Advanced Aspects of Network Security

Authorship: Jordi Herrera-Joancomartí, Joaquin Garcia-Alfaro, Xavier Perramon-Tornil

Protection mechanisms

Table of contents

1. Cryptography foundations

Throughout history, various techniques have been devised to hide the meaning of information that is not desirable for outsiders to know. Some of them were already used in ancient Greece or the Roman Empire: for example, Julius Caesar is credited with inventing a cipher for sending encrypted messages that could not be interpreted by the enemy.

Cryptography

The terms **cryptology**, **cryptography**, **cryptanalysis**, etc. come from the Greek root *kryptós*, which means "hidden".

When the protection we seek is to guarantee the secrecy of information, that is, confidentiality, we use the cryptographic method known as encryption.

Use of ciphers

The use of encryption techniques is based on the assumption that trying to prevent an intruder from intercepting information is very costly. Sending encrypted messages is easier; they will not be able to interpret the information they contain.

Assume M is the message we want to protect, i.e., the plaintext. The encryption of M consists of applying an encryption function f that transforms the plaintext into another message, C, for *ciphertext*. This can be expressed as:

$$C = f(M)$$

For this cipher to be useful, there must be another transformation, or **decryption function** f^{-1}, that allows the original message to be recovered from the ciphertext:

$$M = f^{-1}(C)$$

The Caesar cipher

For example, the Caesar cipher we mentioned earlier consisted of replacing each letter of the message with the one that is three positions further down in the alphabet (starting again with the letter A if we reach the letter Z). This way, if we apply the Caesar cipher to the plaintext "ALEA JACTA EST" (and using the current Latin alphabet, because in Caesar's time letters like the "W" did not exist), we obtain the ciphertext "DOHD MDFWD HVW". The decryption in this case is very simple: it is only necessary to replace each letter by one that exists three positions earlier in the alphabet.

A scheme like the Caesar cipher has the disadvantage that if the enemy discovers the encryption algorithm (and deduces the reverse algorithm from this), he will be able to interpret all the encrypted messages he captures. This would require training all the army communications officers to learn a new algorithm, which could be complex. Instead, what is currently done is to use as an algorithm a function with a parameter called the key. A scheme like the Caesar cipher has the disadvantage that if the enemy discovers the encryption algorithm (and deduces the reverse algorithm from this), he will be able to interpret all the encrypted messages he captures. This would require training all the army communications officers to learn a new algorithm, which could be complex. Instead, what is currently done is to use as an algorithm a function with a parameter called **the key**.

We can then refer to an encryption function e with an **encryption key** k, and a decryption function d with a **decryption key** x:

$$\begin{aligned} C &= e(k,M) \\ M &= d(x,C) = d(x,e(k,M)) \end{aligned}$$

Thus, one solution to the problem of a spy who learns how to decrypt messages could be to continue using the same algorithm, but with a different key.

Example of using a key

Caesar's algorithm can be generalized by defining a key k that indicates how many positions each letter should advance in the alphabet. The Caesar cipher uses k equal to 3. For decryption, the same algorithm can be used but with the key reversed ($d \equiv e, x = -k$).

A fundamental premise in modern cryptography is what is known as the **Kerckhoffs's principle**. This principle establishes that algorithms must be publicly known, and their security depends solely on the key. Instead of trying to hide the workings of algorithms, it is much more secure and effective to keep only the keys secret.

Security through obscurity

Throughout history, there have been cases that have demonstrated the danger of basing protection on keeping algorithms secret (what is known as "security through obscurity"). If the algorithm is widely known, it's easier to detect weaknesses or vulnerabilities and quickly correct them. If not, an expert could deduce the algorithm through reverse engineering and end up discovering that it has weak points that can be attacked, as happened with the A5/1 algorithm for GSM mobile telephony.

An algorithm is considered **secure** if it is impossible for an adversary to obtain the plaintext M even if they know the algorithm e and the ciphertext C. That is, it is impossible to decrypt the message without knowing the decryption key. The word "impossible" must be considered with different nuances. A cryptographic algorithm is **computationally secure** if, applying the best known method, the amount of resources required (computing time, number of processors, etc.) to decrypt the message without knowing the key is much larger (a few orders of magnitude) than anyone can achieve. In the limit, an algorithm is **unconditionally secure** if it cannot be reversed even with infinite resources. The algorithms used in practice are (or attempt to be) computationally secure.

The act of attempting to decrypt messages without knowing the decryption key is known as a *decryption attack* If the attack is successful, it is often colloquially referred to as *breaking* the algorithm. There are two ways to carry out a *decryption attack*:

- By conducting **cryptanalysis**, i.e., by analytically exploring feasible ways to deduce the plaintext from the ciphertext.
- By **brute force**, i.e., by testing one by one all possible values of the decryption key x until one is found that produces a meaningful plaintext.

Attacks on the Caesar cipher

Continuing with the example of the Caesar cipher, a cryptanalytic attack could consist of analyzing the statistical properties of the ciphertext. The ciphertext letters that are most frequently repeated will likely correspond to vowels or the most frequent consonants; the most frequently repeated combinations of two (or three) consecutive letters probably correspond to the digraphs (or trigraphs) that typically appear most frequently in a text. On the other hand, a brute-force attack would involve attempting decryption with each of the 25 possible key values ($1 \leq x \leq 25$, if the alphabet has 26 letters) and seeing which of them gives an intelligible result. In this case, the amount of resources required is so modest that the attack could even be performed by hand. Therefore, this cipher would be an example of an insecure or weak algorithm.

Depending on the information available to the adversary, several types of attack can be distinguished (in order of least to most advantageous for the adversary):

Security evaluation

When evaluating the security level of an algorithm or, what is the same, the difficulty of attacks, it is usually considered that adversaries will have the possibility of carrying out attacks with known clear text.

Ciphertext-only attack. The adversary only has access to previously intercepted encrypted messages.

Attack with known plain text. The adversary knows which plaintexts correspond to certain ciphertexts, or can deduce some of these plaintexts from the context. This can allow for a faster search for the key.

Chosen ciphertext attack

Another type of attack can also be considered, which would consist of choosing ciphertexts and decrypting them to see which plaintexts they correspond to.

Attack with chosen clear text. The adversary has the possibility of obtaining the ciphertexts corresponding to the plaintexts he wants. With this, he can compare the results of introducing different variations, deduce characteristics of the key, etc.

We present next a more detailed presentation of some representative cryptographic systems used to protect communication systems, paying attention at the main characteristics of clear messages, encrypted messages, and keys as sequences of bits.

1.1. Symmetric key cryptography

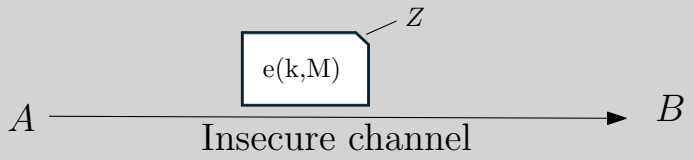
Symmetric-key cryptosystems are characterized by the fact that the decryption key x is either identical to the encryption key k or can be deduced directly from it.

For simplicity, we assume that in this type of encryption, the decryption key is equal to the encryption key: $x = k$ (otherwise, we can always assume that the first step in the decryption algorithm is to calculate the key x from the k). This is why these cryptographic techniques are called symmetric-key, or sometimes also shared-key. Thus, we have:

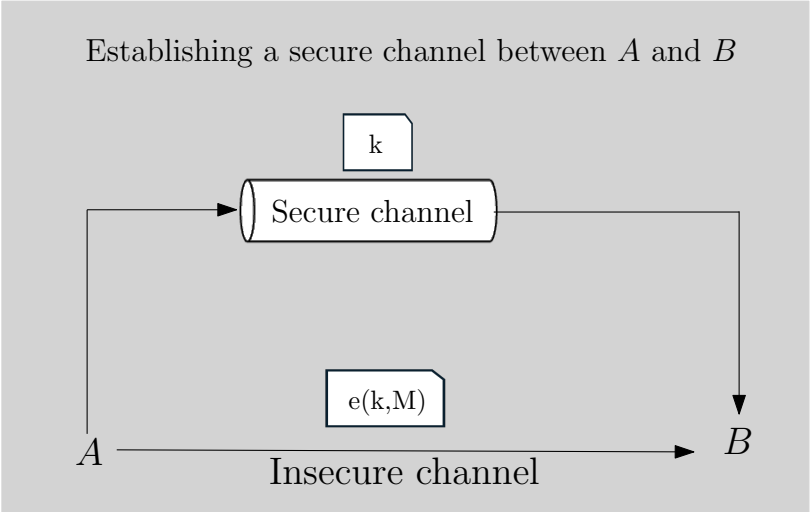
$$\begin{aligned} C &= e(k,M) \\ M &= d(k,C) = d(k,e(k,M)) \end{aligned}$$

The security of the system depends on keeping the key k secret. When participants in a communication want to exchange confidential messages, they must choose a secret key and use it to encrypt the messages. They can then send these messages over any communication channel, with the confidence that, even if the channel is insecure and susceptible to inspection by third parties, no spy Z will be able to interpret them.

Message M encrypted with key k known by A and B



If the system is a shared-key system, it is necessary that the value of the secret key k used by A and B be the same. How can we ensure that this is the case? It is clear that they cannot send the chosen key through the communication channel available to them, because the initial hypothesis is that this channel is insecure and anyone could discover the information transmitted through it. One possible solution is to use a separate channel that can be considered sufficiently secure:



This solution, however, has some drawbacks. On the one hand, it is assumed that the secure channel is not as easy or agile as an insecure channel (if it were, it would be much better to send all confidential messages unencrypted over the secure channel and forget about the insecure channel!). Therefore, it may be difficult to keep changing the key. On the other hand, this scheme isn't general enough: we may need to send encrypted information to someone we can't reach by any other means. As we will see later in this chapter, these problems related to key exchange are solved with public-key cryptography.

Secure channels

Examples of secure channels include traditional (non-electronic) mail or a physical courier service, a telephone conversation, or a face-to-face conversation, etc.

We survey next some important characteristics of the most representative symmetric-key cryptographic algorithms, which we can group in two main categories: stream cipher algorithms and block cipher algorithms.

1.1.1. Stream cipher algorithms

The operation of a stream cipher consists of combining the plaintext M with a ciphertext S obtained from the symmetric key k. Decryption requires only performing the reverse operation on the ciphertext and the same ciphertext S.

The main combination used by stream ciphers is based on additions. The inverse operation is therefore the subtraction. If the text is made up of characters, this kind of algorithms behave like the Caesar cipher, in which the key changes from character to character. The corresponding key is given by the ciphertext S (known as the *"keystream"*).

If we consider text made up of bits, addition and subtraction are equivalent.

Addition and subtraction of bits

In the world of Boolean algebra, as in the world of modulo 2 arithmetic, the following is always true:

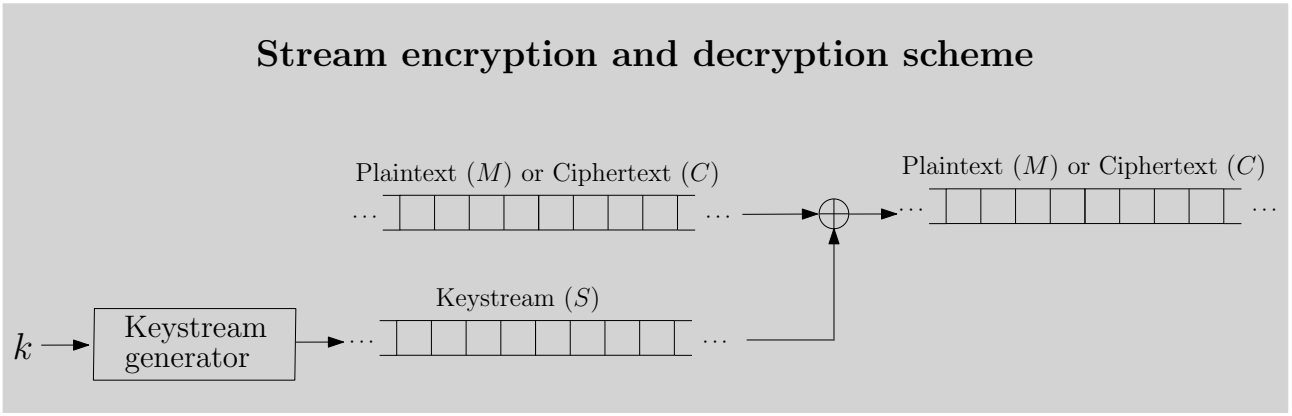
$$0+0=0 \quad 0-0=0$$
$$0+1=1 \quad 0-1=1$$
$$1+0=1 \quad 1-0=1$$
$$1+1=0 \quad 1-1=0$$

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

Indeed, when applied digit by digit on binary numbers, addition and subtraction are equivalent to the *exclusive OR* logical operation, denoted by the XOR operator (*"eXclusive OR"*) or the $\oplus$ symbol. Therefore:

$$C = M \oplus S(k)$$
$$M = C \oplus S(k)$$

In stream ciphers, the plaintext M can be of any length, and the ciphertext S must be at least as long. In fact, it is not necessary to have the entire message before beginning to encrypt or decrypt it, since the algorithm can be implemented to work with a *data stream* generated from the key (the ciphertext). This is where the name of this type of algorithm comes from. The following figure illustrates the basic mechanism of its implementation.



There are different ways to obtain the ciphertext S depending on the key k.

- If we choose a sequence k shorter than the message M, one possibility would be to repeat it cyclically as many times as necessary to add it to the plaintext.

  The main inconvenient with this method is that it can be easily broken, especially the shorter the key (in the minimal case, the algorithm would be equivalent to the Caesar cipher).

- At the other extreme, we could simply take $S(k) = k$. This means that the key itself must be as long as the message to be encrypted. This is the principle of the well-known **Vernam cipher**. If k is a completely random sequence that does not repeat cyclically, we have an example of unconditionally secure encryption, as defined at the beginning of this module. This encryption method is called *"one-time pad"*.

The problem in this case is that the receiver must have the same random sequence to be able to decrypt it, and if it has to reach him through a secure channel, the question is immediate: why not send the confidential message M, which is as long as the key k, directly through the same secure channel? It is clear, then, that this algorithm is very secure but not very practical.

- What is used in practice are functions that generate **pseudo-random sequences** from a **seed** (a number that acts as a parameter to the generator), and what is exchanged as the secret key k is only this seed.

  Pseudorandom sequences are so-called because they attempt to appear random, but they are clearly generated algorithmically. At each step, the algorithm will be in a specific state, determined by its internal variables. Since the variables will be finite, there will be a maximum number of possible distinct states. This means that after a certain period, the generated data will be repeated. For the algorithm to be secure, it is important that the repetition period be as long as possible (relative to the message to be encrypted), in order to make cryptanalysis difficult. Pseudorandom sequences must also have other statistical properties equivalent to those of purely random sequences.

  Synchronous and asynchronous encryption

  If the ciphertext S depends exclusively on the key k, the cipher is said to be synchronous. This cipher has the problem that if bits are lost (or arrive repeated) due to a transmission error, the receiver will become desynchronized and add bits of the text S with bits of the ciphertext C that do not match, so the deciphered text will then be incorrect.

  This can be avoided with asynchronous (or "self-synchronizing") encryption, in which the plaintext S is computed from the key k and the ciphertext C itself. That is, instead of being fed back its own state bits, the generator is fed back the last n encrypted bits transmitted. Thus, if m consecutive bits are lost in the communication, the error will affect at most the decryption of $m+n$ bits of the original message.

Using the Vernam Cipher

Sometimes, communications between aircraft carriers and aircraft use the Vernam cipher. In this case, the fact that at any given time (before takeoff) both the aircraft and the carrier are in the same location is used, and swapping, for example, a 20GB hard drive with a random sequence is not a problem. Later, when the aircraft takes off, it can establish secure communication with the carrier using a Vernam cipher with the random key that both parties share.

Pseudo-random functions

Examples of pseudo-random functions are those based on feedback shift registers (FSRs). The initial value of the register is the seed. To obtain each pseudo-random bit, all the bits in the register are shifted one position, and the one that comes out of the register is taken. The remaining free bit at the other end is filled with a value that is a function of the rest of the bits.

Stream cipher algorithms currently in use have the advantage of being inexpensive to implement. Hardware implementations are relatively simple and therefore efficient in terms of performance (in terms of bits encrypted per second). However, software implementations can also be very efficient.

The characteristics of stream ciphers make them suitable for environments where high performance is required and resources (computing capacity, power consumption) are limited. For this purpose, they are often used in mobile communications: wireless local networks, cell phones, etc.

Sample stream cipher

A well known stream encryption algorithm is **RC4 (short for Ron's Code 4)**. It was designed by Ronald Rivest in 1987 and published on the Internet by an anonymous sender in 1994. It is the most widely used stream cipher algorithm in many applications thanks to its simplicity and speed. For example, the WEP (Wired Equivalent Privacy) protection system incorporated in the IEEE 802.11 standard for wireless LAN technology initially used this stream cipher cryptosystem.

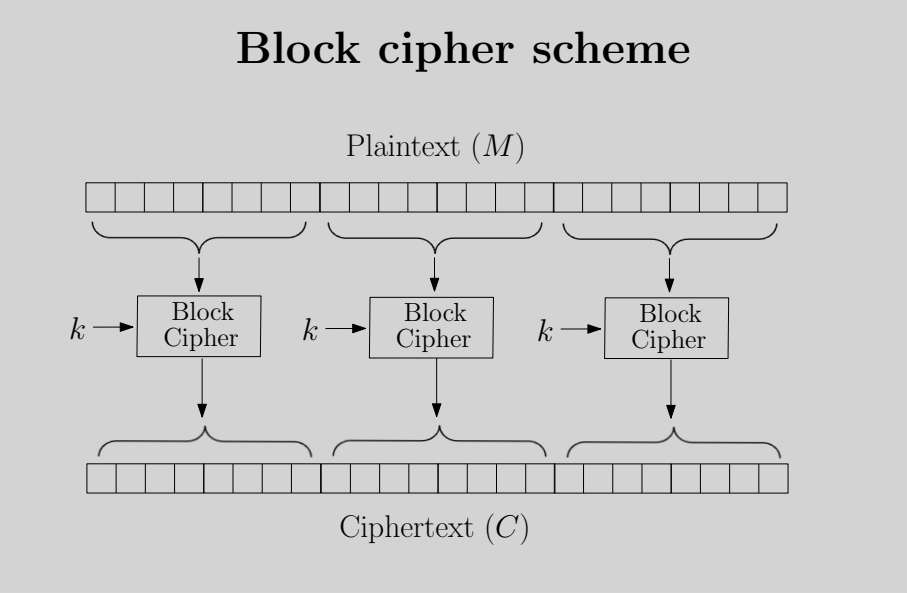
RC4 works with a 2048-bit (256-byte) state vector, initialized from a key k that can also have up to 2048 bits (if it is shorter, it is extended to 2048 bits by repeating it as many times as necessary). Each of the 256 bytes of the vector is exchanged with another, and the sum modulo 256 of both is calculated. The position of the second byte is determined by the value of the first. The result of the sum gives the next 8 bits of the ciphertext S. When the 256 bytes of the vector have been traversed, it starts again with the first. Logically, each time the bytes will be in a completely different order.

1.1.2. Block ciphers

> In a block cipher, the encryption algorithm or decryption is applied separately to input blocks fixed length b, and for each of them the result is one block of the same length.

To encrypt a plain text of L bits we must divide it into blocks of b bits each and encrypt these blocks one by one. yes L is not a multiple of b, additional bits can be added until you reach a number full of blocks, but then it can be necessary to indicate in some way how many bits were actually in it the original message. Deciphering must also be done block by block to block.

The following figure shows the basic scheme of block cipher:



Most block ciphers are based on the combination of two basic operations: substitution and transposition.

- **Substitution** consists of translating each group of bits from the entry into another, according to a determined permutation.

  The Caesar cipher would be a simple example of substitution, where each group of bits would correspond to a letter. In fact, this is a particular case of **alphabetic substitution**. In the most general case, the letters of the ciphertext do not have to be at a constant distance (the k of the algorithm, as we have defined it) from the letters of the plaintext. The key can then be expressed as the correlative sequence of letters that correspond to A, B, C, etc. For example:

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

  Plaintext: A L E A  J A C T A  E S T
  Plaintext: Q S T Q  P Q E Z Q  T L Z
  ```

- **Transposition** involves rearranging information in plaintext according to a given pattern. An example might be forming groups of five letters, including spaces, and rewriting each group $(1,2,3,4,5)$ in the order $(3,1,5,4,2)$:

  ```
  Plaintext:  A L E A   J A C T A   E S T
  Ciphertext: E A   A L C J A T A S     T E
  ```

  Transposition alone does not make cryptanalysis extraordinarily difficult, but it can be combined with other operations to add complexity to encryption algorithms.

Alphabetical substitution key

It is clear that the alphabetical substitution key must be a **permutation** of the alphabet, i.e., there cannot be any repeated letters or missing symbols. Otherwise, the transformation would not be invertible, afterward.

The **cipher product**, or cascading combination of different cryptographic transformations, is a very effective technique for implementing fairly secure algorithms in a simple manner. For example, many block cipher algorithms are based on a series of iterations of substitution-transposition products.

Confusion and diffusion

Two desirable properties in a cryptographic algorithm are (i) *confusion* which involves hiding the relationship between the key and the statistical properties of the ciphertext; and (ii) *diffusion*, which spreads the redundancy of the plaintext throughout the ciphertext so that it is not easily recognizable.

Confusion means that changing a single bit in the key changes many bits in the ciphertext, and diffusion means that changing a single bit in the plaintext also affects many bits in the ciphertext.

In a circuit combining several ciphers, substitution contributes to confusion, while transposition contributes to diffusion. The combination of these simple transformations, repeated several times, causes changes in the input to propagate throughout the output due to an avalanche effect.

Sample block ciphers

DES (Data Encryption Standard). For many years, DES was the most studied and widely used block cipher. Developed by IBM during the decade of 70s, DES was adopted as the de facto standard for data encryption in 1977 by the American National Bureau of Standards (NBS).

The American National Bureau of Standards (NBS)...

... was renamed in 1988 to NIST (for National Institute of Standards and Technology).

The algorithm supports a 64-bit key, but only 7 out of every 8 bits are involved in the encryption. Additional bits of the DES key are used as parity bits. Thus, the effective key length is 56 bits. The text blocks to which DES is applied must be 64 bits each.

The core of the algorithm consists of dividing the input into groups of bits, performing a distinct substitution on each group, and then transposing all the bits. This transformation is repeated 16 times: in each iteration, the input is a distinct transposition of the key bits added bitwise (XOR) with the output of the previous iteration. As designed, the decryption is performed the same as the encryption, but performing the key transpositions in reverse order (starting with the last one).

Triple DES. Although over the years the DES algorithm proved to be highly resistant to cryptanalysis, its main problem ultimately became its vulnerability to brute-force attacks, due to the key's length of only 56 bits. While in the 1970s it was very expensive to search among the 2^{56} possible combinations to attack DES, by the late 1990s technology had made it possible to break the algorithm in increasingly shorter times. For instance, in 1999 it was already possible to break DES in less than 24 hours. For this reason, in 1999 NIST decided to change the DES algorithm to *Triple DES*' as the official standard, waiting for the results of a later competition to decide the followup (AES, which we will see later). Triple DES, as its name suggests, consists of apply-

ing DES three consecutive times. This can be done with three keys (k_1, k_2, k_3), or with just two different ones (k_1, k_2, and again k_1). The total key length with the second option is 112 bits (two 56 bit keys), which is now considered sufficiently secure; the first option provides more security, but at the cost of using a total key of 168 bits (three 56 bit keys), which may cost a little more to manage and exchange. To make the system adaptable to the older standard, Triple DES uses an encryption-decryption-encryption (E-D-E) sequence instead of three ciphers:

$$\begin{aligned} C &= e(k_3, d(k_2, e(k_1, M))) \\ \text{or: } C &= e(k_1, d(k_2, e(k_1, M))) \end{aligned}$$

Thus, taking $k_2 = k_1$ we have a system equivalent to the simple DES.

Why not doble DES?

The reason NIST decided to apply three iterations of the original DES algorithm, instead of two iterations, is because an adversary with sufficient resources could break the result of computing Double DES by brute force, using a known plaintext attack, with less than 2^{57} encryption operations instead of the expected 2^{112}. Likewise, applying DES encryption with one key and re-encrypting the result with another key is not equivalent to a single DES encryption (i.e., no single key gives the same result as the other two combined). If this were not the case, repeating DES would not be more secure than simple DES.

AES (Advanced Encryption Standard). Since the DES standard was becoming outdated, mainly due to its short key lengths, and Triple DES was not particularly efficient when implemented in software, in 1997 NIST called on the cryptographic community to submit proposals for a new standard, AES, to replace DES. Of the fifteen candidate algorithms accepted, five were chosen as finalists, and in October 2000 the winner was announced: the Rijndael algorithm, proposed by Belgian cryptographers Joan Daemen and Vincent Rijmen.

Rijndael can work in blocks of 128, 192, or 256 bits (although the AES standard only provides for 128 bits), and the key length can also be 128, 192, or 256 bits. Depending on the key length, the algorithm's iteration count is 10, 12, or 14, respectively. Each iteration includes a fixed byte-by-byte substitution, a transposition, a transformation consisting of bit shifts and XORs, and a binary addition (XOR) with bits obtained from the key.

1.1.3. Using symmetric key algorithms

When using symmetric encryption to protect communications, you can choose the algorithm that best suits the needs of each application: typically, higher security means lower encryption speed, and vice versa.

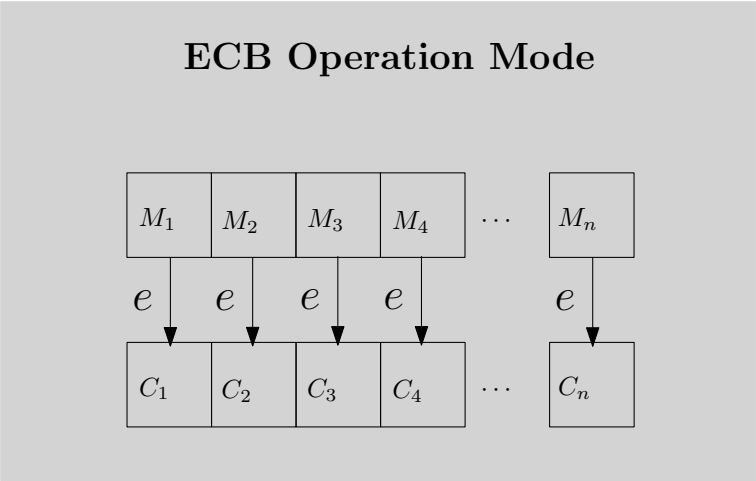
One aspect to keep in mind is that, while encryption can prevent an attacker from directly discovering transmitted data, it is sometimes possible to indirectly deduce information. For example, in a protocol that uses messages with a fixed header, the appearance of the same encrypted data multiple times in a transmission can indicate where the messages begin.

This happens with a stream cipher if its period is not long enough, but in a block cipher, if two plaintext blocks are equal and the same key is used, the encrypted blocks will also be equal. To counteract this property, various modes of operation can be applied to the block cipher.

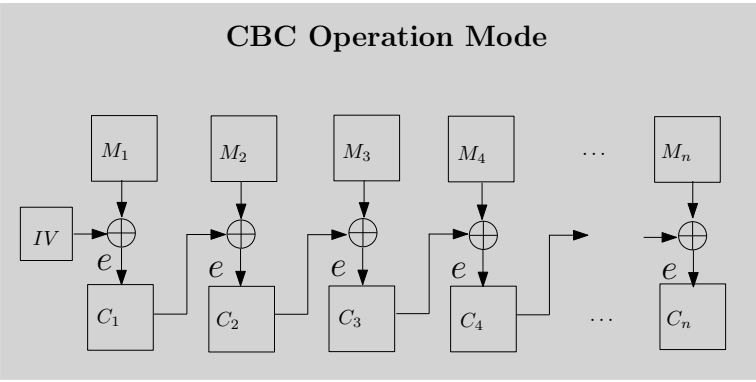
- The ECB (Electronic Codebook) mode is the simplest, and consists of dividing the text into blocks and encrypting each one independently. This mode addresses the problem of matching blocks when the input contains identical blocks, as shown below:

ECB mode

The ECB name (i.e., *Electronic Codebook*) already gives the idea that it can be considered as a simple block-by-block substitution, according to a code or dictionary (with many entries, of course) that is given by the key.



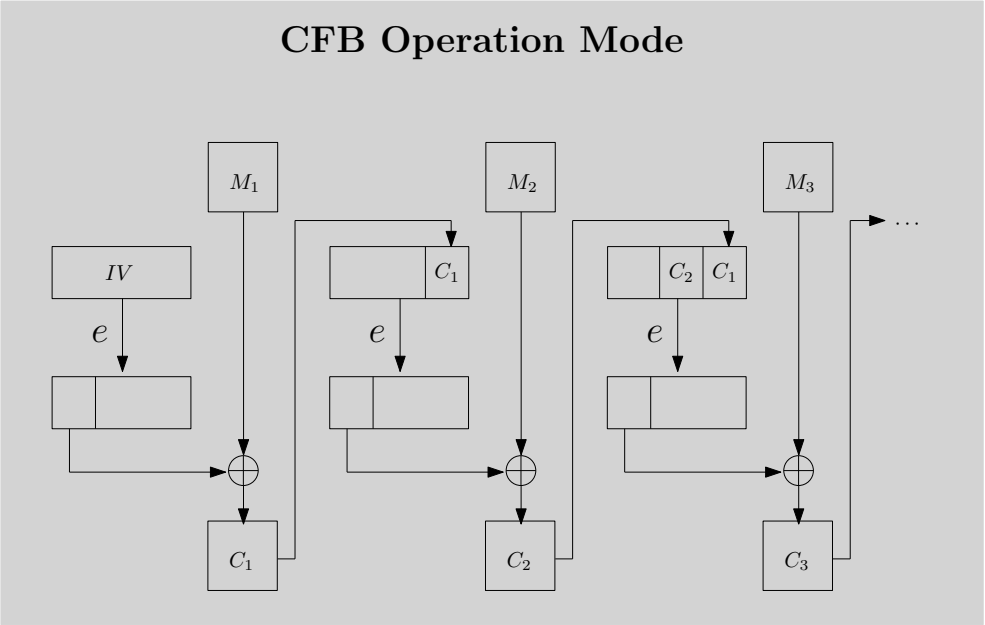
- Another option is to use the CBC (*Cipher Block Chaining*) mode, in which before encrypting a plaintext block, the previous encrypted block is added (bit by bit, with an XOR). An **initialization vector (IV)** is added to the first block, which is a set of random bits of the same length as a block. By choosing different vectors each time, even if the plaintext is the same, the encrypted data will be different. The receiver must know the value of the vector before starting to decrypt, but it is not necessary to keep this value secret; it is normally transmitted as the header of the ciphertext. The idea is depicted below:



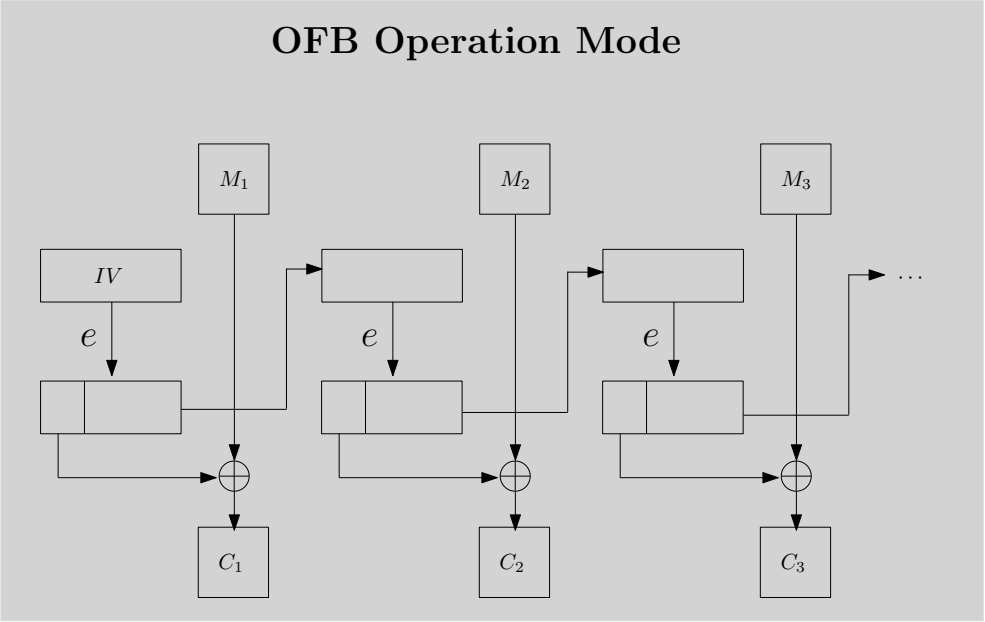
- In CFB (*Cipher Feedback*) mode, the encryption algorithm is not applied directly to the plaintext but to an auxiliary vector (initially equal to the VI). From the encryption result, n bits are taken and added to n bits of the plaintext to obtain n bits of ciphertext. These encrypted bits are also used to update the auxiliary vector. The number n of bits generated in each iteration can be less than or equal to the block length b. Taking, for example $n = 8$, we have a cipher that generates one byte at a time without having to wait for a whole block to be able to decrypt it, as shown in the next figure:

CFB mode as stream cipher

A block cipher in CFB mode (and also in OFB mode) can be used as the generator function of a stream cipher.



- The OFB (*Output Feedback*) mode is like CFB, but instead of updating the auxiliary vector with the ciphertext, it is updated with the result obtained from the encryption algorithm. The property that differentiates this mode from the others is the fact that an error in the recovery of an encrypted bit affects only the decryption of that bit. The idea is shown below:



- From the above modes, several variants can be defined. For example, the CTR (*Counter*) mode is like the OFB mode, but the auxiliary vector is not fed back with the previous encryption; it is simply a counter that keeps incrementing.

There is another technique for preventing identical input texts from producing identical ciphertexts, which can also be applied to ciphers that do not use an initialization vector (including stream ciphers). This technique involves mod-

ifying the secret key with random bits before using it in the encryption (or decryption) algorithm. Since these random bits serve to give the key a different *flavor*, they are often called **salt bits**. Like the initialization vector, the salt bits are sent in the clear before the ciphertext.

1.1.4. Hash functions

Cryptography is more than just encrypting data. It can also provide techniques that are used to guarantee the authenticity of messages. A well-known technique is the use of secure hash functions (also known as message digest functions).

In general, we can say that a hash function allows us to obtain a relatively short, fixed-length bit string from a message of arbitrary length:

$$H = h(M)$$

For messages M of equal length, a hash function h provides the same same digest H summary. However, when two messages give the same exact result H, they do not necessarily have to be equal. This is because there is only a limited set of possible values H, since their length is fixed, and there can be many more messages (whose length can be arbitrarily long, of any length, hence there will be infinitely many messages providing the same H).

To be applicable in an authentication system, the h function must be a secure hash function.

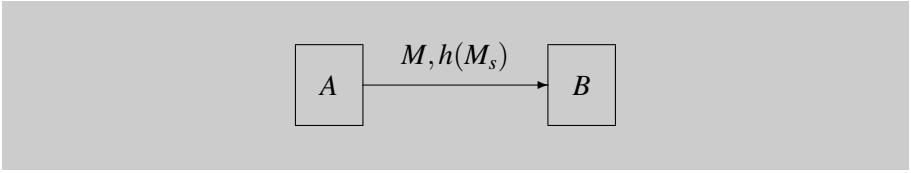
Algorithm of a hash

Note that the algorithm of a hash function needs to be public (i.e., known by all parties), since everyone should be able to compute the hash in the same way.

A hash function is considered to be secure if it meets the following conditions:

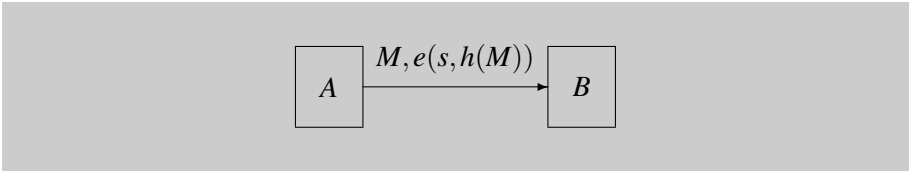
- It is **one-way**, i.e., if we have $H = h(M)$ it is computationally infeasible to find M from the summary H.
- It is **collision-resistant**, that is, given any message M it is computationally infeasible to find a message $M' \neq M$ such that $h(M') = h(M)$.

These properties allow the use of secure hash functions to provide an authenticity service based on a secret key s shared between two parties A and B. Taking advantage of unidirectionality, when A wants to send a message M to B, they can prepare another message M_s, for instance, by concatenating the original with the key: $M_s = (M, s)$. They can then send B the message M and the digest of the message M_s as follows:



To verify the authenticity of the received message, B verifies that the digest actually corresponds to M_s. If so, it means that it was generated by someone who knows the secret key s (which would have to be A), and also that no one has modified the message.

Another technique would be to calculate the message digest M and encrypt it using s as the encryption key:



To verify authenticity, the digest must be sent and retrieved, then decrypted with the secret key s, and compared with the original message M. An adversary who wants to modify the message without knowing the key could try to replace it with another message that gives the very same digest, in a way that B does not detect the forgery. However, if the hash digest is collision-resistant, this would be impossible for the adversary.

Authenticity and confidentiality

Encrypting only the digest, rather than the entire message, is more efficient because there are fewer bits to encrypt. This, of course, assumes that only authenticity is required, not confidentiality. If the message also requires confidentiality, then it is necessary to encrypt the entire message.

To make attacks against hash functions difficult, algorithms must define a complex relationship between the input bits and each output bit. Brute-force attacks are countered by making the digest length sufficiently long. For example, currently used algorithms generate 128- or 160-bit digests. This means that an adversary might have to try on the order of 2^{128} or 2^{160} input messages to find a collision (i.e., a different message that gives the same digest).

Strong collision resistance

The collision resistance of digest algorithms, as we have defined above, is sometimes called *weak resistance*, while the property of being resistant to birthday attacks is known as *strong resistance*. Such an attack assumes that the adversary can choose the message to be authenticated. The victim reads the message and, if they agree, authenticates it with their secret key. But the adversary presented this message because they found another one that gives the same digest, and therefore they can trick the recipient into believing that this other one is the authentic one. And this can be achieved by performing a brute-force search with far fewer operations: on the order of 2^{64} or 2^{80}, if the digest is 128 or 160 bits, respectively.

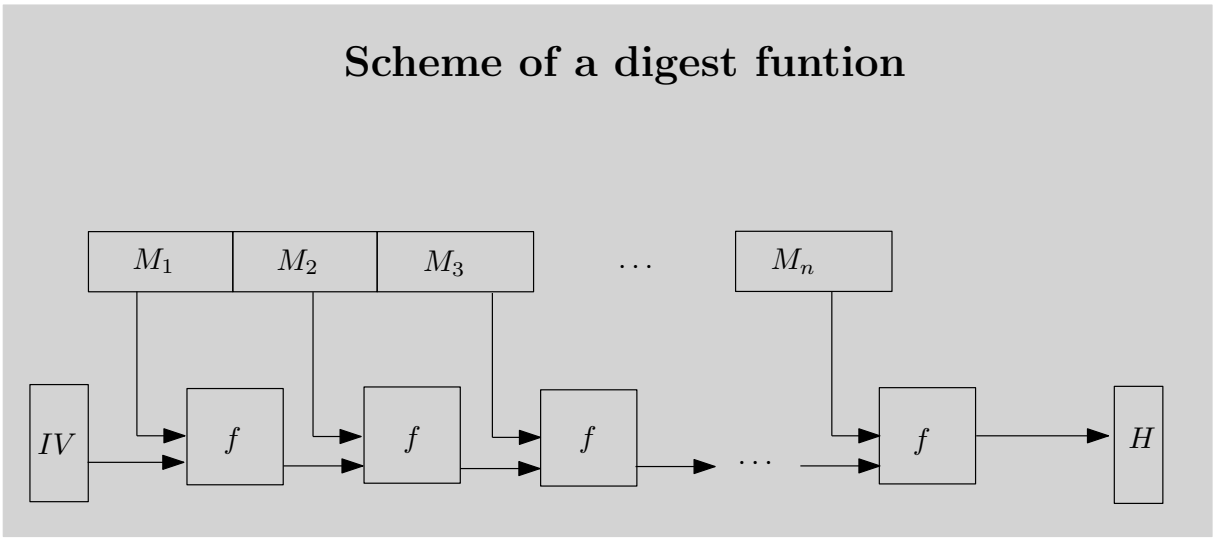
Birthday Attack

Another type of attack that is more advantageous for the adversary is the **birthday attack**. The name of this type of attack comes from a classic probability problem known as the *birthday paradox*. The problem consists of finding the minimum number of people in a group so that the probability that at least two of them celebrate their birthday on the same day is greater than 50%. An intuitive response might be that the solution is on the order of 200, but this result is not correct. It might be correct if one wanted to obtain the names of people necessary to obtain a 50% probability of matching *a* given person. If we allow the match to be between *any* pair of people, the solution is a much smaller name: 23.

The conclusion is that if a hash function can give N different values, because the probability of finding two messages with the same digest is 50%, the number of messages that need to be tested is of the order of $\sqrt{N}$.

Examples of secure hash functions

The scheme of most currently used hash functions is similar to that of block cipher algorithms: the input message is divided into blocks of equal length, and each block is subjected to a series of operations along with the result obtained from the previous block. The result left after processing the last block is the message digest, see below:



The goal of these algorithms is for each bit of the output to depend on all the bits of the input. This is achieved through different iterations of operations that *shuffle* the bits together, similar to how the succession of transpositions in block ciphers causes an *avalanche effect* that ensures bit diffusion.

MD5 vs. SHA hash functions In the early 90s, a frequently used hash algorithm was MD5 (Message Digest 5). However, because this hash function has only 128 bits, and because it is known to generate partial collisions for this algorithm, NIST recommended using other, more secure algorithms, such as SHA-1 (Secure Hash Algorithm-1). The SHA-1 algorithm, published in 1995 as a NIST standard (as a revision of an earlier algorithm simply called SHA), produces 160-bit hash rates. In 2002, NIST published variants of the SHA-1 algorithm that produce 256-, 384-, and 512-bit hash outputs.

MD5 digest length

Since the MD5 hash length is 128 bits, the number of operations for an anniversary attack is on the order of 2^{64}. Compare this to a brute-force attack against DES (less than 2^{56} operations), which is not too far behind.

1.2. Public key cryptography

1.2.1. Public key algorithms

As we saw in the previous subsection, one of the problems with symmetric-key cryptography is key distribution. This problem can be solved by using public-key algorithms, also called asymmetric-key algorithms.

In a public-key cryptographic algorithm, different keys are used for encryption and decryption. One of them, the public key, can be easily obtained from the other, the private key, but, conversely, obtaining the private key from the public key is computationally very difficult.

Public-key algorithms typically allow encryption with the public key (k_{pub}) and decryption with the private key (k_{pr}):

$$\begin{aligned} C &= e(k_{\text{pub}}, M) \\ M &= d(k_{\text{pr}}, C) \end{aligned}$$

There may also be algorithms that allow encryption with the private key and decryption with the public key (this property will be explained in detail in the sequel):

$$\begin{aligned} C &= e(k_{\text{pr}}, M) \\ M &= d(k_{\text{pub}}, C) \end{aligned}$$

Public-key algorithms are based on mathematical problems that are *easy* to pose from the solution, but *hard* to solve. In this context, a problem is considered easy if the solution time, as a function of the data length n, can be expressed in polynomial form, such as $n^2 + 2n$ (in complexity theory, these problems are said to be of *class P'*). If the solution time grows more rapidly, such as with 2^n, the problem is considered hard. Thus, a value of n can be chosen such that the approach is viable but the solution is computationally intractable.

Adaptation to difficult problems

If technological advances reduce the solution time, the length n can be increased, which will require a few more operations for the approach, but the complexity of the solution will grow exponentially.

An example of a problem that is easy to pose but difficult to solve is that of discrete logarithms. If we work with arithmetic modulo m, it is easy to calculate this expression:

$$y = b^x \bmod m$$

The value x is called the discrete logarithm of y to base b modulo m. Choosing b and m conveniently can make it difficult to calculate the discrete logarithm of any y. One possibility is to try all the values of x: if m is an n-bit

number, the time to find the solution increases proportionally to 2^n. There are other, more efficient methods for calculating discrete logarithms, but the best known algorithm also takes longer than can be expressed polynomially.

Example of operations modulo *m*

For example, to obtain $14^{11} \bmod 19$ we can multiply 11 times the number 14, divide the result by 19 and take the remainder of the division, which is equal to 13. But we can also take advantage of the fact that the exponent 11 is 1011 in binary ($11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$), and then $14^{11} = 14^8 \cdot 14^2 \cdot 14^1$, to obtain the result with fewer multiplications:

$$
\begin{array}{lrcrcll}
14^1 = 14 & & & & \equiv\ 14 & \pmod{19} & \rightarrow\ 14 \\
14^2 = 14^1 \times 14^1 & \equiv & 14 \times 14 & = 196 & \equiv\ 6 & \pmod{19} & \rightarrow\ 6 \\
14^4 = 14^2 \times 14^2 & \equiv & 6 \times 6 & = 36 & \equiv\ 17 & \pmod{19} & \\
14^8 = 14^4 \times 14^4 & \equiv & 17 \times 17 & = 289 & \equiv\ 4 & \pmod{19} & \rightarrow\ 4 \\
 & & & & & & \overline{336} \equiv 13 \pmod{19}
\end{array}
$$

Thus, we know that $\log_{14} 13 = 11 \pmod{19}$. But if we had calculated the logarithm of any other number y, we would have to try the exponents one by one until we found one that results in y. And if instead of dealing with 4- or 5-bit numbers like these, they were numbers with more than 1000 bits, the problem would be intractable.

Performance of asymmetric cryptography

Encryption and decryption performance of public-key algorithms is assumed to be lower than encryption and decryption performance of symmetric cryptography.

Public-key cryptosystems must ensure that it is impossible to calculate the private key from the public key. It must also be impossible to reverse them without knowing the private key. Yet, encryption and decryption must be able to be performed in a relatively short time. In practice, the algorithms used in asymmetric cryptosystems comply such constraints, but they are all considerably slower than their symmetric cryptography counterparts. Therefore, public-key cryptography is typically used only to address problems that symmetric cryptography cannot solve, e.g., mostly to handle key distribution and non-repudiation authentication schemes (i.e., digital signatures). More precisely:

- **Key distribution mechanisms** allow two parties to agree on the symmetric keys they will use to communicate, without a third party listening in on the dialogue being able to deduce what these keys are.

  For instance, A can choose a symmetric key k, encrypt it with the public key of B, and send the result to B. Then B will decrypt the received value with its private key, and will know which key k was chosen by A. The rest of the communication will be encrypted with a (much faster) symmetric algorithm, using this key k. Adversaries, not knowing B's private key, shall not be able to deduce the value of k.

- Public-key **authentication** can be used if the algorithm allows the keys to be used in reverse: the private key for encryption and the public key for decryption. If *A* sends a message encrypted with his private key, anyone can decrypt it with *A*'s public key, and at the same time everyone will know that the message could only have been generated by someone who knows the associated private key (which would be *A*). This is the basis of **digital signatures**.

Sample public key algorithms

Diffie-Hellman key exchange. It consists of a series of mechanisms proposed by Whitfield Diffie and Martin Hellman in 1976. Two participants choose their private keys a and b and make public the values α^a and α^b in modulo-n arithmetic. The two can compute $\alpha^{a \cdot b}$ by exponentiating their private key with the other's public key, and they can use this value as their secret key between them. Because of the difficulty of computing discrete logarithms, no one else can deduce what this value is.

RSA. One of most widely public-key algorithms, its name comes from the initials of its designers in 1977: Ronald Rivest, Adi Shamir, and Leonard Adleman. The public key consists of a number n, computed as the product of two very large prime factors ($n = p \cdot q$), and an exponent e. The private key is another exponent d, computed from p, q, and e, such that encryption and decryption can be performed as follows:

$$\begin{aligned} \text{Encryption:} \quad C &= M^e \bmod n \\ \text{Decryption:} \quad M &= C^d \bmod n \end{aligned}$$

As you can see above, the public and private keys are interchangeable: if either is used for encryption, the other must be used for decryption.

The strength of RSA is based on the difficulty of obtaining M from C without knowing d (discrete logarithm problem). At the same time, the difficulty of obtaining p and q (and therefore d) from n (i.e., by factoring large numbers), is considered to be a hard computational problem.

Values used in RSA

The problem of factoring 1024-bit numbers is complex, but manageable if sufficient resources are available. Therefore, it is recommended to use public keys with a value of n from 2048 bits onwards. Simple values such as 3 or 65537 ($2^{16}+1$) are typically used as the public exponent e because they make encryption faster.

ElGamal. Another encryption scheme, in this case based on the Diffie-Hellman problem, was initially proposed by Taher ElGamal in 1985. The public key is obtained from the private key by exponentiating it (α^a). Encryption is performed by choosing a random number r (which is not sent directly, but hidden in α^r) and multiplying the message by $\alpha^{a \cdot r}$ (that is, by the public key raised to r). The owner of the private key can computed as $\alpha^{a \cdot r}$ from α^r and divide to recover the original message.

DSA (Digital Signature Algorithm). Published by NIST in various versions of the *Digital Signature Standard* (DSS), the first of which was in 1991. It is a variant of the ElGamal signature algorithm, which in turn follows the ElGamal encryption scheme. DSA is not an encryption algorithm; it only allows signatures to be generated and verified.

ECC (Elliptic-Curve Cryptography). While previous cryptosystems basically work with products of very large numbers in modular arithmetic, ECC is another branch of public key cryptosystems using operations defined on elliptic curve points in a two-dimensional integer coordinate space. The advantage of this technique is that it provides security equivalent to that of other cyrptosystems like RSA and ElGamal but with fewer key lengths, hence improving performance over them.

Security with elliptic-curve algorithms

The security provided by a 1024-bit RSA is assumed to be equivalent with less than 170-bit keys when using elliptic curve cryptography.

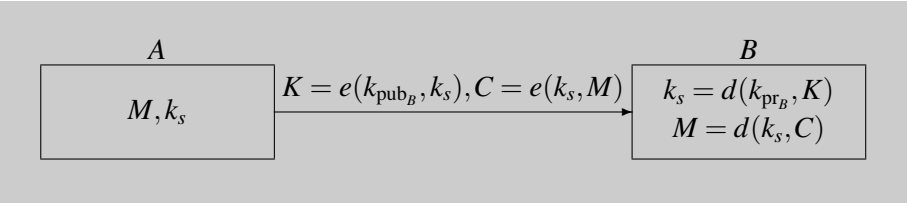
1.2.2. Using public key cryptography to secure communications

We have seen previously that the main applications of public key cryptography are key exchange to provide confidentiality, as well as digital signatures to provide authenticity and non-repudiation. More precisely:

Digital envelope

As we will see later, this technique for providing confidentiality with public key cryptography is often called *digital envelope* in the context of secure e-mail.

- The confidentiality problem between two parties who only have an insecure channel to communicate is solved by public-key cryptography. When A wants to send a secret message M to B, there is no need to encrypt the entire message with a public-key algorithm (this could be very slow), but a symmetric key k_s, sometimes called a **session key** or **transport key**, is chosen and the message is encrypted with a symmetric algorithm using this key. The only thing that needs to be encrypted with B's public key (k_{pub_B}) is the session key. Upon receiving, B uses its private key (k_{pr_B}) to recover the session key k_s, and can then decrypt the encrypted message.

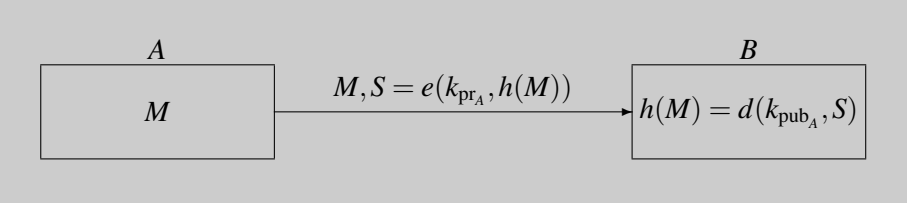


Since the session key is a relatively short message (for example, if it is a DES key, it will only have 56 bits), an attacker could attempt to break the encryption by brute force, but not by trying to decrypt the message with the possible values of the private key, k_{pr_B}, but rather by encrypting the possible values of the session key, k_s with the public key, k_{pub_B}. In the case of a DES session key, regardless of the number of bits in the public key, the attacker would only need an effort of about 2^{56} operations.

To prevent this type of attack, the information actually encrypted with the public key is not the secret value itself (in this case, k_s), but rather a string of random bits is added to this value. The receiver simply discards these random bits from the decryption result.

- A **digital signature** is basically a message encrypted with the signer's private key. However, for efficiency reasons, what is encrypted is not the message to be signed, but only its digest calculated with a secure hash function.

  Assume A wants to send a signed message. First, A computes a digest of the message and encrypts it with the private key k_{pr_A}. To verify the signature, the recipient must decrypt it with the public key k_{pub_A} and compare the result with the message digest: if they are equal, it means that the message was generated by A and no one has modified it. Since the digest function is assumed to be collision-resistant, an attacker will not be able to modify the message without rendering the signature invalid. The process is summarized in the following figure:



1.3. Public Key Infrastructure (PKI)

We have seen that public-key cryptography allows us to solve the key exchange problem by using the public keys of the participants. However, it raises a new problem: if an entity claims to be A and their public key is announced to be k_{pub}, how can one know that k_{pub} is really the public key of A? How can we assure that an adversary Z, who previously generated a key pair (k'_{pr}, k'_{pub}) is not impersonating A and illegitimately releasing k'_{pub} as k_{pub}?

A possible solution to this problem is to have a trusted entity that can assure us that the public keys effectively belong to their supposed owners. This entity can sign a document stating that *the public key of A is* k_{pub_A} and publish it publicly in a way that all potential users in the system get aware about it. This type of document is called a **public key certificate** or **digital certificate** and is the basis of what is known as a **public key infrastructure** or PKI.

1.3.1. Public key certificates

A public key certificate or digital certificate consists of three basic parts:

- A user ID, such as the user's name.
- The content of this user's public key.
- A digital signature of the two previous elements.

If the author of the signature is someone we trust, the certificate serves as a guarantee that the public key belongs to the user identified in the certificate. The person signing the certificate may be an authority responsible for reliably verifying the authenticity of the public keys. In this case, the certificate is said to have been generated by a Certification Authority (CA).

The X.500 directory

The X.500 directory specification is published in the ITU-T X.500 Series of Recommendations, one of which is Recommendation X.509.

There may be different certificate formats, but the most commonly used is the **X.509 certificates**, specified in the **X.500 directory service** definition. The X.500 directory allows the storage and retrieval of information, expressed as attributes, about a set of objects. X.500 objects can represent, for example, countries, cities, or companies, universities (generally organizations), departments, faculties (generally organizational units), people, etc. All these objects are organized hierarchically in the form of a tree (each node in the tree contains an object), and within each level, the objects are identified by a distinctive attribute. At a global level, each object is identified by a Distinguished Name (DN), which is nothing more than the concatenation of the distinctive attributes between the root of the tree and the object in question.

The naming system is, therefore, similar to the Internet's DNS, except that the components of a DNS name are simple character strings, while those of an X.500 DN are attributes, each with a type and a value.

Some examples of attribute types that can be used as distinguishing attributes in a DN are: *countryName* (usually denoted with the "`c`" abbreviation), *stateOrProvinceName* ("`st`"), *localityName* ("`l`"), *organizationName* ("`o`"), *organizationalUnitName* ("`ou`"), *commonName* ("`cn`"), *surname* ("`sn`"), etc.

The X.500 directory defines as well a service access protocol that allows query operations and also modifications to object information. These latter operations are normally only permitted to certain authorized users, and therefore require user authentication mechanisms. These mechanisms are defined in Recommendation X.509. There is a basic mechanism that uses passwords, and a more advanced mechanism that uses certificates.

Next you can see the structure of an X.509 certificate, including its fields and subfields. In the notation used here, “rep.” means that the field can be repeated one or more times, and “opt.” means that the field is optional (and therefore “opt. rep.” means that the field can be repeated zero or more times).

Field	**Type**
`toBeSigned`	
`version` (opt.)	integer
`serialNumber`	integer
`signature`	
`algorithm`	unique identifier
`parameters`	(algorithm dependant)
`issuer`	DN
`validity`	

[……]

2. The SSH protocol

SSH is an application designed to replace certain remote access tools traditionally used on Unix systems, such as `rsh` (*Remote Shell*), `rlogin` (*Remote Login*), or `rcp` (*Remote Copy*), with new versions with security services.

The name of this application, SSH, is short for *Secure Shell*, which means "secure version of the *Remote Shell* program."

The application defines its own protocol for secure data transmission, the **SSH** protocol, which provides some security services similar to those of the SSL/TLS protocol. In addition to establishing secure connections, the SSH protocol also offers other features, such as TCP port forwarding and communication between clients and servers over an SSH connection.

The first version of the SSH protocol was defined by Tatu Ylónen, from the Helsinki University of Technology, in 1995. Since then, several improvements were conducted on the second version of the protocol, which incorporates many improvements and is substantially different from the first version. The first and second version of the protocol are commonly referred to as SSH1 and SSH2, respectively. In this section we focus primarily on the SSH2 protocol.

2.1. Features of the SSH protocol

SSH provides security services equivalent to those of the SSL/TLS protocols.

Confidentiality. SSH is used to communicate data, which typically consists of the input of a remote application and the output it generates, or information transmitted through a forwarded port. The confidentiality of this data is guaranteed through encryption.

As with the SSL/TLS protocol, SSH applies symmetric encryption to the data, and therefore requires a secure key exchange between the client and server. One difference from SSL/TLS is that SSH2 can use different encryption algorithms in both directions of communication.

An additional service provided by SSH is the confidentiality of the user's identity. While in SSL 3.0 and TLS 1.0, if you choose to authenticate the client, the client must send its certificate in clear text, in SSH (and also in early versions of the SSL protocol, like SSL 2.0) user authentication is performed when the packets are already sent encrypted.

On the other hand, SSH2 also allows you to hide certain traffic characteristics, such as the actual length of the packets.

Entity Authentication. The SSH protocol provides mechanisms to authenticate both the server computer and the connecting user. Server authentication is usually performed in conjunction with the key exchange. In SSH2, the key exchange method is negotiated between the client and the server, although currently only one is defined, based on the Diffie-Hellman algorithm.

There are different methods to authenticate the user; depending on the method used, client computer authentication may also be required, while other methods allow the properly authenticated user to access the server from any client computer.

Message Authentication. As with SSL/TLS, in SSH2, data authenticity is guaranteed by adding a MAC code calculated with a secret key to each packet. It is also possible to use different MAC algorithms in each direction of communication.

Unofficial Algorithms

Names for unofficial algorithms must be of the form "*name*`@`*domain*", where *domain* is a DNS domain controlled by the organization that defines the algorithm (for example "`cypher@example.com`").

Like SSL/TLS, SSH is also designed with the following additional criteria:

Efficiency: SSH includes compression of the data exchanged to reduce packet length. SSH2 allows you to negotiate the algorithm to use for each direction of communication, although only one is defined in the protocol specification. This algorithm is compatible with the one used by programs such as `gzip` (RFC 1950-1952).

Unlike SSL/TLS, SSH does not provide for the reuse of keys from previous sessions: the keys are recalculated for each new connection. This is because SSH is designed for connections that last more or less a long time, such as interactive work sessions with a remote computer, and not for the short but consecutive connections that are more typical of the HTTP application protocol (which was initially intended to be protected with SSL). However, SSH2 defines mechanisms to try to shorten the negotiation process.

Extensibility: SSH2 also negotiates encryption, user authentication, MAC address, compression, and key exchange algorithms. Each algorithm is identified by a string representing its name. Names can correspond to officially registered algorithms, or to experimentally proposed or locally defined algorithms.

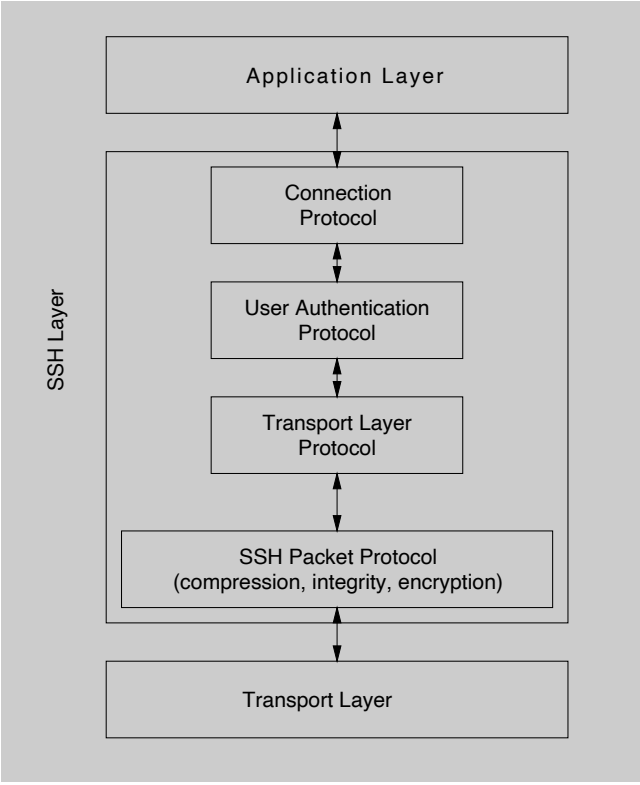
Adapting SSH to local languages

SSH2 facilitates the adaptation of implementations to local languages. Where the protocol provides for the transmission of an error or informational message that can be displayed to the human user, a tag identifying the language of the message is included, in accordance with RFC 1766.

Both these messages and usernames are represented using the ISO/IEC 10646 universal character set using UTF-8 encoding, in accordance with RFC 2279 (the ASCII code is a subset because in UTF-8, characters with a code less than 128 are represented as a single byte, equal in value to the code).

2.2. The SSH transport layer

As SSL/TLS, SSH distinguishes two sublayers in the secure transport layer. In addition, in SSH2, the upper layer is structured into three protocols, one above the other, as shown in the following figure.



At the bottom level of the SSH layer is the **SSH packet protocol**. The three protocols above this are:

- The **transport layer protocol**, which handles key exchange;
- The **user authentication protocol**;
- The **connection management protocol**.

2.2.1. The SSH packet protocol

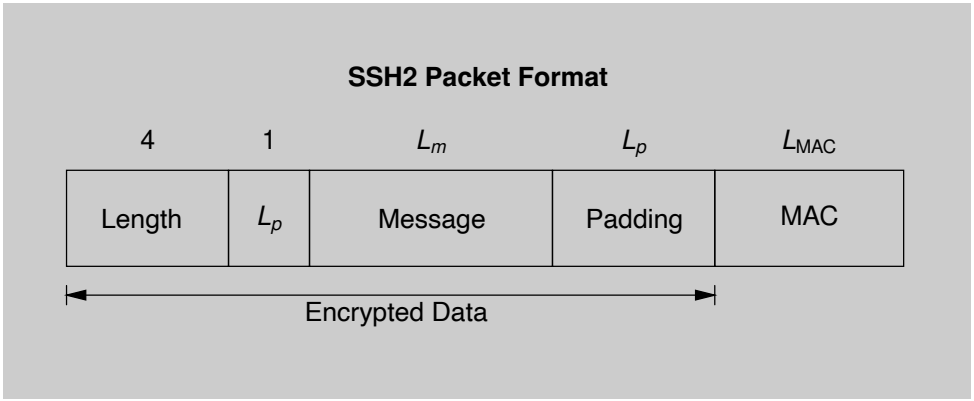
The SSH packet protocol is responsible for building and exchanging protocol units, which are the **SSH** packets.

When sending data, the following are applied to messages at higher levels (in this order):

- Compression;
- MAC authentication codes;
- Encryption.

Upon reception, each packet undergoes reverse processing (decryption, authenticity verification, and decompression).

The format of SSH2 packets is as follows:



These are the fields in an SSH2 packet:

- The first is the length of the rest of the packet, excluding the MAC address (therefore, it is equal to $1 + L_m + L_p$).
- The second field indicates how many bytes of *padding* there are. This number of bytes must be such that the total length of the packet, excluding the MAC address, is a multiple of 8 (or the block length in block ciphers, if it is larger than 8).
- The third field is the content of the message, compressed if necessary. The first byte of the content always indicates what type of message it is, and the structure of the remaining bytes depends on the type.

- The fourth field is the random bytes of *padding*. They are always present, even when using a stream cipher, and their length must be at least 4. Therefore, the minimum length of a packet, not including the MAC, is 16 bytes.
- The fifth field is the MAC authentication code, obtained using the HMAC technique from a secret key, an implicit 32-bit sequence number, and the value of the other four fields in the packet. The length of the MAC depends on the agreed algorithm, and can be 0 if the null algorithm is used.

Padding bytes

The padding of bytes ensures that the length of the data to be encrypted is appropriate for block ciphers.

When packets are encrypted, encryption is applied to all fields except the MAC address, but including the length. This means that the receiver must decrypt the first 8 bytes of each packet to determine the full length of the encrypted portion.

2.2.2. The SSH transport layer protocol

The SSH transport layer protocol is responsible for establishing the transport connection, server authentication and key exchange, and service requests from other protocols.

The client connects to the server using the TCP@ protocol. The server must be listening for connection requests on the port assigned to the SSH service, which by default is port number 22.

Version compatibility 1

SSH2 defines a compatibility mode with SSH1, in which the server identifies its version with the number 1.99: SSH2 clients should consider this number to be equivalent to 2.0, while SSH1 clients will respond with their actual version number.

Once the connection is established, the first step is to negotiate the version of the SSH protocol to be used. Both the client and the server send a line containing the text "`SSH-`*`x.y-implementation`*", where *`x.y`* is the protocol version number (for example, `2.0`) and *`implementation`* is a string identifying the client or server software. If the version numbers do not match, the server decides whether or not to continue: if not, it simply closes the connection. Before this line of text, the server may also send other lines with informational messages, as long as they do not begin with "`SSH-`".

Once they have agreed on the version, the client and server begin exchanging messages using the SSH packet protocol discussed above, initially unencrypted and without a MAC address. To save time, the first SSH packet can be sent along with the line indicating the version, without waiting to receive the line from the other party. If the versions match, the protocol continues normally; if not, it may need to be restarted.

First, the key exchange takes place. In SSH2, each party sends a KEXINIT message containing a 16-byte random string called a cookie and lists the supported algorithms in order of preference: key exchange algorithms and, for each direction of communication, symmetric encryption, MAC, and compression algorithms. A list of supported languages is also included for informational messages. For each algorithm type, the first one from the client list that is also on the server list is chosen.

Cryptographic algorithms in SSH2

The cryptographic algorithms supported by SSH2 are, at least, the following ones:

- For key exchange: Diffie-Hellman;
- For encryption: RC4, Triple DES, Blowfish, Twofish, IDEA, and CAST-128;
- For MAC@ codes: HMAC-SHA1 and HMAC-MD5 (likely to be deprecated soon);

The following packets are key exchange packets, and depend on the chosen algorithm (although SSH2 only supports the Diffie-Hellman algorithm).

It can be assumed that most implementations will have the same preferred algorithm for each type. Thus, to reduce response time, the first key exchange message can be sent after the KEXINIT without waiting for the other party's message, using these preferred algorithms. If the assumption is correct, the key exchange continues normally; if not, the packets sent early are ignored and resent using the correct algorithms.

Regardless of the algorithm, the key exchange results in a shared secret and a session identifier. With the Diffie-Hellman algorithm, this identifier is the hash of a string consisting of, among other things, the client and server's cookies. The encryption and MAC keys and initialization vectors are calculated by applying hash functions in various forms to different combinations of the shared secret and the session identifier.

To complete the key exchange, each party sends a NEWKEYS message, indicating that the next packet will be the first to use the new algorithms and keys.

This entire process can be repeated as needed to regenerate the keys. The SSH2 specification recommends doing this after every gigabyte transferred or every hour of connection time.

If an error occurs during the key exchange, or at any other point in the protocol, a DISCONNECT message is generated, which may contain explanatory text about the error, and the connection is closed.

Other messages that can be exchanged at any time include:

- `IGNORE`: Its contents should be ignored, but it can be used to counteract traffic flow analysis;
- `DEBUG`: They are used to send informative messages;
- `UNIMPLEMENTED`: They are sent in response to messages of unknown type.

In SSH2, after the key exchange is complete, the client sends a request of type `SERVICE_REQUEST` message to request a service, which can be user authentication or direct access to the connection protocol if authentication is not required. The server responds with `SERVICE_ACCEPT` if it allows access to the requested service, or with `DISCONNECT` otherwise.

2.2.3. The user authentication protocol

In SSH, the following user authentication methods are contemplated:

1) **Null Authentication.** The server allows the user to directly access the requested service without any verification. An example would be accessing an anonymous service.
2) **Access list-based authentication.** Based on the address of the client and the name of the user requesting access, the server may consult a list to determine if the user is authorized to access the service. This mode mimics early user authentication methods used in `rsh`, in which server may grant access by consulting the `.rhosts` and `/etc/hosts.equiv` files. This method is highly vulnerable to IP address spoofing attacks. For this reason, we may find it implemented in early version of the protocol (e.g., in SSH1, but not in SSH2).
3) **Access list-based authentication with client authentication.** Same as above, but the server verifies that the client system is indeed who it claims to be, to prevent address spoofing attacks.
4) **Password-based authentication.** The server grants access if the user provides a correct password (similar to the `login` process followed on Unix systems).
5) **Public key-based authentication.** Instead of providing a password, the user authenticates by proving that they possess the private key corresponding to a public key recognized by the server.

In SSH2, the client sends USERAUTH_REQUEST messages, which include the username (which can be changed from message to message), the requested authentication method, and the service to which access is desired. If the server allows access, it will respond with a USERAUTH_SUCCESS message; if not, it will send a USERAUTH_FAILURE message, which contains the list of authentication methods that can be attempted further, or the connection will be closed if there have been too many attempts or too much time has passed. The server may optionally send an informational message USERAUTH_BANNER before authentication.

USERAUTH_BANNER

The USERAUTH_BANNER message may include, for example, text identifying the server system, notices about usage restrictions, etc. This is the type of information that Unix systems typically display before the username prompt, and is usually found in the /etc/issue file.

Authentication request messages contain the following information depending on the method:

1) No additional information is required for null authentication.

2) For access list-based authentication (only applicable to SSH1), the username on the client system must be provided. The address of this system is assumed to be available through the underlying protocols (TCP/IP).

3) When using access lists with client authentication, in SSH2 the client sends its fully qualified DNS name, the local username, the client system's public key (and certificates, if any), the signature of a byte string that includes the session ID, and the algorithm with which this signature was generated. The server must validate the public key and signature to complete authentication.

4) Password authentication requires only the password to be sent directly. For obvious reasons, this method should not be allowed if the SSH transport sublayer protocol is using the null encryption algorithm.

5) Public key-based authentication is similar to access lists with client authentication. In SSH2, the client must send a message containing the user's public key (and any certificates, if any), the algorithm corresponding to this key, and a signature that includes the session ID. The server will accept the authentication if it correctly verifies the key and signature.

   Optionally, to avoid unnecessary processing or interactions with the user, the client can first send a message with the same information but without the signature, so that the server can respond if the offered public key is acceptable.

Password Expired

SSH2 anticipates that the user's password on the server system has expired and needs to be changed before continuing. The change should not be allowed if no MAC (null algorithm) is being used, because an attacker could modify the message containing the new password.

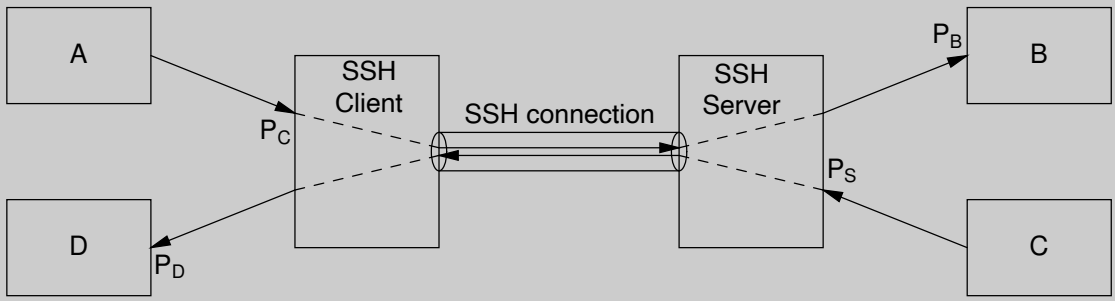
Once the authentication process has been successfully completed, SSH2 switches to the service the client requested in its last USER-AUTH_REQUEST message (the one that resulted in successful authentication).

2.2.4. The connection protocol

The connection protocol manages interactive sessions for remote command execution, sending input data from the client to the server and output data back. It also handles TCP port forwarding.

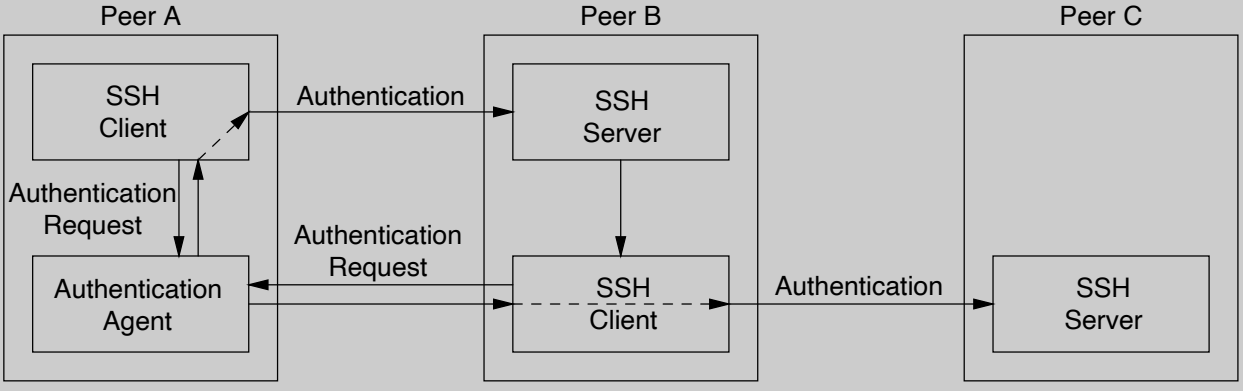
As the following figure shows, with TCP redirection it is possible to have connections made to a specific port P_C on the client redirected to a port P_B on a computer B on the server, or connections made to a specific port P_S on the server redirected to a port P_D on a computer D on the client. This way, the SSH connection can be used, for example, as a tunnel for other connections through a firewall located between the client and the SSH server.

TCP port forwarding with SSH



In addition, SSH provides the option to use what is known as an authentication agent. This agent is a process that automates user authentication based on public keys when it needs to be performed from a remote computer. For example, consider the situation in the following figure:

Using the SSH authentication agent



The user on computer A uses an SSH client to connect to computer B and work in an interactive session. Computer A could be, for example, a laptop where the user has stored their private key and never wants to leave it. They then need to establish an SSH connection (for example, another interactive session) from computer B to computer C, and they need to authenticate with their personal key. The client on computer B, instead of performing authentication directly, which would require the user's private key, asks the agent on computer A to sign the appropriate message to prove that they possess the private key. This scheme can also be used locally by clients on the same computer A.

> Each interactive session, redirected TCP connection, or connection to an agent is a **channel**. Multiple channels can be open on a single SSH connection, each identified by a number on each end (the numbers assigned to a channel on the client and server can be different).

SSH2 provides several types of messages that can be sent during the connection phase. These are some of the functions that allow messages to be sent:

Open a channel. Channels of different types can be opened: session interactive, windows X channel, redirected TCP connection, or connection with an authentication agent.

Configure channel parameters. Before starting an interactive session, the client can specify whether it needs a pseudo-terminal assigned to it on the server, as the Unix `rlogin` program does (the `rsh` program does not, however), and if so, with what parameters (terminal type, dimensions, control characters, etc.). There are other messages to indicate whether a connection to the authentication agent or X-window connection redirection is desired.

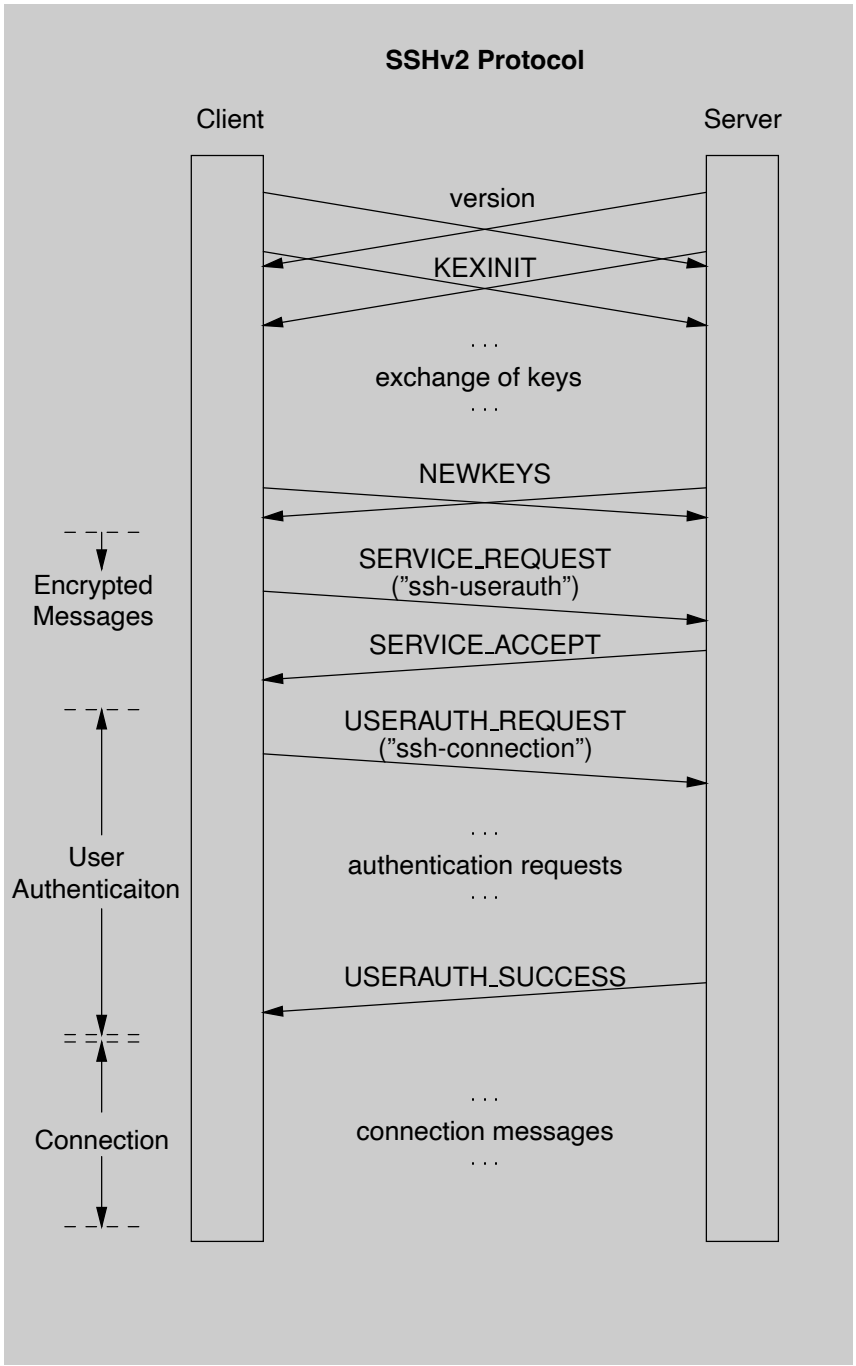
Start interactive session. Once the necessary parameters have been configured, the client can specify the name of a command to be executed on the server (as in `rsh`), or indicate that it wants to run a shell (as in `rlogin`). In addition to a remote process, SSH2 also offers the option of starting a "subsystem," such as a file transfer.

Send data. In SSH2, there are two types of messages for this purpose: one for sending normal data in any direction and for any channel (including interactive sessions), and another for sending special data (for example, the error output `stderr`). In addition to session data, the client can also send a message to indicate that it has received a signal or that a change in the terminal's dimensions has occurred.

Close channel. When the normal execution of the process or shell terminates, the server sends a message indicating the exit code (the numeric value returned by the process). If it terminated due to a signal, in SSH2 it sends a message with the signal number. Other messages are used to indicate that there is no more input data, to request the closure of a channel from one end, and to confirm the closure from the other end.

Other operations (not associated with an open channel). The client can request that connections arriving at a specific TCP port on the server be redirected so that they can be forwarded to another address.

The following figure summarizes the message exchange in SSH2.



2.3. Attacks against the SSH protocol

Many of the protection considerations provided by SSL/TLS also apply to the SSH protocol. The main goal is to guarantee that an attacker cannot read the content of messages or alter them (e.g., sequence changes).

Confidentiality is guaranteed with the use of key exchange methods based on public-key cryptography, mainly to protects against *man-in-the-middle* attacks we saw in the section on SSL/TLS@. In addition, SSH comes with protection methods agains impersonation (e.g., to ensure a client that it is connecting to the legitimate server). SSH verifies the authenticity of public keys using authentication certificates or stored keys in known databases. Clients keep a local database containing the keys of previously recognized servers. Servers also authenticate users using a public key (associated to the client or to a third party used by the client to connect the server, depending on the precise authentication method).

If the use of previously exchanged certificates is not possible, the protocol provides the option (although not recommended) to accept a server's public key the first time a connection is established, without the need for any prior communication. This is not appropriate, since can be exploited by *man-in-the-middle* attacks. As with many other protocols, and whenever a widely deployed key infrastructure is not available, directly accepting keys received for the first time is a compromise between ease of use and security.

An interesting feature in SSH2 is that packet lengths are sent encrypted. An attacker who views the data exchanged as a stream of bytes cannot know where each SSH2 packet begins and ends (if they have access to the TCP packet level, they can try to make inferences, but without complete certainty). This, along with the ability to include arbitrary padding (up to 255 bytes) and send `IGNORE` messages, can be used to hide traffic characteristics and hinder attacks using known cleartext.

SSH2 security improvements

SSH1 is known to be vulnerable to several attacks, including packet replay, packet drop, or packet reordering attacks (due to sequence numbers absence). SSH1 is also vulnerable to packet forwarding manipulation (an attacker could forward packets in the opposite direction if a single encryption key was used for both directions). Most of those problems are no longer present in SSH2, even if other vulnerabilities may affect future versions of SSH2 under certain conditions (e.g., man-in-the-middle interception, followed by sequence numbers manipulation).

On the other hand, it is worth noting that user authentication methods using access lists are based on the server's trust in the client system administrator (just as protocols `rsh` and `rlogin`):

- When the client system is not authenticated (a possibility contemplated only in SSH1), the server only has to accept connections coming from a privileged TCP port (less than 1024) because it is difficult for any user to send packets impersonating another user.
- When the client system is authenticated, the server trusts that users will not have access to the private key of that system, because otherwise they could use it to generate authentication messages with another user's identity.

[. . . . .]