Information Security

Introduction and History

In the past cryptography was the **art** of encrypting messages (mostly for military applications). There was a lack of precise definitions and it was usually insecure. Nowadays it is the **science** of securing digital communication and transactions (not only encryption, but also authentication, digital signatures, e-cash, cryptocurrencies, ...). Now there are formal definitions, there is a systematic design and the constructions are very secure. This transition was possible in the seventies thanks to the following scenario:

- **Technology:** affordable hardware
- **Demand:** companies and individuals started to do business electronically
- **Theory:** computational complexity theory was born (and this allowed researchers to reason about security in a **formal way**)

The following are some kind of arbitrary definitions which are worth mentioning:

- Cryptography: constructing secure systems
- Cryptanalisis: breaking the system
- Cryptology = cryptography + cryptanalisis (often abbreviated *crypto*)

We want to construct schemes that are **provably secure**:

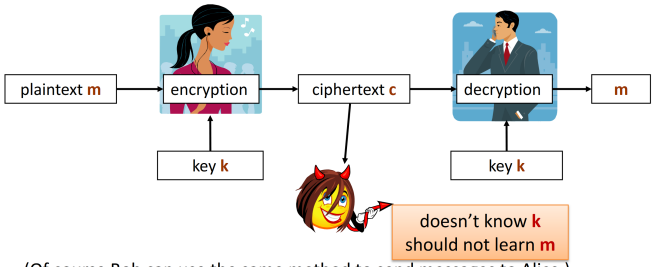
- **Why?** In many areas of computer science formal proofs are not essential. For example, instead of proving that an algorithm is efficient we can simulate it on a *typical* input. In cryptography this is not true because there cannot exist an experimental proof that a scheme is secure. This can not exist because a notion of *typical adversary* does not make sense. Moreover security definitions are useful for modularity purposes.
- **How do we define it?** We use the Kerchoffs principle, which says that since the enemy knows the system, the cipher should remain secure even if the adversary knows the specification of the cipher. The only thing that is **secret** is the key k (that is usually chosen u.a.r from a key space). Not respecting this principle means that we have *security by obscurity*.

Now we are going to see some historical ciphers for the **encryption problem**, i.e. we have a key space $\mathcal{K}$, a plaintext space $\mathcal{M}$ and a ciphertext space $\mathcal{C}$. An encryption scheme is a tuple (Enc, Dec, Gen) where:

- $Enc : \mathcal{K} \times \mathcal{M} \Rightarrow \mathcal{C}$ is an encryption algorithm
- $Dec : \mathcal{K} \times \mathcal{C} \Rightarrow \mathcal{M}$ is a decryption algorithm

and the correctness property is:

$$\forall k \in \mathcal{K} \quad Dec_k\left(Enc_k(m)\right) = m$$



(Of course Bob can use the same method to send messages to Alice.)
(That's why it's called the **symmetric setting**)

Now we are going to see some examples of ciphers for encryption.
Caesar's Shift cipher: just take a letter and shift it by k positions. For example for $k = 3$, a becomes D, b becomes E, y becomes B and so on. Formally we have $\mathcal{K} = \{0, 1, \ldots, 25\}$. In order to break this cipher one can check all possible keys, i.e. try all keys until we find a message that makes sense. This is called **brute force attack**. Moral of the story: the key space must be large. This is a necessary condition, but is not sufficient.
Substitution cipher: we choose a permutation of the letters and we use it for encryption, decryption. This is less easy to break than the shift cipher, but one can use statistical pattern of the language (e.g. vowels are very frequent, not too many consonants in a row, ...) to break it. The key observation is that the ciphertext has the same frequency distribution as the plaintext and one can exploit this.
Vigenere's cipher: instead of using a single shift for the whole message we use different shifts for each letter (periodically). Concretely we write the plaintext and the keyword repeated, then we shift the plaintext letter by cipher key. For example:

```
key:        de cep tivedecept ived eceptive
plaintext:  we are discovered save yourself
ciphertext: ZI CVT WQNGRZGVTW AVZH CQYGLMGJ
```

Although this cipher is more involved, it is not secure. In facts, if the length of the keyword is known, one can perform a frequency attack. In order to find the length of the key there are several techniques, for example one can try different length (sequentially, i.e. 1, 2, 3, ...) and compute individual letter probabilities. If those are similar to the one of the English language, the length is probably determined.
In contemporary cryptography ciphers are designed in a systematic way. The main goals are:

1. Define meaning of security
2. Construct schemes that are provably secure

We begin with the first point. Defining *security of an encryption scheme* is not trivial. Consider the following experiment:

1. The key k is chosen u.a.r. from $\mathcal{K}$
2. $C = Enc_k(m)$ is given to the adversary for a meaningful message m

How do we define security?

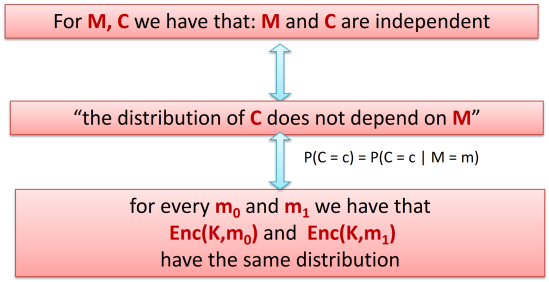
- **Idea 1:** *the adversary should not be able to compute k.* Problem: what if $Enc_k(m) = m$? Also if the adversary can not compute k, the encryption scheme is not secure (it does not encrypt at all).
- **Idea 2:** *the adversary should not be able to compute m.* Problem: what if the adversary can compute the first half of m? This is not what we want.

- **Idea 3:** *the adversary should learn no information about m.* Problem: the adversary may have some a priori information about m (e.g. the language of the text). If the adversary knows only this information the system is still secure.
- **Idea 4:** *the adversary should learn no additional information about m.* **This makes sense, but how do we formalize it?**

Idea 4 is also called **information-theoretically secret** or **unconditionally secret**. We say that an encryption scheme is **perfectly secret** if for random variables M, C and every $m \in \mathcal{M}$, $c \in \mathcal{C}$, we have:

$$P(M = m) = P(M = m|C = c)$$

in other words, the distributions of the variables M and C are independent. Some more equivalent definitions are represented in the following figure:



One-time pad: a perfectly secret scheme This scheme is very simple: given a parameter t, we have $\mathcal{K} = \mathcal{M} = \{0,1\}^t$ (i.e. t is the maximal length of the message in bits and we use a key with the same length of the message). The encryption and decryption work as follows:

- $Enc_k(m) = k \otimes m$
- $Dec_k(m) = k \otimes c$

where $\otimes$ is the bitwise XOR operator. Correctness is trivial because $Dec_k(Enc_k(m)) = k \otimes k \otimes m = m$. Now we have to prove that this scheme is perfectly secret. In order to do that we use the definition, i.e. we calculate:

$$\begin{aligned} Pr[M = m|C = c] &= \frac{\overbrace{Pr[C = c|M = m] \cdot Pr[M = m]}^{1}}{\underbrace{Pr[C = c]}_{2}} \\ &= \frac{2^{-t} \cdot Pr[M = m]}{2^{-t}} \\ &= Pr[M = m] \end{aligned}$$

because:

1. $Pr[C = c|M = m'] = Pr[m' \otimes K = c] = Pr[K = m' \otimes c] = 2^{-t}$
2. $Pr[C = c] = \sum_{m' \in \mathcal{M}} Pr[C = c|M = m'] \cdot Pr[M = m'] = 2^{-t} \cdot \sum_{m' \in \mathcal{M}} Pr[M = m'] = 2^{-t}$

We observe that OTP can be generalized by letting $\mathcal{K} = \mathcal{M} = \mathcal{C} = G$ where $(G, \star)$ is a group and using the operation $\star$ instead than XOR.
This scheme, however, has some drawbacks:

- The key is as long as the message (and this is not practical)
- The parties must know in advance the length of the message
- The key can be used only once, otherwise, given two encrypted messages c and c' one can do the following: $c \otimes c' = (m \otimes k) \otimes (m' \otimes k) = m \otimes m'$. Knowing the bitwise XOR of the two messages is sufficient to perform frequency analysis (as happened in the VENONA project)
- It is hard to generate truly random strings

In general, we have that if an encryption scheme is perfectly secret, then $|\mathcal{K}| \geq |\mathcal{M}|$ holds. In order to show the statement we show that if $|\mathcal{K}| < |\mathcal{M}|$ then the scheme cannot be perfectly secret. Assume $|\mathcal{K}| < |\mathcal{M}|$ and consider the uniform distribution over $\mathcal{M}$ and let $c \in \mathcal{C}$ be a ciphertext that occurs with non-zero probability. Let $\mathcal{M}(c)$ be the set of all possible messages that are possible decryptions of c. We have $|\mathcal{M}(c)| \leq |\mathcal{K}|$. If $|\mathcal{K}| < |\mathcal{M}|$ there is some $m' \in \mathcal{M}$ such that $m' \notin \mathcal{M}(c)$. But then we have:

$$Pr[M = m'|C = c] = 0 \neq Pr[M = m']$$

and the scheme is not perfectly secure.
This is a special case of **Shannon's theorem**, which states:

Let (Gen, Enc, Dec) be an encryption scheme with message space $\mathcal{M}$, for which $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$. The scheme is perfectly secret if and only if:

1. Every key $k \in \mathcal{K}$ is chosen u.a.r. from Gen
2. For every $m \in \mathcal{M}$ and every $c \in \mathcal{C}$, there exists a unique key $k \in \mathcal{K}$ such that $Enc_k(m)$ outputs c

Computational Security

We have seen that perfect security requires that M and $Enc_K(M)$ are independent. This assumption may be too strong for practical purposes, in fact we require that M and $Enc_K(M)$ are independent **from the point of view of a computationally-limited adversary with high probability**. This can be formalized with the help of complexity theory. Concretely we construct schemes that in principle can be broken (e.g. with a brute force attack by iterating over all possible keys) but in order to be broken the adversary needs **huge computing power** and/ or **a lot of luck**. Typically we assume that a scheme X is secure if for all probabilistic polynomial-time Turing Machines M the probability that M breaks the scheme X is negligible (i.e. it decays faster than a polynomial). In order to get a more precise intuition about this concept consider the following game:

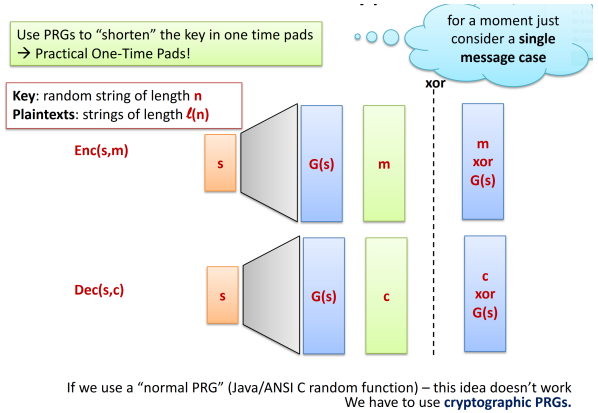
- An adversary E chooses two messages m_1 and m_2 of the same length.
- E gives m_1 and m_2 to an oracle.
- The oracle selects a random message between m_1 and m_2 and encrypts the chosen message with a key k chosen u.a.r.
- The oracle gives the encrypted message back to R.

- E has to guess which of the messages was chosen by the oracle.

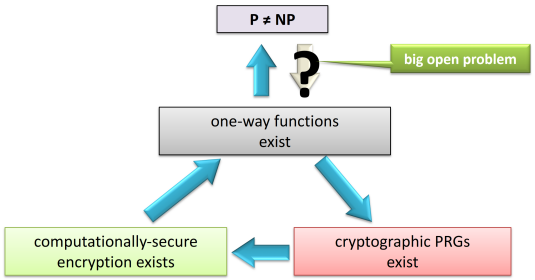
We say that (Enc, Dec) is **indistinguishable encryption** if any randomized polynomial time adversary guesses the message correctly with probability at most $0.5 + \varepsilon(n)$, where ε is negligible.
Note that we haven't made any assumptions about what happened before the game, so we must keep in mind the situation of a **chosen-plaintext attack (CPA)**, where E can send an arbitrary message to the oracle and know the encrypted version of its message. Also in this case E must not be able to find a strategy to win the game. It is easy to see that E can send the messages m_1 and m_2 both in the learning and in the challenging phase and the results must be different. For this reason we observe that every **CPA-secure** encryption has to be randomized or *have a state*. At this point one could ask if it is possible to prove **IND-CPA encryption**, i.e. an algorithm (Enc, Dec) with $|k| < |m|$ that satisfies the previous requirements **without any hardness assumption**. The bad news is that if such an algorithm exists, then $P \neq NP$ (and since no one has ever proven the P vs NP problem it is presumably very difficult to find). Finding IND-CPA encryption (without hardness assumption) is at least as difficult as proving $P \neq NP$.

Stream Ciphers

In cryptography, when we are in the context of computational security, we can prove conditional results. That is, we can show theorems of the type: *suppose that computational assumption A holds, then scheme X is secure*, or *suppose that scheme Y is secure, then scheme X is secure*. This means that we have to believe in some assumptions (such as the Diffie-Hellman assumption that computing the discrete logarithm in general is a difficult computational problem). In this section we work with pseudo random generators. We know them from everyday programming, but the pseudo random generators don't just need to pass some statistical tests in order to be considered useful for cryptographic purposes, but they need to satisfy more involved properties. Once we have a pseudo random generator, secure encryption and decryption are possible as showed in the following figure.



In the context of cryptography, we say that a generator is pseudorandom if, for all probabilistic polynomial time adversary, it is not possible to distinguish an output of the generator from a truly random string with non-negligible probability. But how do we construct a pseudo random generator? We begin with a very theoretical example: one-way functions. If we have a function such that it is easy to compute $f(x) \forall x$, but it is difficult, given a y, to find an x such that $f(x) = y$, then we can construct a pseudo random generator (this is one of the most famous results in symmetric cryptography, and we don't prove it here). Moreover we have that, if computationally-secure encryption exists, also one-way functions exist (because the game of encrypting/decrypting satisfy its purposes if it is easy in one direction and difficult in the other one. A summary of those implications is presented in the following figure.



More practical pseudo random generators are stream ciphers: given a seed s they output an infinite stream of bits. Of course, to encrypt multiple messages, we need to change the random bits that we use. An idea would be taking different spots of the outputs of the stream cipher that we obtained with a single random seed, but this imposes some practical challenges. A more practical idea is to apply as input to the stream cipher not only the seed, but also an initialization vector that changes in time and is later in cluded in the ciphertext. An example of this is RC4 which, given a seed, applies a key-scheduling algorithm based on permutations.

Private Key Authentication

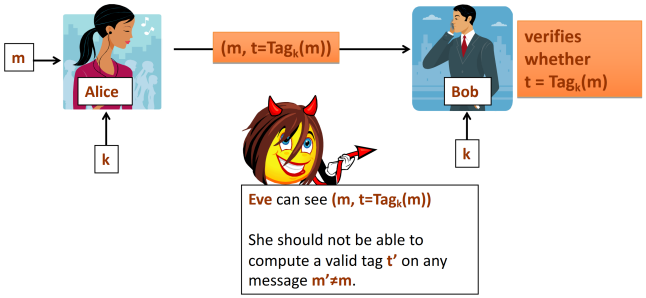
Authentication is often melt with integrity, let's define what those concepts are:

- **Integrity:** if Alice sends a message m to Bob, then a malicious user E can not interfere with the transmission (i.e. it can not modify the message in any way or insert a new message without Bob noticing).
- **Authentication:** when Bob gets a message m from Alice, he can check that this message actually comes from Alice.

Those concepts, in some applications, could be even more important then confidentiality. Think for example to bank transactions: it is crucial that nobody can change the content of the transaction (e.g. by changing the amount of money or the beneficiary of the transaction), while in some cases it is fine to read the transaction (think for example to Bitcoin transaction where only pseudonyms are used).

In general encryption does not guarantees integrity. One could think that if the decrypted message *makes sense* in some context, then the message was not corrupted. However this is not true: for example in OTP it is possible that by changing some bits of the ciphertext, the decryption returns a (different) message that still makes sense.

The basic idea of message authentication is the following:

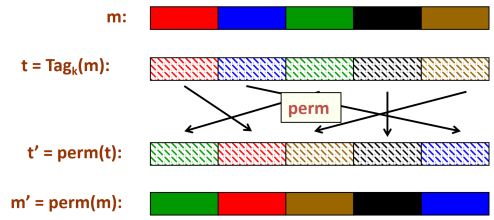


Since we have to be as pessimistic as possible the above schema must hold also if the attacker has several examples of messages with the corresponding tag. The same holds also if the attacker can choose the set of messages to be tagged.

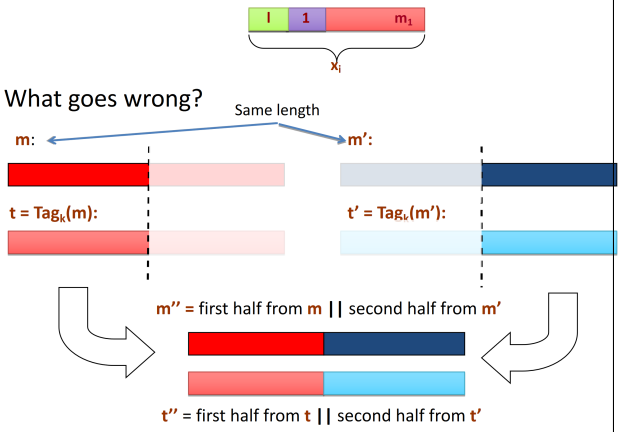
We say that an algorithm MAC is secure (in the context of authentication) if for all polynomial-time adversaries, the probability that the adversary breaks the algorithm (i.e. it generates an arbitrary tagged message accepted by the receiver), is negligible in terms of the security parameter n. Note that this definition does not protect against replay attacks: since the verify algorithm has no state there is no way to detect whether the tagged message is fresh or not. This problem has to be solved by higher applications layers via time-stamping, sequence numbers or other methods.

Now we see how we can construct a MAC starting from a block cipher. If we want to work only with messages of a given length n we can simply use a block cipher and everything works properly. If the length is less than n we can use zero padding. But what if the length is greater than n? We proceed step by step:

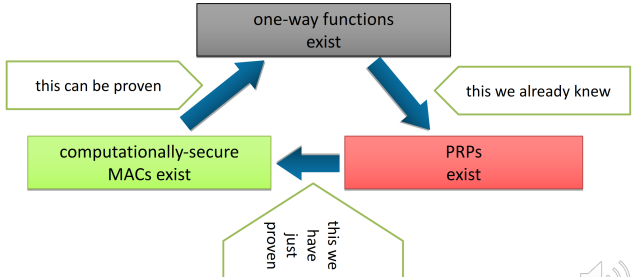
- We could just divide the message in several blocks, use the block cipher on every block and concatenate the results in order to obtain the tag. This does not work because we can perform permutations on the blocks as shown in the next figure:



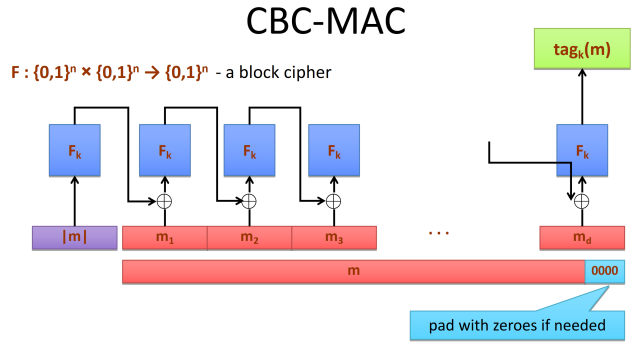
- In order to avoid the problem of the previous slide one could think that adding a counter to each block would solve the problem. However it is not the case: in fact, by cutting both the message and the tag in some spots one would be able to compute a new message with a valid tag.
- An improvement to the previous issue is to add the length of the message to each block. However this is still not the solution, in fact one could break this algorithm as shown in the next figure:



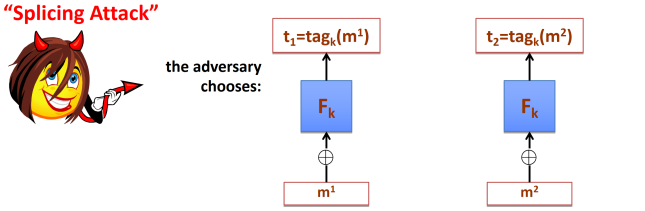
- The solution that works is to add a (per message) random value to each block. This random value is also added at the beginning of the tag, such that the receiver can take it and recompute the tag from the message in order to authenticate what it gets. If we use a pseudorandom permutation this algorithm is provably secure. In fact if we suppose that this is not a secure MAC, an adversary A could break it with non negligible probability and hence we could construct a distinguisher that distinguish F from a random permutation.

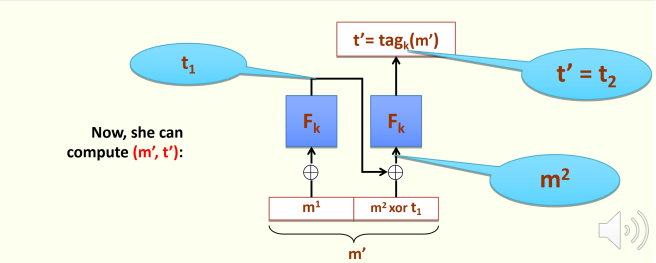


In order to construct a practical MAC (what we just showed is correct but not very efficient), we can use CBC-MAC:



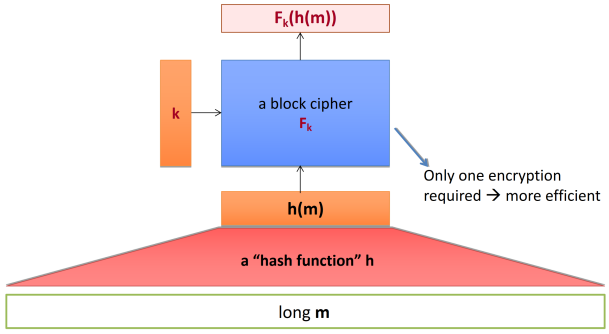
Note that the length parameter is necessary, otherwise one could perform a splicing attack:





Some people don't like CBC-MAC because hash functions are more efficient and they are not protected by export regulations.

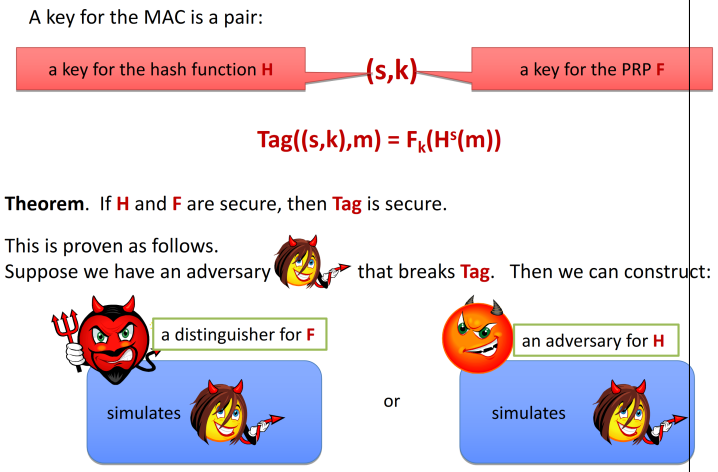
Hash functions: the idea is very simple, when we are dealing with long messages (more than one block), we use a hash function to compress it to the size of one block and then we use the keyed block cipher as in the single block case we have seen before.



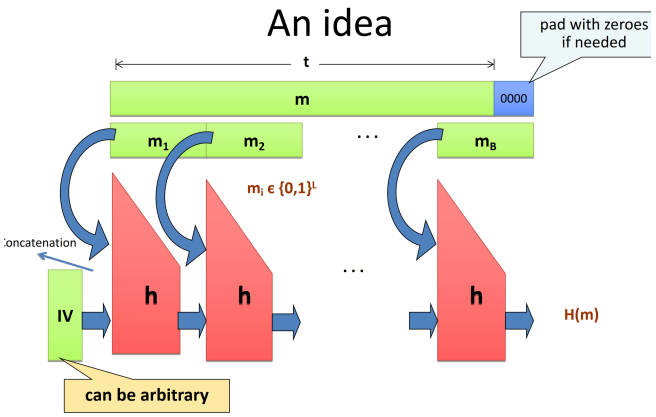
By the way: a similar method is used in public-key cryptography (it is called "**hash-and-sign**").

However, we have to chose the hash function properly since we are concerning with a cryptographic application. One of the most important properties that are required from the hash functions used in this context is collision resistance. This means that given the hash function, it should be difficult to find a pair (m, m') such that $H(m) = H(m')$. In fact, if an attacker would be able to forge such a pair, the authentication property would be invalid: in facts, according on how the final tag is computed, the equivalence of the value of the hash functions implies that the final tag is the same for both messages and hence the attacker would be able to forge a tag for a message different than the original one. Of course is impossible to avoid collisions since we are using a function from a very large domain (the one of all possible messages) to a much smaller domain (the one of all possible messages of a given size): the idea is, although those collisions exist, to design the hash function such that finding them is difficult for an adversary. In other words we say that *H is a collision-resistant hash function if it is practically impossible to find collisions in H* (where for *practically impossible* we mean with negligible probability in terms of a security parameter n). Examples of such functions are the family of SHA hash functions. In general terms we define an hash function as a probabilistic polynomial-time algorithm H such that H takes as input a key $s \in \{0,1\}^n$ and a message $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{L(n)}$ where $L(n)$ is some fixed function. Of course $L(n)$ can not be too small, otherwise finding a collision becomes trivial.

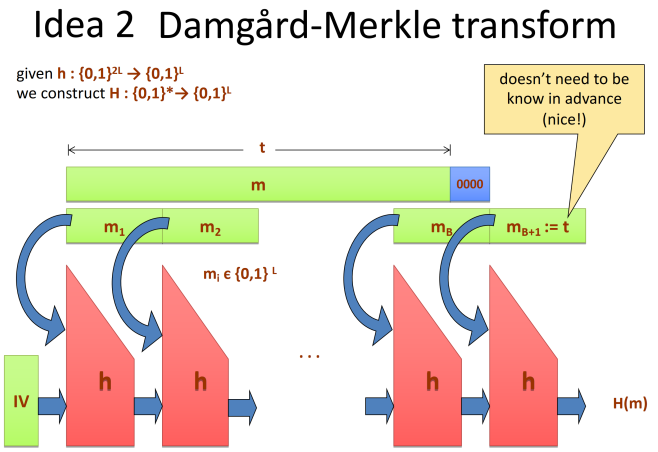
In general we have that, if H and F are secure, then the way we generate our tag is also secure. An informal proof of this fact is shown in the next figure:



In general, how do we construct H? We start from an easy case and then we generalize it. If we want to find an hash function that starts from a string of length $2L$ and outputs a string of length L we could just take the message and a key of the same length and use a block cipher. We can use this hash function (which we denote h) to construct an arbitrary hash function H. A first approach to do it is the following:



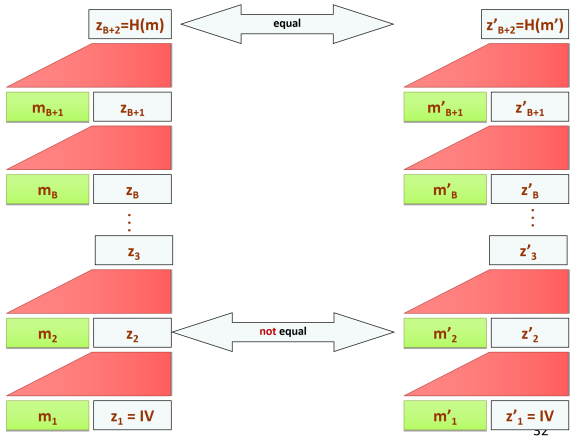
However this approach does not work since it introduces problems with the zero padding. There is no difference between a message that ends with zero and another one which is shorter and is made longer with zero padding. This problem can be easily overcome as shown in the following figure:



Now we want to show that this construction is secure. That is we want to prove that if h is a collision-resistant compression function, then H is also a collision-resistant hash function. We prove this indirectly, i.e. we assume that H is not collision-resistant and we show that h is also not collision-resistant. We distinguish two cases:

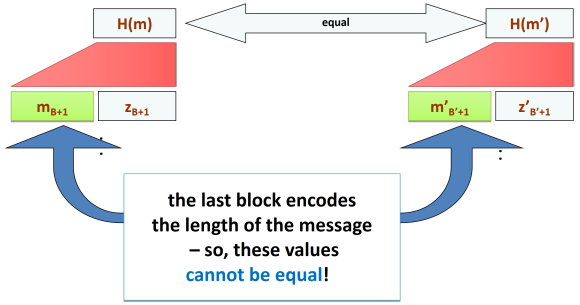
- $|m| = |m'|$: since we assume that H is not collsion-resistant there exist two different messa-

ges m and m' such that $H(m) = H(m')$. From the construction of H we get the following:



Formally we are finding an i such that $(m_i, z_i) \neq (m'_i, z'_i)$ and, since we assumed that $m \neq m'$ such an i always exists.

- $|m| \neq |m'|$: the construction is even simpler.

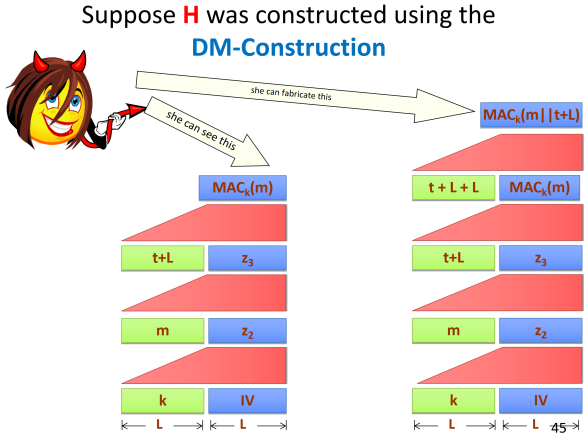


So, again, we have found a collision!

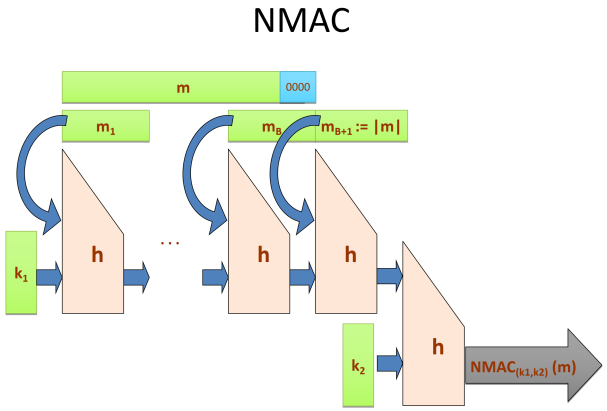
Until now we only concerned with collision-resistance, but this is not the only property that one could handle when treating hash functions. Two other (weaker) notions are:

- **Preimage resistance:** given hash value v, find x such that $h(x) = v$.
- **2nd preimage resistance:** given x, find $x' \neq x$ such that $h(x) = h(x')$.

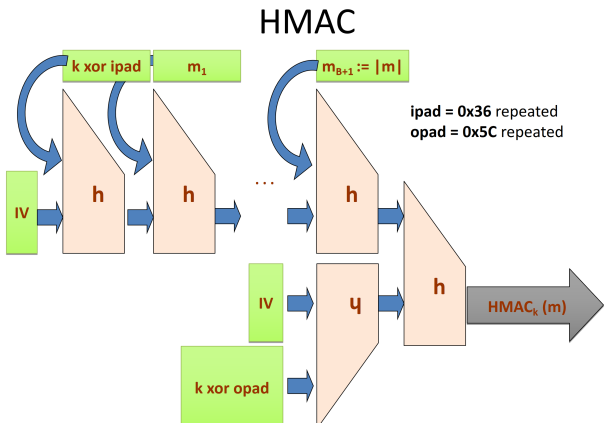
Until now we have seen that hash functions are a useful tool to simplify the case where we have long messages. But is it possible to construct MACs **only** from hash functions? A first idea would be to just return the value of the hash function applied to the message concatenated with the key. Unfortunately this is not secure, as shown in the next figure:



This naive idea does not work, but there are variants that are secure. A first example is **NMAC**:



NMAC is secure, however it has some drawbacks: first most real-world hash functions do not permit to use two different keys and second the new "key" (composed by k_1 and k_2 is too long). A solution to those drawbacks is **HMAC:**



This construction seems artificial but has some nice provable properties and can be easily implemented as:

$$HMAC_k(m) = H((k \text{ xor } opad)||H(k \text{ xor } ipad||m))$$

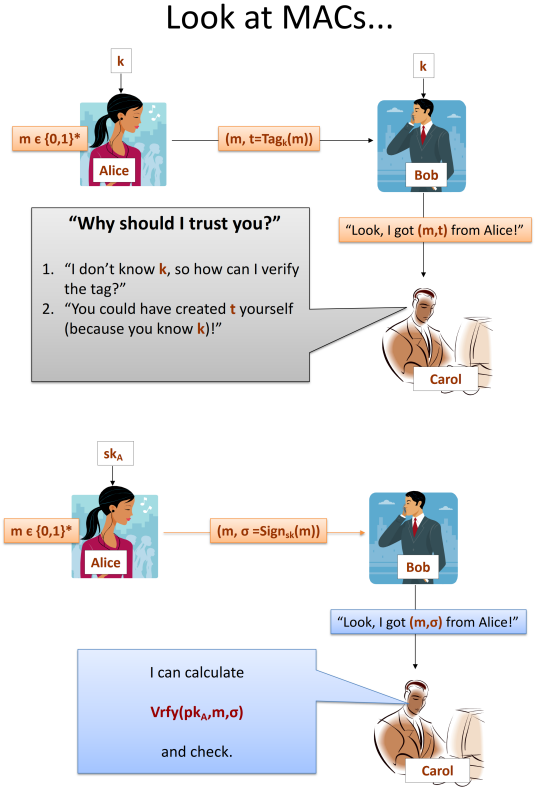
Public Key Cryptography: Introduction and Diffie-Hellman

Until now we have studied private key cryptography, which is also called **symmetric cryptography**. This names comes from the fact that both Alice and Bob can use the same scheme (with the same key) for both encryption and decryption (analogously, to sign and check the integrity of the message). Now we study public key cryptography or **asymmetric cryptography**. The idea is to use two different keys, a public one and a secret one. Concretely:

- The public key is used for encryption, the secret key for decryption. More in detail: Bob decides a public key and generates a secret key; then it makes the public key available to everyone such that everybody can send him an encrypted message (and Bob is the only one that can decrypt it since it is the only one that knows the secret key). In this context we speak of asymmetric cryptography: if Bob wants to send a message to Alice, he can not use this keys, he needs the public key decided by Alice.

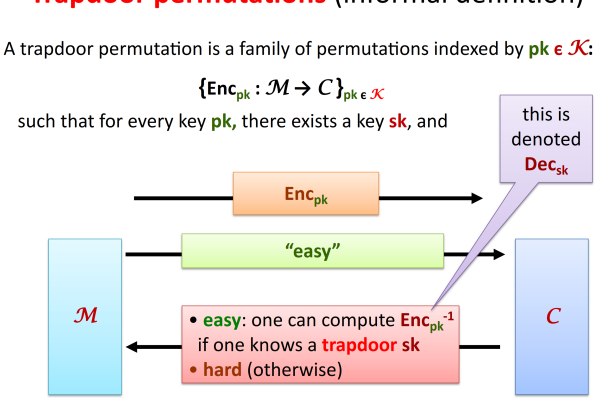
- In the case of authentication the secret key is used to compute the tag and the public key is used to verify the correctness of the tag. Hence every actor will make a public key available and, every time he sends a message, all other people will be able to verify that the tag was generated by that actor.

Public key cryptography has several advantages over private key cryptography: first of all we have that secrecy is not required and we need less keys (a key for each actor, not a key for each pair of actors). In the case of digital signatures we have, that are publicly verifiable and transferable. That is:

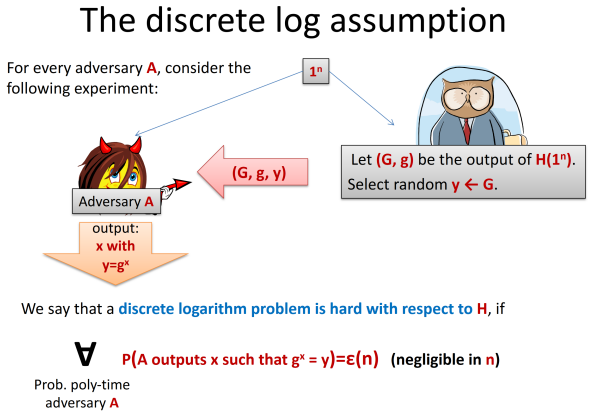


Moreover this mechanism provides non-repudiation: if Bob says that has got (m, σ) from Alice, Alice can not say *it's not true, I have never signed m.* A judge can easily verify whether the message was signed by Alice or not by using Alice's public key.

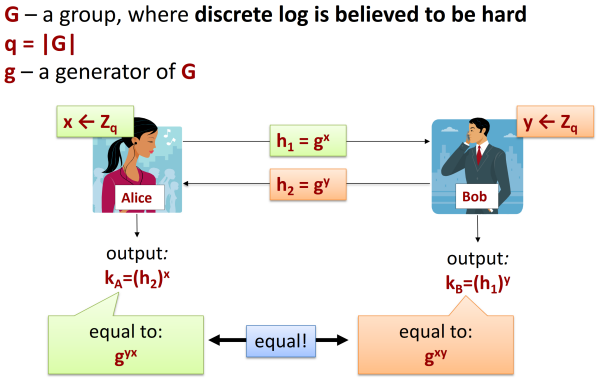
Is public key cryptography possible? Yes, an analogy in the real world is the one of a lock. A person can distribute several (open) locks to everybody. Then when someone wants to encrypt a message, he puts it in a box and he locks the box with the lock. Only the original person (the one who has the lock of the key), can unlock the lock and read the message in the box. In general we assume that there exist functions which are easy in one way and easy to invert if and only if one has the secret key. The idea is shown in the next figure:



What kind of *one-way functions* are used in practice? Here number theory comes into play, in facts we use mathematical problems that are believed to be difficult. Pay attention to the fact that some of the problems that are believed to be hard for classical computers are shown to be easy for quantum computers: an example of this is factorization, which is difficult for classical computer but hard for quantum computers. Another very famous problem that is exploited in this context is the **discrete logarithm problem**: i.e. it is easy to compute g^x where g is a generator of a cyclic group G, but it is difficult to do the inverse (of course, if $P = NP$ then computing discrete logarithms is easy). Examples of such groups are $\mathbb{Z}_p^*$, where p is a (large) prime.



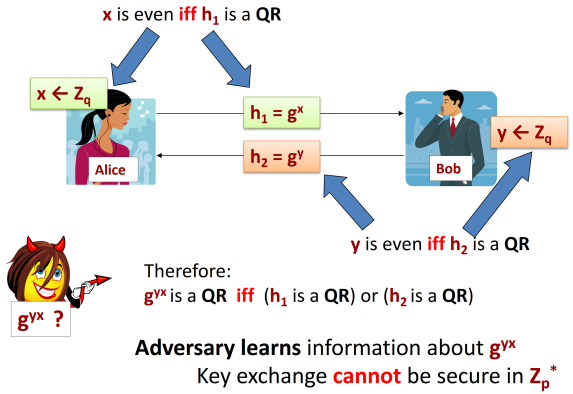
With discrete logarithm we have a one-way function. Now we will use this function to construct a **key exchange protocol**, which we will convert later into a public key encryption protocol. The key exchange protocol we study now is known as **Diffie-Hellman protocol** and is represented in the schema below:



We say that this protocol is secure if an adversary that knows G, g, g^x and g^y can obtain no additional information about g^{xy}, i.e. it can not distinguish between g^{xy} and a random string of the same length with non-negligible advantage. It is easy to see that if computing the discrete logarithm is easy, then this protocol is insecure (one could simply calculate x and y and then computing the key). However, if the discrete logarithm is a hard problem, then the protocol could still be insecure. In fact we have the following:

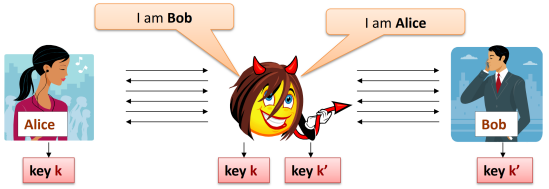
- An element y is called **quadratic residue** (QR) if there exists an a such that $a^2 = y$.

- Testing whether $y \in \mathbb{Z}_p^*$ is a QR is possible in polynomial time.
- By observing $y = g^x$ one can determine, whether x is even or odd. This holds because g must be odd (otherwise it could not be a generator) and if x is even, then g^x must have a quadratic residue, if x is odd then there is no quadratic residue.
- This implies the following problems in $\mathbb{Z}_p^*$:



- Solution: don't consider $\mathbb{Z}_p^*$ (because here we can gain information about the key), but consider the subgroup QR_p such thath p is a safe prime (i.e. $p = 2q + 1$, with q prime).

Note that the Diffie-Hellman key exchange protocol is secure only against a passive adversary, but is insecure against active adversaries that can launch a **man-in-the-middle** attack as follows:

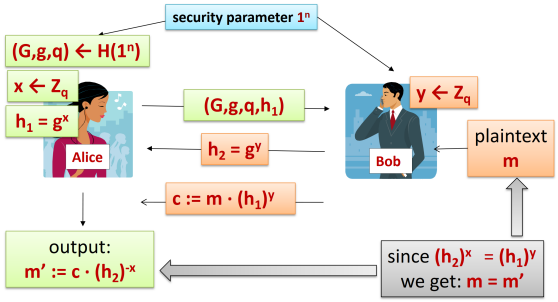


A very realistic attack!

So, is this thing totally useless?
No! (it is useful as a building block)

Does this means that Diffie-Hellman is useless? No, in some applications this risk might be acceptable and, in general, this is a very useful building block when combined with other measures.

Now we study **Elgaman encryption**, a public-key encryption scheme based on Diffie-Hellman key exchange that combines the idea of OTP.



In general, it can be proven that if the DDH problem is hard relative to G, then the Elgamal encryption scheme is CPA-secure.

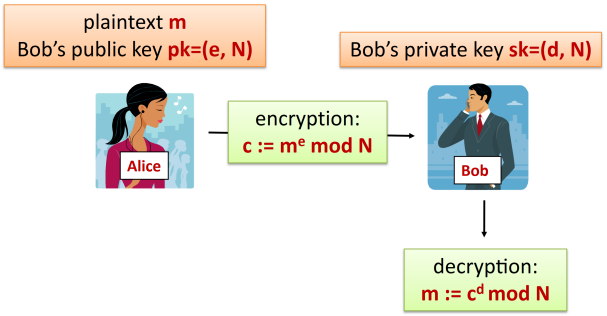
Public Key Cryptography: RSA

RSA encryption scheme was invented by Rivest, Shamir and Adleman in 1977 and in a nutshell is represented below:

RSA key generation

1. Choose two **prime** numbers p and q of size n
2. Compute $N = pq$
3. Compute totient $\Phi(N) = (p-1)(q-1)$
4. Choose e relative prime to $\Phi(N)$, i.e. $\gcd(e, \Phi(N)) = 1$
5. Compute $d = e^{-1} \bmod \Phi(N)$

Output: public key $pk = (e, N)$ and private key $sk = (d, N)$



Correctness: we have to show that $m = (m^e)^d \bmod N$. We give an intuition of the proof:

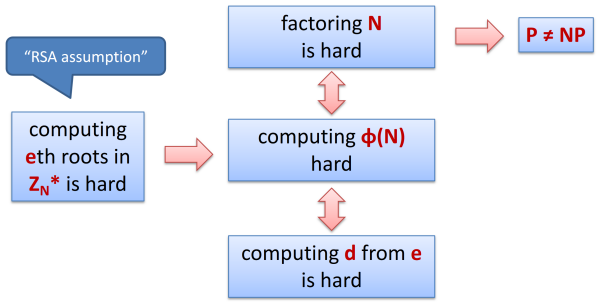
$$\begin{aligned}(m^e)^d &= m^{1 \bmod \Phi(N)} \\ &= m^{1+k\Phi(N)} \\ &= m\left(m^{\Phi(N)}\right)^k \\ &= m(1 \bmod N)^k \\ &= m \bmod N\end{aligned}$$

Where the fact that $x^{\Phi(N)} = 1 \bmod n$ follows from Euler's theorem (under certain conditions and also if those conditions don't hold the correctness holds, but the proof gets more elaborate).

Security and assumptions: in general RSA is secure if computing d from e is hard. One can show that this is equally difficult as computing $\Phi(n)$ or as factorizing. Moreover, there is also the RSA assumptions that comes into play (i.e. the fact that computing $e-$th root in Z_N^* is hard. The assumptions are summarized in the following figure:

Assumptions summary

N – a product of two large primes



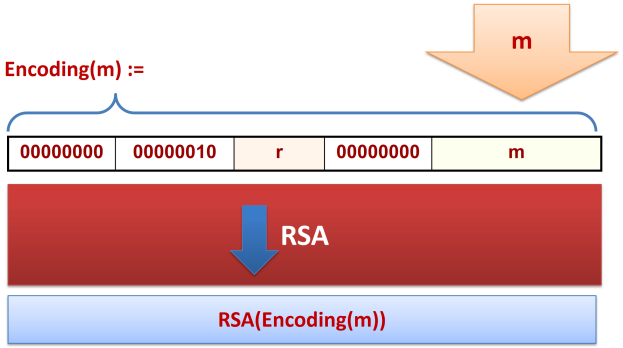
An important property of RSA is the homomorphicity, that is:

$$\begin{aligned} RSA_{e,N}(m_0 \cdot m_1) &= (m_0 \cdot m_1)^e \\ &= m_0^e \cdot m_1^e \\ &= RSA_{e,N}(m_0) \cdot RSA_{e,N}(m_1) \end{aligned}$$

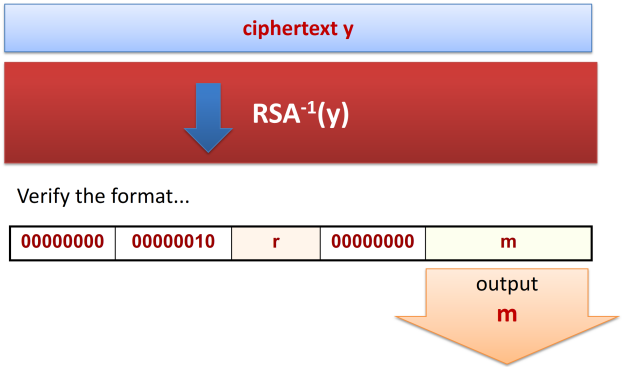
which has the bad consequence that by checking if $c_0 \cdot c_1 = c$ the adversary can detect, whether $RSA_{d,N}(c_0) \cdot RSA_{d,N}(c_1) = RSA_{d,N}(c)$

However, the RSA discussion we discussed until now, is not secure if we consider the chosen-plaintext attack we saw for public-key encryption. In that case we wanted that an adversary can not distinguish between the ciphertext of messages m_1 and m_2. In this case, since the adversary can compute the ciphertexts by himself, can easily win the game. In general, we said that no deterministic encryption scheme is secure. How can we add some randomness to RSA? The solution is presented graphically in the following figures:

How to encrypt?

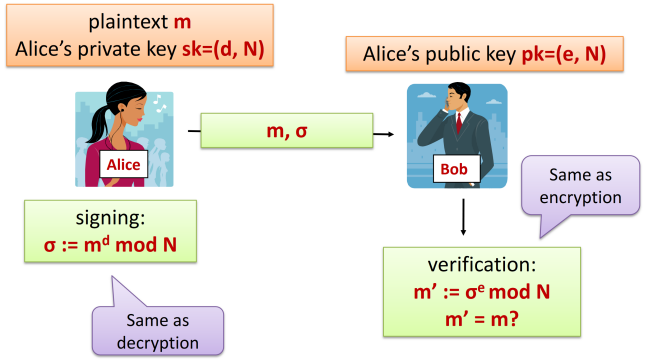


How to decrypt?



This method works only for messages which are shorter than the key. What can we do for longer texts? A first, naive solution would be to divide the message in blocks and encrypt each block separately. A better solution is combining private-key encryption with public-key encryption. For example one could use private-key encryption to exchange a public key k and then use public-key encryption with k with some efficient methods of the previous sections to encrypt the message.

RSA signatures: the mechanism is very similar to encryption.



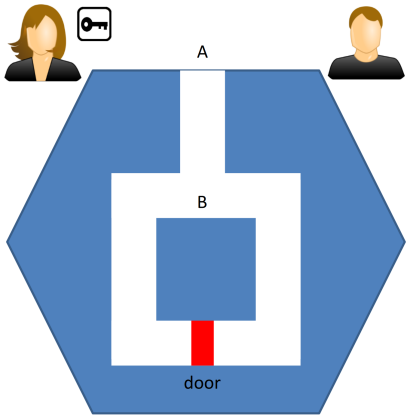
We say that this scheme is secure if an adversary, by having an arbitrary set of examples of signed messages, can not forge another one for a new message with non-negligible probability. This does not hold with the scheme above, in facts one could design the following attack by exploiting the homomorphic property of RSA:

1. Query signature σ_1 for m_1
2. Query signature σ_2 for m_2
3. Output $(m_1 \cdot m_2, \sigma_1 \cdot \sigma_2)$

The solution to this issue is to hash the message before sign it. This method is widely used, but there is no proof that, when H is only collision-resistant, then the signature scheme is unforgeable. In order to get a formal proof we need H to be a random function.

Public Key Cryptography: Zero Knowledge Proofs

The basic idea of **Zero Knowledge Proofs** is very simple: we want to find a protocol to show that we know a secret s without revealing s. An analogy is the following:

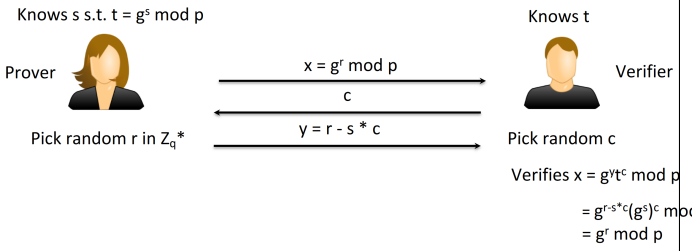


We have a prover P that has to show that he knows s (without revealing it) and a verifier V that has to be sure that P knows s. In this case we can imagine the following maze with a door in the middle. In order to unlock this door one needs to know s. The zero knowledge proofs is simply that P goes into the maze and takes an arbitrary direction. V can watch P all the time and V is happy if and only if P enters the maze in a direction and exits from the other one. In this case V is sure that P knows s.
Some properties of a zero knowledge proofs are:

- **Completeness:** if both P and V are honest, the protocol works.
- **Soundness:** no P without knowledge of s can convince V with non-negligible probability. That is, the prover can not make false statements. Moreover, assume that P can convince V with non-negligible probability. Then there must exist a knowledge extractor algorithm that, given P, computes the secret.
- **Zero knowledge:** the proof does not leak any information about s.
- There exists a **simulator** that, on input the initial state of V, outputs a fake transcript of the protocol.

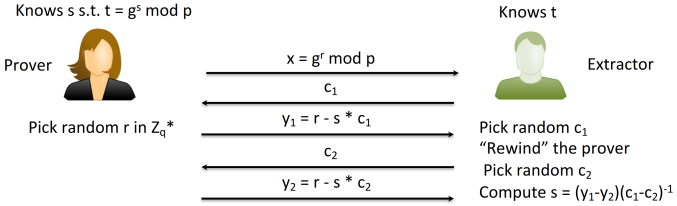
An example is given by **Schnorr identification protocol**:

- Prove knowledge of discrete log
  - Public parameters:
    - Prime p and q such that q divides p-1
    - g generator of an order-q subgroup of Z_p^*



Schnorr - Soundess

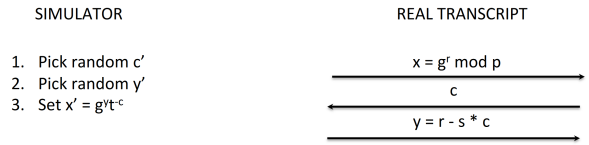
- Knowledge extractor
  - Black-box access to P
  - Extract secret s
  - "Run" P twice on the same randomness



if the prover can produce two accepting transcripts, she must know s, as s can be extracted from these transcripts.

Schnorr – Honest Verifier ZK

- Simulator produces a transcript that is indistinguishable from a real transcript
  - The simulator does NOT know s



x,c,y is indistinguishable from x',c',y'

Public Key Cryptography: Commitment

The idea is that a party wants to commit a value x without revealing it, but without the possibility of changing it later. A common analogy is the one of a bid in an auction: everyone writes the bid, puts it in an envelope and puts it on the table. In this way other parties don't know the value of the bid, but the value is fixed and can't be changed.
In the **commit** stage, P locks x in a box and sends it to V. In the **reveal** stage, P gives the key box to V, V opens the box and learns x.
The properties of a commitment protocol are:

- **Hiding:** after commit phase, V learns nothing about x.
- **Binding:** after the commit phase, there is only one value (i.e. x), that P can succesfully reveal.

Let's discuss a first example:

- Commit: P generates a key k and outputs $c = Enc_k(x)$
- Reveal: P sends k to V and V decrypts c with k in order to learn x.

This scheme is hiding (because there is no way for V to learn x after the commit phase without decrypting the message), but is not binding: in facts, P can send a key $k' \neq k$ to V in the reveal phase and in this way V learns $x' = Dec_{k'}(Enc_k(x)) \neq x$.
Another one could be the following:

- Commit: P outputs $c = H(x)$
- Reveal: P sends x to V, V verifies that $c = H(x)$

This scheme is binding (because revealing a $x' \neq x$ would mean finding an x' such that $H(x') = H(x)$ and this is a contraddiction to the fact that H is collision resistant), but not hiding.
The following scheme, known as **Pedersen commitment scheme**, works as follows:

- Setup (receiver): pick primes p and q such that q divides $p-1$. Receiver picks generators g, h of the order q subgroup of $\mathbb{Z}_p^*$. The public parameters are p, q, g, h, while there is a secret parameter a which is the discrete logarithm of k basis g, i.e. $h = g^a \bmod p$.
- Commit: pick random r from $\mathbb{Z}_q$ and output $c = g^x h^r \bmod p$.
- Reveal: output x and r.
- The verifier checks that $c = g^x h^r \bmod p$.

This scheme is perfectly hiding because for any x' there exists an r' such that $g^x h^r = g^{x'} h^{r'}$ and therefore, for any given c, any value x is likely to be committed. Note that $r' = (x - x')a^{-1} + r \bmod q$. This scheme is also computationally binding: in facts, given c, we have that if P can reveal x and x', then P can compute the discrete logarithm. In facts P has to know x, r, x', r' such that $g^x h^r = g^{x'} h^{r'} \bmod p$ and since $h = g^a \bmod p$ we have $x + ar = x' + ar'$ and P can compute a, which is the discrete logarithm of h.