Computer Security

1 Introduction

Any digital asset (hardware system, piece of software, network or a combination of these) is vulnerable to attacks that try to take advantage of the system or to make it unusable. This is done for the gain of the attacker at the expense of the asset owner.

Let's first start by distinguishing between the concepts of threat, exploit and attack. A threat is any potential violation of security that could harm an asset. The threat could be anything ranging from a person, an insecure service, all the way to some unacknowledged piece of information, service or system. An exploit is any software or tool that is built to take advantage of a vulnerability. An attack is any intentional or unintentional event that harms, or tries to harm, an asset. Known vulnerabilities are published in documents called CVE (Common Vulnerability and Exposure).

Attacks and threats can be further categorized based on the type of target. An *untargeted threat* tries to access and target any compatible machine it may find on the network; this type of threat includes worms and internet-distributed malware. A *targeted attack* is a deliberate attack on a single, specific machine.

Many cyberattacks are carried out through *bot networks*. A bot network is a group of hacked machines, each of which may belong to a different unknowing individual owner. The entire bot network is controlled by a single

entity (a hacker or a group of hackers) and used for malevolent purposes. The most common use is in DDoS (Distributed Denial of Service) attacks. A DDoS attack tries to shut down a web service by saturating it with a constant burst of requests. Bot networks are mostly located in the United States, where static IPv4 addresses are more commonly given to domestic internet connections compared to other parts of the world. To mitigate this issue, ISPs try to identify both the single machines (to cut off their connection and notify their legitimate owners) and the control systems (to shut them down by notify the authorities).

An APT (Advanced Persistent Threat) is a persistent, patient and well-prepared enemy, constantly looking for a way to harm a specific asset. The presence of an APT can cause a high emotional impact on the asset owner. APTs try to infiltrate the system and wait, while gathering information as discretely as possible. The final attack is unleashed only when the attacker is fully ready.

It is hard to build a system that is both secure and can handle complexity in a simple way. These two dimensions are subject to a trade-off. As the system grows in complexity, the amount of components that cannot be fully trusted increases. Along with it, the level of redundancy needed to maintain it reliable also has to increase. Furthermore, making a system hard to access for attackers inevitably makes it harder to access for its legitimate users too. For this reason, a perfectly secure system is also a perfectly unusable one.

Let's now define the concept of risk. A risk is the product of two factors: the probability of a threat taking advantage of the system and the size of the potential damage caused by the attack. The size of the potential damage further depends on two factors. The first factor is the value of the asset, which may be asymmetrical or symmetrical. For instance, user data is highly valuable to the legitimate owner but of little use for the attacker (unless it is put under a ransom). On the other hand, an attack on a bank is symmetrical. Money is of equal use and value to both the legitimate owner and the thief. The second factor to consider is the exposed surface. Insecure applications installed on the system, conflicting security policies and a poor level of configuration enlarge the exposed surface. Being surrounded by a myriad of different internet-connected devices all day also increases the exposed surface. To solve this, a strong level of authentication is required on both client and server side. Authorization and access control systems need to be highly reliable. The detection of suspicious activity, known as abuse control, needs to be quick and effective. An example of abuse control is IDS (Intrusion Detection System), a log-based technique that can employ stanrdard or machine learning algorithms to detect malicious patterns in the system activity. All protocols and applications, all the way down to the operating system, need to have a secure design and a bug-free implementation. Security policies have to be perfect. Users would also need to be "perfect", in the sense of being trained not to fall for simple, social engineering attacks.

In the real world, effective security protections are often not deployed at all. Even when deployed, in many cases they are not patched in a timely manner. Web servers do not, in many cases, properly monitor and restrict access to their internal hosts or resources. Even when defined, security policies are often not completely implemented. Organizations do not allocate enough funds and manpower to security tasks. Users are uneducated about matters of security. To restate, perfect security cannot exist. All systems have a trade-off between functionality, usability and security. Rather than reducing attacks, it is more advisable in most cases to simply reduce their success rate, while limiting the scope of their potential damage. An excess of security reduces the usability and performance of the system, thus causing potential damages that are larger than the risk of an attack. The lack of security has a cost, but so does security itself. This introduces the concept of minimum viable security, which is the intersection between what customers will buy and what the security team demands.

As a final note, the security of computer systems is not completely determined within the cybersecurity domain. Non-technological security also matters. For example, a data center with no security guards, no surveillance and poor locks may get its data physically stolen by having thieves remove disks from the racks. Logical security, in the form of access control and restricted permission, mitigates the risk of having an attacker within the ranks of the organization. This also interlocks with organizational security, that is, properly assigning critical roles to the right personnel.

A *honeypot* is a decoy resource, put in systems to detect attackers. It serves no purpose to the legitimate owners, other than to be a bait for the attackers. Whoever accesses it can be immediately identified as an attacker.

2 Classical encryption: history and principles

In the antiquity, many techniques were used to *hide* secret messages. These methods, collectively known as *steganography*, suffered from a common issue: the message, when found, could be immediately read by the enemy. This lead to the creation of *cryptography*, the collective name given to techniques that can turn a readable message, known as *plaintext*, to a *ciphertext* that is unreadable to everyone except to those who know the *secret*. The secret may be the algorithm, a code, a key or a combination of these. A single cryptographic algorithm is known as a *cipher*. Traditional ciphers are text-based. Modern ciphers are digital, to cover all kinds of information.

2.1 Replacement ciphers

The simplest form of encryption is called *replacement cypher* or *monoalphabetic substitution*. It's a fixed cipher in which each letter of the alphabet is substituted with another, with a univocal correspondence. In an even simpler version known as *shift cipher*, in which each letter is substituted with the letter that comes a fixed number of positions after it in the alphabet (three for Caesar's cipher, its most famous historical example). A replacement cipher is hard to break by brute force, because the possible combinations of letter couples are many. However, letters only change their individual identity and not their groupings. By analyzing the relative frequency of letters and comparing it with the distribution of letters in the target language, the replaced letters can easily be matched with the original plaintext letters.

A simple approach to mitigate the issue is the use of *nulls*. This approach uses the least frequent symbols in positions that do not alter the meaning. For example, "X" may be used to indicate a space. The symbol itself can be called through an escape sequence, like repeating it twice. Another approach uses *homophones*. A homophone is a set of several symbols that substitute a single frequently used character. The goal is to make the symbol distribution less recognizable, to make relative frequency matching harder. Another approach uses *code words*, which consist in whole word substitutions. The code needs to be diffused along with the message, making it prone to falling into the attacker's hand. This approach does not offer a significant increase of protection over simple monoalphabetic substitution.

2.2 More complex approaches

Attempts to go beyond simple monoalphabetic substitution may take two main routes. *Multiple encrypting alphabets* alternate cyclically between different encryptions depending on the letter position within the message. The two main examples are the Leon Battista Alberti cipher disk and the Vigenère cipher. The latter, in particular, was the first to use some basic principles that are still employed in modern cryptography. The second approach treats multiple letters as a unit. Examples include the Porta and the Playfair ciphers.

The *Alberti cipher disk* uses a rotating disk with letters both inside and outside the disk. Rotating the inner section changes the letter pairings. Typically, the disk is alternated between two positions for even and odd letters in the message.

The *Vigenère cipher* is based on a keyword. First, the text is stripped of whitespaces. If the key has t letters, the text will be divided in blocks of t letters. Each letter from each block is treated like a number (its ordinal in the alphabet) and summed to the corresponding letter in the key by their positions in the alphabet. The alphabet is treated as cyclical for sums that exceed the number of letters in it. This substitution can be pre-computed into a 26×26 table in which rows represent key letters and columns represent plaintext letters. The Vigenère cipher was considered unbreakable for a long time. This is because the possible keys for a key size of t are 26^t. It is resistant to frequency analysis. Babbage and Kasiski realized that the length of the key could

be deduced by analyzing repeating letters, which later lead to a technique to break the cipher. By measuring the distance between repeating letters, an estimate of the key length could be computed through the $\gcd$ of such distances. Then, the text could be cut into blocks. Finally, frequency analysis could be performed on the single positions of each block, treating each as a different monoalphabetic substitution.

The *Porta cipher* creates a 26×26 letter matrix that assigns each pair of letters to a symbol in the corresponding position in the matrix. By changing the order of the letters on the side of the matrix, the key can be changed.

The *cipher grids*, invented by Girolamo Cardano, are even-sized grids with holes in certain positions. The message is written and read on a square with the same size as the grid, through the grid itself, by rotating it in its four possible orientations.

The *Playfair cipher* consists in a 5×5 matrix. The key is at the beginning, and the rest of the alphabet follows it, in order. The cipher considers letters in pairs called *digrams*. If a digram contains the same letter twice, an "X" is added in between. Each digram marks the vertices of the diagonal of a rectangle within the matrix. The digram then gets substituted by the letters on that diagonal. The Playfair cipher is better than monoalphabetic substitution, yet relatively easy to break by using digram frequency analysis.

2.3 Perfect ciphers

A perfect cipher bears no statistical relation between the unencrypted and encrypted versions of a message. In other words, plaintext and ciphertext have no correlation. The technique that achieves this perfectly is called *one-time pad*. The message is converted in binary form. A single-use key, with the same length as the message, is created by selecting random independent bits. The message and the key are then combined through a XOR operation. Since the XOR operator is the inverse of itself, the same procedure is applied to retrieve the plaintext from the ciphertext. The one time pad, because of its single-use nature, is perfect but of little use.

2.4 Cipher disks and rotors

The Alberti cipher was the state of the art until the First World War. By the end of it, the first electromechanical rotors were built. The idea is simple. An electromechanical rotor pairs letter by connecting pins on two coaxial metal disks with wires. A single unit works like an Alberti cipher, but many rotors could be stacked, and their positioning could be changed at each letter, to produce an electromechanical device that could automatically encrypt and decrypt messages. Speed and security are both increased significantly compared to a manually encrypted, single-disk Alberti approach. For instance, n connected rotors produce 26^n possible permutations.

The Enigma machine used by Germany in WW2 was based on three electromechanical rotors. Their configuration was changed every day. The rotors were connected to a *reflector*, that is, a disk that sends the signal back through the rotors. This makes the cipher symmetrical, with both input and output being passed through the same side. A keyboard was used for input, while the output was displayed through a set of lights representing letters. The first rotor rotated by one position with each input letter. The second rotor rotated by one position for each full rotation of the first rotor. The third rotor rotated by one position for each full rotation of the second rotor. This produced $26 \times 25 \times 16 = 16900$ possible permutations. To further increase the number of permutations, 6 couples of letters were manually swapped with each other by moving wires on a connector board.

Every day a new key was sent, written twice. The system had a few weaknesses. First of all, no letter encrypts itself. Then, no letter is encrypted by contiguous letters in the alphabet. The cipher was symmetrical and the keys were short. Furthermore, some words were in a fixed position in each message. For example, the daily weather report by the Army always started with the word `WETTER` ("weather" in German). This gave the Allies an advantage in code breaking. The team lead by Alan Turing was able to build two "decoding bombs", brute

force code breaking machines based on Enigma devices stolen from the enemy. The first, Victory, took around two weeks to break the code, which made it unusable for keeping up to date with daily communications. The second, Agnus Dei, was able to break daily codes in around two hours, contributing significantly to the war effort. The version of Enigma used by the Kriegsmarine employed an 8-rotor system with a rotating reflector. The messages also had no stereotypical formulas, which made code breaking much harder.

3 Symmetric-key encryption

A *symmetrical cipher* is a type of cipher in which both writer and reader use the same key. In other words, the same key is used for both encryption and decryption.

First of all, let's lay out what is known as Kerckhoff's principle.

> *Kerckhoff's Principle*: the security of a cryptographic system must depend solely on the security of the shared secret, and not on the secrecy of the encryption system itself.

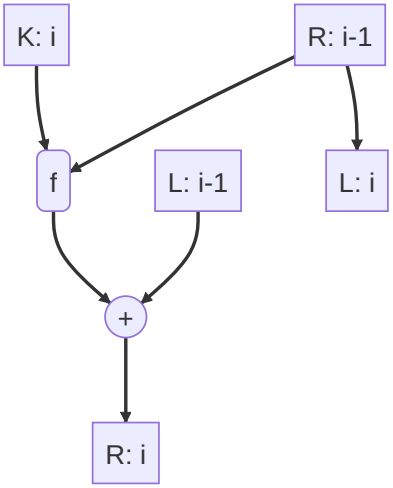
In other words, a cryptographic algorithm is considered good if the decryption of its results depends only on the key. The knowledge of the encrypted message and of the cryptographic algorithm should not facilitate the attacker in their message decryption efforts. Despite this, attacks are still possible. Most types of attacks can be grouped into four categories. The first is called *known plaintext attack*. In this case, the unencrypted message is captured by the attacker. In a similar case, called *known ciphertext attack*, the attacker has access to the encrypted form of messages. The third type of attack is called *chosen plaintext*. In this case, the attacker can choose which message will be encrypted, either by manipulating the victim or by getting access to the encryption machine. The last type of attack, *known ciphertext*, consists in the possibility for the attacker to get any ciphertext decrypted, while still not knowing the key. For all four attack types, the intended goal is to obtain the key.

The Vigenère cipher is easily broken by deploying a known ciphertext attack. First, the attacker needs to find repeating letter patterns within the ciphertext. The length of the key is a multiple of the distance between those patterns. For this, the attacker can rely on a metric called *concidence index*. It is defined as the probability of two randomly picked characters within a string being the same. This changes depending on the target language. The coincidence index can be used to find the length of the key. After finding that length, the attacker can rely on the *mutual coincidence index*. This metric represents the chance of two randomly chosen characters, the first picked from a string and the second from another, being equal. It can be used to find the correct key given its length. This process was described in more detail on the section about the Vigenère cipher.

Ciphers can be split into two families. The family of *stream ciphers* encrypts the plaintext character by character. A notable example is the monoalphabetic substitution. On the other hand, *block ciphers* split the plaintext into equally sized blocks which are then encrypted as a whole, one at a time. The Vigenère cipher was the first example of a block cipher. Most modern block ciphers rely on the model of the *Feistel block encryption*, created in 1973. It is based on permutations and substitution within blocks. The Feistel model is derived from Shannon's work on information theory from 1949. The main two principles are *diffusion* and *confusion*. The principle of diffusion consists in having multiple plaintext characters influence any single ciphertext character. Small changes in the plaintext or key will thus result in widespread and unpredictable changes in the ciphertext. This ensures that statistical patterns in the input are dispersed across the output, making the cipher more resistant to cryptanalysis. The result is confusion: the statistical relationships between plaintext and ciphertext are unclear. Many important algorithms, like DES and Blowfish, are based on Feistel's model.

Larger blocks increase security at the expense of speed. The same consideration applies to key size. The Feistel encryption process is iterative. Each iteration is called *round*. Each round has the same structure but employs a different subkey. All subkeys are derived from the full key in an operation known as *key scheduling*.

At the start of each round, the block is divided in half. The *round function* (permutation and substitution based on the round subkey) is only performed on the right half of the block, as shown in the graph below. The left half is XORed in later. This becomes the new right half. The original right half becomes the left half for the next iteration. The asymmetry of the process is a potential source of weakness, because it may leave one half of the block more vulnerable than the other. Since the halves are swapped at each iteration, increasing the number of rounds will mitigate the issue and increase security.



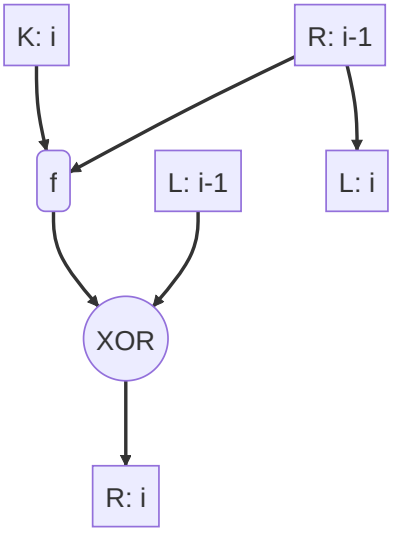
In all cases, increasing block size and key length enhances security at the expense of speed. For instance, the default number of rounds for AES-256 is 14 rounds. AES has never been broken for high round numbers. However, even with hardware support for encryption that enables the execution of an entire round within a single clock cycle, the procedure still needs to be repeated for a number of cycles at least equal to the number of rounds. This is not a problem for most modern consumer devices, but may be a problem for other domains.

3.1 DES (Data Encryption Standard)

DES is a block cipher that uses 64-bit blocks and a 56-bit key. It is based on a multi-round Feistel model. It was required by NBS (which then became NIST) in 1973 and published in 1977.

The key starts from 56 bits and is then extended to 64 bits by adding a parity bit at the end of each byte. The parity bit is defined as the XOR of the previous 7 bits. The plaintext is passed through a four step round function:

- an IP permutation (based on a 8×8 permutation matrix)
- 16 iterations of key scheduling on 48-bit blocks
- an exchange
- a reverse IP^{-1} permutation
  The following scheme represents a single iteration:



The function f used in each iteration works in four steps. First, the 32-bit input (the right half of the block) is *expanded* to 48 bits by repeating twice half of the bits. This is achieved through an *expansion table* defining which bits to copy and where to put them. The expanded 48-bit block is then XORed with a 48-bit subkey obtained by *key scheduling* (more on that later). The XOR output is divided in 8 blocks that are 6 bit long. Each of these 6-bit blocks is passed through a *S-Box*. The output of the S-box is reunited and de-expanded, producing a 32-bit block. Following the Feistel scheme, the output is XORed with the left block, then swapped with the right block. This final swap is not performed on the 16th round.

The subkeys are the output of the *key scheduling* step. The 64-bit key is passed through a PC-1 permutation. This permutation is table-based, similarly to the IP and IP^{-1} permutations used in the first and last step. The parity bits (positions 8, 16, 24, 32, 40, 48 and 64 within the key) are not considered. The remaining 56 bits are shifted left of 1 or 2 bits depending on the iteration. The total shift for the usual 16 iterations is of 28 bits. The left shift results are passed through a PC-2 permutation which produces the 48-bit subkey for the current iteration. PC-2 is also table-based and suppresses the bits in positions 9, 18, 22, 25, 35, 38, 43 and 54.

The S-Box was introduced by NBD in 1976. S-Box stands for Substitution Box. It is a 4×16 substitution matrix based on a non-linear function, in which each cell contains 4 bits. Each row is a permutations of integers from 0 to 15. Multiple S-Boxes are defined. The S-Box enjoys a few important properties. First of all, no S-Box is a linear function of its inputs. This means that changing one bit in the input alters at least two bits in the output. The 6 input bits are used to find positions in the box. The first and last bit specify the row, while the center bits specify the column. The 4-bit output is the content of the cell XORed with `001100`. This guarantees that any output will differ for at least two bits with any other output. S-boxes are defined to be statistically balanced. The number of zeros in the output block is on average the same as the number of ones.

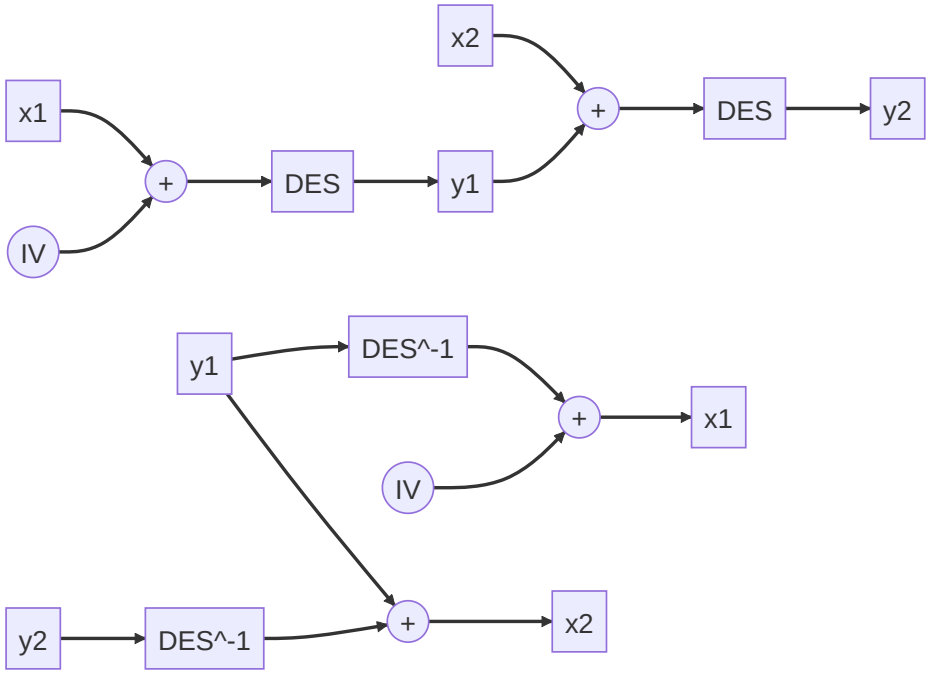
DES decryption uses the same process as encryption, but reverses the order of the generated subkeys. This makes the algorithm symmetrical.

3.1.1 Encryption modes

Encryption modes are different possible approaches to handle the problem of encrypting messages that are longer than one block. There are many possible ways to chain blocks, but few are safe. The main tried-and-tested approaches that gained widespread community approval are only a handful. The first five approaches come from the original DES Modes of Operation standard from 1977. They are: Electronic Codebook (ECB), Cipher Block Chanining (CBC), Cipher FeedBack (CFB), Output FeedBack (OFB) and Counter (CTR). Two more modes, called Galois Counter Mode (GCM) and Xor Encrypts Xor (XEX), were added later. GCM ended up being the most used approach in both DES and its successor AES. Let's now analyze each mode individually.

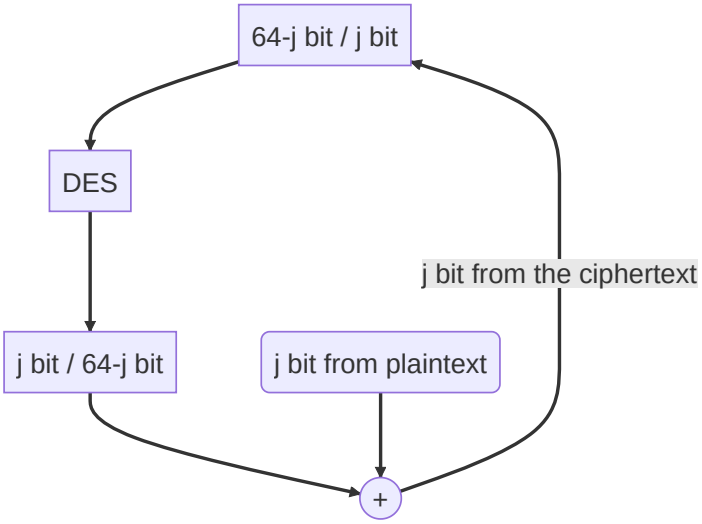
The Electronic CodeBook (ECB) is the simplest possible approach. First, the plaintext is cut into 64-bit blocks. If its length is not a multiple of 64, the last block is padded with 100.000. Each block is then encrypted independently using the same key. Encrypted blocks hence have the same ordering as the corresponding plaintext blocks. ECB is the quickest of all modes and it can be fully parallelized in both encryption and decryption. Errors do not propagate between different blocks. This means that, in case of flipped bits in one block, the rest of the message is still recoverable. However, block independence eases substitution attacks. This is especially the case with redundant plaintext. Messages with a fixed structure (for example with sender and receiver headers at the beginning) might facilitate the attacker in breaking the code.

In Cipher Block Chaining (CBC), each plaintext block is XORed with the encrypted output of the previous block. The XOR result is then passed through DES to produce a new block. This ordering is very important: using XOR after the DES step would result in a much weaker encryption. The first block, having no predecessor, is XORed with an (usually public) Initialization Vector (IV). The IV needs to be fresh, that is, produced by a strong source of entropy for each different message. All blocks are encrypted with the same key. Decryption works in reverse. The ciphertext is DES-decrypted, then XORed with either the previous ciphertext block, or the IV in the case of the first block.

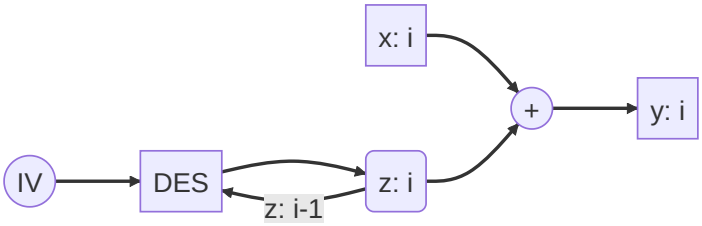


CBC is slower than ECB and, unlike it, suffers from error propagation. This makes it sensitive to the Padding Oracle attack. The process is serial and cannot be parallelized. On the flipside, blocks depend on each other, preventing substitution attacks. Until the discovery of the previously mentioned Padding Oracle attack, CBC was considered the reference solution for DES. Let's now analyze the vulnerability behind this type of attack. First of all, changing one bit within the IV has a 50% chance of flipping all the bits in the first block. The change might then propagate in the consecutive blocks. The Padding Oracle attack relies on the padding checker, which is, the component that validates the structure of padding after decryption. This mechanism can be exploited to reverse engineer an IV that produces a correctly padded output. There is a special type of IV, called a zeroing IV, that can turn the whole first block into an array of zeroes. The zeroing IV has another interesting property, highly useful to attackers: the XOR between it and the ciphertext produces the plaintext.

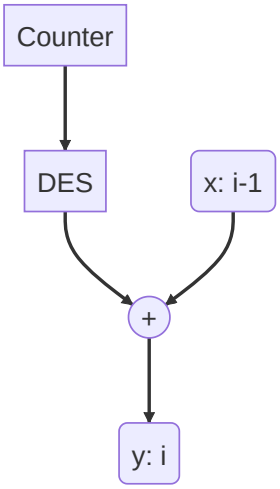
Cipher FeedBack (CFB) substitutes the last j bits from each block with the last j bits from the previous encrypted blocks. In the case of the first block, the j bits come from an IV. The substituted block is then DES-encrypted. The first j bits are XORed with j bits from the plaintext, and this produces the block output. From it, the last j bits are selected and used for substitution int he next block, and so on. The value of j can be chosen arbitrarily, but increasing it negatively affects the speed. It's an encryption-only mode.



In Output FeedBack (OFB), the plaintext is divided in n fixed-size blocks. An IV is then fed back through DES n times. This produces n sequences that are all independent from the message. Finally, each plaintext block is XORed with the corresponding DES-outputted sequence. The operation can be modified to employ a j-bit approach similarly to CFB. The XOR operations are quick and the blocks are independent. This gives OFB similar pros and cons to ECB. The main issue is that changing one bit in the plaintext modifies exactly one bit in the ciphertext. However, this makes message tampering very clear and easy to detect. Like CFB, OFB is strictly serial and encrypt-only.



The last classical mode is Counter (CTR). It employs a counter with the same dimension as the plaintext. This counter is then DES-encrypted and XORed with the plaintext. The technique is simple and efficient on both hardware and software implementations. The counter can be precomputed. It allows for random access and the safety is easily demonstrable. Like CFB and OFB, CTR is encrypt-only. Like OFB, CTR is very fragile because changing one input bit changes exactly one bit in the output.



Galois Counter Mode (GCM) is based on the mathematical theory of Galois fields. It XORs the output of CTR with the Galois field transform of the previous ciphertext block (or initial authentication data for the first block) to produce an authentication tag.

XOR Encrypts XOR (XEX) mode is used for large stored data because it has no size overhead. An interesting property is that the same plaintext content in different areas will correspond to different ciphertext. Instead of using a standard padding, the last plaintext block is padded with the corresponding section of the second-to-last ciphertext block. This practice is known as *ciphertext stealing*. This prevents Padding Oracle attacks.

3.1.2 Security issues and multiple key encryption

Let's now analyze the weaknesses of the algorithm. A key is called *weak* if it leads to the generation of 16 equal subkeys for all iterations. A couple of keys is said to be *semi-weak* if they produce only two different subkeys, each produced by 8 rounds. There are four weak keys and six semi-weak couples. As another weak spot, DES enjoys the *complement property.* This means that the DES encryption of the complement of the plaintext with the complement of the key produces the complement of the output. This halves the number of independent keys. For this reason, a brute force chosen plaintext attack only takes 2^{55} attempts instead of 2^{56}. DES was eventually broken in the late 1990s through a series of rewarded challenges that were opened to private users and companies by NIST. These were brute-force attacks with 2^{56} possibilities, or 2^{55} for optimized algorithms working with complements. More sophisticated attacks, based on linear (2^{43} attempts) and differential (2^{47} attempts) crypto-analysis, were carried out through a combined plaintext / ciphertext attack.

The NIST challenges made clear that the 56-bit DES keys were too short to offer significant protection. Thus, the idea of *double encryption* came forth. It consists in encrypting the plaintext with one key, and then re-encrypting it with a second key. The main question is whether a double encryption might give any advantage over a single key. The first factor to consider is that DES has 2^{56} possible keys. The permutations provided by a double DES encryption are $2^{64}! = 10^{10^{20}}$ corresponding to 2^{64} possible inputs. Therefore, the result of most double encryptions is not achievable with a single encryption using a different key. This means that the set of DES key permutations is not close by composition. This last result was mathematically proven in 1992.

There is, however, a possible type of attack that takes away the utility of a double encryption. This is known as the *meet in the middle* attack. It requires at least one known plaintext / ciphertext couple. The first step is to encrypt the plaintext with all the possible 2^{56} values of the first key. The second is to decrypt the output with all the possible 2^{56} values of the second key, while looking for matches with the provided ciphertext. The process thus requires 2^{56} encryptions, 2^{56} decryptions and 2^{56} table lookups, which may have $O(1)$ complexity when using a hash table or require up to 56 iterations in an ordered array. The two keys are found by brute force. The chance of finding a match that is accidental, that is, finding a different couple of keys than the ones really used, is 2^{48}. However, by making a cross-test over another available plaintext/ciphertext couple, the chance of an accidental match is reduced to 2^{-16} which is statistically insignificant.

The importance of the attack is that it demonstrates that an attack with 2^{56} attempts can break a double encryption whose total key length is $112 = 56 + 56$. Having three or more steps makes the attack more difficult. This is because the storage tables become very large. This is because the table size is the Cartesian product of key sizes. However, there is a variant of the attack that only requires 2^{56} storage entries even in this case. The time complexity is still affected by the 2^{112} encryption / decryption cycles. The pattern of the attack is that n encryption passes with key size k are equivalent to a single encryption pass with key length $(n-1) \cdot k$.

A convenient solution, that was readily adopted in the period of time between the demise of DES and the rise of AES, is called 3-DES. It is a chain of $\mathrm{DES}_k \to \mathrm{DES}_{k'}^{-1} \to \mathrm{DES}_k$. The decryption is carried out with the $\mathrm{DES}_k^{-1} \to \mathrm{DES}_{k'} \to \mathrm{DES}_k^{-1}$. If $k = k'$, then 3-DES is equal to DES.

3.2 AES

As the weaknesses of DES became more and more evident in the years leading up to 2000, NIST started to envision a new standard called AES. It was intended to be a faster, more secure and more efficient alternative to 3DES. The proposals were required to be royalty-free. The minimum length requirement for blocks was 128 bits. Supported key sizes needed to include 128-bit, 192-bit or 256-bit keys. NIST released a public bid that

was won by the Rijndael proposal. It won not mainly for its security, but for its speed on hardware, software, and smart cards. Another point of favor was its ease of implementation. This algorithm, unlike DES, is not based on the Feistel cipher. Feistel ciphers process one half of the data block at a time and then swap halves. Rijndael, on the other hand, can process the entire input block in parallel. It's an iterative block cipher which uses a variable number of rounds depending on key length (10 rounds for 128-bit, 12 round for 192-bit, 14 rounds for 256-bit). Block sizes, like key sizes, can be 128, 192 or 256 bits. Key scheduling employs 44, 52 or 60 32-bit subkeys.

Each AES round is a parallel and uniform composition of four transform steps:

- `SubBytes` (S-Box substitution)
  - inversion based on Galois fields
  - affine transformation
- `ShiftRows` : cyclical shift of matrix rows
- `MixColumns` : linear transformation
- `AddRoundKey` : `XOR` the state with the round key

3.3 Other symmetric ciphers

This section analyzes some other symmetric ciphers. These include both block ciphers and stream ciphers. Stream ciphers can be seen as time-dependent encryptions of single characters from the plaintext. In a stream cipher, messages are encrypted one bit (or byte) at a time. The message is combined bitwise with a *keystream*, that is, a pseudo-random sequence generated from the key. Stream ciphers are very fast and efficient. Security depends on the length and randomness of the keystream.

With the turn of the century, the cryptography world slowly started to move away from the traditional Feistel architecture for block ciphers. Modern block ciphers are characterized by a few notable characteristics. Many of them offer a variable key length, a variable block size and a variable number of rounds. The inner workings include the use of different mathematical and boolean operators, to increase variability and security. Other characteristics include data-dependent rotations, key-dependent S-boxes and whole-block, parallel operations.

3.3.1 Blowfish

Blowfish was invented by Bruce Schneider in 1993. It's a Feistel block cipher with 64-bit blocks, iterated over 16 rounds. The key has a variable length, and can be composed of 1 up to 14 32-bit words. Blowfish is designed to be fast and efficient. On 32-bit CPUs, it can encrypt data at a rate of 18 cycles per byte. It also has a very low (5 KB) memory footprint, and is relatively easy to implement and analyze.

The process begins with *key expansion*. This phase converts the key, known as k-array, into an expanded *p-array* of 18 subkeys, each 32 bits long. Blowfish employs 4 key-dependent S-Boxes. Each of them has a size of 8×32, for a total of 256 32-bit words for each. These boxes are generated during key expansion. Both the *p-array* and the S-boxes are initialized from the fractionary part of π. After the initialization, the *p-array* and *k-array* are XORed together. The round function is $F[a, b, c, d] = ((S_{1,a} + S_{2,b}) \oplus S_{3,c}) + S_{4,d}$. Since the algorithm is XOR-based, decryption simply follows the same process but in reverse.

521 rounds of encryption are necessary to create the final p-array and S-boxes. This makes Blowfish unapt for situations in which keys are changed frequently. On the other hand, the algorithm is highly resistant to brute force attacks, because it would require 521 attempts for each possible key. Another strong point of Blowfish is its updated take on the Feistel architecture. Unlike a regular Feistel cipher, Blowfish protects both the left and right side of each block. The right half, however, is still weaker than the left half.

3.3.2 RC4

RC4 is a stream cipher designed by Ron Rivest in 1987. It uses a variable-length key to initialize a permutation of all 256 possible bytes, which is then used to generate a keystream. This keystream is XORed with the plaintext to produce the ciphertext, enabling fast and efficient encryption.

3.3.3 RC5

RC5 is a symmetric-key block cipher designed by Ronald Rivest in 1994. It features a variable block size, key size, and number of rounds, making it highly flexible. RC5 uses a combination of modular addition, bitwise XOR, and data-dependent rotations to encrypt data, providing strong security with efficient performance. It also uses two magical constants, one based on the binary expansion of e and the other on the golden ratio φ. However this architecture, relying heavily on shifts, only works on little-endian CPUs.

4 Hash functions

A *hash function* is an operator that produces a fixed-size, unique output (called *hash*) from a variable-size input. Hash functions need to be deterministic, which means that the same output should always be produced from the same input. The fixed-size output has to be, on average, shorter than the message, to achieve some form of compression. The hash needs to be a non-ambiguous and non-falsifiable representation of the original message. Practically, this means that hash functions must be designed to minimize collisions. A collision occurs when two different inputs produce the same output. By ensuring visibly different outputs for different inputs, hash functions can be used for many different applications in the computer security domain. On top of being collision-resistant, hash functions are also required to be hard to reverse. Knowing the hash and the function should not be enough to determine what was the original message.

Hash functions are used in a variety of applications including, but not limited to, digital signatures, data integrity checks and timestamp certifications. In the domain of digital signatures, the signature is not applied directly to the document. This is because the whole document could be very long. For this reason, the signature is conventionally applied to the hash of the document. For a similar reason, data integrity checks are more easily performed by comparing the hashes of two different versions of a document. This is because the fixed-size hashes are orders of magnitude smaller than the average file.

4.1 Attacks and exploits on hashing

The basic principle behind most attacks on hash-based protocols is finding collisions. Collisions can be exploited by an attacker to produce an altered copy of a document with the same hash as the original. Such document would pass hash-based integrity checks exactly as the original would. Collisions are certain to happen when the number of possible inputs to a hash function is greater than the possible outputs. A number of outputs below 2^{40} is considered highly insufficient. A number of 2^{80} is on the border of feasibility. For modern non-critical applications, 2^{160} possible inputs may be enough.

Let's now consider a specific type of attack. Given $h : X \to Z$ with $|X| = m$ and $|Z| = n$, the attacker randomly chooses different messages to hash until a collision is found. What is the chance to find a collision with a limited number of attempts? This problem relates to what is known as the *birthday paradox*.

> **Birthday paradox**: *what is the minimum size of a group of people such that at least two people have the same birthday with a probability higher than 50%?*

In a perfectly uniform distribution, the chances are low. In reality, birthdays are not uniformly distributed and most births are centered around a few periods of the year. More importantly, it is enough to find a match within *a couple* of people. This is advantageous because the number of couples increases quadratically with the number of people: $n_{couples} = \frac{(n_{people}) \cdot (n_{people} - 1)}{2}$. This leads to finding that the minimum number is around 22 people.

In general, by randomly choosing two elements z_1, z_2 from a set of cardinality n, the chance of them being different is

$$\text{Prob}(z_2 \neq z_1) = 1 - \frac{1}{n}$$

By picking three elements (z_1, z_2, z_3), the chances increase:

$$\text{Prob}(z_3 \neq z_1 \wedge z_2 \neq z_1 \wedge z_2 \neq z_3) \geq \text{Prob}(z_2 \neq z_1)$$

By noticing that the pattern becomes a converging series, this can be generalized to a time formula:

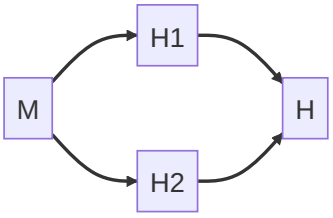
$$t \approx \sqrt{n \cdot 2 \ln\left(\frac{1}{1-\varepsilon}\right)}$$

If $\varepsilon = 0.5$ then $t \approx 1.17\sqrt{n}$. The flipside of this principle is that to completely cover a set with random extractions of its n elements, the number of attempts is around n^2.

The main takeaway for hash function attacks is simple. Given a hash function with n possible inputs, there is a 50% chance of breaking it by making $\sqrt{n}$ random attempts. This means that a function with 2^{80} inputs could be realistically broken with 2^{40} attempts. This assumes that all the equivalence classes have the same cardinality, which is the best case for hash function creators. This "birthday attack" approach may give a significant advantage compared to a systematic brute-force approach in which each possibility is attempted in order. A special type of data structure called *rainbow table* stores pre-computed hashes for the purpose of making collisions attacks faster, at the expense of significantly increasing their memory footprint.

4.2 Countermeasures, security techniques and implementation

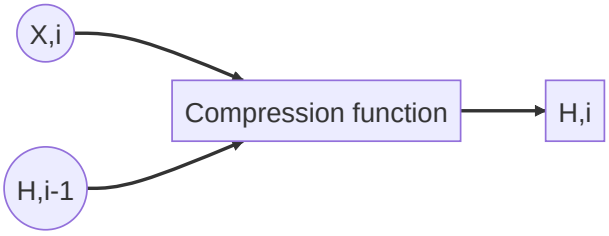
A simple solution against collision attacks is found in *hash function composition*. The technique consists in passing the same message through two different hash functions in parallel. The outputs are then combined into a single final hash.



$$H(M) = H_1(M) \circ H_2(M)$$

Finding a collision for the overall function $H(M)$ means finding two inputs that make both functions H_1 and H_2 collide at the same time, which is highly unlikely.

Being able to accept an arbitrary-size input was one of the requirements for hash functions listed at the beginning of this section. However, most compression functions have a fixed size input. Messages shorter than input size need to be padded. Messages longer than the input size need to be cut into fixed-size blocks (of which the last may be padded). Then, the function is applied iteratively to each block, combined with the hash of the previous one in a chain. The first block, having no predecessor, is combined with an IV. The iterative process is shown in the diagram below:



4.3 MD functions

One of the most popular families of hash functions is MD (Message Digest), created by Ron Rivest. These functions have been widely used for integrity checks of downloaded data, especially in the open source world.

Message Digest 4 (MD4) was created by Rivest in 1990. It is optimized for little-endian 32-bit CPUs. MD4 produces a 128-bit output from any string of arbitrary length. Its original requirements were strong security (at least for the time), direct security, speed, simplicity and compactness. *Direct security* means that the security of the function does not depend on hard mathematical problems. Conversely, many asymmetric cryptography algorithms are based on non-polynomial problems, which offer indirect security. MD4 processes the message in blocks of 16 32-bit words, which totals to 512 bits. Each message is thus padded to reach a multiple of 512 bits in length. The algorithm works in 3 rounds, containing 3 logical functions and 2 additive constants. The three rounds are applied 16 times in total. MD4 only lasted 5 years before being broken in 1995.

The successor to MD4, called MD5, was published by Rivest in 1991. It employs 4 rounds of 16 operations, 4 logical functions and 64 additive constants. The result of each step is added to the result of the previous. MD5 is not secure anymore, having been broken by the 2010s, but is still employed in minor *weak security* tasks.

4.4 SHS

The *Secure Hash Standard* (SHS) was defined by NIST around 1993-1994. It lead to the creation of the *Secure Hash Algorithm* (SHA). SHA was originally designed to be efficient on 32-bit big-endian CPUs. Its first iteration, SHA1, is similar to, but more secure than, MD4 and MD5. Its higher security comes from having a greater input size: 160 vs 128 bits. Compared to MD, SHA1 is also simpler. It uses 512-bit blocks, with padding for shorter blocks. Each block is then divided in 80 positions, 32 bits apart, that are processed in 80 separate iterations. Each of iterations uses a different additive constant.

The next revision, SHA2, was defined in 2000 by the NSA and adopted in the Federal Information Processing Standard (FIPS) in 2001. It was released in three versions, called SHA-256, SHA-386 and SHA-512. As the name implies, they respectively produce 256-bit, 386-bit and 512-bit hashes. SHA-256 kept the basic profile of SHA1, with the same 512-bit block size and 32-bit words. SHA-512 uses 1024-bit blocks and 64-bit words. It is more efficient than SHA-256 at the cost of a larger output. SHA-384 uses the first 384 bits of SHA-512 and uses modified constants.

4.5 Cryptographic hash functions

Cryptographic hash functions are a special family of hash functions that enjoy some additional properties that make them safer. Two important properties are *pre-image resistance* and *second pre-image resistance*. Pre-image resistance means that it should be computationally unfeasible (or at least very hard) to find the input given the hash. Second pre-image resistance means that it should be computationally unfeasible (or at least very hard) to find two colliding inputs. These properties make cryptographic hash functions particularly apt for digital signatures, key derivations, bit commitments and message authentication. A digital signature marks a digital document in a way that is authenticated and not falsifiable. Key derivation is a way to produce a certain number of keys by recursively hashing a master key, then using the produced hashes as secondary keys. Bit commitments are a way to commit to releasing some information in the future, while proving that it was effectively written in the past, and that it was not tampered between the time of writing and of opening. Finally,

message authentication can be achieved by putting a hash-based Message Authentication Code (MAC) in each message header.

Let's first dive into a description of the next generation of cryptographic hash functions, and then analyze their uses separately, in detail.

4.5.1 KECCAK hash functions

All basic hash functions are based on the ARX principle: Addition (modulo 2^n with n equals to 32 or 64), Rotation (cyclic shifts) and XOR (bitwise). ARX is elegant but is tied to a specific integer coding. It's only efficient in its software implementation when the CPU has n-bit words. The principles of ARX are behind the famous *Merkle-Damgård* architecture used by many hash functions. ARX also enjoys good cryptographic properties, but they are hard to analyze and prove. Another element of traditional hash function is a construct known as *Merkle-Damgård strengthening*. In this process, the input length is added to the input string. This helps prevent certain types of attacks, such as length extension attacks, by ensuring that the hash function processes the entire message, including its length, In some cases, an IV' is added into the last step for additional security. Other principles include *mask generating functions*. In such functions, multiple hashings are performed in parallel. The last block of each parallel hash includes a counter value juxtaposed with the modified initialization vector (IV'). The outputs are concatenated to produce a mask. This mask can be used for enhancing the security of the hash function. One last piece of classical hash functions is a Davies-Meyer compression function. This construct uses a block cipher to process the last block, using the hash of the previous block as a key. The output is then XORed with the key itself.

SHA1, MD4 and MD5 were all broken by 2010. For this reason, NIST opened a competition for the definition of SHA-3 in 2007. The security requirements included strong pre-image resistance and second pre-image resistance. NIST desired to find a function that did not allow any attack that was more efficient than the most basic brute force approach. The winner was the proposal by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche, which was submitted in 2008. KECCAK uses a sponge construction, which is fundamentally different from the Merkle-Damgård construction used by previous hash functions like SHA-2. The sponge construction involves absorbing input data into a large internal state, applying a permutation function, and then squeezing out the hash value. Specifically, KEKKAC uses an array of $5 \times 5 \times 2^l$ bits to represent the state. This is an expansion of the idea of matrix shuffling used by AES. The approach provides flexibility and efficiency for flexible input and output sizes (224, 256, 384, and 512-bit outputs), making SHA-3 adaptable to different use cases. The KECCAK proposal won the NIST bid in 2012 and became the official algorithm behind SHA3.

A derivative version, called TurboShake ("turbo SHA3"), is a version of Kekkac optimized for speed. Another fast family of hash functions is the Blake family. Blake2 was created in 2013. Blake3, the current iteration, was released in 2020. At the moment, Blake3 is the fastest modern hash function, at the cost of not being the most secure. Blake functions are based on the ChaCha stream cipher by D. J. Bernstein.

4.6 Timestamps and signatures

Timestamps and signatures are ways to demonstrate, respectively, the temporal origin or the authenticity of digital documents. Both are based on cryptographic principles.

A *digital timestamp* is used to prove that a document was created at a specific time. It can also be used to prove that it came before, or after, a specific moment in time. Without recurring to cryptography, it's generally easier to prove that a document was written after a certain date than before it. For this purpose, it's usually enough to reference past events with a sufficient level of detail. Classical approaches to formally prove that the document was created before a certain date include notaries, self-mailing, newspaper publishing and protocol registries. All these methods rely on trusted authorities and can be combined with digital signatures. Authority-based approaches can prove naïve because the authority could be corrupt. Other issues may come from file

size and privacy. For example, even if patented ideas are protected by law, they are made public at the moment of registration.

Using the hash of the document, instead of the document itself, solves both size and privacy issues. The solutions for authority trust issues cluster into two possible families. The first is the family of *distributed protocols*. The trusted authority is removed altogether, in favor of multiple witnesses distributed in time. The document is hashed and its hash is sent to a certain number of randomly picked verifiers on a network. The distributed protocol suffers from two main issues. First, many potential responders need to be available on the network at any given time. Second, the lifetime of the produced signatures might be affected by the signature model being broken, or compromised private keys on part of the verifiers. Verifiers might also disappear from the network altogether.

The other family of solution relies on *linked protocols*. In this case, the trusted authority is kept. In addition, the stamps are chained with each other, so that a single record cannot be easily altered without breaking the chain. This method is described in the Haber-Stornetta protocol from 1991. The trusted authority receives all hashed requests at fixed intervals, then chains them and returns timestamps to each requester. The requests are grouped through a *hash tree*. In this data structure, invented by R. Merkle, each node contains the hash of the content of its two children. The leaves represent individual requests. A *superhash* is produced at the root of the tree with reference to the superhash of the previous block. To prove that a certain block was part of another, only some parts of the tree need to be provided. This is $\log_2 n$ hash values for a hash tree of n elements. The robustness of the cryptographic hash function is expected to prevent the insertion or modification of elements of the tree after the root hash is determined and published.

The two solution families are united in the concept of *blockchain*. Blockchains are distributed, link-based protocols. They use *proof-of-work* or *proof-of-stake* protocols that allow decentralized block generation and validation. However, they are less efficient than centralized linked protocols, especially in the case of proof-of-stake.

Like a physical signature, a *digital signature* needs to be easy to be made, easy to check and hard to falsify. The naïve solution would be to stick a digitalized signature image to a document. However is very easy to falsify. Modern solutions are based on asymmetric cryptography. The signer uses a private key, unknown to the public, to sign the hash of the document. The signature can be checked by anyone else, using the known public key of the signer. The main two solutions in this domain are the RSA and DSA algorithms. The latter was implemented for smart cards around 1993. RSA offers slow signing and fast checking. DSA offers fast signing and slow checking. DSA also employs a pre-computation step, which is slower but increases security. RSA used to be patented and needed therefore the payment of a fee.

Digital signatures live in the domain of asymmetric cryptography. The equivalent in the domain of symmetric cryptography are *Message Authentication Codes* (MAC). They are used to check the authenticity and integrity of messages sent over networks. In general, symmetric cryptography is faster than asymmetric cryptography, usually at the cost of being less flexible. It might also be less secure because it only employs one single secret instead of two. It is important to note that MACs only provide authentication and integrity. To also add confidentiality, a second layer of encryption needs to be added. The shared encryption key must be different than the MAC key. This step might be placed either before or after MAC. Either position is acceptable, but the most used approach is to put encryption before MAC.

The possible attacks on MAC are the following. The first is the *known message* attack: the attacker is able to retrieve a list of messages along with their corresponding MACs. The second type is the *chosen message* attack: the attacker is able to compute the MAC of chosen messages. The third type, similar to the second, is the *adaptive chosen message* attack, based on previous messages. A *total break* attack consists in trying to find the MAC key. The fifth type is called *selective forgery*: the attacker is able to falsify a message by building a correct MAC for it. The last type of attack is known as *existential forgery* and consists in finding a message/MAC couple such that the MAC would be the correct for the given message..

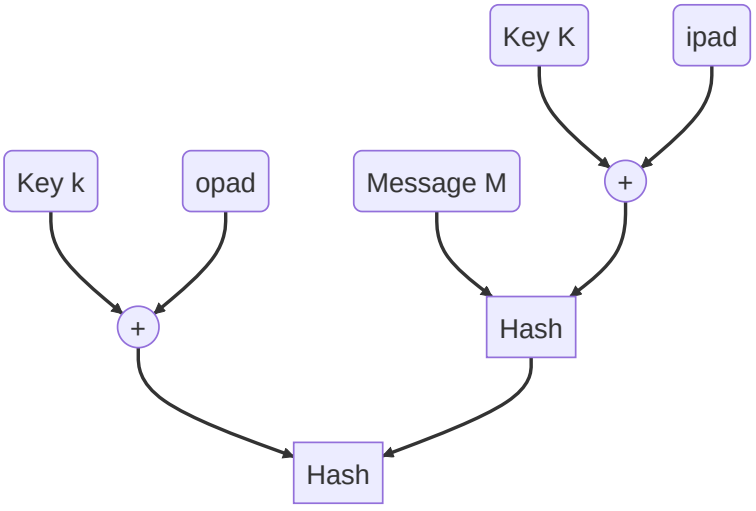
Given $k > n$, with k possible keys and n bits of length for the MAC field, around 2^{k-n} matches can be found through an exhaustive search on the keys. After this step, 2^{k-2n} matches can be found over a new message. If $k = \beta \cdot n$, around β rounds will be needed. This means that the computational effort is equals to $\min(2^k, 2^n)$. This value should be greater or equal than 128 by design to provide an acceptable security in a MAC system.

MACs are used not only for network message, but for integrity checks in relational databases too. While their impact on internet packets is very small, the cost can prove to be significant when used on single database tuples. Macs can either be constructed with block ciphers (known as CBC-MAC) or with hash functions (secret prefix HMAC, secret suffix HMAC). The historical approach used to be CBC-MAC. However this technique, being based on the CBC encryption mode, became obsolete after the discovery of the Padding Oracle attack. Hence, most modern solutions are based on HMAC.

CBC-MAC is based on DES, using the Cipher Block Chaining mode. The IV is set to 0. CBC-MAC was defined in two different standards, FIPS PUB 113 and ANSI X9.17, developed between . CBC mode was chosen because of its block diffusion property. The plaintext is cut in 64-bit blocks and the last block is used as MAC. Hash-based MAC (HMAC) used to be the fastest option before the widespread adoption of AES hardware acceleration. Hash functions still win in speed for software-only implementations. Hash functions have the ulterior advantage of usually having less restrictive licenses compared to block ciphers. In iterated hashing approaches, the message is padded and divided in blocks. The blocks are then compressed iteratively. The last output is used as MAC.

The version of iterated hashing that uses the secret as a prefix is susceptible to a type of attack known as *existential forgery*. By knowing the MAC and using it as the starting block, the process can be reiterated to process a new message whose last section is appended by the attacker. This process produces a modified message with a valid MAC. A possible solution against existential forgery consists to add the message lenght as a hashing parameter. By doing this, modified messages become detectable. Message with suffix secrets are also susceptible to a variant of existential forgery too. A "birthday attack" can be used to compute a second message with the same hash as the original message. In this case, the forged message will have the same MAC as the original.

HMAC was defined in three standards, RFC 2104, ANSI X9.71 and FIPS 198, between 1997 and 2002. HMAC uses hash functions as black boxes, with no modifications to their algorithm. It can therefore be adapted to any hash function. Changing the used function is easy, and so are key creation and storage. The process follows the scheme depicted below:



The result of the last hashing iteration is $H(K \oplus \text{opad}, H(K \oplus \text{ipad}, M))$. Input and output pads are `0x36` and `0x5C` respectively. Each is repeated $b/8$ times, where b is the block size. Sometimes only the first t bits of the hash are used. To provide an appropriate level of security, it should use at least half of the bits produced by the hash. The best possible attacks on HMAC are based on the birthday paradox.

4.7 Public Key Infrastructure for key distribution

The infrastructure used to publish public keysets for asymmetric cryptography is known as Public Key Infrastructure (PKI). There are many available techniques. The main approaches and variants can be clustered into five families. The first technique is based on point-to-point distribution on a trusted channel. The second is public announcement. The third is a publicly available key directory. The fourth is a public key authority. The fifth and last consists in public key certificates. In all cases, the foundational problem is that public keys may be subject to a man-in-the-middle attack. Asymmetric cryptography is based on the idea that anyone might be able to check the authenticity of a document through its public key. By releasing a fake public key in someone else's name, an attacker might deploy falsified documents that will look authentic when examined through the fake public key.

When two users want to communicate, they need to follow a three-step process. The first sends a message to the second, along with a personal ID and a nonce (a single-use random sequence), encrypted through the other user's public key. The second user will reply with a message encrypted through the first user's public key. This message will contain the nonce sent by the first user along with a new nonce. If the first nonce corresonds, the first user will reply back with the second user's nonce. This third message is encrypted through the second user's public key. After this process which confirms the identity of both, the two users can proceed with their communication.

A good example of the *public announcement model* is the PGP (Pretty Good Privacy) system, created by Phil Zimmerman in 1991. PGP provides a comprehensive cryptography model that offers encryption, decryption, digital signatures and compression. It combines symmetric-key encryption for data and public-key encryption for key exchange. It is commonly used for securing emails, files, and communications. PGP is based on a *web of trust* model, in which the public key becomes the signature for all the online presence of its owner, including e-mails and forum posts. This makes the falsification of the key much harder, because the attacker would need to build a whole online identity and persona around the falsified key. On the other hand, a true public key from a widely known person may become a much more highly prized target for an identity fraud attack. By stealing the corresponding private key, the messages forged by the attacker would be much harder to spot as false.

In a *public directory* model, a publicly available repository of public keys is managed by a trusted authority. Through an authenticated process, either in person or digitally, users can register their public key. Existing keys can be revoked when too old or when their paired private key is compromised. Key access requires a secure and authenticated connection. *Public key authorities* have a public key available to all of their users. Each user can request the public key of any other user. The PKA provides it, along with a digital signature signifying its authenticity. The main disadvantage is the need for a constantly available online server, which might become a bottleneck. Public keys might be cached to increase speed. Caching, however, might become a problem when keys are revoked. An invalid key might stay in the cache for a while after its revocation. This problem and its possible solutions will be discussed in a later paragraph.

A *digital certificate* can be seen as the digital equivalent of an ID card. A trusted authority binds the owner's name to a public key. The digital signature from the Certification Authority (CA) guarantees the link between the name and the key. Each certificate has a name, a key, a signature, a validity period and some more additional information. The most common format is defined by the ITU-T X.509 standard, which will be examined in more detail later. Digital certificates can also be exchanged directly by two users, without the intervention of the CA. This is different from the Identity Provider model, in which the authority intervenes in each transaction.

When a private key is compromised, the certificate relative to its corresponding public key can be revoked by its legitimate owner. There are many ways to manage the process. The so-called *short-lived certificates* auto-expire after a deadline. This model can only succeed if renewal has no additional costs to the end users. Another technique consists in *manual notification*: for small scale systems, the owner might manually inform other users through a specialized channel. The third option, associated with CAs, is a *public file with revoked*

keys. An official *Certificate Revocation List*, signed by the CA with a release date, contains the serial numbers of revoked certificates along with the expiry date. The last approach, also tied to the use of CAs, is the *revocation certificate*. This certificate substitutes the original certificate in the main directory, acting as an "anti-certificate". As explained before, caching within the CA can cause issues with serving expired or revoked certificates after their due date.

The *X.509 certificate standard* was issued in 1988 by ITU-T. Versions 2 and 3 were released in 1993 and 1995 respectively. X.509 is used in S/MIME (email certificates), SSL/TLS, SET and IPSEC. The standard defines some name-value pairs that define the user, which are part of the `Issuer name` field. These include user ID (`uid`), email address (`e`), common name (`cn`), organization (`o`) and country (`c`). Other fields include the standard version, a unique serial number, the algorithm used for the signature, the issuing CA, the validity (not before / not after fields), the public key and finally a hash of the whole certificate. The most commonly used encoding is `base64`, which uses 64 plain text characters to represent binary data. Certificates issued by the CA to users are called *forward certificates*. Certificates that prove the CA's right to issue certificates are called *reverse certificates*. An example of reverse certificates are *root certificates* that are installed on user devices. Certification authorities need to cooperate to allow cross-authority interoperability to their users. To achieve this, X.509 prescribes a hierarchical tree model. This feature was never implemented concretely, because the major CAs managed to get their root certificates (whose name derives from being at the root of the tree) within all the major browsers and operating systems by paying the developers.

Another feature of the standard that was quickly restricted in the interest of certificate authorities is *peering*. According to X.509, a valid certificate can be used to vouch for the creation of other certificates. CAs soon tried to restrict this model to prevent their users from creating a competitor service through one of their certificates. This is obtained by using *certification path constraints*. The *basic constraints* assert whether a user can act as a lower-level certification authorities. If so, the length of the path may be restricted. In other words, child certificates can only vouch for other certificates for a limited number of levels. *Name constraints* are used to define a namespace to be imposed on all the certificates in a path. *Policy constraints* can be used to ban further policy mapping down the chain.

X.509 describes three possible authentication procedures. In *one-way authentication*, user A sends a signed message containing a timestamp, a nonce, the receiver's name (B), signature data and a fresh session key encrypted through B's public key. The same nonce is used until it expires. This, combined with timestamps, reduces the chance of a replay attack, in which the attacker re-uses the nonce from an intercepted message. In *two-way authentication*, B replies with a similarly structured message, that is, a signed message containing timestamp, nonce, receiver, signature data and a session key encrypted through A's public key. The last two elements are optional for B. In particular, sending the key may be optional for B unless it is based on asymmetric cryptography. Two-way authentication might still be susceptible to replay attacks, therefore it still needs timestamps. The third and final approach, *three-way authentication*, removes the need for timestamp checking. This can be useful for systems with non-synchronized clocks. A and B both send signed messages with the same structure as before. After this exchange, A sends B's nonce to B in a signed message.

In Italy, digital signatures and certificates have been regulated by law, therefore having legal course, since 1997, with the introduction of DPR 513/97. This was followed the next year by EU regulation 1999/93/CE.

PKIs suffer from many issues that limited their diffusion. First, the name itself can cause an ambiguity. Public Key Infrastructure can be interpreted as either "infrastructure for public keys" or "public infrastructure for keys". Only the first interpretation has produced usable implementations. Even among those, many still fared way below the expectations. Perhaps the most successful PKI is the SWIFT system used for digital banking. While being privately owned, SWIFT has become the most used provider of public keys for authenticating cross-bank transactions internationally. Other PKI issues include hard naming and the constant need for CRL access. CRL, being an explicit and open descriptions of invalidated certificates, can be exploited to find false certificates to use, while the information about their expiry still hasn't spread. This is analogous to what happened with booklets of stolen credit cards in the 1970s: scammers would forge cards with numbers taken

from the most recent stolen cards ("hot cards") and use them in remote areas, moving faster than the delivery of booklets.

Other problems with PKIs are the necessity to educate users, DNS name binding and an unclear liability model. In the SSL/TLS model (described in a later section), certificates are used to bind owner names to DNS registry entries. However, CAs do not check the DNS. Additionally, as it is also the case for digital payments, the name of the owner of the certificate of an online shop might be very different from the expected name of the seller. Finally, an unclear liability model can hold back users from adopting the model. One aspect that lead to the mass adoption of credit cards was the exclusion of liabilities to the end users - the credit card operator would refund any misuse from scammers. Physical signature systems are insecure from a physical standpoint but are made safe by their inefficiencies: many iterations, many documents required, many interactions between parties. Fast and efficient systems, on the other hand, have no intrinsic way to stop an ongoing transaction, if it's instantaneous. For potential users, taking on the burden of liability for misuse of stolen credentials could negatively offset any possible advantage to digital systems. An unclear policy from PKIs on this front was one of the causes of their slow and uneven adoption.

PKIs are used for public keys. The proper way to handle private keys is a different problem, that is usually left to the user. If individuals store their private keys locally, the keys are only as safe as their local computer. Using external providers may expose the user to the risk of a mass attack on the provider.

5 SSL/TLS

SSL (Secure Sockets Layer) is a cryptographic protocol designed to secure communication over a network by encrypting data between client and server.
It ensures privacy and data integrity by preventing eavesdropping or tampering.
It's commonly used for securing HTTPS web traffic, email, VPNs, and other sensitive transmissions. SSL authenticates servers (and optionally clients) via digital certificates. The first version, SSL 1.0, was designed by the Netscape project. Its use was not widespread. The second version, SSL 2.0, while gaining some more widespread adoption, had a few vulnerabilities. These include a vulnerable short key and weak construction of the MAC field. The padding length was unauthenticated. SSL 2.0 was also vulnerable to an attack known as *ciphersuite rollback*. The SSL connection supports different subprotocols to allow for different levels of security depending on the resources available by both client and server. The connection is thus established with the lowest common denominator protocol that is supported by both parties. Attackers could modify the `hello` messages at the moment of connection establishment to weaken it to the lowest possible level.

The length and complexity of the protocol specification kept growing with the years up until SSL 3.0. SSL is now largely replaced by TLS (Transport Layer Security) but the protocol still colloquially referred to as "SSL" or "SSL/TLS". The first version of TLS, TLS 1.0, was made to surpass SSL 3.0. TLS consists in two subprotocols. The first is the *handshake protocol*, used in the connection setup phase. It consists of three steps. First, the version of the protocol and the cryptoalgorithms to be used are negotiated between client and server. Then, optionally, client and server authenticate by exchanging their digital certificates. Finally, the two parties exchange a shared secret key, asymmetrically encrypted through each other's public keys. The second subprotocol is known as *record protocol*. The record protocol is used for the actual connection. Client and server exchange data packets secretly via symmetric encryption using the shared key. The integrity of packets is checked via HMAC. According to the standard, clients can check server certificates and, in principle, the server might check the users too. In most cases, only the server certificate is checked.

6 Other cryptographic techniques

The *interlock protocol* is a way to prevent man-in-the-middle attacks in the absence of a certificate authority. The connection is established in a five-step process. First, the two parties exchange their public keys. Then, each party encrypts half of their first message (for example, only the even bits) through the other's public key,

and sends it. After this, the first user sends back the second half of the message. Then, the second reconstructs the full message and sends the other half. Finally, the first user reconstructs the full message.

The *SKEY technique* is used to send a password on an open unidirectional channel. Since the channel is not bidirectional, it would not be possible to employ the usual challenge-response method. SKEY is based on a finite set of throwaway codes, made by recursively hashing an initial random values. Thus, the technique only works a limited number of times before exhausting all the available throwaway codes.

The *Needham-Schroeder protocol* is based on a *trusted third party* (TTP). Each user owns a symmetric key to communicate with the TTP. The TTP uses asymmetric cryptography to produce certificates for the users. The users can then use these certificates to communicate with each other independently from the TTP. The third party also produces nonces that one party may use to start the communication with the other. When user A wants to communicate with user B, it sends a symmetric key (generated by the TTP) within the first message. User B replies with a nonce. A sends back the nonce minus 1. The critical part of the protocol is the key. This is because the key has no expiry date, so it may be captured by a man-in-the-middle and used for replay attacks. This type of attack is prevented by adding timestamps to the messages.

The *Kerberos protocol* is a variant of Needham-Schroeder in which the TTP, instead of producing a non-expiring key, generates a set of three elements: a timestamp, a lifetime and a session key. The protocol has two issues. The first comes from the use of timestamps: both users and the TTP server need to have their clocks properly synchronized. The first issue is the need for TTP intervention for each exchange. The Kerberos implementation distinguishes between three categories of servers: *Authorization Server* (top-level), *Ticket Granting Server* (for refreshing expired keys) and generic servers. Another distinction is between *tickets* (reusable keys) and *authenticators* (single-use tokens generated by the client to access the servers). Kerberos was later extended into the *Active Directory* standard.

Let's now analyze *multiple public key encryption*. To send a message to n users, n keys are created. Each user will receive all the $n-1$ keys except for key i, where i is the ordinal number of each user. To get the original message, all the missing keys need to be applied. During decryption, the ciphertext is used as a power base and the key is the exponent. Keys are built in such a way that, by applying each necessary key as the exponent, the plaintext will appear. The process is a variant of the mechanics used by RSA. The main advantage of the protocol is that one single broadcast message can be sent to all users without worrying about the wrong destinataries in the group, which will not be able to open it. However, the system is easily broken by colluding users that may exchange missing keys. Another issue is that everyone knows how many users are allowed to see each message.

Secret sharing is a technique that enables making a secret readable if and only if a certain number of users collaborate. The simplest form is called *secret splitting* and requires *all* users to collaborate. A basic implementation of secret splitting uses a set of random sequences, one per user, to XOR with the secret. Given the message and n random codes, the process will produce $n+1$ splits: the n random codes and one overall result. More complex secret sharing systems use *threshold mechanisms*: at least m out of n users, $m \leq n$, are needed to unencrypt the secret. This approach prevents the risk of losing the secret due to the loss of a key or the unavailability of its owner. The classical implementation for (m, n) uses a $(m-1)$-grade polynomial, of which the last parameter is the secret, and the others are random. A piece of the polynomial is assigned to each user. The algorithm is based on modular algebra; a prime number is used as the ceiling for the message which is treated like a number. Having m or more elements is enough to reconstruct the full polynomial and thus the secret message.

A *fail-stop digital signature* is a technique envisioned to easily detect forgery on signed documents. A single public key can verify the product of a set of 2^n distinct private keys. Each key produces a valid signature, yet detectably different from the signature of the others.

Bit commitment refers to a family of approaches to make a non-revokable commitment to reveal the content of a bit (or a string of bits) without having to reveal the content in advance. The term *bit commitment* and the focus on one single bit is meaningful. This is because the more compact the secret is, the more difficult it becomes to prove that the choice was made earlier and not forged at a later time. A bit is the shortest possible content to make a commitment on: if a technique works for a single bit, it can also work for longer secrets. There are two main possible options, of which one is interactive, requiring active participation from the verifier from the get-go, and the other is not interactive. Let's refer to the committer as *user A* and the verifier as *user B*. In the first stage of the interactive technique, B sends a nonce to A. A picks a key and uses it to encrypt the nonce and the committed bit, then sends the result to A. In the second phase, when the commitment needs to be revealed, A sends the key to B. B can then inspect the bit while verifying the correctness of the nonce. The assumption behind this approach is that A cannot easily find a second key that would produce the same encrypted message but with the bit flipped. In the non-interactive solution, user A creates two nonces, then hashes their concatenation along with the committed bit. The hash is sent to B along with one of the two nonces. In the verification step, A sends the complete set (two nonces + the bit) to B. B hashes it and checks if the result matches with the original hash previously sent by A. In this case, the assumption is that A cannot easily find a collision in the chosen hash function. The advantage of the non-interactive approach is that the second party does not need to actively participate in the creation of the commitment, but only in the verification.

Other cryptographic techniques include *homomorphic encryption*, *oblivious RAM* and *secure multiparty computation*. The idea behind homomorphic encryption is to encrypt the data in a way that allows to apply transformations on it that are *homomorphic* to operations that could be applied to the plaintext. This means that the ciphertext can be directly updated without having to perform the usual decrypt/process/re-encrypt cycle. The approach could potentially save computation and bandwidth when used on cloud-stored data. In most real use cases, though, homomorphic techniques do not offer any significant advantage and may even perform worse than decryption and re-encryption. Homomorphic encryption that only allows addition operations is called semi-homomorphic. The most popular semi-homomorphic algorithm is *Paillier*, an exponentiation-based approach similar to RSA. Fully homomorphic encryption also allows multiplication. By combining addition and multiplication, all other transformations can be re-created. Oblivious RAM is a set of techniques to make the memory controller oblivious to the content of the memory. It is a part of the broader *oblivious computer system* model, which has the goal of allowing security in a distributed computing scenario. A notable example of oblivious RAM is the *Path ORAM* protocol. A concept related to the oblivious computer system model is that of *secure multiparty computation*, which relies on cryptographic techniques to provide confidentiality on the result of a distributed computation. These approaches can be applied in *outsourcing* (e.g. of data to a cloud provider) and *federated learning* (the training of ML algorithms on multiple decentralized devices holding local data samples without data sharing between nodes).