Applied Cryptography

Ivan Valentini
Telegram: @IvanV1337
Github: https://github.com/IvanValentini/Applied_Cryptography

February 1, 2023

Contents

Chapter 1

Introduction to Cryptography

1.1 Introduction

Cryptography refers to hidden writing. Its goal is to enable a secure communication between two users (Alice and Bob), and making it impossible for an eavesdropper to understand the information being exchanged.

By channel we mean any physical or logical medium of communication from one user to another. A channel becomes secure when the information exchanged over it cannot be overheard or tampered with by eavesdroppers. By default a channel is considered insecure, so the intent of cryptography is to make secure, an insecure channel.

To transmit data in a secure way Alice and Bob rather than transmitting the message in a plain form they first covert it to a disguised form. Formally these are called plaintext ($\mathcal{P}$) and ciphertext ($\mathcal{C}$). The idea is to transform a plaintext into a ciphertext, so that Alice sends the latter to Bob, and Bob is able to reconstruct the plaintext from the received ciphertext while this is very difficult (almost impossible) for Eve.

In practice there is a pair of functions:

- $enc : \mathcal{P} \to \mathcal{C}$
- $dec : \mathcal{C} \to \mathcal{P}$

Such that $dec(enc(m)) = m$ for every $m \in \mathcal{P}$. For this to work Alice and Bob need to agree on what encryption and decryption scheme to use without disclosing it to Eve. Eve will only be able to observe ciphertexts, but notice that if the sets of plaintexts and ciphertexts are too small, then

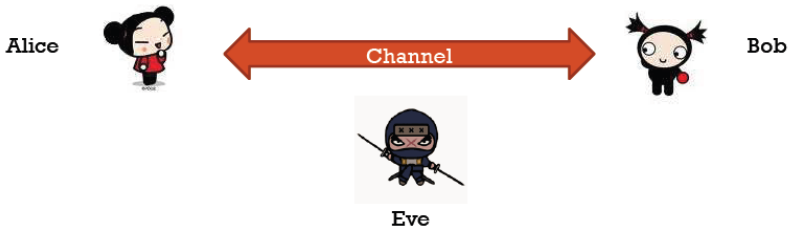


Figure 1.1: Alice, Bob, and the adversary Eve

Eve can try all the plaintext-ciphertext pairs (exhaustive search) or even if the sets of plaintexts and ciphertexts are large enough to make exhaustive search impractical, encryption and decryption can be defined in obvious way to allow Eve to easily reconstruct them (guessing). There is a problem with this formalization: these two functions, must be kept secret. These function can be used to create a secure channel, but how do you share these function? We need a secure channel, but if we had a secure channel, why not use it in the first place?

Since otherwise we would have to define and encryption and decryption functions for each pair of people that want to communicate securely we introduce the concept of a cryptographic key. We denote the set of (cryptographic) keys with $\mathcal{K}$, and consider a map $\varphi : \mathcal{P} \times \mathcal{K} \to \mathcal{C}$ such that for every key $k \in \mathcal{K}$ the function $\varphi(\cdot, k) : \mathcal{P} \to \mathcal{C}$ is an encryption function.

There are some key differences with respect to the old functions:

- The definition of φ (encryption algorithm) can be very complex but it can be public(i.e. known to anyone) and Alice and Bob can agree on it over an insecure channel. Before the encryption algorithm had to be private.
- The only component that must be kept secret(and thus exchanged over a secure channel) is the cryptographic key k, that defines the encryption function to use

The Kerchkhoff principle states that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge. The advantages to only exchanging cryptographic keys rather than ciphers are that it is easier to keep secret k than φ, and if the key is discovered it is sufficient to choose another key. The main disadvantage is that the attacker only needs to find the key to break the system.

1.2 Attacks

Let's now consider some useful attack models expressed in terms of what the attacker can observe and what queries they can make to the cipher. A query for our purposes is the operation that sends an input value to some function and gets the output in return, without exposing the details of that function. An encryption query, for example, takes a plaintext and returns a corresponding ciphertext, without revealing the secret key. We call these models black-box models, because the attacker only sees what goes in and out of the cipher. There are several different black-box attack models. Note that the higher the attacker skills and complexity of the attack the less (computational) effort the attacker needs to put to break the system.

1.2.1 Ciphertext-only attackers (COA)

Ciphertext-only attackers (COA), or known-ciphertext attackers, observe ciphertexts but don't know the associated plaintexts, and don't know how the plaintexts were selected. Attackers in the COA model are passive and can't perform encryption or decryption queries. The task of the attacker is very difficult and a lot of computational power is required to mount such an attack, this is because the attacker needs to check for every possibile key of if the decrypted ciphertext is meaningful 1.2.

In some cases the attacker only needs to know the probability distribution of the plaintexts, it could obtain a lot of information merely by observing some ciphertexts. This holds under the assumption that all plaintexts are encrypted with the same cipher and the same key. The method may be difficult (if possible at all) to apply to short messages or messages that contain words with many occurrences of letters with low frequencies. More information can be found here.

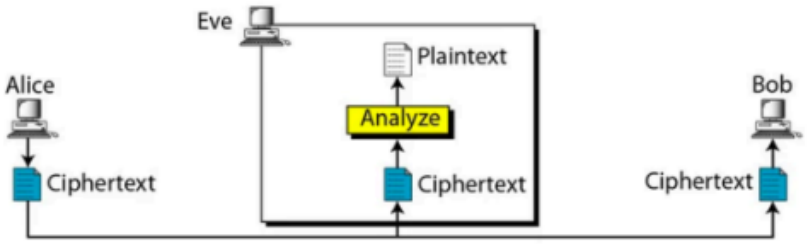


Figure 1.2

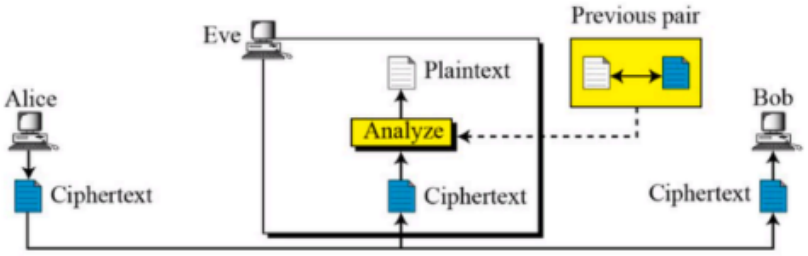


Figure 1.3

1.2.2 Known-plaintext attackers (KPA)

Known-plaintext attackers (KPA) observe ciphertexts and do know the associated plaintexts. Attackers in the KPA model thus get a list of plaintext–ciphertext pairs, where plaintexts are assumed to be randomly selected. Again, KPA is a passive attacker model, thus the attacker can not choose a plaintext to encrypt.

Essentialy the attacker will attempt to do what is known as a key recovery attack. But recovering the key might be a difficult problem to solve, so Eve might try to discover a functionally equivalent algorithm for encryption and decryption, or else design cryptographic algorithm that, even without knowing the key k, produces the same result as those of the cipher with the key k. This kind of attacks are known as Global Deduction/Reconstruction attacks 1.3.

1.2.3 Chosen-plaintext attackers (CPA)

Chosen-plaintext attackers (CPA) can perform encryption queries for plaintexts of their choice and observe the resulting ciphertexts. This model captures situations where attackers can choose all or part of the plaintexts that are encrypted and then get to see the ciphertexts. Unlike COA or KPA, which are passive models, CPA are active attackers, because they influence the encryption processes rather than passively eavesdropping 1.4.

With this the attacker may try to guess previously unknown plaintext-ciphertext pairs.

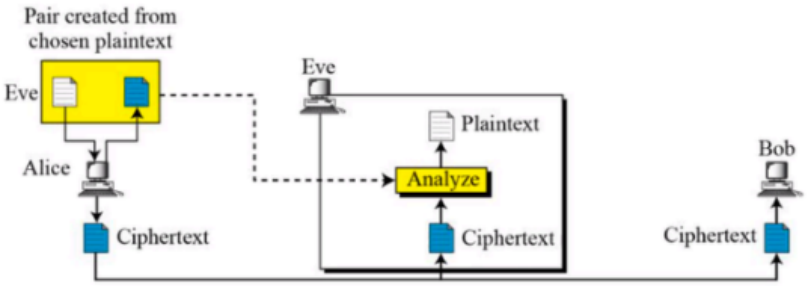


Figure 1.4

1.2.4 Chosen-ciphertext attackers (CCA)

Chosen-ciphertext attackers (CCA) [1] can both encrypt and decrypt; that is, they get to perform encryption queries and decryption queries. The CCA model may sound ludicrous at first—if you can decrypt, what else do you need?—but like the CPA model, it aims to represent situations where attackers can have some influence on the ciphertext and later get access to the plaintext. Moreover, decrypting something is not always enough to break a system.

1.3 Shannon Theorem

A cipher is broken if a method of determining the plaintext from the ciphertext is found without being legitimately given the decryption key. Any cryptosystem can be broken by an exhaustive key search. There exist ciphers that cannot be broken that are called perfect, and even exhaustive key search is of limited use for these ciphers. A cipher is perfect if, after seeing the ciphertext, an attacker gets no extra information about the plaintext other than what was known before the ciphertext was observed. The attacker might know the kind of content hidden but not the content itself, the point is that the knowledge of the attacker about the plaintext is not increased after intercepting the ciphertext. Let:

- $\mathcal{K}$ be the set of keys
- $\mathcal{M}$ be the set of messages
- $\mathcal{C}$ be the set of ciphertexts
- φ is a cipher such that $\forall k \in \mathcal{K}$

$$\varphi_k : \mathcal{P} \to \mathcal{C}$$

is an encryption function. Furthermore φ_k is invertible: there exists $\varphi_k^{-1}(\varphi_k(m)) = m$. We will give a notion of probabilities:

- if $m \in \mathcal{P}$, we use $P(m)$ to indicate the a priori probability that m occurs.
- if $k \in \mathcal{K}$, $P(k)$, indicates the probability that k is the chosen key

[1] Not in the slides, maybe not required to know this

- if $c \in \mathcal{C}$, $P(c)$ is the probability that c is the transmitted ciphertext
- $P(\mathcal{P})$ is the probability distribution of the plaintexts
- $P(\mathcal{C})$ is the probability distribution of the ciphertexts
- $P(\mathcal{K})$ is the probability distribution of the keys

We can assume $P(m) > 0$ because otherwise m never occurs and we can remove it from $\mathcal{P}$. In the same way we assume $P(k) > 0$ and $P(c) > 0$. The two probability distribution $P(\mathcal{P})$ and $P(\mathcal{K})$ determine the probability distribution $P(\mathcal{C})$, because only one ciphertext can be obtained using the encryption algorithm φ with a given plaintext and key. Clever boy assumption states that plaintext and the keys are indipendent. So we do not select a particular key for a particular plaintext, for any plaintext we can choose any key. A cipher is called perfect if:

$$\forall m \in \mathcal{P} \wedge \forall c \in \mathcal{C} \Longrightarrow P(m) = P(m \mid c)$$

So the probability of having the plaintext m is equal to the conditional probability of observing m over the channel once you have seen the ciphertext c. The conditional probability $P(x \mid y)$ denotes the probability that x occurs given that y occurred. We say that x and y are independent if $P(x \wedge y) = P(x)P(y)$. The concept of independence formalises the perception that past events do not influence the outcome of future ones or provide any information about them. In other words, if two events are independent, the order in which they occur is of no importance. So seeing the ciphertext c does not add any knowledge about the plaintext message sent, or any future message exchanged.

Shannon theorem: let us fix an integer n. Assume that: keys, plaintext, ciphertext have all n bits. Assume also that the plaintexts and the keys are independent (Clever boy assumption) and any n-bit string may be either a key or a plaintext. Then φ is a perfect cipher if and only if both the following conditions hold:

1. the keys are perfectly random
2. for any pair (m, c) of plaintext-ciphertexts, there is one and only one key k such that $c = \varphi(m, k)$

1.3.1 Consequences of Shannon Theorem

If φ is a perfect cipher, then all ciphertexts have the same probability to be received. Even if Eve knows the probability distribution of the plaintexts, she observes is a perfectly random cipher. Hence Eve cannot recover any information on the sent message from the intercepted ciphertext. But we made a very big assumption: this is true only if Eve can intercept **one** ciphertext. If Eve intercepts more ciphertexts encrypted with the same key, then the Shannon theorem no longer guarantees perfect secrecy. So in practical sense, a perfect cipher is not unbreakable, and not necessarily ideal, because every time we need to use a different key (the key space has to be as large as the message space) also the key has to be as large as a message transferred. Additionally we did not guarantee any sort of integrity!

1.4 Vernam Cipher - One Time Pad

The One Time Pad is an example of a perfect cipher. We work under the same assumptions of the Shannon Theorem. In the Vernam cipher we define encryption by means of the bitwise XOR

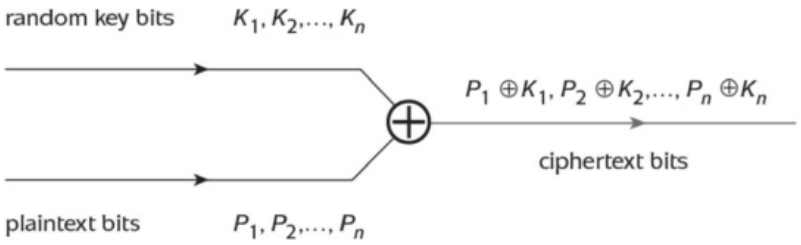


Figure 1.5

operation 1.5:

$$c = \varphi_k(m) = k \oplus m$$

Only if the key used is randomly chosen, this cipher is perfect. But there are a few problems: how do we choose the key randomly, how is the key shared, and the fact length of the key used that has to be as long as the message shared (the storage needed doubles) and having to use a key just once. Let's say that Eve known a single plaintext-ciphertext pair (m_1,c_1), the key can be easily recovered as $m_1 \oplus c_1 = m_1 \oplus (m_1 \oplus k) = (m_1 \oplus m_1) \oplus k = 0 \oplus k = k$. So any further messages exchanged with this key can be recovered by the attacker.

1.5 From perfect to ideal

We want the ability to encrypt a long message (e.g. a file of several megabytes) using a short key (e.g. a few hundred bits). We do not consider all possible adversaries, but only computationally feasible adversaries, that is, "real world" adversaries that must perform their calculations on real computers using a reasonable amount of time and memory. This leads to a weaker definition of security called semantic security. Furthermore, for our purposes a cipher is to be considered secure long as they do not leak any useful information about an encrypted message to an adversary other than the length of the message. Since the focus is on the "practical", instead of the "mathematically possible", one shall also insist that the encryption and decryption functions are themselves efficient algorithms and not just arbitrary functions. So we'll study ciphers are regarded as secure in practice because the known theoretical attacks take too much time to conduct. In other words, to implement such theoretical attacks requires resources which are unrealistic for any attacker.

1.5.1 Practical secure ciphers

Characterizing the notion of practical security is not an easy task as it must consider several different aspects including:

- Cover time: the time window in which a plaintext must be kept secret. This suggests that no attack on the cipher can be conducted in less than the cover time and implies that an exhaustive key search takes longer than the cover time. Evaluation should be repeated if a new attack is discovered or other parameters are changed such as the available computation power.
- Computational complexity: what computational processes are involved in known attacks on the cryptosystem how much time it takes to conduct these processes. measuring the time taken to perform the processes requires a way of measuring the time it takes to run a process, or a

function and this is expressed as a function of the size of the input that expresses the number of elementary operations performed, known as the Big O notation.

Most modern ciphers are regarded as secure in practice because the known theoretical attacks take too much time to conduct. Exhaustive key search has complexity 2^n, assuming a computer can attempt a million keys per second, an exhaustive search in a 30-bit key space will be covered in about a thousand seconds. Establishing the complexity of any known attacks is important and useful, but brings no guarantees of practical security and this is because there are undiscovered theoretical attacks, problems with the implementation, key management issues and so on. How can we be sure an attacker will require a large amount of work to break a non-perfect system with every method? An alternative, it is to show that breaking the cipher can be reconducted to a computationally difficult problem. This is typically the approach used to show the security of public key ciphers.

Chapter 2

Stream Ciphers

2.1 The problem with Vernam cipher

There are few issues: generating a truly random key as long as the message, find a secure channel for transportation of the key to the message recipient and do this for every single message to be exchanged. In summary, the problem becomes to securely transfer large quantities of secure keys. There are two approaches that are seen as an improvement to Vernam cipher: stream ciphers and block ciphers.

2.2 Symmetric encryption

In stream ciphers we take a seed (a small vector of a few random bits) that must be kept secret, then build a keystream (a very long sequence of pseudorandom bits), finally xor the keystream with the plaintext bitwise to calculate the ciphertext. The problems are how are we going to generate a perfectly random keys of arbitrary size and how can we replicate the random streams of bytes for decryption. In block ciphers we use the same key multiple times in a way that does not compromise the cipher. The problems are how can we reuse multiple times the same key without enabling an attacker to perform cipher-text only attacks and how can we avoid attackers to exploit the block structure.

These two ciphers are known as symmetric encryption algorithms, where stream ciphers perform operations in a way such that the plaintext is processed one bit at a time, and the algorithm selects one bit of plaintext, performs a series of operations on it, and then outputs one bit of ciphertext, block ciphers perform operations in a way such that the plaintext is processed in blocks (groups) of bits at a time, and the algorithm selects a block of plaintext bits (typically 64 bits), performs a series of operations on them, and then outputs a block of ciphertext bits. Notice that a stream cipher can be seen as a block cipher with blocksize set to 1 bit, but there are also stream ciphers that process data in bytes, and hence could be regarded as block ciphers with a block size of 8, as a rule of thumb if the blocksize is less than 64 bits we talk about stream ciphers otherwise we talk about block ciphers.

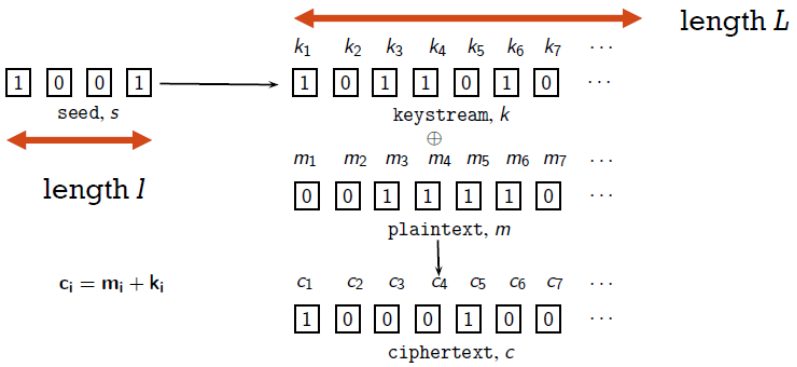


Figure 2.1: Stream cipher

2.2.1 Stream ciphers

The real work in designing a good stream cipher goes into designing the keystream generator. Keystream generators produce output which appears to be randomly generated, but is actually not randomly generated, these are referred to as pseudorandom generators. In many cases, stream ciphers combine the keystream with the plaintext in more complex ways than a simple bitwise xor operation.

For a stream cipher to be good, Eve should not be able to: recover the seed by making the set of possible seeds so large that and exhaustive search is very hard in practice and also predict the rest of the keystream by eliminating any patterns from the keystream. More precisely: choose a short l-bit (much smaller than the lenght L of the plaintext to be encrypted) seed s as the encryption key and stretch the seed into a longer L-bit string (the key) that is used to mask the message and decrypt the ciphertext. The seed s is stretched using some efficient, deterministic algorithm G that maps l-bit strings (seeds) to L-bit strings(keys) 2.1. Formally:

- Encryption: $G(s) \oplus m$ for any seed s (of size l) and plaintext m (of size L)
- Decryption: $G(s) \oplus c$ for any seed s (of size l) and ciphertext c (of size L) where G is called a pseudo-random number generator

Notice that if $l < L$, then by Shannon's Theorem, stream ciphers cannot be perfect, however, if G has certain properties, then stream ciphers are secure in practice. Suppose s is a random l-bit string and r is a random L-bit string, if Eve cannot effectively tell the difference between $G(s)$ and r, then it should not be able to tell the difference between stream ciphers and one-time pad. Since the one-time pad cipher is secure, so should be the stream cipher.

An algorithm that is used to distinguish a pseudo-random string $G(s)$ from a truly random string r is called a statistical test. How might one go about designing an effective statistical test? One basic approach is the following: given an L-bit string, calculate some statistic, and then see if this statistic differs greatly from what one would expect if the string were truly random. For example, a very simple statistic that is easy to compute is the number k of 1's appearing in the string. For a truly random string, we would expect $k \approx L/2$. If the PRG G had some bias towards either 0-bits or 1-bits, we could effectively detect this with a statistical test that, say, outputs 1 if $|k - 0.5L| < 0.01L$, and otherwise outputs 0. This statistical test would be quite effective if the PRG G did indeed have some significant bias towards either 0 or 1.

A stream-cipher is well equipped to encrypt a single message from Alice to Bob. If two messages are encrypted with the same key there may be problems. As an example consider the case in which Alice and Bob want to exchange messages m_1 and m_2, let $c_1 = m_1 \oplus G(s)$ and $c_2 = m_2 \oplus G(s)$. If Eve is able to intercept both ciphertexts, then it is able to calculate $c_1 \oplus c_2 = (m_1 \oplus G(s)) \oplus (m_2 \oplus G(s)) = (m_1 \oplus m_2) \oplus (G(s) \oplus G(s)) = m_1 \oplus m_2$, and as english text contains enough redundancy that given $m_1 \oplus m_2$, Eve can recover both m_1 and m_2 in the clear by using frequency analysis (given that both are sufficiently long). For this reason a stream cipher key should never be used to encrypt more than one message.

Stream-ciphers are said to be malleable since an attacker can cause predictable changes to the plaintext, this is because and attacker can intercept ciphertext c and forwarding $c' = c \oplus d$, effectively the receiver will get $m' = c' \oplus G(s) = (c \oplus d) \oplus G(s) = (c \oplus G(s)) \oplus d = m \oplus d$. So again stream-ciphers do not provide integrity. Regarding key management, stream ciphers do not require the key to be as long at the message encrypted, like the one time pad does. The key used to generate the keystream (and that must be distributed) is much shorter. In one-time pad the key has to be truly randomly generated, which involves costly generation techniques. The keystream in a stream cipher is pseudorandom and thus is much cheaper to generate. A keystream generator is a deterministic process since every time the same seed is input into the keystream generator, it will result in the same keystream being output. If we reuse a seed to produce the same keystream and then encrypt two plaintexts using the same portion of the keystream, then, just as in a one-time pad, the xor between the two ciphertexts will tell us the difference between the two corresponding plaintexts. We can avoid this problem by, e.g., making the keystream dependent on time varying data or generating the ciphertext with more complex operations than a xor.

In summary stream-ciphers do not give rise to error propagation as each bit in the ciphertext depends on just one bit in the plaintext and 1 bit transmission errors will result in 1 bit error in the plaintext, also they are very fast making them ideal for real time applications (e.g. mobile communication services) and easy to implement in hardware and don't require large memory capabilities. Since stream ciphers process data bitwise, it is crucial that sender and receiver keep their keystreams in perfect synchronization and 1 bit data loss may have catastrophic consequences as decryptions are performed on the wrong bits after the receiver is out of sync of the sender and re-synchronization mechanisms must be put in place to avoid these problems.

2.2.2 Vigenère cipher

It can be seen as a variant of Vernam cipher whereby the key is a sequence of bits of fixed length. The key, and plaintext are a string of bytes, to encrypt: XOR each character in the plaintext with the next character of the key and wrap around in the key as needed.

Vigenere cipher can be attacked by first determining the key length and determining each byte of the key by using frequency analysis.

2.3 Examples of symmetric ciphers

2.3.1 Examples of stream ciphers

- RC4: Simple and fast stream cipher with a relatively low level of security, probably the most widely implemented stream cipher in software and widely used in SSL/TLS, WEP, and Microsoft Office

Type	Cipher	Applications
Stream	A5/1, A5/2, A5/3	Phone (GSM)
	RC4	Internet
	E0	Bluetooth
Block	DES	(old)
	3DES	Smart cards
	AES	Everywhere
	PRESENT	sensor networks

Figure 2.2: Symmetric ciphers

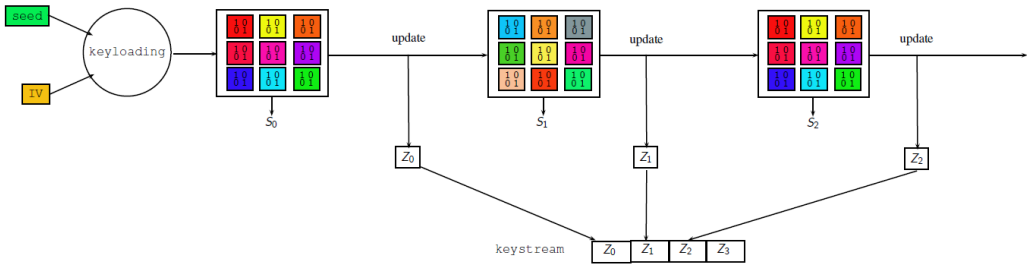


Figure 2.3: Stream ciphers internals

- A5/1: One of the stream cipher algorithms used in GSM to secure the communication channel over the air from a mobile phone to the nearest base station
- E0: The stream cipher algorithm used to encrypt Bluetooth communications

2.4 Implementation of stream ciphers

First we want to tackle the problem of generating a randomly a key (known as keystream) that is as long as possible. Keystream generators should be fast (as in computable in polynomial time as function of number l of bits in the seed) and be secure, so intuitively, a string of L bits produced by a keystream generator should look random. (i.e. it should be impossible in a polynomial amount of time in l to distinguish between a truly random bit string of length L and a string of the same length returned by the keystream generator). More importantly, a keystream generator, given the same seed it should produce the same sequence! The main components are 2.3:

- States: vector of bits organized in registers $(S_0, S_1, \cdots)$
- Update function: function mapping a state to the next state (clock function)
- Output function: function extracting a bit from a state. Concatenating all bits returned by this function, it is possible to obtain the keystream
- Key loading: function that takes the seed (secret) and a (public) initialization vector (IV) to compute the initial state for the update function. Each IV should be used only once.

2.5 Warm Up

The first output bits strongly depend on the initial state. To avoid potential problems, it is customary to run a warm-up phase before starting encryption. This preliminary phase consists in applying the update function several times without outputting any bits of keystream and it is a highly recommended security best practice. Given any initial state, the states are periodic, since they are in a finite number and at some point we will obtain again one of the previous states. The keystream is also periodic and this is impossible to avoid. The smallest number i such that $update(\cdots(update(S))) = S$ is called the period of the keystream (it depends on the initial state), and as a requirement is that the period of the keystream shall be quite large, regardless of the initial state, and can be achieved by a suitable design of the update function. As an example let's take an un update function as follows $f : (x, y, z) \Rightarrow (y + z, x, y)$. One can see that the repeated application of f to the inital state $(1, 0, 1)$ will yield $(1, 0, 1) \Rightarrow (1, 1, 0) \Rightarrow (1, 1, 1) \Rightarrow (0, 1, 1) \Rightarrow (0, 0, 1) \Rightarrow (1, 0, 0) \Rightarrow (0, 1, 0) \Rightarrow (1, 0, 1)$ with a period of 7. We are using linear functions because they are easy to implement and compute. Later we will see that linear functions (alone) should never be used.

2.6 Linear feedback shift register (LFSR)

A linear feedback shift register of length n is a shift register composed by n bits. At any clock the following operations are executed:

- the last bit on the right is output and forms part of the keystream
- the other bits in the register are shifted to the right by one position
- the XOR of some bits of the register are put in the first position on the left.

A shift register is a type of digital circuit using a cascade of flip flops (device which stores a single bit of data) where the output of one flip-flop is connected to the input of the next. Flip-flops share a single clock signal, which causes the data stored in the system to shift from one location to the next. A linear-feedback shift register (LFSR) 2.4 is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is xor. An LFSR is usually a shift register whose input bit is driven by the xor of some bits of the overall shift register value, and the initial value of the LFSR is called the seed. Since the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state.

Since the register has a finite number of possible states ($2^n - 1$), it must eventually cycle. An LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle. Typically the linear feedback function has the following form: $c_1q_1 \oplus c_2q_2 \oplus \cdots \oplus c_nq_n$ where $\oplus$ denotes the XOR operation. The non null c_i are called taps. Given a seed s, the period of s is the number of steps the LFSR takes to return to s. The period of the LFSR is the maximum period achieved for any seed. For any number n of taps there exists a maximal LFSR. Of the $2^n - 1$ possible LFSRs, which taps correspond to maximal LFSRs? To answer this question we can use finite fields.

2.7 LFSR and finite fields

Note: This is an informal introduction aiming to convey ideas with no attempt to mathematical rigour.

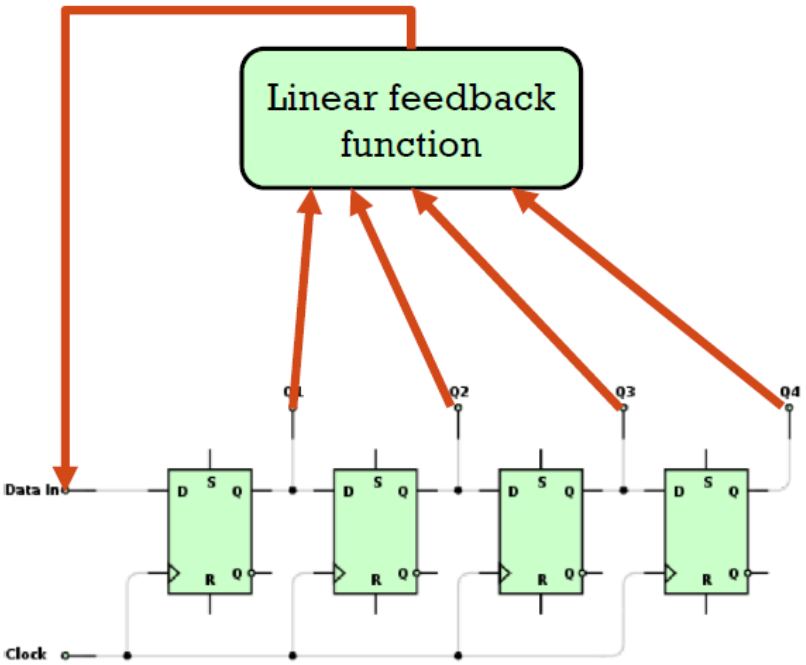


Figure 2.4: Linear feedback shift register

Consider addition modulo a certain number N, this is an instance of the notion of group, namely:

- a set closed under a binary operation (addition modulo N)
- the operation is associative (addition modulo N is so)
- there is an identity (0)
- each element has an inverse (e.g., 1 is the inverse of 11 when considering addition modulo N=12 as adding them gives 0)

The order of a group is the number of elements in its set (in the case of addition modulo a number N, the order of the group is N). The order of an element a in a group is the number of times one needs to apply the operation to a to produce the identity (example, the element 4 in the group modulo 12 has order 3 since $4 + 4 + 4 = 0$). Consider now the multiplication modulo some number N. It is interesting to consider the order of its various elements that may be regarded to form cycles, containing the elements obtained by repeatedly applying the operation to the initial element up to its order, covering all (in case the order of the element is N) or some of the N elements:

- 3 has order 3 since $3*3 \bmod 13 = 9$, $9*3 \bmod 13 = 1$ and $1*3 \bmod 13 = 3$, the cycle generated is $\{3, 9, 1\}$
- 2 has order 12 since $2 * 2 \bmod 13 = 4$, $4 * 2 \bmod 13 = 8$, $8 * 2 \bmod 13 = 3$, $3 * 2 \bmod 13 = 6$, and so on, the cycle generated is $\{2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1\}$

It is not difficult to see that there are other elements that can generate cycles containing all 12 elements of the group, namely 6, 7, and 11. Elements whose order is equal to the order of the group

0	1	x	$x+1$
x^2	x^2+1	x^2+x	x^2+x+1
x^3	x^3+1	x^3+x	x^3+x+1
x^3+x^2	x^3+x^2+1	x^3+x^2+x	x^3+x^2+x+1

Figure 2.5

are called generators. Multiplication modulo a prime number forms a group with similar properties to those discussed considering multiplication modulo 13. The order of an element is always a divisor of the group order by Lagrange's theorem. Because some elements are generators of the group itself, it is called a cyclic group. The number of generators of the "group multiplication modulo a prime number" p having the full order $p-1$, is $\varphi(p-1)$ where $\varphi(\cdot)$ is Euler's totient function, namely:

$$\varphi(n) = n\prod_{p|n}(1-\frac{1}{p})$$

It counts the positive integers up to a given integer n that are relatively prime to n. For $p=13$ in the previous example, we have that $\varphi(12) = 12 * \frac{1}{2} * \frac{2}{3} = 4$ which corresponds to what we have observed, i.e. that the 4 elements 2, 6, 7, and 11 are generators.

Consider again a generator $g=2$ of the group multiplication modulo the prime $p=13$. We can write g^0 to denote 1, g^1 to denote 2, g^2 to denote 4, this allows us to list the elements of the group as powers of the generator, i.e. $1=g^0$, $2=g^1$, $4=g^2$, ... and this can be done for any group modulo a prime.

A ring is a group such that:

- the group binary operation is commutative
- has another binary operation which is: closed over the ring's elements, associative, has an identity element and distributes over the group operation

A finite field has:

- a finite set of elements
- two binary operations, which are abstract analogues to addition and multiplication
- analogues to subtraction and division using additive and multiplicative inverses

A **field** is a ring where both operations are commutative and have inverses. Intuitively a ring has addition, subtraction, and multiplication well-defined, a field adds division. A finite filed is also called Galois field and integer arithmetic modulo a prime is a finite field.

The simplest Galois field is $GF(2)$: it contains two elements (0 and 1) and the binary operations are addition and multiplication both modulo 2. On top of $GF(2)$ it is possible to construct: $GF(2)[x]$ that is the set of polynomials in x with only the elements of GF(2) allowed as coefficients and $GF(2)[x]/p$ that is the quotient of the set of polynomials by p.

Consider the polynomial $p(x) = x^4 + x + 1$. In this case, $GF(2)[x]/p$ contains all possible polynomials of degree less than 4, namely 2.5

Bits (n)	Feedback polynomial	Taps	Taps (hex)	Period ($2^n - 1$)
2	$x^2 + x + 1$	11	0x3	3
3	$x^3 + x^2 + 1$	110	0x6	7
4	$x^4 + x^3 + 1$	1100	0xC	15
5	$x^5 + x^3 + 1$	10100	0x14	31
6	$x^6 + x^5 + 1$	110000	0x30	63
7	$x^7 + x^6 + 1$	1100000	0x60	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	10111000	0xB8	255
9	$x^9 + x^5 + 1$	100010000	0x110	511
10	$x^{10} + x^7 + 1$	1001000000	0x240	1,023
11	$x^{11} + x^9 + 1$	10100000000	0x500	2,047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	111000001000	0xE08	4,095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	1110010000000	0x1C80	8,191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	11100000000010	0x3802	16,383
15	$x^{15} + x^{14} + 1$	110000000000000	0x6000	32,767
16	$x^{16} + x^{15} + x^{13} + x^4 + 1$	1101000000001000	0xD008	65,535
17	$x^{17} + x^{14} + 1$	10010000000000000	0x12000	131,071
18	$x^{18} + x^{11} + 1$	100000010000000000	0x20400	262,143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	1110010000000000000	0x72000	524,287
20	$x^{20} + x^{17} + 1$	10010000000000000000	0x90000	1,048,575
21	$x^{21} + x^{19} + 1$	10100000000000000000	0x140000	2,097,151
22	$x^{22} + x^{21} + 1$	11000000000000000000	0x300000	4,194,303
23	$x^{23} + x^{18} + 1$	10000100000000000000	0x420000	8,388,607
24	$x^{24} + x^{23} + x^{22} + x^{17} + 1$	11100001000000000000	0xE10000	16,777,215

Figure 2.6

Polynomials $p(x)$ such that x is a generator of $GF(2)[x]/p(x)$ are called primitive polynomials. In practice, this means that it is possible to start with 1 and, by using the procedure below, it is possible to generate all the elements of $GF(2)[x]/p(x)$

- consider 1
- multiply by x and if the result contains a term x^N with $N = \textit{degree of } p(x)$, then subtract $p(x)$
- stop when the result is 1, this happens when the power of the generator is $2^N - 1$

Primitive polynomials $p(x)$ allow us to design maximal LFSRs by exploiting the following correspondence between these and $GF(2)[x]/p(x)$. In conclusion $GF(2)[x]/p(x)$ "corresponds" to $GF(2^N)$ when $p(x)$ is a primitive polynomial of degree N. These are useful because guarantee to generate the longest possible cycle.

Given a maximum-length LFSR of n bits and reading $2^n - 1$ consecutive bits of the m-sequence that it produces, we have that:

- one half of the bits are 1 and one half are 0 (actually the 1's are one more than the 0's)

- there are 2^{n-1} runs: 1/2 of the runs has length 1, 1/4 of the runs has length 2, ..., $1/2^i$ of the runs has length i (for $2 \leq i \leq n-2$), there is only one run of $n-1$ zeros and none of the runs has $n-1$ ones, there is only one run of n ones and none of the runs has n zeros.

As an example consider using as feedback polynomial the primitive polynomial x^5+x^2+1, hence the period is $2^5-1=31$ bits and we expect $2^{5-1}=16$ runs. Starting from the state (10000), the obtained m-sequence is:

$$0000-1-00-1-0-11-00-11111-000-11-0-111-0-1-0-1$$

There are indeed 16 runs and 8 runs have length 1, 4 runs have length 2 , 2 runs have length 3, there is only 1 run of 4 zeros and none of 4 ones and there is only 1 run of 5 ones and none of 5 zeros.

2.8 LFSR and linear algebra

For analysing LFSRs, it is useful to investigate their relationships with Linear Algebra. For this, the key idea is the notion of companion matrix: given the polynomial $p(x) = c + c_1 * x + \cdots + c_n * n^n$ in $GF(2)[x]$ its companion matrix is:

$$C(p) = \begin{bmatrix} 0 & 0 & \cdots & 0 & c_0 \\ 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_{n-1} \end{bmatrix}$$

It is possible to show that multiplication by C is equivalent to multiplication by x in $GF(2)[x]/p(x)$. This connection allows us to consider the state recovery problem, i.e. predicting the state vector of the LFSR from its output bits. If we can solve this we can recover the initial state or, equivalently, the seed. And this requires polynomial time computations, this leads to the LFSRs main weaknesses: LFSRs can not be used directly in cryptography because of their linearity. Given any LFSR of length n, the j-th bit of the keystream, for can be obtained as a linear combination of its previous n bits keystream.

Let the bits of the output sequence denoted by $a_0, \ldots, a_n$, there exists n bits $\lambda_0, \ldots, \lambda_n$ such that $\sum_{i=0}^{n} \lambda_i a_i = 0, \lambda_n = 1$, note that the values $\lambda_0, \ldots, \lambda_n$ do not depend on when Eve starts intercepting the ciphertext.

Knowing the values of $\lambda_0, \ldots, \lambda_n$ is equivalent to knowing the feedback polynomial: $g = \sum_{i=0}^{n} \lambda_i x^i$ and thus being able to predict the keystream. Eve may obtain the required subsequence of the keystream mounting a known-plaintext or chosen-plaintext attack, hence LFSRs are vulnerable with respect to known-plaintext attacks and they must not be used as keystream generators, although they can be used as components in particular when iterated and combined by using some kind of non-linear functions. Linearity is good because it is easy to implement, but on the other hand is not secure enough.

2.8.1 DVD encryption

DVD encryption uses the CSS stream cipher to encrypt movie contents using a 40-bit secret key. The CSS stream cipher is particularly weak as it can be broken in far less time than an exhaustive search over all 2^{40} seeds 2.7.

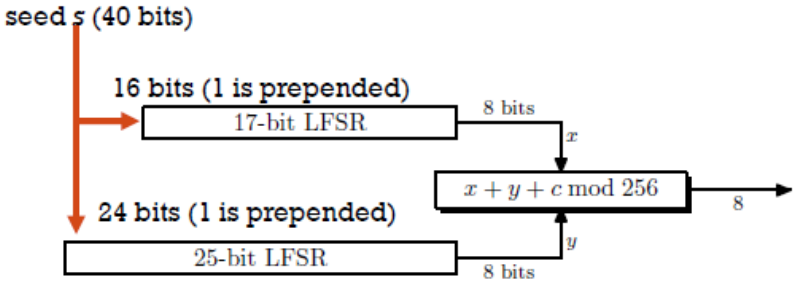


Figure 2.7: DVD CSS Encryption

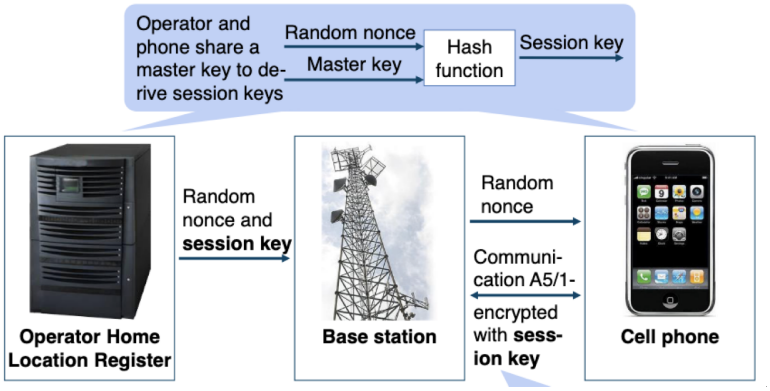


Figure 2.8: GSM

The two LFSRs are run in parallel for 8 cycles and then the resulting bits are considered as integers and added modulo 256. The simplest attack is simple bruteforce: suppose Eve knows the first 100 bytes of the output sequence, then just guess a 40 bits seed, run for 100 iterations and compare with the output sequence, if they match the seed was found, otherwise try another seed. And even smarter approach could be used to only attempt at most 2^{16} seeds.

2.9 A5 family of ciphers

2.9.1 GSM

The GSM (Global System for Mobile communication) standard was designed from 1982-1991, the level of security specifications regarding both authentication and encryption were limited: algorithms were never officially published (achieving security by obscurity), thus algorithms were reverse-engineered or leaked, leading to revelations of several possible attacks, and within a few months after the release, most of the cryptographic schemes had been compromised and some were even proven to be close to useless 2.8.

Each frame is numbered by a frame number, obtained by a frame counter initialized with 0 at conversation-start and incremented by 1 mod 2^{22} with each frame sent. An algorithm of the A5 family takes the session key Kc (symmetric) and a frame counter Fn and generates 228 pseudo random bits (PRAND) called a key stream. The keystream is then XORed with a 228 bit segment

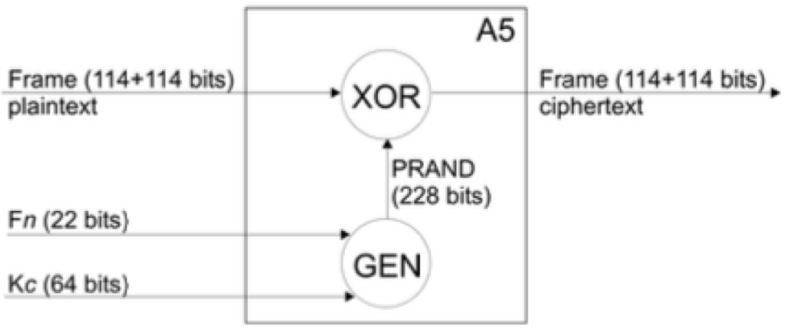


Figure 2.9: A5 ciphers operation

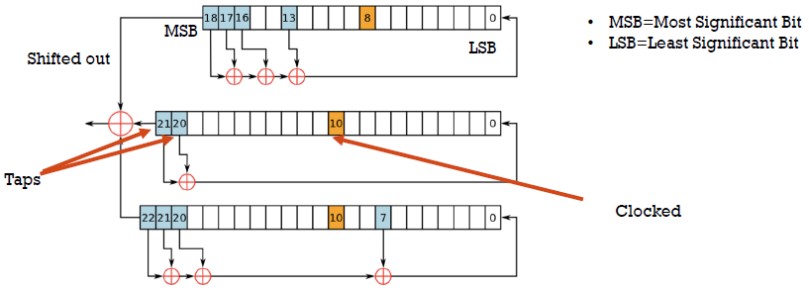


Figure 2.10: A5/1

of plain text yielding 228 bits of ciphertext 2.9.

2.9.2 A5/0

It is the weakest of the A5 versions as it offers no encryption. It is a no-operation cipher, that generates the pseudo random bits by negating the input frame, thus leaving out the XOR function. The result is an algorithm that outputs the plain text it received as an input. This version is found in third world countries or countries with UN sanctions.

2.9.3 A5/1

It uses 3 LFSRs (with lenghts 19,22,23, total 64 bits of state) and has a keystream lenght of 228 bits 2.10.

When a register is clocked, its tapped bits are XORed and the result is stored in the registers LSB, the register MSB is shifted out of the register, and its value is forgotten.

In short first a seed (secret vector K of 64 bits) is selected from the session key, then an initialization vector (public vector IV of 22 bits) that is the frame counter, from K and IV we get the inital state (namely a vector of 64 bits) using a keyloading function $kl(K, IV)$. The function kl is defined by iteration: initially the registers contain all zeros, the bits in K are injected by xor-ing them with other bits, and similarly for the bits of IV. After the keyloading the warmup starts, the registers R1, R2, and R3 are irregularly clocked 100 times, without producing output, and it is as follows: in each step, the clock bits, each one among R1, R2, and R3 is updated if the clock bits

agree with the majority of the clock bits. In this way, it is possible to show that each register clocks with a probability of 3/4. At this point, the initialization of the registers is complete and the stream cipher is ready to output the key-stream.

The keystream is is composed of 228 bits that are obtained in 228 steps, at each clock tick the update function is defined as follows:

- The register R1 is updated if its clock bit agrees with the majority of the others
- The register R2 is updated if its clock bit agrees with the majority of the others
- The register R3 is updated if its clock bit agrees with the majority of the others

The ouput function is defined as: the xor of the output of the three registers R1, R2, R3 is the output bit. After 228 bits of output, the key-loading phase is executed again.

2.9.4 A5/2

The key stream length remains the same as A5/1 but the number of LFSRs is increased to 4 respectively with lengths 19,22,23,17 with a total 81 bits of state.

In short a seed (secret vector K of 64 bits) is selected, and an initialization vector (public vector IV of 22 bits), from K and IV we get the inital state (namely a vector of 64 bits) using a keyloading function $kl(K, IV)$. kl and warmup are defined similarly to A5/1.

At each clock tick the update function is defined as follows:

- R_1 is updated if $R_4[6] = \text{maj}(R_4[6], R_4[13], R_4[9])$
- R_2 is updated if $R_4[13] = \text{maj}(R_4[6], R_4[13], R_4[9])$
- R_3 is updated if $R_4[9] = \text{maj}(R_4[6], R_4[13], R_4[9])$

For the output a majority function is applied as non-linear filtering function to each of the first three registers After a month an attack was found that could break the cipher almost in real time with a known plaintext attack.

2.9.5 A5/3

A5/3 is the last stream cipher of the A5 family and provides users with a higher level of security than both A5/1 and A5/2. It is based on the block cipher KASUMI. The keyspace is 128 bits and the message space is 64 bits 2.11.

The problem is that the key generated Kc is only 64 bits, so the maximal exhaustive search complexity is (only) 2^{64}. To make things worse Kc generated only once after the cell phone registers with the network and stays active for all communication, until the telco requests a new one or the cell phone deregisters (so if one key is compromised every other communication is compromised until a new key is requested), and Kc is artificially shortened in deployed systems when zeroing 10 bits, lowering search complexity to 2^{54}, and encryption is applied after error correction.

2.10 Remarks on the A5 Family

- A5/1 is affected by a number of serious weaknesses, and its use is strongly discouraged, since there are practical attacks that can break the cipher

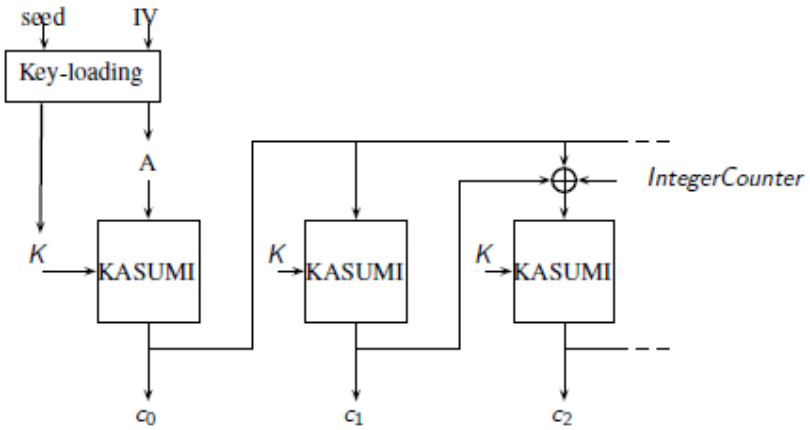


Figure 2.11

- A5/2 is extremely weak and it can be broken in real time with inexpensive equipment; it is therefore no longer supported by new mobile phones
- A5/3 is the common standard for the new generation of mobile and it is considered secure, even though there do exist practical attacks to KASUMI that suggest some significant weaknesses of the cipher

2.11 Practicality of attacking GSM communication

In theory, the attacks are relatively simple. In practice, a considerable amount of hardware is necessary to actually intercept GSM communications. The hardware must at least consist of a radio receiver device which is capable of receiving and decoding digital data that is exchanged over-the-air. Hypothetically a simple GSM mobile phone already has all these capabilities (except the decrypting of an unknown A5 stream), so it might be possible to use such a phone for eavesdropping nevertheless a huge amount of know-how, time and money is needed.

2.12 Bluetooth

Bluetooth is a wireless technology standard invented by Ericsson in 1994 for exchanging data over short distances using short-wavelength UHF radio waves (Range: 2.4 to 2.485 GHz) from fixed and mobile devices. The standard offers methods for generating keys, authenticating users, and encrypting data. BLE was introduced in 2011 as Bluetooth 4.0, the main difference between BLE and Bluetooth is power consumption. BLE uses the AES cipher with 128-bit key length to provide data encryption and integrity over the wireless link.

2.12.1 E0

Used in Bluetooth standard and introduced in 1999. It uses 4 linear registers with lengths 25,31,33,39 with a total linear par of 128 bits. It has also one non linear register that is 4 bits. The update functions depend on the non-linear register 2.12.

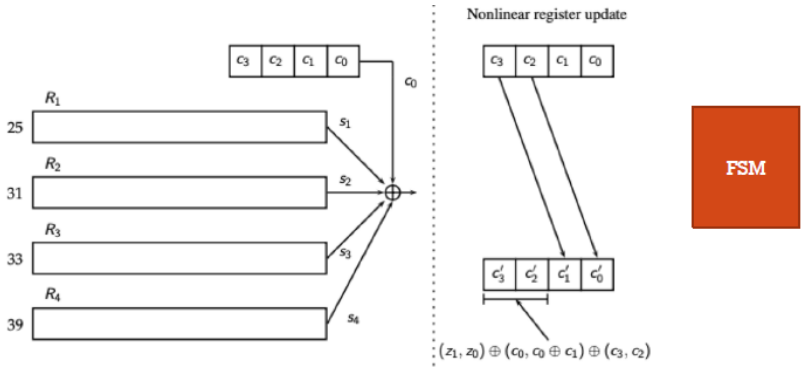


Figure 2.12

2.13 RC4

Rivest Cipher 4 was designed in 1987. At first its implementation was trade secret, but then someone reverse engineered it, later Rivest confirmed that the code that was leaked was in fact correct. RC4 is used mainly because it is easy to implement in hardware and it is fast. RC4 was rapidly adopted in commonly used encryption protocols and standard like WEP and SSL.

RC4 generates a pseudorandom stream of bytes, the key stream. To generate the keystream, the cipher makes use of a secret internal state which consists of two parts: a permutation of all 256 possible bytes (denoted S) and two 8-bit index-pointers (denoted i and j). The permutation is initialized with a variable length key, typically between 40 and 2048 bits, using the so-called key-scheduling algorithm (KSA), once this has been completed, the stream of bits is generated using the pseudorandom generation algorithm (PRGA). While many stream ciphers are based on LFSRs (efficient in hardware but less so in software), the design of RC4 avoids the use of LFSRs and is ideal for software implementation, as it requires only byte manipulations.

The basic data structure needed is an array of 256 8-bit integers, called the state vector S that is initialized with the encryption key T with the following procedure:

1. S is initialized with entries from 0 to 255 in the ascending order
2. S is further initialized with the help of a temporary 256-element vector denoted T that also holds 256 integers. The vector T is initialized with the encryption key as follows: let K be the encryption key represented as a vector 8-bit integers of size 16 (in case of a 128-bit key), i.e. K stores 16 non-negative integers whose values will be between 0 and 255, then initialize the 256-element vector T by placing in it as many repetitions of the key as necessary until T is full
3. Use the 256-element vector T to produce the initial permutation of S as in 2.13

The keystream is generated with the algorithm in 2.14

But why use $j = (j + S[i] + T[i]) \bmod 256$? Because it works in practice. It is also easy to compute. But also the way you pick indexes is biased and this introduces vulnerabilities.

Theoretical analysis shows that for a 128 bit key length, the period of the pseudorandom sequence of bytes is likely to be greater than 10^{100}, but it has been shown to be vulnerable to attacks especially if the beginning portion of the output pseudorandom byte stream is not discarded. As a consequence use of RC4 is prohibited in SSL/TLS protocol since 2015.

```
j  =  0
for i = 0 to 255
    j  =  ( j + S[i] + T[i] )  mod  256
    SWAP  S[i], S[j]
```

Figure 2.13: Key Scheduling Algorithm (KSA)

```
i, j  =  0
while ( true )
    i  =  ( i + 1 )  mod  256
    j  =  ( j + S[i] )  mod  256
    SWAP S[i], S[j]
    k  =  ( S[i]  +  S[j] )  mod  256
    output S[k]
```

Figure 2.14: Generating the pseudorandom byte stream

WiFi security started with RC4 in the WEP protocol. After it was discovered that the encryption key used in WEP could be acquired by an adversary in almost no time, WiFi security has now moved on to the WPA2 protocol that uses AES for encryption.

Unlike modern stream ciphers, RC4 does not take a separate nonce alongside the key. If a single long-term key is to be used to securely encrypt multiple streams, the protocol must specify how to combine the nonce and the long-term key to generate the stream key for RC4. One approach to addressing this is to generate a "fresh" RC4 key by hashing a long-term key with a nonce. Unfortunately, many applications that use RC4 simply concatenate key and nonce; this gives rise to related key attacks that are famous for breaking the WEP standard. Because RC4 is a stream cipher, it is more malleable than block ciphers. If not used together with other techniques (e.g., message authentication codes), then encryption is vulnerable to bit-flipping attacks.

Stream ciphers require a shared secret, a seed, which should be transmitted via a secure channel and once Alice and Bob have the seed, they can exchange via an insecure channel other parameters, like the IV and start encrypting/decrypting. The use of stream ciphers guarantees confidentiality. However, the stream cipher in itself does not guarantee that the seed has been exchanged securely, this has to be done in other ways. On the other hand, the attacker might tamper with the ciphertext and Alice/Bob will not understand immediately that the messages have been corrupted. In other words, stream ciphers provide neither authentication nor integrity.

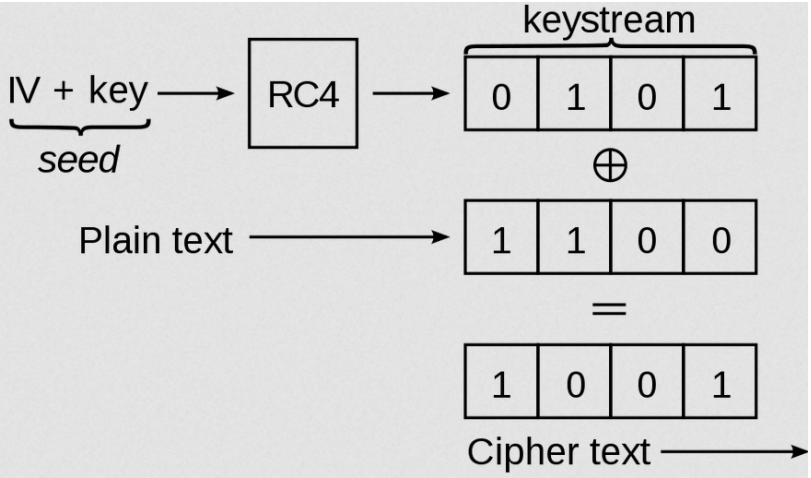


Figure 2.15

2.14 RC4 in WEP

2.14.1 WIFI Protocol

- Wired Equivalent Privacy (WEP) is the oldest protocol and has been proven to be vulnerable.
- Wi-Fi Protected Access (WPA) improved security but still vulnerable.
- Wi-Fi Protected Access II (WPA2) while not perfect is the most secure choice. It used two different types of cryptographic techniques to secure communication and those are Temporal Key Integrity Protocol (TKIP) and Advanced Encryption Standard (AES).

TKIP is used to authenticate the client and exchange messages in a secure way. Authentication is based off the Pre Shared Key, a hard wired string written on the device.

2.14.2 WIFI Authentication

Client and the station have to agree on a key used to encrypt messages using symmetric encryption. How do we share this secret key? They have to agree over an insecure channel, so we need a protocol to do this. So the Client reads this preshared key on the device to authenticate to the device, but this key won't be used to encrypt messages! Because it is shared between clients! So TKIP used a key mixing function that takes as input this preshared key and an initialization vector and passing it to RC4 cipher initalization 2.15.

2.14.3 Security of WPA2

Vulnerable to KRACK attack. It exploits a functionality where if a user is disconnected from WiFi then for reconnecting it performs an easier problem to derive a new session.

Chapter 3

Block Ciphers

3.1 The problem of stream ciphers

The problem is generating a key as long as the plaintext. In block ciphers we encrypt block of bits, typically 64 bits. We do not see the plaintext as a stream of bits, but as collection of blocks. If the plaintext is not divisible by the block size we add padding. Additionally every block is encrypted in isolation or combination and these are called modes of operation.

3.2 Introduction to block ciphers

The main idea is to replace a block of N bits from the plaintext with a block of N bits from the ciphertext.

The relationship between the input blocks and the output block is completely random. But it must be invertible for decryption to work. Thus, it has to be one-to-one, i.e. each input block is mapped to a unique output block. Usually, N=64, 128, 256. If the block size is too small, then the number of different plaintext blocks that can ever be encrypted may be too small for an attacker to launch a type of dictionary attack by building up a dictionary of plaintext/ciphertext pairs, if the block size is too large, then the block cipher becomes inefficient to operate, particularly for plaintexts smaller than the block size as they need padding. The encryption key for the ideal block cipher is the table (also called the codebook) that shows the relationship between the input and the output blocks. Think of each possible input block as one of 2^N integers and for each such integer we can specify an output N-bit block. This means that the encryption key for the ideal block cipher using N-bit blocks will be of size $N * (2^N)$. An ideal block cipher is called a random permutation. If N is 64 then the codebook will be of size $N * (2^N)$ which is around 10^{21}, but this is not practical since we can can not share such keys.

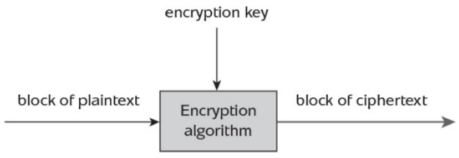


Figure 3.1

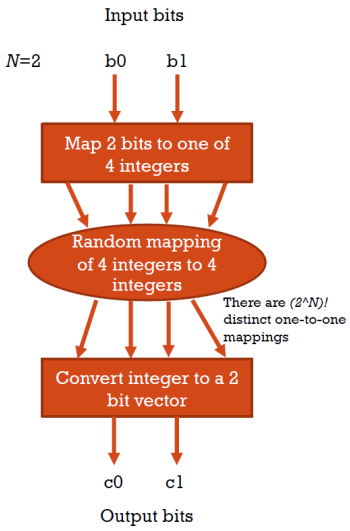


Figure 3.2

To make this practical a block cipher is a keyed family of pseudorandom permutations. For each key, we have a single permutation that is independent of all the others. Think of each key as corresponding to a different codebook and our strategy is to choose 2^K permutations uniformly at random from the set of all $(2^N)!$ permutations. We need a 1-to-1 function to map bits to their respective representation 3.2. To encrypt we simply divide the plaintext in blocks, and for each block ask the random number generator to return the ciphered block given the plaintext block. During decryption the process is reversed, round keys are used in reverse orderd.

Block ciphers are slower than stream ciphers but are generally considered more secure than the latter. Block ciphers have a property called diffusion: 2 blocks differing in a single bit shall generate 2 very different ciphertexts. So even a small transmission error will give rise to errors in around half of the bits in the plaintext. Block ciphers are the basic building block of other cryptographic primitives. Today AES is the reference choice for block ciphers.

3.2.1 Examples of block ciphers

- DES: The default block cipher of the 1980s and 1990s, but now considered broken due primarily to its small key size. The two variants based on repeated DES applications commonly known as 3DES are still respected block ciphers, although there are now faster block ciphers available.
- AES: A default block cipher based on the encryption algorithm Rijndael which won the AES design competition by NIST to identity the successor of DES
- Serpent: A respected block cipher with a block size of 128 bits and key lengths of 128, 192, or 256, which was a finalist with AES in the NIST competition. Considered slower but somehow more secure than AES but not as widely adopted

3.2.2 Unicity distance

The unicity distance of a cipher encrypting English plaintexts is the minimum of ciphertext required for a (computationally unlimited) attacker to decrypt a ciphertext uniquely (i.e. to recover the particular key used). For example the ciphertext *FJKFPO* could be *thatis,season,ofyour,...*

Intuitively, the longer the ciphertext, the fewer possible plaintexts there are that generate the ciphertext. The question is how long does a piece of ciphertext need to be, before it has only one possible decryption? The minimum length is called the unicity distance. And it turns out that the unicity distance depends on the redundancy in the plaintexts.

The unicity distance of a cipher encrypting English plaintexts is the minimum of ciphertext required for a (computationally unlimited) attacker to decrypt a ciphertext uniquely. Given WNAIW, is it possible to guess a unique plaintext that when a 1-to-1 function is applied to this plaintext we get the ciphertext? No... RIVER and WATER are valid options, but there are many more that are unreasonable ones, because they are not English words (in this case we used a substitution cipher).

We need to take into consideration that the distribution of letters is not uniform. It is redundant, "qu" is more common than "ql", so the redundancy of English has been evaluated to 3.2 bits per character. In English, the substitution cipher has a key that consists of 26 letters since the total number of keys is the number of permutations in which we can arrange 26 letters, is 26!. The amount of storage (expressed in number of bits) required for all permutations is $log(26!) = 88.28$, so in case of the substitution cipher the unicity distance is $88.28/3.2 = 27.6$. This is because the redundancy of English has been evaluated to be around 3.2 bits per character. In a system with an infinite length random key, it is possible to prove that the unicity distance is infinite.

In practice the unicity distance measures the amount of ciphertext required such that there is only one reasonable plaintext. We want the unicity distance as large as possible.

Let us generalize a bit by considering a block cipher with a N as the block size and K as the key size. The question becomes then: what is the unicity distance under a known plaintext attack? More formally: given t pairs of plaintexts with corresponding ciphertexts, which is the value of t, before it is likely that only one value of the key could have encrypted the plaintext? The answer is the unicity distance, i.e. the number of bits required to express the key space divided by the redundancy in bits per character.

Let's say that Eve intercepts n ciphertexts $c_1, c_2, \ldots$ encrypted from n plaintexts $m_1, m_2, \ldots$ using the same key. Redundancy exists in the messages. There is a value of n equal to the unicity distance such that only one value for the key recreates a possible plaintext (unless Alice and Bob use the One Time Pad).

The observation: the same block with the same key produces always the same output ciphertext independently of its position in a sequence. We will see this is the problem with AES-ECB encryption. The attacker could create a table for each plaintext-ciphertext combination, with one table for each key, but we can defend against this by changing key often, or make sure there are too many keys to try, and thus unicity distance is not reached, and increasing the length of the key to have larger unicity distance and encode with larger blocks, in AES the key size must be at least 128 bits.

Definition of unicity distance under Shannon Theorem assumptions: the minimum amount of ciphertext required to allow a computationally unlimited adversary to recover the unique encryption key. A small unicity distance does not necessarily imply that a block cipher is insecure in practice. Consider a 64-bit block cipher with a unicity distance of two ciphertext blocks: It may still be computationally infeasible for an attacker (of reasonable but bounded computing power) to recover the key, although theoretically there is sufficient information to allow this.

Diffusion means that if we change a single bit of the plaintext, then about half of the bits in

the ciphertext should change. Permutations creates diffusion. Ideally, if one bit of the plaintext is changed, then the ciphertext should change completely, in an unpredictable or pseudorandom manner.

- Avalanche criterion: Flipping a fixed set of bits should change each output bit with probability one half
- Strict avalanche criterion: For a randomly chosen input, if one flips the $i-th$ bit, then the probability that the $j-th$ output bit will change should be one half, for any i and j

Confusion means that each bit of the ciphertext should depend on several parts of the key. It refers to making the relationship between the key and the ciphertext as complex and involved as possible. Substitutions creates confusion. This can be achieved by ensuring that the system is nonlinear. Each bit of the ciphertext should depend on the entire key, and in different ways on different bits of the key. Changing one bit of the key should change the ciphertext completely.

Modern block ciphers are typically obtained by mixing substitutions and permutations to obtain both confusion and diffusion.

3.3 Anatomy of block ciphers

A block cipher is a keyed family of pseudorandom permutations. For each key, we have a single permutation that is independent of all the others. The strength of a block ciphers depends on how we design ways to choose 2^K permutations uniformly at random from the set of all $(2^N)!$ permutations. For a block cipher to be good, Eve should not be able to recover the key even using multiple plaintext-ciphertext pairs.

Then we create a substitution-permutation network that is simplest way to achieve both diffusion and confusion: the plaintext and the key often have a very similar role in producing the output, hence it is the same mechanism that ensures both diffusion and confusion. It takes a block of the plaintext and the key as inputs, and to produce the ciphertext block applies several alternating "rounds" or "layers" of substitution boxes (S-boxes) and permutation boxes (P-boxes) 3.3.

- S-Box: It substitutes a small block of bits by another block of bits. Substitution should be one-to-one, to ensure invertibility (hence decryption). An S-box is usually not just a permutation of the bits. Rather, a good S-box will have the property that changing one input bit will change about half of the output bits (with an avalanche effect). It will also have the property that each output bit will depend on every input bit. Could be thought of as a substitution cipher 3.4.
- P-Box: Permutation of all bits. Takes the output of S boxes of one round and permutes the bits and feeds them to the next S box. A good P-box has the property that the output bits of any S-box are distributed to as many S-box inputs as possible. Could be thought of as a transposition cipher. 3.5

How we define the numbers in the table 3.4? In DES it was a secret, and NSA released some details some time ago, but nobody knows all the detail and how those numbers were chosen. An S-Box may or may not be invertible. When it is, the number f input and output bit should be the same.

A straight P-Box 3.6 is invertible whereas neither a compression nor an expansion P-Box is so 3.7.

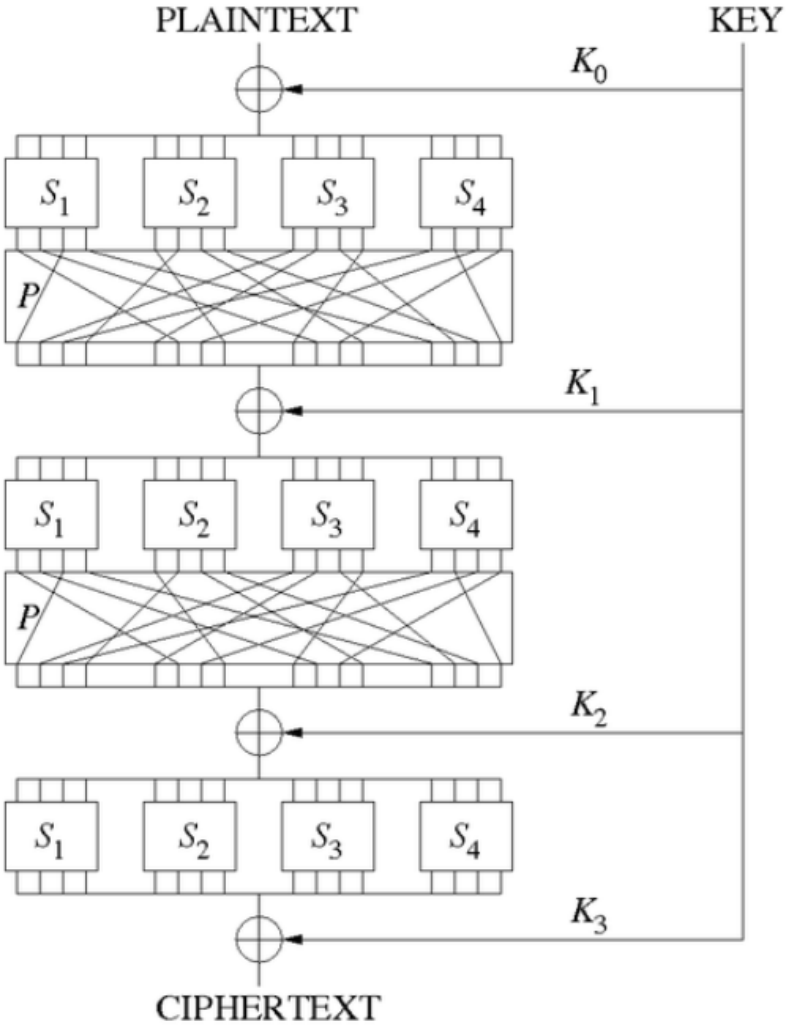


Figure 3.3: Substitution permutation network

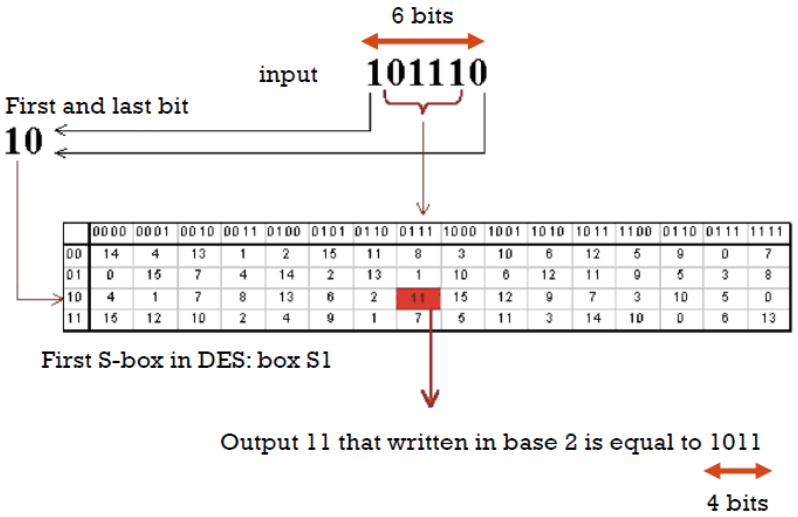


Figure 3.4: The first S-box in DES

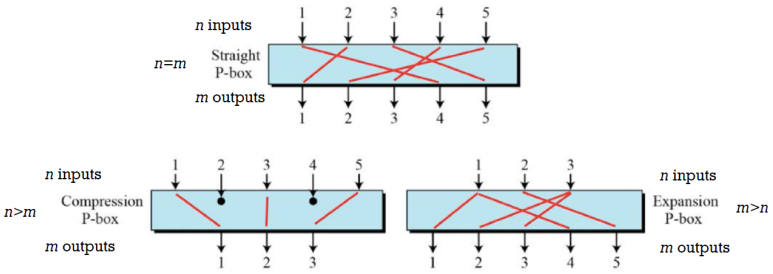


Figure 3.5: P-Box

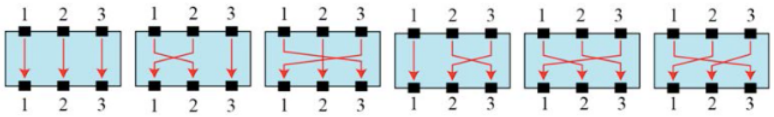


Figure 3.6: Example of P-Box straight of size 3

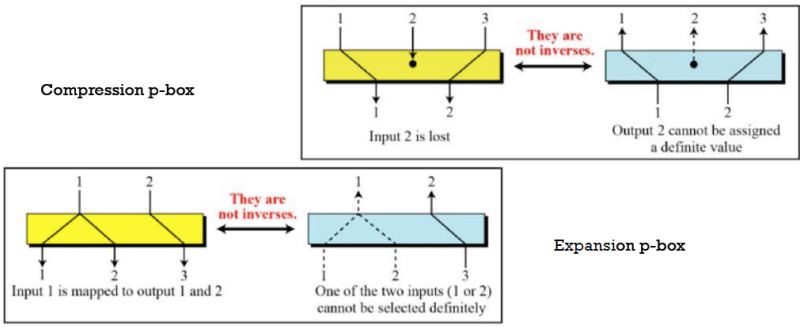


Figure 3.7: Types of P-Box

3.4 DES (Data Encryption Standard)

Based on the Feistel Structure 3.10. It uses the same basic algorithm for both encryption and decryption. It consists of multiple rounds of processing of the plaintext, with each round consisting of a substitution step followed by a permutation step 3.8. In each round:

- the right half of the block, R, goes through unchanged
- the left half, L, goes through an operation that depends on R and the encryption key, called round key, derived from the main encryption key
- the operation carried out on the left half L is referred to as the Feistel Function F
- The permutation step consists of swapping the modified L and R

Main advantage of DES is that the same hardware can be used for encryption and decryption. For decryption it is just needed to use the key in the reverse order. In DES the number of rounds is 16 times. The output of each round during decryption is the input to the corresponding round during encryption, except for the left-right switch between the two halves, and this holds true regardless of the choice of the Feistel function F. DES uses a 56-bit encryption key.

3.4.1 Feistel function

The Feistel function 3.10 is composed of various steps. First the input data is divided in two parts of 32-bits each. Then the right half RE is expanded into a 48-bit block in the Expansion permutation step by attaching a bit at the beginning and a bit at the end of each 4-bit sub-block. The expansion is needed because RE is 32-bits and the key derived by the key scheduling algorithm is 48-bits.

1. divide the 32-bit block into eight 4-bit words
2. attach an additional bit on the left to each 4-bit word that is the last bit of the previous 4-bit word
3. attach an additional bit to the right of each 4-bit word that is the beginning bit of the next 4-bit word

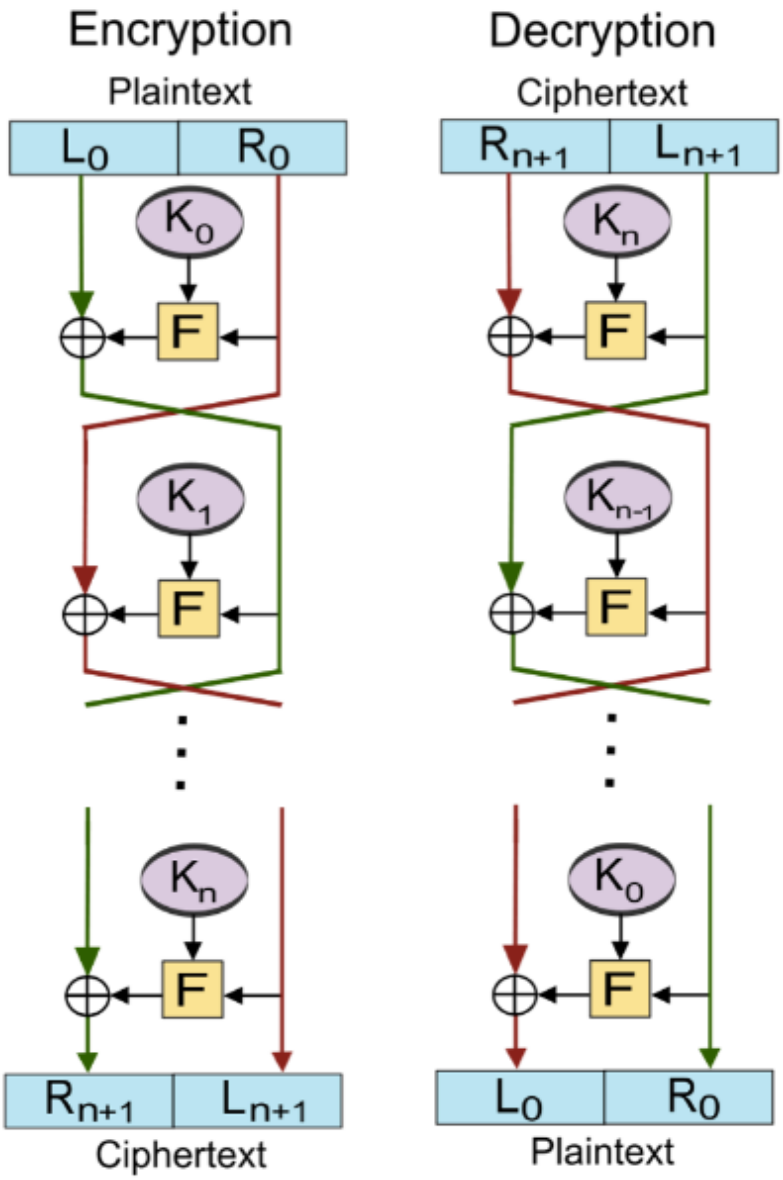


Figure 3.8: DES Overview

P-Box Permutation								
15	6	19	20	28	11	27	16	1st byte
0	14	22	25	4	17	30	9	2nd byte
1	7	23	13	31	26	2	8	4th byte
18	12	29	5	21	10	3	24	5th byte

Figure 3.9: DES P Box

The second step is called key mixing. And consists of taking the 48 bits of the expanded output produced by the Expansion permutation step and XOR it with the round key, then the output produced is broken into eight 6-bit words. Each 6-bit word goes through a substitution step; its replacement is a 4-bit word. The substitution is carried out with an S-Box. So after all the substitutions, we again end up with a 32-bit word 3.11. Each of the eight S-boxes consists of a 4×16 table lookup for an output 4-bit word. The first and the last bit of the 6-bit input word are decoded into one of 4 rows and the middle 4 bits decoded into one of 16 columns for the table lookup. The goal of the substitution carried out by an S-Box is to enhance diffusion 3.4.

After, the 32-bits of the previous step then go through a P-box based permutation. What comes out of the P-box 3.9 is then XOR-ed with the left half of the 64-bit block that we started out with. The output of this XOR operation gives us the right half block for the next round. The table 3.9 should be read as: the 0th output bit will be the 15th bit of the input, the 1st output bit the 6th bit of the input, and so on for all of the 32 bits of the output that are obtained from the 32 bits of the input.

3.4.2 Differential attacks

The S-boxes were tuned to enhance the resistance of DES to differential attacks. It is an instance of a chosen plaintext attack. Recall that in a chosen plaintext attack the attacker must be able to obtain ciphertexts for some set of plaintexts of their choosing. Differential cryptanalysis of block ciphers consists of presenting to the encryption algorithm pairs of plaintext bit patterns with known differences between them and examining the differences between the corresponding ciphertexts. Typically the notion of difference between two plaintexts or ciphertexts is the XOR of the bits performed position-wise. By feeding into a cipher several pairs of plaintext blocks with known dX (differences in the plaintexts) and observing the corresponding dY (differences in the ciphertexts), it is possible to discover parts of the round keys.

3.4.3 DES Key Scheduling

The 56-bit encryption key is represented by 8 bytes, with the least significant bit of each byte used as a parity bit. The relevant 56 bits are subject to a permutation before any round keys are generated. The table 3.12 specifies the 0th bit of the output will be the 56th bit of the input (in a 64 bit representation of the 56-bit encryption key), the 1st bit of the output the 48th bit of the input, and so on, until we have for the 55th bit of the output the 3rd bit of the input.

At the beginning of each round we divide the 56 relevant key bits into two 28 bit halves and circularly shift to the left each half by one or two bits, depending on the round, according to the table 3.13. The key permutation with the one-bit or two-bit rotation of the two key halves prior to each round aims to ensure that each bit of the original encryption key is used in roughly 14 of the 16 rounds.

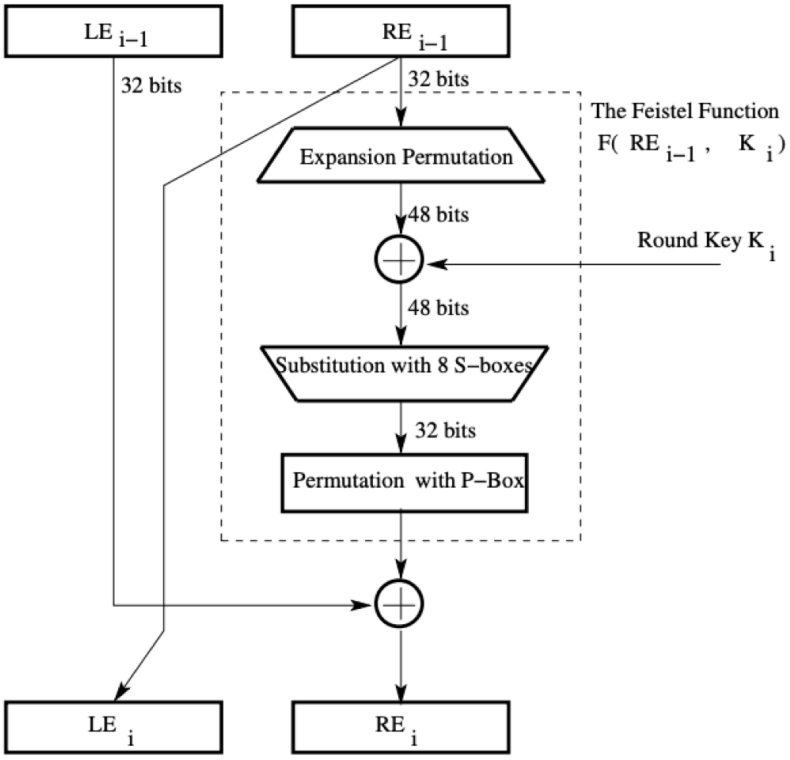


Figure 3.10: Feistel function

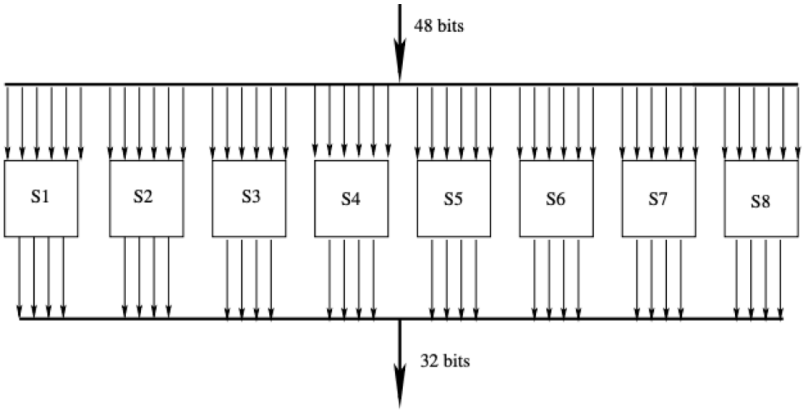


Figure 3.11: DES S Box

Initial permutation

Key Permutation 1						
56	48	40	32	24	16	8
0	57	49	41	33	25	17
9	1	58	50	42	34	26
18	10	2	59	51	43	35
62	54	46	38	30	22	14
6	61	53	45	37	29	21
13	5	60	52	44	36	28
20	12	4	27	19	11	3

Figure 3.12: DES Key Scheduling

Round number	Number of left shift
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Figure 3.13: DES Key Shift

For generating the round key, we glue together the two halves and apply a 56 bit to 48 bit contracting permutation (this is referred to as Permutation Choice 2) to the joined bit pattern. The resulting 48 bits constitute the round key 3.14. The bit addressing now spans the 0 through 55 index values for the 56 bit key. Out of this index range, the permutation shown above retains only 48 bits for the round key. Since there are only six rows and there are 8 positions in each row, the output will consist of 48 bits. The permutation order for the bits is given by reading the entries shown from the upper left corner to the lower right corner.

3.4.4 Remarks on DES

The substitution step is very effective in supporting diffusion: if one changes just one bit of the 64-bit input data block, on the average it propagates out to affect 34 bits of the ciphertext block. The manner in which the round keys are generated from the encryption key is also very effective in supporting confusion: if one changes just one bit of the encryption key, on the average that affects

Key Permutation 2							
13	16	10	23	0	4	2	27
14	5	20	9	22	18	11	3
25	7	15	6	26	19	12	1
40	51	30	36	46	54	29	39
50	44	32	47	43	48	38	55
33	52	45	41	49	35	28	31

Figure 3.14: Key Permutation 2

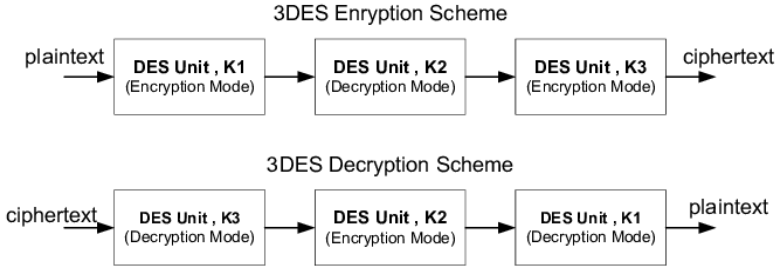


Figure 3.15: 3DES

35 bits of the ciphertext.

The 56-bit encryption key means a key space of size $2^{56} \approx 7.2 * 10^{16}$. In a brute-force attack, a machine able to process 1,000 keys per microsecond would need roughly 13 months to break the code. But the EFF in 1999 created a computer that was able to enumerate the 2^{56} keyspace and crack DES in just 10 hours. So DES was deprecated, but not immediately, before they issued a call for defining a new standard, but it took several years because cryptographers needed to scrutinize the algorithm, and for the time beign NIST issued a new directive that year that required organizations to use Triple DES, i.e. 3 consecutive applications of DES 3.15, and it is still used today!

To understand why triple DES was chosen and not just Double DES we need to take into consideration the Meet-In-The-Middle attack.

3.4.5 DoubleDES and the Meet-In-The-Middle attack

A naive approach to increase the strength of a block cipher with short keys is to use two keys k_1 and k_2 instead of one and encrypt the block twice with them. The hope is that the scheme will provide the same security of using a key of length $k_1 + k_2$. This is not the case because of the meet-in-the-middle attack where insted of performing $2^{k_1+k_2}$ guesses, the attacker only needs to perform $2^{k_1+1} = 2^{k_1} + 2^{k_2}$ guesses, let $C = ENC_{k_2}(ENC_{k_1}(P))$ and $P = DEC_{k_2}(DEC_{k_1}(C))$:

$$C = ENC_{k_2}(ENC_{k_1}(P))$$

$$DEC_{k_2}(C) = DEC_{k_2}(ENC_{k_2}(ENC_{k_1}(P)))$$

$$DEC_{k_2}(C) = ENC_{k_1}(P)$$

Now consider the two sides of the equality: $DEC_{k_2}(C) = ENC_{k_1}(P)$, the attacker can compute $DEC_{k_2}(C)$ for all possible values of k_2, then compute $ENC_{k_1}(P)$ for all possible values of k_1 for

a total of $2^{k_1} + 2^{k_2}$ or 2^{k_1+1} if k_1 and k_2 are the same size operations. If the result from any of the $ENC_{k_1}(P)$ operations matches a result from the $DEC_{k_2}(C)$ operations, the pair of k_1 and k_2 is possibly the correct key, called a candidate key. The complexity difference compared to the bruteforce method is quite significant: $2^{k_1+k_2} = 2^{k_1} * 2^{k_2}$ versus $2^{k_1+1} = 2^{k_1} + 2^{k_2}$.

3.5 Triple DES

Triple DES uses a key bundle comprising 3 DES keys (k_1, k_2, k_3), each one of 56 bits length. Encryption works as follows: $E_{k_3}(D_{k_2}(E_{k_1}(P)))$. Decryption is the inverse: $D_{k_1}(E_{k_2}(D_{k_3}(C)))$. Each triple encryption works on blocks of 64 bits of data. In each case the middle operation is the reverse of the first and last. This improves the strength of the algorithm when using keying option 2 and provides backward compatibility with DES with keying option 3.

3.5.1 Choosing Keys for Triple DES

Choosing keys; Care should be put in selecting the keys in key bundle

- All three keys are independent. Strongest option, still vulnerable to meet-in-the-middle but requires $2^{2*56} = 2^{112} \approx 5.19 * 10^{29}$ operations
- k_1 and k_2 are independent but $k_3 = k_1$. Similar to double DES and vulnerable to the same attack with equal complexity, deprecated
- All three keys are equal. For backward compatibility with DES, since two operations cancel out, forbidden because equal to DES

Additionally, some specific values for keys are forbidden. With these restrictions, Triple DES has been reapproved with keying options 1 and 2 only although it is current best practice to use only option 1 as keys should be generated by using random generators.

3.6 AES (Advanced Encryption Standard)

Notified by NIST as a standard in 2001, it has a block length 128bits. The key scheduling (how you derive the keys in each round) changes based on the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bits keys. Except for the last round, all other rounds are identical.

Unlike DES, AES is an example of key-alternating block ciphers: each round first applies a diffusion-achieving transformation operation to the entire incoming block, which is then followed by the application of the round key to the entire block. DES is based on the Feistel structure in which, for each round, one-half of the block passes through unchanged and the other half goes through a transformation that depends on the S-boxes and the round key. Key alternating ciphers lend themselves well to theoretical analysis of the security of the ciphers.

3.6.1 AES Round: Encryption

Each round includes

1. one single-key based substitution step

2. a row-wise permutation step
3. a column-wise mixing step
4. the addition of the round key

The order of these steps is different for encryption and decryption. Each round takes an input state array and returns an output state array. The output state array produced by the last round is rearranged into a 128-bit output block. Unlike DES, decryption differs substantially from encryption although similar transformations are used. One bit change of the input introduces dramatic changes in the output ciphertext.

DES is based on the Feistel (substitution-permutation) network. DES involves substitutions and permutations. Permutations are based on the Feistel notion of dividing the input block into two halves, processing each half separately, and then swapping the two halves.

AES uses a substitution-permutation network in a more general sense. Each round in AES involves key-level substitutions followed by word-level permutations. Like DES, AES is an iterated block cipher in which plaintext is subject to multiple rounds with each round applying the same overall transformation function to the incoming block.

3.6.2 Structure of AES

The 128 bits key is arranged in the form of an array of $4 * 4$ keys. The four column words of the key array are expanded into a schedule of 44 words. Each round consumes four words from the key schedule (recall that AES has 10 rounds with a key length of 128 bits). The first 4 words are used for adding to the input state array before any round can begin. The remaining 40 words are used for the 10 rounds. Recall that key expansion generates 44 keys. The first 4 are mixed with the input blocks (AddRoundKey).

- Encryption: the input state array is XOR-ed with the first four words of the key schedule.
- Decryption: same as above except that the ciphertext state array is XOR-ed with the last four words of the key schedule

The remaining 40 are used in each one of the 10 rounds 4 at a time. We are using tables in AES, and in decryption the inverse of those tables.

3.6.3 Structure of a round

There are some encryption steps: Substitute bytes, Shift rows, Mix columns, Add round key, and decryption steps: Inverse shift rows, Inverse substitute bytes, Add round key, Inverse mix columns. The final round for encryption does not involve the Mix columns step. The final round for decryption does not involve the Inverse mix columns step. In the case of AES-128, the 128 bits are organized as a 4x4 array of cells, where each cell is made up of eight bit.

3.6.4 Round Steps

Round Step 1

`SubBytes` for byte-by-byte substitution during encryption. `InvSubBytes` for the corresponding substitution during decryption. It consists of using a 16x16 lookup table to find a replacement byte for a given byte in the input state array. The entries in the lookup table are created by using the notions

	0	1	2	3	4	5	6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
1	10	11	12	13	14	15	16	17	18	19
	...									

Figure 3.16: Round Step 1

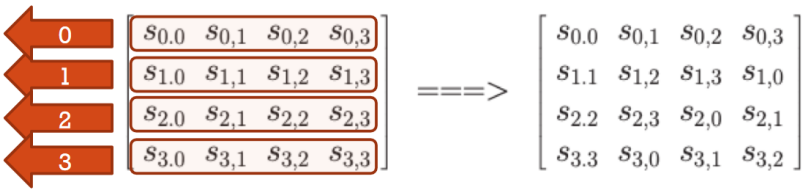


Figure 3.17: Round step 2

of multiplicative inverses in $GF(2^8)$ and bit scrambling to eliminate the bit-level correlations inside each byte 3.16.

The table is filled in a precise way, in DES it was a secret. Replace the value in each cell by its multiplicative inverse in $GF(2^8)$ based on the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. 00 has no multiplicative inverse so it is just replaced by itself. With the mapping considered above, the 00 input byte is mapped to c i.e. 0x63 and, at the same time, all other bytes are mapped one-to-one. Why 0x63? Because it works in practice. The table created must be thought of as an S-Box. The main requirement is that the table must be invertible for decryption.

Round Step 2

Remember that the state of AES is a matrix. `ShiftRows` for shifting the rows of the state array during encryption. `InvShiftRows` for the corresponding transformation during decryption.

ShiftRows transformation consists of 3.17:

1. not shifting the first row of the state array at all
2. circularly shifting the second row by one byte to the left
3. circularly shifting the third row by two bytes to the left
4. circularly shifting the last row by three bytes to the left

We are scrambling! Permuting data. For decryption, the corresponding step shifts the rows in exactly the opposite direction.

Round Step 3

`MixColumns` for mixing up of the bytes in each column separately during encryption. `InvMixColumns` for the corresponding transformation during decryption. This is the 3rd step in encryption and 4th in decryption! This step replaces each byte of a column by a function of all the bytes in the same column. The combination of the shift-rows and the mix-column steps causes each bit of the ciphertext to depend on every bit of the plaintext after 10 rounds of processing. In DES, one bit of

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

Figure 3.18: Round step 3

plaintext affected roughly 31 bits of ciphertext. In AES, the goal is that each bit of the plaintext will affect every bit position of the ciphertext block of 128 bits. The operation in 3.18 should be read as: each byte in a column is replaced by two times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows. The operation must be carried out in $GF(2^8)$, For instance, 'two times' means multiplication by bit pattern 000000010 (x) and 'three times' by bit pattern 00000011 ($x+1$). The words 'next' and 'follow' refer to bytes in the same column, and their meaning is circular, in the sense that the byte that is next to the one in the last row is the one in the first row. This fixed matrix is invertible over $GF(2^8)$ so that the entire transformation is invertible.

Round Step 4

`AddRoundKeys` for adding the round key to the output of the previous step during encryption. This is the 4th step in encryption and 3rd in decryption! `AddRoundKey` or `InvAddRoundKey` for inverse add round key transformation during decryption.

3.6.5 Key Expansion

Each round has its own round key that is derived from the original 128-bit encryption key as described in the following. One of the four steps of each round, for both encryption and decryption, involves the XOR-ing of the round key with the state array. The AES Key Expansion algorithm is used to derive the 128-bit round key for each round from the original 128-bit encryption key. The logic of the key expansion algorithm is designed to ensure that if one changes one bit of the encryption key, it should affect the round keys for several rounds.

In the same manner as the 128-bit input block is arranged in the form of a state array, the algorithm first arranges the 16 bytes of the encryption key in the form of a 4 x 4 array of bytes 3.19.

The first four bytes of the encryption key constitute the word w_0. The next four bytes the word w_1. The next four bytes the word w_2. The last four bytes the word w_3. The algorithm subsequently expands the words w_0, w_1, w_2, w_3 into a 44-word key schedule $w_0, w_1, \ldots, w_{42}, w_{43}$ 3.20.

But how does the Key Expansion Algorithm expands four words w_1, w_2, w_3, w_4 into 44 words $w_0, w_1, \ldots, w_{42}, w_{43}$? On a high level the key expansion algorithm works like this: the key expansion takes place on a 4-word to 4-word basis, in the sense that each grouping of 4 words decides what the next grouping of 4 words will be. Let's say we have four words of the round key for the i-th round: $w(i), w(i+1), w(i+1), w(i+3)$ we want to generate $w(i+4), w(i+5), w(i+6), w(i+7)$. Function g takes the four words and returns a single word that is then XOR-ed with each word and we get the words of the next round 3.21. Except for the first word in a new 4-word grouping, each word is obtained by XOR-ing the previous word and the corresponding word in the previous 4-word grouping. Mathematically, the transformation is expressed as follows:

- $w_{i+4} = g(w_{i+3}) \oplus w_i$

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

Figure 3.19: Key Expansion Algorithm

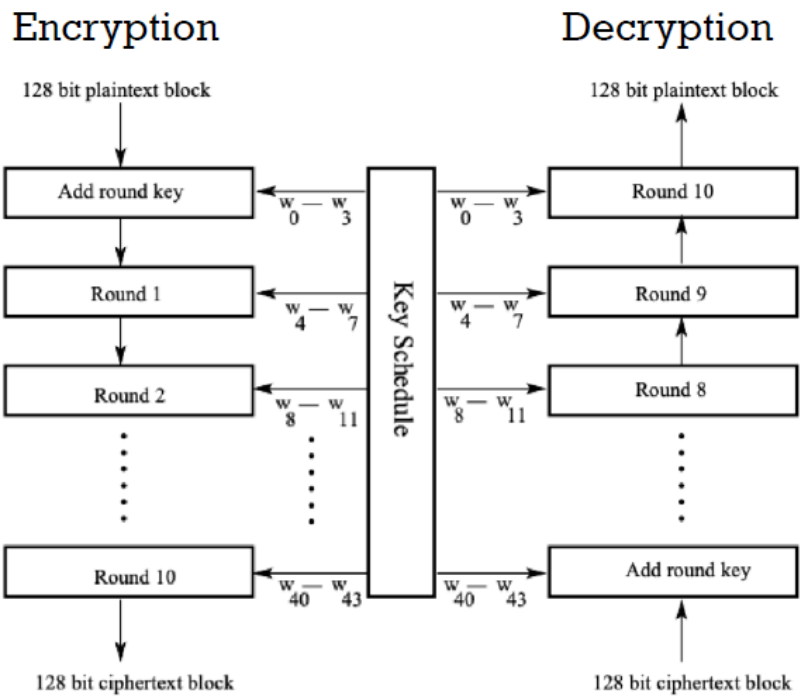


Figure 3.20: Key Expansion Algorithm

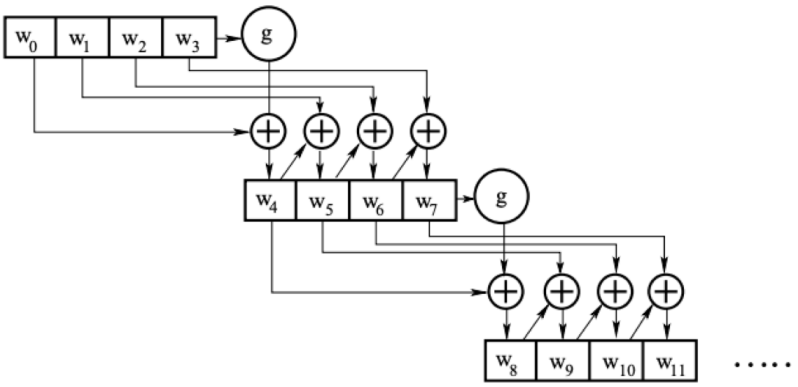


Figure 3.21: Key Expansion Algorithm

- $w_{i+5} = w_{i+4} \oplus w_{i+1}$
- $w_{i+6} = w_{i+5} \oplus w_{i+2}$
- $w_{i+7} = w_{i+6} \oplus w_{i+3}$

The function g consists of the following three steps:

1. Perform a one-byte left circular rotation on the argument 4-byte word
2. Perform a byte substitution for each byte of the word returned by the previous step by using the same 16 x 16 lookup table as used in the SubBytes step of the encryption rounds
3. XOR the bytes obtained from the previous step with a round constant. The round constant is a word whose three rightmost bytes are always zero and XOR-ing with the round constant amounts to XOR-ing with just its leftmost byte

The key expansion algorithm ensures that AES has no weak keys.

3.6.6 Security of AES

Cryptographers are constantly probing AES for weaknesses. This is essential, because if it was not being thoroughly tested by academics, then criminals or nation states could eventually find a way to crack it without the rest of the world knowing.

In 2009, a series of related-key attacks were discovered. These involve observing how a cipher operates under different keys. These attacks are only possible against protocols that are not implemented properly.

In 2009, there was a known-key distinguishing attack against an eight round version of AES-128. This attack uses a key that is already known in order to figure out the inherent structure of the cipher. As this attack was only against an eight round version, there is not much to worry about for everyday users of AES-128.

So far, researchers have only uncovered theoretical breaks and side channel attacks. This means that AES is essentially unbreakable at the moment.

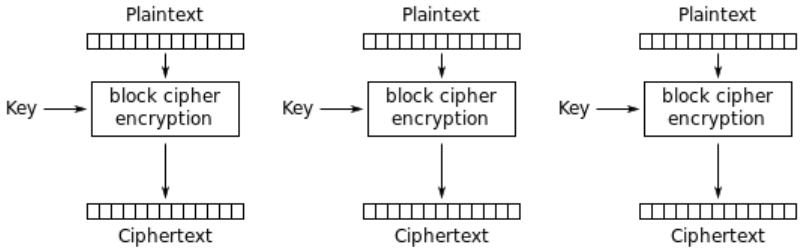


Figure 3.22: ECB Encryption

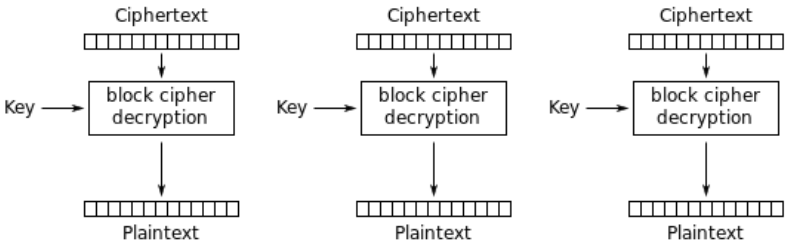


Figure 3.23: ECB Decryption

3.7 Modes of operation for block ciphers

3.7.1 Electronic codebook (ECB)

When a block cipher is used in ECB mode, each block of plaintext is coded independently. This makes it not very secure for long segments of plaintext, especially plaintext containing repetitive information. Primarily used for secure transmission of short pieces of information, e.g., encryption keys. When each block of a plaintext file is encrypted independently of the other blocks, the structure of the information in the ciphertext file can hold important clues to what is in the plaintext file. Given a key, a block cipher always encrypts the same contents the same way. At first, this does not seem to be a problem because the output is still encrypted but it can reveal information, for example a picture 3.24 even if encrypted with ECB the repeating patters still show, in the picture we can clearly distinguish the text and the background. ECB mode block encryption can leave too many clues in the ciphertext for an attacker. As a consequence, the ECB mode is good only for short messages or messages without too much repetitive structure Another shortcoming of ECB is that the length of the plaintext message must be integral multiple of the block size, if this condition is not met the plaintext must be padded 3.22 3.23 .

3.7.2 Cipher block chaining (CBC)

To overcome the security problems of the ECB mode, the input to the encryption algorithm consists of the XOR of the plaintext block and the ciphertext produced from the previous plaintext block. This makes it difficult to identify patterns in the ciphertext that may correspond to the known

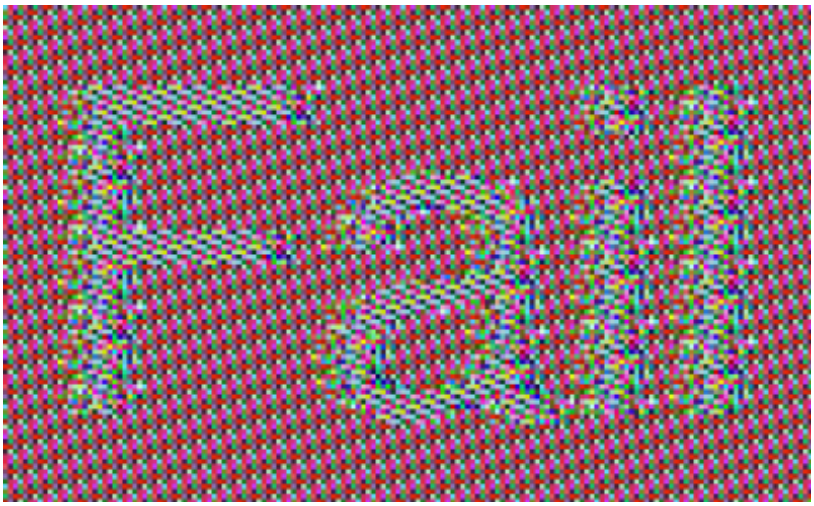


Figure 3.24: ECB Fail

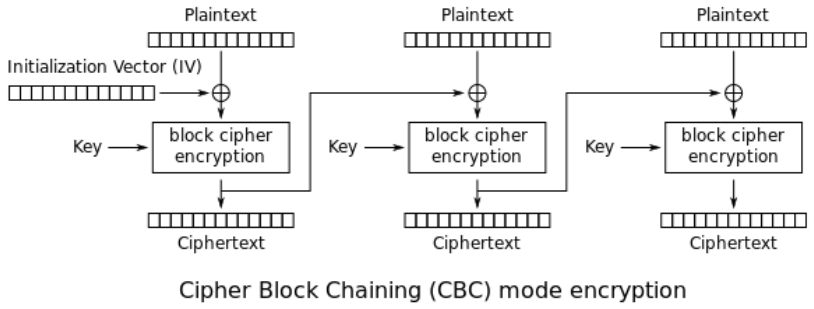


Figure 3.25: CBC Encryption

structure of the plaintext. The chaining scheme shown in the figure needs an Initialization Vector (IV) for the first invocation of the encryption algorithm. The IV is sent separately as a short message using the ECB mode. With CBC, the ciphertext block for any given plaintext block becomes a function of all the previous ciphertext blocks 3.25 3.26.

3.7.3 Counter (CTR)

Counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting successive values of a "counter". The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. Notice that only the encryption algorithm is used in both encryption and decryption. This can be an important implementation-level detail for those block ciphers for which the encryption and the decryption algorithms are significantly different such as AES 3.27 3.28.

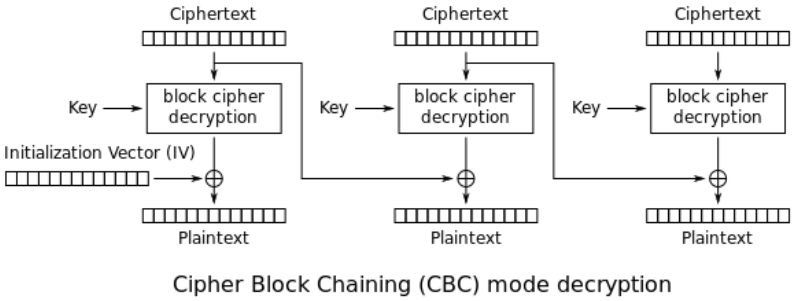


Figure 3.26: CBC Decryption

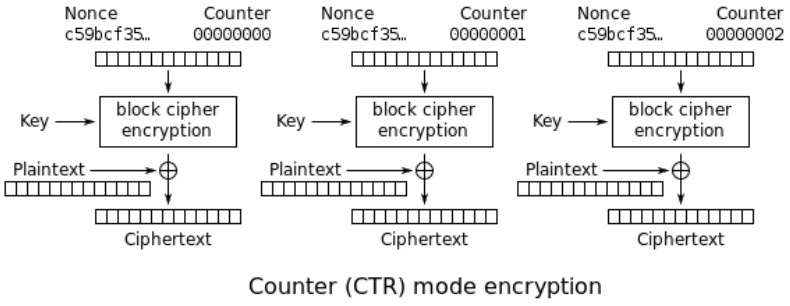


Figure 3.27: CTR Encryption

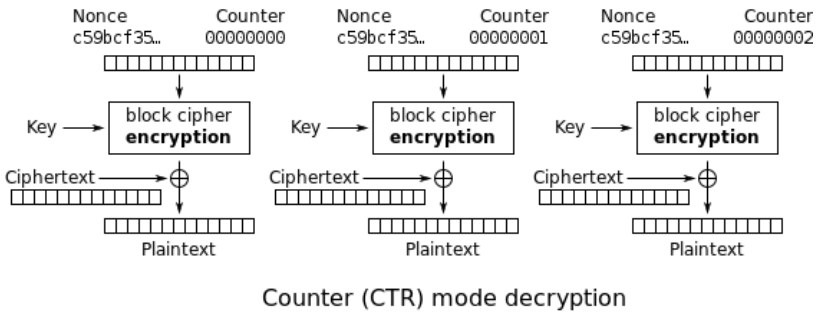


Figure 3.28: CTR Decryption

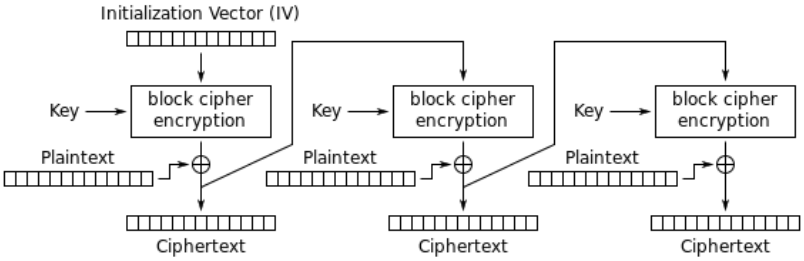


Figure 3.29: CFB Encryption

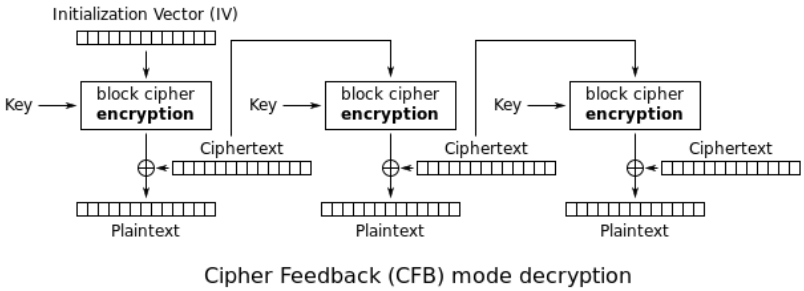


Figure 3.30: CFB Decryption

3.7.4 Cipher feedback (CFB)

The cipher feedback (CFB) mode, in its simplest form uses the entire output of the block cipher. In this variation, it is very similar to CBC, makes a block cipher into a self-synchronizing stream cipher 3.29 3.30. Notice that only the encryption algorithm is used in both encryption and decryption.

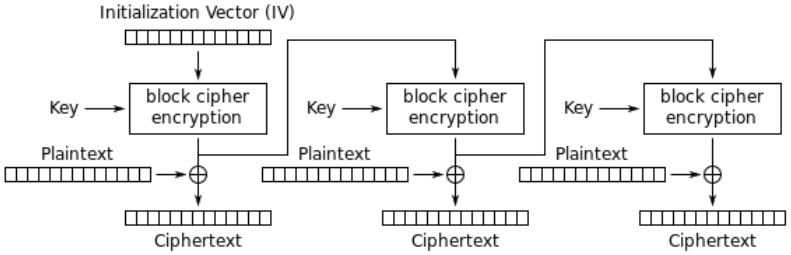
3.7.5 Output feedback (OFB)

The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then XOR-ed with the plaintext blocks to get the ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error-correcting codes to function normally even when applied before encryption 3.31 3.32. Notice that only the encryption algorithm is used in both encryption and decryption.

3.7.6 Galois/counter mode (GCM)

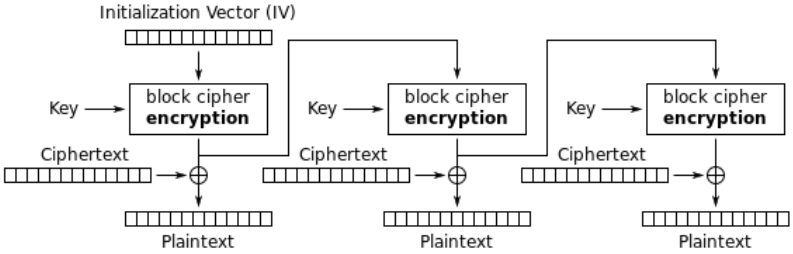
Galois/counter mode (GCM) combines the well-known counter mode of encryption with the new Galois mode of authentication. The key feature is the ease of parallel computation of the Galois field multiplication used for authentication. This feature permits higher throughput than encryption algorithms and GCM is defined for block ciphers with a block size of 128 bits. Widely used today in many protocols such as TLS.

Like in CTR, blocks are numbered sequentially. The block number is combined with an IV and encrypted with a block cipher (usually AES). The result of this encryption is XOR-ed with the



Output Feedback (OFB) mode encryption

Figure 3.31



Output Feedback (OFB) mode decryption

Figure 3.32

plaintext to produce the ciphertext 3.33. Like all counter modes, this is essentially a stream cipher, and so it is essential that a different IV is used for each stream that is encrypted. The ciphertext blocks are considered coefficients of a polynomial which are then manipulated in the corresponding Galois field. The result is then encrypted, producing an authentication tag that can be used to verify the integrity of the data.

3.8 DES vs AES

The basic difference between DES and AES is that the block in DES is divided into two halves before further processing whereas in AES entire block is processed to obtain ciphertext. The DES algorithm works on the Feistel Cipher principle and the AES algorithm works on substitution and permutation principle. The key size of DES is 56 bit which is comparatively smaller than AES which has 128,192, or 256-bit secret key. The rounds in DES include Expansion Permutation, XOR, S-Box, P-Box, XOR and Swap whereas rounds in AES include Subbytes, Shiftrows, Mix columns, Addroundkeys. DES is less secure than AES because of the small key size and the relatively small block size. AES is comparatively faster than DES.

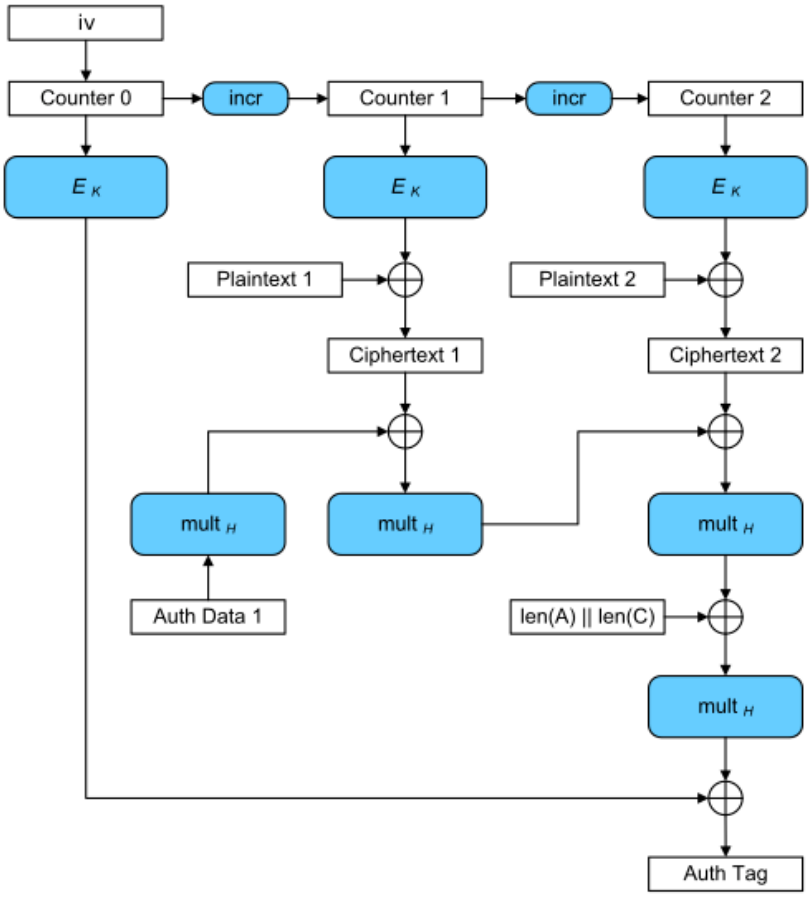


Figure 3.33: GCM Mode

Chapter 4

Cryptographic Protocols and Key Distribution

4.1 Communication Protocols

It is a system that allows two or more entities to exchange information. It defines the rules, syntax, semantics and synchronization of communication and possible error recovery methods. It can be implemented by hardware, software, or a combination of both. It uses messages with well-defined formats (syntax) and each message has an exact meaning intended to elicit a response from a range of possible responses pre-determined for that particular situation (semantics). The specified behaviour is typically independent of how it is to be implemented.

4.2 Cryptographic Protocol

It is a communication protocol (usually a small one, meaning that its specification is typically quite compact, e.g., few exchange of messages in given formats) designed to secure communication (various security goals) by using cryptographic primitives (e.g., ciphers, hash functions, ...).

Typical security goals/properties

- Secrecy: May an intruder learn some secret message between the two honest participants Alice and Bob?
- Authentication: Is the agent Alice really talking to Bob?
- Non-repudiation: Alice sends a message to Bob. Alice cannot later deny having sent this message. Similarly, Bob cannot deny having received the message.

What do we mean by "The specified behavior [of a protocol] is typically independent of how it is to be implemented"? Observe that you really want to specify a protocol independently of its implementation because you want to abstract away a lot of details that are present in an implementation: type of processors used by the entities involved in the protocol, operating systems, programming language, libraries. We focus only on the essence of the cryptographic protocol, understand if it satisfies the security goal, and then consider the problem of implementing it. The problem admits at least two possible solutions that have in common one feature, namely that the syntax and semantics

of (cryptographic) protocols together with their security goals can be precisely described by using appropriate mathematical objects. In this way, it is possible to prove (in a mathematical way) that a certain protocol achieves a security goal under the assumption that a characterization of the attacker capabilities is also included in the specification of the protocol. Typically, the attacker capabilities that are considered in the context of cryptographic protocols are the following: an (active) attacker can:

- intercept all messages sent on the network
- compute messages
- send messages on the network

So we say that the attacker is the network, or that the attacker carries the message. Session: is set of message exchange between entities that relates logically to the same protocol specification. We'll see that a protocol can be instantiated to a specific session. We characterize the attacker based off of how can he interfere with the properties of the cryptographic protocol. We take into consideration the Dolev-Yao (or Formal) adversary model. Unlike in the real world, the adversary can neither manipulate the encryption's bit representation nor guess the key. The attacker may, however, re-use any messages that have been sent and therefore become known. The attacker can encrypt or decrypt these with any keys he knows, to forge subsequent messages.

In this model we assume that the cryptographic primitives are black-boxes and that the messages are expressions built out of the cryptographic primitive. Additionally we say that cryptographic primitive cannot be broken or, equivalently, we assume that cryptography is perfect.

Needless to say this is an abstraction and maybe a coarse abstraction that ignores all the weaknesses that may be existing in available cryptographic primitives. This implies that a proof of the security of a certain protocol in this model holds under the assumption that the attacker does not attempt to break the cryptographic primitives and, of course, that the protocol implementation correctly implements its specification.

There are other models, such as the computation model that assumes the following:

- The messages are bit-strings
- The cryptographic primitives are functions on bit-strings
- The attacker is any probabilistic (polynomial-time) Turing machine

A probabilistic Turing machine is a non-deterministic Turing machine that chooses between the available transitions at each point according to some probability distribution (roughly by tossing a coin at each step in the computation).

The main difference with the Dolev-Yao model is that cryptographic primitives are no more perfect but are functions operating on bit-strings which is a less coarse abstraction; however, notice that it is still an abstraction since bit-strings are mathematical objects that are quite different from the bit-strings datatypes that we may find in programming languages. A probabilistic (polynomial-time) Turing machine characterizes the capabilities of a computationally constrained attacker that can solve problems using randomized polynomial time algorithms including those required to break a certain cryptographic primitive. Despite being more realistic than the Dolev-Yao model, the computational model still ignores several details that should be considered when implementing a cryptographic protocol including side channel attacks such as: timing, power consumption, noise, physical attacks against smart card. A side-channel attack aims to gather information from or influence the execution of a system by measuring or exploiting indirect effects of the system or its

hardware rather than targeting the program or its code directly. In other words, side channel attacks do not exploit weaknesses in the implemented system but rather gain information from the execution of the system itself.

An attack in the Dolev-Yao model implies a (practical) attack in the computational model. A proof in the Dolev-Yao model does not always imply a proof in the computational model. The Dolev-Yao model allows for automated verification. Most proofs in the computational model are manual.

4.3 Security issues in protocols because of symmetric ciphers

Assume that a large number of people, processes, or systems that want to communicate with one another in a secure fashion and that this group of people/processes/systems is not static, i.e. individual entities may join or leave the group at any time. The naive solution would be to have point-to-point key establishment, so each party physically exchange an encryption key with every one of the other parties, but these way we need a quadratic $N * (N-1)/2$ number of keys. But each entity has many keys to manage and has to manage them in an appropriate way. This is not feasible for large groups, especially if the group membership evolves over time and keys have to be rotated frequently.

Secondly we need a way of exchanging the first key that then can be used by the two parties to communicate securely. So we need to introduce the notion of session/ephimeral key. Point-to-point key establishment has a few pros and cons: in terms of security if subject is compromised only its communications are compromised; communications between two other subjects are not compromised, on the other side it has poor scalability as the number of keys is quadratic in the number of subjects and has poor handling of dynamic topologies: a new member joining and a member leaving affect all current members. For large networks this becomes impractical and we need to design an alternative solution based on a trusted 3rd party that mediates the establishment of a secure connection.

4.4 Key Distribution Center

Key distribution as a trusted 3rd party. Each entity needs to change a master key with the key distribution center, so we need only a linear number of keys. Provide every group member with a single key, called the master key, for securely communicating with a key distribution center (KDC). When A(lice) wants to establish a secure communication link with B(ob), A(lice) requests a session key from KDC for communicating with B(ob). But there are a few issues we need to solve:

- Assuming that A is the initiator of a session-key request to KDC, when A receives a response from KDC, how can A be sure that the sending party for the response is indeed the KDC?
- Assuming that A is the initiator of a communication with B, how does B know that some other party is not masquerading as A?
- How does A know that the response received from B is indeed from B and not from someone else masquerading as B?
- What should be the lifetime of the session key acquired by A for communicating with B?

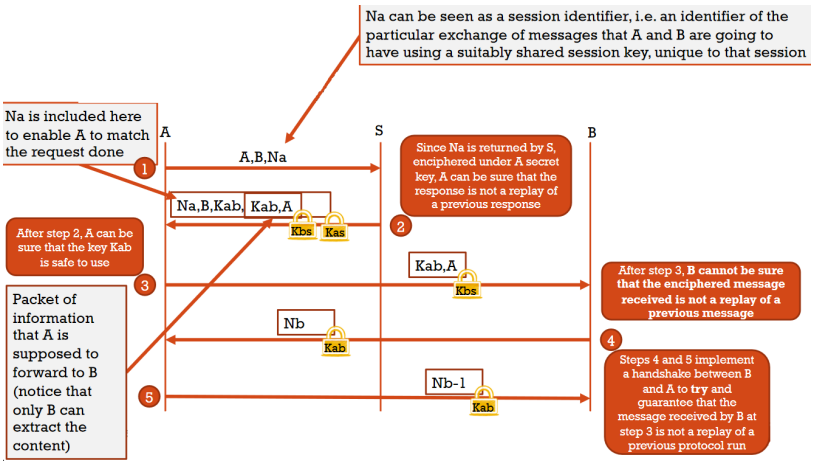


Figure 4.1: Needham–Schroeder Protocol

4.4.1 Needham-Schroeder key distribuition

A party A wants to establish a secure communication link with another party B 4.1. Both A and B possess master keys K_{AS} and K_{BS}, respectively, for communicating privately with a key distribution center (KDC). Needham-Schroeder is a shared-key authentication protocol designed to generate and propagate a session key, i.e., a shared key for subsequent symmetrically encrypted communication. Notice that there is no public key infrastructure in place and we've made the assumption that both A and B have a shared communication key with S. It is crucial that the key shared between A and S (K_{AS}) is protected otherwise anyone can impersonate A (similarly for B). S is the server that is the trusted 3rd party in this communication and it allows pair of users to establish a session key. Each user shares a long-term key, a priori with S. The overall number of long-term keys is linear in the number of users. S maintains a database containing pairs associating an identifier of a user and the key shared between the user and S. It has to be considered as an honest participant of the key distribution protocol. This creates a single point of failuere so we need a way of protecting keys stored in the KDC. Usually an hierarchical structure of KDCs is preferred, because it limits the damage of a faulty KDC.

Needham-Schroeder uses nonces (numbers used once), that are randomly generated values included in messages. If a nonce is generated and sent by A in one step and returned by B in a later step, A knows that B message is fresh and not a replay from an earlier exchange, the only assumption is that it has not been used in any earlier interchange. The nonce generated by A will be noted as N_A and the one from B will ne noted as N_B. The nonce can be seen as a session identifier, i.e. an identifier of the particular exchange of messages that A and B are going to have using a suitably shared session key, unique to that session. K_{AB} is a symmetric, generated key, which will be the session key of the session between A and B.

The procedure has these steps:

1. A contacts S, by sending its identifier A, the identifier of B and then a nonce N_A.
2. S responds by sending N_A, B, and creates K_{AB}. Then adds K_{AB} and A encrypted with K_{BS} and encrypts the whole message with K_{AS}.
3. A then sends K_{AB} and A encrypted with K_{BS} to B. At this point B knows that A wants to

communicate with them, and also knows the session key K_{AB}. Then B sends N_B encrypted with K_{AB} to A.

4. A sends $N_B - 1$ encrypted with the session key K_{AB} to B.

This protocol is vulnerable to an attack called Replay attack where if the attacker compromises K_{AB} it can then send the message K_{AB} and A encrypted with K_{BS} to B, who who will accept it, being unable to tell that the key is not fresh, because it doesn't contain nonces. At this point B believes that A sent that message all will send back N_B encrypted with K_{AB} to the attacker that will answer accordingly. Now the attacker has successfully impersonated A.

This flaw is fixed in the Kerberos protocol by the inclusion of a timestamp.

1. A contacts S, by sending its identifier A, the identifier of B.
2. S responds by sending T, B, and creates K_{AB}. Then adds K_{AB}, A, and T encrypted with K_{BS} and encrypts the whole message with K_{AS}.
3. Then A sends K_{AB}, A, and T encrypted with K_{BS} to B.

But the clock on each machine can drift, and introduces the problem again. There needs to be an infrastructure to synchronize clocks! This does not scale easily, but it is fine with few devices.

4.5 Kerberos

Kerberos operates on the principle of shared secret keys. Its goal is to establish a direct authenticated and confidential communication link between the host (where a print request originates) and the printer. Since printers generally are rudimentary when it comes to general purpose computing, you may not expect a printer to contain all of the software that generally is required (such as the SSL/TLS libraries) for establishing such links.

In Kerberos there is a service which sole purpose is to authenticate. Both users and services implicitly trust the Kerberos Authentication Service (AS) that mediates their interaction. For this to work, both the user and the service must have a shared secret key registered with the AS. Such keys are called long-term keys, since they last for weeks or months.

There are three basic steps involved in authenticating a user to an end service

- The user U sends a request to the AS, asking it to authenticate U to the service S. This request consists only of the service's name, although in practice, it contains some other information (recall the nonce in the Needham-Schroeder protocol).
- The AS prepares to introduce the user U and the service S to each other, it generates a new, random secret key that will be shared only by U and S and it sends the user a two-part message, one part contains the random key along with the service's name, encrypted with the user's long-term key, the other part contains that same random key along with the user's name, encrypted with the service long-term key. Notice that at this point, only the user U knows the session key, provided it really is the user U and knows the appropriate long-term key.
- User U generates a fresh message (called authenticator) and encrypts it with the session key, then sends the authenticator, along with the ticket, to the service. The service S decrypts the ticket with its long-term key to recover the session key. In turn, the session key is used to decrypt the authenticator. The service S trusts the AS, so it knows that only the legitimate user could have created such an authenticator and this completes the authentication of the

Key	Description
Kcl	secret key held by AS for the Client
Ktgs	• secret key held by AS for TGS • TGS also has this key
Ksp	• secret key held by AS for the Service Provider (SP) • SP also has access to this key
Kcl-tgs	**session key** that AS send to the Client for communicating with TGS
Kcl-sp	**session key** that TGS send to the Client for communicating with the SP

Figure 4.2

user to the service. If the user U want the service S to be authenticated in return, then the service S takes N_{ab} from the authenticator, adds the service's own name to it, and encrypts the whole message with the session key. This is then returned to the user U.

The main difference between the Needham-Schroeder protocol and the Kerberos protocol is that the latter makes a distinction between the clients, on the one hand, and the service providers, on the other.

One of the inconveniences of using a password is that each time you access a service, you have to type it in. Kerberos resolves the last problem by introducing a new service, called the ticket granting server (TGS). The purpose of the TGS is to add an extra layer of indirection so that the user only needs to enter a password once and the ticket and session key obtained from that password are used for all further tickets 4.3. The client can not communicate directly with the TGT, it has to ask the KDC for a Ticket Granting Ticket (TGT). Afer having this ticket it can request any service ticket, and communicate with that service with that ticket 4.4 4.2 4.5.

1. Client sends a request in plain text to the AS, this request is for a service that the Client expects from the SP

2. AS sends back to the Client the following two messages encrypted with K_{cl}. In the database maintained by AS, K_{cl} is specific to the Client and will remain unchanged as long as the client does not alter its password. This encryption key is not directly known to the Client. The two messages are: a session key K_{cl-tgs} that the client can use to communicate with TGS and a Ticket-Granting Ticket (TGT) that is meant for delivery to TGS including the client user ID ($clID$), client network address ($clIP$) a validation time (VT) and the same K_{cl-tgs} session key. The ticket is encrypted with the K_{TGS} secret key that the AS server maintains for TGS

3. After step 3 and before step 4 the client is asked to provide the password in a dialog box. Then the password is converted into the K_{cl} if it is correct. This key is then used to extract the session key K_{cl-tgs} and the ticket meant for TGS from the information received from AS.

4. In steps 4 and 5 the client sends the following two messages to TGS: the encrypted ticket meant for TGS followed by the ID of the requested service and a Client Authenticator that is composed of the client ID and the timestamp, both encrypted with the K_{cl-tgs} session key.

5. Before steps 6 and 7 TGS recovers the ticket from message 4 above, from the ticket, it recovers the K_{cl-tgs} session key. The TGS then uses the session key to decrypt message 5 above that allows it to authenticate the Client

6. in steps 6 and 7 TGS sends back to the Client the following two messages: a Client-to-Service Provider ticket that consists of the Client ID ($clID$), the Client network address ($clIP$), the

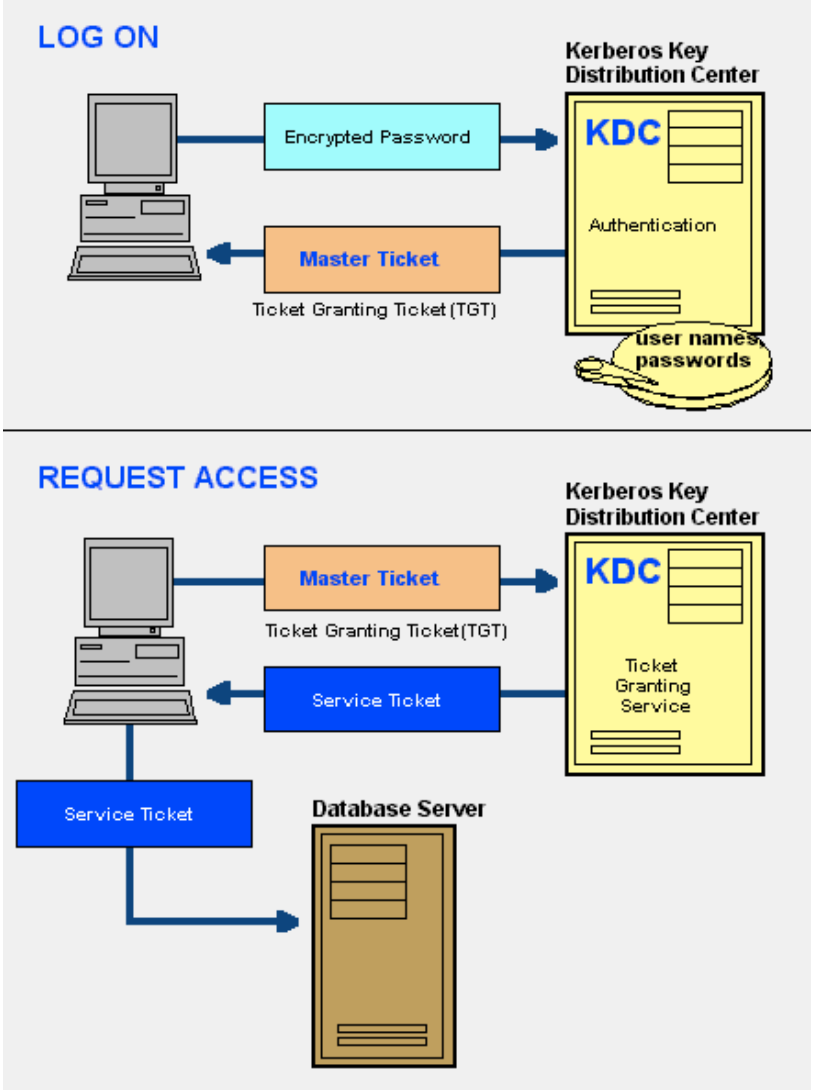

LOG ON
Kerberos Key Distribution Center
Encrypted Password
KDC
Authentication
Master Ticket
Ticket Granting Ticket (TGT)
user names passwords
REQUEST ACCESS
Kerberos Key Distribution Center
Master Ticket
Ticket Granting Ticket(TGT)
KDC
Ticket Granting Service
Service Ticket
Service Ticket
Database Server

Figure 4.3

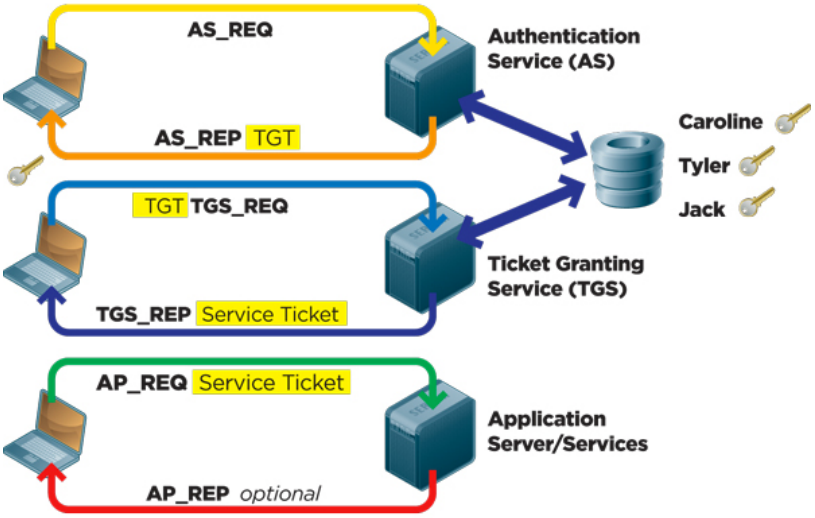


Figure 4.4

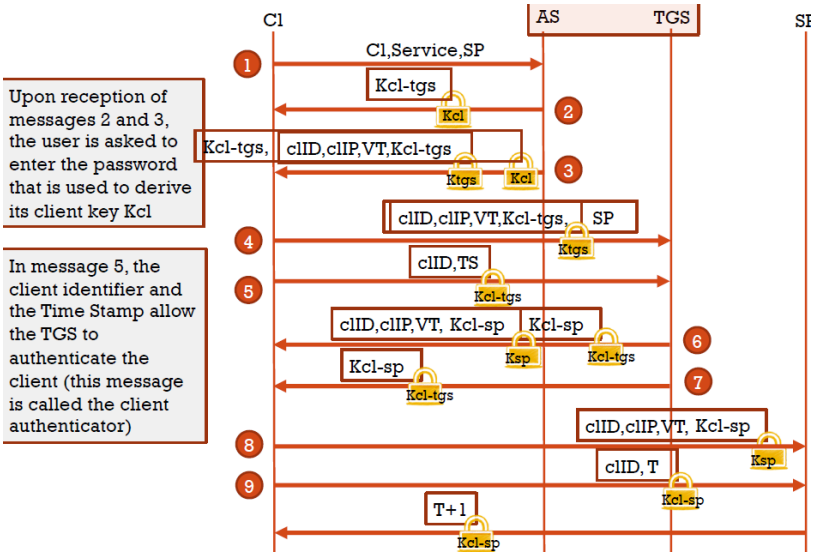


Figure 4.5: The protocol

validation time (VT), a session key K_{cl-sp} for the Client and the Service Provider encrypted with the K_{sp} key that is known to TGS and another message containing the same K_{cl-sp} session key encrypted with the K_{cl-tgs} session key

7. Before steps 8 and 9 the client recovers the ticket meant for the service provider with the K_{cl-tgs} session key

8. In steps 8 and 9 the client next sends the following two messages to the service provider: the Client-to-ServiceProvider ticket that was encrypted by TGS with the K_{sp} key and an authenticator that consists of the Client ID and the timestamp, encrypted with the K_{cl-sp} session key

9. Before step 10 the SP decrypts the ticket with its own K_{sp} key, it extracts the K_{cl-sp} session key from the ticket and then uses the session key to decrypt message 9 received from the client.

10. In step 10 SP sends to the Client a message that consists of the timestamp in the authenticator received from the Client plus one, encrypted with K_{cl-sp}

11. After step 10 the client decrypts the message received from the Service Provider using the K_{cl-sp} session key and makes sure that the message contains the correct value for the timestamp. If that is the case, the client can start interacting with the Service Provider

4.6 Kerberos Security Issues

The primary way in which an attacker will attempt to compromise a Kerberos Infrastructure would be to attack the Kerberos servers. If an attacker can gain root access to a KDC, the attacker will have access to the database of shared keys of the principals.

Since the security of Kerberos authentication is in part based upon the time stamps of tickets, it is critical to have accurately set clocks on Kerberos servers. It is advisable to set a short lifetime for tickets to prevent attackers from performing successful brute force attacks or replay attacks. If one allows for server clocks to drift, the network will become vulnerable to such attacks. If clocks are not synchronized within a reasonable time window, Kerberos will report fatal errors and refuse to function. machine with an inaccurate clock will be failed by the KDC in authentication attempts due to the time difference with the KDC's clock. The Network Time Protocol (NTP) is available for the time synchronization of servers.

4.7 Protocol Specification & Analysis

We want to give structured methods to prove or disprove that a given protocol satisfies a certain security property and achieves a certain security goal despite an attacker.

It is already difficult to understand the security of certain protocols, of the absence of a problem. Even with few messages, can we argue no attack is possible? We want evidence that the protocol does exactly what we designed, and we have achieved the security goal.

4.7.1 Digression on notation

Alice&Bob–notation is commonly used to describe security protocols as sequences of message exchange steps of the form: $\mathcal{A} \rightarrow \mathcal{B} : \mathcal{M}$ where $\mathcal{A}$ and $\mathcal{B}$ are the parties involved in the message exchange and $\mathcal{M}$ is the message being exchanged. But there are some issues that need clarification:

1. Need to make explicit what is known (public, private) before a protocol run, and what is to be generated freshly during a protocol run (e.g. nonces).
2. Need to make explicit which checks the individual principals are expected to carry out on the reception of messages. This is because the message sent by $\mathcal{A}$ may never be received by $\mathcal{B}$, or $\mathcal{B}$ might receive a different message from the one sent by $\mathcal{A}$.
3. Principals act concurrently, in contrast to the apparently sequential idealized execution of a run according to a narration. So messages could be received in different order from the expected one.
4. Concurrency occurs also at the level of different protocol sessions, which may happen to be executed simultaneously while sharing principals across.

A more precise characterization of $\mathcal{A} \rightarrow \mathcal{B} : \mathcal{M}$ is that $\mathcal{A}$ asynchronously sends $\mathcal{M}$ towards $\mathcal{B}$, that is then received by $\mathcal{B}$. Finally $\mathcal{B}$ checks that the message received is the intended one by checking the properties of the message (e.g. the gained knowledge is consistent with previously acquired knowledge).

Due to the distributed nature of the system, protocols are highly asynchronous. A party to a protocol does not know anything about the current run of the protocol except the messages it has received and sent. Except for the initiator, other parties will not even know that they are participating until they receive their first message.

There are many different attacks that can be performed on an exchange of messages, and these attacks boil down to these questions:

- Are both authentication and secrecy assured?
- Is it possible to impersonate one or more of the parties?
- Is it possible to interject messages from an earlier exchange (replay attack)?
- What tools can an attacker deploy with which capabilities?
- If any key is compromised, what are the consequences?

During protocol design we consider the Dolev-Yao model, in which the attacker can overhear, intercept, and synthesize any message and is only limited by the constraints of the cryptographic methods used. In other words: "the attacker carries the message" but it cannot break the cryptographic primitives that are considered perfect. The protocol should be robust in the face of such a determined and resourceful attacker.

4.8 Verification of cryptographic protocols

Protocols can be notoriously difficult to get correct. Protocols often depend on assumptions that are not clearly stated. It would be nice to be able to reason with mathematical rigour about protocol correctness.

There are three approaches to protocol verification:

- Belief logic allow reasoning about what principals within the protocol should be able to infer from the messages they see. Allows abstract proofs, but may miss some important flaws
- State exploration methods (model checking) treat a protocol as a state machine and conduct an exhaustive search checking that all reachable states are safe. Is automated Tamarin.
- Theorem proving uses induction over potential traces of protocol execution

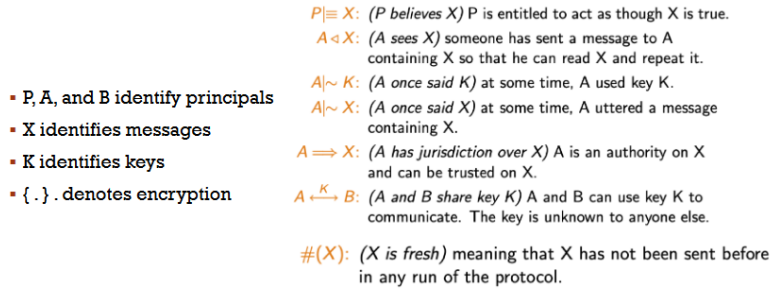


Figure 4.6: Ban logic operands

4.8.1 Belief Logic

A belief logic is a formal system for reasoning about beliefs. Any logic consists of a set of logical operators and rules of inferences that allows us to symbolically derive true statements.

Write mathematical expression, that describes the state of various entities (example: entity knows a key shared with another entity). A mathematically precise language two write expressions to represent states. It is made of inference rules, that allows to derive (infer) other expressions.

4.8.2 Ban Logic

Introduced in 1989, after its inventors: Burrows, Abadi, Needham. It is based on an agreed set of deduction rules for formally reasoning about the authentication protocols and is often referred to as a logic of authentication. It is a mathematically rigorous method for verifying that two principals (people, computer, services) are entitled to believe they are communicating with each other and not the intruders. The logic cannot prove that a protocol is wrong, i.e. to disprove that a security goal cannot be achieved. However, if one can not prove a protocol correct, then it is wise to consider that protocol with great suspicion. It helps clarify the protocol's assumptions by formally stating them. This is a general phenomenon that is intrinsic to the formalization process, i.e. the fact of using strictly specified inference rules immediately points out which hypotheses are missing.

One assumption that we make in the Needham-Schroeder is that N_A and N_B is a nonce (N_A is a nonce to A, because A uses an algorithm to generate every time fresh nonces, but how can B and S know if A generated it in a correct way, so they have to trust A), also that K_{AB} is fresh, never seen before.

It is a logical systems based on principal beliefs and their evolution after message exchanges: it concentrates on the beliefs of trustworthy parties involved in the protocol and the evolution of these beliefs through communication processes. A logical system is a deductive system together with additional axioms and a semantics. A deductive system consists of the axioms and rules of inference that can be used to derive theorems of the system. An axiom is a statement that is taken to be true, to serve as a premise or starting point for further reasoning and arguments. Example: It is possible to draw a straight line from any point to any other point. An inference rule is a function which takes premises, analyzes their syntax, and returns a conclusion for example Modus Ponens. The semantics gives the meaning to the expressions of the deductive system.

Principals are entities participating in the message exchange.

$$\frac{A|\equiv (A \overset{K}{\longleftrightarrow} B), A \triangleleft \{X\}_K}{A|\equiv (B|\sim X)}$$

Figure 4.7: Rule 1

$$\frac{A|\equiv (\#(X)), A|\equiv (B|\sim X)}{A|\equiv (B|\equiv X)}$$

Figure 4.8: Rule 2

4.8.3 Ban logic rules

Rule 1

A believes that A and B share the key K, A sees the message with a format X encrypted with key K. So A believes that B created that message. So A belives that that message comes from B, because no one else knows the shared key K other than B, because we are using symmetric cryptography 4.7.

Rule 2

A believes that X is fresh (X is the nonce), A believes that B sent a message containing X. A believes everything said by B because it was said by B with the fresh nonce 4.8.

Rule 3

A believes B is trusted in saying message X, A believes that B believes X, A believes X 4.9.

4.8.4 Protocol analysis

The steps of BAN logic to analyse the original protocol are as follows:

1. The protocol is transformed into some idealized form
2. Identify initial assumptions in the language of BAN logic
3. Use the axioms and rules of the logic to deduce new expressions
4. Interpret the statements proved by the process: are the goals achieved?

Authentication rests on communication protected by shared session key, so the goals of authentication may be reached between A and B if there is a shared K such that:

- Key authentication: $A \mid\equiv A \Leftrightarrow^K B$ and $B \mid\equiv A \Leftrightarrow^K B$

$$\frac{A|\equiv (B \Longrightarrow X), A|\equiv (B|\equiv X)}{A|\equiv X}$$

Figure 4.9: Rule 3

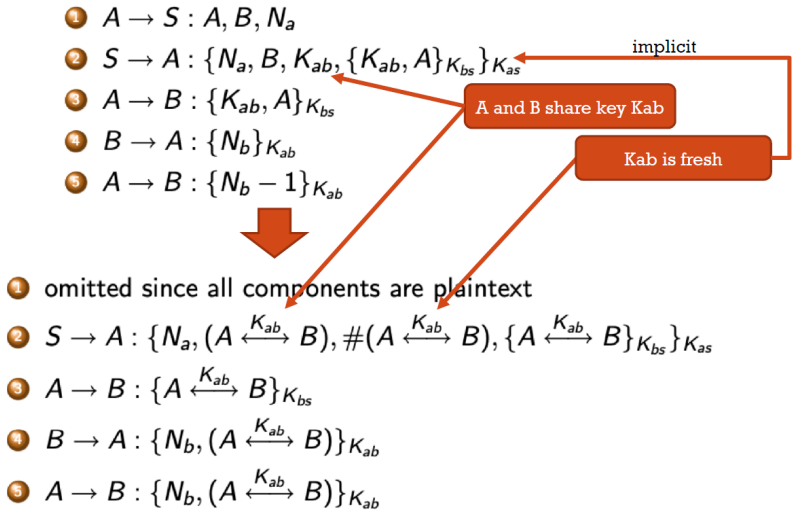


Figure 4.10: Formalization 1

- Key confirmation: $A \mid\equiv B \mid\equiv A \Leftrightarrow^K B$ and $B \mid\equiv A \mid\equiv A \Leftrightarrow^K B$
- Key freshness: $A \mid\equiv \#(A \Leftrightarrow^K B)$ and $B \mid\equiv \#(A \Leftrightarrow^K B)$

4.9 Application to Needham-Schroeder

4.9.1 Formalization

Steps to formalization: 4.10, 4.11, 4.12, 4.13.

Regarding 4.11:

- K_{AS} is a shared key by A and S; both A and S believes this
- K_{BS} is a shared key by B and S; both B and S believes this
- K_{AB} is a shared key by A and B; only S believes this

Regarding 4.12: for any key K: S is an authority to say that K is a shared key by A and B, and it is trusted to say this; both A and B believes this. S is an authority to say that K is a fresh shared key by A and B, and it is trusted to say this; only A believes this.

Regarding 4.13:

- N_A is fresh and A believes this
- N_B is fresh and B believes this
- K_{AB} is a fresh shared key by A and B; S believes this.

Regarding 4.13: For any key K, B believes that K is a fresh shared key by A and B. Notice that A and B trusts S in different ways: A trusts also the fact that S is able to generate fresh keys, B does not trust this and assumes that any shared key with A is fresh. This paves the way to the attack.

Needham and Schroeder did not realize they were making this assumption.

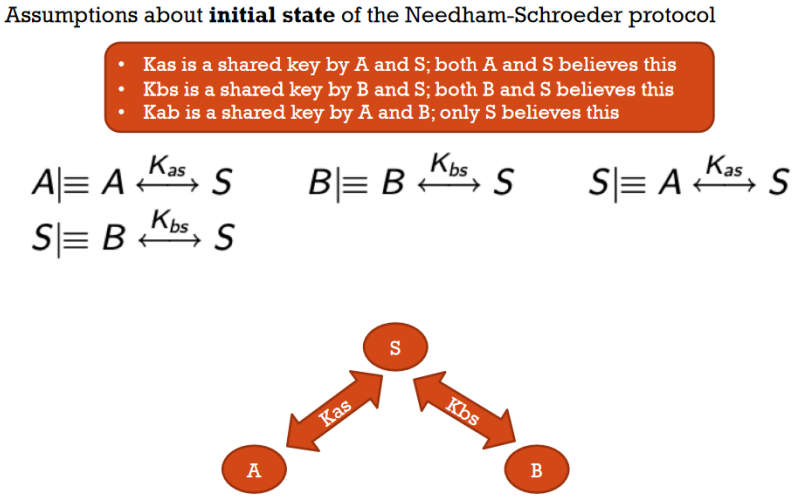


Figure 4.11: Formalization 2.1

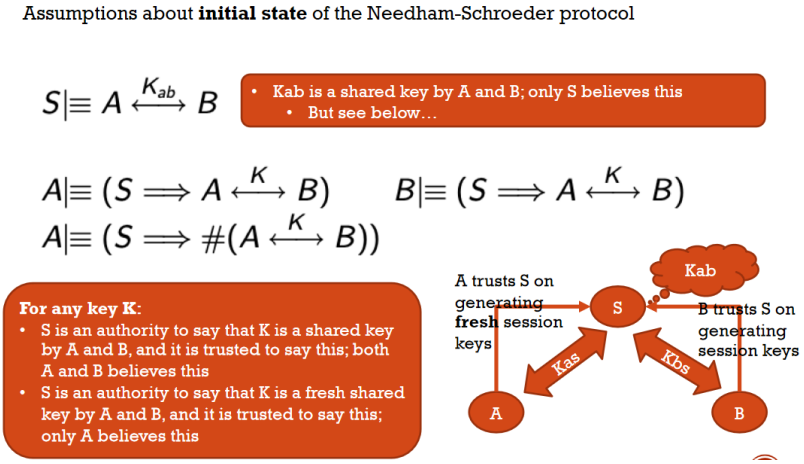


Figure 4.12: Formalization 2.2

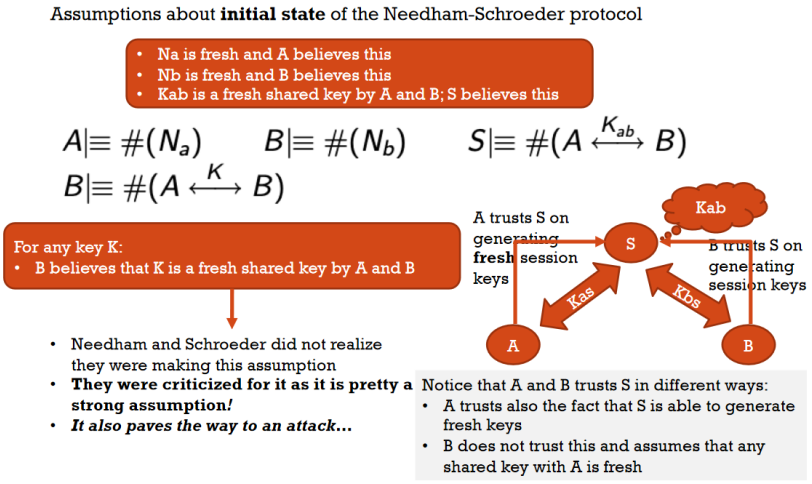


Figure 4.13: Formalization 2.3

4.10 Deduction

The assumption $B \mid\equiv \#(A \Leftrightarrow^K B)$ is dubious because it is different from all other freshness assumptions since those were all based on values that the believing principal had herself generated. This one expresses B belief that a random value someone else has generated is fresh! The consequence of this strange assumption is the attack described above where an attacker is assumed to be able to compromise an old session key. Since B is ready to accept the freshness of any key generated by S, it is willing to accept also the compromised key that can be replayed by the attacker that was able to compromise it.

The attack relies on the fact that B has no way to actually be assured that message 3 is fresh. An attacker could spend whatever time is needed to break the session key K_{AB} as long as it can do so within the lifetime of K_{BS}, it can run the replay attack described above. B will think it has confirmed sharing K_{AB} with A, when in reality A is not present and the attacker knows the key. Notice that the replay attack is not directly uncovered by a BAN analysis of the protocol; rather the analysis shows that the protocol cannot achieve any sort of authentication for B without making the dubious assumption that underlies the attack.

BAN logic is a belief system and it is much different from a knowledge system. Knowledge systems have an inference rule of the form "if a principal known p, then p is true". Belief systems do not have this rule, since a belief in p says nothing about the truth or falsity of p.

In BAN logic, it is assumed that all principals taking part in a protocol are honest, in the sense that each principal believes in the truth of each message it sends. Unfortunately, this assumption is far from being reasonable in presence of attackers.

$S \rightarrow A : \{N_a, (A \overset{K_{ab}}{\longleftrightarrow} B), \#(A \overset{K_{ab}}{\longleftrightarrow} B), \{A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{bs}}\}_{K_{as}}$

From step 2 of the (idealized) protocol:

$$A \triangleleft \{N_a, (A \overset{K_{ab}}{\longleftrightarrow} B), \#(A \overset{K_{ab}}{\longleftrightarrow} B), \{A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{bs}}\}_{K_{as}}$$

The *Nonce Verification Rule* says:

$$\frac{A|\equiv (\#(X)), A|\equiv (S|\sim X)}{A|\equiv (S|\equiv X)}$$

Since A believes N_a to be fresh, we get:

$$A|\equiv (S|\equiv A \overset{K_{ab}}{\longleftrightarrow} B)$$

Figure 4.14: Deduction 1

The *Jurisdiction Rule* says that:

$$\frac{A|\equiv (S \Longrightarrow X), A|\equiv (S|\equiv X)}{A|\equiv X}$$

From this we obtain:

$$A|\equiv A \overset{K_{ab}}{\longleftrightarrow} B$$

Key freshness (for A) $\quad A|\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$

Figure 4.15: Deduction 2

Since A has also seen the part of the message encrypted under B's key, he can send it to B. B decrypts the message and obtains:

$B|\equiv (S|\sim A \overset{K_{ab}}{\longleftrightarrow} B)$ 3 $A \rightarrow B : \{A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{bs}}$

meaning that B believes that S once sent the key.

At this point, we need the final dubious assumption:

$B|\equiv \#(A \overset{K}{\longleftrightarrow} B)$

With it, we can get:

$B|\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

Key freshness (for B) derived from built in belief of B

Figure 4.16: Deduction 3

4 $B \rightarrow A : \{N_b, (A \overset{K_{ab}}{\longleftrightarrow} B)\}_{K_{ab}}$

5 $A \rightarrow B : \{N_b, (A \overset{K_{ab}}{\longleftrightarrow} B)\}_{K_{ab}}$

From the last two messages, we can infer the following.

Key authentication

$A|\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

$B|\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

Key confirmation

$A|\equiv (B|\equiv A \overset{K_{ab}}{\longleftrightarrow} B)$

$B|\equiv (A|\equiv A \overset{K_{ab}}{\longleftrightarrow} B)$

Figure 4.17: Deduction 4

Chapter 5

Public Key Cryptography

5.1 Introduction

Use a pair of keys: public and private. Deriving a private key from a public key would require to solve a computationally difficult mathematical problem.

One way trapdoor function are function that easy to compute whereas their inverse function is difficult to compute if a secret t is not known but becomes easy to invert when the secret t is available. As an example:

- Multiplication vs. factorization: given prime numbers, say 3 and 7, it is easy to calculate the product. Given number 21 that is a product of two primes, it is not easy to determine the prime factors. Problem becomes much harder if we start with primes that have, say, 400 digits or so, because the product will have 800 digits.
- Exponentiation vs. logarithms: take the number 3 to the 6th power; it is relatively easy to calculate $3^6 = 729$. Start with number 729; it is difficult to determine the two integers, x and y s.t. $log_x 729 = y$.

It is unknown if one-way trapdoor functions exist, we believe that certain are indeed so.

Kerckhoffs principle: do not rely on the secrecy of algorithms; the key should be the only secret that needs protection. The security of Public Key Cryptography boils down to the protection of the private key.

All parties interested in secure communications publish their public keys. SSL/TLS uses X.509 certificates that are exchanged between clients and servers during handshakes. This side-steps the problem of key distribution that plagued symmetric key cryptography.

The communication between entities with PKC is made to be confidential 5.1 and authentic 5.2: Party $\mathcal{A}$ can encrypt a message using $\mathcal{B}$ publicly available key. Such a message would only be decipherable by $\mathcal{B}$ under the assumption that only $\mathcal{B}$ have access to the corresponding private key. Party $\mathcal{A}$ can encrypt the message with $\mathcal{A}$ own private key. Since this message would only be decipherable with A public key, that would establish the authenticity of the message, i.e. $\mathcal{A}$ is indeed the source of the message. It is possible to combine the two techniques and provide both confidentiality and authenticity for any message exchanged between $\mathcal{A}$ and $\mathcal{B}$.

PKC has proved indispensable for key management, for distributing the keys needed for the more traditional symmetric key cryptography, for digital signature applications, symmetric-key systems continue to be widely used for content encryption because of the greater computational cost associated with PKC. To preserve both at the same time:

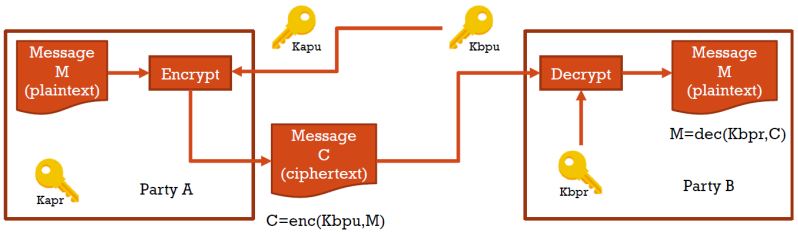


Figure 5.1: Confidentiality

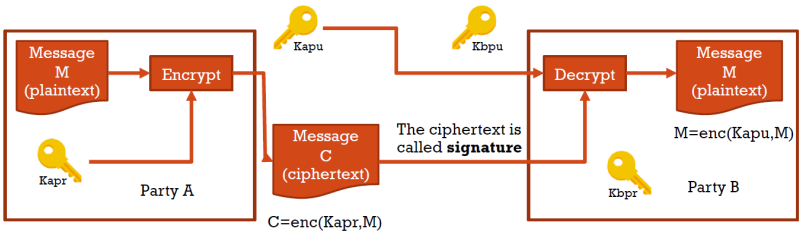


Figure 5.2: Authenticity. Note: in reality, it is not the whole message to be encrypted but rather a digest produced by a hash function

- On the sender side, party $\mathcal{A}$: $\mathcal{C}_1 = enc(\mathcal{KA}_{pr}, \mathcal{M})$, encrypt with its private key plaintext so to enable the check of authenticity and $\mathcal{C}_2 = enc(\mathcal{KB}_{pu}, \mathcal{C}_1)$ encrypt with public key of receiver so to enable the check for confidentiality.
- On the receiver side, party $\mathcal{B}$: $\mathcal{M}\prime = dec(\mathcal{KB}_{pr}, \mathcal{C}_2)$ decrypt with its private key so to check for confidentiality: notice that $\mathcal{M}' = \mathcal{C}_1$ and $\mathcal{M}'' = dec(\mathcal{KA}_{pu}, \mathcal{M}')$ decrypt with sender public key to check for authenticity: notice that $\mathcal{M}'' = \mathcal{M}$

The price to pay for achieving both confidentiality and authentication is that a message must be processed 4 times per exchange, 2 on the sender side and 2 in the receiver side. This comes at the price of the public-key algorithm. PKC does not make obsolete symmetric-key cryptography because of the greater computational cost associated with PKC, symmetric-key systems continue to be widely used for content encryption. PKC has proved indispensable for key management, for distributing the keys needed for the more traditional symmetric key cryptography, for digital signature applications

5.1.1 Forward secrecy

If someone steals the server private key, then all past and future communications are compromised with all clients. This is so because if attackers have recorded past encrypted messages, then all past and future symmetric keys can be decrypted. (Perfect) forward secrecy gives assurances that symmetric keys will not be compromised even if private keys are compromised.

To obtain forward secrecy there are two ways:

- A temporary private key is generated for each session. Every session has a different set of keys. It is impossible to decrypt past traffic without having complete access to the server at the time-zero.

- Use ephemeral key exchange. It requires generating a new public/private key pair per session. It is impossible to decrypt past communications even if the private key of the involved parties is compromised later

5.2 RSA: Rivest, Shamir, Adleman

The RSA method was published in scientific paper in 1977. An RSA user creates and publishes a public key based on two large prime numbers. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decoded by someone who knows the prime numbers. The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the factoring problem. There are no published method to defeat the system if large enough key is used.

5.2.1 RSA main idea

Euler Theorem for every positive integer n and every a that is coprime to n, the following must be true:

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

where $\phi(n)$ is the totient of n. Note that $a^{\phi(n)} \equiv 1 \bmod n$ simply means that $a^{\phi(n)} = 1 + kn$ for some integer k.

Notice that Euler's totient function $\phi(n)$ counts the positive integers up to a given integer n that are relatively prime to or, equivalently, coprime with n. It is possible to use Euler theorem to greatly simplify the computations required to compute large powers modulo n. For example consider computing $7^{222} \bmod 10$. 7^{222} is huge, so instead observe that 7 and 10 are coprime and that $\phi(10) = 4$. Because of Euler theorem we have that $7^4 \equiv 1 \bmod 10$, thus we can derive: $7^{222} \equiv (7^4)^{55} * 7^2 \equiv 1^{55} * 7^2 \equiv 49 \equiv 9 \bmod 10$. More generally it can be shown that *if* $x \equiv y \bmod (\phi(n))$,*then* $a^x \equiv a^y \bmod n$.

Let $\mathcal{M}$ be is an integer that represents a message. Pick two integers e and d such that $ed \equiv 1 \bmod \phi(n)$. Assume that $\mathcal{M}$ is coprime with the modulus n. Then we have $M^{ed} \equiv M^{ed \bmod \phi(n)} \equiv M \bmod n$. The assumption that $\mathcal{M}$ and the modulus n are coprime can be weakened by requiring that the modulus n is the product of two primes. In fact, under this hypothesis, the last congruence relation above holds for any value of the message $\mathcal{M}$ between 0 and $n - 1$ (both included).

5.2.2 Core algorithm

Preconditions are:

- n a modulus for modular arithmetic
- $\phi(n)$ the totient of n
- e an integer that is coprime with $\phi(n)$. The coprime requirement guarantees the existence of the multiplicative inverse modulo $\phi(n)$.
- d an integer that is the multiplicative inverse of e modulo $\phi(n)$

Encryption: as an input a plaintext message $\mathcal{M}$ encoded as an integer between 0 and $n - 1$ (included), and as output $\mathcal{C} = \mathcal{M}^e \bmod n$.

Decryption: as an input a ciphertext $\mathcal{C}$ obtained as explained above, and as output a plaintext $\mathcal{M} = \mathcal{C}^d \bmod n$.

Observe that $\mathcal{C}^d \bmod n = (\mathcal{M}^e \bmod n)^d \bmod n = \mathcal{M}^{ed \bmod \phi(n)} \bmod n = \mathcal{M} \bmod n = \mathcal{M}$.

5.2.3 RSA for confidentiality

An entity $\mathcal{A}$ who wishes to receive messages confidentially will use:

- the pair of integers $\{e, n\}$ as its public key
- the pair of integers $\{d, n\}$ as the private key
- The parameters n, e, and d are as introduced above

Another entity $\mathcal{B}$ wishing to send a message $\mathcal{M}$ to $\mathcal{A}$ confidentially will encrypt $\mathcal{M}$ using $\mathcal{A}$'s public key $\{e, n\}$ to create the ciphertext $\mathcal{C}$. Only $\mathcal{A}$ will be able to decrypt $\mathcal{C}$ using its private key $\{d, n\}$. If the plaintext message $\mathcal{M}$ is too long, $\mathcal{B}$ may choose to use RSA as a block cipher for encrypting the message meant for $\mathcal{A}$. When RSA is used as a block cipher, the block size is likely to be half the number of bits required to represent the modulus n. In practice, because of its high computational overhead, RSA is more likely to be used just for server authentication and for exchanging a secret session key. Subsequently, a session key generated with the help of RSA-based encryption can be used for content encryption using symmetric-key cryptography.

Now we will ask ourselves: how should we choose the modulus n for the RSA algorithm to be considered secure?

5.2.4 Parameters selection

Recall that $\mathcal{M}^{ed} \equiv \mathcal{M}^{ed \bmod \phi(n)} \equiv \mathcal{M} \bmod n$ under the assumption that $\mathcal{M}$ is coprime with the modulus n. It was shown by Rivest, Shamir, and Adleman that the same property holds for all $\mathcal{M}$ if the modulus n is a product of two prime numbers, i.e. $n = 7 * 3$ for some prime p and prime q. It is crucial for p and q to be large so also n is so, and the problem of factorizing n becomes harder even with one of the modern integer factorization algorithms.

5.2.5 Key generation

- Generate two distinct primes p and q
- Calculate the modulus $n = pq$
- Calculate the totient $\phi(n) = (p-1)(q-1)$
- Select for public exponent an integer e such that $1 < e < \phi(n)$ and $gcd(\phi(n), e) = 1$
- Calculate for the private exponent a value d such that $d = e^{-1} \bmod \phi(n)$
- Set public key to $\{e, n\}$
- Set private key to $\{d, n\}$

But how do we generate primes? We need to generate sequences of numbers that look random. To be considered random, it must exhibit two properties: all the numbers in a designated range must occur equally often (uniform distribution) , and even if one knows some or all the numbers up to a certain point in a random sequence, he/she should not be able to predict the next one (or any of the future ones) (independence). Truly random numbers can only be generated by physical phenomena, computer offer only an approximation by generating them with specific algorithms.

Linear congruent generators are used: $X_{n+1} = (a * X_n + c) \bmod m$ where X_0 is the seed. Under suitable conditions, it is possible to obtain sequences that starts repeating after m randomly generated numbers.

But this is not cryptographically secure since it is possible to recover the values of a, c, and m by solving 3 congruences:

- $X_1 = (a * X_0 + c) \bmod m$
- $X_2 = (a * X_1 + c) \bmod m$
- $X_2 = (a * X_2 + c) \bmod m$

As a consequence, even when a PRNG produces a random looking sequence, it may not be secure enough for cryptographic applications. A pseudorandom sequence produced by a PRNG can be made more cryptographic secure by restarting the sequence with a different seed after every N numbers.

Also Linear Feedback Shift Registers could be used but it is possible to perform attacks based on Linear Algebra, so it's better to combine them with non-linear components.

5.2.6 Primality Testing

Primality testing is much easier than factoring. Indeed, to check primality there is no need to factor the number under consideration. There are two types of primality testing algorithms: deterministic tests determine with absolute certainty whether a number is prime, probabilistic tests can potentially (although with very small probability) falsely identify a composite number as prime (although not vice versa). In general probabilistic tests are much faster than deterministic tests. Numbers that have passed a probabilistic prime test are therefore properly referred to as probable primes until their primality can be demonstrated deterministically.

Recall Fermat's Little Theorem: if p is prime, then for any integer a that is coprime with p, we have $a^{p-1} \equiv 1 \bmod p$, or equivalently, if p is prime then for any integer a that is coprime with p, we have that $a^{p-1} - 1$ is divisible by p. It is possible to use this result to design a probabilistic algorithm for primality testing. Let n be the integer that we would like to check for primality. Randomly select an integer a, if $a^{n-1} \bmod n$ does not yield 1, then n is not a prime; otherwise, you cannot be certain, this is the idea underlying what is known as the Miller-Rabin algorithm for primality testing. With t probes the probability of erroneous classification is significantly less than 4^{-t} provided that certain implementation best practices are adopted.

5.2.7 Generating primes

Preliminarily, set the size of the modulus n. Assume a modulus represented by a bitvector of B bits typically $B = \{1024, 2048, 3072, 4096, \ldots\}$. To generate prime p use a high-quality pseudo-random number generator, generate a random number of size $B/2$. Then set the lowest bit of the integer generated, this ensures the number generated is odd, and set the two highest bits so we are sure the number is large, and finally check if the resulting integer is prime (e.g., by using the Miller-Rabin

test), if not, increment the integer by 2 and check again. Generate the prime q in a similar way, finally check that the two generated integers are distinct.

5.2.8 Public exponent selection

Recall that the public exponent should be an integer e such that $1 < e < \phi(n)$ and $gcd(\phi(n), e) = 1$, recall also that $\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$. So from the $gcd(\phi(n), e) = 1$ we derive that: $gcd(p-1, e) = 1$ and $gcd(q-1, e) = 1$. To simplify computations, one typically chooses e that is prime, has as few bits as possible equal to 1 for fast multiplications, and is cryptographically secure. Typically $e = \{3, 5, 17, 65537(= 2^{16} + 1)\}$. Clever algorithms are used to do fast modular exponentiation.

5.2.9 An algorithm for modular exponentiation

We still need to devise a procedure to compute $A^B \bmod n$ as these can grow pretty rapidly. Calculations can be sped up by realizing that B can be expressed as a sum of smaller parts, then the result is a product of smaller exponentiations. [1]

```
result = 1
while (B>0):
        if (B & 1):
                result = (result * A) % n
        B = B >> 1
        A = (A * A) % n
return result
```

5.2.10 On being cryptographically secure

Small values for e, such as 3, are considered cryptographically insecure. To understand why, consider $\mathcal{A}$ sends the same message $\mathcal{M}$ to three different receivers using their respective public keys that have the same $e = 3$ but different values of n, say n_1, n_2, n_3 respectively. Assume that an attacker can intercept all transmissions so that it can collect the following 3 ciphertexts:

$$\mathcal{C}_1 = \mathcal{M}^e \bmod n_1, \mathcal{C}_2 = \mathcal{M}^e \bmod n_2, \mathcal{C}_3 = \mathcal{M}^e \bmod n_3$$

Assuming that n_1, n_2, n_3 are relatively prime on a pairwise basis, the attacker can use the Chinese Remainder Theorem to compute $\mathcal{M}^3 \bmod (n_1 * n_2 * n_3)$. The attacker then can compute $\mathcal{M}$ from $\mathcal{M}^3$ which is not a computationally complex task.

5.2.11 Chinese remainder theorem

It states that in modulo M arithmetic, if M can be expressed as a product of n integers that are pairwise coprime, then every integer in the set $\{0, 1, 2, \ldots, M-1\}$. can be reconstructed from residues with respect to those n numbers. [2]

[1] TODO: Slide 53..56 6-PKC_RSA-2p.pdf
[2] TODO: Slide 60..63 6-PKC_RSA-2p.pdf

5.2.12 Private exponent selection

Recall that we need to find a value d of the private exponent such that:

$$ed \equiv 1 \bmod \phi(n)$$

This means to compute the modular inversion of the public exponent e:

$$d = e^{-1} \bmod \phi(n)$$

Since d is the multiplicative inverse of e modulo $\phi(n)$, we can use the Extended Euclid Algorithm for calculating d, recall that $\phi(n) = (p-1)(q-1)$. The main source of security in RSA is keeping both p and q secret and thus also keeping $\phi(n)$ secret.

5.2.13 Euclid's algorithm for GCD

It's based on the following properties:

- $gcd(a, a) = a$
- If b divides a, then $gcd(a, b) = b$
- $gcd(a, 0) = a$, this is so because it is always the case that a divides 0.

The recursive step of the algorithm is: $gcd(a, b) = gcd(b, a \bmod b)$.

5.2.14 Finding multiplicative inverses

The multiplicative inverse of a with respect to n (with $a < n$) is the integer $b < n$ such that $a * b \equiv 1 \bmod n$. The multiplicative inverse exists for each integer $a < n$ such that a and n are coprime, i.e. $gcd(a, n) = 1$. Notice that if n is a prime, then it is always possible to find a multiplicative inverse for any non-zero integer $a < n$.

Bezout identity: $gcd(a, b) = x * a + y * b$ for integers x and y (positive or negative). This identity allows us to find multiplicative inverses. By Bezout identity we derive that there exists x and y such that $x * a + y * n = 1$, if we take the equality mod$n$ we have: $x * a \bmod n + y * n \bmod n = 1 \bmod n$, that simplifies to $x * a \bmod n = 1$.

We can sue Euclid's algorithm with some modifications:

- at each step, write the expression of the remainder in the form $a * x + n * y$.
- when the remainder becomes 1 (which will happen only when a and n are relatively prime, x will automatically be the multiplicative inverse of a w.r.t. n.

TODO [3]

5.3 Security of RSA

As an example if a very short message is sent and e is set to 3, $\mathcal{C} = \mathcal{M}^3 \bmod n$ will be just equal to $\mathcal{M}^3$ so the attacker can recover $\mathcal{M}$ easily. To prevent this, padding is added to the plaintext message.

[3] TODO: Slide 71 6-PKC_RSA-2p.pdf

5.3.1 Chosen ciphertext attacks

Consider entity A sends a ciphertext message $\mathcal{C} = \mathcal{M}^e \bmod n$. With $\{e, n\}$ being the public key of B, and $\{d, n\}$ being the private key of B. The Moore's attack consist of: Eve intercepts $\mathcal{C}$, it can guess the plaintext $\mathcal{M}$ without knowing the private key of B as follows

- Eve randomly chooses an integer s with a multiplicative inverse s^{-1} w.r.t. the modulus n
- Eve constructs a new message $C' = s^e * C \bmod n$ and sends it to Bob
- Bob decrypts the message C'
- Bob sends back the signed ciphertext $M' = C'^d \bmod n = (s^e * C \bmod n)^d * C^d \bmod n = s * M \bmod n$ (because the attacker has somehow convinced B to do so)
- Eve will be able to record the orginal message M by multiplying M' by s^{-1}: $M' * s^{-1} = (s * M \bmod n) * s^{-1} = M$

[4]

There was another attack called the Bleichenbacher attack in which the Eve has access to an oracle that tells if any ciphertext is conformant to the PKCS standard. Eve craft a sequence of ciphertexts to be sent to the oracle. It is possible to show that Eve gains information about the ciphertext by making the sequence of ciphertexts adapting (i.e. the next ciphertext is crafted according to the previous answers of the oracle). Eve crafts a sequence of randomly selected integers s to form a candidate sequence of ciphertexts $C' = s^e * C \bmod n$. Eve chooses the integers s one at a time, forms the ciphertext C' and sends it to the oracle to find out if C' is conformant to the PKCS1 standard or not, i.e. in particular if its first two bytes have the integer values of 0 and 2. Each positive answer from the oracle allows the attacker to enlarge the size of s and make an increasingly narrower estimate for the value of the plaintext integer M that corresponds to the original Cs.
[5].

Two different types of chosen ciphertext attacks: CCA1 and CCA2.

- CCA1 (also called passive chosen ciphertext attack): The attacker can consult the decryption oracle an arbitrary number of times, but only until the attacker has acquired the ciphertext C through eavesdropping
- CCA2 (also called adaptive chosen ciphertext attack): The attacker can continue to consult the oracle even after seeing the ciphertext C

Bleichenbacher attack is a CCA2. This implies that PKCS#v1.5 is not CCA2 secure. RSA is made resistant to CCA2 when the padding bytes are set according to OAEP (Optimal Asymmetric Encryption Padding).

5.3.2 Low entropy on keys

The entropy of a random number generator is at its highest if all numbers are equally likely to be produced within the range of numbers that the output is designed for. If a generator can produce 512-bit random numbers with equal probability, its entropy is at its maximum and it equals 512 bits. If the probabilities associated with the output random numbers are nonuniform, the entropy will be less than 512 bits. The greater the nonuniformity of the probability distribution,

[4] TODO : Slide 76..77 6-PKC_RSA-2p.pdf
[5] TODO: Slide 80..83 6-PKC_RSA-2p.pdf

the smaller the entropy, and the entropy is zero for deterministic output. Consider the situation in which an attacker has identified a common factor p of two moduli n_1, n_2 by using Euclid algorithm (with quadratic complexity in the number of bits representing the integers). This implies that n is GCD of both n_1, n_2. We can easily recover q_1, q_2 by division. To compute the private key then $d_i = {e_i}^{-1} \bmod (p-1)(q_i - 1)$ with $i = 1, 2$. Turns out that finding a pair of RSA moduli with a common factor is quite commonly found in the wild. This implies that random number generators used in key generation should have high enough entropy so that every modulus is unique vis-a-vis the moduli used by any other communication device anywhere on earth.

5.3.3 Mathematical attacks

Various mathematical techniques have been developed for solving the integer factorization problem involving large numbers. There exists the Fermat's factorization method, sive based method, Pollard method,...

5.4 Pragmatics on keys

It is important that the modulus is shared in a way such that everybody can use it.

RSA recommends that the two primes that compose the modulus should be roughly of equal length. So if one wants to use 1024-bit RSA encryption, then the modulus integer will have a 1024 bit presentation; this, in turn, implies that the two primes must be roughly 512 bits each.

Doubling the size of the key, i.e. the size of the modulus, will, in general, increase the time required: for encryption by a factor of 4 and for decryption by a factor of 8. Operations involving the private key are less affected because the public key exponent e does not have to change as the key size increases while the private key exponent d changes in direct proportion to the size of the modulus. Key generation time increases by a factor of 16 when the size of the key is doubled (but this is relatively infrequent).

For the public key, in addition to storing the encryption exponent and the modulus, the key may also include information such as: the time period of validity, the name of the algorithm used for key generation. Same goes for the private key. Typically, the formats require the keys to be stored using Base64 encoding so that they can be displayed using printable characters.

An example public key in the SSH representation is:

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDeBCk5iHdJ40Ojj7X9dzr8ZMx139Imx[...]
    ZfpmoetXK04jj3s5RriUK6VF5weIhWbiL0eXAv39Vo56e1lO8HfhpD6/tZEztaibADaVc=
    ivan@DESKTOP-L0NSSBD
```

An example private key is:

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEA3gQpOYh3SeNDo4+1/Xc6/GTMdd/SJsX1Rb/EhvPL4WjnQcXyBTXh
5wNf6RMuX6C8Tmx/Nmbe6TMRuxvcB5+GimTXpfE0BYwkRsxmTaDm1DaLomJsWqFg8tY2gt
[...]
sRL0AwR7I9/9FRJNlQH4gAs3rWUDqVG4YGg12pNVz4kBioPXhu9rCJiEF6KnJWx0tbBf+H
RG/orJ64KugdgtAAAAFGl2YW5AREVTS1RPUC1MME5TU0JEAQIDBAUG
-----END OPENSSH PRIVATE KEY-----
```

5.4.1 Final remarks

Assuming we are using the best possible random number generator to create candidates for the primes needed to compute the modulus and a version of RSA resistant to chosen ciphertext attacks, all the security still lies in the difficulty of factoring large integers. So using keys larger than 1024 bits is highly advised. A comparison between RSA and other cryptographic primitive can be seen in table 5.1. The table tells us that the computational overhead of RSA encryption/decryption goes up as the size of the modulus increases. This makes RSA inappropriate for encryption/decryption of actual message content for high data-rate communication channel. RSA plays a key role for the exchange of secret keys that can subsequently be used for the more traditional (and much faster) symmetric-key encryption and decryption of the message content.

5.4.2 Key exchange with public key cryptography

RSA, is one of the method used for key exchange based on public key cryptography in TLS. Public key cryptography is also used to bind a real world entity with a certificate. There is an hierarchy of certification authorities. There are some root CA, and a level below some other CAs that have a certificate signed by a root CA, and so on with every level.

5.4.3 RSA and lack of forward secrecy

(Perfect) Forward Secrecy is a property of key-exchange protocols in which the exposure of long-term keying material, used in the protocol to negotiate session keys, does not compromise the secrecy of session keys established before the exposure. The key sharing protocol based on RSA does not provide forward secrecy. To see why, consider the situation in which an attacker records all encrypted messages between a client and a server. In this way, the attacker also records the messages that the client and the server exchange during the key exchange protocol: a client gets the server's certificate to authenticate the server and to extract the server RSA public key for creating a secret session key. The client generates and then sends a shared key encrypted with the server's public key.

If the attacker can get its hands on the server's private key it will immediately be able to decrypt all the messages that the client and the server has exchanged during a session because it is able to decrypt the message, encrypted with the server's public key, containing the session key that has been generated by the client. Notice that the attacker can do this for any session whose messages it has recorded! To understand the likelihood of this attack, we need to understand how likely it is that a private is leaked unfortunately, it is likely and it has happened in the past because of insider attackers (e.g., disgruntled employees exfiltrating the private key of company servers). TLS heartbleed attack that was able to dump arbitrary memory portion of the server possibly including sensitive information such as its private key RSA is used in TLS to do authentication. But if for

Symmetric Key Algorithm	Key size for symmetric key cryptography	Comparable key size for RSA algorithm
2-Key DES	112	1,024
3-Key DES	168	2,048
AES-128	128	3,062
AES-192	192	7,680
AES-256	256	15,360

Table 5.1: Comparison between RSA and other cryptographic primitives.

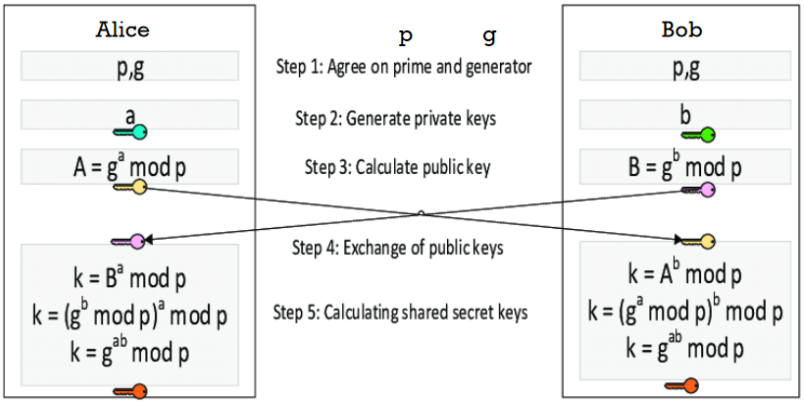


Figure 5.3: Diffie-Hellman key exchange

some reason the private key of the server is leaked we are in trouble since there is no forward secrecy. Intuitively, if the data exchanged are quite sensitive but such data is useful for a short time (i.e. they have a limited lifetime), then an attacker will be likely to decrypt the data only after the period in which they are useful. To avoid this problem, we need to come up with a method to share a session key that avoids to send the key over the channel!

5.5 Diffie-Hellman Key Exchange Algorithm

To solve this we need to use the Diffie-Hellman protocol! Given the following two public parameters (i.e. shared between the parties willing to agree on a shared session key): p is prime, $g < p$ is a group generator.

The Diffie-Hellman algorithm for key exchange between two parties A and B is:

1. $\mathcal{A} \rightarrow \mathcal{B} : g^a \bmod p$
2. $\mathcal{B} \rightarrow \mathcal{A} : g^b \bmod p$

$\mathcal{A}$ can compute the (pre-)shared key with $\mathcal{B}$ as $(g^b \bmod p)^a$ and B can compute the (pre-)shared key with $\mathcal{A}$ as $(g^a \bmod p)^b$. It turns out that $(g^b \bmod p)^a = g^{ab} \bmod p = (g^a \bmod p)^b$ 5.3. Notice that the (pre-)shared key has been never sent over the channel! This is crucial for forward secrecy.

Every time $\mathcal{A}$ and $\mathcal{B}$ want to establish a communication channel, they have to generate a **new** private key, that has never been used before. This guarantees forward secrecy. This is known as ephemeral Diffie-Hellman key exchange. Extensions of the Diffie-Hellman key exchange support forward secrecy.

5.5.1 Digression on discrete logarithm problem

The security of RSA is based on the observation that while it is computationally efficient to perform multiplications of even large integers, it is infeasible or computationally very expensive to compute the prime factors of large integers. Note that a different way of attacking RSA is not known. In other words the security of RSA is not equivalent to the problem of factoring large integers.

The security of Diffie-Hellman is based on the observation that while it is computationally efficient to calculate powers of large integers, it is infeasible or computationally very expensive to compute the discrete logarithm of large integers.

For any positive integer N, the set of all integers $i < N$ that are coprime to N form a group with modulo N multiplication as the group operator. For example $N = 8$, the set $\{1, 2, 5, 7\}$ forms a group with modulo 8 multiplication. When $N = p$ with p a prime, the group is denoted with $\mathbb{Z}^*{}_p$ and consists of all the integers in the set $\{1, 2, \ldots, p-1\}$ that are coprime with p since this is prime. For some N, the set of support of the group with modulo N multiplication contains an element whose various powers (computed modulo N) are all distinct and span the entire set of support. Such an element is called the primitive element, the primitive root modulo N, or the generator of the group with modulo N multiplication. Primitive roots modulo N can be used to define discrete logarithm.

Recall that $x^y = z$ and $log_x z = y$. Similarly $x^y \equiv z \bmod N$ and $dlog_{x,N} z = y$. Observe that the unique discrete logarithm mod N to some base a exists only if a is a primitive root modulo N.

$\mathbb{Z}^*{}_p$ is a cyclic group, i.e. each element can be expressed as $g^i \bmod p$ for some i and g. For example $\mathbb{Z}^*{}_{17}$ is a cyclic group for $g = 3$, this means if we compute $3^i \bmod 17$ for $i = 0, 1, \ldots$ it is possible to get all possible elements in the support set of $\mathbb{Z}^*{}_{17}$, i.e $\{1, 2, \ldots, 16\}$. A subset of the support set of $\mathbb{Z}^*{}_p$ is a cyclic subgroup if the group operator is multiplication modulo p and all the elements of the subgroup can be generated through the powers of one of the elements of the subgroup.

For example $\{1, 2, 4, 8, 16, 15, 13, 9\}$ is a cyclic subgroup for $g = 2$, it is easy to check this by computing $2^i \bmod 17$ for $i = 0, 1, \ldots$

Lagrange's theorem in Group Theory: If M is the order of a cyclic subgroup of $\mathbb{Z}^*{}_p$, then M will be a divisor of $p - 1$. In the previous example we have that $\{1, 2, 4, 8, 16, 15, 13, 9\}$ is a cyclic subgroup for $g = 2$, the order of the subgroup is the cardinality of the set of support, i.e. $M = 8$. Indeed $M = 8$ is a divisor of $p - 1 = 17 - 1 = 16$.

Property of a cyclic group of order M: $g^M \equiv 1 \bmod p$. For the example above $2^8 \equiv 1 \bmod 17$ since $2^8 \bmod 17 = 256 \bmod 17 = 1$.

We are interested in those cyclic subgroups of $\mathbb{Z}^*{}_p$ whose order M is large. More precisely, we are interested in picking a generator g such that the order M of the induced cyclic subgroup is a large prime that is also a factor of $p - 1$. The security of the Diffie-Hellman algorithm crucially depends on the size of the cyclic subgroup. In the Diffie-Hellman key exchange three parameters are made public: (p, g, M), where p is a prime, g is the generator of a cyclic subgroup of $\mathbb{Z}^*{}_p$ and M is the order of the cyclic subgroup generated by g. It is important that the choice of p and g yield a large value for M. A typical value of g is 2.

[6]

5.5.2 Security of Diffie-Hellman

The crucial property of the Diffie-Hellman algorithm is that an attacker having access to the public keys of both $\mathcal{A}$ and $\mathcal{B}$ is still not be able to figure out the shared key. This is so also because parties $\mathcal{A}$ and $\mathcal{B}$ create a shared secret key without either party having to send it directly to the other (as it was the case with RSA). The Diffie-Hellman algorithm is also referred to as the ephemeral secret key agreement protocol because, typically, the shared secret key is used only once.

Based on the fact that while it is relatively easy to compute the powers of an integer in a finite field, it is extremely hard to compute the discrete logarithms. This means that while it is possible to quickly compute $y = g^x \bmod p$ to determine the public key of an entity, for an attacker is computationally infeasible to figure out the private key x of the entity from the knowledge the public

[6] TODO: Slide 122..124 6-PKC_RSA-2p.pdf

parameters p, g, M and the public key y, this is so because the attacker would need to perform the following discrete logarithm calculation $x = dlog_{g,p}y$ for which an efficient algorithm is not known.

Recall that $g^s \equiv k \bmod p$ and $dlog_{g,p} = s$. If an attacker can solve $g^s \equiv k \bmod p$ with respect to s for given values of g and k, the security of the Diffie-Hellman algorithm is violated. An obvious way to solve this is by brute force enumeration, compute $g^i \bmod p$ for increasing values of i until a solution is found. The computation complexity is proportional to p. If p represented in binary requires n bits, then the complexity is proportional to 2^n and thus grows exponentially with the size of p in bits.

The most serious vulnerability of Diffie-Hellman is the man-in-the-middle attack. Assume an attacker $\mathcal{I}$ that can intercept all messages and decide to replay some of them in a possibly modified form.

1. $\mathcal{A} \rightarrow \mathcal{I} : g^a \bmod p$
2. $\mathcal{I} \rightarrow \mathcal{B} : g^{i_a} \bmod p$
3. $\mathcal{B} \rightarrow \mathcal{I} : g^b \bmod p$
4. $\mathcal{I} \rightarrow \mathcal{A} : g^{i_b} \bmod p$

Notice that the attacker is free to modify the content of the exchanged messages between $\mathcal{A}$ and $\mathcal{B}$ at will without being detected. $\mathcal{A}$ computes the shared key with whom it believes to be $\mathcal{B}$ but is $\mathcal{I}$ instead: $K_a = g^{ai_b} \bmod p$, $\mathcal{B}$ computes the shared key with whom it believes to be $\mathcal{A}$ but is $\mathcal{I}$ instead: $K_b = g^{bi_a} \bmod p$. $\mathcal{I}$ computes the same two shared keys: one with $\mathcal{A}$ and the other with $\mathcal{B}$. $\mathcal{I}$ can decrypt the messages from $\mathcal{A}$ using K_a and re-encrypt them with K_b before forwarding to $\mathcal{B}$ and vice versa from the messages received from $\mathcal{B}$ and it is able to read all the exchanged message! The problem is that Diffie-Hellman provides no authentication, so $\mathcal{A}$ can not be sure it is talking to $\mathcal{B}$ and vice versa. There is a variant of Diffie-Hellman that includes authentication called authenticated Diffie-Hellman, that consists of:

- each party acquires a certificate for the other party
- the public key that each party sends to the other party is digitally signed by the sender using the private key that corresponds to the public key on the sender's certificate
- this phase is usually performed by using RSA cryptography

Authenticated Diffie-Hellman tries to combine the best of both Diffie-Hellman and RSA cryptography: it supports forward secrecy and avoids men-in-the-middle-attacks.

Prime selection with the desirable properties for use with the Diffie-Hellman algorithm as they must yield multiplicative subgroups of large order and satisfy several other properties is not always straight forward.

5.6 Elliptic curve cryptography (ECC)

Elliptic curve cryptography (ECC) can provide the same level and type of security as RSA (or Diffie-Hellman) with much shorter keys. The computational overhead of both RSA and ECC grows as $O(N^3)$. However, it takes far less computational overhead to use ECC on account of the fact that shorter keys in ECC provide the same level of security of much longer keys in RSA 5.2.

Other advantages deriving from shorter keys:

- ECC algorithms can be implemented on smartcards without mathematical coprocessors
- Contactless smart cards work preferably with ECC because other algorithms require too much induction energy
- Since shorter key lengths translate into faster handshaking protocols, ECC is also becoming increasingly important for wireless communications and more recent technologies like wireless sensors networks

It is possible to support forward secrecy and authentication, by using ECDHE-RSA, where ECDHE is Elliptic Curve Diffie-Hellman Ephemeral. RSA is used for certificate based authentication. ECDHE is used for creating a one-time session key.

RSA is used for authentication because a majority of the certificates in use today are based on RSA public keys. This is changing as more and more organizations use ECC based certificates. These new certificates use the ECDSA algorithm for authentication. When authentication is done by ECDSA and the session key generated with ECDH or ECDHE, the combined algorithm is denoted ECDHE-ECDSA or ECDH-ECDSA.

5.6.1 ECC: Main idea

ECC is based on a generalization of the idea of repeatedly applying the group operation and the difficulty of computing how many times this has been applied. The main difference is that ECC uses points in a two dimensional space rather than integers.

Let $\mathcal{E}$ be a (finite) set of points on the plane and define a binary operation $+$ on the points of $\mathcal{E}$ that satisfies the properties of a group operator (i.e. closure, associativity, identity and invertibility). Given points $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{P} + \mathcal{Q} = \mathcal{R}$ in $\mathcal{E}$. We are interested in computing $\mathcal{G} + \mathcal{G}, \mathcal{G} + \mathcal{G} + \mathcal{G}, \ldots$, i.e. k-times addition of $\mathcal{G}$, $k * \mathcal{G}$. The crucial property that the set $\mathcal{E}$ needs to satisfy to be useful for cryptography in a sense similar to the Diffie-Hellman is that after we have calculated $k * \mathcal{G}$ for a given point $\mathcal{G}$ in $\mathcal{E}$ it is extremely difficult to recover k from $k * \mathcal{G}$, this is also called the discrete logarithm problem. We assume that the only way to recover k from $k * \mathcal{G}$ is to try every possible summation of $\mathcal{G}$.

Again we take a $\mathcal{G}$ point, and make it public. $\mathcal{A}$ chooses an integer X_a as private key and computes $Y_a = X_a * G$ and shares it with $\mathcal{B}$ that will in turn choose an integer X_b, compute $Y_b = X_b * G$ and share it with $\mathcal{A}$. Y_a, Y_b are the public keys. Finally $\mathcal{A}$ computes $K_a = X_a * Y_b$ and $K_b = X_b * Y_a$, it can be shown that $K_a = K_b$. All of the assumptions about the set $\mathcal{E}$ of points are satisfied when the points in $\mathcal{E}$ are taken from an elliptic curve.

AES	RSA/Diffie-Hellman	ECC
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Table 5.2: Comparison between RSA and other cryptographic primitives.

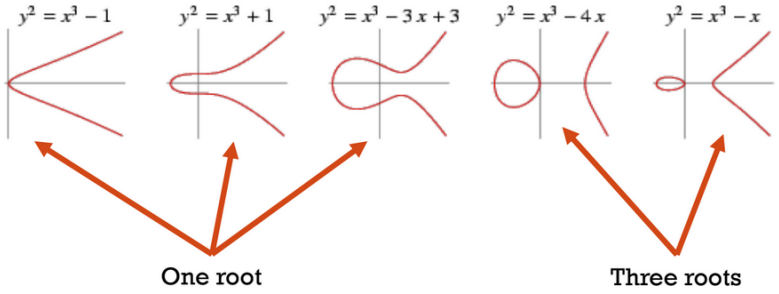


Figure 5.4

5.6.2 Remarks

Before introducing elliptic curves, let us remark that if the security of ECC depends on finding out how many times a point $\mathcal{G}$ participates in a sum like $\mathcal{G} + \mathcal{G} + \ldots + \mathcal{G}$, why would it take an attacker any more work to figure that out than it would take for a party to calculate the sum? It would seem that all that the attacker would need to do would be to keep on adding $\mathcal{G}$ to itself until the attacker obtains the value of the sum. In other words, if some integer X_a is the private key of $\mathcal{A}$ and if $\mathcal{A}$ derives its public key by adding the point $\mathcal{G}$ to itself X_a times, the amount of computational effort $\mathcal{A}$ expends in adding $\mathcal{G}$ to itself X_a times should be the same as what the attacker would need to expend if it kept on adding $\mathcal{G}$ to itself until reaching a value that is $\mathcal{A}$'s public key. To understand why it is not the case, the amount of computational effort that it takes to add a point $\mathcal{G}$ to itself X_a number of times is logarithmic in the size of X_a. This is so because $\mathcal{A}$ can compute $\mathcal{G} + \mathcal{G} = 2 * \mathcal{G}$, then adds $2 * \mathcal{G}$ to itself and gets $4 * \mathcal{G}$, and so on. However, the attacker would not know the value of X_a and so it would not be able to take advantage of such exponentially increasing jumps. Additionally, in most commonly used ECCs, all the calculations are carried out modulo a prime p. Thus, as the attacker keeps on adding $\mathcal{G}$ to itself, the size of what it gets cannot serve as a guide to how many more times the attacker must repeat that addition to get to the final value.

5.6.3 Elliptic curves

Elliptic curves are called so because of their relationship to elliptic integrals. An elliptic integral can be used to determine the arc length of an ellipse. The Weierstrass equation of characteristic 0: $y^2 = x^3 + ax + b$. Notice that elliptic curves are symmetric on the x axis 5.4.

5.6.4 Features of elliptic curves

The points on an elliptic curve can be shown to constitute a group. A group operator; an identity element with respect to the operator; closure and associativity with respect to the operator; existence of inverses w.r.t. the operator.

The group operator for the points on an elliptic curve is called addition although its definition has nothing to do with the conventional arithmetic addition 5.5.

The definition of two points $\mathcal{P}$ and $\mathcal{Q}$ is the following:

- join $\mathcal{P}$ with $\mathcal{Q}$ with a straight line
- call the third point $\mathcal{R}$ of the intersection of the straight line through $\mathcal{P}$ and $\mathcal{Q}$ with the curve, if such an intersection exists. If it does not exist we pick infinity by the special symbol $\mathcal{O}$ that

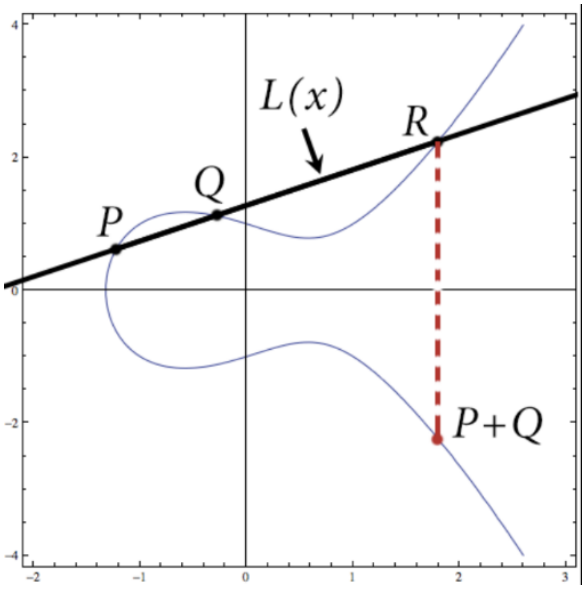


Figure 5.5

will be defined as the additive identity element for the group operator. Note that by definition $\mathcal{P} + \mathcal{O} = \mathcal{P}$ and if $\mathcal{P} + \mathcal{Q} = \mathcal{O}$ then $\mathcal{Q} = -\mathcal{P}$

- the mirror image of $\mathcal{R}$ with respect to the x-axis is the point $\mathcal{P} + \mathcal{Q}$.

We take a point $\mathcal{G}$ and add it to itself. To do it we take a second point $\mathcal{H}$, and move $\mathcal{H}$ closer and closer, in the limit the two overlap, and the line joining the two points will be tangent to the elliptic curve. So $\mathcal{G} + \mathcal{G}$ is defined as the tangent in that point, and then determine the intersection of the tangent with the curve, and then take the reflection of such intersection with respect to the x-axis. Take two points, then move $\mathcal{Q}$ closer and closer to $\mathcal{P}$. Take the generator point $\mathcal{G}$, take the tangent, and obtain another point on the curve, then reflect it, and calculate the tangent, and so on.

It is possible to show that inverting the addition of points is difficult.

$E(a, b)$ denotes the set of points belonging to the elliptic curve described by $y^2 = x^3 + ax + b$ extended with the point $\mathcal{O}$ at infinity. $E(a, b)$ can be shown to be closed under the operation of addition as defined above, that it has an inverse for every element, and the addition is associative. In other words, $E(a, b)$ is a group. [7]

5.7 Elliptic curves over finite fields

This kind of arithmetic computations cannot be used for cryptography because calculations with real numbers are prone to round-off error whereas cryptography requires error-free arithmetic. The solution to this problem is to restrict the values of the parameters a and b, the value of the independent variable x, and the value of the dependent variable y to some finite field Z_p for p a prime number. In other words, we consider elliptic curves that satisfy the following congruence: $y^2 \equiv x^3 + ax + b \bmod p$. Security depends on the selection of a and b.

[7] TODO: Slide 151..153 6-PKC_RSA-2p.pdf

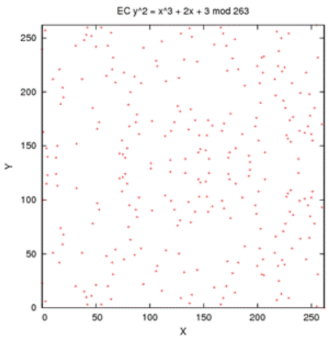


Figure 5.6: EC over finite fields

An elliptic curve over a finite filed looks like some points 5.6, where each point belonging to the curve and has integer coordinates.

Similarly to what we have done for elliptic curves over the reals, we denote by $E_p(a,b)$ the set of points that satisfy $y^2 \equiv x^3 + ax + b \bmod p$. Indeed $E_p(a,b)$ is no longer the set of real points belonging to a curve, but a collection of discrete points in the Cartesian product. Since the points in $E_p(a,b)$ no longer form a curve, we cannot use the geometrical construction to illustrate the action of the addition operator. Luckily, the algebraic expressions derived for these operations continue to hold good provided the calculations are carried out modulo p.

It possible to count the number of points in an elliptic curve over a finite field by using Schoof's algorithm, knowing the number of points is very important for judging the difficulty of solving the discrete logarithm problem in the group of points of the elliptic curve. Hasse theorem provides an estimate of the number of points on an elliptic curve over a finite field, bounding the value both above and below. The higher the points on an EC the better it is! Of particular interest, it is the definition of elliptic curves over Galois fields, i.e. over $GF(2^n)$. For these curves to be cryptographically secure, it is mandatory to take n a prime. Recall that over Galois fields, addition is equivalent to bit-wise xor. This implies that $x+x=0$ that in turn implies that the field has characteristic 2. This prevents the use of the class of elliptic curves described by the congruence $y^2 \equiv x^3 + ax + b \bmod p$ since they become singular and this is detrimental to security.

The elliptic curve equation to use when the underlying field is described by $GF(2^n)$ is the following $y^2 + xy = x^3 + ax + b$. [8]

5.8 Elliptic Curve Cryptography

Just as RSA uses exponentiation, i.e. repeated multiplication, as its basic arithmetic operation, ECC uses the addition group operator as its basic arithmetic operation. Assume $\mathcal{G}$ is a point, chosen by a user, over an elliptic curve $E_q(a,b)$ where q is a prime (finite field over a prime) or $q = 2^n$ for n a prime (Galois field). With an appropriate choice for $\mathcal{G}$, whereas it is relatively easy to calculate $\mathcal{C} = \mathcal{M} * \mathcal{G}$ for an integer $\mathcal{M}$ encoding a plaintext it is infeasable to derive $\mathcal{M}$ from $\mathcal{C}$ in the case of knowing $\mathcal{G}$ and $E_q(a,b)$. Deriving it is called the discrete logarithm problem. An attacker could try to recover $\mathcal{M}$ from $\mathcal{C} = \mathcal{M} * \mathcal{G}$ by calculating $2 * \mathcal{G}, 3 * \mathcal{G}, 4 * \mathcal{G}, \ldots, k * \mathcal{G}$ with increasing values of k. In the worst case, the process will span all points in the elliptic curve $E_q(a,b)$ and requires to check

[8] TODO: Slide 161..163 6-PKC_RSA-2p.pdf

whether or not the result is equal to $\mathcal{C}$. If q is sufficiently large and the point $\mathcal{G}$ on the curve $E_q(a, b)$ is chosen carefully, this would take too long in practice and it can thus be considered as infeasible.

5.9 ECDH: Elliptic Curve Diffie-Hellman

The goal is to establish a secret session key between two parties. A community of users wishing to engage in secure communications with ECC chooses the parameters q, a, and b for an elliptic curve $E_q(a, b)$ together with a base point $\mathcal{G}$ on the curve. Recall that q can either be a prime or a power of 2.

- $\mathcal{A}$ selects an integer $\mathcal{X}_a$ as its private key and compute $\mathcal{Y}_a = \mathcal{X}_a * \mathcal{G}$ as its public key.
- $\mathcal{B}$ selects an integer $\mathcal{X}_b$ as its private key and compute $\mathcal{Y}_b = \mathcal{X}_b * \mathcal{G}$ as its public key.
- $\mathcal{A}$ and $\mathcal{B}$ exchange their public keys
- $\mathcal{A}$ computes $\mathcal{K}_a = \mathcal{X}_a * \mathcal{Y}_b$ and $\mathcal{B}$ computes $\mathcal{K}_b = \mathcal{X}_b * \mathcal{Y}_a$.

  Turns out that $\mathcal{K}_a = \mathcal{K}_b$.

5.10 ECDSA: EC digital signature algorithm

Consider a finite fields Z_p over a large prime p... [9]

5.11 Security of ECC

Certain classes of curves are weak (singular, super-singular curve). It is possible to reduce the problem of knowing how many times to add the generator to itself to computing the discrete logarithm in a finite field. To avoid this attack the elliptic curve has to satisfy the MOV condition.

When using Galois fields $GF(2^n)$ another security consideration relates to what is known as the Weil descent attack. To avoid this attack n has to be prime.

Elliptic curves for which the total number of points on the curve equals the number of elements in the underlying finite field are also considered cryptographically weak.

There are standards that help choosing safe to use elliptic curves. [10]

[9] TODO: Slide 169..170 6-PKC_RSA-2p.pdf
[10] TODO: Slide 172..174 6-PKC_RSA-2p.pdf

Chapter 6

Hash Functions

6.1 Introduction

A hash function takes a variable sized input message and produces a fixed-sized output. Every cryptographic hash function is a hash function. Not every hash function is a cryptographic hash function. Non-cryptographic hash functions try to avoid collisions for (non-malicious) input. Cryptographic hash function need to be very fast to compute, have a very low probability of collisions and it has to be computationally infeasible to invert (e.g. given an hash say what was the value that generated that hash).

The digest can be seen as a fixed-sized fingerprint of a variable-sized message. Since a message digest depends on all the bits in the input message, any alteration of the input message during transmission would cause its message digest to not match with its original message digest.

6.2 Characterizing cryptographic hash functions

A hash function is called cryptographically secure if it is computationally infeasible to find:

- a message that corresponds to a given digest, this is called the one-way property of a hash function. Remark: a hash function must possess this property regardless of the length of the messages. It should be just as difficult to recover from its digest a message that is as short as, a single byte as a message that consists of millions of bytes.
- two different messages that hash to the same digest, this is called the strong collision resistance property of a hash function.

Both of these conditions need to be true. A weaker form of the strong collision resistance property is that for a given message, there should not correspond another message with the same digest. This is called the weak collision resistance property of a hash function. Collision resistance refers to the likelihood that two different messages will result in the same digest. If the digest produced by a hash function consists of n bits, then there are only 2^n distinct digests. If we place no constraints on the messages and if there can be an arbitrary number of different possible messages, then obviously there will exist multiple messages giving rise to the same diges. An example cryptographic hash function is SHA-512.

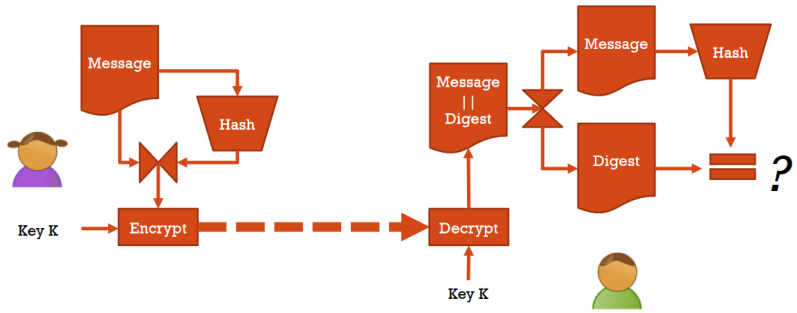


Figure 6.1: Hash functions for message authentication

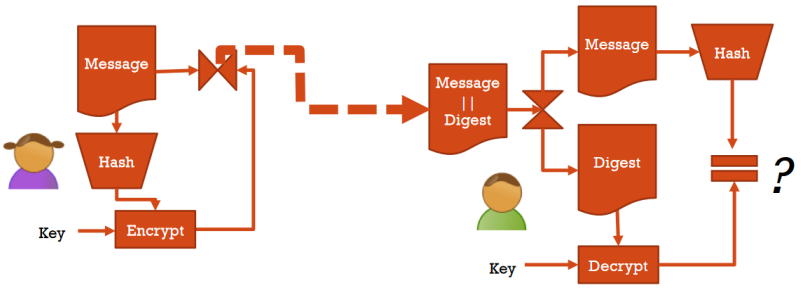


Figure 6.2: Hash functions for message authentication (2)

6.3 Applications of Hash functions

The main applications are authenticity 6.1 and integrity.

We assume that symmetric key cryptography is used.

- Both Alice and Bob use the same shared key $\mathcal{K}$
- Alice concatenates the message and its digest to form a composite message that is then encrypted and sent over the network
- Bob decrypts the message and separates out its digest, which is then compared with the digest calculated from the received message
- The digest provides authentication and the encryption provides confidentiality

Interpretation 1: symmetric key cryptography is used 6.2

- Variant of previous schema where only the digest is encrypted
- This scheme is efficient to use when confidentiality is not an issue but message authentication is critical
- Only the receiver with access to the secret key knows the real digest for the message and can verify whether or not the message is authentic
- A digest produced in this way is known as the Message Authentication Code (MAC) and the hash function producing it as a keyed hash function (more on this below)

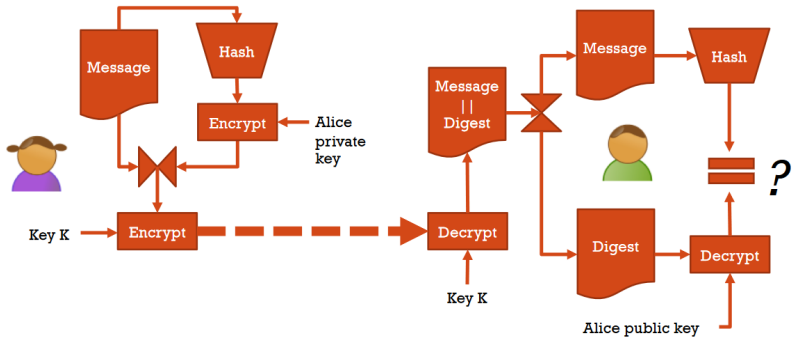


Figure 6.3: Hash functions for message authentication (3)

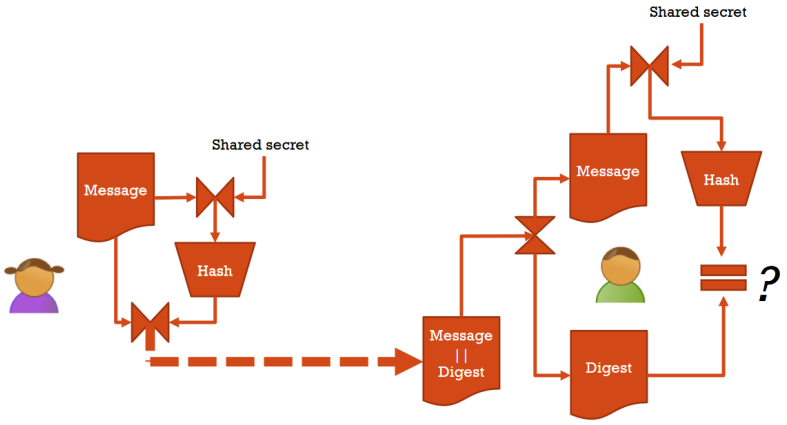


Figure 6.4: Hash functions for message authentication (4)

Interpretation 2: public key cryptography is used 6.2

- Alice uses its private key while Bob uses Alice public key
- Confidentiality is not considered
- The sender encrypting with its private key the digest of the message constitutes the basic idea of digital signatures, as explained in previous lectures

We assume the availability of both public and symmetric key cryptography 6.3.

- Commonly used approach when both confidentiality and authentication are important
- Alice encrypts the digest of the message with its private key, concatenates it with the message and then encrypts the result with the shared key
- Bob decrypts the received message with the shared key, calculates the digest from the extracted message and compares it with the decrypted digest by using Alice public key

We do not need to assume the availability of neither public nor symmetric key cryptography 6.4.

- In this scheme, no message is encrypted

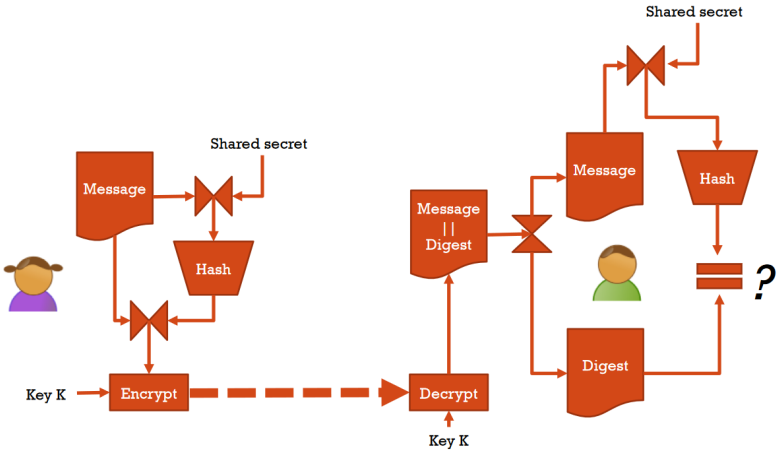


Figure 6.5: Hash functions for message authentication (5)

- Alice appends a shared secret string known also to Bob, to the message before computing its digest
- Before checking the digest of the received message for its authentication, Bob appends the same shared secret string to the message
- It would not be possible for anyone to alter such a message, even when they have access to both the original message and the overall digest
- Notice that in this case, confidentiality is not considered

We assume the availability of symmetric key cryptography 6.5.

- This scheme is similar to the previous one with the addition of symmetric key cryptography for confidentiality
- Alice appends a shared secret string known also to Bob, to the message before computing its digest
- Before checking the digest of the received message for its authentication, Bob appends the same shared secret string to the message
- It would not be possible for anyone to alter such a message, even when they have access to both the original message and the overall digest

6.4 Authenticated encryption with additional data (AEAD)

When using symmetric ciphers, it is crucial to avoid replay attacks. For example assume Alice sends a message to Bob containing "You can take holidays tomorrow" encrypted with key K. An Bob takes holidays as expected. The next day the attacker replays the message and Bob takes another day of vacation, and Alice is left wondering why Bob is not at work.

To avoid the replay attack, what is needed is to bind the cipher to a network connection or a session, in order that Eve cannot recreate the same scenario. With enhanced encryption methods,

known as Authenticated Encryption with Associated Data (AEAD), it is possible to both authenticate the cipher and prove its integrity. For this we provide additional data to authenticate the encryption process, and where we can identify where the ciphertext has been modified, and for it not to be decrypted. With most conventional AEAD methods we create a nonce value and add additional data (AD) that is authenticated but not encrypted. Additional data can include addresses, ports, sequence numbers, protocol version. Additional Data allows for binding network packets to the encrypted data, and provide integrity, so that an intruder cannot copy and paste valid ciphertext from other transmissions. If we bind to a packet sequence number and the port, the authentication would fail for another sequence number or another port. Indeed, the wider the range of additional data, the more difficult it will be for an attacker to replicate it. Examples of AEAD are AES-GCM and ChaCha20/Poly1305.

Encryption takes as input the plaintext, key, and optionally a header in plaintext that will no be encrypted, but will be covered by authenticity protection. Its output is the ciphertext and authentication tag (Message Authentication or MAC). Decryption takes as input the ciphertext, key, authentication tag, and optionally a header (if used during the encryption). Its output is the plaintext, or an error if the authentication tag does not match the supplied ciphertext or header. The header part is intended to provide authenticity and integrity protection for networking or storage metadata for which confidentiality is unnecessary, but authenticity is desired.

The need for authenticated encryption emerged from the observation that securely combining separate confidentiality and authentication block cipher operation modes could be error prone and difficult. This was confirmed by a number of practical attacks introduced into production protocols and applications by incorrect implementation, or lack of authentication (including SSL/TLS).

6.5 Structure of hash functions

All algorithms for computing the digest of a message view it as a sequence of n-bit blocks. The message is processed one block at a time in an iterative fashion in order to generate its digest. The simplest hash function consists of starting with the first n-bit block, perform the bitwise xor with the second n-bit block perform the bitwise xor with the next n-bit block and so on... This procedure is known as the xor algorithm or the longitudinal parity check as every bit of the digest represents the parity at that bit position if we look across all of the n-bit blocks. The digest generated by the xor algorithm can be useful as a data integrity check in the presence of completely random transmission errors. In the presence of an adversary trying to deliberately tamper with the message content, the xor algorithm is useless for message authentication. An adversary can modify the main message and add a suitable bit block before the digest so that the final digest remains unchanged.

When hashing regular text and the character encoding is based on ASCII, the collision resistance property of the xor algorithm suffers even more because the highest bit in every byte will be zero. Ideally, one would hope that, with an n-bit digest, any particular message would result in a given digest value with a probability of 2^{-n}. When the highest bit in each byte for each character is always 0, some of the n bits in the digest will predictably be 0 with the xor algorithm. This reduces the number of unique digests available and thus increases the probability of collisions. An attempt to avoid this problem is a variation on the basic xor algorithm that consists of performing a one-bit circular shift of the partial digest obtained after each n-bit block of the message is processed, this algorithm is known as the rotated XOR, but it suffers to a similar problem. XOR should be avoided as the right way out is to include the length of the message in the digest.

6.6 Birthday attack

We are interested in computing the probability that any of the messages is going to have its digest equal to a particular value h. More formally: given a pool of k messages produced randomly by the message generator, what is the value of k so that the pool contains at least one message whose digest is equal to h with probability 1/2?

To establish k consider that the digest can take on N different but equiprobable values, and pick a message x at random from the pool of messages. Since all N digests are equiprobable, the probability of a message x having its digest equal to h is $\frac{1}{N}$. Since the digest of message x is either equal to h or not, the probability of the latter is $1 - (\frac{1}{N})$. If we pick, say, two messages x and y randomly from the pool, the events that the digest of neither is equal to h are probabilistically independent. This implies that the probability that none of two messages has its digest equal to h is $(1 - (\frac{1}{N}))^2$. More generally, in a pool of k messages, the probability that none of the messages in a pool of k messages has its digest equal to h is $(1 - (\frac{1}{N}))^k$. Thus the probability that at least one of the k messages has its digest equal to h is $1 - (1 - \frac{1}{N})^k$. This can be simplified to $\frac{k}{N}$ by observing that $(1+a)^n \approx 1 + a * n$ as n gets close to 0. So, given a pool of k randomly produced messages, the probability there will exist at least one message in this pool whose digest is equal to a given value h is $\frac{k}{N}$. To answer the original question it is sufficient to solve the equation $\frac{k}{N} = 1/2$ i.e. $k = \frac{N}{2}$.

Consider a digest of 64 bits, then $N = 2^{64}$ and $k = N/2 = 2^{63}$, so we must construct a pool of 2^{63} messages so to be able to find in it a digest equal to h with probability of 50%.

In a more friendlier way, what is the probability that, in a class of 20 students, someone else has the same birthday as yours? The answer is $19/365 = 5.2\%$..

Next, we are interested in computing, given a pool of randomly selected messages, the probability that at least two messages in the pool have the same digest. More precisely: given a pool of k messages produced randomly by the message generator, what is the probability that there exist at least two messages in the pool with the same digest?

Let M_1 be be the total number of ways in which we can construct a pool of k message with no duplicate digests. Let M_2 be the total number of ways in which we can construct a pool of k messages allowing for duplicates. Then $\frac{M_1}{M_2}$ is the the probability of constructing a pool of k messages with no duplicates. Subtracting this from 1 yields the probability that the pool of k messages will have at least one duplicate digest. For M_1 we need to find out in how many different ways we can construct a pool of k messages so that we are guaranteed to have no duplicate digests in the pool. For the first message there are N different candidates, for the second $N - 1$ and so on, so in total $N * (N - 1) * \ldots * (N - k + 1)$ that is equivalent to

$$\binom{N}{k} = \frac{N!}{(N-k)!}$$

For M_2 we need to figure out he total number of ways in which we can construct a pool of k messages without worrying at all about duplicate digests. There are N ways to choose the first message to insert in the pool, for the second message there are still N ways, so in total N^k.

Thus we have:

$$\frac{M_1}{M_2} = \frac{N!}{(N-k)! * N^k}$$

. To answer our question of the probability that the pool of k elements has at least one duplication in the digest is:

$$1 - \frac{N!}{(N-k)! * N^k}$$

.

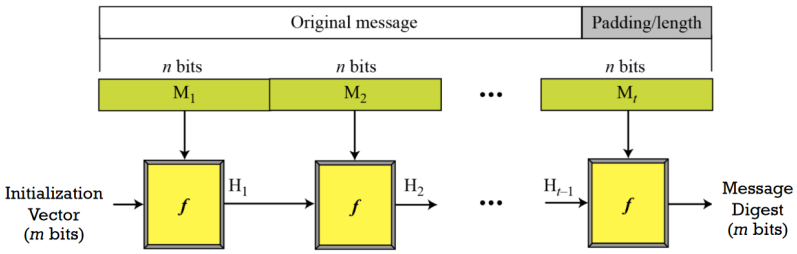


Figure 6.6: Hash function

To estimate the size k of the pool so that the pool contains at least one pair of messages with equal digests with a probability of 0.5 is $k \approx 1.18 * \sqrt{N}$, when k is large. Considering the case in which the digest have size 64 bits, then $N = 2^{64}$. We need to create a pool of $1.18 * 2^{32}$ messages so that the probability that two messages share the same digest with a probability of 50%.

In a more friendlier way, what is the probability that there exists at least one pair of students in a class of 20 students with the same birthdays? The answer is $190/365 = 52\%$. If we increase the number of students to 60, then the probability becomes 90%. [1]

6.7 Secure Hash functions

A hash function is secure if: it is computationally infeasible to find collisions, so it is computationally infeasible to construct messages whose digest would equal a specified value, and it is strictly one way, so it lets us compute the digest of a message, but does not let us figure out a message for a given digest (even for very short messages).

The input message is partitioned into t number of bit blocks, each of size n bits. If necessary, the final block is padded so that it is of the same length as others 6.6. The final block also includes the total length of the message whose hash function is to be computed. This enhances the security of the hash function since it places an additional constraint on the counterfeit messages.

Each stage of the Merkle structure takes two inputs: the n-bit block of the input message meant for that stage and the m-bit output of the previous stage. For the m-bit input, the first stage is supplied with a special m-bit pattern called the Initialization Vector (IV). The function f that processes the two inputs, one n bits long and the other m bits long, to produce an m bit output is usually called compression function. This is so since $n > m$, i.e. the output of the function f is shorter than the length of the input message segment. The compression function f may involve multiple rounds of processing of the two inputs to produce the output. The precise nature of f depends on what hash algorithm is being implemented. The better f is designed the better the hash function will be.

6.8 The SHA family

The last column refers to how many messages would have to be generated before two can be found with the same digest with a probability of 0.5 6.7. As shown above, for a secure hash algorithm that has no security holes and that produces n-bit digests, one would need to come up with $2^{n/2}$ messages to discover a collision with a probability of 0.5.

[1] TODO: Slide 44..47 10-HASH-2p.pdf

		Message size (bits)	Block size (bits)	Word size (bits)	Message digest (size)	Security (bits)
	SHA-1	<2^64	512	32	160	80
SHA-2	SHA-256	<2^64	512	32	256	128
SHA-2	SHA-384	<2^128	1,024	64	384	192
SHA-2	SHA-512	<2^128	1,024	64	512	256

Figure 6.7: SHA overview

6.8.1 SHA-1

SHA-1 was cracked theoretically in the year 2005 by two different research groups, in 2017, SHA-1 was broken in practice shattered.io.

6.8.2 SHA-512

SHA-512 has the following structure 6.6. It has 4 steps.

SHA-512: Step 1

This step consists in padding with length value. The goal is to make pad the message so that its length is an integral multiple of the block size = 512 bits. Proviso: the last 128 bits of the last block must contain a value that is the length of the message. Even if the original message were by chance to be an exact multiple of 512, one still needs to append another 512-bit block at the end to make room for the 128-bit message length integer. Leaving aside the trailing 128 bit positions, the padding consists of a single 1-bit followed by the required number of 0-bits.

SHA-512: Step 2

The goal is to initialize the hash buffer with IV. The hash buffer is represented by 8 registers of 64-bits that are labelled by a, b, c, d, e, f, g, h.

Generate the message schedule required for processing a 1024-bit block of the input message. The message schedule consists of 80 64-bit words: the first 16 of these words are obtained directly from the 1024-bit message block, the rest of the words are obtained by applying permutation and mixing operations to some of the previously generated words.

SHA-512: Step 3

Apply round-based processing to each 1024-bit input message block 6.8. There are 80 rounds to be carried out for each message block. For round-based processing store the hash values calculated for the previous message block in temporary 64-bit variables a, b, c, d, e, f, g, h. In the i-th round, permute the values stored in these eight variables and, with two of the variables, mix in the message schedule words and a round constant. In the round-based processing of each input message block, the i-th round is fed the 64-bit message schedule word W_i and a special constant K_i. These constants are meant to be random bit patterns to break up any regularities in the message blocks. The round function consists of a sequence of transpositions and substitutions designed to diffuse to the maximum extent possible the content of the input message block 6.9.

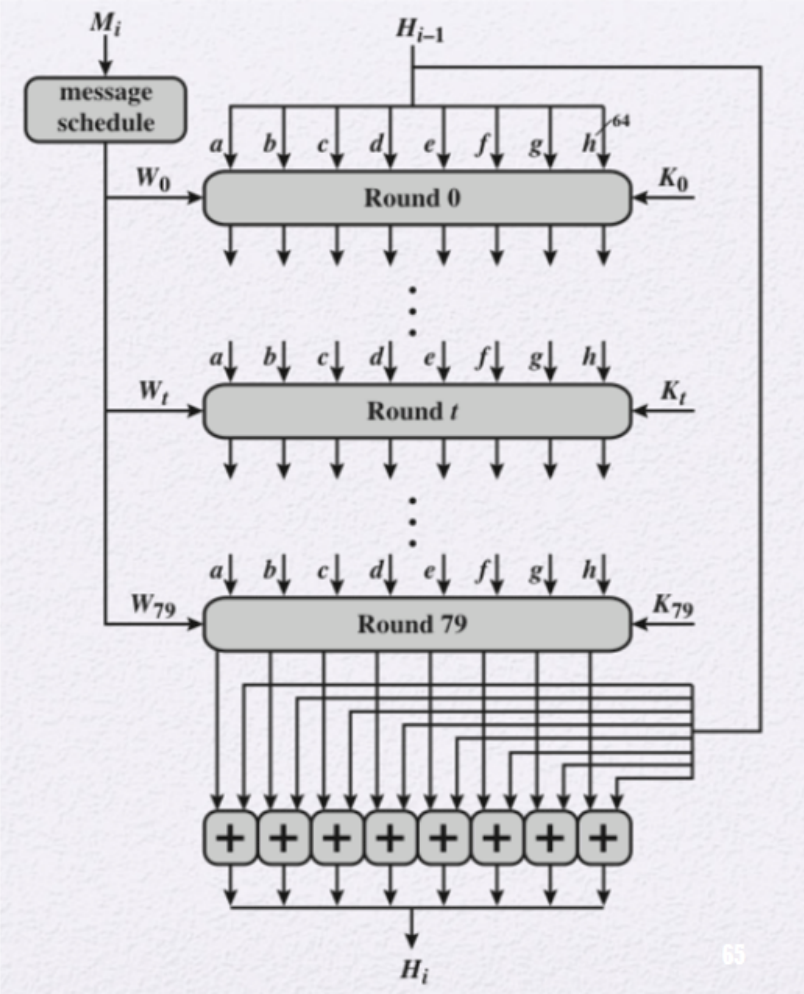
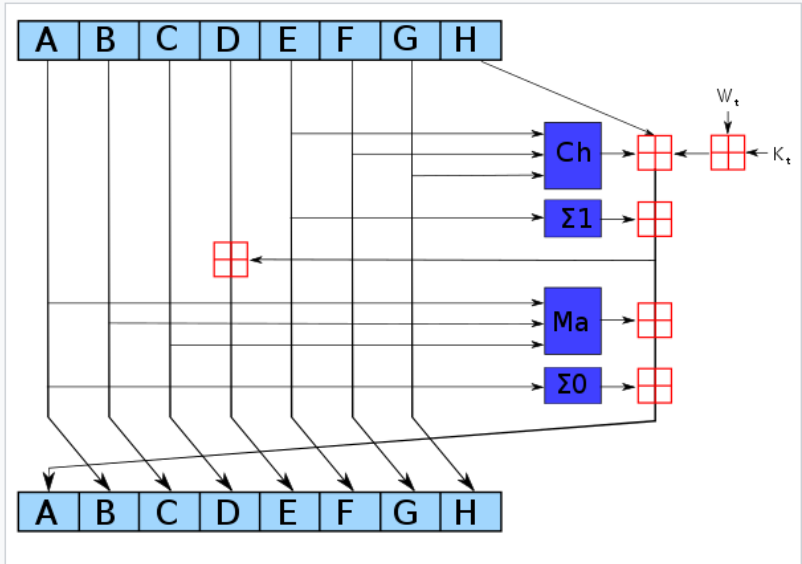


Figure 6.8: Function f



One iteration in a SHA-2 family compression function. The blue components perform the following operations:

$$\mathrm{Ch}(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$$
$$\mathrm{Ma}(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$$
$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$
$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

The bitwise rotation uses different constants for SHA-512. The given numbers are for SHA-256.
The red ⊞ is addition modulo 2^{32} for SHA-256, or 2^{64} for SHA-512.

Figure 6.9: SHA round function definition

SHA-512: Step 4

After all the t message blocks have been processed, the content of the hash buffer is the message digest.

6.9 Final remarks on hash functions

Cryptographic hash functions should have three properties:

- Pre-image resistance: given a digest $h(a)$ it is computationally infeasible to find b such that $h(b) = h(a)$. Functions that lack this property are vulnerable to a pre-image attack
- Second pre-image resistance: given a it is computationally infeasible to find b such that $h(a) = h(b)$. Functions that lack this property are vulnerable to a second pre-image attack
- (Strong) Collision resistance: it is computationally infeasible to find a pair (a, b) where $a \neq b$ such that $h(a) = h(b)$. For functions that lack this property a collision could be found via a birthday attack

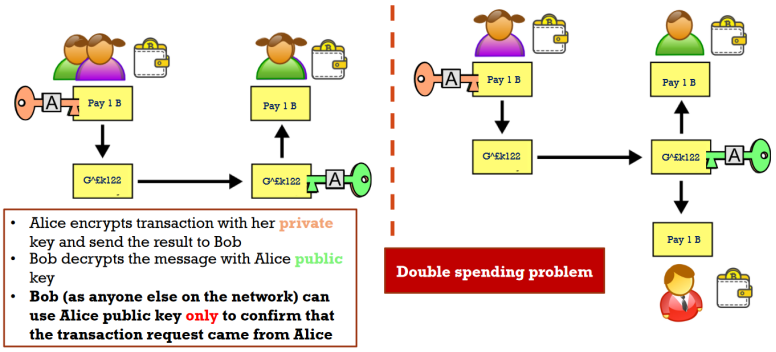


Figure 6.10: A transaction in the blockchain

In reality, hash functions are not close to one-way functions for which: it is easy to compute the digest, but it is computationally hard to compute a preimage. There is no proof that one-way hash functions exist, or even concrete evidence that they can be constructed. Pragmatically, there are example hash functions (e.g., SHA-2 and SHA-3) that seem to be one-way hash functions that are easy to compute but we know of no easy way to find a preimage.

Despite all these issues, for evaluating the theoretical security of hash functions, the random oracle model is used. A random Oracle is a function that gives fixed length output. If it is the first time that it receives a particular input, it gives random output. If it already has received the input, it will repeat the corresponding output. This is the ideal strongly collision resistant function. It can be used to prove security of encryption/signing under the assumption that the functions used behave like a random oracle. This is usually treated as strong evidence rather than a real proof. The evidence falls if weaknesses are found in the functions used.

6.10 Application to cryptocurrencies

Cryptocurrency is a digital currency in which encryption techniques are used to regulate the generation of units of currency and verify the transfer of funds, operating independently of a central bank 6.10.

The solution of the double spending problem is: mining. Mining is the process of issuing bitcoins. By proof of work we mean solving mathematical puzzles based on the notion of (cryptographic) hash function. Example find the number appended to "Hello, world!" that gives an hash that starts with 4 zeros (e.g. SHA256("Hello, world!4250") = 0000c3af42...). Since there is no way of predicting what the output will be, one has to guess. While it is hard to find a solution for the puzzle, it is easy to verify that the solution is correct. The mines that finds the correct solution gets rewarded bitcoins.

Hashing finds extensive use in crypto currency algorithms:

- Used in the creation of bitcoin addresses to improve security and privacy.
- Proof-of-work calculations for mining a new coin
- Authenticate ownership of coins
- Establish a protocol for transferring the ownership from one node to another in a crypto currency network

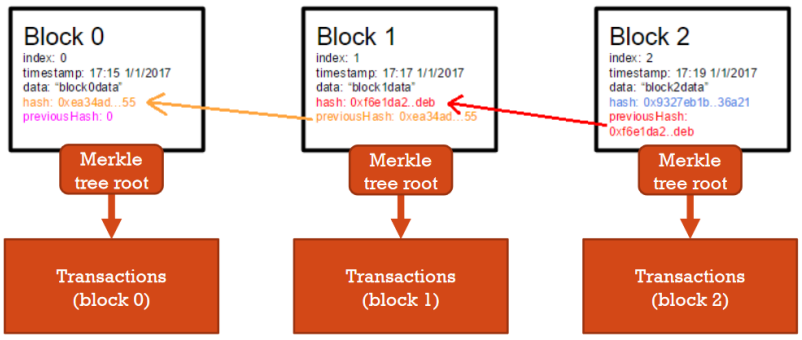


Figure 6.11: Blockchain data structure

- Create a protection against the double spending problem: no node should be able to sell the same coin to multiple other nodes in the network

Coins in cryptocurrencies are rare software object. The requirements are: it must be extremely difficult to produce the object and it must be extremely easy to verify that it was extremely difficult to produce the object.

Bitcoin blockchain can be thought of as a decentralized database that is managed by distributed computers on a peer-to-peer network. Each peer maintains a copy of the ledger: updates and validations are reflected in all copies simultaneously. A consensus algorithm is a process used to achieve agreement on a single data value among distributed processes or systems. Consensus algorithms are designed to achieve reliability in a network involving multiple unreliable nodes. Although any party can submit a chain of blocks to the ledger, the amount of computing resources required to fake consensus is too great to make it worthwhile to a dishonest party 6.11. The notion of a block as a set of transactions and, even more importantly, the notion a blockchain that is used to implicitly embed in each block a hash of all the previous transactions carried out so far. This provides security against the double spending problem and makes it possible to use crypto currencies in a manner similar to real currencies. [2].

6.11 Fault tolerant consensus

A group of generals must decide whether to attack or retreat. Some generals may prefer to attack, while others prefer to retreat. The important thing is that every general agrees on a common decision, for a half-hearted attack by a few generals would be much worse than either a coordinated attack or a coordinated retreat. The problem is complicated by the presence of treacherous generals who may not only cast a vote for a suboptimal strategy, they may do so selectively. Example: if 9 generals are voting, 4 of whom support attacking while 4 others are in favor of retreat, the 9th general may send a vote of retreat to those generals in favor of retreat, and a vote of attack to the rest... The problem is complicated further by the generals being physically separated and having to send their votes via messengers who may fail to deliver votes or may forge false votes.

In the simplified version of this problem two generals need to agree on the time to attack. General 1 has to send a messenger across the enemy's camp that will deliver the time of the attack to General 2. There is a possibility that the messenger will get captured by the enemies and thus the message

[2] TODO: Slide 90..111 10-HASH-2p.pdf

will not be delivered resulting in General 1 attacking while General 2 not. Even if the first message goes through, General 2 has to acknowledge that he received the message, so he sends a messenger back, thus repeating the scenario where the messenger can get caught. This extends to infinite chains of acknowledgements with generals unable to reach agreement.

Byzantine Fault Tolerance is the characteristic which defines a system that tolerates the class of failures that belong to the Byzantine Generals Problem. Bitcoin is Byzantine Fault Tolerant by solving the Byzantine Generals Problem by using the Proof-of-Work approach.

Chapter 7

E2EE & TOR

7.1 End to End encryption

The overall goal of Information Security is preserving the integrity and/or confidentiality of the data from its sending point to the receiving point in a network. Information Security must also provide for the receiver to be certain that the sender is actually the entity that the information was received from (authenticity). In the context of networks, these properties can be achieved by using the notion of end-to-end encryption. Not only the communication stays encrypted during transport but also that the provider of the communication service is not able to decrypt the communications either by having access to the private key or by having the capability to undetectably inject an adversarial public key as part of a man-in-the-middle attack. Encryption-In-Transit is different from End-to-End encryption because in the Encryption-In-Transit the communication is encrypted only from client to server, and vice versa, but the server can decrypt the communication. In the case of End-To-End encryption, the communication stays encrypted even when transiting on the server.

Weakening/suppression of E2EE is dangerous for democracy and can have a negative impact on our fundamental rights and freedom.

7.1.1 Security of E2EE

It is still vulnerable to Man-in-the-Middle attack, if the attacker can interfere with the key exchange protocol. To prevent this, Certificates (assuming a public key infrastructure is available), Web-of-trust, Fingerprints (short sequence of bytes used to identify longer public keys) are used.

E2EE does not solve the problem that users' computer can still be hacked and private keys can be stolen that way.

E2EE should guarantee the following desired properties:

- Authentication: when information is received from a source, authentication means that the source is indeed as alleged in the information. The information was not altered along the way (integrity)
- Confidentiality: the information is safe from being eavesdropped on during its transit from the sending point to the receiving point
- Choosing the best security parameters and key management. The choice refers to the fact that, in general, the two endpoints of a communication link may possess different computational

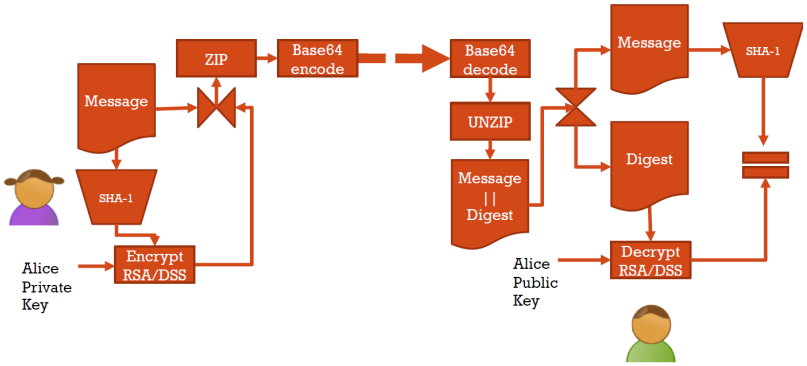


Figure 7.1: PGP Mode Authentication Only

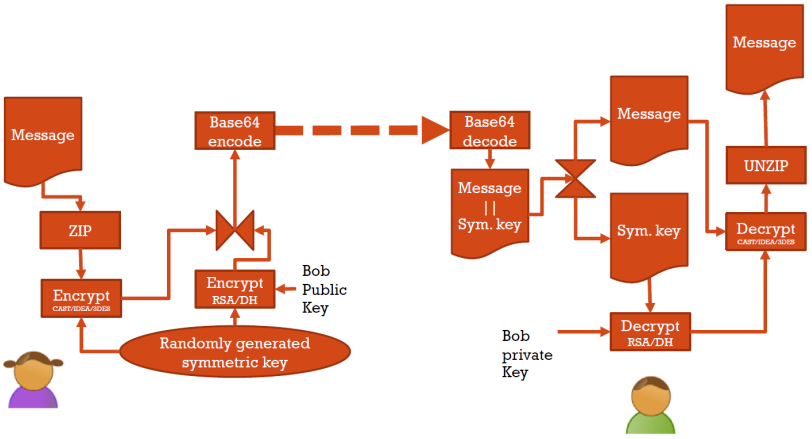


Figure 7.2: PGP Mode Confidentiality Only

capabilities and, also, in general, may not have access to exactly the same set of security algorithms. Key management refers to providing solutions to the sort of practical problems that arise when users possess multiple public/private key pairs

7.2 PGP

7.2.1 Authentication

Sender authentication consists of the sender attaching its digital signature to the email and the receiver verifying the signature using public-key cryptography 7.1.

7.2.2 Confidentiality

PGP uses symmetric-key encryption for confidentiality. The CAST-128,IDEA,3DES block ciphers are supported 7.2.

The block ciphers are used in the Cipher Feedback Mode (CFB) 3.7.4. The encryption key

(session key) is generated for each email message separately. The session key is encrypted using RSA with the receiver's public key. Alternatively, the session key can also be established using the El Gamal algorithm.

The message sent over the network is the email message after it is encrypted first with the session key and then with the receiver's public key. If confidentiality and sender-authentication are needed simultaneously, a digital signature for the message is generated using the hash code of the message plaintext and appended to the email message before it is encrypted with the session key.

7.3 PGP Services

7.3.1 Compression

By default, PGP compresses the email message after appending the signature but before encryption. This makes long-term storage of messages and their signatures more efficient. This also decouples the encryption algorithm from the message verification procedures. Compression is carried out with the ZIP algorithm.

7.3.2 Compatibility

Since encryption, even when it is limited to the signature, results in arbitrary binary strings. Since network message transmission is character oriented, one must represent binary data with ASCII strings. PGP uses Base64 encoding for this purpose. Base64 is a widely used encoding method to guarantee interoperability across different platforms and exchange binary data over networks.

7.3.3 Segmentation

For long email messages (generally messages containing attachments), many email systems place restrictions on how much of the message will be transmitted as a unit. For example, some email systems segment long email messages into 50,000 byte segments and transmit each segment separately. PGP has built-in facilities for such segmentation and re-assembly.

7.3.4 Message format

The PGP message format 7.3.

7.4 Web of Trust

Public key encryption is central to PGP as it is used for both authentication and for confidentiality. A sender uses its private key for placing its digital signature on the outgoing message. A sender uses the receiver's public key for encrypting the symmetric key used for content encryption for ensuring confidentiality. People are likely to have multiple public and private keys for a number of different practical reasons. For example, an individual may wish to drop an old public key, but, to allow for a smooth transition, may decide to make available both the old and the new public keys for a while. PGP must allow for the possibility that the receiver of a message may have stored multiple public keys for a given sender. There are two questions: PGP uses one of the public keys made available by the recipient, how does the recipient know which public key it is? Next, the sender uses one of the multiple private keys that at its disposal for signing the message, how does the recipient know

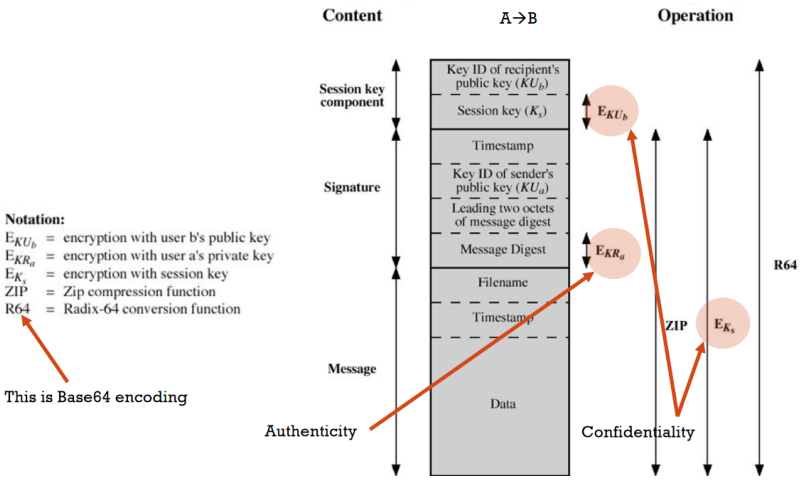


Figure 7.3: PGP message format

which of the corresponding public keys to use? Both of these problems can be solved by the sender also including the public key used, this is indeed a waste in space because RSA public keys can be very large (and getting larger and larger for security as time went by).

PGP solves these problems by using the notion of a relatively short key identifier (called key ID) and requiring that every PGP agent maintain its own list of paired private/public keys in what is called as a Private Key Ring a list of the public keys for all its email correspondents in what is called as the Public Key Ring. Key IDs are included in messages so that the recipient of the message knows which public key to use by using its Public Key Ring. A Private Key Ring contains the public/private key pairs for a user.

The private key ring contains:

- Timestamp: date/time when key pair generated
- Key ID: least significant 64 bits of public key
- Public key: public-key portion of the pair
- Private key: private key portion of the pair this field is encrypted

The public key ring contains: Timestamp, Key ID, Public Key, and User ID have the same meaning as in the private key ring. The remaining fields are used for handling trust Owner Trust, Key Legitimacy, Signature(s), and Signature Trusts are used to assess how much trust to place in the public keys belonging to other people.

How to measure the degree of trust is implementation dependent. A can fully trust B key if A and B have exchanged the keys manually (e.g., using USB keys) and so on. A unique feature of PGP is its own notion of a "certificate authority" for authenticating the binding between a public key and its owner. This notion is based on PGP's web of trust that is a bottom-up approach to establishing trust for authentication. Notice that this is in contrast with the approach used in PKI that is top-down since trust can only flow downwards from the root node (that must always be trusted implicitly) to the CAs at the other nodes that descend from the root node. In PGP's web of trust, a user's public key can be signed by any other use 7.4.

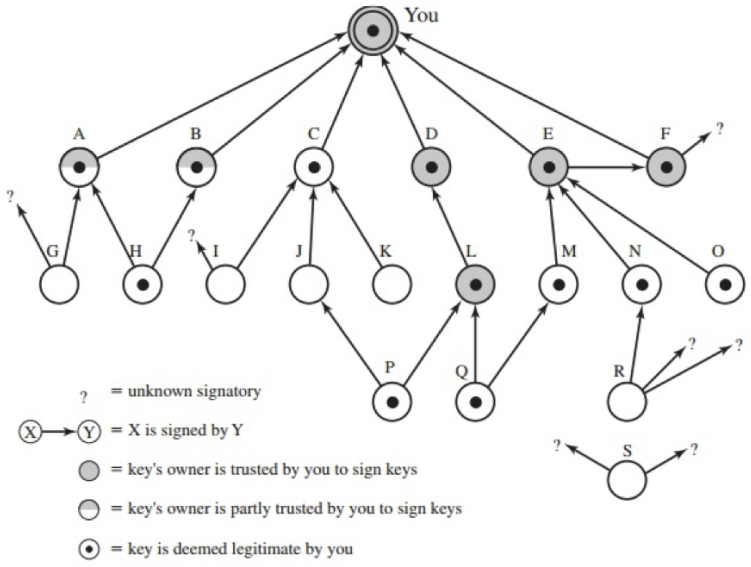


Figure 7.4: Web of trust

7.5 TOR

Figure out a way to set up internet communications so that an adversary snooping on the en route packet traffic would not be able to analyze the packet headers for the purpose of finding out who was talking to whom. Gleaning information regarding the original source of the packets and their ultimate destination is referred to as the traffic analysis attack even when protocols based on IPSec, TLS or VPN are used for establishing encrypted communication channels for the transfer of information between the web browsers and the web servers, the packet headers are always in clear text. Tor is a significant attempt to address the privacy problem over the Internet.

7.5.1 Onion routing

At each hop, the node "unwraps" a layer from the packet via symmetric keys, revealing the next destination. Clever combination of RSA and Diffie-Hellman (including its Elliptic Curve variant) cryptographic techniques.

The source of the data sends the onion to Router A, which removes a layer of encryption to learn only where to send it next and where it came from (though it does not know if the sender is the origin or just another node). Router A sends it to Router B, which decrypts another layer to learn its next destination. Router B sends it to Router C, which removes the final layer of encryption and transmits the original message to its destination 7.5.

7.5.2 TOR basics

Tor is based on the twin notions of Onion Proxies (OP) and Onion Routers (OR). A user's OP first queries a Tor directory for the IP addresses of the ORs in the Tor overlay. An overlay network is a telecommunications network that is built on top of another network and is supported by its infrastructure. An overlay network decouples network services from the underlying infrastructure

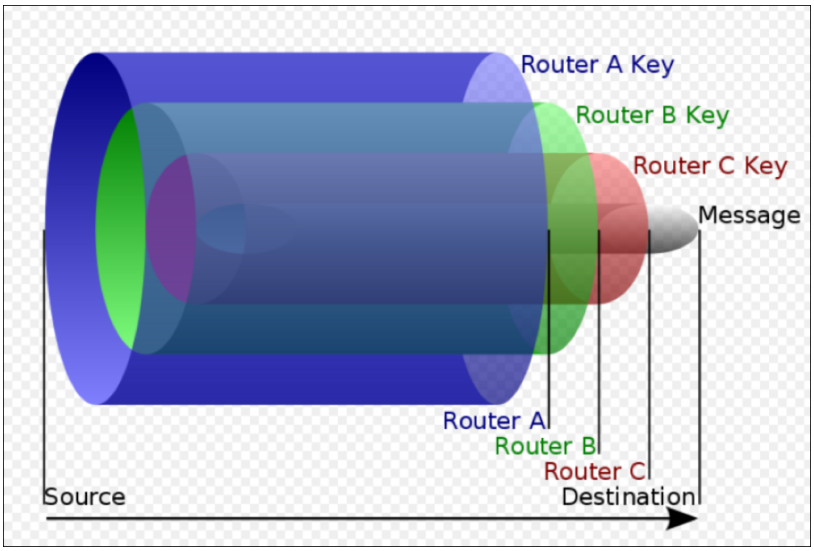


Figure 7.5: Tor routing

by encapsulating one packet inside of another packet. The user then selects a subset of these ORs (typically 3) for constructing a path to the destination resource. The specification "onion" in OP and OR is related to the layers of encryption placed on the Tor messages such that, except for the user's OP, the routing knowledge at any single node on a path through the Tor overlay is limited to exactly two nodes, the immediately preceding node on the path and the immediately following node. A user's OP constructs a path through the Tor overlay, that is called a circuit. The two parties at the two end of a circuit may use it for an arbitrary number of TCP streams. TCP first sets up a connection to the receiver then sends the data in segments which is carried by IP packets. This is called stream because it keeps the stream of data between to ends during transfer.

There are two types of packets: control tor packet and relay tor packets. The role of control tor packet is to alter the relationship between the sender node and the next node on the path that receives such a packet. The role of relay tor packet is to as paths are constructed (and torn down) incrementally by a user's OP the first link of the path can be constructed directly by the OP using a control tor packet, any extensions to the path are going to require that the commands for doing so be relayed to the currently last node on the path.

Initially, the control and the relay tor packets work together to create an end-to-end path (i.e. a circuit) in the Tor overlay in such a way that each interior node on the path has only local knowledge of the path. While the basic purpose of a relay tor packet is to carry the data that is exchanged between the two endpoints, that can only be done after a path is fully constructed. During the process of path construction, the data carried by relay tor packets is for the purpose of extending the path beyond the current termination point. Such relay tor packets generate control tor packets at the current terminal node on the path for extending the path.

7.5.3 Circuit creation

How does a user's OP use the control tor packets to create an end-to-end circuit incrementally, one hop at a time, in the Tor overlay?

We assume every OR node has a public RSA key that it makes available to the user's OP and

any communication sent to an OR that is encrypted with its RSA public key can only be understood by that OR. Next, Diffie-Hellman (DH) keys are created on the fly between the user's OP and each of the ORs on the path chosen by the user, the purpose of the DH keys is that when the user's OP wants to send a message to a designated OR on the path, it is encrypted with the session key derived from the OP's DH key and that OR's DH key. AES is used for the symmetric-key encryption with DH session keys.

The user OP (A) sends a create control tor packet to the first node (B) in the path chosen by the user. The packet contains: the CircID field (circuit identifier) set to a fresh value, the DATA field of this packet contains A's DH key Y(A→B) that is encrypted with B RSA public key. B responds back to A with the created control tor packet, B→A control tor packet. The packet contains in the DATA field B's DH Y(B→A). At this point, both A and B can calculate the secret session key K(AB) for their link.

All communications between any pair of nodes in the underlying network are over TLS: the public DH keys are not be visible to a packet sniffer, the RSA public/private keys used in the transmission of the control and relay tor packets are not to be confused with the RSA public/private keys used by TLS.

A and B can start exchanging relay tor packets that use the circuit identifier provided by A. To extend the circuit, A sends B a relay tor packet with the relay extend command. The packet contains in the DATA field a DH key Y(A→C) that is meant specifically for the new terminal node C on the path and also includes the identity of the new node. To guarantee that the key Y(A→C) is not seen by B, it is encrypted with C's RSA public key. The DATA field in the relay extend tor packet from A to B is encrypted with the session key K(AB). When B receives the relay extend tor packet from A, it knows that it is the current endpoint on the path and generates a control tor packet for C. B→C: control tor packet, the packet contains a DATA field with A DH key Y(A→C) that is meant specifically for node C and that is encrypted with C's RSA public key. This DATA field is encrypted with C RSA public key.

The control packet sent by B to C uses a new randomly generated number for the circID field that becomes the identifier for the segment of the circuit between the nodes B and C. There is no need for A to know this identifier. Only node B knows both circID identifiers (the one for the circuit between A and B and the one for the circuit between B and C). This fact plays an important role in ensuring that each node on the path has only the local knowledge of the path. Node C responds back to B with a created control tor packet. C→B: control tor packet. This packet contains a DATA field with C's DH key Y(C→A) meant for A. Node B sends this acknowledgment back to A using the relay extended tor packet. B→A: relay extended tor packet. This packet contains a DATA field with the key Y(C→A). Now both A and C can calculate the secret session key K(AC) for any messages that A may want to send to C through B that B is not allowed to see.

The path may be extended in the same manner to the node D by using a combination of control and relay tor packets. Notice that, in constructing the end-to-end circuit, there was never a need for using A's public RSA key. As a consequence, the user A remains anonymous to all the ORs in the circuit. But all the ORs in a circuit are known to the user A. This is somehow obvious since, remember, A has chosen all the nodes in the circuit 7.6.

To sum up:

1. The user OP A gets the public keys of the OR
2. The user OP A sends a control tor packet to node B. This control is encrypted with B RSA public key. This control tor packet contains the CircID and the DH public key (A→B)
3. B Receives this and sends back its DH public key (B→A). At this point A and B know K(AB) and can communicate and encrypt their communication using AES

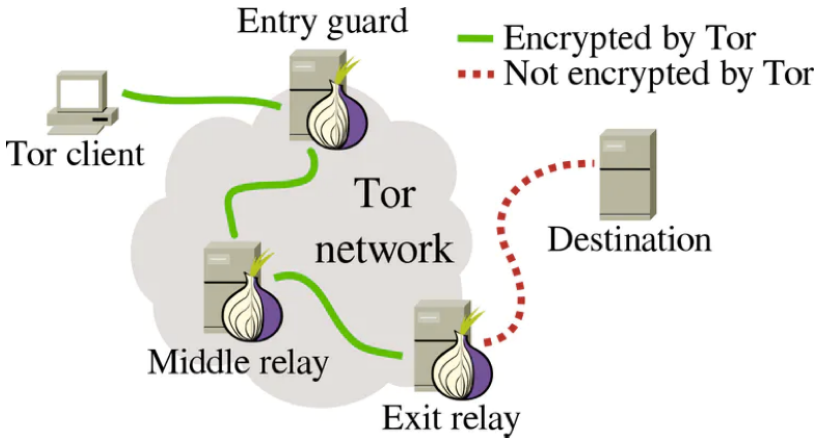


Figure 7.6: The circuit

4. To extends the circuit, user OP A sends an extend control packet to B that is encrypted with K(AB) and contains the public DH key (A→C), and is encrypted with C RSA public key.

5. B Receives this packet and creates a control tor packet from B to C with a brand new CircID. This packet contains the public DH key (A→C) encrypted with C RSA public key.

6. C receives this packet and answers with DH public key (C→A), that is sent to B and then forwarded to A. At this point A and C can start communicating with K(AC), and use it for any message that A don't want to be seen to B.

7. The path can be extended by repeating steps 4..6

7.5.4 Using the circuit

A can start pushing data into the circuit that is meant for the final destination E. Preliminarily, A sends a relay begin tor packet to B, from where it is forwarded to the next node on the circuit, and so on, thus creating an end-to-end stream between A and E. The user A is allowed to create an arbitrary number of streams sharing the same circuit. While the different TCP streams will have different streamID values in the relay tor packets that carry the stream data, they will have the same value for the circID field. Even though the value of this circID field will change from hop to hop in a circuit.

Assume the following path in the Tor overlay: A → B → C → D. The stream data that the user A places on the wire is encrypted with the K(AD) session key, followed by its encryption with K(AC) session key, followed by its encryption by K(AB) session key. As these stream data bearing relay data tor packets are received by B from A, the node B uses the session key K(AB) to decrypt the top layer of encryption and forward the stream to node C in the circuit. This process continues until the stream data reaches the final node D, from where it goes via the normal TCP transmission to the application running at the destination E.

Can the exit node operator see the source IP address, meaning the IP address of node A? In principle, the answer is no, i.e. it should not be possible for the exit node to know the source IP address. This is so because the Tor logic that keeps A's IP address shielded from the exit node D is the same as the logic that keeps B's IP address shielded from D. The packets that go out from D to

the web server at E should only bear D's IP address in the source fields. When D receives replies to those packets from the web server, it simply forwards them back to C. Unfortunately, there is an attack that allows the exit node operator to identify the IP address of the source.

Can the exit node operator see the data payload of the source packet? If node A is trying to reach an HTTPS web site, that implies the usage of encryption of the payload in the packets. In that case, the exit node operator obviously cannot see inside the packets that A is sending out.

Chapter 8

Cloud Encryption

8.1 Cloud computing

Cloud computing is defined as using remote, Internet-based servers for computational or storage resources. This has some pros and cons: first it is accessible from anywhere, any time and maintenance done by someone else (unless self-hosted!), it's cost-effective, as part of a shared data-center and it has scalable resources, on the other side availability relies on Internet connection and server reliability, long-distance communication creates a performance hit and your data is on someone else's machines.

Services are provided by third parties and not in house, users have no real guarantee that the cloud service providers are doing what they claim to be doing: trust becomes a key issue. The main security problem with cloud computing is that data loaded into the cloud is no longer in the control of the data creator/data controller and cloud service providers may be honest but curious.

When the data has to be retrieved, processed or searched, use of standard encryption is not useful: one has: either to allow the cloud to decrypt the data (this needs trust in cloud service providers) or to download the entire data set to the user's machine (this spoils the benefits of moving to the cloud in the first place). Even if processing is not required, one needs some additional cryptographic functionality to be able to search and retrieve data which has been stored (encrypted) in the cloud. One of the advantages of using the cloud is the possibility to use well-engineered and managed security services, it is typically still the responsibility of customers to configure such services appropriately (split/shared responsibility model).

8.1.1 Fully homomorphic encryption (FHE)

An arithmetic circuit can be applied to a ciphertext and the result is the encryption of the output of the arithmetic circuit as if it had been evaluated on the underlying plaintext. Data owners encrypt their data $\mathcal{D}$, to obtain $enc(\mathcal{D})$ and send it to the cloud provider. Then suppose the data owners wanted to perform some operation f on the data to obtain $f(\mathcal{D})$ (for example search). The cloud can compute $enc(f(\mathcal{D}))$ which is then returned to the data owners, the data owners then decrypt this ciphertext to obtain $f(\mathcal{D})$. The main problem is efficiency and scalability. Each operation on the ciphertext results in compounding noise. But it results in loss of homomorphic property after a certain number of operations. Also the combination of the noise production followed by the noise reduction makes the scheme completely impractical: in order to perform one search on Google using this encryption, the amount of computations needed would increase by a trillion.

8.1.2 Somewhat homomorphic encryption

The scheme can do both addition and multiplication to the original plaintexts but its capability is heavily limited. For example, it is possible to perform unlimited additions but only one level of multiplication to the ciphertext.

8.1.3 Partial homomorphic encryption

Partial homomorphic encryption: Homomorphism property only with respect to some operations and not others. For example if homomorphic with respect to addition (multiplication), it is not possible to compute multiplications (additions, respectively) over ciphertexts.

8.1.4 Multi-party computation (MPC)

It allows the equivalent of FHE with the use of multiple servers as opposed to a single one. Thus data can be securely outsourced to multiple cloud providers who then can compute on this data, such that one can tolerate a set of colluding adversaries (up to some bound on the number).

Data owners split their data $\mathcal{D}$ into chunks $\mathcal{D}_1, \ldots, \mathcal{D}_n$ via a secret sharing scheme. The shares are then distributed to n distinct cloud providers. The cloud providers are now able to compute any function $f(\mathcal{D})$, but they obtain partial results (called shares) that are then returned to the data owners for combining into the final solution $f(\mathcal{D})$ (computing shares may require some exchange of information among providers). As long as the data owners trust $n - t$ out of the n providers, their data is still kept confidentially where t is the maximum number of adversaries which can be tolerated.

8.1.5 Searchable encryption

Solve the following problem: a user outsources a database to a server and then wants to query the server to obtain data. Question: what does security mean in this context? The database contents should remain private to the client and in some cases also the access patterns should be private. Two variants: public or symmetric key. In both variants, the encrypted database is augmented with a set of tokens. Each token is associated with a keyword. The database is encrypted with any encryption algorithm, the searchable encryption scheme is solely used to construct and search for the tokens. Public key variant is vulnerable to selected keywords attack and thus is less adopted. Symmetric key variant is combined with Oblivious RAM techniques to guarantee also the privacy of access patterns. A server, which maintains a data storage system, can gain information about its users' habits and interests, and violate their privacy, even without being able to decrypt the data that they store. The server can monitor the queries made by the clients and perform different traffic analysis tasks. It can learn the usual pattern of accessing the encrypted data, and try to relate it to other information it might have about the clients. For example, if a sequence of queries q_1, q_2, q_3 is always followed by a stock-exchange action, a curious server can learn about the content of these queries, even though they are encrypted, and predict the user action when the same (or similar) sequence of queries appears again. Moreover, it is possible to analyze the importance of different areas in the database, e.g., by counting the frequency of the client accessing the same data items. If the server is an adversary with significant but limited power, it can concentrate its resources in trying to decrypt only data items which are often accessed by the target-user.

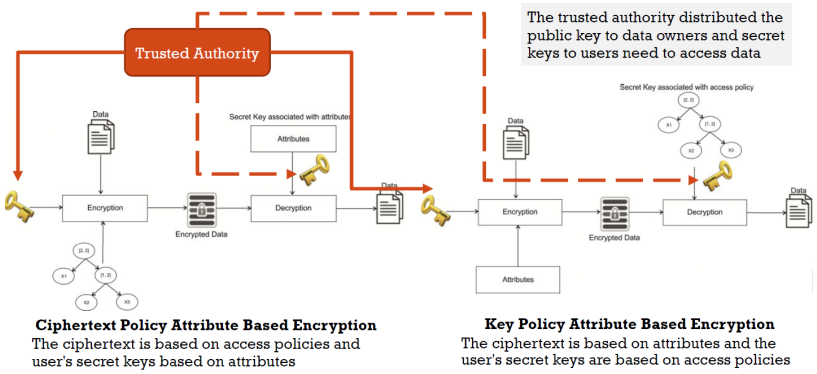


Figure 8.1

8.1.6 Order preserving encryption

It solves a variant of the searchable encryption problem in another way. It enables ciphertexts to be ordered in a way which respects the ordering of the underlying plaintexts. Thus binary search can be conducted on the ciphertexts. The key problem is that the security properties are very weak.

8.1.7 Attribute based encryption

It extends Identity Based Encryption where decryption of data is allowed on the basis of whether the decryptor satisfies a set of policies associated with given secret keys, which in turn are associated to attributes. Complex access control policies for encrypted outsourced data can be embedded within the ciphertext itself. This means that a data owner does not need to trust the cloud provider to implement a policy they require 8.1.

Identity-based encryption is a type of public-key encryption in which a user can generate a public key from a known unique identifier such as an email address and a trusted third-party server calculates the corresponding private key from the public key. In this way, there is no need to distribute public keys ahead of exchanging encrypted data. The sender can simply use the unique identifier of the receiver to generate a public key and encrypt the data. The receiver can generate the corresponding private key with the help of the trusted third-party server.

8.1.8 Delegated computation

A resource constrained client can delegate a computation to a powerful service provider. The client can check whether the provider has performed the correct computation. Two running modes, in the non-private mode cloud server learns the clients data, in the private mode the input data, and perhaps the function, are kept private. Theoretical results that combine techniques from FHE, ABE, and MPC.

8.1.9 Maturity levels

Maturity levels in picture 8.2.

Technology	Ready for Deployment	Short Term Research Needed	Longer Term Research Needed
Fully Homomorphic Encryption	×	×	√
Multi-Party Computation	√	√	×
Searchable Encryption	√	√	×
Order Preserving Encryption	×	×	√
Attribute Based Encryption	×	√	×
Delegated Computation	×	×	√

Figure 8.2: Maturity levels

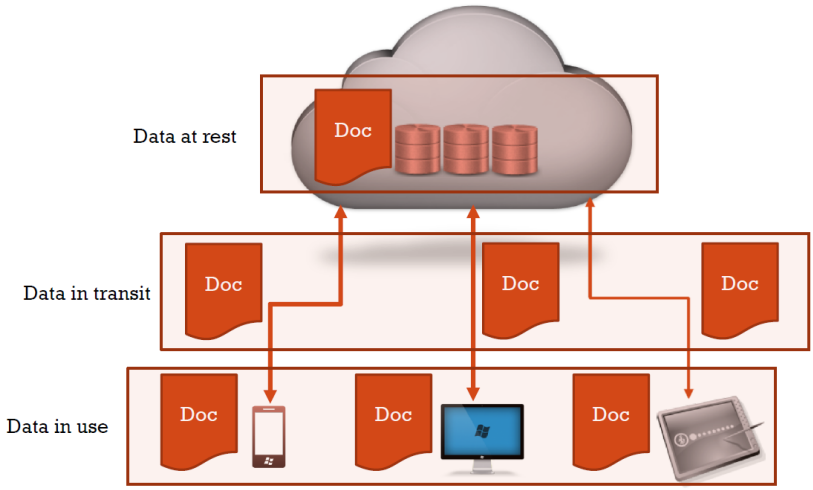


Figure 8.3: Cloud storage

8.2 Cloud storage

Essentially your data/files are stored on a remote server and we have conflicting requirements: need to keep data confidential with respect to the curious but honest cloud service provider and support controlled sharing of data, i.e. employees should be able to selectively access data stored in files to be able to perform their jobs. Use cryptography to enforce access control policies so that sharing among employees is permitted while protecting against curious service providers. Role based access control uses role hierarchies to make the specification of policies more compact. These can always be compiled away without loss of generality. We have two tables: UA: user-role assignment, PA: permission-role assignment. User u has permission p iff there exists role r such that (u, r) in UA and (r, p) in PA 8.3.

8.2.1 Hybrid cryptography

Each user u is equipped with a pair of private and public key: $(ks(u), kp(u))$. Each role r is equipped with a pair of private and public key: $(ks(r), kp(r))$. Each file f is encrypted with a unique symmetric key: $k(f)$. If (u, r) in UA, then compute $\{ks(r)\}kp(u)$ i.e. encrypt the secret key $ks(r)$ associated to r with the public key $kp(u)$ associated to u. In this way, u will be able to retrieve the secret key associated to the role by using its private key. If $(r, (read, f))$ in PA, then compute $\{k(f)\}kp(r)$ i.e. encrypt the symmetric key $k(f)$ associated to the file f with the public key of the role r. In this way, a user u with read permission on f (i.e. such that (u, r) in UA and $(r, (read, f))$ in PA) will be

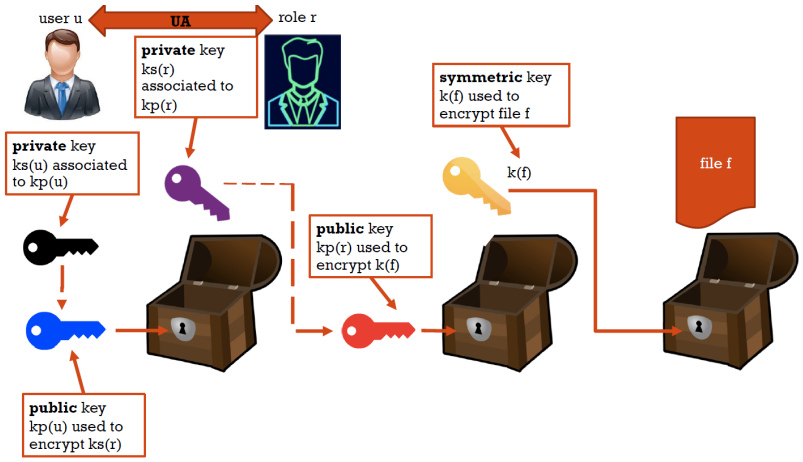


Figure 8.4: Hybrid Cryptography

able to retrieve the symmetric key associated to f by first retrieving the secret key $ks(r)$ associated to the role which can then be used to retrieve the symmetric key of the file f which is encrypted with the public key $kp(r)$ associated to r 8.4.

Notice that further auxiliary data (e.g., files version numbers and digital signatures), together referred to as metadata, are needed to enforce policies. Users store their private keys in secure personal devices (e.g., laptops provided with an antivirus) access to which is protected through passwords or similar authentication techniques. Both encrypted data and metadata are stored in the cloud or in a secure area within the organization. To write on a file f, a user performs the same operations to obtain the symmetric key $k(f)$ which is used to encrypt the new file. An entity (usually called Reference Monitor) checks whether the user has actually write permission before accepting the new file and storing it in the cloud.

8.3 Database-as-a-service (DBaas)

Not explained [1].

8.4 Order preserving encryption (OPE)

Not explained [2].

8.5 Homomorphic preserving encryption

Not explained [3].

[1] TODO: Slide 27..31 12-Cloud_Encryption-2p.pdf
[2] TODO: Slide 37..42 12-Cloud_Encryption-2p.pdf
[3] TODO: Slide 43..49 12-Cloud_Encryption-2p.pdf

8.6 Partially homomorphic

Homomorphic with respect to multiplication: RSA (1978). Fix a large number $n = p * q$ where p and q are large primes. Find pair of numbers (e, d) such that $e * d \equiv 1 \bmod \phi(n)$. Set (e, n) to be the public key and (d, n) the private key $Enc(m) = m^e \bmod n$. Multiplicative Homomorphism property:

$$Enc(m_1) * Enc(m_2) = m_1^e * m_2^e \bmod N = (m_1 * m_2)^e \bmod N$$

A scheme is based on the notion of pairing. The advantage of pairing is that it allows us to check if the value of r is equal to the product of a and b. This can be done as follows: given g^r, g^a, g^b checking if $e(g^a, g^b) = e(g^r, g^1)$ is equivalent to $g^{a*b} = g^r$.

It is possible to combine additive homomorphic scheme with pairing in such a way to compute both the addition and multiplication of the exponents in the group. However, only a limited number of multiplications can be performed as the group on which pairing is applied may generate a group that does not satisfy the conditions for both the additive homomorphic scheme and the pairing.

By simply multiplying two ciphertexts together, one obtains the encryption of the product of the original plaintexts. Fully homomorphic means we can accommodate any function, but RSA is only homomorphic to multiplication. Pallier is homomorphic to addition but not multiplication.

8.7 Full homomorphic encryption

It centers around a function which introduces a certain level of noise into the encryption. Each operation on the ciphertext results in compounding noise. The noise is resolved with the bootstrapability of the encryption.

8.8 Multi party computation for homomorphic encryption

Not explained [4].

[4] TODO: Slide 58..60 12-Cloud_Encryption-2p.pdf

Chapter 9

Post Quantum Cryptography

Quantum computing is the use of quantum phenomena such as superposition and entanglement to perform computation. [...] Quantum computers are believed to be able to solve certain computational problems, such as integer factorization (which underlies RSA encryption), substantially faster than classical computers. Superposition refers to a combination of states we would ordinarily describe independently. To make a classical analogy, if you play two musical notes at once, what you will hear is a superposition of the two notes. Entanglement is a famously counter-intuitive quantum phenomenon describing behaviour we never see in the classical world. Entangled particles behave together as a system in ways that cannot be explained using classical logic. Quantum computing is based on the fact that tiny particles, such as electrons, can simultaneously take on states that we would normally deem mutually exclusive. We never see this superposition of different states in ordinary life because it somehow disappears once a system is observed: when you measure the location of an electron, all but one of the possible alternatives are eliminated and you will see just one. There is more to quantum physics than just superposition. If you look at a system of more than one qubit, then the individual components are not generally independent of each other; instead, they can be entangled. When you measure one of the qubits in an entangled system of two qubits, for example, then the outcome — whether you see a 0 or a 1 - immediately tells you what you will see when you measure the other qubit. Particles can be entangled even if they are separated in space, a fact that caused Einstein to call entanglement "spooky action at a distance".

A quantum computer works with particles, representing quantum bits (qubits), that can be in superposition. I.e. qubits can take on the value 0, or 1, or both simultaneously. Entanglement implies that a quantum computer cannot be described using classical information theory since its states are not simply the result of stringing together the descriptions of the individual qubits but you need to describe all the correlations between the different qubits. As you increase the number of qubits, the number of those correlations grows exponentially: for n qubits there are 2n correlations. While a quantum algorithm can take entangled qubits in superposition as input, the output will also usually be a quantum state —and such a state will generally change as soon as you try to observe it. The art of quantum computing is to find ways of gaining as much information as possible from the unobservable.

Noise is the central obstacle to building large-scale quantum computers. Quantum decoherence happens when qubits lose information to the environment over time. In other words, there is a certain threshold for noise (called fault tolerance) where quantum computers will theoretically be reliable enough to be considered useful.

The more qubits the more powerful it becomes, but also there is more noise. The idea is to use

machine learning algorithms that are capable of correcting errors by learning.

Qubits carry huge amount of information until they are observed. Unfortunately, to learn the result of computation one has to measure and this collapses qubits to basis state, i.e. 1 qubit leads 1 classical bit of information. The game is to increase the number of qubits while allowing for extracting useful information from quantum states.

Quantum supremacy is the goal of demonstrating that a programmable quantum device can solve a problem that no classical computer can solve in any feasible amount of time (irrespective of the usefulness of the problem).

9.1 Relevance to cryptography

Shor Algorithm: polynomial-time quantum computer algorithm for integer factorization. Problem: Given an integer N, find its prime factors. Complexity: polynomial in $log(N)$. Exponentially faster than the most efficient known classical factoring algorithm, the general number field sieve method. If a quantum computer with a sufficient number of qubits could operate without succumbing to quantum noise and other quantum-decoherence phenomena, then Shor's algorithm could be used to break public-key cryptography schemes, such as the widely used RSA scheme. Shor's algorithm shows that factoring integers is efficient on an ideal quantum computer, so it may be feasible to defeat RSA by constructing a large quantum computer.

Grover algorithm is a quantum computer algorithm that finds with high probability the unique input to a black box function that produces a particular output value, using just $O(\sqrt{N})$ evaluations of the function where N is the size of the function domain. Unlike Shor algorithms, which provides an exponential speedup, Grover's algorithm provides only a quadratic speedup. Grover's algorithm could brute-force a 128-bit symmetric cryptographic key in roughly 2^{64} iterations, or a 256-bit key in roughly 2^{128} iterations. It is sometimes suggested that symmetric key lengths be doubled to protect against future quantum attacks.

9.1.1 Risk assessment

Let k be the number of years for which the keys need to be secure. Let i be the number of years to harden cryptographic infrastructure to be secure against quantum computer attacks. Let q be the number of years to build a large scale quantum computer. If $q < k + i$ then quantum computing is relevant.

Moore's law states that power doubles every two years (exponential increase). Neven's law: quantum computers are gaining computational power on classical ones at a doubly exponential rate.

The root cause for Neven's law is that if a quantum circuit has four quantum bits, it takes a classical circuit with 16 ordinary bits to achieve equivalent computational power.

9.1.2 Post Quantum Cryptography

The idea is to design cryptosystems based on mathematical problems that are not solvable by Shor algorithm. Since 2015, NIST is actively researching new, quantum-resistant algorithms to improve and extend its official standards which are binding for all federal entities, including Digital signatures and Key exchange protocols.

The NIST requires that public-key encryption shall include algorithms for key generation, encryption, and decryption, key encapsulation mechanism (KEM) shall include algorithms for key generation, encapsulation, and decapsulation, digital signature: shall include algorithms for key generation, signature generation and signature verification.