# CS 387
# Applied Cryptography

David Evans

written by

Daniel Winter

special thanks to:

Wolfgang Baltes

16.04.2012

# Contents

# 1 Symmetric Ciphers

## 1.1 Cryptology, Symmetric Cryptography & Correctness Property

**Definition** *cryptography, cryptology*
*cryptography* comes from Greek with *crypto* means "hidden, secret" and *graphy* means "writing". A broader definition is *cryptology* with Greek "-logy" means "science".
**Example** 1:
These actions involve cryptology:

- Opening a door
- Playing poker
- Logging into an internet account

.
**Definition** *symmetric Cryptography*
*Symmetric Cryptography* means all parties have the same key to do encryption and decryption. The keys may be identical or there may be a simple transformation to go between the two keys.
**Definition** *symmetric Cryptosystem, decryption function, encryption function*
In this paper a *symmetric Cryptosystem* always looks as follows:

$$\begin{tikzcd}
m \arrow[r] & E \arrow[r, squiggly, "c"] & X \arrow[r, squiggly, "c"] & D \arrow[r] & m \\
& k \arrow[u] & & k \arrow[u] &
\end{tikzcd}$$

where

$m$ is a plaintext message from the set $\mathcal{M}$ of all possible messages,

$k$ is the key from the set $\mathcal{K}$ of all possible keys and

$c$ is a ciphertext from the set $\mathcal{C}$ of all possible Ciphertexts.

$E$ ist the *encryption function*,

$X$ is a possible eavesdropper of the insecure channel and

$D$ is the *decryption function* with:

$$\begin{aligned} E &: M \times K \to C : (m,k) \mapsto c \\ D &: C \times K \to M : (c,k) \mapsto m \end{aligned}$$

.
**Definition** *correctness property*
In order we get the same message after decrypting the ciphertext (encrypted message) we need the *correctness property*: $\forall m \in \mathcal{M}, \forall k \in \mathcal{K}$

$$D_k(E_k(m)) = m$$

**Example** 2:
These functions satisfy the correctness property for a symmetric cryptosystem:

1. $E_k(m) = m + k, D_k(c) = c - k$ with $\mathcal{M} = \mathcal{K} = \mathbb{N}$ because

$$D_k(E_k(m)) = D_k(m + k) = m + k - k = m$$

2. $E_k(m) = am + b, D_k = a^{-1}(c - b)$ with $\mathcal{M} = \mathcal{K} = \mathbb{Z}/\mathbb{Z}m, a \in (\mathbb{Z}/\mathbb{Z}m)^\times$ because

$$D_k(E_k(m)) = D_k(m) = m$$

.

## 1.2 Kerchoff's Principle & *xor*-function

**Theorem 1.2.1 (Kerchoff's Principle)**
*The general assumption is that the channel for message transmission is not secure.*
*Even if the encryption function E and decryption function D are public, the message is still secure due to a usage of a secret. Only the key has to be secret. If the key gets public you have to use another key. So the keys must be kept secret in a cryptosystem.*

**Definition** *xor function*
the *xor function* or *exclusive or* (symbol: $\oplus$) is given by it's truth table:

| $A$ | $B$ | $A \oplus B$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Useful properties $\forall x, \forall y, \forall z$ of the *xor* function are:

- Distributivity

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

- Commutativity

$$x \oplus y = y \oplus x$$

- Negation

$$x \oplus 1 = \overline{x}$$

- Identity

$$x \oplus x = 0$$

which are applied in ciphers.
It follows

$$x \oplus y \oplus x = x \oplus x \oplus y = y \Leftrightarrow m \oplus k = c \text{ and } c \oplus k = m$$

that the *xor function* is kommutative and assoziative and how to compute the ciphertext $c$ with the message $m$ and the key $k$ and decrypt the ciphertext $c$ with the same key $k$ to get $m$.

## 1.3 One - Time Pad

**Definition** *One - Time Pad*
Let's assume the set of all possible messages $\mathcal{M} := \{0,1\}^n$ of lenght $n$. That means every message is represented as a string of zeros and ones with fixed length $n \in \mathbb{N}$. Any key $k \in \mathcal{K}$ has to be as long as the message $m \in \mathcal{M}$. It follows that $\mathcal{K} := \{0,1\}^n$. The encryption function with $m = m_0 m_1 \ldots m_{n-1}$ and $k = k_0 k_1 \ldots k_{n-1}$ looks as follows:

$$E : M \times K \to C : (m,k) = m \oplus k = c_0 c_1 \ldots c_{n-1} = c$$

with the value of each bit of the ciphertext defined for all $i$:

$$c_i = m_i \oplus k_i$$

**Example** 3:
Let assume our message is $m =$'CS'. We have to convert the string to an element of $\mathcal{M} := \{0,1\}^n$ where $n = 7$. That means every charakter in every message is represented by 7 bits. `Python` provides a built-in function `ord(<one charakter string>)` which maps every charakter to a specific dezimal number. The code for converting a string to a valid message looks as follows:

```
def convert_to_bits(n,pad):
   result = []
   while n > 0:
      if n % 2 == 0:
         result = [0] + result
      else:
         result = [1] + result
      n = n / 2
   while len(result) < pad:
      result = [0] + result
   return result

def string_to_bits(s):
   result = []
   for c in s:
      result = result + convert_to_bits(ord(c),7)
   return result

string_to_bits(’CS’) => [1,0,1,0,0,1,1,1,0,0,0,0,1,1]
```

and it follows with a random choosen key $k$:

$$\begin{array}{rcl} m = \text{'CS'} & = & \overbrace{1010011}^{\text{C}}\overbrace{1000011}^{\text{S}} \\ & & \oplus \\ k & = & 11001000100110 \\ m \oplus k = c & = & 01101111100101 \end{array}$$

If someone got the ciphertext but not the key - the person is not able to figure the original message out. Taking $c$ and another key $k = 11001010100110$ and trying to get the message

$\forall i \in \{0, 1, \ldots, n-1\}$:
$$c_i \oplus k_i = m_i \Rightarrow m = 10100101000011$$

if $m$ is separated in 2 parts of length 7: 1010010 and 1000011, convert each to a decimal number and apply the built-in `Python` function `chr(<number>)=ascii charakter` the result is 'BS' instead of the correct string 'CS'.
**Example** 4:
Suppose an eavesdropper $X$ knows $A$ is sending $B$ a message $m \in \{00, 01, 10, 11\}$ using a *one-time pad* were the key $k$ is a perfectly random key unknown to $X$ and the distribution of messages is uniform - each message is equally likely. Then the conditional probability

$$P[m = 01 | X \text{ intercepts } c = 01] = 0.25$$

because since $m$ is encrypted using a *one-time-pad* we gain no information about $m$ from $c$, therefore:
$$P[m = 01 | X \text{ intercepts } c = 01] = P[m = 01] = \frac{1}{4} = 0.25$$
Additionally, suppose $X$ learns that $A$ generated $k$ using a biased random number generator that outputs 0 with 0.52 probability. Then

$$P[m = 01 | X \text{ intercepts } c = 01] = 0.2704$$

for $c = 01$ to map to $m = 01$ the key has to be $k = 00$. It follows

$$P[k = 00] = 0.52 \cdot 0.52 = 0.2704$$

## 1.4 Probability

**Definition** *Probability Space* $\Omega$
The *probability space* $\Omega$ is the set of all possible outcomes $(\omega_i)_{i \in I}$.
**Example** 5:

- Flipping a coin. The coin can land on Head $H$, Tail $T$ or Edge $E$, hence $\Omega = \{H, T, E\}$
- Rolling a die. The outcomes are $1, 2, 3, 4, 5$ or 6, hence $\Omega = \{1, 2, 3, 4, 5, 6\}$

.
**Definition** *Bernoulli Trial*
A *Bernoulli Trial* is an experiment whose outcome is random and can be either of two possible outcomes:

- Success ($S$)
- Failure ($F$)

As probabilities, we assign

$p$ as the **probability for success**

and

$1-p$ as the **probability for failure**

.

**Definition** *Geometric Distribution*
As experiment, we perform a *Bernoulli trial* until success. This meas $k-1$ trials fail and success at the $k^{th}$ trial for $k = 1, 2, \ldots, \infty$. The probability space $\Omega$ is never changed by any trial. It follows:

- Using the symbols $S$ for success, $F$ for failure and $F^n S := \overbrace{FF\ldots F}^{n \text{ times}} S$ then $\Omega$ is an infinite set with
$$\Omega = \{S, FS, FFS, FFFS, F^4S, F^5S, \ldots\}$$
- Since successive trials are independent, the probability distribution is given by
$$P(F^n S) = P(F)^n \cdot P(S) = (1-p)^n \cdot p$$
- The average number of trails until success is
$$\sum_{k=1}^{\infty}(1-p)^{k-1} \cdot p \cdot k = -p \cdot \frac{d}{dp}\left(\sum_{k=1}^{\infty}(1-p)^k\right) = -p \cdot \frac{d}{dp}\left(1 - \frac{1}{p}\right) = \frac{1}{p}$$

.

**Definition** *Uniform Distribution*
A *uniform distribution* has a underlying probability space $\Omega$ a finite set, e.g. $\Omega = \{\omega_1, \omega_2, \ldots, \omega_r\}$ with $r$ elements, and a probability measure $P$ that has (by definition) the property

$$P(\{\omega_i\}) = \frac{1}{r} = \frac{1}{|\Omega|}$$

**Example** 6:
Assume $\Omega = \{H, T\}$ for *Head* and *Tail* (without Edge) then

$$P(H) = P(T) = \frac{1}{2}$$

**Example** 7:
Assume a die with $\Omega = \{1, 2, 3, 4, 5, 6\}$ then

$$\forall i \in \Omega : P(i) = \frac{1}{6}$$

**Example** 8:
Another example is given by a similar, but slightly different experiment to that one described for the geometric distribution which deals with a set $\Omega_0 := \{\omega_1, \omega_2, \ldots, \omega_N\}$ of $N$ numbers. Let's assume that exactly one of the numbers $\omega_i$ be prime. Again, we perform a draw until success, where success means 'we draw a prime'. Additionally, in case the number drawn is not prime, we don't place this number back into $\Omega$, what will effect the probability space $\Omega$ for the next draw.
The probability-distribution of this experiment's outcomes is the *uniform distribution.* It follows

- The probability space $\Omega$ using $S$ for success, $F$ for failure and $F^nS := \overbrace{FF\dots F}^{n \text{ times}}S$ then $\Omega$ is the finite set:

$$\Omega = \{S, FS, FFS, FFFS, F^4S, F^4S, \dots, F^{n-1}S\}$$

- Successive trials reduce the actual probability space (each time by one element), so we evaluate the outcome's probabilities as follows

$$\begin{aligned} P(S) &= \frac{1}{N} \\ P(FS) &= \frac{N-1}{N} \cdot \frac{1}{N-1} = \frac{1}{N} \\ P(F^{n-1}S) &= \frac{N-1}{N} \cdot \frac{N-2}{N-1} \cdot \frac{N-3}{N-2} \cdot \dots \cdot \frac{1}{2} \cdot \frac{1}{1} = \frac{1}{N} \end{aligned}$$

hence $\forall k \in \{0, 1, 2, \dots, n-1\}$

$$P(F^kS) = \frac{1}{N}$$

- The average number of trials until success is

$$\sum_{i=1}^{n} P(F^{i-1}S) \cdot i = \sum_{i=1}^{n} \frac{1}{N} \cdot i = \frac{N \cdot (N+1)}{2N} = \frac{N+1}{2}$$

.

**Definition** *Event*

An *Event* $\mathcal{A}$ is a subset of the probability space $\Omega$ (i.e. $\mathcal{A} \subset \Omega$). In this course $\mathcal{A}$ consists of finite or infinitely many outcomes.

**Example** 9:

An event $\mathcal{A}$ of tossing a coin would be landing on head, therefore $\mathcal{A} = \{H\}$. The event $\mathcal{B}$ a valid coin toss is considered as $\mathcal{B} = \{H, T\}$

**Definition** *Probability Measure, certain event, impossible event*

The *Probability Measure* is a function that maps each outcome to a non-negative value lower-equal than 1. That means

$$P : \Omega \longrightarrow [0, 1] : \omega \mapsto P(\omega)$$

where $\omega \in \Omega$ is an outcome.

If $P(A) = 1$ then the event $A$ is called *certain event.* Recall that also $P(\Omega) = 1$, but in general it holds $A \neq \Omega$. And similar behavior we have for *impossible events*, i.e. in case of $P(B) = 0$ it may be that $B \neq \emptyset$ (only for 'elementary' probability spaces it holds $A = \Omega$ and $B = \emptyset$).

**Example** 10:

Roll 7 on a die is an impossible event, because $\Omega = \{1, 2, 3, 4, 5, 6\}$ and

$$P(\{7\}) = \frac{0}{6} = 0$$

On the other hand rolling a $1, 2, 3, 4, 5$ or $6$ with a die is

$$P(\{1, 2, 3, 4, 5, 6\}) = P(\Omega) = \frac{6}{6} = 1$$

**Theorem 1.4.1:**

*Assume the probability space $\Omega$ and the probability function $p$. Then it holds*

$$\sum_{\omega \in \Omega} P(\omega) = 1$$

**Example** 11:
Let's assume $\Omega = \{H, T, E\}$ with probabilities

$$\begin{aligned} P(H) &= 0.49999 \\ P(T) &= 0.49999 \end{aligned}$$

The probability for edge $E$ is given by

$$1 = P(\Omega) = P(H) + P(T) + P(E) = 0.49999 + 0.49999 + P(E) \Rightarrow P(E) = 0.00002$$

**Theorem 1.4.2:**
*The probability of an event $\mathcal{A}$ is given by*

$$P(\mathcal{A}) = \sum_{\omega \in \mathcal{A}} P(\omega)$$

**Example** 12:
The probability for a valid coin toss $\mathcal{B} = \{H, T\}$ is

$$P(\mathcal{B}) = P(H) + P(T) = 0.49999 + 0.49999 = 0.99998$$

**Definition** *Conditional Probability*
Given two events, $\mathcal{A}$ and $\mathcal{B}$, in the same probability space $\Omega$, the *conditional probability* of $\mathcal{B}$, given that $\mathcal{A}$ occured is:

$$P(\mathcal{B}|\mathcal{A}) = \frac{P(\mathcal{A} \cap \mathcal{B})}{P(\mathcal{A})} \tag{1.1}$$

**Example** 13:
Given that a coin toss is valid, the probability it is heads is given with $\mathcal{A} = \{H, T\}$ is the event that a coin toss is valid and $\mathcal{B} = \{H\}$ is the event that the coin toss is heads. It follows with $P(\mathcal{A}) = P(\{H, T\}) = P(\{H\}) + P(\{T\}) = 0.49999 + 0.49999 = 0.99998$ and $P(\mathcal{B}) = P(\{H\}) = 0.49999$:

$$P(\mathcal{B}|\mathcal{A}) = \frac{P(\mathcal{A} \cap \mathcal{B})}{P(\mathcal{A})} = \frac{P(\{H, T\} \cap \{H\})}{P(\{H, T\})} = \frac{P(\{H\})}{P(\{H, T\})} = \frac{0.49999}{0.99998} = \frac{1}{2}$$

**Example** 14:
The relative frequencies of the vowels in English, as a percentage of all letters in a sample of typical English text:

e: 13%, a: 8%, o: 7%, i: 7%, u: 3%

For the letter $x$ drawn randomly from the text, it is

- $P(x$ is a vowel):

$$P(x \in \{a, e, i, o, u\}) = 0.13 + 0.08 + 0.07 + 0.07 + 0.03 = 0.38$$

- $P(x$ is 'e'$|x$ is a vowel) :

$$\begin{aligned} P(x=e|x\in\{a,e,i,o,u\}) &= \frac{P(x=e\cap x\in\{a,e,i,o,u\})}{P(x\in\{a,e,i,o,u\})} \\ &= \frac{P(x=e)}{P(x\in\{a,e,i,o,u\})} \\ &= \frac{0.13}{0.38}=0.34 \end{aligned}$$

- $P(x$ is a vowel$|x$ is not $a\}$ :

$$\begin{aligned} P(x\in\{a,e,i,o,u\}|x\neq a) &= \frac{P(x\in\{a,e,i,o,u\}\cap x\neq a)}{P(x\neq a)} \\ &= \frac{P(x=e)+P(x=i)+P(x=o)+P(x=u)}{1-P(x=a)} \\ &= \frac{0.13+0.07+0.07+0.03}{1-0.08}=\frac{0.3}{0.92}=0.32 \end{aligned}$$

.

## 1.5 Secret Sharing

**Definition** *Secret Sharing*
A useful property of *xor* is that it can be used to share a secret (message) amount at least 2 (yourself and 2 others) people as follows:

1. The secret message of length $n$ is
$$x=x_0x_1x_2\dots x_{n-1}$$
2. Generate a random key $k\in\{0,1\}^n$ :
$$k=k_0k_1k_2\dots k_{n-1}$$
3. Compute
$$c=k\oplus x$$
with $\forall i\in\{0,1,2,\dots,n-1\}:c_i=k_i\oplus x_i$
4. Give $c$ and $k$ to different person and keep x yourself.

The message can only be decryptet by computing $c\oplus k$. So the people with the information $c$ and $k$ may not meet.

**Theorem 1.5.1:**
*To share a n bit long secret x amoung m people it is needed to compute* $(m-1)\cdot n$ *key bits (equally: to compute* $m-1$ *keys).*

**Proof:**
Show that $m-1$ keys are enough to share a secret securely amoung $m$ people with the property that only all $m$ people together can decrypt the message.
Compute $m-1$ keys:
$$k_1,k_2,\dots,k_{m-1}$$

with $\forall i \in \{1, 2, \ldots, m-1\} : k_i \in \{0,1\}^n$.
Then for $m$ people $p_i$ with $i \in \{0, 1, \ldots, m-1\}$ the information maps as follows:

$$\begin{array}{rcl} x \oplus k_1 \oplus k_2 \oplus \cdots \oplus k_{m-1} & \mapsto & p_0 \\ k_1 & \mapsto & p_1 \\ k_2 & \mapsto & p_2 \\ & \vdots & \\ k_{m-1} & \mapsto & p_{m-1} \end{array}$$

so every person $p_i$ with $i \in \{0, 1, 2, \ldots, m-1\}$ gets an information, that the secret is perfectly shared. Thus, every key holds $n$ bits. It follows with $m-1$ keys hold $(m-1) \cdot n$ bits together. Therefore

$$(m-1) \cdot n$$

are needed to compute (choose randomly) are spreeded amoung $m$ people to provide secrecy amoung $m$ people. □

## 1.6 Perfect Cipher

**Definition** *perfect cipher*
The ciphertext provides an attacker with **no** additional information about the plaintext. Assume $m, m^* \in \mathcal{M}, k \in \mathcal{K}, c \in \mathcal{C}$. The property for a *perfect cipher* is given by

$$P[m = m^* | E_k(m) = c] = P[m = m^*] \tag{1.2}$$

That means: for an attacker/eavesdropper the probability that $m = m^*$ without knowing the ciphertext is equal $m = m^*$ with knowing the ciphertext. The property

$$P[m = m^* | E_k(m) = c] = \frac{1}{|\mathcal{M}|}$$

where $|\mathcal{M}|$ is the cardinality of $\mathcal{M}$ (number possible messages). This woult be correct if, a priori, the attacker knew nothing about the messages, therefore all messages are equally likely (whats obviously not correct - not all sentences make sense).
**Theorem 1.6.1:**
*The one-time pad is a perfect cipher.*

**Proof:**
Remember the perfect cipher property (1.2) and the definition of the conditional probability (1.1).
It follows with $\mathcal{A} = (m = m^*)$ and $\mathcal{B} = (E_k(m) = c)$

$$P(\mathcal{B}) = P(E_k(m) = c) = \sum_{m_i \in \mathcal{M}} \sum_{k_i \in \mathcal{K}} \frac{P(E_{k_i}(m_i))}{|\mathcal{M}| \cdot |\mathcal{K}|}$$

For any message-ciphertext pair, there is only one key that maps that message to that ciphertext, therefore

$$\sum_{k_i \in \mathcal{K}} P(E_{k_i}(m) = c) = 1$$

and summing over all messages the value of 1 leads to

$$P(B) = P(E_k(m) = c) = \sum_{m_i \in \mathcal{M}} \sum_{k_i \in \mathcal{K}} \frac{P(E_{k_i}(m_i))}{|\mathcal{M}| \cdot |\mathcal{K}|} = \frac{|\mathcal{M}| \cdot 1}{|\mathcal{M}| \cdot |\mathcal{K}|} = \frac{1}{|\mathcal{K}|}$$

That's the probability of event $\mathcal{B}$, which is the probability that some message encrypts to some key (computed over all the messages).
Then

$$P(\mathcal{A}\cap\mathcal{B}) = P(m = m^*\cap E_k(m) = c) = P(m = m^*)\cdot P(k = k^*) = P(m = m^*)\cdot\frac{1}{|\mathcal{K}|} = \frac{P(m = m^*)}{|\mathcal{K}|}$$

to see this consider $k^* \in \mathcal{K}, m^* \in \mathcal{M}$ and the distribution of $\mathcal{M}$ is not uniform (not all messages are equally likely) and every key maps each message to only one ciphertext and the keys are equally likely (the distribution of the keys is uniform), therefore $P(k = k^*) = \frac{1}{|\mathcal{K}|}$.
Plugging all together in the conditional probability formula gives

$$P[m = m^*|E_k(m) = c] = \frac{P(m = m^* \cap E_k(m) = c)}{P(E_k(m) = c)} = \frac{\frac{P(m=m^*)}{|\mathcal{K}|}}{\frac{1}{|\mathcal{K}|}} = P(m = m^*)$$

Which is the definition of the perfect cipher. It follows that the *one-time pad* is a perfect cipher
□
**Definition** *malleable cipher, impractical cipher*
A cipher is

- *malleable* then the encryptet message $E_k(m) = c$ can be modified by an active attacker $X$, which means

$$\begin{tikzcd} m \arrow[r] & E_k(m) \arrow[r, squiggly, "c"] & X \arrow[r, squiggly, "c'"] & E_k(c') \arrow[r] & m' \\ & k \arrow[u] & & k \arrow[u] & \end{tikzcd}$$

- *impractical* if and only if

$$|\mathcal{K}| \geq |\mathcal{M}|$$

  The one-time pad is very impractical, because the keys have to be as long as the messages, and a key can never be reused. That means

$$|\mathcal{K}| = |\mathcal{M}|$$

  Unfortunately, Claude Shannon proved that finding a practical perfect cipher is impossible.

.

**Theorem 1.6.2 (Shannon's Keyspace Theorem)**
*Every perfect cipher is impractical.*

**Proof:** Proof by contradiction:
Assume having a perfect cipher that does not satisfy the impractical property.
That's equal to:
Suppose $E$ is a perfect cipher where $|\mathcal{M}| > |\mathcal{K}|$.
Let $c_0 \in \mathcal{C}$ with $P(E_k(m) = c_0) > 0$. That means there is some key that encrypts some message $m$ to $c_0$.
Decrypt $c_0$ with all $k \in \mathcal{K}$ with the decryption function $D$ (not necessarily the same as $E$).
Since the cipher is correct - in order to be perfect it has to both be correct and perfectly secure.
That means the decryption function must have the property

$$D_k(E_k(m)) = m$$

Let

$$\mathcal{M}_0 = \bigcup_{k \in \mathcal{K}} D_k(c_0)$$

Therefore $\mathcal{M}_0$ is the set of all possible messages decrypting $c_0$ with every key $k \in \mathcal{K}$ (brute-force attack). It follows

$(a)$ $|\mathcal{M}_0| \leq |\mathcal{K}|$
Because of construction of $\mathcal{M}_0$ (union over all keys)

$(b)$ $|\mathcal{M}_0| < |\mathcal{M}|$
Because of the assumption $|\mathcal{M}| > |\mathcal{K}|$ and combined with $(a) : |\mathcal{M}_0| \leq |\mathcal{K}|$.

$(c)$ $\exists m^* \in \mathcal{M} : m^* \notin \mathcal{M}_0$
Follows exactly from $(b) : |\mathcal{M}_0| < |\mathcal{M}|$

Considering the perfect cipher property

$$P[m = m^* | E_k(m) = c] = P[m = m^*]$$

Due to $(b) : P(m = m^*) = 0$ but due to $(c) : m^* \in \mathcal{M} \Rightarrow P(m = m^*) \neq 0$
We have contradicted the requirement for the pefect cipher. Therefore the assumption $|\mathcal{M}| > |\mathcal{K}|$. Thus:
There exists **no** perfect ciphers where

$$|\mathcal{M}| > |\mathcal{K}|$$

Therefore every cipher that is perfect must be impractical. □

## 1.7 Monoalphabetic Substitution Cipher (Toy-Cipher)

**Definition** *Monoalphabetic Substitution Cipher (Toy-Cipher)*
The *Toy-Cipher* is a *monoalphabetic substitution cipher* where each letter in the alphabet is mapped to a substitution letter. The decryption is done by the reversed mapping. The cipher uses $\mathcal{M} = \{A, B, C, \ldots, Z\}^n$ (words of length $n$, using letters from the alphabet) and has $\mathcal{K} = \sigma_{26}$ (permutation of 26 letters) as keyspace.
**Example** 15:
One possible mapping is given by (every letter maps to the next letter and $z$ maps to $a$)

$$\begin{aligned} a &\mapsto b \\ b &\mapsto c \\ &\vdots \\ y &\mapsto z \\ z &\mapsto a \end{aligned}$$

Thus the encryption function $E$ encrypts the message $m$ =hello as follows:

$$\begin{aligned} h &\mapsto i \\ e &\mapsto f \\ l &\mapsto m \\ l &\mapsto m \\ o &\mapsto p \end{aligned}$$

It follows $E_1(m) = ifmmp$ where the key $k = 1$ is the *translation* or *shift* of each letter.

**Theorem 1.7.1:**
*The Monoalphabetic Substitution Cipher (Toy-Cipher) is imperfect for a minimum message length of* 19

**Proof:**
*Shannon's keyspace theorem* claims that a cipher is perfect if and only if

$$|\mathcal{K}| \geq |\mathcal{M}|$$

(The keyspace is as least as big as the message space).
It follows a cipher is imperfect if and ony if

$$|\mathcal{K}| < |\mathcal{M}|$$

Applying this inequality delivers

$$|\mathcal{K}| = 26 \cdot 25 \cdot 24 \cdot \dots \cdot 2 \cdot 1 = 26!$$

because the key is just a permutation of the alphabet. There are 26 choices for what $a$ can map to, 25 choices for what $b$ can map to, and so on.
The number of possible messages (the message space) of length $n$ is

$$|\mathcal{M}| = 26^n$$

Thus the smallest $n$ follows by

$$26! < 26^n \Rightarrow n \geq 19$$

□

**Proof (by counterexample)**
Any two-letter ciphertext with same letters (e.g. $aa, bb, \cdots$) could not decrypt to a non two letter message with different letters (e.g. $ab, dk, lt, \cdots$). As a letter always decrypts to the same letter (i.e. $aa$ can only decrypt to a message with to identical letters) □

## 1.8 Lorenz Cipher Machine

**Theorem 1.8.1:**
*Given two cipertexts $m \oplus k = c = c_0c_1 \dots c_{n-1}, m' \oplus k = c' = c'_0c'_1 \dots, c'_{n-1}$ with $c, c' \in \mathcal{C}$ and $\exists j \in I := \{0, 1, 2, \dots, n-1\}$:*

$$c_j \neq c'_j$$

*then by xor'ing $c \oplus c' = m \oplus k \oplus m' \oplus k = m \oplus m'$. If there is only a slightly difference between c and $c'$ it is possible by guessing $m^* \sim m$ and getting a possible message via (xor'ing with the intercepted cipertexts)*

$$m^* \oplus c \oplus c'$$

*which should give back the other message $m'$.*
*Once the two messages $m, m'$ are given, it's easy to get the key with*

$$k = m \oplus c$$

**Definition** *Lorenz Cipher Machine*

Each letter of the message $m$ would be divided into 5 bits $m_0m_1m_2m_3m_4$, and those would be *xor'd* with the value coming from the corresponding and different sized $k$-wheels which had at each position a 0 or a 1. The result would also be *xor'd* with the value of the $s$-wheels, which worked similarly. The $k$-wheels turned every charakter, the $s$-whells turned conditionally on the result of 2 other wheels, which were the $m_1$-wheel (which turned every time) and the $m_2$-wheel (which would rotate depending on the value of the $m_1$-wheel) and depending on the $m_1 \oplus m_2$ either all the $s$-wheels would rotate by 1 or none of them would. The result of all these *xors* is the cipher text $c = c_0c_1c_2c_3c_4$.
The Lorenz Cipher works similarly to an One - Time Pad: *xor'ing* a message with a key leads to a ciphertext.
Knowing the structure of the machine is not enough to break the cipher. It's necessary to know the initial configuration.

**Example** 16:
Let $z = z_0z_1z_2z_3z_4z_5z_6z_7 \dots z_{n-1}$ be the interceptet message. The ciphertext $z$ is broken into 5 channels $c$ where each bit on position $i$ with $i \in \{0, 1, 2, \dots, n-1\}$ is transmitted over channel $(i + 1) \bmod 5$. Thus for 5 channels $c_1, c_2, c_3, c_4, c_5$:

$$\begin{aligned} c_1 &\longrightarrow z_0z_5z_{10}z_{15}\cdots \\ c_2 &\longrightarrow z_1z_6z_{11}z_{16}\cdots \\ c_3 &\longrightarrow z_2z_7z_{12}z_{17}\cdots \\ c_4 &\longrightarrow z_3z_8z_{13}z_{18}\cdots \\ c_5 &\longrightarrow z_4z_9z_{14}z_{19}\cdots \end{aligned}$$

So channel 1 transmit the first part of the first letter $z_0$, the first part of the second letter $z_5$, channel 2 transmit the second part of the first letter $z_1$, the second part of the second letter $z_6$, and so on.
Now subscripting $z$ by th channel and the letter for that channel $z_{c,i}$. It follows

$$\begin{aligned} z_{0,i} &= z_0, z_5, z_{10}, \cdots \\ z_{1,i} &= z_1, z_6, z_{11}, \cdots \end{aligned}$$

The subscripts break up the ciphertext into channels and therefore, with the weakness of the cipher (all $s$-wheels move in turn). Thus

$$z_{c,i} = m_{c,i} \oplus k_{c,i} \oplus s_{c,i}$$

and by separating the ciphertext into these 3 pieces, it's possible to take advantage of properties that they have. The importance is that the $s$-wheels don't always turn. Looking at subsequent characters, there is a good chance that the $s$-wheels have not changed. Let's define

$$\Delta z_{c,i} := z_{c,i} \oplus z_{c,i+1}$$

notice that $z_{c,i}, z_{c,i+1}$ are 5 characters apart in the interceptet ciphertext, but they are adjacent for that channel. It follows

$$\begin{aligned} \Delta z_{0,i} \oplus \Delta z_{1,i} &= z_{0,i} \oplus z_{0,i+1} \oplus z_{1,i} \oplus z_{1,i+1} \\ &= m_{0,i} \oplus k_{0,i} \oplus s_{0,i} \oplus m_{0,i+1} \oplus k_{0,i+1} \oplus s_{0,i+1} \oplus m_{1,i} \oplus k_{1,i} \oplus s_{1,i} \oplus m_{1,i+1} \oplus k_{1,i+1} \oplus s_{1,i+1} \\ &= \underbrace{m_{0,i} \oplus m_{0,i+1} \oplus m_{1,i} \oplus m_{1,i+1}}_{=:\Delta m} \oplus \underbrace{k_{0,i} \oplus k_{0,i+1} \oplus k_{1,i} \oplus k_{1,i+1}}_{=:\Delta k} \oplus \underbrace{s_{0,i} \oplus s_{0,i+1} \oplus s_{1,i} \oplus s_{1,i+1}}_{=:\Delta s} \\ &= \Delta m \oplus \Delta k \oplus \Delta s \end{aligned}$$

**Theorem 1.8.2:**
*With the example above follows*

(a) $P(\Delta m = 0) > \frac{1}{2}$

(b) $P(\Delta s = 0) > \frac{1}{2}$

**Proof:**

(a) $P(\Delta m = 0) > \frac{1}{2}$ depends on subsequent message letters:
If adjacent letters in the message are the same, that ensures that $\Delta m = 0$ (repeated letter: 'wh<u>ee</u>ls', 'le<u>tt</u>ers', for German 0.61)

(b) $P(\Delta s = 0) > \frac{1}{2}$ follows by the structure of the machine:
When the $s$-wheel advance this probability is about $\frac{1}{2}$ but when they don't advance, $\Delta s$ is always 0. This means, the probability that $\Delta s = 0$ is significanlty greater than $\frac{1}{2}$ (for the structure of the Lorenz Cipher Maschine it's about 0.73)

□

**Example** 17:
Assume $P(\Delta k = 0) = \frac{1}{2}$ and $P(\Delta m = 0) > \frac{1}{2}$ and $P(\Delta s = 0) > \frac{1}{2}$ with $\Delta z_{c,i} = \Delta m_{c,i} \oplus \Delta k_{c,i} \oplus \Delta s_{c,i}$. It's possible to break the cipher knowing more about the key $k$. If key is uniformly distributed, whatever patterns $m$ and $s$ have are lost when they get *xor'ed* with $k$ in $\Delta z_{c,i} = \Delta m_{c,i} \oplus \Delta k_{c,i} \oplus \Delta s_{c,i}$. The $k$-wheels in the Lorenz Cipher machine produce key. Looking at $\Delta z$ for two channels, i.e. only at the first two $k$-wheels (size 41 and size 31). Then there are $41 \cdot 31 = 1271$ different configurations for $k_0$ and $k_1$. That means that every 1271 letters those wheels woult repeat, and there are only 1271 different possible settings for the $k$-wheels. Trying all 1271 possible setting and for 1 of those possible settings we are goint to know all the key bits and if we guess the right setting then $\Delta k = 0$. If we guess right then $P(\Delta k = 0) = 1$ otherwise (false guess) $P(\Delta k = 0) = \frac{1}{2}$. With $\Delta z = \Delta m \oplus \Delta k \oplus \Delta s$ it follws $P(\Delta z = 0) = 0.55$ because:

$$\begin{aligned} \Delta z = \Delta m \oplus \underbrace{\Delta k}_{=0} \oplus \Delta s \Rightarrow P(\Delta z = 0) &= P(\Delta m = 0) \wedge P(\Delta s = 0) + P(\Delta m = 1) \wedge P(\Delta s = 1) \\ &= 0.61 \cdot 0.73 + (1 - 0.63) \cdot (1 - 0.73) \\ &= 0.55 \end{aligned}$$

Computing the sum of all $\Delta z$. If the output is nearly $\frac{|z|}{2}$ is was a bad guess otherwise if the result is about $0.55 \cdot |z|$ is was a good guess.
Assume having a 5000 letters message with all 1271 configurations of $k_0$ and $k_1$ and for all configuration its necessary to compute the summation of the $\Delta z$. Guessing that the $\Delta s = 0$ therefore

$$\Delta z_{0,i} \oplus \Delta z_{1,i} = m_{0,i} \oplus m_{0,i+1} \oplus m_{1,i} \oplus m_{1,i+1} \oplus k_{0,i} \oplus k_{0,i+1} \oplus k_{1,i} \oplus k_{1,i+1} \oplus \underbrace{s_{0,i} \oplus s_{0,i+1} \oplus s_{1,i} \oplus s_{1,i+1}}_{=:\Delta s = 0}$$

Thus computin 7 *xors* for each character and counting the number of times that's equal to 0. It follows the number of *xors* are $5000 \cdot 1272 \cdot 7 = 44485000$ whats the maximum number of *xors* we have to do (expect about half of 44485000 operations to find the correct configuration of $k_0$ and $k_1$ and then do similar thinks with the other $k$-wheels and then we can decrypt the whole message). With a 2 GHz processor we need a fraction of a millisecond

**Theorem 1.8.3 (Goal of Cipher)**
*The goal of a cipher is to hide statistical properties of the message space and key (which should be perfectly random).*
*Two properties of the operation of a secure cipher are:*

- *Confusion:*
  *making the relationship between the plaintext and the ciphertext as complex and involved as possible*

- *Diffusion:*
  *non-uniformity in the distribution of the individual letters (and pairs of neighbouring letters) in the plaintext should be redistributed into the non-uniformity in the distribution of much larger structures of the ciphertext, which is much harder to detect (the output bits should depend on the input bits in a very complex way - see also avalanche effect).*

**Theorem 1.8.4 (Goal of Cryptoanalyst)**
*The goal of a cryptoanalyst is to find statistical properties in ciphertext and use those to break the key and/or message (Lorenz Cipher Machine has statistical properties when you looked acrosss channels at subsequent letters which was not hidden by the cipher and because of a mechanical weakness that all the s-wheel either all moved or didn't move and matehmatical weakness - only 1272 different positions of the first two k-wheels.)*

## 1.9 Modern Symmetric Ciphers

**Definition** *modern symmetric ciphers, stream ciphers, block ciphers*
There are two main types of *modern symmetric ciphers*:

- stream cipher:
  consists of a stream of data and the cipher can encrypt small chunks at a time (usually 1 byte at a time)

- block cipher:
  the data is separated in larger chunks and the cipher encrypts a block at a time (usually a block size is at least 64 bits and can be up to 128 or 256 bits)

The only differences is changing the block size. The different ciphers are designed for different purposes.

**Definition** *Advanced Encryption Standard (AES), Data Encryption Standard (DES)*
*Advanced Encryption Standard* or *AES* is the most important block cipher nowadays (since 1997) and works on blocks on 128 bits and displaced the *Data Encryption Standard* or *DES*, which had been a standard for the previous decades. *AES* was the winner of a competition that was run by the United States. The main criteria for the submissed ciphers in the competition where

- **security** (as provable security is only achievable for the one-time pad) computed with

  $$\text{security} \sim \frac{\text{actual \# round}}{\text{minimal \# of rounds}}$$

  where breakable means anything that showed you could reduce the search space even a little bit woult be enough

- **speed**: implementing it both in hardware and in software and
- **simplicity** which is usually against security.

The winner of the *AES* competition was a cipher known as *Rijndael* (developed by two belgian cryptographers). A brute force attack with a 128 bit key would require on average $\frac{2^{128}}{2} = 2^{127}$ attempts. The best known attack needs $2^{126}$ attempts.
The *AES* works with *xor* and has two main operations

- **shift** (permuting bits - moving bits around)
- **s-boxes** (non-linearity: mixes up data in way that is not linear):
  This is done by lookup-tables:
  A *s-box* takes 8 bits and have a lookup table (with 256 entries) mapping each set of 8 bits to some other set of 8 bits. Designing the lookup table is a challenge. The lookup table has to be as nonlinear as possible and make sure there is no patterns in the data in this table.

The way *AES* works is combining *shifts* and *s-boxes* with *xor* to scramble up the data and do this multiple rounds and put them back through a series of *shifts* and *s-boxes* with *xor*. The number of rounds depens on the key size: for the smallest key size for *AES* (128 bits) we would do 10 rounds going through the cycle, getting the output cipher text for that block.

# 2 Application of Symmetric Ciphers

## 2.1 Application of Symmetric Ciphers

Ciphers provide 2 main functions:

- Encryption:
  Takes a message $m$ from some message space $\mathcal{M}$ and a key $k$ from some key space $\mathcal{K}$.
- Decryption:
  Is the inverse of encryption. It takes a ciphertext and if it takes the corresponding key $k'$ it will produce the same message that we got.

The correctness property (as mentioned earlier):

$$D_k(E_k(m)) = m$$

All of our assumptions about security depend on the key.
There are 2 main key properties:

- $k$ is selected *randomly* and *uniformly* from $\mathcal{K}$. This means each key is equally likely to be selected and there is no predictability about what the key is.
- $k$ can be *kept secret* (but shared). That means that the adversary can't learn the key but it can be shared between the 2 endpoints.

## 2.2 Generating Random Keys

**Definition** *(Kolomogorov) Randomness*
A string of bits is *random* if and only if it is shorter than any computer program that can produce that string (*Kolmogorov Randomness*).
This means that random strings are those that cannot be compressed.
**Example** 18:
$k \in \mathcal{K}$ for $\mathcal{K} := \{0,1\}^n$ with some $n \in \mathbb{N}$. If there are no visible patterns (e.g. 100100100...) and enough repititions (e.g. 100000101111101011111...), then it is likely that the key is random.
**Definition** *Complexity of a Sequence (Kolomogorov Complexity)*
How random a certain string is, is a way of measuring the *complexity* $K$ of some sequence $s$. This is defined as the length of the shortest possible description of that sequence:

$$K(s) = \text{length of the shortest possible describtion of } s$$

where a description is e.g. a Turing-Maschine, a python program or wathever we select as description language and as long as that description language is powerful enough to describe any algorithm, which it's a reasonable way to define complexity.
**Definition** *Random Sequence*
A sequence $s$ is *random* if and only if

$$K(s) = |s| + C$$

That means, making the sequence longer the description gets longer at the same rate with the constant $C$.
Therefore a short program that can produce the sequence is not random as there is a structure in the program and the program shows what is that structure.
If there isn't a short program that can describe that sequence, that's an indication that the sequence is random (there is no simpler way to understand that sequence other than to see the whole sequence).
**Theorem 2.2.1:**
*For a given sequence $s$ it is theoretically impossible to compute $K(s)$.*

**Proof:**
If $s$ is truly random then

$$K(s) = |s| + C$$

would be correct.
But if $s$ is not truly random, there might be some shorter way to compute $K(s)$. So $K(s)$ gives the maximum value of the Kolomogorov complexity of a sequence $s$ e.g. `''print '' + s` which prints out $s$. It's length would be the length of $s$ plus the 5 characters (4 for `print` plus 1 for the space). But that doesn't proof that there is a shorter program that can produce $s$. □
**Example** 19:
The Berry Paradoxon gives an idea of the proof of the former theorem:

`'What is the smallest natural number that cannot be described in eleven words?'`

Which has 2 properties:

- Set of natural numbers that cannot be described in eleven words (a set).
- Any set of natural numbers has a smallest element.

The answer:

'The smallest natural number that cannot be described in eleven words'

has $\underline{11}$ words. That suggest there is no such number but this contradicts the 2 properties (paradox)

**Definition** *Statistical Test*
A *statistical test* shows that something is non-random. The *statistical test* **can't** prove that something is random.

**Definition** *Unpredictability*
*Unpredictability* is the requirement to satisfy the randomness for generating a good key.

**Example** 20:
Assuming a sequence $s$ of lenght $n$

$$s = x_0, x_1, x_2, \dots, x_{n-1}$$

with $x_i \in [0, 2^{n-1}]$.
Even after seeing $x_0, x_1, x_2, \dots, x_{m-1}$, it's only possible to guess $x_m$ with probability $\frac{1}{2^n}$.

**Definition** *Physically Random Events*
*Physically random events* are in

- Quantum Mechanics (e.g.: events in the universe, radiactive declay, and others)
- Thermal noise
- Key presses or user actions
- many others

.

## 2.3 Pseudo Random Number Generator (PRNG)

**Definition** *Pseudo Random Number Generator, seed, state*
A *pseudo random number generator* takes as input a small amount of physically randomness (*seed*) and produces a long sequence of 'random' bits. The *PRNG* is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the *PRNG's state*, which includes a truly random *seed.*

**Example** 21:
Assume extracting a seed $s$ from a *Random Pool* (finite many true random numbers) and using $s$ as key. The *PRNG* may look as follows

$$\begin{tikzcd}
 & 0 \arrow[d] & & 1 \arrow[d] & \\
s \arrow[r] & E \arrow[d] & s \arrow[r] & E \arrow[d] & \cdots \\
 & x_0 & & x_1 &
\end{tikzcd}$$

For the first random output $x_0$, we get 0 encrypting that with $s$ and so on.

**Theorem 2.3.1:**
*It's impossible to wirte a program, that test a sequence of bits if they are truly random and a sequence that passes this test can't be truly random.*

## 2.4 Modes of Operation

The *modes of operation* is the procedure of enabling the repeated and secure use of a block cipher (AES) under a single key. That means *modes of operation* are ways to encrypt a file that doesn't give that much information about the message $m$ from the ciphertext $c$.

### 2.4.1 Electronic Codebook Mode (ECB)

**Definition** *Electronic codebook mode (ECB)*
The *electronic codebook mode* maps for each $i \in \{0, 1, 2, \ldots, 2^j - 1\}$ (in AES $j = 128$) inputs the value of $E_k(i)$. That is (for only one key):

$$\begin{array}{rcl} 0 & \mapsto & E_k(0) \\ 1 & \mapsto & E_k(1) \\ & \vdots & \\ 2^j - 1 & \mapsto & E_k(2^j - 1) \end{array}$$

Thus for $m = m_0 m_1 m_2 \ldots m_{n-1}$ it is $\forall i \in \{0, 1, 2, \ldots, n-1\}$

$$E_k(m_i) = c_i$$

and therefore

$$c = c_0 c_1 c_2 \ldots c_{n-1}$$

The *electronic codeblock mode* works as follows:

1. The message $m$ is divided into blocks

   $$m = m_0 m_1 m_2 \ldots$$

   with a block length depeding on the cipher (assume each block $\forall i \in \{0, 1, 2, \ldots\} : m_i$ has a block length of 128 bits).

2. The ciphertext is

   $$c = c_0 c_1 c_2 \ldots$$

   where $\forall i \in \{0, 1, 2, \ldots\}$ :

   $$c_i = E_k(m_i)$$

.
Assue $E$ has perfect secrecy (impossible due to reusing the key, therefore $|\mathcal{K}| < |\mathcal{M}|$). Then the attacker (knowing only $c$) can only figure out:

- The length of $m$ because the lenght of $c$ is equal to the length of $m$.
- Which blocks in $m$ are equal. For an 128 bit encryption and an 8 bit character length there are only $\frac{128}{8} = 16$ characters per block. That means after 16 characters a new block starts.

The 2 main problems of *electronic codebook mode* are

- The *electronic codebook mode* doesn't hide repititions
- An attacker can move or replace blocks and decryption would result in a perfectly valid message with the blocks in a different order.

### 2.4.2 Cipher Block Chaining Mode (CBC)

**Definition** *Cipher block chaining mode*
The idea of *CBC* mode is using the ciphertext from the previous block to impact the next block. Breaking the message $m$ into blocks $m = m_0 m_1 m_2 \dots m_{n-1}$ of block size $b$, then the *CBC* may look as follows

$$\begin{tikzcd}
 & m_0 \arrow[d] & & m_1 \arrow[d] & & m_2 \arrow[d] & \cdots \\
\mathrm{IV} \arrow[r] & \oplus \arrow[d] & & \oplus \arrow[d] & & \oplus \arrow[d] & \cdots \\
k \arrow[r] & E \arrow[d] & k \arrow[r] & E \arrow[d] & k \arrow[r] & E \arrow[d] & \cdots \\
 & c_0 \arrow[uurr, bend left] & & c_1 \arrow[uurr, bend left] & & c_2 & \cdots
\end{tikzcd}$$

This means instead of doing each block independently, each message block will be *xor*ed with the previous cipher block and then encrypted:

1. The first message block $m_0$ will be *xor*ed with a *initialization vector* (IV), which is a random block of size $b$, and then encrypted with $E$ using $k$ to get $c_0$. The IV don't need to be kept secret but it's helpful to not to reuse an IV.

2. $m_1$ will be *xor*ed with $c_0$ and then encrypted with $E$ using $k$ to get $c_1$.

3. Repeating this until $m_{n-1}$ will be *xor*ed with $c_{n-2}$ and then encrypted with $E$ using $k$ to get $c_{n-1}$.

The result of *CBC* is $\forall i \in \{1, 2, 3, \dots, n-1\}$:

$$\begin{aligned} c_0 &= E_k(m_0 \oplus \mathrm{IV}) \\ c_i &= E_k(m_i \oplus c_{i-1}) \end{aligned}$$

Note that the *CBC* still encrypts the output of $m_0 \oplus \mathrm{IV}$.
**Theorem 2.4.1 (Recovering $m$)**
*Loosing the value of IV but having $c$ and $k$, then the message $m$ (excepts $m_0$) can be recovered with the formula*

$$\begin{aligned} c_i = E_k(m_i \oplus c_{i-1}) \;&\Rightarrow\; m_{i-1} = D_k(c_{i-1}) \oplus c_{i-2} \\ &\Rightarrow\; m_i = D_k(c_i) \oplus c_{i-1} \end{aligned}$$

*except*

$$c_0 = E_k(m_0 \oplus IV) \Rightarrow m_0 = D_k(c_0) \oplus IV$$

*thus $m_0$ is lost.*

**Example** 22:
Implementing *ciper block chaining mode* in `Python` may look as follows

```
from Crypto.Cipher import AES

def non_encoder(block, key):
    """A basic encoder that doesn’t actually do anything"""
    return pad_bits_append(block, len(key))

def xor_encoder(block, key):
    block = pad_bits_append(block, len(key))
    cipher = [b ^ k for b, k in zip(block, key)]
    return cipher

def aes_encoder(block, key):
    block = pad_bits_append(block, len(key))
    # the pycrypto library expects the key and block in 8 bit ascii
    # encoded strings so we have to convert from the bit string
    block = bits_to_string(block)
    key = bits_to_string(key)
    ecb = AES.new(key, AES.MODE_ECB)
    return string_to_bits(ecb.encrypt(block))


# this is an example implementation of
# the electronic cookbook cipher
# illustrating manipulating the plaintext,
# key, and init_vec
def electronic_cookbook(plaintext, key, block_size, block_enc):
    """Return the ecb encoding of ‘plaintext"""
    cipher = []
    # break the plaintext into blocks
    # and encode each one
    for i in range(len(plaintext) / block_size + 1):
        start = i * block_size
        if start >= len(plaintext):
            break
        end = min(len(plaintext), (i+1) * block_size)
        block = plaintext[start:end]
        cipher.extend(block_enc(block, key))
    return cipher

    ###############
def xor(x,y):
    return [xx^yy for xx,yy in zip(x,y)]
def cipher_block_chaining(plaintext,key,init_vec,block_size,block_enc):
#plaintext = bits to be encoded
#key = bits used as key for the block encoder
#init_vec = bits used as initialization vector for the block encoder
#block_size = size of blocks used by block_enc
#block_enc = function that encodes a block using key
    cipher = []
    xor_input=init_vec
    # break the plaintext into blocks
    # and encode each one
    for i in range(len(plaintext) / block_size + 1):
        start = i * block_size
        if start >= len(plaintext):
            break
        end = min(len(plaintext), (i+1) * block_size)
        block = plaintext[start:end]
        input_=xor(xor_input,block)
        output=block_enc(input_,key)
        xor_input=output
        cipher.extend(block_enc(block, key))
    return cipher
    ####################
```

```
def test():
    key = string_to_bits(’4h8f.093mJo:*9#$’)
    iv = string_to_bits(’89JIlkj3$%0lkjdg’)
    plaintext = string_to_bits("One if by land; two if by sea")

    cipher = cipher_block_chaining(plaintext, key, iv, 128, aes_encoder)
    assert bits_to_string(cipher) == ’\xeaJ\x13t\x00\x1f\xcb\xf8\xd2\x032b\xd0\xb6T\xb2\xb1\x81\xd5h
    \x97\xa0\xaeogtNi\xfa\x08\xca\x1e’

    cipher = cipher_block_chaining(plaintext, key, iv, 128, non_encoder)
    assert bits_to_string(cipher) == ’wW/i\x05\rJQ]\x05\\\r\x05\x0e_G\x03 @Ilkj3$%/hd\x00\x00\x00’

    cipher = cipher_block_chaining(plaintext, key, iv, 128, xor_encoder)
    assert bits_to_string(cipher) == ’C?\x17\x0f+=sb0037/7|c\x03 @Ilkj3$%/hd9#$’

###################
# Here are some utility functions
# that you might find useful

BITS = (’0’, ’1’)
ASCII_BITS = 8

def display_bits(b):
    """converts list of {0, 1}* to string"""
    return ’’.join([BITS[e] for e in b])

def seq_to_bits(seq):
    return [0 if b == ’0’ else 1 for b in seq]

def pad_bits(bits, pad):
    """pads seq with leading 0s up to length pad"""
    assert len(bits) <= pad
    return [0] * (pad - len(bits)) + bits

def convert_to_bits(n):
    """converts an integer ‘n‘ to bit array"""
    result = []
    if n == 0:
        return [0]
    while n > 0:
        result = [(n % 2)] + result
        n = n / 2
    return result

def string_to_bits(s):
    def chr_to_bit(c):
        return pad_bits(convert_to_bits(ord(c)), ASCII_BITS)
    return [b for group in
            map(chr_to_bit, s)
            for b in group]

def bits_to_char(b):
    assert len(b) == ASCII_BITS
    value = 0
    for e in b:
        value = (value * 2) + e
    return chr(value)

def list_to_string(p):
    return ''.join(p)

def bits_to_string(b):
    return ''.join([bits_to_char(b[i:i + ASCII_BITS])
                    for i in range(0, len(b), ASCII_BITS)])

def pad_bits_append(small, size):
    # as mentioned in lecture, simply padding with
    # zeros is not a robust way way of padding
    # as there is no way of knowing the actual length
    # of the file, but this is good enough
    # for the purpose of this exercise
    diff = max(0, size - len(small))
    return small + [0] * diff
```

.

### 2.4.3 Counter Mode (CTR)

**Definition** *Counter Mode (CTR)*
A message $m$ is divided into blocks $m = m_0 m_1 \dots m_{n-1}$. In the $CTR$ instead of just having a message block go in the encryption function there is a counter (some value that cycles through the natural numbers) which is the input to the encryption function and so the results are some encrypted blocks. These blocks *xor'd* with the corresponding messageblock are the final ciphertext blocks. To avoid the problem of using the same sequence of counters every time, we add a nonce (in fact: appending the nonce with the counter value). A nonce is simply a one-time, unpredictable value (similar to a key) which isn't need to be kept secret (e.g. with AES: the size of a block is always 128 bits, therefore the nonce and the counter are each 64 bits long). The $CTR$ mode may look as follows:

$$\begin{tikzcd}
 & \text{nonce}|0 \arrow[d] & & \text{nonce}|1 \arrow[d] & & & \text{nonce}|n-1 \arrow[d] \\
k \arrow[r] & E \arrow[d] & k \arrow[r] & E \arrow[d] & \cdots & k \arrow[r] & E \arrow[d] \\
 & \bigoplus m_0 \arrow[d] & & \bigoplus m_1 \arrow[d] & & & \bigoplus m_{n-1} \arrow[d] \\
 & c_0 & & c_1 & & & c_{n-1}
\end{tikzcd}$$

It follows (encryption)

$$c_i = E_k(\text{nonce}|i) \oplus m_i$$

and (decryption):

$$m_i = c_i \oplus E_k(\text{nonce}|i)$$

### 2.4.4 CBC versus CTR

Due to former definitions:

|  | CBC | CTR |
|---|---|---|
| Encryption | $c_i = E_k(m_i \oplus c_{i-1})$<br>$c_0 = E_k(m_0 \oplus \mathrm{IV})$ | $c_i = E_k(\mathrm{nonce}\vert i) \oplus m_i$ |
| Decryption | $m_i = D_k(c_i) \oplus c_{i-1}$<br>$m_0 = D_k(c_0) \oplus \mathrm{IV}$ | $m_i = c_i \oplus E_k(\mathrm{nonce}\vert i)$ |
| Speed | slower<br>encryption of $c_i$ requires<br>encryption of $c_{i-1}$<br>(no parallel encryption) | faster<br>can do encryption $E_k(\mathrm{nonce}\vert i)$ without<br>knowing message. Encryption is more<br>expensive than *xor* operation |

### 2.4.5 Cipher Feedback Mode (CFB)

**Definition** *Cipher Feedback Mode (CFB)*
The *CFB* mode takes as input some $n$-bit long $x$ values $x_0, x_1, x_2, \ldots$ with the property that the first value is an initialization vector $x_0 = \mathrm{IV}$. Each $x$ value is separated into 2 blocks: the first block $\widetilde{x}_i$ has a size of $s$ ($s$ block) and the second block has a size of $n-s$ ($n-s$ block). The encryption function takes the $n$ block of a $x$ value and a key $k$ of length $n$ and gives as output a $n$ bit long result. The result is separated into two blocks: the first block $c'_i$ has a length of $s$ ($s$ bock) and so the second block $c''_i$ has a length of $n-s$ ($n-s$ block). The message $m$ is divided into blocks of length $s$: $m = m_0 m_1 m_2 \ldots$. Each message block *xor'd* with the corresponding $s$-block ($c'_i$) to get the ciphertext block of length $s$. The next $x$ value is composited: the first part is the $n-s$ block of the former $x$ value and the second part is the ciphertext from the former encryption function (after *xor'd* with the corresponding $m$ block). This may look as follows:

$$\begin{tikzcd}
x_0 : & \mathrm{IV} \arrow[d] \arrow[rr, bend left] & x_1 : & \widetilde{x}_0||c_0 \arrow[d] \arrow[rr, bend left] & x_2 : & \widetilde{x}_1||c_1 \\
k \arrow[r] & E \arrow[d] & k \arrow[r] & E \arrow[d] & & \cdots \\
& c'_0||c''_0 \arrow[d] & & c'_1||c''_1 \arrow[d] & & \\
& m_0 \oplus c'_0 \arrow[d] & & m_1 \oplus c'_1 \arrow[d] & & \\
& c_0 \arrow[uuuurr, bend right] & & c_1 \arrow[uuuurr, bend right] & &
\end{tikzcd}$$

where the output of $E$ has size $n$ and is separated into $c'_0$ ($s$ block) and $c''_0$ ($n-s$ block). Then $c'_0$ is used to compute $c_0$ of size $s$ by *xoring* $c'_0$ with $m_0$. To get the next $x$ value simply append the $n-s$ block from the former $x$ (noted as $\widetilde{x}$) with the ciphertext of the former computation to get an $n$ bit long input for $E$. And so on. Updating the $x$ value works as follows:

$$\begin{aligned} x_i &= \widetilde{x}_{i-1}||c_{i-1} \\ x_0 &= \mathrm{IV} \end{aligned}$$

and the ciphertext values are given by:

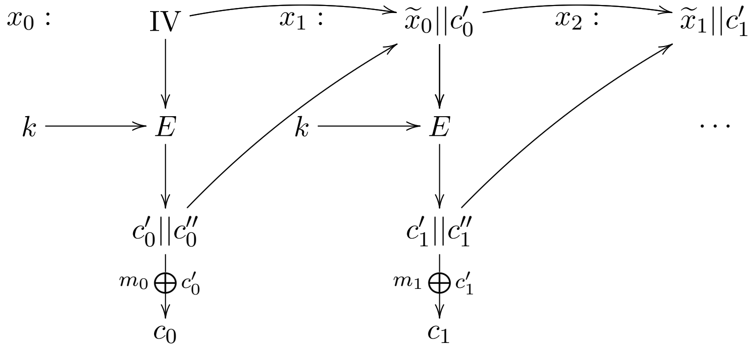
$$c_i = \underbrace{E_k(x_i)}_{=c'_i} \oplus m_i$$

The decryption of a message given the ciphertext $c = c_0 c_1 \dots c_{n-1}$:

$$\begin{aligned} m_i &= c_i \oplus \underbrace{E_k(x_i)}_{=c'_i} \\ x_i &= \widetilde{x}_{i-1} || c_{i-1} \\ x_0 &= \text{IV} \end{aligned}$$

### 2.4.6 Output Feedback Mode (OFB)

**Definition** *Output Feedback Mode*
The *OFB* mode is similar to *CFB* mode but instead of taking the ciphertext and putting that block into the $x$ value. We take the output from the encryption $E$ and take that into the next $x$ value. That's the only difference between *OFB* and *CFB*. This may look as follows



Unlike *CFB* with *OFB* it is possible to recover most of an encrypted file if one cipher block is lost. Therefore *OFB* could **not** be the basis of a cryptographic hash function, because in cryptographic hash functions the cipherblock text does depend on the previous message block (not given in *OFB*: $c_2$ doesn't depend on $m_1$).

### 2.4.7 CBC versus CFB

| | CBC | CFB |
|---|---|---|
| Requires $E$ is invertible | true | false |
| Reqires IV to be secret | false | false |
| Can use small message blocks | false | true |
| Protect against tampering | false | false |
| Final $c_{n-1}$ depens on all message blocks | true | true |

### 2.4.8 Parallel Decrypting Modes

**Theorem 2.4.2:**
*Mode of operation that can perform most of the decryption work in parallel:*

- *ECB*
- *CTR*
- *CBC*

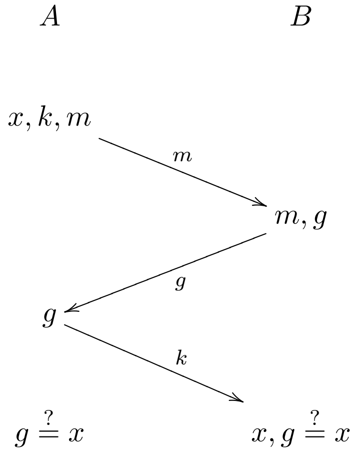
- *CFB*

## 2.5 Protocol

**Definition** *Protocol*
A *protocol* involves 2 or more parties and is a precisely definition of a sequence of steps. Each step can involve som computation and communication (sending data between the parties). A cryptographic protocol also involves a secret.
**Definition** *Security Protocol*
A *security protocol* is a protocol, that provides some guarantee even if some of the participants cheat (don't follow the protocol's steps as specified).
**Example** 23:
Making a coin toss over a channel via 2 parties $A$ and $B$:

1. $A$ picks a value for $x \in \{0, 1\}$ with 0 representing 'Heads' and 1 representing 'Tails' and a random key $k$ of length $n$ (security parameter): $k \in \{0, 1\}^n$.
2. $A$ will create a message $m$ by encrypting $x$ with $k$: $m = E_k(x)$.
3. $A$ sends the message $m$ to $B$.
4. $B$ receives $m$ and makes a guess $g \in \{0, 1\}$.
5. $A$ receives $g$ from $B$
6. $A$ sends $k$ to $B$ so that $B$ gets the result of the coin toss by decrypting $m$ with $k$: $x = D_k(m)$. If $x = g$ $B$ knows who won the coin toss.

This looks as follows:



Note that $A$ can cheat by finding 2 keys $k_0, k_1$ where

$$E_{k_0}(0) = E_{k_1}(1) \text{ or } E_{k_0}(1) = E_{k_1}(0)$$

and $A$ will win depends on the guess of $B$: $A$ sends a different key to $B$ for every choice $B$ makes. A harder way to cheat is finding 2 keys $k_0, k_1$ where:

$$E_{k_0}(0) = E_{k_1}(1) \text{ and } E_{k_0}(1) = E_{k_1}(0)$$

which will always lead to the opposite of $B$'s guess. And another harder way to cheat is finding $k'$ such that

$$E_{k'}(0) = E_k(1)$$

## 2.6 Padding

**Definition** *Padding*
If using a block cipher that requires an input of $n$ bits long (minimum $n = 128$ for the message, the key and therefore the output in $AES$) the message has to be *padded* sometimes to reach the required length. The simplest form is *zero padding* (filling up with zeroes until the required length is reached).
**Example** 24:
For the protocol we used in 23 the value of $x$ was only 1 bit (0 or 1). Using the $ECB$ mode requires 128 bits (in $AES$). The simplest solution is *zero padding*: padding the input with 127 0-bits added after $x$.

## 2.7 Cryptographic Hash Function

**Definition** *Cryptographic Hash Function*
The *Cryptographic Hash Function* $H$ is a function that takes some large value as input and outputs a small value:

$$h = H(x)$$

Regular Hash functions have these properties:

- Compression:
  $H$ takes large input and gives a small fixed output
- Distribution:
  $H$ is well distributed: $P(H(x) = i) \sim \frac{1}{N}$, where $N$ is the size of the output (output range: $[0, N)$).

A *cryptographic hash function* has additional these properties:

- Pre-image resistance ("one-way-ness"):
  Given $h$ then it's hard to solve for $h = H(x)$ for $x$.
- Weak collision resistance:
  Given $h = H(x)$, it's hard to find any $x'$ such that $H(x') = h$.
- Strong collision resistance:
  It's hard to find any pair $(x, y)$, such that $H(x) = H(y)$

.
**Example** 25:
An almost good cryptographic hash function is to use $CBC$ to encrypt $x$ and take the last output block as the value of the hash function because this provides the compression property as well as the collision resistance properties. This construction is similar to *Merkle-Dangard Construction.* For the hash function using the same key will work (select key being 0)

## 2.8 Random Oracle Assumption

**Definition** *Random Oracle Assumption*
A *random oracle assumption* is an ideal (has all required properties) cryptographic hash function. That maps any input to $h$ with an uniform distribution. An attacker trying to find collusion can do no better than a brute force search on the size of $h$:

$$H(x) \mapsto h$$

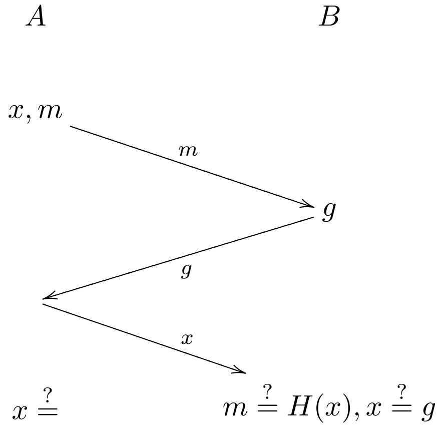
**Theorem 2.8.1:**
*It is impossible to construct a random oracle.*

**Proof:**
The hash function must be deterministic, so it produces the same output for the same input. We want to produce uniform distribution, so that means it needs to add randomness to what comes in, but that's impossible. Since it's deterministic, the only randomness it could use it what's provided by $x$. So there's no way to amplify that to provide more randomness in the output. □

**Example** 26:
Consider a coin toss as in Example 23 with a hash function:

1. $A$ picks a number $x \in \{0,1\}$ and computes the ideal cryptographic hash function (despit the ideal cryptographic hash function doesn't exist) $m = H(x)$
2. $A$ sends $m$ to $B$
3. $B$ makes a guess $g \in \{0,1\}$ and sends it to $A$.
4. $A$ sends $x$ to $B$
5. $B$ can check if $m = H(x)$. If $m \neq H(x)$ then $B$ suspects $A$ has cheated.

If $x = g$ then $B$ wins, otherwise $A$ wins. This may look as follows



In this protocol $B$ can easily cheat:
The hash function is not ecryption. There are no secrets that go into it. It's only providing this commitment to the input. $B$ can compute $H(0)$ and $H(1)$ and check them weather they are equal $m$ and instead of guessing, $B$ can pick whichever one did.
**Example** 27:

Assuming an attacker has enough computing power to perform $2^{62}$ hashes, then 63 bits should the (ideal) hash function produce to provide *weak collision resistance.* Let's assume an attacker success probability: $P(\text{attacker can find } x' \text{ where } H(x') = h) \leq \frac{1}{2}$ in $2^{62}$ tries. Consider (as $b$ the number of output bits and $k$ as the number of guesses: $g = 2^{62}$):

$$P(\text{ one guess } H(x') = h) = 2^{-b}$$
$$P(\text{ bad guess}) = 1 - 2^{-b}$$

And over s series of guesses, the probability that they are all bad:

$$P(k \text{ guesses all bad }) = (1 - 2^{-b})^k$$

Computing gives the following output:

$$\begin{aligned}(1 - 2^{-62})^{2^{62}} &\sim 0.63\\ (1 - 2^{-63})^{2^{62}} &\sim 0.39\end{aligned}$$

That means 63 is the fewest number of bits to provide the attacker would less than 50% chances of finding a pre-image that maps to the same hash value in $2^{62}$ guesses.
Let's assume

$$\text{weak collision resistance} \sim 2^b$$
$$\text{strong collision resistance} \sim 2^{\frac{b}{2}}$$

This means, for weak collision resistance an attacker needs to do about $2^b$ work, where $b$ is the number of hash bits. To obtain strong collision resistance is actually much harder, and we need about twice as many bits for that. The attacker effort is more like $2^{\frac{b}{2}}$, so we need about twice as many output bits in our hash function to prove this.
**Example** 28:
The *birthday paradox* (not really a paradox) gives an unexpected result of the probability that 2 people have the same birthday.
Assume a group of $k$ people. The probability that 2 people have the same birthday is given by (no leap-years, birthdays are uniform distributed):
The complement probability (that there are no duplicates) is:

$$\begin{aligned}P(\text{no duplicates}) &= \frac{365 \cdot 364 \cdot 363 \cdot \dots \cdot (365 - k + 1)}{365 \cdot 365 \cdot 365 \cdot \dots \cdot 365}\\ &= \frac{\frac{n!}{(n-k)!}}{n^k}\end{aligned}$$

where $365 \cdot 364 \cdot 363 \cdot \dots \cdot (365 - k + 1)$ are the number of ways to assign birthdays with no duplication among $k$ people and $365 \cdot 365 \cdot 365 \cdot \dots \cdot 365$ is the number of ways to assign with duplication. $n$ is the number of possible days (hash outputs) and $k$ is the number of trials. The probability that there are duplicates is:

$$P(\text{one or more duplicates}) = 1 - \frac{\frac{n!}{(n-k)!}}{n^k}$$

It follows:

$$\begin{array}{lcl}\text{strong collusion resistance} & & \text{weak collusion resistance}\\ 1 - \frac{\frac{n!}{(n-k)!}}{n^k} & > & 1 - (1 - \frac{1}{n})^k\end{array}$$

**Example** 29:

For $n = 365$ days and $k$ people there is

| $k$ | $P(\text{at least one duplicate})$ |
|---|---|
| 2 | 0.0027 |
| 3 | 0.0082 |
| 6 | 0.0405 |
| 20 | 0.4114 |
| 21 | 0.4437 |
| 23 | 0.5073 |

That means: in a group of 23 people, it's more likely that 2 person have a birthday in common that no duplicate birthdays occur.
With $n = 2^{64}$ hash and an attacker can do $k = 2^{32}$ work(operations) the probability of a hash collision is about 0.39. For $k = 2^{34}$ the probability of a hash collision is about 0.99.
Conclusion:
Given an ideal hash function with $N$ possible outputs, an attacker is expected to need $\sim N$ guesses to find an input $x'$ that hashes to a particular value (weak):

$$H(x') = h$$

but only needs $\sqrt{N}$ guesses to find a pair $x, y$ that collide (strong):

$$H(x) = H(y)$$

This assumes an attacker can store all those hash values as the attacker try the guesses. This is teh reason why hash functions need to have a large output.
**Example** 30:
*SHA-1* uses 160 bits of output and was broken. There is a ways to find a collision with only $2^{51}$ operations.
*SHA-2* uses 256 or 512 bits of output. For an ideal hash function, this would be big enough to defeat any realistic attacker
**Example** 31:
This is an example to find a *hash collision* in `Python` assuming we have already implemented a hash function using *CTR* mode to encrypt and then *xor*ing all the blocks fo ciphertext that came out and using that as the hash output:

```
from Crypto.Cipher import AES
from copy import copy

def find_collision(message):
    new_message = copy(message)
    def swap_blocks(block_a,block_b,cblock_a,cblock_b):
        #returns the 2 blocks necessary for the message text
        #in order to swap 2 blocks in the cipher text
        eblock_a=xor_bits(block_a,cblock_a)
        eblock_b=xor_bits(block_b,cblock_b)
        new_block_a=xor_bits(eblock_a,cblock_b)
        new_block_b=xor_bits(eblock_b,cblock_a)
        return new_block_a,new_block_b
    block_size,block_enc,key,ctr=hash_inputs()
    cipher = counter_mode(message, key, ctr, block_size,block_enc)
    #swap block 1 and 2
    block_a=get_block(message, 0,block_size)
    block_b=get_block(message,1,block_size)
    cblock_a=get_block(message,0,block_size)
    cblock_b=get_block(message,1,block_size)
    new_block_a,new_block_b=swap_blocks(block_a,block_b,cblock_a,cblock_b)
    new_message[0:block_size]=new_block_a
    new_message[block_size:2*block_size]=new_block_b
    return new_message


def test():
    messages = ["Trust, but verify. -a signature phrase of President Ronald Reagan",
                "The best way to find out if you can trust somebody is to trust them. (Ernest Hemingway)",
                "If you reveal your secrets to the wind, you should not blame the wind for revealing them
                 to the trees. (Khalil Gibran)",
                "I am not very good at keeping secrets at all! If you want your secret kept do not tell
                me! (Miley Cyrus)",
                "This message is exactly sixty four characters long and no longer"]
    for m in messages:
        m = string_to_bits(m)
        new_message = find_collision(m)
        if not check(m, new_message):
            print "Failed to find a collision for ’%s’" % m
            return False
    return True

from Crypto.Cipher import AES


# Below are some functions
# that you might find useful

BITS = (’0’, ’1’)
ASCII_BITS = 8

def display_bits(b):
    """converts list of {0, 1}* to string"""
    return ’’.join([BITS[e] for e in b])

def seq_to_bits(seq):
    return [0 if b == ’0’ else 1 for b in seq]

def pad_bits(bits, pad):
    """pads seq with leading 0s up to length pad"""
    assert len(bits) <= pad
    return [0] * (pad - len(bits)) + bits

def convert_to_bits(n):
    """converts an integer ‘n‘ to bit array"""
    result = []
    if n == 0:
        return [0]
    while n > 0:
        result = [(n % 2)] + result
        n = n / 2
    return result

def string_to_bits(s):
    def chr_to_bit(c):
        return pad_bits(convert_to_bits(ord(c)), ASCII_BITS)
    return [b for group in
            map(chr_to_bit, s)
            for b in group]

def bits_to_char(b):
    assert len(b) == ASCII_BITS
    value = 0
    for e in b:
        value = (value * 2) + e
    return chr(value)

def list_to_string(p):
    return ’’.join(p)

def bits_to_string(b):
    return ’’.join([bits_to_char(b[i:i + ASCII_BITS])
                    for i in range(0, len(b), ASCII_BITS)])

def pad_bits_append(small, size):
    # as mentioned in lecture, simply padding with
    # zeros is not a robust way way of padding
    # as there is no way of knowing the actual length
    # of the file, but this is good enough
    # for the purpose of this exercise
    diff = max(0, size - len(small))
    return small + [0] * diff

def xor_bits(bits_a, bits_b):
    """returns a new bit array that is the xor of ‘bits_a‘ and ‘bits_b‘"""
    return [a^b for a, b in zip(bits_a, bits_b)]

def bits_inc(bits):
    """modifies ‘bits‘ array in place to increment by one

    wraps back to zero if ‘bits‘ is at its maximum value (each bit is 1)
    """
    # start at the least significant bit and work towards
    # the most significant bit
    for i in range(len(bits) - 1, -1, -1):
        if bits[i] == 0:
            bits[i] = 1
            break
        else:
            bits[i] = 0

def aes_encoder(block, key):
    block = pad_bits_append(block, len(key))
    # the pycrypto library expects the key and block in 8 bit ascii
    # encoded strings so we have to convert from the bit array
    block = bits_to_string(block)
    key = bits_to_string(key)
    ecb = AES.new(key, AES.MODE_ECB)
    return string_to_bits(ecb.encrypt(block))

def get_block(plaintext, i, block_size):
    """returns the ith block of ‘plaintext‘"""
    start = i * block_size
    if start >= len(plaintext):
        return None
    end = min(len(plaintext), (i+1) * block_size)
    return pad_bits_append(plaintext[start:end], block_size)

def get_blocks(plaintext, block_size):
    """iterates through the blocks of blocksize"""
    i = 0
    while True:
        start = i * block_size
        if start >= len(plaintext):
            break
        end = (i+1) * block_size
        i += 1
        yield pad_bits_append(plaintext[start:end], block_size)

def _counter_mode_inner(plaintext, key, ctr, block_enc):
    eblock = block_enc(ctr, key)
    cblock = xor_bits(eblock, plaintext)
    bits_inc(ctr)
    return cblock

def counter_mode(plaintext, key, ctr, block_size, block_enc):
    """Return the counter mode encoding of ‘plaintext"""
    cipher = []
    # break the plaintext into blocks
    # and encode each one
    for block in get_blocks(plaintext, block_size):
        cblock = _counter_mode_inner(block, key, ctr, block_enc)
        cipher.extend(cblock)
    return cipher

def counter_mode_hash(plaintext):
    block_size, block_enc, key, ctr = hash_inputs()
    hash_ = None
    for block in get_blocks(plaintext, block_size):
        cblock = _counter_mode_inner(block, key, ctr, block_enc)
        if hash_ is None:
            hash_ = cblock
        else:
            hash_ = xor_bits(hash_, cblock)
    return hash_

def hash_inputs():
    block_size = 128
    block_enc = aes_encoder
    key = string_to_bits("Vs7mHNk8e39%CXeY")
    ctr = [0] * block_size
    return block_size, block_enc, key, ctr

def _is_same(bits_a, bits_b):
    if len(bits_a) != len(bits_b):
        return False
    for a, b in zip(bits_a, bits_b):
        if a != b:
            return False
    return True

def check(message_a, message_b):
    """return True if ‘message_a‘ and ‘message_b‘ are
    different but hash to the same value"""

    if _is_same(message_a, message_b):
        return False

    hash_a = counter_mode_hash(message_a)
    hash_b = counter_mode_hash(message_b)
    return _is_same(hash_a, hash_b)
```

.

## 2.9 Strong Passwords

> Any web application that can send you your password (in cleartext)
> is doing some things very very bad.

Even if someone can get access to the database and read all the usernames and password information they can't break into your account.
Bad ideas for storing usernames and passwords:

- Store usernames and passwords in cleartext
- Generate a random key $k \in \{0, 1\}^n$ and each password is stored as $E_k(password)$ using *CFB* with $s = 8$ is a bad idea because
  - It reveals the length of the password because encrypting the password, the output that's stored password will reveal the length. Any revealed information about the password is bad and it's easy to find short passwords (easier to break).
    Solution: use hash function so the size of the output doesn't depend on the size of the input. No matter how long the password is, the output length is always the same
  - It reveals if 2 users have the same password
  - If $k$ is compromised, it reveals all passwords, because we need the key $k$ to decrypt and we need the $E_k$ function an every check in (on the account). So the program running on the server and check the passwords needs this key all the time and if the password file is compromised the key is also compromisedd as it's available in the memory and stored in the program.
    Solution: Don't invert the password to check if it's correct. Only recompute from the entered password some function that checks that with what's stored. So there is no reason to have a key stored.

A slightly better idea is:
For each user, store:
$E^n_{\text{password}}(0)$, which means the encryption is done $n$ times with the password as the key. There is no key kept secret on the server and encrypting twice, doubles the work to check a password (and if the result is 0). This scales more the attackers work than the checking work. Unix uses $n = 25 : x = E^{25}_{\text{password}}(0)$. Unix worked with *DES* which works with a 56 bit key. Longer passwords will be cut off. Due to restrictions of *DES* (only 8 bits and only upper and lowercase character ans numbers allowed) there are only $26 + 26 + 10$ possible characters. This means there are only $62^8$ possible password ($62^8 < 2^{48}$)

## 2.10 Dictionary Attacks

**Definition** *Dictionary Attacks*
An attacker can pre-compute a dictionary. Pre-compute

$$E^n_w(0)$$

(Unix: $n = 25$) for a set of common password $w$, store those pre-computed values and then every new password file that's compromised check all the passwords against this list and have a good likelyhood of finding some accounts that can be broken into.
Making Dictionary Attacks Harder:

- Train (coerce) user to pick better passwords (won't work properly)
- Protect encrypted passwords better
- add *salt*

### 2.11 Salted Password Scheme

The password file include

- username (userID)
- salt
- encrypted password

**Definition** *salt*
*Salt* are random bits (Unix: 12 bits). They don't need to be kept secret. The *salt* bits are different for each user. The encrypted password is the result of hashing the salt concatenated with the user's password.

$$x = E^n_{\text{salt}||\text{password}}$$

That means: as long as the salts are different, even if the passwords are the same, the encrypted passwords will be different (Unix: $DES^{25}_{\text{salt}||\text{password}}(0)$). An attacker who compromises the password file has not much harder work because the salt is not kept secret and the attacker needs to try possible passwords concatenated with that salt and find one that matches the hash value. An attacker with an offline dictionary attack has $2^n$ (number of different $n$ bits long salts containing only 0 and 1 ; $n$ lenght of salt) more work because the attacker needs to pre-compute that dictionary for all the different salt values to be able to look for password matches. *Salting* adds a lot of value for very littel cost. Just by some extra bits which don't need to be kept secret, that are stored in the password file.

### 2.12 Hash Chain, S/Key Password Scheme

**Definition** *Hash Chain*
A *hash chain* is the result of hasing a value (secret) $s \in \{0, 1\}$ over and over again: computing hash of that secret: $H(s)$ and do this again: $H(H(s))$ and so on.
**Example** 32:
A user $u$ checks into a webpage (server $S$) gets the hash function and computes $H(s), H(H(s)), \ldots$. The only thing the server stores is the last value of this *hash chain*. So for hashing $n$-times the server only stores $H^n(s)$.
The first log in protocol works as follows (assume $n = 100$):

1. $U$ gets the hash function an computes a hash chain until $H^{n-1}(s) = H^{99}(n)$ with his secret $s$.
2. $U$ sends $p = H^{99}$ to $S$

3. $S$ checks if $H(p) = x = H^{100}(s)$

4. $S$ sets the next hash chain endpoint to $n - 2 = 98$

$$\begin{tikzcd}
U & S \\
H(s), H(H(s)), \dots \arrow[dr, "p"] & \\
& H(p) \overset{?}{=} x, x \mapsto p
\end{tikzcd}$$

The next log in requires user sends $H^{98}(s)$. The *hash chain* is going backwards and hashes can only be verified in one direction. The hash is hard to compute in one direction and easy in the other (valuable property of the hash function). If someone just knows $x$ (intercepts $p$), knows the previous password value and can easy compute $H^{100}(s), H^{101}(s), \dots$ but $H^{98}(s)$ is hard to compute.
**Definition** *S/Key Password Scheme*
In *S/Key Password Scheme* the server would generate the hash chain of length $n$:

$$H^n(s), H^{n-1}(s), \dots, H(s)$$

The user prints out the received hash values.
The server only stores the last entry in the hash chain and so what's stored on the server can not used to log in. The downside is that the user has to carry around a list of passwords and has a look on the list on ervery log in, use the correct password, cross this one off and uses the next password on the next log in. And the user has to get a new password list after using the last password.

# 3 Key Distribution

## 3.1 Key Distribution

In *symmetric ciphers* all participating parties have the same key to do encryption and decryption. The keys may be identical or there may be a simple transformation to go between the two keys. The important property that makes a cryptographic *symmetric* is that the same key is used to encrypting and decrypting. If 2 or more parties want to talk to each other, they first have to agree on a *secret* key. That means there has to be a way for the parties to communicate this key without exposing the key. Earlier this was done with a *codebook* (which was physically distributed to the endpoints) which is not practical. Nowadays there are different ways how to establish a secure key.

## 3.2 Pairwise Shared Keys

**Definition** *Pairwise Shared Key*
A *pairwise shared key* is a secure key only used by 2 parties to communicate to each other. That means every party has a different key to any other party. This works with a small number of people otherwise it gets pretty expensive, due to the number of different keys.
**Example** 33:
Consider 4 parties: $A, B, C, D$. Then

- $A$ has a key with $B, C, D$
- $B$ has a key with $C, D$
- $C$ has a key with $D$.

For 4 parties 6 keys are needed.
**Theorem 3.2.1:**
*The number of pairwise keys for $n$ people is:*

$$\sum_{i=1}^{n-1}(n-i) = (n-1) + (n-2) + \cdots + \underbrace{(n-(n-1))}_{=1} = \sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \tag{3.3}$$

**Proof:**
Consider $n$ people. The first person has to make a secret key to everyone excepts himself. Therefore in the first step $(n-1)$ keys are needed. The second person has to make a secret key to everyone excepts himself and the first person (exists already). Therefore in the second step $n-2$ keys are needed and so on until the penultimate person has to make a key only to the $n$th person. The last one has already a key to the other parties.
Reading the summation backwards $1 + 2 + 3 + \ldots + (n-2) + (n-1)$ we get a much simpler summation.
With induction it's simply to show that $\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2}$:
Initial step: $n = 1$:

$$\sum_{i=1}^{1-1} i = \sum_{i=1}^{0} = 0 = \frac{1 \cdot (1-1)}{2}$$

Assume the formula (3.3) is true for $n$, then for $n+1$ is

$$\begin{aligned}\sum_{i=1}^{(n+1)-1} i = \sum_{i=1}^{n} = 1 + 2 + \cdots + (n-2) + (n-1) + n &= \frac{n \cdot (n-1)}{2} + n = \frac{n \cdot (n-1)}{2} + \frac{2n}{2} \\ &= \frac{n^2 - n + 2n}{2} = \frac{n^2 + n}{2} \\ &= \frac{(n+1) \cdot n}{2} = \frac{(n+1) \cdot ((n+1)-1)}{2}\end{aligned}$$

□

**Example** 34:
Assume the network has 100 people using pairwise shared keys. The the number of needed keys are

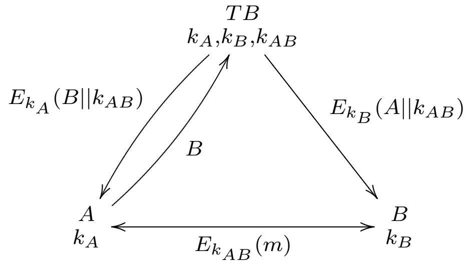
$$\sum_{i=1}^{99} i = \frac{100 \cdot (100-1)}{2} = 4950$$

For a goup of $10^9$ people we would need approximately $5.0 \cdot 10^{17}$ keys to have pairwise keys.

## 3.3 Trusted Third Party

**Definition** *Trusted Third Party*
A *trusted third party* is some trustworthy place $TP$ which has a shared secret with each network

individual.
**Example** 35:
Assume 2 parties $A, B$ in the network and a trustworthy place $TP$ has a secret key to each party: One secret key $k_A$ to $A$ and one secret key $k_B$ to $B$. When $A$ and $B$ want to communicate the protocol looks as follows:



This means:

1. $A$ sends a request to $TP$ for a communication with $B$.
2. $TP$ generates a random key $k_{AB} \in \{0,1\}^n$
3. $TP$ sends $A$ the encrypted key $k_{AB}$ concatenated with $B$'s name (userID) with the shared key $k_A \in \{0,1\}^n$ and $B$ the encrypted key $k_{AB}$ concatenated with $A$'s name (userID) with the shared key $k_B \in \{0,1\}^n$.
4. $A$ and $B$ can communicate with the key $k_{AB}$.

The problems with a *trusted third party* are:

- $TP$ can read all messages between $A$ and $B$, because $TP$ generates the key $k_{AB}$, so $TP$ can read every intercepted message over the insecure channel between $A$ and $B$.
- $TP$ can impersonate every customer. $TP$ can generate a fake conversation, make it seem like $A$ is communicating with $B$.
- An attacker can maybe tamper with $E_{k_A}(B||k_{AB})$ to steal $k_{AB}$ (depends on encryption and modes of operation used)

So the $TP$ is a more theoretical thinking than even implementing.

## 3.4 Merkle's Puzzle

**Definition** *Merkle's Puzzle*
The *Merkle's Puzzle* was the first key exchange protocol, without the parties sharing a secret key with another or third party (trusted place).
The idea behind *Merkle's Puzzle* is that 2 parties $A, B$ want to share a secred key. First the parties agree on some parameters:

- Encryption function: $E$
- Security parameters: $s, n, N$ with $s \leq n$

The protocol works as follows:

1. $A$ creates $N$ secrets:
$$s' = [\{0,1\}^s, \{0,1\}^s, \ldots, \{0,1\}^s]$$
which means every secret is a $s$ bit long random number.

2. Then $A$ creates a set of $N$ puzzles:
$$p = [E_{s'_1||0^{n-s}}(\text{'Puzzle': 1}], E_{s'_2||0^{n-s}}(\text{'Puzzle': 2}], \ldots, E_{s'_N||0^{n-s}}(\text{'Puzzle': } N]$$
which means the $i$-th encrypted puzzle uses the $i$-th secret concatenated with enough 0s, so that the right key length is achieved. The message include the word *Puzzle* followed by the corresponding number of the secret(in $AES$ every key has 128 bits).

3. $A$ shuffles the puzzle set and sends all $N$ puzzles to $B$.

4. $B$ picks randomly one of the puzzles $m$ and does a brute force key search. That is, $B$ tries to
$$D_{g||0^{n-s}}(m)$$
with a guessed key $g \in \{0,1\}^s$ and a known decryption function (inverse of $E$).

5. Eventually $B$ is going to find one that decrypt to
$$\text{'Puzzle': } p$$
where $1 \leq p \leq N$ the number of the puzzle.

6. So $B$ knows the guess $g$ and the puzzle number $p$. For longer keys it's better to use:
$$\text{'Puzzle': } p, \text{'Key': } k$$
where $k$ is a new random key of any length associated with that secret.

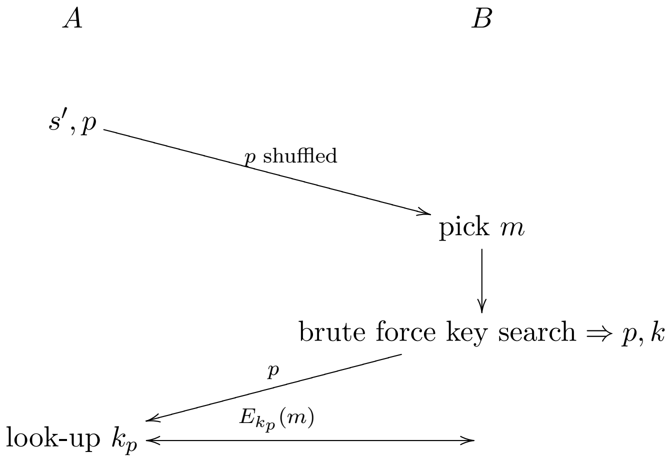
7. $A$ will keep track of the keys in a list
$$k' = [k_1, k_2, \ldots, k_N]$$

8. If $B$ decrypts a puzzle, $B$ acquires the puzzle number and the key number.

9. $B$ sends the number of the puzzle back to $A$

10. $A$ does a lookup in the key list and figure out which key was the one in the puzzle decrypted.

The protocol looks as follows:



The *Merkle's Puzzle* is an impractical idea as it requires a lot of secrets and puzzles to create for $A$ and a good bandwith to send this information to $B$ so that an attacker can't get the key too easy.

**Theorem 3.4.1:**
*Assuming perfect encryption and random keys, an attacker, who got all sended information, expects to need* $\frac{N}{2}$ *times as much as* $B$ *to find the key*

**Proof:**
Due to $B$ randomly picks a puzzle out of the shuffled set $p$ and solve it with a brute force key search and sends $A$ the number of the puzzle back, the attacker doesn't know which of the encrypted puzzles correspond to this number. The attacker has to try to break all of the encrypted puzzles and will be expected to find the one that matches the one that $B$ picked after trying about $\frac{N}{2}$ of them □

## 3.5 Diffie-Hellman Key Exchange

**Definition** *Properties of Multiplication*
The *Properties of Multiplication* (used in $DHKE$) are

- Commutativity: $\forall a, b \in \mathbb{Z}, \forall n \in \mathbb{N}$ :

$$a \cdot b \,(\text{mod} n) = b \cdot a \,(\text{mod} n)$$

- Commutativity of powers (follows directly from commutativity): $\forall a \in \mathbb{Z}, \forall c, b \in \mathbb{N}, \forall c, b \in \mathbb{Z} \Leftrightarrow a + \mathbb{Z}n \in (\mathbb{Z}/\mathbb{Z}n)^{\times}, \forall n \in \mathbb{N}$:

$$\left(a^b\right)^c (\text{mod} n) = a^{bc} (\text{mod} n) = a^{cb} (\text{mod} n) = (a^c)^b \,(\text{mod} n)$$

Note: even using the mod operation the properties are valid.

**Definition** *Primitive Root*
For $q \in \mathbb{P}$ consider the multiplicative group $((\mathbb{Z}/\mathbb{Z}q), \odot)$ (hence: operation is multiplication). We know that $(\mathbb{Z}/\mathbb{Z}q)^{\times}$ has $q-1$ elements, namely

$$(\mathbb{Z}/\mathbb{Z}q)^{\times} = \{1, 2, \ldots, q-1\}$$

We say that a number $g \in \mathbb{Z}$ is a *primitive root* of $q$ if and only if

$$(\mathbb{Z}/\mathbb{Z}q)^{\times} = \{g^i \bmod q | i = 1, 2, \ldots, q-1\}$$

**Example** 36:
Consider $q = 7$ a prime number, then $g = 3$ is a primitive root of $q$ because

$$\begin{array}{ll} g^1 = 3 & 3 \equiv 3 \bmod 7 \\ g^2 = 9 & 9 \equiv 2 \bmod 7 \\ g^3 = 27 & 27 \equiv 6 \bmod 7 \\ g^4 = 81 & 81 \equiv 4 \bmod 7 \\ g^5 = 243 & 243 \equiv 5 \bmod 7 \\ g^6 = 729 & 729 \equiv 1 \bmod 7 \end{array}$$

Every number $x \in \{1, 2, 3, 4, 5, 6\}$ occurs once so 3 is a primitive root of 7.

**Theorem 3.5.1:**
*If* $p$ *is a prime number and* $p > 2$*, then there are always at least* 2 *primitive roots.*

**Example** 37:
The `Python` code for finding all primitive roots of an integer $n \in \mathbb{P}$ may look as follows:

```
def square(x):
   return x*x
def mod_exp(a,b,q):    #recursive definition
   if b==0:            #base case: a^0=1
      return 1
   if b%2==0:
      return square(mod_exp(a,b/2,q))%q
   else:
      return a*mod_exp(a,b-1,q)%q
def primitive_roots(n):
    roots=[]
    def is_primitive_root(r):
        s=set()
        for i in range(1,n):
            t=mod_exp(r,i,n)
            if t in s:
                return False
            s.add(t)
        return True
    for i in range(2,n):
        if is_primitive_root(i):
            roots.append(i)
    return roots
```

.

**Definition** *Diffie-Hellman Key Exchange(DHKE)*
The *Diffie-Hellman Key Exchange* allows 2 parties without any prior agreement be able to establish a shared secret key.
The protocol for the $DHKE$ with 2 parties $A, B$ is

1. $A$ picks some values $q$ (large prime number) and $g$ (primitive root of $q$) and sends them to $B$.
2. $A$ selects a random value $x_A \in \{0,1\}^n$ and $B$ selects a random value $x_B \in \{0,1\}^n$
3. $A$ computes $y_A = g^{x_A} \bmod q$
4. $B$ computes $y_B = g^{x_B} \bmod q$
5. $A$ and $B$ exchange their computed values
6. $A$ computes the key $k_{AB} = y_B^{x_A} \bmod q$
7. $B$ computes the key $k_{BA} = y_A^{x_B} \bmod q$

This looks as follows:

$$\begin{tikzcd}
A & B \\
q,g,x_A \arrow[dr, "q,g"] & x_B \\
y_A \arrow[dr, "y_A"] & y_B \arrow[dl, "y_B"] \\
k_{AB} & k_{BA}
\end{tikzcd}$$

First the 2 parties $A, B$ agree on some values: $q$ a large prime number and $g$ a primitive root of $q$. Then $A$ and $B$ select a random of length $n$. $A$ computes $y_A$ and $B$ computes $y_B$ the way shown in the protocol above. Then $A$ and $B$ exchange their computet values and comute the key $k_{AB}$ and $k_{BA}$.

**Theorem 3.5.2 (Correctness Property of DHKE)**
*In the DHKE protocol is:*

$$k_{AB} = k_{BA}$$

**Proof:**

$$\begin{array}{rcl} A: & & \\ k_{AB} & = & y_B^{x_A} \bmod q \\ & = & (g^{x_B})^{x_A} \bmod q \\ & = & g^{x_B x_A} \bmod q \end{array}$$

$$\begin{array}{rcl} B: & & \\ k_{BA} & = & y_A^{x_B} \bmod q \\ & = & (g^{x_A})^{x_B} \bmod q \\ & = & g^{x_A x_B} \bmod q \end{array}$$

Due to the commutativity of power in multiplication is

$$g^{x_B x_A} = g^{x_A x_B}$$

□

**Theorem 3.5.3 (Security Property of DHKE (against passive attacker))**
*The passive (listen only) attacker gets the public values $q, g, y_A, y_B$. It's possible to show that reducing same known hard problems to the Diffie-Hellman problem. This shows, anyone who can solve the Diffie-Hellman problem efficiently would also be able to solve some problem that we already known as hard. To break efficiently the Diffie-Hellman problem need a way to compute descrete logarithm efficiently.*
*The security of the scheme depends on it being difficult to solve*

$$a^x = b \bmod n$$

*for $x$ given $a, b, n$ (discrete logarithm problem).*
*If modulus $n$ is not prime, the Diffie-Hellman scheme would still be correct (2 participants produce the same key) but might not be sure, because of some non prime number calculating the discrete logarithm is not hard.*

**Theorem 3.5.4 (Security Property of DHKE (against active attacker))**
*The protocol of DHKE isn't secure against an active (change, write sended messages) attacker. If an attacker can change the value of $y_A$ and the value of $x_B$ (e.g. if the attacker changes $y_A$ and $y_B$ to 1 then the secret key will be 1 raised to the personal secret key which is still 1). or the attacker can get all values $q, p, y_A$ and establish a secure fake connection with $A$ with the key $k_{AM}$ and $B$ with the keys $k_{BM}$. So the attacker is in the middle:*

$$\underbrace{c = E_{k_{AM}}(m)}_{A} \overset{c}{\rightarrow} \underbrace{m = D_{k_{AM}}(c), m \rightarrow m', c' = E_{k_{BM}}(m')}_{M} \overset{c}{\rightarrow} \underbrace{m' = D_{k_{BM}}(c')}_{B}$$

*This means: $A$ sends an encrypted message to $M$ thinking it's $B$. $M$ can now decrypt the message and change it and forward it to $B$ who thinks he receives a message from $A$.*

## 3.6 Discrete Logarithm Problem

**Definition** *Continous Logarithm*
The solution of the equation for known $a, b \in \mathbb{R}$

$$a^x = b$$

is

$$x = \log_a b$$

which can be solved in efficient ways.
**Example** 38:
It is $\log_2 8 = 3$ because $2^3 = 8$.
**Definition** *Discrete Logarithm*
The solution of the equation for known $a, b, n \in \mathbb{R}$

$$a^x = b \bmod n$$

is

$$x = \text{dlog}_a b$$

where dlog is the discrete logarithm. This turns out to be really hard problem and $n$ is a large prime number. It's not clear that the dlog always exists (for certain choices of $a, b, n$ it would not exists).
Here $a$ is a *generator* which means

$$a^1, a^2, \ldots, a^{n-1}$$

is a permutation of $1, 2, 3, \ldots, n-1 \in \mathbb{Z}_n$ (that means in general: every number occurs exactly once).
**Conlusion:**
Given a power it's hard to find the corresponding number in the list in Example 36 (for greater values than 7). The fastest known solution need exponential time (not polynomial). That means the only way to solve this is to try all possible powers until you find the one that work. You can do a little better by trying powers in a clever way (and to exclude some powers) but there is no better ways than doing this exponential search which is exponential in the <u>size</u> of $n$ (linear in the <u>value</u> of $n$).
If it's possible to compute dlog efficiently then the attacker, who knows $q, g, y_A, y_B$ hat to compute

$$k = y_B^{\text{dlog}_g y_A \text{mod } q} \bmod q$$

where $\text{dlog}_g y_A \text{mod } q = x_A$ and $k$ is the key.

## 3.7 Decisional Diffie-Hellman Assumption

**Definition** *Decisional Diffie-Hellman Assumption*
Assume discrete lograithm is hard then breaking *Diffie-Hellman* implies solving discrete logarithm efficiently (not provable). The security of *Diffie-Hellman* relies on a strong assumption: the *Decisional Diffie-Hellman Assumption* (a bit circular).
The *Decisional Diffie-Hellman Assumption* is with former notation $x = x_A, y = x_B$:

$$k = g^{xy} \bmod q$$

is indistinguishable from random given $q, g, g^x, g^y$ (intercepted message).
This assumption is <u>not</u> true for certain values.

## 3.8 Implementing Diffie-Hellman

The first issue is to do fast modular exponentiation:

$$g^{x_A} \bmod q$$

where $x_A$ is some large random number and $q$ is a large prime number and $q$ a primitive root of $p$.

**Theorem 3.8.1:**

*Computing $a^n$ with $n \in \mathbb{N}, a \in \mathbb{R}$ can be done using $\mathcal{O}(\log n)$ multiplications. That means modular exponentiation scales linear in the size (bits to represent) of the power. It follows: making the power $n$ a very large number and still compute $a^n$ quickly.*

**Example** 39:

Using $x^{2a} = (x^a)^2$ then for $2^{20}$ there are at least 5 multiplications needed:

$$2^{20} = \left(2^{10}\right)^2 = \left(\left(2^5\right)^2\right)^2 = \left(\left(2 \cdot 2^4\right)^2\right)^2 = \left(\left(2 \cdot \left(2^2\right)^2\right)^2\right)^2 = \left(\left(2 \cdot (2 \cdot 2)^2\right)^2\right)^2$$

as seen 5 multiplications are needed to compute $2^{20}$

**Example** 40:

A fast modular exponentiation that turns 3 values $a, b, q$ to $a^b \bmod q$ which has a running time that is linear in the size of $b$ is given by this `Python` code:

```
def square(x):
   return x*x

def mod_exp(a,b,q):     #recursive definition
   if b==0:             #base case: a^0=1
      return 1
   if b%2==0:
      return square(mod_exp(a,b/2,q))%q
   else:
      return a*mod_exp(a,b-1,q)%q
```

.

**Theorem 3.8.2:**

*The fast modular exponentiation technique used in this notes suffers from an important security flaw. The time it takes to execute depends on the value of the power which may be secret. This means, an attacker who can measure precisely how long the encryption takes can learn something about the key.*

**Example** 41:

Assume modulus and multiplication costs:

$$\text{mod and } \cdot \text{ costs} \begin{cases} 0 \text{ time unit by 1 or 2} \\ 1 \text{ time unit otherwise} \end{cases}$$

In binary, telling if a number is odd or even depends on the last digit. E.g.

$$\begin{array}{rcl} 10 & = & 1010 \rightarrow \text{ last digit } 0 \Rightarrow \text{ even} \\ 11 & = & 1011 \rightarrow \text{ last digit } 1 \Rightarrow \text{ odd} \end{array}$$

and dividing by 2 is a shift right:

$$\begin{tikzcd} 20 \arrow[r, ":2"] & 10 \arrow[r, ":2"] & 5 \\ 10100 & 01010 & 00101 \end{tikzcd}$$

In the *modulo exponentiation* routine:

- If the exponent is even, we divide by 2 and this costs 1 multiplication
- If the exponent is odd, we substract 1 from it which will make it even and then we divide by 2, which costs in total 2 multiplications

Now written die exponent in binary it follows:

- For every 1 in the exponent (in binary) we do 2 multiplications
- For every 0 in the exponent (in binary) we do 1 multiplication

Having 4 exponents the most expensive is:

| dec. | bin | costs |
|---|---|---|
| 1023 | 0001111111111 | $3 \cdot 0 + 20 \cdot 2 = 20$ |
| 1025 | 0010000000001 | $1 \cdot 2 + 9 \cdot 1 + 1 \cdot 2 = 13$ |
| 4096 | 1000000000000 | $1 \cdot 2 + 12 \cdot 1 = 14$ |
| 4097 | 1000000000001 | $1 \cdot 2 + 11 \cdot 1 + 1 \cdot 2 = 15$ |

Therefore 1023 is here the most expensive operation.
**Example** 42:
Suppose $A$ and $B$ execute the *Diffie-Hellman* protocol, but $A$ picks a value for $g$ that is <u>not</u> a primitive root of $q$. Then

- the generated key would be more vulnerable to an eavesdroppter
- the number of possible keys that could be generated would be smaller than $q$

Remember that the *Diffie-Hellman* protocol relies upon the difficulty of *discrete logs.*
If $g$ is a primitive root of $q$ then

$$\left|\{g^1, g^2, g^3, \ldots\}\right| = q$$

If $g$ is <u>not</u> a primitive root of $q$ then

$$\left|\{g^1, g^2, g^3, \ldots\}\right| < q$$

This implies if $g$ is <u>not</u> a primitive root of $q$ it is easier to solve the *discrete log*, which means the generated key would be more vulnerable to an eavesdropper and also means that the number of possible keys that could be generated would be smaller then $q$.

## 3.9 Finding Large Primes

**Theorem 3.9.1:**
*There are infinitely many prime numbers.*

**Proof (by contradiction - Euclid)**
Assume there are a limited set of primes:

$$P = \{p_1, p_2, \ldots, p_n\}$$

with $|P| = n$.
Computing the product

$$p = p_1 \cdot p_2 \cdot \cdots \cdot p_n = \prod_{i=1}^{n} p_i$$

Then

$$p' = p + 1 = p_1 \cdot p_2 \cdot \dots \cdot p_n + 1 = \prod_{i=1}^{n} p_i + 1$$

is not a prime number due to the assumption that $P = \{p_1, p_2, \dots, p_n\}$ are the only primes and $p' \notin P$.
Since $p'$ is not prime, it must be a product of a prime and an integer $q$. So $p'$ is a *composite* number:

$$p' = p_i \cdot q$$

But

$$p' = p_1 \cdot p_2 \cdot \dots \cdot p_n + 1 = p_i \cdot q$$

Dividing by $p_i$ results in

$$\frac{p_1 \cdot p_2 \cdots p_n + 1}{p_i} = q \Leftrightarrow p_1 \cdot p_2 \cdot \dots \cdot p_{i-1} \cdot p_{i+1} \cdots p_n + \frac{1}{p_i} = q$$

Since $q$ was an integer and $p_1 \cdot p_2 \cdot \dots \cdot p_{i-1} \cdot p_{i+1} \cdots p_n$ is an integer and $p_i \in P \Rightarrow p_i \neq 1$ so $\frac{1}{p_i} \notin \mathbb{Z}$. It follows

$$p_1 \cdot p_2 \cdot \dots \cdot p_{i-1} \cdot p_{i+1} \cdots p_n + \frac{1}{p_i} \notin \mathbb{N}$$

but $q \in \mathbb{N}$. Contradiction.
Thus the assumption is false, it follows: there are infinitely many primes. □

**Definition** *Asymptotoc Law Of Distribution Of Prime Number (Density of Primes)*
The *Density of Primes* is the number of primes $N$ that occur given an upper bound $x$:

$$N \leq x \sim \frac{x}{\ln x} \tag{3.4}$$

Therefore, assuming the set of all prime numbers $\mathbb{P}$, the probability that a number $x$ is prime is given by:

$$P(x \in \mathbb{P}) \sim \frac{1}{\ln x}$$

**Theorem 3.9.2:**
*The expected number of guesses N needed to find an n-decimal digit long prime number is*

$$N \sim \frac{\ln 10^n}{2} \tag{3.5}$$

**Proof:**
Follows directly of the asymptotic law of distribution of prime numbers

$$P(x \in \mathbb{P}) \sim \frac{1}{\ln x}$$

It's also the probability that a randomly selected odd integer in the invervall from $10^y$ to $10^x$ is prime is approximately using (3.4)

$$\frac{\text{number of primes}}{\text{number of odd integers}} = \frac{\frac{10^x}{\ln 10^x} - \frac{10^y}{\ln 10^y}}{\frac{10^x - 10^y}{2}} = \frac{2}{10^{x-y} - 1}\left(\frac{10^{x-y}}{x \ln 10} - \frac{1}{y \ln 10}\right)$$

□

**Example** 43:
The probability of a 100 digit number is prime is approximately

$$\frac{2}{9}\left(\frac{10}{100\ln 10}-\frac{1}{99\ln 10}\right)\sim 0.008676$$

and on average

$$\frac{1}{0.008676}\sim 115$$

odd 100-digit numbers would be tested before a prime number is found.
Or directly from (3.5):

$$\frac{\ln 10^{100}}{2}\sim 115$$

**Example** 44:
A `Python` procedure for finding large prime numbers for some randomly selected large number $x$ may be look as follows:

```
def find_prime_near(x):
   assert x%2==1           #only odd numbers
   while True:
      if is_prime(x):
         return x
      x=x+2             #skip even numbers

def is_prime(x):
   for i in range(2,x):
      if x%i==0:        #test divisibility
         return False
   return True
```

but this is exponentially in the size of $x$ so wouldn't work for large $x$ efficiently and the prime test is very naive.

## 3.10 Faster Primal Test

**Definition** *Faster Primal Test, Probability Test*
A *faster primal test* uses a *probability test*:

$$x \text{ passes the test } \Rightarrow P(x \notin \mathbb{P}) \leq 2^{-k}$$

That means if $x$ passes the primal test, then the probability that $x$ is composite (not prime) is equal or less then some value $2^{-k}$. Noramlly $k = 128$.

## 3.11 Fermat's Little Theorem

**Definition** *Fermat's Little Theorem*
A useful property of prime numbers is the *Fermat's litte theorem* with $p \in \mathbb{P}, a \in \mathbb{N} : 1 \leq 1 < p$ is

$$a^{p-1} \equiv a \bmod p$$

or if $a \neq n \cdot p$ for some $n \in \mathbb{Z}$ then

$$a^{p-1} \equiv 1 \bmod p$$

With this, it's easy to try a lot of $a$ and if it always holds $p$ is probably prime but some composite numbers *Charmichael* numbers where the $a^{p-1} \equiv 1 \bmod p$ also holds for all $a$ relatively prime $p$. This test isn't fast enough to try all $a$ with $1 \leq a < p$ because it's runningtime is exponential in the size of $p$.

## 3.12 Rabin-Miller Test

**Definition** *Rabin-Miller Test*
The *Rabin-Miller test* is a probabilistic prime number test.
Start by guessing an odd numer $n \in \mathbb{N}$ (if $n$ is even, then it's not prime). If $n$ is odd then $n$ can be broke into

$$n = 2^t s + 1$$

Next choose some random $a \in [1, n) \subset \mathbb{N}$. If $n \in \mathbb{P}$ then

$$a^s \equiv 1 \bmod n \tag{3.6}$$

or for some $0 \leq j < t$:

$$a^{s2^j} \equiv n - 1 \bmod n \tag{3.7}$$

The big advantage is it's sufficient trying only a small number of values.
**Example** 45:
For $n < 1373653$ it's sufficient to try $a = 2$ and $a = 3$.
**Theorem 3.12.1:**
*Finding any value that sadisfied* (3.6) *or* (3.7) *than this value is composite.*
*The difference of the Rabin-Miller test to the Fermat test is that the probability that a composite number passes the test is always less than some constant and there are no bad numbers like the Charmichael number in the Fermat test.*

**Example** 46:
If the probability that the guess $n$ is composite is less than $2^{-128}$ we need to run the test 64 times for random selected $a$ values because

$$\left(2^{-2}\right)^{64} = 2^{-128}$$

Therefore 64 test runds would be sufficient.
**Example** 47:
The *Rabin -Miller* test in Python is:

```
from random import randrange

#randrange returns a random integer between start and end:
#r=randrange(start,end)
def square(x):
   return x*x
def mod_exp(a,b,q):     #recursive definition
   if b==0:             #base case: a^0=1
      return 1
   if b%2==0:
      return square(mod_exp(a,b/2,q))%q
   else:
      return a*mod_exp(a,b-1,q)%q

def rabin_miller(n,target=128):
```

```
def calculate_t(n):
    n=n-1
    t=0
    while n%2==0:
        n=n/2
        t=t+1
    return t
if n%2==0:
    return False
#n=(2**t)*s+1
t=calculate_t(n)
s=(n-1)/(2**t)
n_test=target/2 #number of test needed to get desired probability
tried = set()
if n_test>n:
    raise Exception('n is too small')
for i in range(n_test): #randomly pick a and if not tried before
    while True:
        a=randrange(1,n)
        if a not in tried:
            break
    tried.add(a)
    #2 tests in Rabin-Miller
    #1st test:
    if mod_exp(a,s,n)==1:
        continue
    #2nd test:
    found = False
    for j in range(0,t):
        if mod_exp(a,2**j*s,n)==(n-1):
            found=True
            break
    if not found:
        #failed both tests => leave
        return False
#if made it untill here, all tests passed
return True
```

.

# 4 Asymmetric Cryptosystems

**Definition** *Asymmetric Cryptosystems I*
In difference to symmetric cryptosystems, where the key $k$ used for encryption and decryption is the same. This leads to distributing the key problem between 2 parties but even solving that problem using the same key for encryption and decryption limits whats possible to do with the cryptosystem.
With *asymmetric cryptosystems* there are different keys for encrypting $k_U$ (public key - no need

to be kept secret) and decrypting $k_R$ (private key). That means:

$$\begin{tikzcd}
m \arrow[r] & E \arrow[rr, squiggly, "c"] & & D \arrow[r] & m \\
& k \arrow[u] & \text{symmetric} & k \arrow[u] & \\
& k_U \arrow[uu, bend left] & \text{asymmetric} & k_R \arrow[uu, bend right] &
\end{tikzcd}$$

**Definition** *Correctness Property of Asymmetric Cryptosystems*
Assume 2 parties $A, B$. $A$ wants to send a private message to $B$ over an insecure channel. $A$ and $B$ haven't a shared key but $A$ has $B$'s public key $k_{UB}$ and $B$ has his own private key $k_{RB}$ that corresponds to the public key $k_{UB}$. So $A$ does:

$$\begin{tikzcd}
m \arrow[r] & E \arrow[r, squiggly, "c"] & D \arrow[r] & m \\
& k_{UB} \arrow[u] & k_{RB} \arrow[u] &
\end{tikzcd}$$

So only knowing $B$'s private key $k_{RB}$ can decrypt the message. So the correctness property here is:

$$D_{k_{RB}}\left(E_{k_{UB}}(m)\right) = m$$

because decrypting with the private key is the inverse of encrypting with the public key.

## 4.1 Signatures

**Definition** *Physical Signature*
*Physical signatures* are used to authenticate documents. When something is signed, that means that the signer is responsible for the message in the document.
*Physical signatures* don't work well for the purpose on digital documents because the signature can be cut and paste or a signed document can be modiefied after it's signed.
**Definition** *Digital Signature*
A *digital signature* is a signature on a digital document. The person who signed the document agreed to the message and the message can not be changed without also breaking the signature.
**Example** 48:
Asymmetric Cryptography used by 2 parties:
Assume 2 parties $A, B$. $A$ signs a message, transmit that message to $B$, $B$ should be able to read the message and know that this message had only come from $A$. This may look as follows:

$$\begin{tikzcd}
m \arrow[r] & D \arrow[r, squiggly, "c"] & E \arrow[r] & m \\
& k_{RA} \arrow[u] & k_{UA} \arrow[u] &
\end{tikzcd}$$

That means:
$A$ uses the own private key $k_{RA}$ (which only $A$ should have) to encrypt the message $m$. Anyone who has $A$'s public key $k_{UA}$ (including $B$) can decrypt the message and if it decrypts to a reasonable message $B$ knows that it came from $A$.

The correctness of this depends on the inverse property. In order for signatures to work we need to be able to encrypt and decrypt in the reverse order and have them still be inverse:

$$E_{k_{UA}}\left(D_{k_{RA}}(m)\right) = m$$

using the apropriate private key for decryption and the public key for encryption.
**Definition** *One-Way Function, Trapdoor Function*
For building an asymmetric cryptosystem we have to build a *one-way function, trapdoor function* which is a function that is easy to compute in one direction $f(x)$ and hard to compute the other direction $f^{-1}(y)$:

$$\begin{tikzcd} x \arrow[r, bend left, "f(x)"] & y \arrow[l, bend left, "f^{-1}(y)"] \end{tikzcd}$$

In asymmetric cryptosystems we want to reveal the function (easy direction) and the reverse direction is hard but we want also some way to do the reverse (hard) direction easily if we know some secret.
**Definition** *Asymmtric Cryptosystems II*
In an *asymmtric cryptosystem* it's hard to do the reverse direction unless you have a key but revealing the easy way to do forward (easy) direction does not reveal the easy way to do the reverse (hard) direction.

## 4.2 RSA Cryptosystems

**Definition** *RSA Cryptosystems*
*RSA Cryptosystems* names after *Rivest, Shamir and Adleman* is the most famous asymmetric cryptosystem used worldwide.
It works as follows:

- The public key is a pair of 2 numbers: $k_U = (e, n)$ where $n = p \cdot q$ the product of 2 large prime numbers and $e$ is a secret number.
- The secret key is a pair of 2 numbers: $k_R = (d, n)$ where $n$ is the *modulus* as before and $d$ is a secret number.
- To perform encryption fo some message $m$:

  $$E_{k_U}(m) = m^e \bmod n$$

- To perform decryption on a ciphertext $c$ using the secret key $k_R$:

  $$D_{k_R}(c) = c^d \bmod n$$

.
**Example** 49:
Suppose $n = 6371$. Then the maximum value for $m$ is 6370 because we need a mapping and encryption to be invertable given that the output is mod$n$ that the possible values woulde be in $\{0, 1, 2, \ldots, 6370\}$. Knowing that

$$E_{k_U}(m) = m^e \bmod n$$

maps each message to a unique ciphertext otherwise this wouldn't be invertable (if 2 messages would map to the same value, we woulnd't know which to decrypt to) that would definitely be the case if we have more than modulus number of values (otherwise we used one value twice). As long are $m$ and $n$ are relatively prime (which means their greatest common divisor is 1) we should generate all the different values and use each of them for a different message. It is

- $m = 0 \Rightarrow m^e \bmod n = 0$ so the encryption function doesn't depend on the key
- $m = 1 \Rightarrow m^e \bmod n = 1$ so the encyption function doesn't depend on the key

.

**Theorem 4.2.1:**
*So it's dangerous to use small values for m (see later)*

**Proof:**
In

$$E_{k_U}(m) = m^e \bmod n$$

the key $k_U$ is public and we assume that an adversary knows $e$. If there is only a small possible set of $m$ values the adversary can just try them all and see which one maps to which. □

**Example** 50:
Let $M$ be a 2-digit number. The value of $m$ is encrypted using $RSA$ and $A$'s private key $k_{RA}$ using no padding:

$$E_{k_{RA}}(m) = c$$

Given the values of $A$'s public key $k_{UA}$:

$$\begin{aligned} n &= 12313925725012632907277372667524325758055072806551025035505 01\\ &\phantom{=}\ 050409610693795712355565952260465542148275107424967999701958811 06\\ &\phantom{=}\ 280437959158707238296876219608776490854355708792879135769996451 40\\ &\phantom{=}\ 492163113013410420709005575154950464186037546035533623561490677 49\\ &\phantom{=}\ 888284684215361787293806414889396202707821452383463 67L\\ e &= 65537 \end{aligned}$$

and the ciphertext:

$$\begin{aligned} c &= 579654303362576360840355738165819750714799492255\\ &\phantom{=}\ 5475032249008245888336713874585037805559949385687100525344312\\ &\phantom{=}\ 7798571076614963471843544300372785467082456903912582920032952\\ &\phantom{=}\ 0758983756800872241184575378396329063230847828100301360222673\\ &\phantom{=}\ 8631730848965119239243053133548975221346897531038080236731282\\ &\phantom{=}\ 5891307058273972L \end{aligned}$$

then $m$ can be computed by

```
def solve(e,n,m):
    for i in range(0,100):
        if pow(i,e,n)==m:
            return i
        return None

print solve(e,n,c)
```

with `pow(a,b,c)` returns $a^b \bmod c$.

## 4.3 Correctness of RSA

**Theorem 4.3.1 (Invertible Property for RSA)**
*Assuming encryption and decryption with*

$$\begin{aligned} E_{k_U}(m) &= m^e \bmod n \qquad (4.8) \\ D_{k_R}(c) &= c^d \bmod n \qquad (4.9) \end{aligned}$$

*then the property for RSA to be invertible is*

$$m^{ed-1} \equiv 1 \bmod n \qquad (4.10)$$

**Proof:**
To get the message after encryption back:

$$\begin{aligned} D_{k_R}\left(E_{k_U}(m)\right) = \left(m^e \bmod n\right)^d \bmod n \quad &\Leftrightarrow \quad m \equiv m^{ed} \bmod n \\ &\Leftrightarrow \quad 1 \equiv m^{ed-1} \bmod n \end{aligned}$$

Therefore the goal is to select values for $e, d, n$ that satisfy:

$$\forall m \in \mathbb{Z} : m^{ed-1} \equiv 1 \bmod n$$

□

Note that (4.8) and (4.9) work in both directions. For signatures we want invertability where we do decryption first and then encryption thats equal to:

$$E_{k_U}\left(D_{k_R}(c)\right) = c^{de} \bmod n \equiv c \bmod n$$

So we have the correctness in both directions.

## 4.4 Euler's Theorem

**Definition** *Totien function, Euler's Function*
The *totiend function (Euler's function)* $\varphi$ of an interger $n \in \mathbb{N}$ returns the number of positive integers less than $n$ that are relatively prime to $n$
**Example** 51:
It is

$$\varphi(15) = 8$$

because $15 = 3 \cdot 5$ so every multiple of 3 and 5 are not in $\varphi(15)$ so

$$\varphi(15) = |\{1, 2, 4, 7, 8, 11, 13, 14\}| = 8$$

**Theorem 4.4.1:**
*It is*

$$n \in \mathbb{P} \Leftrightarrow \varphi(n) = n - 1$$

**Proof:**
If $n$ is prime than all of the positive integers less than $n$ are relatively prime to $n$ □

**Example** 52:
It follows for $n = 277$ a prime number:

$$\varphi(277) = 277 - 1 = 276$$

**Theorem 4.4.2:**
*If $n = p \cdot q$ with $p, q \in \mathbb{P}$ then*

$$\varphi(n) = \varphi(p) \cdot \varphi(q)$$

**Proof:**
It follows

$$\varphi(n) = p \cdot q - 1 - (q-1) - (p-1) = p \cdot q - (p+q) + 1 = (p-1)(q-1) \overset{4.4.1}{=} \varphi(p) \cdot \varphi(q)$$

which is useful for RSA:
If we know the factors of $n$, we have an easy way to compute the value of the totient of $n$ but if we don't know $p$ and $q$ it appears to be hard to compute the number of $\varphi(p \cdot q) = \varphi(n)$. □

**Theorem 4.4.3 (Euler's Theorem)**
*If $gcd(a, n) = 1$ (that means if $a$ and $n$ are relatively prime, $\underline{g}$reatest $\underline{c}$ommon $\underline{d}$ivisor) then*

$$a^{\varphi(n)} \equiv 1 \bmod n \tag{4.11}$$

*where $\varphi(n)$ is the totient of $n$. Then we set*

$$\varphi(n) = ed - 1$$

*then we have the correctness property we need with the assumption that $a, n$ are relatively prime.*

## 4.5 Proving Euler's Theorem

**Theorem 4.5.1 (Fermat's Little Theorem)**
*If $n \in \mathbb{P}$ and $gcd(a, n) = 1$ then*

$$a^{n-1} \equiv 1 \bmod n$$

**Proof:**
It follows

$$\{a \bmod n, 2a \bmod n, \ldots, (n-1) \cdot a \bmod n\} = \{1, 2, \ldots, n-1\}$$

so

$$\begin{aligned}
& a \cdot 2a \cdot 3a \cdot \cdots \cdot (n-1) \cdot a \equiv (n-1)! \bmod n \\
\Leftrightarrow\ & (1 \cdot 2 \cdot \cdots \cdot (n-1)) \cdot a^{n-1} \equiv (n-1)! \bmod n \\
\Leftrightarrow\ & (n-1)! \cdot a^{n-1} \equiv (n-1)! \bmod n \\
\Leftrightarrow\ & a^{n-1} \equiv 1 \bmod n
\end{aligned}$$

□

**Theorem 4.5.2 (Euler's Theorem)**
*If $gcd(a,n) = 1$ for $a, n \in \mathbb{N}$ then*

$$a^{\varphi(n)} \equiv 1 \bmod n$$

**Proof:**
Follows from Fermat's little Theorem 4.5.1. If

- $n \in \mathbb{P} \Rightarrow \varphi(n) = n - 1$. So

  $$a^{n-1} \equiv 1 \bmod n \Leftrightarrow a^{\varphi(n)} \equiv 1 \bmod n$$

- $n \notin \mathbb{P}, \gcd(a,n) = 1 \Rightarrow R = \{x_1, x_2, \dots, x_r\} : \forall i \in \{1, 2, 3, \dots, r\} : \gcd(x_i, n) \neq 1$.
  This means $R$ is the set of numbers which are not relatively prime to $n$ since $n \notin \mathbb{P}$. Multiplying $R$ with mod$n$ leads to:

  $$S = R \bmod n = \{x_1 \bmod n, x_2 \bmod n, \dots, x_r \bmod n\}$$

  and it follows:

  $$|S| = |R| = \varphi(n)$$

  because the set $S$ and $R$ consists of $\{1, 2, \dots, r\}$ elementes and $R$ was defined the same way as $\varphi(n)$.
  Due to $a, n$ are relatively prime:

  $$ax_i \bmod n = ax_j \bmod n \Leftrightarrow x_i = x_j$$

It follows

$$\begin{aligned}\prod(R) = \prod_{i=1}^{r} x_i = \overbrace{x_1 \cdot x_2 \cdot \dots \cdot x_r}^{\varphi(n)} &= \prod(S)\\ &= \prod_{i=1}^{r}(ax_i \bmod n)\\ &= (ax_1 \bmod n) \cdot (ax_2 \bmod n) \cdot \dots \cdot (ax_r \bmod n)\\ &= a^{\varphi(n)} \cdot (x_1 \cdot x_2 \cdot \dots \cdot x_r) \bmod n\end{aligned}$$

and therefore

$$x_1 \cdot x_2 \cdot \dots \cdot x_r = a^{\varphi(n)}(x_1 \cdot x_2 \cdot \dots \cdot x_r) \bmod n \Leftrightarrow 1 \equiv a^{\varphi(n)} \bmod n$$

□

## 4.6 Inversibility of RSA

Following 4.3.1 it is:
**Definition** *Inversibility Property of RSA*
Assume $n = p \cdot q$ a product of 2 primes $p, q \in \mathbb{P}$ then there are the encryption function (4.8) and the decryption function 4.9. The *invertibility property* we need to know:

$$m^{ed-1} \equiv 1 \bmod n \tag{4.12}$$

From Euler we know:

$$a^{\varphi(n)} \equiv 1 \bmod n$$

with $\gcd(a, n) = 1$.
So if we can pick $e$ and $d$ such that

$$e \cdot d - 1 = k \cdot \varphi(n)$$

We can't gurantee that $\gcd(m, n) = 1$ (that $m$ and $n$ are relatively prime). We only know:

$$n = p \cdot q \text{ and } m < n$$

and because it's possible that

$$m = c_1 \cdot p \text{ or } m = c_2 \cdot q$$

which some values $c_1, c_2$ and therefore (if $c_1 < q$ or $c_2 < p$): $m < n$ and it follows that $\gcd(m, n) \neq 1$.
So we can't use Eulers theorem directly - we have to deal with special cases.

## 4.7 Pick and Compute $e$ and $d$

For correctness we need that with $n = p \cdot q$ and $p, q \in \mathbb{P}$:

$$ed - 1 = k\varphi(n) = k\varphi(p \cdot q) = k\varphi(p)\varphi(q) = k(p-1)(q-1)$$

Now $d$ should be selected randomly because the private key $k_R = (d, n)$ includes the secret $d$, the public key $k_U(e, n)$ includes $e$and if we want the private key th be secret and hard to guess then include something that is unpredictable and that will be the value of $d$. Due to $n$ is part of the public key, it doens't provide any security for the private key.
Since $e$ is public it's okay if $e$ is computed in a predictable way and in fact $e$ is often choosend to be a small value.
**Theorem 4.7.1:**
*Since d is relatively prime to $\varphi(n)$, it has a multiplicativer inverse such that:*

$$d \cdot e = 1 \bmod \varphi(n)$$

**Theorem 4.7.2 (Compute $e$)**
*Given d and $\varphi(n)$ we can use extended Euclidian algorithm to compute e.*

**Theorem 4.7.3 (Compute $d$)**
*Given e and n in*

$$de = 1 \bmod \varphi(n)$$

*it's hard to compute d otherwise RSA would be insecure, which is only true if n is big enough and a product of 2 prime numbers.*

## 4.8 Security Property of RSA

An attacker who didn't have access to the private key has a difficult probelm perfoming decryption.

**Definition** *Security Property of RSA*

The *security property of RSA* is given $e$ and $n$ it is hard to find $d$ unless for someone who knows the factors of $n = p \cdot q$ because:

$$d = e^{-1} \bmod \varphi(n) = e^{-1} \bmod \varphi(p \cdot q) = e^{-1} \bmod (\varphi(p) \cdot \varphi(q)) = e^{-1} \bmod ((p-1)(q-1))$$

So the security argument relies on 2 things:

1. Showing that all ways of breaking RSA would allow easy ways to factor $n$.
2. Claim that factoring $n$ is hard, where $n$ is the result of multiplying two large prime numbers

ad 1 :

We show that's not possible to compute $\varphi(n)$ more easily than factoring $n$ or equivalent:
We show that given $\varphi(n)$ there is an easy way to compute $p$ and $q$.
We know:

$$\varphi(n) = \varphi(p \cdot q) = \varphi(p) \cdot \varphi(q) = (p-1)(q-1) = pq-(p+q)+1 = n-(p+q)+1 \Leftrightarrow p+q = n-\varphi(n)+1$$

The goal is: if we know $\varphi(n)$ we can easily find $p$ and $q$!
First consider:

$$\begin{array}{rcl}
(p-q)^2 & = & p^2 - 2pq + q^2 = p^2 - 2n + q^2 \\
(p+q)^2 & = & p^2 + 2pq + q^2 = p^2 + 2n + q^2 \\
(p-q)^2 - (p+q)^2 & = & p^2 - 2n + q^2 - p^2 - 2n - q^2 = -4n \\
(p-q)^2 & = & (p+q)^2 - 4n \\
 & = & (n - \varphi(n) + 1)^2 - 4n \\
\Leftrightarrow (p-q) & = & \sqrt{(n - \varphi(n) - 1)^n - 4n}
\end{array}$$

putting all together and add the 2 equations:

$$\begin{array}{l}
p + q = n - \varphi(n) - 1 \\
p - q = \sqrt{(n - \varphi(n) + 1)^2 - 4n} \\
\hline
2p = n - \varphi(n) - 1 + \sqrt{(n - \varphi(n) + 1)^2 - 4n}
\end{array}$$

and so

$$p = \frac{n - \varphi(n) - 1 + \sqrt{(n - \varphi(n) + 1)^2 - 4n}}{2}$$

which is easy to compute because $p \in \mathbb{P} \subset \mathbb{N}$ and therefore $\sqrt{(n - \varphi(n) + 1)^2 - 4n} \in \mathbb{N}$.
It follows:

$$\varphi(n) = (p-1)(q-1) \Leftrightarrow \frac{\varphi(n)}{p-1} = q - 1 \Leftrightarrow q = \frac{\varphi(n)}{p-1} + 1$$

ad 2 :

If factoring is hard, breaking RSA would be hard and factoring is indeed hard.

.

**Theorem 4.8.1:**
*There isn't an easier way to compute $d$ than finding the factors of $n$.*

**Proof:**
This follows by:

$$ed = 1 \bmod \varphi(n)$$

if we know $\varphi(n)$, we can easily find the factors $p$ and $q$ because the correctness of RSA depends on this property.
That means there is some $k \in \mathbb{N}$ such that

$$k \cdot \varphi(n) = ed - 1$$

We already know the value of $e$. Now if finding out the value of $d$, we know a multiple of the totient of $n$.
Once we know a multiple of the totient it's easy to find the factors $p$ and $q$.
So if there is some easier way to find $d$ than factoring the modulus $n$ would provide an easy way to factor. That shows that all the obvious mathematical ways to break RSA are equivalent to factor $n$. □

**Example** 53:
Given the public key $n$ and $e$:

$$\begin{aligned} n &= 114381625757888867669235779976146612010218296721242362562561842935706935245733897830597123563958705058989075147599290026879543541 \\ e &= 9007 \end{aligned}$$

and the intercepted ciphertext

$$c = 96869613754622061477140922254355882905759991124574319874695120930816298225474$$

Figure out the message can be done by factoring $n$. The time to factor $n$ is about 17 years and is:

$$\begin{aligned} n = p \cdot q &= 3490529510847650949147849619903898133417764638493387843990820577 \\ &\cdot 32769132993266709549961988190834461413177642967992942539798288533 \end{aligned}$$

So RSA keys must be longer than 129 digits.
**Example** 54:
Suppose for the public key: $e = 79, n = 3737$, the private key: $d = 319, n = 3737$ and the intercepted ciphertext $c = 903$. Then the plaintext is $m = 387$ because

$$\begin{aligned} E &: m^e \bmod n \\ D &: c^d \bmod n \end{aligned}$$

so

$$903^{319} \bmod 3737 = 387$$

To check the answer simply encrypt 387:

$$387^{79} \bmod 3737 = 903$$

**Example** 55:
Generating public key $k_U = (e, n)$ and private key $k_R = (d, n)$:

1. Pick 2 random prime numbers $p, q$
$$p = 11, q = 13$$

2. Compute modulus $n = p \cdot q$
$$n = 11 \cdot 13 = 143$$

3. Compute $\varphi(n)$
$$\varphi(n) = \varphi(p \cdot q) = \varphi(p) \cdot \varphi(q) = (p-1) \cdot (q-1) = 10 \cdot 12 = 120$$

4. Choose $e$ considering $gcd(n, e)$ with $1 < e < \varphi(n)$
$$e = 23 \Rightarrow k_U = (23, 143)$$

5. Compute multiplicative inverse (using extended euclidian algorithm) of $e$ (which is $d$) using $e \cdot d \equiv 1 \bmod \varphi(n)$. It follows : $e \cdot d + k\varphi(n) = 1 = gcd(e, \varphi(n))$
$$23 \cdot d + k \cdot 120 = 1 = gcd(23, 120) \Rightarrow d = 47, k = -9 \Rightarrow k_R(47, 143)$$

$p, q, \varphi(n)$ are no longer needed but these values can be easily recomputed using $e, d, n$.
Encrypting a message:

1. Encrypting using
$$m = 7, k_U = (23, 143)$$

2. Using $c \equiv m^e \bmod n$ considering $m < n$
$$7^{23} \bmod 143 \equiv 2 \Rightarrow c = 2$$

Decrypting a message:

1. Given
$$c = 2, k_R = (47, 143)$$

2. Using $m \equiv c^d \bmod n$
$$2^{47} \bmod 143 \equiv 7 \Rightarrow m = 7$$

.

## 4.9 Difficulty of Factoring

The largest RSA public key broke so far was RSA-786 (bits) which are equivalent to 232 decimal digits.
So if we want to know that RSA is secure we need to understand how the costs of factoring depends on the size of the numbers that we need to factor.
Desired: if we pick a large enough key even an adversary with large amount of computational power still won't be able to factor the number and break the RSA.

### 4.9.1 Best Known Algorithms

Measure the size (number of bits) of the input $b$ in

$$b = \log_2 n$$

with the RSA modulus $n$.

- Brute Force Algorithm:
  A brute force search would look in `Python` as follows:

  ```
  for i in range(2,sqrt(n)):
      if is_factor(i,n):
          return i
  ```

  assuming `is_factor` (finding gcd) in a constant time. Then we need to go through this loop $\sqrt{n}$ times. So the running time will be linear in $\sqrt{n}$ but $b = \log_2 n$ so the running time will be $\mathcal{O}(2^{\frac{b}{2}})$ which will not work for large $b$

- General Number Field Sieve (for classic computers):
  The general number field sieve is faster than the brute force but still requires all possibles. The runningtime is exponential with $b^{\frac{1}{3}} \log b^{\frac{2}{3}}$ which is still much worse than being polynomial.

- Shor's Algorithm (for quantum computers):
  Shor's Algorithm has a polynomial running time $\mathcal{O}(b^3)$ in the number of bits.
  So factoring is in the complexity class *BQP: bounded error quantum but polynomial time.*
  It's unknown wheather factoring is in the complexity class *NP-hard*, which are the hardest problems that can be solved by a non deterministic Truring-Machine in polynomial time.

**Theorem 4.9.1:**
*If it is proven that factoring is NP-hard then*

$$NP \subset BQP$$

## 4.10 Public Key Cryptographic Standard (PKCS#1), Insecurity of RSA in Practice

The security property of RSA assumed a large, random number $m$.
**Example** 56:
Suppose $n$ =RSA-1024, $e$ and the ciphertext $c$ is:

$$\begin{aligned}
n &= 135066410865995223349603216278805969938881475605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563 \\
e &= 17 \\
c &= 232630513987207
\end{aligned}$$

Then

$$c = m^e \bmod n$$

and if $m^e < n$ then we never wrapped around the modulus $n$.
That means decryption is as easy as finding

$$\sqrt[e]{c}$$

and so

$$\sqrt[17]{232630513987207} = 7 = m$$

**Example** 57:
Suppose we wnat to send a small number like the day of the year:

$$m \in \{1, 2, \ldots, 365\}$$

using RSA. To avoid message guessing we add some random padding to make $m$ large and unpredictable.
**Definition** *Public Key Cryptographic Standard (PKCS#1)*
The *PKCS#1* adds some padding to make $m$ long enough and unpredictable and avoid an attacker from message guessing.
The new message $m'$ is:

$$m' = 00\ldots010||r||00000000||m$$

where $r$ are some random bits with $|r| \geq 64$ depending on the length of $m$ it may use more bits. This prevents the small message space attack since even if the set of possible messages is fairly small an attacker needs to try all possible choices for the random bits (at least 64 of them) in order to test those messges.
A better way to do this:
**Definition** *Optimal Asymmetric Encryption Padding (OAEP)*
The idea of *OAEP* is to *xor* the message with the output of the cryptographic hash function that takes in a random value but the recipient can still decrypt the message because they can obtain the random value and *xor* out the result of the cryptographic hash.

## 4.11 Using RSA to Sign a Document

The straight forward way may look as follows:

$$\begin{tikzcd}
 & A & B & \\
m \arrow[r] & D \arrow[r, squiggly, "c"] & E \arrow[r] & m \\
 & k_{RA} \arrow[u] & k_{UA} \arrow[u] &
\end{tikzcd}$$

$A$ decrypts a message $m$ using his private key $k_{RA}$. That produces the ciphertext $c$ which is really the signed document. Anyone who has $A$'s the public key $k_{UA}$ (include $B$) can now use the encryption usint the public key on that signed document and obtain the document and verified because this document was decrypted using $A$'s private key that only $A$ has created.
**Example** 58:

Suppose $A$ has a public $k_{UA}$ and a private $k_{RA}$ key and $B$ has a public $k_{UB}$ and private $k_{RB}$ key. $A$ wants to send a message to $B$ in a way that protects it from eavesdroppers and proofs to $B$ that the message was generated by $A$ and indented to $B$, so $A$ should send to $B$:

- $E_{k_{UB}}(E_{k_{RA}}(m))$:
  $B$ can use his private key to decrypt the entire message and then use $A$'s public key to get the message and verify it came from $A$.

- $E_{k_{RA}}(E_{k_{UB}}(m))$:
  Every eavesdropper can reverse the first encryption using $A$'s public key but cannot decrypt the full message without $B$'s private key.

.

### 4.11.1 Problem with RSA

RSA is very expensive.
Don't use it for large documents. It costs about 1000 times as much computing power to do 1 RSA encryption as it does to do symmetric encryption. So we don't want to encrypt the whole document like this.
To avoid lots of computational costs, we assume $A$ got $k_{UA}, k_{RA}, RSA$ and a cryptographic hash function $H$ then

$$m' = \langle m, RSA_{k_{RA}}(H(m)) \rangle$$

So we send the document in cleartext (enough for signature) but we send along with it something that proves it's the document that $A$ indented. The output of the hash function $H(m)$ is a small fixed value and we can encrypt that with $RSA_{k_{RA}}$ much more cheaper than encrypting the whole document using RSA.

# 5 Cryptographic Protocols

In *cryptographic protocols* we are going to solve problems using

- symmetric ecryption
- cryptographic hash functions
- asymmetric encryption

**Definition** *Cryptographic Protocol*
The main problem in *cryptographic protocols* that we have to solve is how to authenticate between a client and a server. In this course we are going to look at these *cryptographic protocols*:

- *Encrypted Key Exchange Protocol (EKE)*
- *Secure Shell (SSH)*
- *Transport Layer Security (TLS)*,
  which is the most widely used *cryptographic protocol* using `https`

These 3 are used to authenticate a client and a server. This can be in either directions or in both directions. They all involve a mix of asymmetric and symmetric techniques.
**Definition** *Threat Model*
Everytime we talked about cryptographic protocols we have to think what our *threat model* is. If we want to argue that our protocol is secure, then we need to understand the *threat model*, which means knowing the capabilities of the adversary. In order to argue that the protocol is secure, we need to argue that an adversary with only those capabilities will not be able to break the protocol. Therefore we need to assume:

- The adversary has limited computational power.
  That means in general:
  - An attacker who intercepts a message encrypted with some key $k$ is not able to decrypt unless knowing $k$ or some other advantage for decrypting the message.
  - Hashfunctions have the property they should, that means preimage resistant, so an adversary who has the hash of some value $x$, can not figure out was $x$ was, and also have strong collision resistant, so an adversary can't find 2 values that hash the some output.
- An passive attacker can only eavesdrop.
  That means the attacker can only listen to messages on the network, but can't modifying the message and can't inject their own message.
- An active attacker contols the network.
  That means the can modify data and messages, replay messages, attack-in-the-middle (intercepting traffic between 2 parties and replace the messages with own messages)

.
**Example** 59:
For an adversary who controls a router on the Internet the threat model:

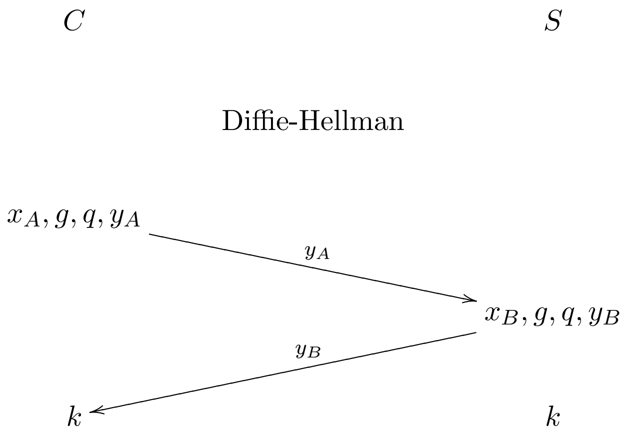
limited computational power and active attacker

would be good, because since the attacker contols the network can modify messages, replay messages, act as a middle attacker. The attacker has lots of things to do, otherwise a passive attacker who can only intercepts messages and analyze intercepted messages.

## 5.1 Encrypted Key Exchange Protocol (EKE)

**Definition** *Ecrypted Key Exchange Protocol (EKE)*
There are many variations on the *encrypted key exchange protocol* and many in which are still used today. The protocol, starting with Diffie-Hellman, looks, for client $C$ and server $S$, as

follows:



This means (after doing Diffie-Hellman):

1. $C$ and $S$ agree on a generator $g$ and modulus $q$
2. $C$ picks a random value $x_A$ and computes $y_A = g^{x_A} \bmod q$ and sends $y_A$ to $S$
3. $S$ picks a random $x_B$ value and computes $y_B = g^{x_B} \bmod q$ and sends the result to $C$
4. $C$ and $S$ can compute the same key $k = g^{x_A x_B} \bmod q$

The problem here is, that an active attacker can change the values of $y_A$ and $y_B$ and finally can act as a middle attacker. The idea of *encrypted key exchange* is to combine this with symmetric encryption to allow the $C$ and $S$ to authenticate each other even if there is a middle attacker. This works as follows:

1. Assume $C$ and $S$ have some secret password $p$
2. $C$ sends $\langle ID, E_p(g^{x_A} \bmod q)\rangle$ to $S$ (the name of $C$ (ID) with the symmetric encrypted value).
3. $S$ can decrypt this with $D_p(E_p(g^{x_A} \bmod q)$ to obtain $g^{x_A} \bmod q$ (which would have been send in the Diffie-Hellman protocol)
4. $S$ can compute a key $k = g^{x_A x_B} \bmod q$ using $S$'s own secret value $x_B$
5. $S$ sends $E_p(g^{x_B} \bmod q)$ to $C$
6. $C$ can decrypt this with $D_p(E_p(g^{x_B} \bmod q))$ to obtain $g^{x_A} \bmod q$
7. $C$ can compute a key $k = g^{x_A x_B} \bmod q$ using $C$'s own secret value $x_A$

This looks like:

$$\begin{tikzcd}
C & S \\
p, x_A, g, q, y_A = g^{x_A} \bmod q \arrow[dr, "\langle \text{id}, E_p(y_A)\rangle"] & \\
& p, x_B, g, q, y_B = g^{x_B} \bmod q \arrow[dl, "E_p(y_B)"'] \\
D_p(E_p(y_B)) & D_p(E_p(y_A)) \\
k = g^{x_A x_B} \bmod q & k = g^{x_A x_B} \bmod q
\end{tikzcd}$$

The drawback of the *encrypted key exchange protocol* for authenticate a user to website is that it requires the server to store passwords in cleartext, which is never a good idea but the *EKE* is <u>not</u> vulnerable to offline dictionary attacks and to in-the-middle attacks.
This protocol, as described, does not provide authentication, because the way to authenticate depens on proving having the same key. To authenticate, the way $C$ needs to prove, is knowing the password and the way $S$ needs to prove, is knowing the password. The assumption is that the password is shared between $C$ and $S$ and only the knowledge of this password proves the authentication. So establish a key does not prove anything.
**Definition** *Encrypted Key Exchange Implementation*
We have to prove, that both parties obtain the same key. For $S$ to obtain the key it needed to be able to decrypt the message from $C$ using the password $p$. So we add to the message that sends $S$ to $C$, instead of just sending $E_p(y_B)$, a challenge, which is some random value $r \in \{0,1\}^n$ of length $n$. So $S$ sends $\langle E_p(y_B), E_k(r)\rangle$. Now $C$ needs to be able to obtain the right key from $E_p(y_B)$, which proves that $C$ knew the password and using that key $C$ can decrypt $E_k(r)$ and obtain $r$. This demonstrates to $C$ that $C$ is talking to the right server $S$, because the server that knew the password and the key $k$ can could only be produced correctly if the server was able to decrypt the message that $C$ send (encrypted with that password). This hasn't proven anything to $S$. To finish the protocol $C$ has to send a response back to $S$ that proves that $C$ was able to obtain $r$. Therefore $C$ sends a message encrypted with the key $k$: $E_k(r||r_A)$ which adds to $r$ another nounce. Now $S$ can decrypt this, check if the $r$ matches and extract $r_A$ and $S$ can send $r_A$ back encrypted with $k$: $E_k(r_A)$.
Finally both, the server and the client have proved knowledge of the password, they established a shard secret (the key $k$), which can be used for further communication and they have done this in the way, that even there is an active attacker, intercepting and modifying all sended messages, if the attacker doesn't know the password $p$ there is no chance to establish a in-the-middle connection or get any information about the messages.

So the former protocol look now as follows:

$$\begin{tikzcd}
C & S \\
p, x_A, g, q, y_A = g^{x_A} \bmod q \arrow[dr, "\langle \text{id}, E_p(y_A)\rangle"] & \\
& p, x_B, g, q, r \in \{0,1\}^n, y_B = g^{x_B} \bmod q \arrow[dl, "\langle E_p(y_B), E_k(r)\rangle"'] \\
D_p(E_p(y_B)) \Rightarrow k = g^{x_A x_B} \bmod q & D_p(E_p(y_A)) \Rightarrow k = g^{x_A x_B} \bmod q \\
r_A \in \{0,1\}^m & \\
r = D_k(E_k(r)) \arrow[dr, "E_k(r||r_A)"] & \\
& r \arrow[uuuu, "\overset{?}{=}"'] \arrow[dl, "E_k(r_A)"'] \\
r_A \overset{?}{=} D_k(E_k(r_A)) &
\end{tikzcd}$$

## 5.2 Secure Shell Protocol (SSH)

**Definition** *Secure Shell Protocol (SSH)*
The *SSH*, based on Diffie-Hellman, uses aspects of symmetric and asymmetric cryptography to solve the problem of client-server authentication. The protocol may look as follows:

$$\begin{tikzcd}
C & S \\
x_C, y_C \arrow[dr, "y_C"'] & k_{US}, k_{RS}, x_S, y_S = \\
& k \\
& H = \arrow[dl, "\langle k_{US}, y_S, \langle H, E_{k_{RS}}(H)\rangle\rangle"'] \\
k, D_{k_{US}}(E_{k_{RS}}(H)) \overset{?}{=} H &
\end{tikzcd}$$

$$H \overset{?}{=} \text{hash}(\text{protocol params}||k_{US}||y_C||y_S||k)$$

Which means:

1. The fist steps are in fact the same to Diffie-Hellman:

2. $C$ picks a large random number $x_C \in \{0,1\}^n$ and sends $y_C = g^{x_C} \bmod p$ to $S$, where $g$ is the generator and $p$ is the modulus just like in Diffie-Hellman.

3. $S$ picks his own large random value $x_S \in \{0,1\}$ and computes $y_S = g^{x_S} \bmod p$ and then $S$ computes the key $k = y_C^{x_S}$.
   So far it's the same as Diffie-Hellman protocol (just changed some names of the variables).

4. $S$ computes a hash $H = \text{hash}(\text{protocol parameters}||k_{US}||y_C||y_S||k)$, so we use some cryptographic hash function. The inputs to that hash function are some protocol parameters, that identify the protocol, concatenated with the public key of the server $k_{US}$, the value of $y_S$ that was send by $C$, that verifies it's part of the same session and prevents replay attacks, because that value was determined by $C$ and finally concatenated by the value of $y_S$, which is the normal Diffie-Hellman response and the key $k$. Note that this is all in a one way hash, so someone who intercepts that hash, won't be able to learn anything about the inputs. Someone who knows the inputs would be able to verify the hash is correct.

5. $S$ sends the value of it's public key $k_{US}$, the value of $y_S$ and the hash signed with $S$'s private key $k_{RS}$.This means sending the hash $H$ along with the hash encrypted with the private key, that what it means to do a signature in asymmetric cryptosystems.

6. $C$ can compute the key $k = y_S^{x_C} \bmod p$

7. $C$ and $S$ have a shared key.

8. $C$ has to check (verify $S$) if

   - $D_{k_{US}}(E_{k_{RS}}(H)) = H$
     which means checking the signature by decrypting using the public key, this verifies that the message was created by someone who knows the private key
   - $H = \text{hash}(\text{protocol params}||k_{US}||y_C||y_S||k)$
     which means recomputing the hash, note that the hash in one-way (we can't use the hash to learn the key but $C$ can compute the key) and then check that the key and the hash matches by computing the hash. Now we know there is no replay attack, because the values $y_C, y_S, k$ are fresh. If there was a replay attack and a different hash value was replayed, then this hash wouldn't match.

.

**Example** 60:
Using $SSH$ where $C$ does not know the value of the public key $k_{US}$ provides no authentication, but establish a shared key, which provides no benefit against an active adversary (attack in the middle) but provides some security against an passive adversary, because the message does get to the right place.

## 5.3 SSH Authentication in Practice

**Example** 61:
using $SSH$ to log into the university of technology in vienna ($*$ stands for private information):

```
>>ssh e*@tuwien.ac.at
The authenticity of host 'tuwien.ac.at (128.130.35.76)' can't be established.
RSA key fingerprint is 70:05:f4:85:ec:f5:3a:59:65:22:f6:4a:35:82:6b:54.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'tuwien.ac.at,128.130.35.76' (RSA) to the list
of known hosts.
>>e0625005@tuwien.ac.at's password: *
/home/>>logout
>>cat ~/.ssh/known_hosts
tuwien.ac.at,128.130.35.76 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA7
+92EdxA6IktvoPVIObaZmK3HjVcjYFoQ30jnMh0khLj1SzFnPoLx1j8M4Ub
Tipp+4DGS9U1fMJG//z09vNSdjjYsrynv2HRFU5AXaWJkNq0qTadmWcHG
tJzw/gC4u9voVoEi1wbPVLHNO0+OFyOlLJl5L6O5aiB1gmlZk+BtL3nYbjk8y
j2vkXZk0ZE1aAqoYOOvc1+Y+1GqBiv0guxZkFCJshCMSUgsFCCJ8tBn90i
LTQ6j8VSuWyS/VPCpH9ztmUupfeBbGaUoFatAl3tHyKxOTzfg6KF6yDufjEH
t7SdYE+9zK3wXgQwQSBpkEuNyH5Jq8uOgT221nel2LrRpw==
>>emacs ~/.ssh/known_hosts #change some value of the public key
>>ssh e*@tuwien.ac.at
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
70:05:f4:85:ec:f5:3a:59:65:22:f6:4a:35:82:6b:54.
Please contact your system administrator.
Add correct host key in /Users/danielwinter/.ssh/known_hosts to get rid of
this message.
Offending key in /Users/danielwinter/.ssh/known_hosts:8
RSA host key for tuwien.ac.at has changed and you have requested strict checking.
Host key verification failed.
```

So first, don't having a public key for that host, so the authenticity of the host can't be established. It shows the fingerprint (a human easy way to see the key, than all the bits). Connecting typing `yes` will store the public key and added to the lists of already known hosts. Now having a secure channel encrypted with some key thats been agreed to using this *SSH* protocol to send the password to the host knowing that it couldn't be intercepted (but this has only value if knowing that the fingerprint is really the public key). Enter the password would lead to log into the server.
Typing `cat ~/.ssh/known_hosts` shows the list of all known hosts. The ip-adress of the host and the public key are stored. Doing again `ssh e*@tuwien.ac.at` will lead directly to the password request, as the public key is now known.
Modifiying the file (simply change some character) and trying to connect again will give a warning message!

## 5.4 Transport Layer Security Protocol (TLS)

**Definition** *Transport Layer Security Protocol (TLS), TLS Handshake Protocol, TLS Record Protocol, Pre-Master Secret, Master Secret*
*TLS* former known as *Secure Socket Layer (SSL)* is a compley protocol for clients (webbrowser) and servers (hosts) to communicate securely, which is one essential thing for e-commerce (allows sending credit-card numbers over the network as well as personal information with some confidence that it only goes to the intended destination).
It constists of 2 main parts:

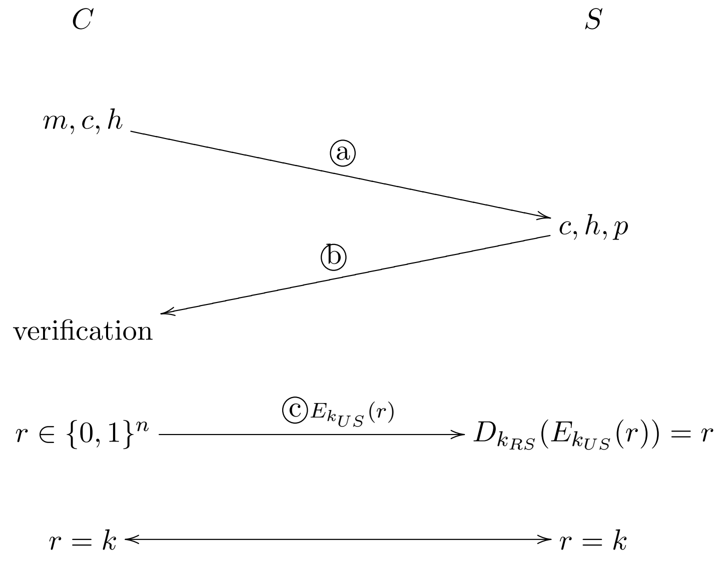
1. *TLS Handshake Protocol*

   - used to authenticate a server to a client (and a client to a server, but this rarely happens, because the client would require a public key and this public key has to be known by the server) using a combination of symmetric and asymmetric cryptography
   - agreement on cryptographic protocol, because *TLS* allows many different encryption algorithms. *TLS* can be implemented on top of many other protocols. The most commonly used implementation is on top of *Hyper Transfer Text Protocol (HTTP)*. Combining *TLS* with *HTTP* results in *Hyper Transfer Text Protocol Secure (HTTPS)*.
   - establish shared session key, a key shared between the server and the client

2. *TLS Record Protocol*

   - starts after sucessfully finished the *TLS Handshake Protocol*
   - enables communication using session key, which was established in the *TLS Handshake Protocol*
   - using symmetric cryptography only, because a session key was established by the end of the *TLS Handshake Protocol* and now using symmetric encryption which is much more faster (cheaper) for encrypting all the content of a page.

A simplified version of *TLS* between a client $C$ and a server $S$ may look as follows ⓐ



Which means:

ⓐ $C$ connect to $S$ sending a message $m$, a list of ciphers $c$ supported by $C$ and a list of hash functions $h$ used by $C$, because different browsers have different ciphers and hash-functions implemented.

ⓑ Due to, $C$ and $S$ have to agree an a cipher and a hash-function, $S$ picks the 'strongest' cipher and hash-function from the received lists and sends the choice back to $C$. $S$ also sends a certificate, which gives the public key $k_{US}$ of $S$ to $C$, in a way, $C$ can trust in. The certificate includes the domain and the public key $k_{US}$ of $S$ and is signed by a *Certificate Authority (CA)*.

ⓒ Next $C$ verifies the certificate, extract $k_{US}$ and picks a random value $r$ and sends $E_{k_{US}}(r)$ back to $S$. $S$ can decrypt $E_{k_{US}}(r)$ using the private key to get $r$, which is used as the shared key $k$.
Now both, $C$ and $S$, have a shared key $k = r$ and can communicate over the channel using symmetric encryption with the key $k$. The protocol to do this is the *TLS Record Protocol.*

The problems of this simplified version of *TLS* are:

(a) An attacker in the middle can hijack ⓒ and replace it with $E_{k_{US}}(r')$ with some $r' \neq r$. Then $S$ has a different key as $C$ and the attacker can now decrypt the messages coming from $S$ but wouldn't be able to get $r$ and figure out the messages of $C$, because the attacker doesn't have a way to decrypt $E_{k_{US}}(r)$ (only $S$ has the private key to decrypt this).

(b) An attacker can force $C$ and $S$ to use a 'weaker' cipher by altering ⓐ. Due to the list of supported ciphers of $C$ is send in plain-text, an attacker can modify the cipher list with only 'weak' ciphers. Now $S$ picks the most powerful cipher from a list of 'weak' ciphers, which is a 'weak' cipher and sends the picked 'weak' cipher back to $C$.

Therefore some changings in the protocol are needed to avoid $(a)$ and $(b)$:

(a) $C$ generates a random value $r_C$ and add this value to ⓐ. $S$ also generates a random value $r_S$ and add this value to ⓑ. The value $r$ also called *pre-master secret* will still be created. Step ⓒ includes some extra information:
$r$ will be padded using *PKCS #1 protocol* (see section 4.10) and add something about the client version. So $C$ sends $E_{k_{US}}(\text{client version}||r)$ to $S$. Instead of making the $k$ just $r$, the key is combining all of the randomness used so far. The key is now called the *Master Secret*:

$$\text{master secret} = H(r, \text{'master secret'} \oplus r_C \oplus r_S)$$

The way to compute the *master secret* is by using a pseudo-random function (like a hash-funcion) with

- the pre-master secret $r$
- a label which is just identifing this as the *master secret*, which gets combinet with
- the random value of $r_C$ (clients randomness) and
- the random value of $r_S$ (servers randomness) as input

The *master secret* (in common cases 384 bits), gets divided into 3 parts (I,II,III) of equal length (128 bits):

(I) will be used as the key $k$ for symmetric encryption (using *RC4* a symmetric encryption algorithm)

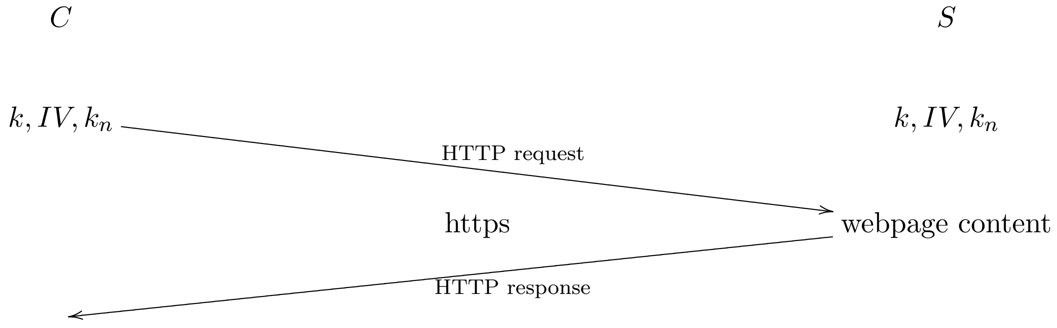
(II) will become the *initialization vector (IV)*, that is needed for $CBC$ mode

(III) will be used for a keyed hash function, which is a hash function that mapping depends on a key $k_n$.

The goal now is to protect the traffic between $S$ and $C$.
This change means, if an active attacker interfere with the message $E_{k_{US}}(\text{client version}||r)$, the attacker still can't control what gets computed here and before the channel is used before anything secure, $C$ and $S$ need to verify that they got the same key.
So to finish the *TLS Handshake Protocol* we added a step:
We encrypt the finish message using the key extracted from the *master secret.* If any of the values: $r$, *master secret*, $r_C$, $r_S$ were modified, the keys will be different for $C$ and $S$ and the *TLS Handshake Protocol* will never be finished and there will never be a secure communication using that key, because the 2 parties have to verify the *TLS Handshake Protocol* before continuing.
Since new random values $r, r_C, r_S$ are used for each protocol execution, there is nothing to replay. Therefore the random values prevents from replay attacks, because the key depends on the values that are used previously.

(b) This can be avoided by also authenticate the messages ⓐ and ⓑ. Not at the beginning, as there is no key established but in later steps.
If an attacker can force $S$ and $C$ to use a 40 bit key, then the attacker can do a brute force attack.

The *TLS Record Protocol* (with the modifications) may look as follows:



$$R = M||HMAC_{k_n}(M)||\text{padding}$$

So first, $C$ requests the content of a webpage. The response is the content of some webpage, which can be quite long, so we need a way to encrypt that response and send it to $C$. We want both:

- confidentiality
- integrity checking

The response is $M$, which includes a mac using the hash-function $H$ of $M$, which uses $k_n$ (the key of the hash-function) and finally we have some padding to fill up the block size.
Now we want to send this whole response over the secure channel.
The way this is done with *TLS Record Protocol* is to use $CBC$ mode 2.4.2 and some encryption function.
In one session there might be multiple responses, so when the next response is done, we don't

want to do the whole *TLS Handshake Protocol* again. In the next response the next message block will be encrypted using *CBC* mode again which produces the cipher block of the next message, but we need an *IV* here (don't use former IV, due to security).

**In *TLS* the last cipherblock of the previous message is used as the IV for the first block of the next message.**

**Example** 62:

With the notation of 2.4.2 suppose an adversary intercepts the first message and has a way to control $m'_0$ (the first input message block of the next message). With the values:

$$\begin{array}{ll} c_3 = 10101010 & c_{n-1} = 11110000 \\ c_4 = 01010101 & m^* = 00000000 \end{array}$$

the adverary can learn if $m_i = m^*$, where $m^*$ is the guess for block $i$.

The adversary can set the value of $m'_0$ and figure out, how that make the server give a particular repsonse and examining the first ciphertext block $c'_0$.

The adversary should use:

$$m'_0 = 01011010$$

Because using *CBC* mode encryption it is

$$c_i = E_k(c_{i-1} \oplus m_i)$$

For the first block of the second message it is

$$c'_0 = E_k(c_{n-1} \oplus m'_0)$$

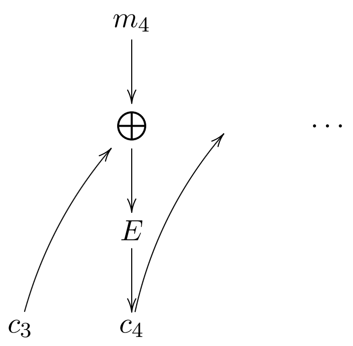
So the danger here is $c_{n-1}$ because the adversary already knows the whole(first) encrypted message and therefore the adversary knows the value of $c_{n-1}$.

That means an adversary can pick a message value for $m'_0$, so that the value of $c'_0$ reveil something. It has to be an input in encryption (assuming the adversary can't break the encryption). The adversary knows the encryption of all the blocks from the previous message and we know, from the way how *CBC* works, if we want to learn the value of $m_4$, we havt to do

$$c_{n-1} \oplus m'_0 = c_3 \oplus m_4 = c_3 \oplus m^*$$

where we don't know $m_4$ (a guess for $m_4$ is $m^*$).

If $c_3 \oplus m^*$ is the input to the encryption function, then we can check if the output matches $c_4$ due to this *CBC* short scheme:



To make this the input, we have the $IV = c_{n-1}$ (the last cipherblock of the previous message is the initialization vector for the next message) and *xor* that out:

$$c_{n-1} \oplus m'_0 = c_3 \oplus m^* \oplus c_{n-1}$$

Here $m^* = 00000000$ can be left out and then

$$c_3 \oplus c_{n-1} = 01011010$$

If 01011010 is used for $m'_0$

$$c'_0 = E_k(c_{n-1} \oplus c_3 \oplus m^* \oplus c_{n-1}) = E_k(c_3 \oplus m^*)$$

because same values cancel out in $xor$ and thats the same as in $CBC$ mode

$$c'_0 = E_k(c_3 \oplus m_4)$$

Knowing $c_3$ we can construct $c_3 \oplus m^* \oplus c_{n-1}$ to pas in the ecryption functio. We don't know what $m_4$ is but we knof if this result is the same then

$$c'_0 = E_k(c_3 \oplus m^*) = c_4$$

and that gives us if $m_4 = m^*$
**Example** 63:
Generalize Example 62:
If $m'_0$ can be picked as

$$m'_0 = c_{n-1} \oplus c_{i-1} \oplus m^*$$

where $c_{i-1}$ is the cipherblock the one before the one we want to get and $m^*$ is the guess. That means what we learn if $c'_0 = c_i$ then we know $m_i = m^*$.
If the block size is small we can guess all of these blocks. Using $RC4$ (symmtric encryption algorithm) we have a 64 bits block size of usin $AES$ we have a 128 bits block size.
The attack is useful if the attacker can guess these 64 bits in a useful way and has a way to control the message which is not a small danger.
**Definition** *Browser Exploit Against SSL/TLS (BEAST)*
One way is if the attacker can figure out many of the bits (say 58 out of 64, 8 bits unknown) than there is an attack called $BEAST$, which uses this cryptographic weakness and injects java script in the page. Then the attacker can control the requests enough to actually use this guessing. Guessing only 8 bits at a time, you expectatly to need 128 guesses for each 8 bits.
If the attacker controls java-script on the page, the victim is using $TLS$ to request, then java-script can make repeated requests to this server. Those are still a part of the encrypted session but the attacker can control what the requests anre and perhaps can design a request that the server will response to in a particular way, giving the attacker control of the first block in the next message.
**Example** 64:
To exploit the $BEAST$ vulnerability in $HTTPS$, suppose the attacker has intercepted the ciphertext, using 4-bit blocks:

$$\begin{tikzcd}
m_0 \arrow[d] & m_1 \arrow[d] & m_2 \arrow[d] & m'_0 \arrow[d] \\
c_0 & c_1 & c_2 & c'_0
\end{tikzcd}$$

1000 1101 0011 ?

where $c_0, c_1, c_2$ are the intercepted cipher-blocks from the message blocks $m_0, m_1, m_2$.
If an attacker wants to find if $m_1$, the second block in the intercepted message is equal to 1010,

then the attacker should use as the target value for $m'_0 = 0001$ (the first block in the next response).
We can test if $m'_0 = 1010$. In order to test this, the value going in the encryption with $m'_0$ is $1010 \oplus c_0$ (a guess of $m_1$ *xored* with $c_0$). It follows

$$1010 \oplus c_0 = m'_0 \oplus c_2 \Leftrightarrow m'_0 = 1010 \oplus c_0 \oplus c_2 = 1010 \oplus 1000 \oplus 0011 = 0001$$

**Theorem 5.4.1:**
*The client can trust it is communicating with the intended server, because the client verifies the certificate using some other key, checks if that matches the server and obtain the server's public key from it.*

**Proof:**
We need some other key, that the client already trust to verify the certificate and then knowing that's the right server and knowing the servers public key □

## 5.5 TLS Information Leaks

Assume the encryption works fine and the *TLS Handshake* protocol is good and all messages between $C$ and $S$ are encrypted but the attacker can listen on this channel (all messages).
Assuming the attacker can't break any encryption but can observe all the messages on this channel then the attacker can learn:

- The approximate size of the requests
- The approximate size of the responses
- The pattern of requests and responses

Due to optimization in $HTTP$ there are often multiple responses to one request, so the pattern is distinguished.

## 5.6 Certificate

**Definition** *Certificate*
A *certificate* can be used to verify that a public key belongs to a server of a network.
The *certificate* has to include something that communicates the public key of the server of the network to the client in a way that the client can trust.
One way to do this is that the server sends the certificate to the client in this form:

$$\text{certificate} = E_{k_{RCA}}(\text{domain}||k_{US})$$

The *certificate* includes the domain of the server as well as the server's public key (the server knows it's own corresponding private key). This message is encrypted using a private key of some *certificate authoritiy (CA)* that the client hat to trust.
To verify the certificate the client has to decrypt the certificate using the public key of $CA$, which is $k_{UCA}$ and checks if the domain matches. If this is the right domain, the client can trust that $k_{US}$ is the right public key of the intended server.

## 5.7 Certificate Details

The details of the certificate of the server `google.com` are

- The *certificate hierarchy*, which gives confident in the certificate. Every entry in the *certificate hierarchy* was signed by the issuer above. At the top of the *certificate hierarchy* we have a self-signed certificate. We trust the top level certificate in the *certificate hierarchy*, because we got it from a 'trusted' source.
- The *issuer*, which indicates who issued the certificate. For `google.com` its:

  ```
  CN=Thawte SCC CA
  O=Thawte Consulting (Pty) Ltd.
  C=ZA
  ```

- The *class of certification.* The higher the class, the more identity checking will be done
- The *expiration time*, is the date the certificate is valid from (`Not Before`) until the date it is valid (`Not After`)
- The *subject*, which gives the name of the owner of the certificate
- The *subject public key info* has information about
  - The *subject public key algorithm*, shows the used algorithm for ecryption. For `google.com` it's `PKCS # 1 with RSA Encryption`
  - The *public key*, showing the modulus and the exponent. For `google.com` its:

    ```
    n:
    128 Byte : A9 50 A4 1D 0F 96 8E 59 07 9F 13 3D 88 77 EC 4F 93 70 1A 5F
    32 DA 7C 90 62 85 63 6D 5B 3C D5 44 BA 36 5A 6E 1B 94 0D EA 6B 1A 72 19
    32 B2 1A C5 C3 FB 5A 66 33 FB 76 79 34 4C 11 AB DF 81 1E 90 0B 75 3D 91
    22 8C 06 52 A7 EE 84 F0 0F 85 83 C1 E4 1C 2F 9C AD B4 98 21 D6 70 30 23
    D5 A4 8E E2 5A 74 F4 4E E3 5E 5A CE 6E F4 51 C5 EB E5 FA F5 80 34 2C
    B7 83 05 1F 0A 6C C5 C3 71 3C 82 FE 6D
    e:
    65537
    ```

  Thats nothing convicing, because any attacker could have generate this. An attacker con generate a certificate that shows all these informations. So we need a signature to verify all these informations

- The *certificate signature algorithm*, is the algorithm used for signing the certificate. For `google.com` its `PKCS #1 SHA-1 with RSA Encryption`, where $PKCS\#1$ is a padding scheme and $SHA-1$ a hash function.
- The *certificate signature value* is, what we need to use to convince ourself that this is a valid certificate.

So the signature is

$$E_{k_{R||\text{issuer}}}(H(\langle\text{certificate content}\rangle))$$

with $E_{k_{R||\text{issuer}}}$ is $PKCS\#1$ using $RSA$ and $H$ is $SHA-1$.
The important part here is that the signature is encrypted with the private key of the issuer.

### 5.8 Signature Validation

In order to trust a certificate, the client needs to validate the signature. To do that, the client needs to know the corresponding public key $k_{U||\text{issuer}}$ that can be used to decrypt the message containing the hash value of the certificate content and can check, that the hash value is the same as computing the hash of the certificate itself.
**Definition** *Public Key Infrastructure (PKI)*
The *PKI* are the ways of distributing public keys. We need a way to securely know the public key of the issuer. Once we know that, we can use the certificate to learn the public key of the website.
**Example** 65:
Suppose *Udacity* would like to add digital signatures to the couse accomplishment certificates that would allow someone who knows $k_{UCA}$, the public key of a certificate authoritiy, to validate that certificate was generated by *Udacity* and earned by the person whose name is in the certificate. Assume $m$ is the certificate string, e.g.: 'Certificate of Accomplishment with Highest Distinction earned by John Doe in CS387.'
Assume $E$ is a strong public-key encryption algorithm, $H$ is a cryptographic hash function and $r$ is a random nonce, then these schemes would allow Udacity to generate unforgable, verifiable, signed certificate.

$$\begin{aligned}\text{cert} &= m||E_{k_{R\text{Udacity}}}(H(m))||E_{k_{RCA}}('\text{Udacity}', k_{U\text{Udacity}})\\ \text{cert} &= m||E_{k_{R\text{Udacity}}}(m \oplus r)||E_{k_{R\text{Udacity}}}(H(r))||E_{k_{RCA}}('\text{Udacity}', k_{U\text{Udacity}})\end{aligned}$$

## 6 Using Cryptography to Solve Problems

This section will deal with

- *Anonymous Communication* using a chain of asymmetric encryption to enable 2 parties to communicate over a network without anyone knowing that they are even talking to each other
- *Voting* with the issue that can it be provided an accurate tally know that each voter is counted without revealing who voted for whom. This will also be done using a chain of asymmetric ecryption but with some added features to ensure the vote tally is correct
- *Digital Cash*, a way to represent and transfer value similar to paper cash. This involves new techniques such as
  - *blind signature*, a cetralized way
  - *Bitcoin* network, a decentralized way that doesn't require any trusted authority but uses *proof of work* to create value

### 6.1 Traffic Analysis

**Definition** *Traffic Analysis*
We use *HTTPS* to do a Handshake first, agree on a secret key and then have a secure channel between a client $C$ and a server $S$.
The messages are going through routers on the internet to reach a distinct destination. There

are maybe many hops between $C$ and $S$ and along these hops will go packets, because using *TLS* every packet consists of 2 parts:

- One part of the packet is the encrypted message and
- Another part of the packet is the routing information. This is necessary so that every router knows the direction to send the message.

Any eavesdropper, who can see one of these messagees can learn that $C$ and $S$ are talking to each other.
This is a form of *Traffic Analysis*, where the important property we want to hide is not the content of the message (which is encrypted and if *HTTPS* works correctly this cannot be understood by the eavesdropper), but what we really want to hide is the fact that $C$ is talking to $S$.
The mere presents of communication of 2 parties is often enough information to cause problems.

## 6.2 Onion Routing

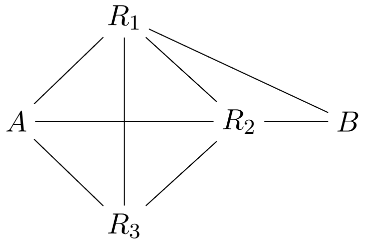
**Definition** *Onion Routing*
Assume 2 parties $A, B$ want to communicate, without anyone being able to know that the 2 parties are communicating. We have a set of routers $\{R_i | i \in \mathbb{N}^\times\} = \{R_1, R_2, \ldots\}$. The $R_i$ are all conected to each other and the 2 parties. So we have a fully connected network assuming we have secure channels between each router and each router with $A$ and $B$.
Now, we select a random router sequence $R_{i_1}, R_{i_2}, \ldots, R_{i_N}$ with $N \in \mathbb{N}^\times$. Then each message in the chain

$$M_{i_k} = E_{k_{UR_{i_k}}}('To : R'_{i_{k+1}}||M_{i_{k+1}}$$

to be the message send to the router $R_{i_k}$ is a message encrypted with the router's public key and it's content explains the next destination as well as the message that should go to that next destination.
Due to we can wrap as many layers as we want this is callen *onion routing.*
The more layers there are, the more hops we go through and the less risk there is that all of the links of the chain can collide or that a party can observe all the communication and leard whats being communicated.
Assume an adversary can listen on the connection from $A$ to $R_1$ (the first router) and from $R_{i_N}$ (the last router) to $B$, then the adversary can learn that $A$ and $B$ are communicating if

- There is no other similar traffic on the whole network
- There is a way the correlate the message from $A$ to $R_1$ and the messge from $R_{i_N}$ to $B$ by e.g. indroducing delays in one(start) message and check if the delay is in the next(last) message.

.
**Example** 66:
For 2 parties and 3 rounters $R_1, R_2, R_3$ the network may look as follows:



where each line is a secure channel.
Assume the random router sequence is $R_2, R_1, R_3$ and the message $m$ then:

- $A$ should send to $R_2$:
$$E_{k_{UR_2}}('To : R_1'||E_{k_{UR_1}}('To : R_3'||E_{k_{UR_3}}('To : B'||E_{k_{UB}}(m))))$$
- $R_2$ should send to $R_1$:
$$E_{k_{UR_1}}('To : R_3'||E_{k_{UR_3}}('To : B'||E_{k_{UB}}(m)))$$
- $R_1$ should send to $R_3$:
$$E_{k_{UR_3}}('To : B'||E_{k_{UB}}(m))$$
- $R_3$ should send to $B$:
$$E_{k_{UB}}(m)$$

.

**Definition** *TOR*
A very sucessful project that project that provides *onion routing* as a service on the internet is called *TOR (torproject).*
It provides a way to connect to a website without revealing to the website where are you connecting from. You need to get a response as well, so that means, in addition to sending a route for reaching the website you need a route for returning (you don't want to include your IP adress which will reveal your location). This project selects routes in both directions. Selecting a random set of routers to reach the website that you want to connect with as well as a way for that website to send a response along another random path.
A client, who wants to reach a website via a random set of routers has to download the public key of all the routers to create a messag that can be send over each hop. This will be done by downloading a list from a trusted directory.

## 6.3 Voting

**Definition** *Permutation*
The notion of permutation is used with several slightly different meanings, all related to the act of permuting (rearranging) objects or values. Informally, a permutation of a set of objects is an arrangement of those objects into a particular order.
**Example** 67:
There are 6 permutations fo the set $\{1, 2, 3\}$, namely:

$$\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}$$

because $|\{1, 2, 3\}| = 3$, therefore 3! different permutations exist.
**Definition** *Voting*
The securitiy properties that a *voting system* should provide are:

- *Anonymity of voters*, that it shouldn't be possible for an adversary to know who somone voted for.
- *Verifiability of count*, which would be easy if each voter be willing to pulic who declared their vote.

- *Coercion resistance* that means, a voter can't prove who they voted for.
- *Reliability*
- *Security*
- *Efficency*

.

**Definition** *Mixed Network (MIXnet)*
The idea (onion routing is based on this) for *MIXnet* is that $n$ voters, who are giving $x_1, x_2, \ldots, x_n$ votes to one of the 3 parties $A, B, C$.

$$\begin{array}{cccccccc} & & A & & B & & C & \\ \left.\begin{array}{r}x_1 \rightarrow\\ x_2 \rightarrow\\ \vdots \rightarrow\\ x_n \rightarrow\end{array}\right\} & f & \left.\begin{array}{c}\rightarrow\\ \rightarrow\\ \vdots\\ \rightarrow\end{array}\right\} & g & \left.\begin{array}{c}\rightarrow\\ \rightarrow\\ \vdots\\ \rightarrow\end{array}\right\} & h & \left.\begin{array}{c}\rightarrow\\ \rightarrow\\ \vdots\\ \rightarrow\end{array}\right\} & \begin{array}{l}\rightarrow v_1\\ \rightarrow v_2\\ \rightarrow \vdots\\ \rightarrow v_n\end{array}\end{array}$$

$B$ collects the votes $x_1, x_2, \ldots, x_n$ from the voters. These are inputs to a random permutation $f$. The permutation randomly scrambles the order of the votes. The position of the votes that came in doesn't match with the position fo the votes that came out. The votes in the new order is passedn along. Now $B$ scrambles also the votes using a random permutation $g$ and those outputs are passed along to $C$ which also does some random permutation using $h$.
In order this to work, we want at the end to know what the actual votes are. The security assumption is that $A, B, C$ wouldn't possible collute. So they can be trusted not to collude with the other parties.
The question is, what should we use for $x_i$ to enable this cain. A 'good' start for the $x_i$ value would be:

$$E_{k_{UA}}(E_{k_{UB}}(E_{k_{UC}}(\text{vote})))$$

The problem using this is that $A, B, C$ and any eavesdropper can learn all the votes, because a vote is from the set of all possible votes $\{'A',' B',' B'\}$. Now everybody who knows the public keys $k_{UA}, k_{UB}, k_{UC}$ of the parties $A, B, C$ can compute the value of $x_i$ for the three possible votes, match thath up to the incoming votes and know exactly what they are - so there is no anonymity to the voters.
To avoid this, the voter neet to add some randomization to the chain.
Adding some random value $r$, selected by the voter and kept secret:

$$E_{k_{UA}}(E_{k_{UB}}(E_{k_{UC}}(\text{vote}||r)))$$

This works only if an eavesdropper doesn't collude with $C$ because $C$ learns all the votes and the random values by encrypting $E_{k_{UC}}(\text{vote}||r)$ and can use that in collaboration with the eavesdropper who heard $E_{k_{UA}}(E_{k_{UB}}(E_{k_{UC}}(\text{vote}||r)))$ to figure out, which voter voted for which party. This solution won't really work. Carring that solution through and add some randomness value to each of this layers:

$$E_{k_{UA}}(E_{k_{UB}}(E_{k_{UC}}(\text{vote}||r_0)||r_1)||r_2)$$

To validate a vote, instead of just publishing the $v_i$ values, it requires that $C$ also published the $r_0$ value. This means the voter can check that the nounce the voter uses is in that list.
In this case $C$ has the most power, because

- $C$ can decide not to include votes
- $C$ could temper with the votes, because the only validation that we have is that a voter can check if their vote is in the list.
- $C$ can add extra votes, if the number of votes in the beginning is unknown.
- $C$ can change or replace some votes and there is no way in this scheme yet, for a voter to prove that $C$ cheated.

To prove a vote, if the tally is published as list $\langle v_k, r_{0_k} \rangle$, then a voter can prove their vote is not included or corrupted by revealing $r_0, r_1, r_2$ and show they produce $x_i$, which requires the *properties of the encryption function*, that it's hard to find $x, y$ with $x \neq y$ such that

$$E(x) = E(y)$$

or even impossible if $E$ is invertible.
This requires that the voter reveals the vote in order to show that the vote was not included by $C$.

## 6.4 Auditing MIXnet

**Definition** *Auditing MIXnet*
The idea here is that each participant in the *MIXnet* can *audit* some of the outputs of the next step. For this we need to provide extra input. Instead of the votes just provides the vote as encrypted to party $A$, the voter provides this to party $B$ as well. So all the incoming votes go to party $A$ and $B$.
$B$ is going to audit $A$ by picking some random set of inputs which are outputs of party $A$ (e.g. $f(x_m) = y_m$). Now $B$ wants $A$ to prove that $y_m$ is a valid output. So $A$ only neet to provide the nounce $v_{m_k}$ (it's helpful to provide the key $k$ as well).
$B$ needs to check that

$$E_{k_{UA}}(y_m || r_{m_k}) = x_k$$

where $B$ knows the value of $x_k$.
This proves to $B$, for the particular output $B$ asks for, that corresponds to one of the inputs to $A$. Now $A$ will privide it's output to $B$ and $C$ and $C$ will be able to verify that $B$ do the mix correctly by picking some random input and having $B$ proving that they are correct and in the final stage $B$ will provide it's output to $C$ as well as to some validator who can validate that $C$ does correct thing.
The number of outputs to be audited depends on teh tradeoff between voters anonymity and chatching cheating. Auditing all the outputs would reveal exactly what the permuatation was.
**Example** 68:
Suppose there are 100 votes and 20 are audited. The probability a mixer can cheat on 4 votes

without getting chaught is 0.4%. This follows from:

$$\begin{aligned}\frac{\text{number of ways to choose without picking cheated vote}}{\text{number of ways to choose}} &= \frac{\binom{96}{20}}{\binom{100}{20}} \\ &= \frac{\frac{96!}{20!(96-20)!}}{\frac{100!}{20!(100-20)!}} \\ &= \frac{96 \cdot 95 \cdot \dots \cdot 77}{100 \cdot 99 \cdot \dots \cdot 81} \\ &= 0.40\end{aligned}$$

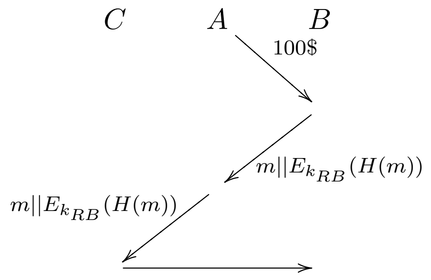
**Example** 69:
For a *MIXnet* with 3 mixers, each auditing 20 out of 100 votes, the probability an eavesdropper could determine which voter cast a given output vote is 0.008%.
At each step there is a 20% probability of seeing that particular vote at both stages and knowing how that was mixed. To chase a vote all the way back we need to see that same vote each time therefore $\left(\frac{1}{5}\right)^3 = \frac{1}{125} = 0.008$.
That means there is a tradeoff here. If we do more auditing (increase from 20% to a higher value) that would increase the probability that a given voter could be identified. It would also decrease the probability thath the *MIXnet* could cheat without getting caught.
So we have a tradeoff between the privacy of the voters and the likelyhood of the detecting cheating.

## 6.5 Digital Cash

**Definition** *Digital Cash*
We want to find some way to use numbers to assimilate something that would be similar (maybe better/worse) than physical cash.
The properties of physical cash are:

- *Universally recognized value*, that means everyone agrees that it's worth something
- *Easy to transfer* between 2 individuals. That means transfer cash without going to a bank or some other trusted third party.
- *Anonymous and untraceable.* Not all want this, e.g. governments.
- *Light and protable*, that means easy to carry.
- *Hard to copy or forge.* If someone can counterfeit the money, it wouldn't have universally recognized value for long.

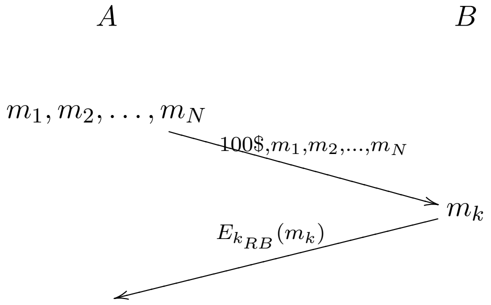
*Digital cash* may work as follows:



$A$ would got to *Indivisible Prime Bank* $B$ which everyone knows is trustworthy and gives 100$ to $B$. Then $B$ writes a message $m$ ='I.P. Bank owes the bearer 100$' and $B$ will send $A$ a signed vrsion of that message: a message along with $B$'s signature using $B$'s private key (using a hash of that message). So the signature proves that it's a valid IOU from $B$ and now $A$ has something representing currency. $A$ gives the currency to $C$ (e.g. buys something at $C$'s shop) and $C$ can take the note (signed message from $B$) from $A$ and give it to $B$ and ask $B$ to deposit into $C$'s account.
Assuming erveryone trusts the bank $B$ and knows it's public key the *digital cash* is

- easy to transfer by just sending the bits to another person.
- anonymous and untraceable, it's the case if alle IOU the bank creates are the same.
- light and portable
- not hard to copy or forge (in fact trivial), because $A$ can send the same bits multiple times and noone knows which ones are valid and which ones are copies. Once we have lost this propertiy we have:
- not universally recognized value

The solution to this are blind signatures.
**Definition** *Blind Signature, cut-and-choose*
This technique gives us a way to associate a unique ID with the bill to be able to detect double spending but doesn't allow the bank to associate the unique ID on the bill with the person who aquires that bill.
The idea is:



This means:

1. $A$ will deposit a bill at the bank $B$ and along with the bill $A$ generates a large number of messages
$$m_i =' \text{ Bill } \#r_{A_i}.B \text{ owes the bearer } 100\$'$$
where $r_{A_i}$ is some unique ID generated by $A$.

2. $B$ uses the *cut-and-choose*:
   $B$ will randomly pick one of the messages $m_k$ and checks all other messages $m_1, m_2, \ldots, m_{k-1}, m_{k+1}, \ldots, m$ that they are valid (correct value). If they all valid, then without looking at message $m_k$ then $B$ will be blindfolded and sign $m_k$.

The point of this is that $A$ generates all the messages, transfer them to $B$, but $B$ doesn't see them until $B$ randomly picks one.
The probability of $A$ being able to cheat without getting caught is $\frac{1}{N}$.
To improve this scheme we are using RSA Blind Signature.

## 6.6 RSA Blind Signature

**Definition** *RSA Blind Signature*
Using *RSA Blind Signature* to blind the message from the signer. The protocol looks as follows:

$$\begin{tikzcd}
A & B \\
k_{UB}, m, k, t \arrow[dr, "t"] & \\
 & t^{d_B} \bmod n_B \arrow[dl, "t^{d_B} \bmod n_B"'] \\
m^{d_B} \bmod n_B &
\end{tikzcd}$$

This means:

1. $A$ wants $B$ to sign a message.

2. $A$ knows $B$'s public key $k_{UB} = (e_B, n_B)$ (the key is a *RSA* key pair with the exponentn $e_B$ and the modulus $n_B$). $m$ is the message $A$ wants $B$ to sign. $A$ also picks a random value $k \in \mathbb{Z}_{n_B}$.

3. $A$ will compute $t = mk^{e_B} \bmod n_B$. If $k \in \mathbb{Z}_{n_B}$ and $\gcd(k, n_B) = 1$ (that menas $k$ is relatively prime to $n_B$) that would make $k^{e_B} \bmod n_B$ a permutation of the values in $\mathbb{Z}_{n_B}$. It follows $k^{e_B} \bmod n_B$ is random and therefore $m = k^{e_B} \bmod n_B$ is random and so $t = mk^{e_B} \bmod n_B$ is random, which doesn't reveal the value of $m$ to $B$, so $A$ can safely send $t$ to $B$.

4. $B$ will sign that message using $B$'s private key $d_B$ (private exponent). That produces the value $t^{d_B} \bmod n_B$ which $B$ sends back to $A$.
   Now using $t = mk^{e_B} \bmod n_B$ $A$ can compute:

$$\begin{aligned}
t^{d_B} \bmod n_B = (mk^{e_B} \bmod n_B)^{d_B} \bmod n_B &= (mk^{e_B})^{d_B} \bmod n_B \\
&= m^{d_B} k^{e_B d_B} \bmod n_B \\
&= m^{d_B} k \bmod n_B
\end{aligned}$$

   knowing that $e_B d_B = 1$.

5. $A$ can divide $k$ out and gets

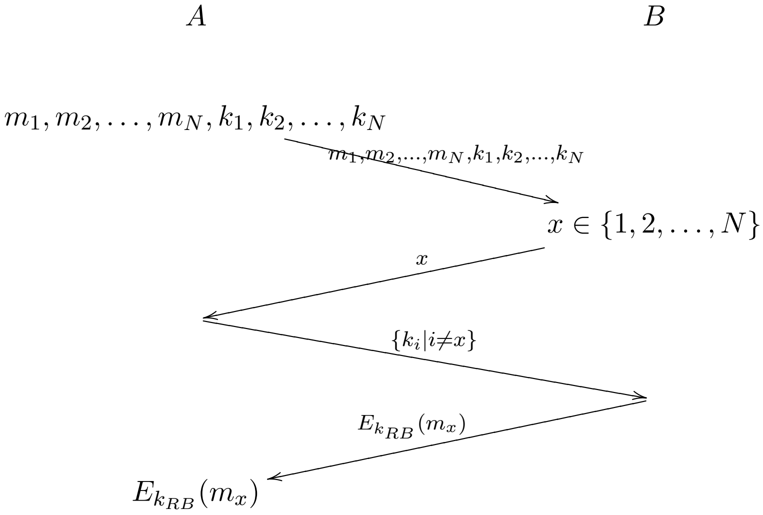
$$m^{d_B} \bmod n_B$$

   That is the message $m$ signed by $B$.
   That means we need to be careful when outputting *RSA decryption* (using private key) - forge a *RSA* signature by multiplying 2 signatures. In this case, the message that's being signed might be use to produce other messages.

.

## 6.7 Blind Signature Protocol

**Definition** *Blind Signature Protocol*
The *blind signature protocol* works as follows:


$A$
$B$
$m_1, m_2, \ldots, m_N, k_1, k_2, \ldots, k_N$
$m_1,m_2,\ldots,m_N,k_1,k_2,\ldots,k_N$
$x \in \{1, 2, \ldots, N\}$
$x$
$\{k_i|i\neq x\}$
$E_{k_{RB}}(m_x)$
$E_{k_{RB}}(m_x)$

1. $A$ generates $N$ messages each one of the form

$$m_i =' \text{Bill } \#r_i.B100\$'$$

   but with different values for $r_i$ and $A$ will also pick $N$ $k$ values and send all $m$ and $k$ values to $B$.

2. $B$ picks some random value $x \in \{1, 2, \ldots, N\}$ and unblinds all the other messages. That means $B$ sends $x$ to $A$.

3. $A$ sends $\{k_i|i \neq x\}$ back to $B$ that $B$ can use to verify that all of the messages other than message $m_x$ are valid.

4. $B$ can decrypt all the messages using $B$'s private key and then divide out $k^{e_B}$ fromt the result.

5. At the end of the *blind signature protocol* $A$ will be able to compute $E_{k_{RB}}(m_x)$ signed by $B$.

6. $A$ can spend the bill, give it e.g. to $C$ and $C$ can verify the signature and deposit the bill at $B$. $B$ checks that this ID has not been spend before and beliefe that it's a valid bill.

7. If $A$ spends the bill again to e.g. $D$ and $D$ deposit it at $B$, $B$ checks the signature but finds that the $r_x$ value was previously used. So $B$ knows, the bill was double spend. The problem is that $B$ doesn't know who double spends the bill.

.

## 6.8 Deanonymizing Double Spenders

**Definition** *Deanonymizing Double Spenders*
The key to *deanonymizing double spenders* is

- Cash is anonymous if spend <u>once</u>
- Identitiy of spender is revealed if cash is spend <u>twice</u>

To do this we need a *One-time pad*, where the key property is that we can split a message into 2 pieces and *xor* them to get the message back.
In the blind signature scheme $A$ creates $N$ messages like

$$m_i =' \text{Bill } \#r_i.\text{Identity: } (I_1, I_2, \ldots, I_m).B100\$'$$

with an identity list, where

$$I_k = H(I_k^0)||H(I_k^1)$$

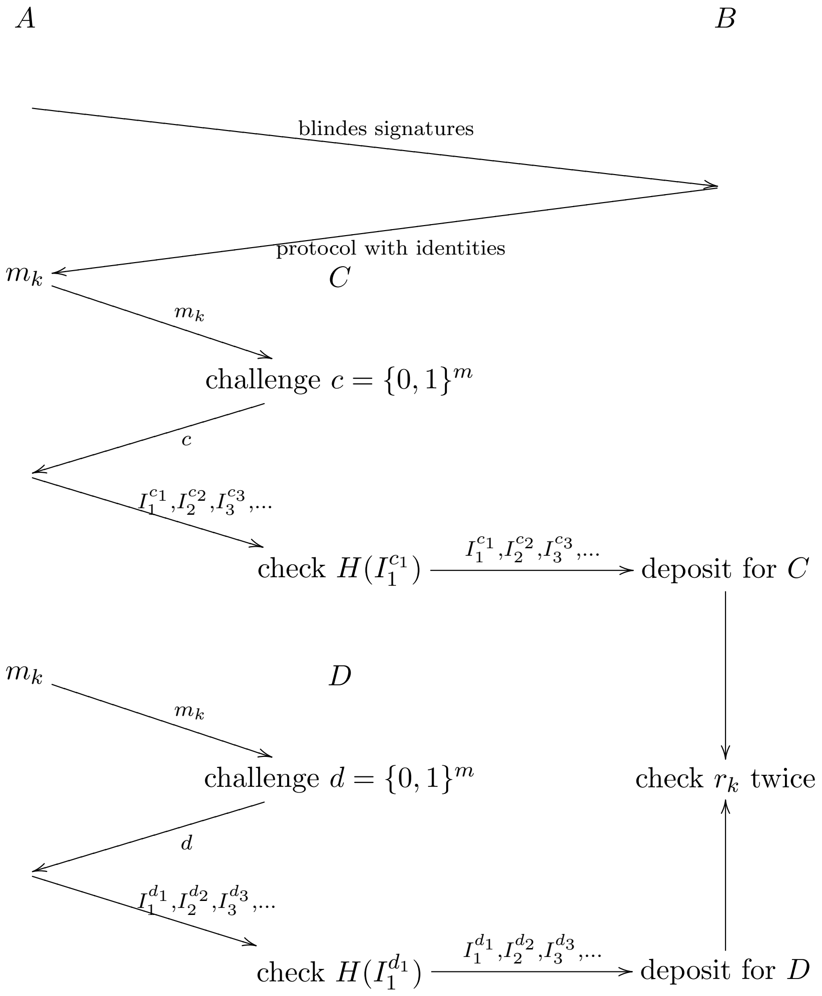
$I_k$ is the concatenation of 2 hash values. $H$ is a cryptographic hash function and the property of $I_k$ is:

$$I_k^0 \oplus I_k^1 = A\text{'s Identitiy}$$

It's easy to create those $I$ values:
Setting $I_k^0$ to a random bit string and $I_k^1$ to $A$'s identity *xor'd* with $I_k^0$.
$B$ need $\forall k : I_k^0$ to verify checked messages in *cut-and-choose* because $B$ needs to validate the hashes. $B$ knows $A$'s identity so $B$ can compute easily $I_k^1$ by xoring $I_k^0$ with $A$'s identity and check if both hashes are correct which validates that each of this identity components are correct.
Now we have a good was to reveal double spenders.

## 6.9 Identity Challenges - Spending Cash

The protocol looks as follows:


A
B
blindes signatures
protocol with identities
$m_k$
C
$m_k$
challenge $c = \{0,1\}^m$
$c$
$I_1^{c1}, I_2^{c2}, I_3^{c3}, \ldots$
check $H(I_1^{c_1})$
$I_1^{c1}, I_2^{c2}, I_3^{c3}, \ldots$
deposit for $C$
$m_k$
D
$m_k$
challenge $d = \{0,1\}^m$
check $r_k$ twice
$d$
$I_1^{d_1}, I_2^{d_2}, I_3^{d_3}, \ldots$
check $H(I_1^{d_1})$
$I_1^{d_1}, I_2^{d_2}, I_3^{d_3}, \ldots$
deposit for $D$

This protocol means:
At the end of the protocol between $A$ and $B$, $A$ has a signed message (bill) $m_k$, where each one of the identity pairs is one of those pairs of hashes.
To spend a bill $A$ sends $m_k$ to $C$ and $C$ sneds a challenge $c$ back to $A$. The challenge is a list of $m$ random bits e.g. $c = [0, 1, 1, 0, \ldots]$. These will tell $A$ which parts of the identity $A$ needs to open. For each bit $A$ has to validate one part of the hash.
Remeber that

$$I_s = H(I_s^0)||H(I_s^1)$$

with the property

$$I_s^0 \oplus I_s^1 = A\text{'s Identity}$$

if the $s^{\text{th}} = c_s$ challenge bit is 0 then $A$ has to send $I_k^0$, if the $s^{\text{th}} = c_s$ challenge bit is 1 then $A$ has to send $I_s^1$.
Now $C$ can check if $H(I_s^{c_s})$ matches with the identities in $m_k$ for all identities. If all matches $C$ accepts the bill. When $C$ deposits it, $C$ has to send all $I$ values to $B$.
Suppose $A$ tries to spend the bill again. This time $A$ sends the bill $m_k$ to $D$. $D$ will do the same protocol making a challenge $d$, send that challenge to $A$ and gets back the corresponding $I$ values. $D$ checks all the values before accepting the bill and deposit the bill to $B$ by sending

all the received $I$ values.
As long as one of the 2 challenge bits at the same position are different $B$ has both parts of the identity. $B$ knows that the bill was double spended, because $B$ sees the $r_s$ twice and also knows the identity of the person who obtained the bill because $B$ can *xoring* the received $I$ values of $C$ and $D$. If e.g. the $s^{\text{th}}$ bit of each challenge is different $B$ computes:

$$I_s^0 \oplus I_s^1 = A\text{'s Identity}$$

Now $B$ knows who double spends the bill.
**Example** 70:
If $A$ spends a bill twice, with the challenge lenght $m = 10$, then the probability that $A$ is cought is $1 - \left(\frac{1}{2}\right)^{10} = 99.9\%$, because $A$ will not be caught only if all bits of the 2 challengens are the same (in every position). So if $C$ and $D$ took exactyl the same challenge ($c = d$)
The challengees are 10 bits long, so the probability that that would happen is $\left(\frac{1}{2}\right)^{10}$ and the probability of the invert event (that $A$ got caught) is just $1 - \left(\frac{1}{2}\right)^{10} = 99.9\%$.

## 6.10 Bitcoin

**Definition** *Bitcoin*
*Bitcoin* is a way to do digital cash in a completely decentralized way. This means there is no bank and no trusted authority, but everyone who participates in the network is considered a peer and they will have an equal say as to what's valued and what's not. *Bitcoin* combines a lot of ideas from previous protocols. The way avoids needing a centralized bank is to keep every single transaction that ever happen. In order to track transactions we have a chain of signatures, which shows the history of transactions. This works as follows:
Some coin $c$ comes in. For $A$ to transfer $c$, $A$ has to create a message including $c$ as well as including that $A$ transfer it to $B$. $A$ signs that message with $A$'s private key $k_{RA}$.
Then $A$ sends this message to $B$. $B$ can verify the signature by using $A$'s public key.
For $B$ to transfer $c$ to $C$, $B$ will add a transfer message and sign the whole thing with $B$'s private key $k_{RB}$. Now $C$ can make the same:
$C$ takes everything that $B$ sends, add a transfer message to it and sign the whole thing with $C$'s private key $k_{RC}$ and then $C$ can transfer it and so on.
Every link in this chain, as long as they have all the public keys, can verify the entire history of transactions.

$$\begin{tikzcd}
 & A & B & C \\
c \arrow[r] & E_{k_{RA}}('\text{To } B :' \,||c) \arrow[r] & E_{k_{RB}}('\text{To } C :' \,||E_{k_{RA}}('\text{To } B :' \,||c)) \arrow[r] & \cdots
\end{tikzcd}$$

The problem in this protocol is that every peer can spend $c$ as many times as they want.
In order for $c$ to have some value, they have to be scorce.
It has to be the case that

- you can't spend them multiple times
- you can't just create them

.

### 6.10.1 Providing Security

**Definition** *timestamp, proof of work*
Everytime someone received a transaction, they don't just accept it but send it into the peer-to-peer network.
When someone wants to verify a $c$, what they need to do is send it into the network, so every transaction can be verified by all the other members in the network and before the transaction considered valid, we need to know that this $c$ hasn't been already spend in some other way.
There are 2 improtant parts to this:

- All nodes must agree on all transactions, that requires some sort of *timestamp*:
  Nodes are going to receive messages at different times. If $c$ was spend twice, that before one node validates the transaction, we need to ensure that if someone attempt to spend a coin twice, both transactions wouldn't be validated by having different parts fo the network have different views of that history of all transactions. For providing this timestamp, we have to rememver that some of the nodes might be malicious. We have no way to know that all nodes are trusted, because anyone who wants can join the network. We just need to have some honest parties to validate the transactions. But we need to know that the honest parties can't invalidate the history of transactions.

- The key to this is requiring a *proof of work*:
  For each timestep, we are goint to have a new block and we need to know that creating those new blocks requires work. If it requires enough work to increase the timestampf then it's unlikely that a malicious user can increase the timestamp faster than the whole network. To make it hard we need some *proof of work* embedded in the timestamp. A way to do a *proof of work*:

  $$\texttt{Find a value } x \texttt{ such that } H(x) \texttt{ starts with } 0^k$$

  In Order to prove you have done some amount of work you need to find a value $x$ where the hash of $x$ starts with $k$ zeroes. Doing that requires work if $H$ is a good cryptographic hash function. The only way to find such an $x$ is keep guessing and looking at the output.

.

**Example** 71:
The average number of guesses of $x$ needed to find one value that $H(x)$ start with $0^{10}$ is 1024, because if the hash is a good cryptographic hash and it produces a uniform distribution the probability that any bit is a 1 or a 0 is $\frac{1}{2}$ and we need to find an output that starts with 10 zeroes, so $2^{10} = 1024$ guesses are needed on average independend to the length of the output.
The number of trials expected to have a greater than 50% probability of finding one where $H(x)$ start with $0^{10}$ we need to comute:

$$\min_{k \in \mathbb{N}} \left( 1 - \left( \frac{1023}{1024} \right)^k \right) > 0.5 \Rightarrow k = 710$$

the expected number of trails

That means finding a hash value with certain properties is expected to require an amound of work and by adjusting those properties we can increase that amout of work.

### 6.10.2 Finding New Blocks

In order to create a new block, which would validate the next history of transactions it's necessary to find some value $x$ such that:

$$H(H(\text{state}||x)) < \text{target}$$

where $H$ is a $SHA - 256$ hash, the stat is the property of the network and $x$ keeps increasing to find one, that satisfies that property and that provides the timestamp, which allows a new block to be generated. The timestamp uses 2 hash functions to increase the required work.
This is the idea *Bitcoin* uses to generate timestamps is you have to keep finding a new block and a block will validate a set of transactions but to generate a new block you have to find a new timestamp which is this target.
So you have to find a value, where the hash fo the hash of the state concatenaded with that value is less than the target. The value of the target controls how hard it is to find such a value $x$. The way *Bitcoin* currency is designed towards is the value of the target is adjusted in a way that make the expected time to find $x$ about 10 minutes. That's the time for the whole network to find the next value. So the value of the target will keep decreasing (harder to find a lower value than the target) as the computing power of the network increase.
The state does 2 improtant things:

- The state includes information about the previous block, this is how the timestamps form a chain

- The state includes some information it's likely to be unique for each member of the network. This is how *Bitcoin* avoids being the case that all members will find the same value $x$.

**Example** 72:
The current value of the target starts with $0^{34}1011\ldots$.
If you find a value that hashes to a result that starts with $0^{35}\ldots$ or $0^{34}1010\ldots$ (something less than the target) then you will be able to create the next block and earn the value of a new block (currently 50 *Bitcoins*, 1 *Bitcoin* = 1 *USD*) and the rest of the network can verify that by computing the hash of the value you found. If the hashed value is less than the target, that will add that block to the *Bitcoin* network.
To do fast computation most participants use *GPU* (graphics processing unit), because there are algorithms for implementing the hash function more efficently compared to *CPU* (central processing unit)

### 6.10.3 Avoid Double Spending

In the network, at each timestamp, a new block is created that validates all the transactions. At the time the block is created this has to be the longest block-chain.
Someone can try to create an alternate block-chain so if someone wants to spend a coin twice, the double spender has to create a chain that is longer than the longest chain.
When a transaction is validated by the network all the signatures in the coin are checked (using transfer-chain) but to prevent double spending there is also a check of the chain of blocks, where the longest chain is the one that is viewed as correct.
Each peer in the network might see a different view of the block-chain. If they see different views the one that has the longest chain is the one that will be viewed as the most correct view of all the transactions. So every participant of the network is keeping track of all the transactions and the version of all transactions thath people trusted the most is the one with

the longest chain.
If an adversary wants to create a longer chain with a different view of transactions it requires finding correct hash values. If the network power exceeds the power of the adversary then it's likely the network have a longer chain then the adversary can produce.
This avoids the need for a central authority but doesn't provide anonymity (in the traditional way) because each transaction is known to the network.
The ways of providing some anonymity is instead of using your actual name in the transaction you can have different identities for each transaction.
**Example** 73:
Assume

- $H$ is a strong cryptographic hash function that produces 128 output bits from any length input. Computing $H(x)$ takes 1 unit of time
- $E$ indicates $RSA$ encryption. $k_U$ is a known public key, but the corresponding private key is not known. Computing $E(x)$ takes 1000 units of time.
- There are no memory limits, but the task has no access to precomputed values.

Then an order by how much work they prove (from least expected work to the most expected work) is

- Find a value $x$ such that $E_{k_U}(r \oplus x)$ ends with $0^{22}$, because an easy way to calculate $x$ such that $E_{k_U}(r \oplus x)$ ends with $0^{22}$ is to calculate $r \oplus x = 0$ and so $x = 0 \oplus r$ and $E_k(0) = 0$. This requires no hashing and no encryptions.
- Find 2 values $x, y$ such that the last 24 bits of $H(r||x)$ are equal to the first 24 bits of $H(r||y)$. Remebering the birthday paradox and the strong collision resistance. There we showed the amount of work

$$\sqrt{N} = \sqrt{2^{24}} = \sqrt{2^{12}} < 2^{20}$$

- Find a value $x$ such that $H(r||x)$ ends with $0^{20}$, which will take about $2^{20}$ hashes
- Find a value $x$ such that $H(H(r||x))$ ends with $0^{20}$, takes twice as much work as only one hash - so it will take about $2^{21}$ hashes..
- Find a value $x$ such that $E_{k_U}(x) = r$ is expensive, because this is equivalent to breaking $RSA$.

.

# 7 Secure Computation

**Definition** *Secure Computation*
Assume 2 parties $A, B$ have some private information. They want to perfomr some *secure computation* and at the end of that, they learn the result of some function on both on their inputs but they don't learn anything about the other party input. This is not achievable using cryptographic only, because our archieving depends on people, assumptions and the adversary as well as the system that actual runs the protocol.
The idea is that any discrete and fixed sized function cna be turned into a logic gate and if we

can find a way to implement logic gates securely we can implement a whole function this way
**Example** 74:
Thinking of a logic gate as a truth table of the function `AND`:

| $A$ | $B$ | $A \wedge B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

To get the value of $A \wedge B$ we need to know the value of $A$ and $B$.
The goal is to encrypt this circuit in such a way, that we can still evaluate it without actually knowing what the inputs are and without learning what the output is.
But we want to produce an output that we can use for the next circuit. If we can evaluate each gate, by keeping the inputs and outputs encrypted we can evalutate the whole circuit and at the end we can map the final result to a meaningful value.

## 7.1 Enctypted Circuits

The first step to create a encrypted circuit is to replace the inputs with encrypted values for each wire. That means we need some way to represent a 0 and 1 on both wires.
Let's assume $a_0.a_1$ are 0 and 1 on the one wire and $b_0, b_1$ are 0 and 1 on the other wire.
We have to replace the nounces in the table with the encrypted values. So the new truth table is

| | | |
|---|---|---|
| $a_0$ | $b_0$ | $x_0$ |
| $a_0$ | $b_1$ | $x_0$ |
| $a_1$ | $b_0$ | $x_0$ |
| $a_1$ | $b_1$ | $x_1$ |

here are the values in the same order. We want to hide any information so we do some permuation:

| | | |
|---|---|---|
| 0010 | 1100 | 0111 |
| 0010 | 0011 | 1001 |
| 1010 | 0011 | 0111 |
| 1011 | 1100 | 0111 |

where $a_0, a_1 \in \{0010, 1010\}$ and $b_0, b_1 \in \{1100, 0011\}$.
Due to this form of the $x$ values it can easily revealed all the $a$ and $b$ values. To hide this pattern and solve this problem we have the outputs $x$ to be encrypted with different keys. If we encrypt the $x$ values with the same key, then either the evaluator would be able to determine all the outputs because the evaluator knows that key or the couldn't determine any of them, because the evaluator doesn't know that key.
So we need to encrypt the outputs ($x$ values) with different keys.
A circuit evaluator, who wants to decrypt $x_0$ using $a_1, b_0$ can use

$$E_{a_1 \oplus b_0}(x_0) \text{ or } E_{a_1}\left(E_{b_0}(x_0)\right)$$

to ensure that a circuit evaluator can decrypt this output and none of the others, because an evaluator has to know both values $a_1, b_0$ to get $x_0$.
In the garbled table we have outputs encrypted with different keys corresponding to the intputs

that correspond to that output value:

$$\begin{array}{lll} a_0 & b_0 & E_{a_0,b_0}(x_0) \\ a_0 & b_1 & E_{a_0,b_1}(x_0) \\ a_1 & b_0 & E_{a_1,b_0}(x_0) \\ a_1 & b_1 & E_{a_1,b_1}(x_1) \end{array}$$

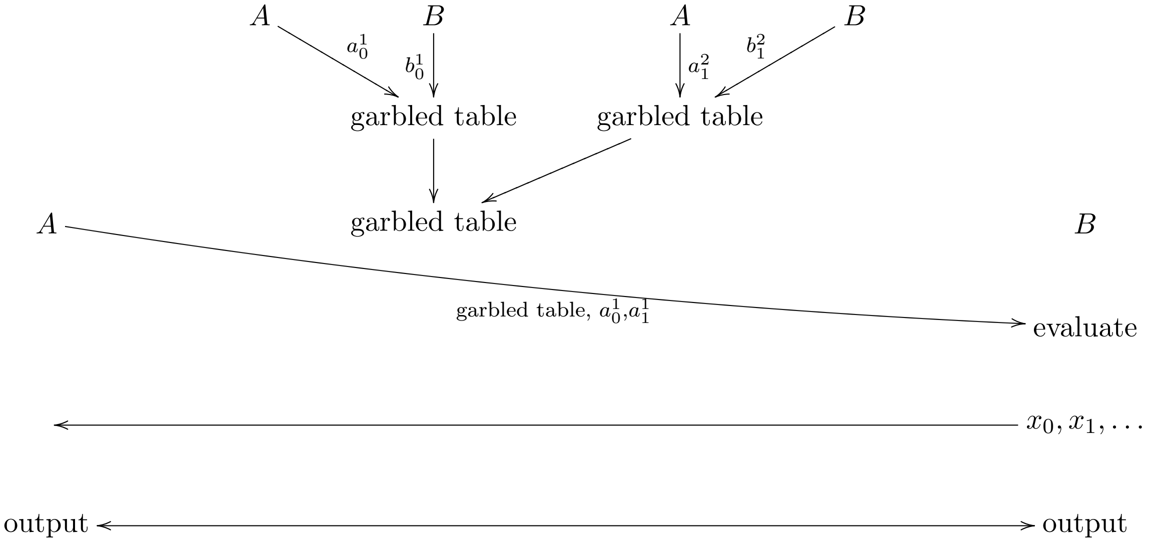
sending the whole table will reveal the values of $a$ and $b$, so we need to remove this part of the table and randomly permute the output values and ass some padding:

$$\begin{array}{l} E_{a_0,b_0}(x_0) \\ E_{a_0,b_1}(x_0) \\ E_{a_1,b_0}(x_0) \\ E_{a_1,b_1}(x_1) \end{array} \Rightarrow \begin{array}{l} E_{a_1,b_0}(\text{pad}||x_0) \\ E_{a_1,b_1}(\text{pad}||x_0) \\ E_{a_0,b_1}(\text{pad}||x_0) \\ E_{a_0,b_0}(\text{pad}||x_1) \end{array}$$

Because each of these values are encrypted with a different key, the evaluator can't tell which one is which. The evaluator is still able to decrypt these to produce the right output.
The evaluator knows the output value of the truth table by trying to decrypt all the entries with input value keys and use the one thath decrypts to pad$||x_{0,1}$

## 7.2 Garbled Circuit Protocol

**Definition** *Garbled Circuit Protocol*
The *Garbled Circuit Protcol* for 2 parties $A$ (generator) and $B$ (evaluator) may look as follows:



This means:

1. In the beginning $A$ and $B$ agreed on some circuit they want to evaluate and it takes inputs from both $A$ and $B$.

2. $A$ generates a garbled table for each logic gate in the circuit and send the garbled circuit to $B$ as well as $A$'s input values, which are random nounces and $B$ can't tell what they mean.

3. $B$ evaluates the circuit using the garbled circuit protocol decrypting one entry from each of these and at the end $B$ gets some output values and then turn that into semantic value. The problem is that $B$ can't obtain his inputs to the circuit, because to evalute the table $B$ needs $A$'s and $B$'s inputs.

.

**Definition** *Oblivious Transfer (1 out of 2 OT)*
The *Oblivious Transfer* means that $A$ can create 2 values $x_0, x_1$ and $B$ will obtain 1 of those values $x_b$ with $b \in \{0, 1\}$. So $B$ learns one of $x_0$ or $x_1$ and $A$ doesn't know which one $B$ obtained:

$$\begin{tikzcd}
A & B \\
x_0, x_1, w_0, w_1, k_{UA} \arrow[dr, "x_0{,}x_1"] & b \\
 & x_b, r, v \arrow[dl, "v"'] \\
k_0, k_1, w'_0, w'_1 \arrow[dr, "w'_0{,}w'_1"] & \\
 & w'_b - r
\end{tikzcd}$$

This means:

1. $A$ has 2 wire labels $w_1, w_2$ which correspond to the inputs to some gate and $A$ wants to transfer one of them to $B$ without revealing the other one. We use $A$'s public key $k_{UA} = (n, e)$ which is know to $B$. The goal is to transfer on of the wire labels to $B$. $A$ creates 2 random values $x_0, x_1$ separated from the wire labels, which are transfered to $B$.

2. $B$ picks some random value $b \in \{0, 1\}$ and picks $x_b$ from the transfered value and $B$ also picks some random value $r$ to blind the response $x_b$, because $B$ cant allow $A$ to learn whether $B$ picks $x_0$ or $x_1$, which would reveal $B$'s input.

3. $B$ computes
$$v = x_b + r^e \bmod n$$
to hide the value of $x_b$ by adding a random value raised to the $e$th power.

4. $B$ sends $v$ to $A$.

5. $A$ performs 2 different $RSA$ decryptions:
$$\begin{aligned} k_0 &= (v - x_0)^d \bmod n \\ k_1 &= (v - x_1)^d \bmod n \end{aligned}$$

6. $A$ sends a message to $B$ that allows $B$ to learn 1 wired label. $A$ adds the keys to the wired label:
$$\begin{aligned} w'_0 &= w_0 + k_0 \\ w'_1 &= w_1 + k_1 \end{aligned}$$
and send $w'_0, w'_1$ to $B$.

7. $B$ computes $w_1$ (if $B$ picked $b = 1$) with $w'_1 - r$ or $w_0$ (if $B$ picked $b = 0$) with $w'_0 - r$

.

**Example** 75:
Assuming $B$ picks $b = 1$ in the $OT$ protocol, then $k_0$ is meaningless but

$$k_1 = r$$

because, considering $ed = 1$:

$$\begin{aligned}
k_1 = (v - x_1)^d \bmod n = ((x_1 + r^e \bmod n) - x_1)^d \bmod n &= (x_1 + r^e \bmod n - x_1)^d \bmod n \\
&= (r^e \bmod n)^d \bmod n \\
&= r^{ed} \bmod n \\
&= r^1 \bmod n = r
\end{aligned}$$

Now $B$ can easily compute $w_1$ with $w_1' = w_1 + k_1 = w_1 + r$ so $w_1 = w_1' - r$
So $B$ obtain his inputs with $OT$. To enable $B$ to learn his inputs to the circuit $A$ sends the garbled circuit along $A$'s inputs. $B$ can evaluate the circuit and then from the encrypted output wire $B$ can obtain the result of the circuit execution and share that with $A$ or flip role and do it again and $A$ would obtain the output.
To actually obtain the output vlaue (the outputs for the garbled table are all encrypted), so at the end of the execution $B$ has a list of encrypted wire labels and $B$ wants to turn that into semantic output.