CS 387
Applied Cryptography

David Evans

written by

Daniel Winter

16.04.2012

Contents

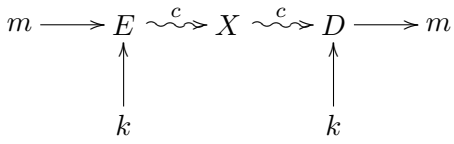
1 Unit 1

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 menas all parties have the same key to do encryption and decryption.
Definition *symmetric Cryptosystem, decryption function, encryption function*
In this paper a *symmetric Cryptosystem* always looks as follows:



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* and D is the *decryption function* with:

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

X is a possible eavesdropper of the insecure channel.
Definition *correctness property*
In order we get the same message after decrypting the ciphertext we need the *correctness property*: $\forall m \in \mathcal{M}, \forall k \in \mathcal{K}$

$$D_k(E_k(m)) = m \Leftrightarrow E_k = D_k^{-1}$$

Example 2:
These functions satisfy the correctness property for a symmetric cryptosystem with $\mathcal{M} = \mathcal{K} = \mathbb{N}$

1. $E_k(m) = m + k, D_k(c) = c - k$
2. $E_k(m) = m, D_k(c) = c$

because

1. $D_k(E_k(m)) = D_k(m + k) = m + k - k = m$
2. $D_k(E_k(m)) = D_k(m) = m$

.

1.2 Kerchoff's Principle, *xor*-function

Theorem 1.2.1 (Kerchoff's Principle)
Even if the encryption function E and decryption function D are public, the message is still secure due to a usage of a key. Only the key has to be private. 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 giben by it's truth table:

A	B	$A \oplus B$
0	0	0
1	0	1
0	1	1
1	1	0

Example 3:
$\forall x, \forall y$:

- $x \oplus y \oplus x = y$
- $x \oplus x = 0$

and these two very important properties of the *xor function* with $\forall x, \forall y, \forall m \in \mathcal{M}, \forall k \in \mathcal{K}, \forall c \in \mathcal{C}$:

- $x \oplus y \oplus x = x \oplus x \oplus y$
- $m \oplus k = c$ and $c \oplus k = m$

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

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

with the value for eacht bit of the ciphertext is defined as: $\forall i \in \{0,1,\dots,n-1\}$

$$c_i = m_i \oplus k_i$$

Example 4:
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:

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

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'.

1.4 Probability

Definition *Probability Space*
The *probability space* Ω is the set of all possible outcomes.
Example 5:

- Flipping a coin: $\Omega = \{(H)eads, (T)ail\}$
- Rolling a dice: $\Omega = \{1, 2, 3, 4, 5, 6\}$

.
Definition *Uniform Distribution*

If the probability space has a *uniform distribution* that menas each outcome has equal probability.

Definition *Probability Function*

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

$$P_i : \Omega \to [0,1] : \omega \mapsto [0,1]$$

where $\omega \in A$ is an outcome. If $P_i(\omega) = 0$ that means that ω never happens (e.g. roll 7 on a dice). If $P(\omega) = 1$ that means that ω alyways happens (e.g. the outcome of a coinflip is in $\{H, T\}$)

Theorem 1.4.1:

Assume the probability space Ω and the probability function p. Then it is

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

Example 6:

Let's assume $\Omega = \{Heads, Tails, Edge\}$ with the probabilities

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

The probability for edge E is given by

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

Definition *Event*

An *Event* $\mathcal{A}$ is a subset of outcomes from a probability space.

Example 7:

An Event of tossing a coin woult be landing on heads, therefore $\mathcal{A} = \{H\}$. A valid coin toss is the set of outcomes $\mathcal{A} = \{H, T\}$

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 8:

The probability for a valid coin toss $\mathcal{A} = \{H, T\}$ is

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

Definition *Conditional Probability*

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

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

Example 9:

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 $P(\mathcal{B}) = P(\{H\}) = 0.49999$:

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

1.5 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

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

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 1.5.1:
The One-Time Pad is a perfect cipher.

Proof: Remember the perfect cipher property:

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

and the definition of the conditional probability:

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

It follows with $A = (m = m^*)$ and $B = (E_k(m) = c)$

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

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 B, wich is the probability that some message encrypts to some key (computet over all the messages).
Then

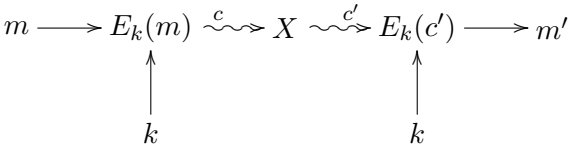
$$P(A \cap B) = P(m = m^* \cap E_k(m) = c) = P(m = m^*) \cdot P(k = k^*) = P(m = m^*) \cdot \frac{1}{|\mathcal{K}|} = \frac{P(m = m^*)}{|\mathcal{K}|}$$

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 $P(k = k^*) = \frac{1}{|\mathcal{K}|}$. Plug 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 the One - Time Pad is a perfect cipher
□ **Definition** *malleable cipher, impractical cipher*

- *malleable*:
  The encryptet message $E_k(m) = c$ can be modified by an active attacker X, that means



- *impractical*:
  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

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

  In general, a cipher is impractical iff (if and only if)

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

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

.
Theorem 1.5.2 (Shannon's Theorem)
Every perfect cipher is impractical.

Proof: Proof by contradiction: Assume to have 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 $P(E_k(m) = c) > 0$. That means there is some key that encrypts some message 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. □

1.6 Lorenz Cipher Machine

Theorem 1.6.1:
Given two cipertexts $m \oplus k = c = c_0 c_1 \dots c_{n-1}, m' \oplus k = c' = c'_0 c'_1 \dots, c'_{n-1}$ *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 $m_0 m_1 m_2 m_3 m_4$, 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 10:
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 $(i+1) \mod 5$. Thus for 5 channels c_1, c_2, c_3, c_4, c_5:

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

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

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

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 1.6.2:
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: 'wheels', 'letters', 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, Δs 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 11:
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 Δz 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 Δz. 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 Δz. 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 1.6.3:
Goal of cipher: Hide statistical properties of message and key (which should be perfectly random).

Goal of cryptoanalyst: 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-

1.7 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:
  our 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 longer 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 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.
In *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 Homework Unit 1

2.1 Conditional Probability

Example 12:
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 $a\}$:

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

.

2.2 Monoalphabetic Substitution Cipher (Toy-Cipher)

Definition *Monoalphabetic Substitution Cipher (Toy-Cipher)*
The *Monoalphabetic Substitution Cipher (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.

Example 13:
One possible mapping is given by

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

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, shift* of each letter.
Theorem 2.2.1:
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

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

Assuming a 26 letter alphabet then the keyspace is

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

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

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

□

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) □

2.3 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
$$x = x_0 x_1 x_2 \dots x_{n-1}$$
2. Generate a random key $k \in \{0,1\}^n$:
$$k = k_0 k_1 k_2 \dots k_{n-1}$$
3. Compute
$$s = k \oplus x$$
with $\forall i \in \{0, 1, 2, \dots, n-1\} : s_i = k_i \oplus x_i$
4. Give s and k to different person and keep x yourself.

The message can only be decryptet by computing $s \oplus k$. So the people with the information s and k may not meet.
Theorem 2.3.1:
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 $\forall i \in \{1, 2, \dots, m-1\} : k_i \in \{0,1\}^n$.
Then for m people p_i with $i \in \{0, 1, \dots, m-1\}$ the information maps as follows:
$$\begin{aligned} x \oplus k_1 \oplus k_2 \oplus \dots \oplus k_{m-1} &\mapsto p_0 \\ k_1 &\mapsto p_1 \\ k_2 &\mapsto p_2 \\ &\vdots \\ k_{m-1} &\mapsto p_{m-1} \end{aligned}$$
so every person p_i with $i \in \{0, 1, 2, \dots, m-1\}$ 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. □

2.4 Challenge Question

waiting for solution - you can help by sending it to inverno@gmx.at
(or i'll try to code it the next week)

3 Unit 2

3.1 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 same 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.

3.2 Generating Random Keys

Definition *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 14:
$k \in \mathcal{K}$ for $\mathcal{K} := \{0,1\}^n$ with some $n \in \mathbb{N}$. If there are no visible patterns (e.g. $100100100\ldots$) and enough repititions (e.g. $100000101111101011111\ldots$)
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 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.
Therefore a short program that can produce the sequence, that means 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 3.2.1:
For a given sequence s it is theoretically impossible to compute $K(s)$ 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 15:
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 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 16:
Assuming a sequence s of lenght n

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

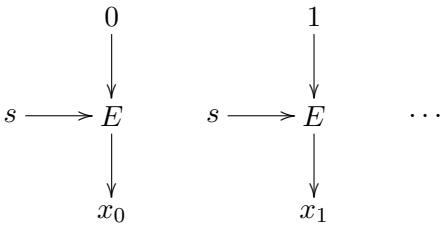
with $x_i \in [0, 2^{n-1}]$.
Even after seeing $x_0, x_1, x_2, \ldots, x_{m-1}$, it's only possible to guess x_m with probability $\frac{1}{2^n}$.
Definition *Physically Random Events*
Physically random events are in

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

.

3.3 Pseudo Random Number Generator (PRNG)

A **Definition** *Pseudo random number generator, seed*
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 17:
Assume extracting a seed s from a *Random Pool* (finite many true random numbers) using s as key. The *PRNG* may look as follows



For the first random output x_0, we get 0 encrypting that with the key s (seed) and so on.

3.4 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.

3.5 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):

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

Thus

$$\begin{aligned} &m = m_0 m_1 m_2 \ldots m_{n-1} \\ &\Downarrow E_k(m_i) = c_i \\ &c = c_0 c_1 c_2 \ldots m_{n-1} \end{aligned}$$

The *electronic codeblock mode* works as follows:
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). We store the cipher text $c = c_0 c_1 c_2 \ldots$ 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.

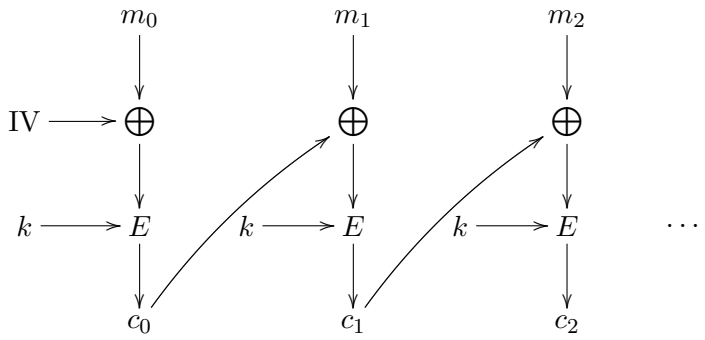
The 2 main problems of *electronic codebook mode* are

- 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.

.

3.6 Cipher Block Chaining Mode (CBC)

Definition *Cipher block chaining mode*
The ideo in *cipher block chaining mode* is using the ciphertext from the previous block to imact 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



Taking each message block, ecrypt it with the encryption function (using the same key for each block) and the outputs are blocks of ciphertext. Instead of doing each block independently (and having the codebook property) for the second block, taking the resulting ciphertext of the first block and *xoring* that with the message block m_1. The fist message block will be *xor'd* with a *initialization vector* (IV) (a random block of size b).
The result of *CBC* is $\forall i \in \{1, 2, 3, \dots, n-1\}$:

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

The IV don't need to be kept secret. Note that the *CBC* sill encrypts the output of $m_0 \oplus \text{IV}$. It's helpful to not to reuse an IV.
Example 18:
Loosing the value of IV but having c and k the message m (excepts m_0 can be recovered with the formula

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

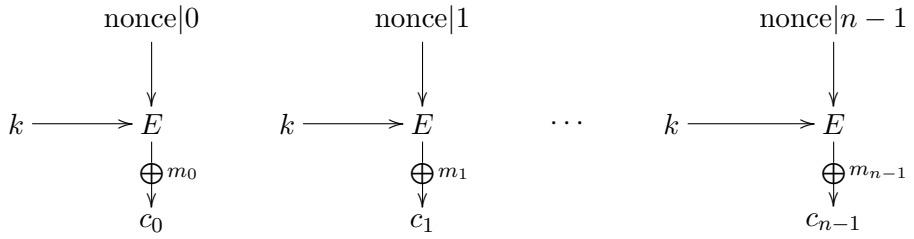
except

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

thus m_0 is lost.

3.7 Counter Mode (CTR)

Definition *Counter Mode (CTR)*
A message m is divided into blocks $m = m_0 m_1 \dots m_{n-1}$. Instead of just having a message block go in the encryption function there is a counter (some value that cycles through the natural numbers) that's the input message so the results are some ciphertext blocks. These blocks *xor'd* with the corresponding messageblock and the result of these 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 *counter mode* may look as follows:



It follows encryption

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

and decryption:

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

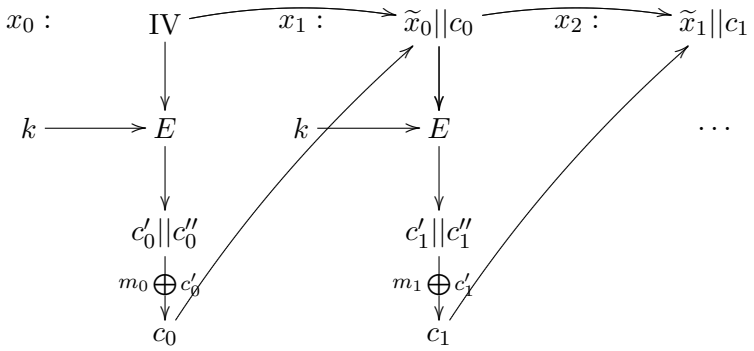
3.8 CBC versus CTR

Due to former definitions:

	CBC	CTR
Encryption	$c_i = E_k(m_i \oplus c_{i-1})$ $c_0 = E_k(m_0 \oplus \text{IV})$	$c_i = E_k(\text{nonce}\|i) \oplus m_i$
Decryption	$m_i = D_k(c_i) \oplus c_{i-1}$ $m_0 = D_k(c_0) \oplus \text{IV}$	$m_i = c_i \oplus E_k(\text{nonce}\|i)$
Speed	slower encryption of c_i require encryption of c_{i-1} (no parallel encryption)	faster con do encryption $E_k(\text{nonce}\|i)$ without knowing message. Encryption is more expensive than xor operation

3.9 Cipher Feedback Mode (CFB)

Definition *Cipher Feedback Mode (CFB)*
The *cipher feedback 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 $x_0 = \text{IV}$, which is separated in to blocks: the first block has a size of s (s block) and the second block has a size of $n - s$ ($n - s$ block). The encryption function takes 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 has a length of s (s bock) and so the second block 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 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:



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:

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

and the ciphertext values are given by:

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

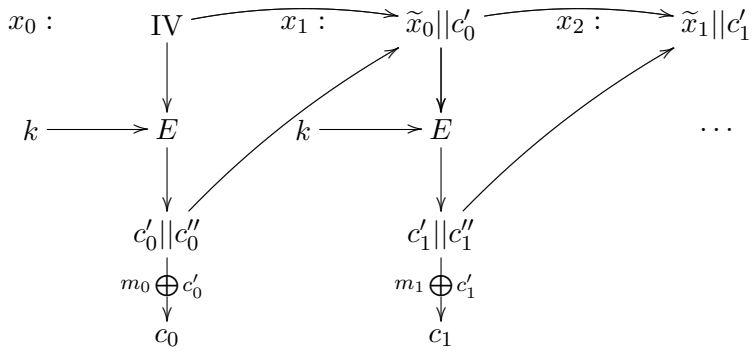
The decryption of a message given the ciphertext $c = c_0 c_1 \ldots c_{n-1}$:

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

3.10 Output Feedback Mode (OFB)

Definition *Output Feedback Mode*
The *output feedback mode* is similar to *CFB* but instead of taking the ciphertext and putting

that block into the x value. We take the outpt 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 crypto hash function, because in crypto hash functions the cipherblock text does depend on the previous message block (not given in *OFB*: c_2 doesn't depend on m_1).

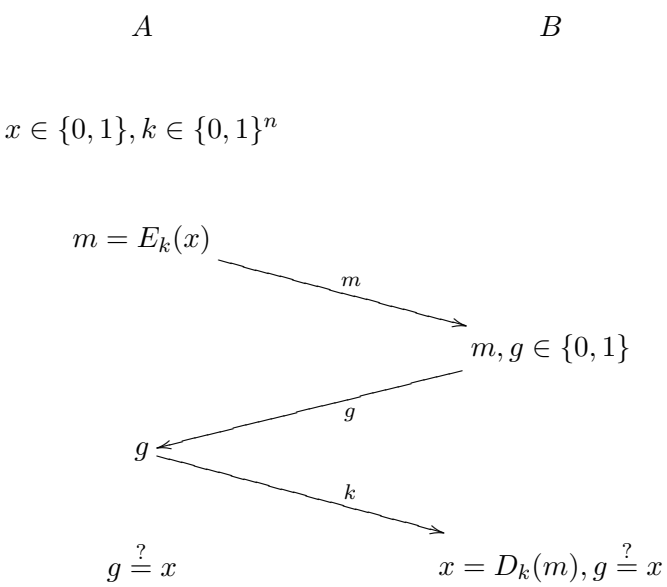
3.11 CBC vs. CFB

	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

3.12 Protocol

Definition *Protocol*
A *protocol* involves 2 or more parties and is a precicely define of a sequence of steps. Each steps can involve som computation, communication (sending data between the parties. A cryptographic protocal 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 19:
Making a coin toss over a channel via 2 parties A and B:
A picks a value for $x \in \{0,1\}$ with 0 representing 'Heads' and 1 representing 'Tails'. A also picks a random key k of length n (security parameter): $k \in \{0,1\}$. A will create a message m by encrypting x with k: $m = E_k(m)$. Next A sends the message m to B. B receives m and makes a guess g: $g \in \{0,1\}$. A receives g from B and 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

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

3.13 Padding

Definition *Padding*
For the protocol we use $x \in \{0,1\}$ which has 1 bit and $m = E_k(x)$. If using a block cipher that assumes that the input is 128 bits, the key is also 128 bits and the resulting ciphertext is 128 bits (minimum for AES). There using the ECB mode padding the input with 127 0-bits added after x.

3.14 Cryptographic Hash Function

Definition *Cryptographic Hash Function*
The *Cryptographic Hash Function* 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
- 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"):
  Gives $h = H(x)$, it's hard to find 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 and y, such that $H(x) = H(y)$

.

Example 20:
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)

3.15 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 and an attacker trying to find collusion can do no better than a brute force search on the size of h:

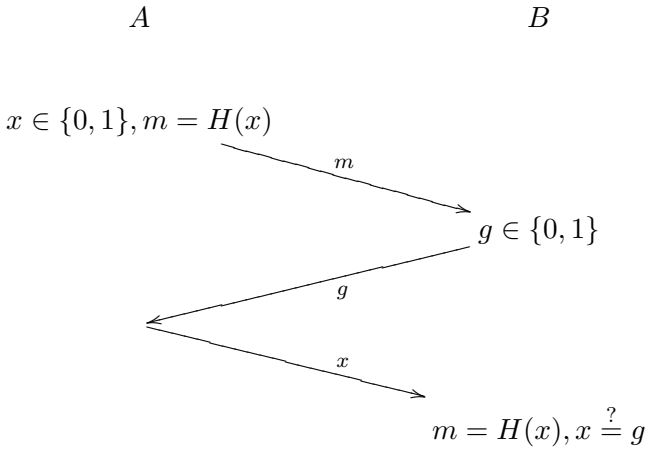
$$H(x) \mapsto h$$

Theorem 3.15.1:
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 21:
Consider a coin toss as in Example 19 with a hash function: A picks a number $x \in \{0, 1\}$ and computes the ideal cryptographic hash function (despit the ideal cryptographic hash function doesn't exist) $m = H(x)$ and sends m to B. B makes a guess and send it back to A. A send x to B and 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 22:
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(attacker can find x' 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 *Birthday-Paradox*23
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:

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

where $365 \cdot 364 \cdot 363 \cdot \cdots \cdot (365-k+1)$ are the number of ways to assign birthdays with no duplication among k people and $365 \cdot 365 \cdot 365 \cdot \cdots \cdot 365$ 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:

strong collusion resistance		weak collusion resistance
$1 - \frac{\frac{n!}{(n-k)!}}{n^k}$	$>$	$1 - (1 - \frac{1}{n})^k$

Example 24:
For $n = 365$ days and k people there is

k	P(at least one duplicate)
2	00.0027
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 thatn 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 25:
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

3.16 Strong Passwords

Any web application that can send you your password 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 mor 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 ($62^8 < 2^{48}$)

3.17 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 propertly)

- Protect encrypted passwords better
- add *salt*

3.18 Salted Password Scheme

The password file include

- username (userID)
- salt
- encrypted password

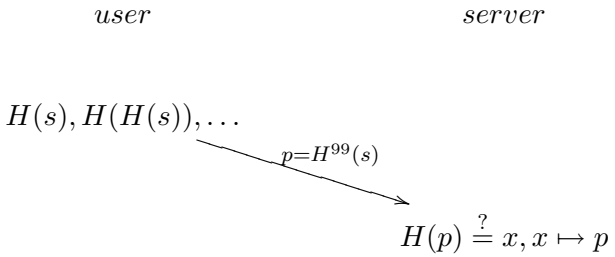
Definition *salt*
Salt are random bits (Unix: 12 bits). They don't need to be kept secret. The *salt* bits are different for each user. The encrypted password is the result of hashing the salt concatenated with the user's password.

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

That means: as long as the salts are different, even if the passwords are the same, the encrypted passwords will be different (Unix: $DES^2_{\text{salt}||\text{password}}5(0)$). 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.

3.19 Hash Chain, S/Key Password System

Definition *Hash Chain*
Selecting a secret $s \in \{0,1\}$ and compute hash of that secret: $H(s)$ and do this again: $H(H(s))$ and so on. Then the result is callen a *hash chain*.
A person check into a webpage gets $H(s), H(H(s)), \ldots$. 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 looks as follows:



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 $H^{100}(s), H^{101}(s), \ldots$ 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:

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

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.

4 Homework Unit 2

4.1 Randomness

Theorem 4.1.1:
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.

Theorem 4.1.2:
Mode of operation that can perform most of the decryption work in parallel:

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

5 Unit 3

5.1 Key Distribution

In *symmetric ciphers* all participating parties have the same key. 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.

5.2 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.
Consider 4 parties: A, B, C, D. Then A has a key with B, C, D, B has a key with C, D and C has a key with D. For 4 parties 6 keys are needed.
Theorem 5.2.1:
The number of pairwise keys for n people is:

$$\sum_{i=1}^{n-1}(n-i) = (n-1) + (n-2) + \cdots + \underbrace{(n-(n-1))}_{=1} = \sum_{i=1}^{n-1} i = \frac{n\cdot(n-1)}{2} \tag{5.1}$$

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 nth 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 (5.1) 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 26:

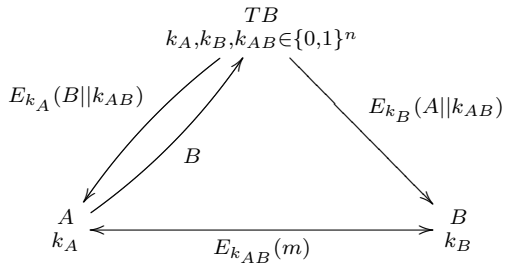
Assume the network has 100 people using pairwise shared keys. The the number of needed keys are

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

For a goup of 10^9 people we would need approximately 10^{18} keys to have pairwise keys.

5.3 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. 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:
A sends a request to TP for a communication with B. TP generates a random key k_{AB} and sends A the encrypted key k_{AB} concatenated with B's name (userID) with the shared key k_A and B the encrypted key k_{AB} concatenated with A's name (userID) with the shared key k_B. Now 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.

5.4 Merkle's Puzzle

Definition *Merkle's Puzzle*
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 $s \leq n$

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. Then A creates a set of N puzzles:

$$p = [E_{s'_1||0^{n-s}}(\text{'Puzzle': } 1], E_{s'_2||0^{n-s}}(\text{'Puzzle': } 2], \ldots, E_{s'_N||0^{n-s}}(\text{'Puzzle': } N]$$

That 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).
Next A shuffles the puzzle set and sends all N puzzles to B. 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).
Eventually B is going to find one that decrypt to

$$\text{'Puzzle': } p$$

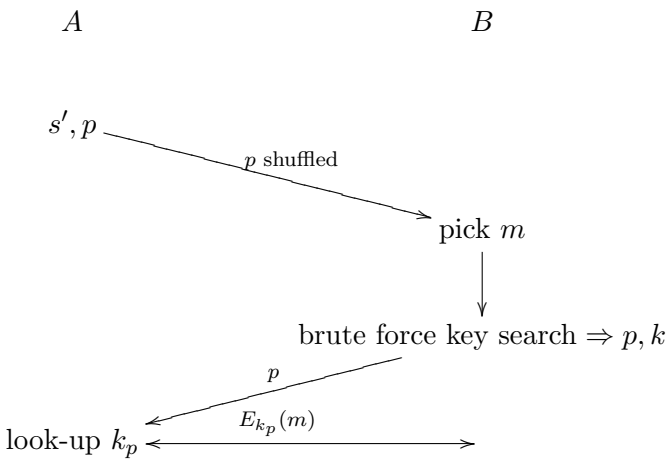
where $1 \leq p \leq N$ the number of the puzzle.
So B knows the guess g and the puzzle number p. For longer keys it's better to use:

$$\text{'Puzzle': } p, \text{'Key': } k$$

where k is a new random key of any length associated with that secret.
A will keep track of the keys in a list

$$k' = [k_1, k_2, \ldots, k_N]$$

So if B decrypts a puzzle, B acquires the puzzle number and the key number. B sends the number of the puzzle back to A and 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 5.4.1:

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 send 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 □

5.5 Diffie-Hellman Key Exchange

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

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

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

- Commutativity of powers (follows directly from commutativity): $\forall a, b, c \in \mathbb{R}, \forall n \in \mathbb{N}$:

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

Note: even using the mod operation the properties are valid.
Definition *Primitive Root*
Consider $q \in \mathbb{P}$ and $g \in \mathbb{R}$ is a *primtive root* of the prime number g if and only if:

$$\forall x \in \{1, 2, \ldots, q-1\} = \mathbb{Z}_q : \exists k \in \mathbb{N} : g^k = x$$

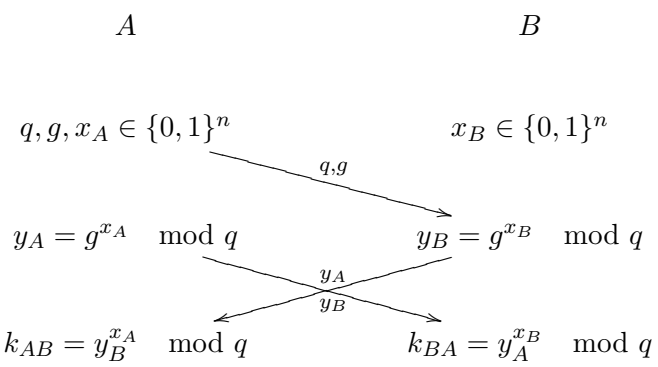
That means that the powers of g include all of the residue classes mod p excepts 0.
Example 27:
Consider $q = 7$ a prime number, then $g = 3$ is a primitive root of q because

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

Every number $x \in \{1, 2, 3, 4, 5, 6\}$ occurs once so 3 is a primitive root of 7.
Theorem 5.5.1:
If p is a prime number and $p > 2$, then there are always at least 2 primitive 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



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 5.5.2 (Correctness Property of DHKE)
In the DHKE protocol is:

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

Proof:

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

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

Due to the commutativity of power in multiplication is

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

□

Theorem 5.5.3 (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 knwon as hard. To break efficiently the Diffie-Hellman problem need a way to compute descrete logarithm efficiently.

Theorem 5.5.4 (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:

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

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.

5.6 Discrete Logarithm Problem

Definition *Continous Logarithm*
The solution fo 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 28:
It is $\log_2 8 = 3$ because $2^3 = 8$.
Definition *Discrete Logarithm*
The solution of the equation for known $a, b, n \in \mathbb{R}$

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

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 27 (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 size of n (linear in the value of n).
If it's possible to compute dlog efficiently then the attacker, who knows q, g, y_A, y_B hat to compute

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

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

5.7 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 $x = x_A, y = x_B$:

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

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

5.8 Impementing Diffie-Hellman

The first issue is to do fast modular exponentiation:

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

where x_A is some large random number and q is a large prime number and q a primitive root of p.
Theorem 5.8.1:
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** 29:

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 30:
A fast modular exponentiation that turns 3 values a, b, q to $a^b \mod q$ 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
```

.

5.9 Finding Large Primes

Theorem 5.9.1:
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

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

is not a prime number due to the assumption that $P = \{p_1, p_2, \ldots, p_n\}$ 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

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

Dividing by p_i results in

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

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

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

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:

$$N \leq x \sim \frac{x}{\ln x} \tag{5.2}$$

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 5.9.2:
The expected number of guesses N needed to find an n-decimal digit long prime number is

$$N \sim \frac{\ln 10^n}{2} \tag{5.3}$$

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 (5.2)

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

□

Example 31:
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 (5.3):

$$\frac{\ln 10^1 00}{2} \sim 115$$

Example 32:
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.

5.10 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 $k = 128$.

5.11 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

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

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

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

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 $a^{p-1} \equiv 1 \mod p$ also holds for all a relatively prime p. This test isn't fast enough to try <u>all</u> a with $1 \leq a < p$ because it's runningtime is exponential in the size of p.

5.12 Rabin-Miller Test

Definition *Rabin-Miller 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

$$a^s \equiv 1 \mod n \tag{5.4}$$

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

$$a^{s2^j} \equiv n - 1 \mod n \tag{5.5}$$

The big advantage is it's sufficient trying only a small number of values.
Example 33:
For $n < 1373653$ it's sufficient to try $a = 2$ and $a = 3$.
Theorem 5.12.1:
Finding any value that sadisfied (5.4) *or* (5.5) *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 <u>always</u> less than some constant and there are no bad numbers like the Charmichael number in the Fermat test.

Example 34:

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.