d be integers such that $ed \equiv_{\Phi(N)} 1$.
3. The RSA function is then: $x \to x^e \ \% \ N$, where $x \in \mathbb{Z}_N$
4. The inverse RSA function is then: $y \to y^d \ \% \ N$, where $y \in \mathbb{Z}_N$

10.3 The Hardness of Factoring N