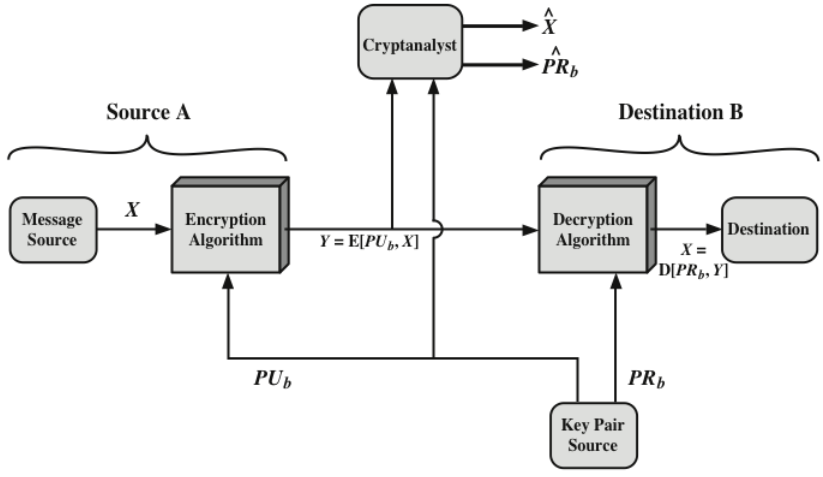
Asymmetric Encryption & Hash Functions

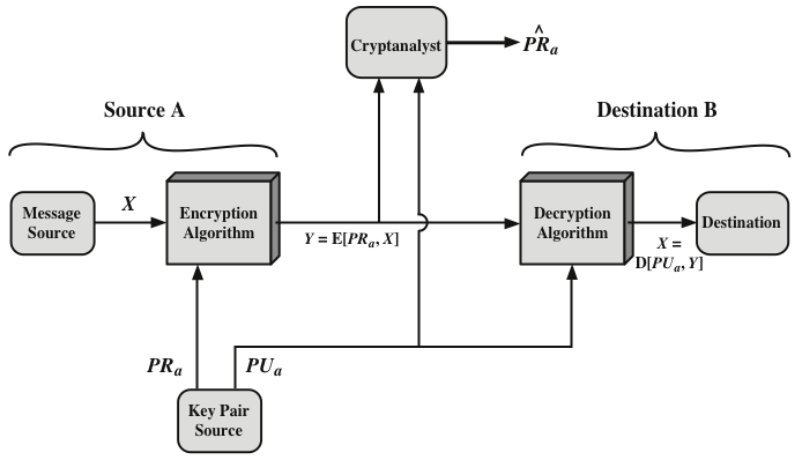
Ibéria Medeiros

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

Encryption with Public-Private Key



Signature with Public-Private Key



Applications

- Typical areas of application with asymmetric cryptography
  - Encryption
  - Signatures
  - Key exchange
- Not all algorithms can be used for all applications

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Conditions that the algorithms **must fulfil**

- It is computationally **easy** for a party B to **generate a pair**
- It is computationally **easy** for a sender A, knowing the public key and the message to be encrypted, to **generate the corresponding ciphertext**
- It is computationally **easy** for the receiver B to decrypt the resulting ciphertext using the private key to **recover the original message**
- It is computationally **infeasible** for an adversary, knowing the public key, to **determine the private key**
- It is computationally **infeasible** for an adversary, knowing the public key and a ciphertext, to **recover the original message**
- Probably we would also like that the two keys can be **applied in either order**

- Need a **trap-door one-way function**
  - » $Y = f_K(X)$ and $X = f_K^{-1}(Y)$ are **easy** if K is known
  - » $X = f_K^{-1}(Y)$ **infeasible** without knowing K

Public-Key Cryptanalysis

- A public-key encryption scheme is vulnerable to a **brute-force attack**
  - countermeasure: ***use large keys***
  - key size must be small enough for practical encryption and decryption
  - larger key sizes have an impact on performance, resulting typically in encryption/decryption speeds that are too slow for general-purpose use
  - public-key encryption is currently **confined to key management and signature applications**
- Another form of attack is to **find some way to compute the private key given the public key**
  - to date it has **not** been mathematically proven that this form of attack is infeasible for a particular public-key algorithm
- Finally, there is a **probable-message attack**
  - for small clear text (e.g., 56 bit key), encrypt all possible values with the public key and then make a match with the ciphertext
  - ***append some random bits*** to simple messages to prevent the attack

Size of the Public/Private Key

- Many public key algorithms have their security based on the complexity of factoring large numbers
  - *example*: given $n = pq$ it is very difficult to find the prime numbers p and q
- Nevertheless, until now no one has ever proved that the problem is hard, and through the years **newer and more efficient methods** have been proposed
- Over the years, **many famous people have been wrong** about future factoring capabilities

  [1977, R. Rivest] said that in order to *factor a number with 125 digits it would take* ***10^{15} years***

  ... but reality sometimes may surprise us ...

Evolution of Factoring Results (RSA algorithm)

Number of Decimal Digits	Approximate Number of Bits	Date Achieved	MIPS-years	Algorithm
100	332	April 1991	7	quadratic sieve
110	365	April 1992	75	quadratic sieve
120	398	June 1993	830	quadratic sieve
129	428	April 1994	5000	quadratic sieve
130	431	April 1996	1000	generalized number field sieve
140	465	February 1999	2000	generalized number field sieve
155	512	August 1999	8000	generalized number field sieve
160	530	April 2003	—	Lattice sieve
174	576	December 2003	—	Lattice sieve
200	663	May 2005	—	Lattice sieve
232	768	December 2009		

and it expected that over the next decade 1024 bits will be factored ....

RSA Algorithm

- Developed in 1977 by Ron Rivest, Adi Shamir and Len Adleman

- Basic ideas
  - the plain text is divided in blocks
  - each block corresponds to a number **M** with a binary value less than ***n*** (each block has a number of bits ***x*** which is less or equal than $log_2(n)$, i.e., in practice $2^x < n$)

Public Key:	$\{e, n\}$
Private Key:	$\{d, n\}$
Encrypt:	$C = M^e \bmod n$
Decrypt:	$M = C^d \bmod n = (M^e \bmod n)^d \bmod n =$ $= (M^e)^d \bmod n = M^{ed} \bmod n$

Calculations of the Parameters

Question: Are there numbers e, d, and n such that $M^{ed} \equiv M \bmod n$ for all $M < n$?

Corollary of the Theorem of Euler: Given two prime numbers p and q and two integers n and m, such that $n = pq$ and $0 < m < n$, then the following is true

$$m^{\Phi(n)+1} \equiv m \bmod n$$

(Recall that the totient function, $\Phi(n)$, returns the number of integers less than n that are relatively prime to n; and, for p and q which are prime, $\Phi(n) = \Phi(pq) = (p-1)(q-1)$)

1. We get the relation, $ed = \Phi(n) + 1$
2. therefore, $ed \bmod \Phi(n) = [\Phi(n) + 1] \bmod \Phi(n) = 1 \bmod \Phi(n)$
3. or that, $ed \equiv 1 \bmod \Phi(n)$
4. and to conclude, $d \equiv e^{-1} \bmod \Phi(n)$

Condition: ***e*** *and* ***d*** *are the multiplicative inverses mod $\Phi(n)$; these inverses exist if* ***d*** *(and therefore* ***e****) are relatively prime to $\Phi(n)$, i.e., $gcd(\Phi(n), e) = gcd(\Phi(n), d) = 1$*

RSA Algorithm

Calculate the keys:

Select p, q	p, q *are prime*	(***private***, ***chosen***)
Calculate $n = p * q$		(***public***, ***calculated***)
Calculate $\Phi(n) = (p-1)(q-1)$		(***private***, ***calculated***)
Select integer e	$gcd(\Phi(n), e) = 1; \quad 1 < e < \Phi(n)$	(***public***, ***chosen***)
Calculate d	$d = e^{-1} \bmod \Phi(n)$	(***private***, ***calculated***)

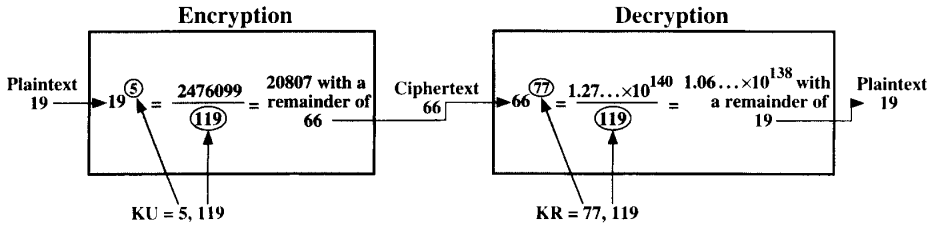
Public key $\{e, n\}$
Private key $\{d, n\}$

Encrypt/Decrypt:

Plaintext:	$M < n$
Encrypt:	$C = M^e \bmod n$
Decrypt:	$M = C^d \bmod n$

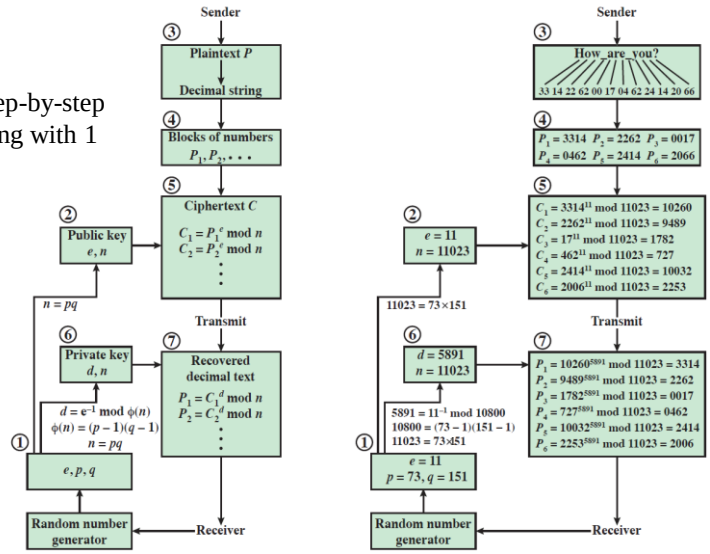
Example

1. Select two prime numbers, $p = 7$ e $q = 17$
2. Calculate $n = pq = 7*17 = 119$
3. Calculate $\Phi(n) = (p-1)(q-1) = 96$
4. Select e in such a way that it is relatively prime to $\Phi(n) = 96$ and less than $\Phi(n)$; choose for example $e = 5$
5. Calculate d in such a way that $de \equiv 1 \bmod 96$ and $d < 96$; therefore $d = 77$ *(using the extended Euclid´s algorithm)*



Example with Several Blocks

Go step-by-step starting with 1



Implementation Aspects

- Selection of **prime numbers (*p*, *q*)**
  - since *n* will be known by the adversary, *p* an *q* have to be chosen from a set with many elements (i.e., with many bits)
  - since today there is **not a good method** to generate large prime numbers, the method to obtain *p* and *q*
    - » select a large random odd number
    - » test if the number is prime (such as the Miller-Rabin test);
    - » if the number is not prime, return to the first step, otherwise terminate the algorithm

Implementation Aspects (cont.)

- after the calculation of Φ(n), one can use the following procedure to **obtain *e* and *d***
  - » select a random *e*, $1 < e < \Phi(n)$
    - *many times one would like to choose like e = 65537 to improve speed of exponentiation since this numbers only has 2 bits at 1*
  - » test if $gcd(\Phi(n), e) = 1$ (otherwise return to previous step)
  - » calculate the value of $d \equiv e^{-1} \bmod \Phi(n)$

Extended Euclides Algorithm
Returns the gcd() of 2 numbers and
$e^{-1} \bmod f$

EXTENDED EUCLID(e, f)
1. (Xl, X2, X3) = (1,0, f); (YI, Y2, Y3) = (0, 1, e)
2. **if** Y3 = 0 **return** X3 = gcd(e, f): no inverse
3. **if** Y3 = 1 **return** Y3 = gcd(e, f); Y2 = e^{-1} mod f
4. Q = ⌊X3/Y3⌋
5. (T1,T2,T3) = (Xl - QY1,X2 - QY2,X3 - QY3)
6. (Xl, X2, X3) = (Y1, Y2, Y3)
7. (Y1, Y2, Y3) = (T1, T2, T3)
8. **goto** 2

Implementation Aspects (cont.)

- **Exponentiation modulo *n*** ($a^m \bmod n$)
  - notice the following property of modular arithmetic

    $(a * b) \bmod n = [(a \bmod n) * (b \bmod n)] \bmod n$
  - one idea is to notice that

    $$a^{16} \bmod n = a*a*a \ldots a*a*a*a*a \bmod n = \left(a^{8} \bmod n\right)^2 \bmod n =$$

    $$= \left(\left(a^{4} \bmod n\right)^2 \bmod n\right)^2 \bmod n = \left(\left(\left(a^{2} \bmod n\right)^2 \bmod n\right)^2 \bmod n\right)^2 \bmod n$$

  - another, consider the following observation for positive integer numbers, *m*, represented in a binary form by $b_k b_{k-1} b_{k-2} \ldots b_0$

    $$m = \sum_{b_i \neq 0} 2^i \qquad a^m = a^{\left(\sum_{b_i \neq 0} 2^i\right)} = \prod_{b_i \neq 0} a^{(2^i)}$$

    $$a^m \bmod n = \left[\prod_{b_i \neq 0} a^{(2^i)}\right] \bmod n = \left(\prod_{b_i \neq 0} \left[a^{(2^i)} \bmod n\right]\right) \bmod n$$

Implementation Aspects (cont.)

- Exponentiation modulo n *(a^m mod n) (continue)*

Calculation of a^m mod n:

```
d = 1
for i = k downto 0
    do
        d = (d * d) mod n
        if b_i = 1 then
            d = (d * a) mod n
return d
```

- exponent m has k+1 bits
- b_i are the bits of exponent m

a = 3, n = 5, m = 12 ->1100

m	1	1	0	0
d	3	2	4	1

RSA Security

- Brute force attacks: *already seen*, and are based on the test of all possible keys until one finds one that decrypts the message (Solution: increase key size)

- Mathematical attacks: based for example on the attempt of factoring n into two prime numbers ($\rightarrow \Phi(n) = (p-1)(q-1) \rightarrow d = e^{-1} \bmod \Phi(n)$)
  - *as already seen*, the currently available algorithms are not very effective for very large numbers, but there have been improvements …….
  - some rules to increase the difficulty of these attacks
    - » p and q should have approximately the same number of bits
    - » $(p-1)$ and $(q-1)$ should have a high prime factor
    - » $gcd(p-1, q-1)$ should be small
  - in the near future, **keys with 2048** bits should provide a sufficiently good security level

RSA Security

- Timing attacks: (can be used with other encryption algorithms)
  - based on the **measurement of the amount of time** certain operations take to be performed (e.g., using a snooper) while decrypting some data
  - does not need to know the clear text

  Example: $a^m \bmod n$ *(see previous algorithm)*
  - » the result is obtained bit by bit, with an **extra** multiplication modulo n every time the exponent bit is 1 ($d = (d * a) \bmod n$)
  - » the multiplication of a and d modulo n for some values takes much more time than the average, and the adversary knows these cases
  - » starting with iteration $i = k$, we try to guess bit b_k
    - if the total execution time of the algorithm is always slow, for the known values of a and d = 1, then the bit is 1
    - if in one of the executions is fast, then the bit is 0
  - » this procedure continue with bits b_{k-1}, b_{k-2} ....

Diffie-Hellman Key Exchange

- Objective: securely exchange a secret key to be employed in encryption

- Based on the difficulty of discrete logarithm calculations
  - given a prime number p
  - given a primitive root a of p (a is a primitive root : $a \bmod p$, $a^2 \bmod p$, ... $a^{p-1} \bmod p$, are all distinct and with values from *1* to *p-1*)
  - given a number b, its **discrete logarithm *i*** is defined as

    $$\boldsymbol{b = a^i \bmod p} \qquad \text{(notice that: } \log_a b = i \rightarrow b = a^i \text{)}$$

  Note 1: *a, b, p are public, and **only i is secret***

  Note 2:

  $w = a^n \bmod p$; $z = a^k \bmod p$

  $w^k \bmod p = (a^n \bmod p)^k \bmod p = a^{nk} \bmod p =$

  $= (a^k \bmod p)^n \bmod p = z^n \bmod p = K$

Diffie-Hellman Protocol

Global Public Parameters

p	prime number
a	primitive root (a < p)

Parameters of A

Select X_A	$X_A < p$
Calculate Y_A	$Y_A = a^{XA} \bmod p$

Parameter of B

Select X_B	$X_B < p$
Calculate Y_B	$Y_B = a^{XB} \bmod p$

Secret key of A

$K = (Y_B)^{XA} \bmod p$

Secret key of B

$K = (Y_A)^{XB} \bmod p$

User A	User B
* Generate random number $X_A < p$	
* Calculate $Y_A = a^{XA} \bmod p$	
Y_A →	
	* Generate random number $X_B < p$
	* Calculate $Y_B = a^{XB} \bmod p$
	* Calculate $K = (Y_A)^{XB} \bmod p$
← Y_B	
* Calculate $K = (Y_B)^{XA} \bmod p$	

Example : $p = 97, a = 5$

$Y_A = 5^{36} = 50 \bmod 97$ $\quad K = 44^{36} = 75 \bmod 97$

$Y_B = 5^{58} = 44 \bmod 97$ $\quad K = 50^{58} = 75 \bmod 97$

Man-in-the-middle Attack

Problem : Alice and Bob do not know that they are talking to each other!

- How to solve the problem?
  - Each user generates a discrete logarithm X_i (*the private key*) and a value Y_i (*the public key)*; the value Y_i are saved in a secure central directory
  - If a user B wants to send a message to A, he goes to the central directory and gets Y_A, generates X_B, and encrypts the message
  - If user A receives a message from B and is able to decrypted using a key created with Y_B, then only B could have sent the message

Problem: *replay attacks …*

Elgamal Protocol for Encryption

Global Public Parameters

p	prime number
a	primitive root (a < p)

Parameters of A

Select X_A	$X_A < p$
Calculate Y_A	$Y_A = a^{X_A} \bmod p$

Parameter of B

Plaintext M	M < p
Select X_B	$X_B < p$
Calculate K	$K = (Y_A)^{X_B} \bmod p$
Calculate C1	$C1 = a^{X_B} \bmod p$
Calculate C2	C2 = K M mod p
Ciplertext	(C1, C2)

Decryption by A

Calculate K	$K = (C1)^{X_A} \bmod p$
Plaintext M	$M = (C2\ K^{-1}) \bmod p$

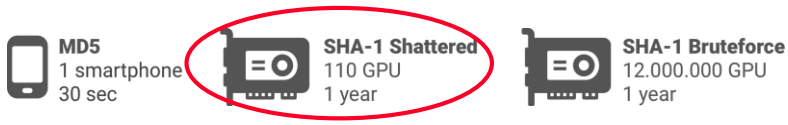
User A	User B
* Generate random number $X_A < p$	
* Calculate $Y_A = a^{X_A} \bmod p$	
Y_A →	• Plaintext M • Generate random number $X_B < p$ • Calculate $K = (Y_A)^{X_B} \bmod p$ • Calculate $C1 = a^{X_B} \bmod p$ • Calculate C2 = K M mod p
← (C1, C2)	
* Calculate $K = (C1)^{X_A} \bmod p$ • Calculate $M = (C2\ K^{-1}) \bmod p$	

Note : ***K should be different per encrypted block;*** *if several blocks need to be encrypted then a distinct K needs to be used in each of them*

Hashing Algorithms

SHA-1 With Hash Collision Attack (Feb 2017)

- Researchers from Google and Centrum Wiskunde & Informatica research center in Amsterdam, Netherlands, have developed a **collision attack** that defeats the **SHA-1 cryptographic algorithm**
- Becomes practical to craft two colliding PDF files and obtain a SHA-1 digital signature on the 1st PDF which can be abused as a valid signature on the 2nd PDF
- Idea (see https://shattered.io/ for details)
  - Start by creating a PDF prefix specifically crafted to allow the generation of two documents with arbitrary distinct visual contents, but that would hash to the same SHA-1
  - Then, lots of computational work
    - » Nine quintillion (9,223,372,036,854,775,808) SHA1 computations in total
    - » 6,500 years of CPU computation to complete the attack first phase
    - » 110 years of GPU computation to complete the second phase



Hash Function

- An hash function generates from an arbitrarily sized amount data a small fixed output that represents it, which is called the *hash*

- The hash can be used for example to
  - detect that a message was changed while being transmitted
  - create a secret key from a passphrase
  - generate a Message Authentication Code (MAC) by using a secret key
  - produce a digital signature by using a private key
  - construct a pseudorandom generator

Homework: Recall how to perform the above operations with an hash function!

(Main) Properties of the Hash Function

1. Given an arbitrary M, it is easy to compute $H(M) = h$, where h has a fixed size
2. Given h, it is very hard to obtain X such that $H(X) = h$ — *Preimage resistance or One-way property*
   - it is important in a authentication mechanism such as $MAC = H(M \,||\, K_{ab})$ (if this property was violated one could get K_{ab})
3. Given M, it is very hard to find an X such that $H(M) = H(X)$ — *Second preimage resistance or Weak collision resistance*
   - it is important in a digital signature mechanism such $Sign = E_{KR\_A}(H(M))$ (if this property was violated one could substitute the signed message for another)
4. It very hard to find M and X such that $H(M) = H(X)$ — *Collision resistance or Strong collision resistance*
   - it is important to avoid *birthday attacks*

Traditionally the property under attack!! Why??

Birthday Paradox

- Consider a group of K people. What is the minimum value of K that ensures with a probability higher that 50% that there are two people that have the same birthday day?

1. Number of ways to organize K people that do **not** have common birthdays
   $N=365 * 364 * ... * (365 – K + 1) = 365! / (365 – K)!$
2. Total number of ways to organize K people (even if there are common birthdays)
   $M = 365 * 365 * 365 * ... = 365^K$
3. Probability that there are no common birthdays
   $Q(365, K) = N / M = 365! / [(365 – K)! 365^K]$
4. Probability that there are common birthdays
   $P(365, K) = 1 – Q(365, K) = 1 - 365! / [(365 – K)! 365^K]$

Example:
$P(365, 23) = 0,5073$ $P(365, 57) > 0,99$ $P(365, 100) = 0,9999997$

Birthday Attack

- Example: Alice convinces Bob to sign a contract
  - Alice prepares two versions of the same contract, one that is favorable to Bob and another that makes him bankrupt
  - Alice makes small changes to each version of the contract ***(--- how? ---)*** and calculates the hash of each one
  - Alice compares the hashes of the various versions and finds two that have the same hash
  - Alice allows Bob to sign the favorable version of the contract using a digital signature algorithm that is based only in the hash (e.g., DSS)
  - Later on, Alice substitutes the contract and .....

INTERESTING PART: *By the Birthday Paradox, Alice only needs to generate two sets of $2^{n/2}$ versions, for an hash value of n-bits*

How can we generate the versions?

- Example 1 : substitute a space with something like space-backspace-space, place one or more spaces at the end of the line, ….
- Example 2 : This letter allows the creation of 2^{37} versions

Dear Anthony,

{This letter is / I am writing} to introduce {you to / to you} {Mr. / --} Alfred {P. / --}

Barton, the {new / newly appointed} {chief / senior} jewellery buyer for {our / the}

Northern {European / Europe} {area / division}. He {will take / has taken} over {the / --}

responsibility for {all / the whole of} our interests in {watches and jewellery / jewellery and watches}

in the {area / region}. Please {afford / give} him {every / all the} help he {may need / needs}

to {seek out / find} the most {modern / up to date} lines for the {top / high} end of the

market. He is {empowered / authorized} to receive on our behalf {samples / specimens} of the

Summary of the security of Hash functions

- **Brute force attacks** (hash with *n* bits)
  - *One-way function :* effort in the order of 2^n
  - *Weak collision resistance* : 2^n
  - *Strong collision resistance* : $2^{n/2}$

  (e.g, in 1994 there was proposal for a 10 million dollars machine that could find a collision in MD5 in 24 days —— MD5 has hashes of 128 bits)

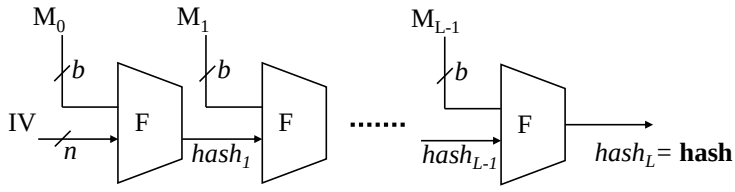
NOTE: not even in MD5 is possible to violate the one-way property

- **Cryptanalysis**
  - usually aim at **finding a collision** in the compression function through the analysis of its internal structure
  - More recently there have been attacks that have successfully found collisions in MD5 (and even SHA-0/1), usually for messages with pre-defined formats

    (e.g., in 2006 it was published a description of an attack that could find a collision in MD5 in the order of 1 minute in a laptop; in 2005 it was described an attack that could find collisions in SHA-0 in 2^{39} operations)

Ideas for Constructing an Hash Function

- Hash functions typically
  - divide the message (of an arbitrary size) in *L* blocks of *b* bits
    - » ***padding*** is normally added to the end
    - » the final block normally includes the ***size*** of the whole message
  - apply a **compression function** to each block that receives two inputs (*n* bits from the previous step, the ***chaining value***, and b bits from the message) and outputs *n* bits



Iterated hash function (by **Merkle-Damgard**)

Example attack to hash functions

- **SHA-1 Freestart Collision** (https://sites.google.com/site/itstheshappening/)
- In the Damgard-Merkle construction for hash functions the compression function takes as input: 1) a ***message block*** and 2) a ***chaining value***
- For the very first block there is not previous "chaining value“, and instead a particular value, called an ***initialization vector*** **(IV)** is used
- A ***freestart collision*** is a collision where the attacker can choose the IV
- Two, slightly, different IVs were found (only two bits are different) to create a collision for particular messages with the whole 80 rounds of the SHA-1
  - IV1: 50 6b 01 78 ff 6d 18 **90 20** 22 91 fd 3a de 38 71 b2 c6 65 ea
  - IV2: 50 6b 01 78 ff 6d 18 **91 a0** 22 91 fd 3a de 38 71 b2 c6 65 ea

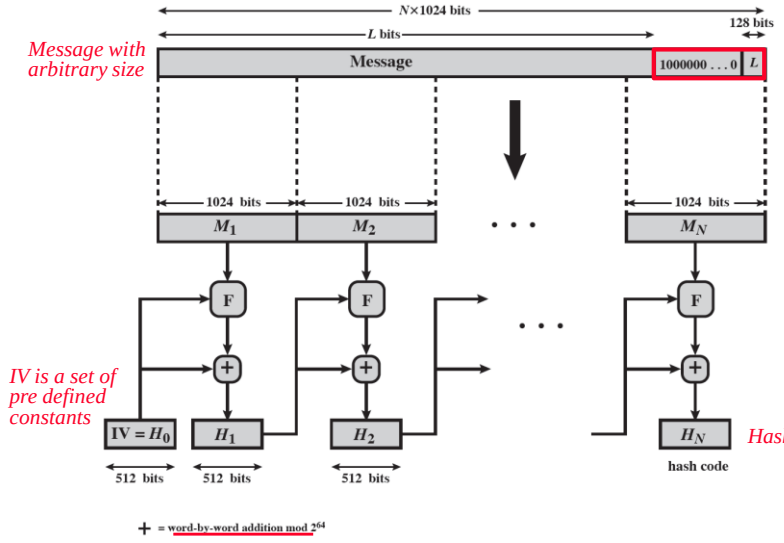
- **Effect:** Even if a freestart collision **does not immediately** give a standard collision, it could be used in multi-block collision search. The chaining value indeed is the compression function output of the previous block.

SHA – Secure Hash Algorithm

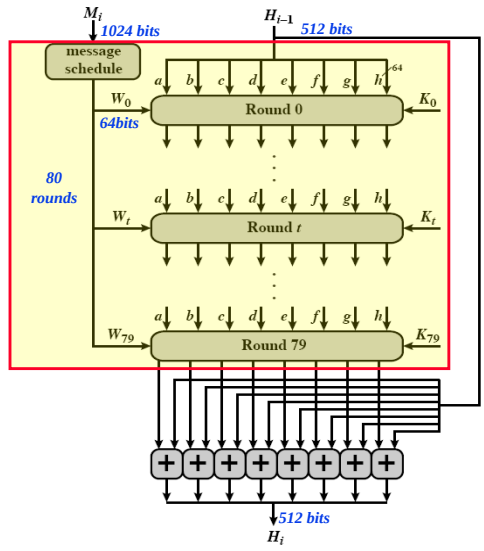
- It is a standard produced by NIST (National Institute of Standards and Technology) that has suffered several evolutions through the years
- The original algorithm SHA-0 was found insecure, and recently there was a successful attack to SHA-1
- Most initial versions follow the same structure and use similar operations
- More recently NIST has selected a new type of algorithm for SHA-3

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
		SHA-2			
Message digest size	160	224	256	384	512
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size (input)	512	512	512	1024	1024
Word size	32	32	32	64	64
Number of steps	80	64	64	80	80

SHA-512 : General Organization



Function *F*



Function F

Wi = 64 bits

a, b, …, h = 64 bits

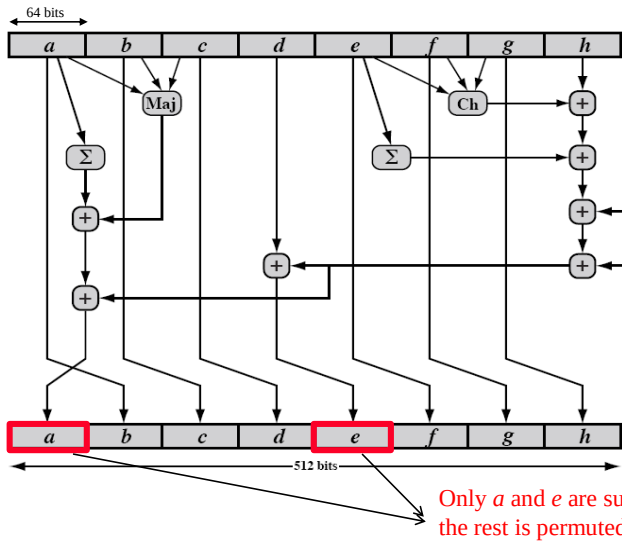
Ki = 64 bits constant

+ = addition modulo 2^64

Operations in each round based:

- *AND | NOT | XOR*
- *Rotation + Additions*
- *Permutations*

Basic Module



+ = addition module 2^64

Maj = (a&b) ⊕ (b&c) ⊕ (a&c)
true if majority of values are true

Ch(e,f,g) = (e&f) ⊕ (NOT e &g)
if e then f else g

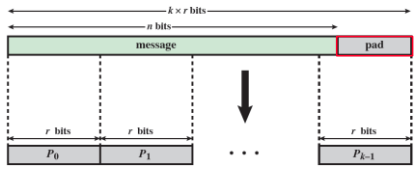
∑ = XOR of three different rotations to the right

SHA-3

- The SHA-3 algorithm resulted from a competition sponsored by NIST to find a new standard hash algorithm, where the winner was *Keccak* (in 2012)
- The algorithm is based on a **sponge construction** that follows the general structure of an hash function, but divides the processing in two phases
  - absorbing phase
  - squeezing phase
- Example padding
  - *Pad 10** : appends a single 1 and then the necessary 0 to make the message multiple of r
  - *Pad 10*1* : similar but ends with a last 1

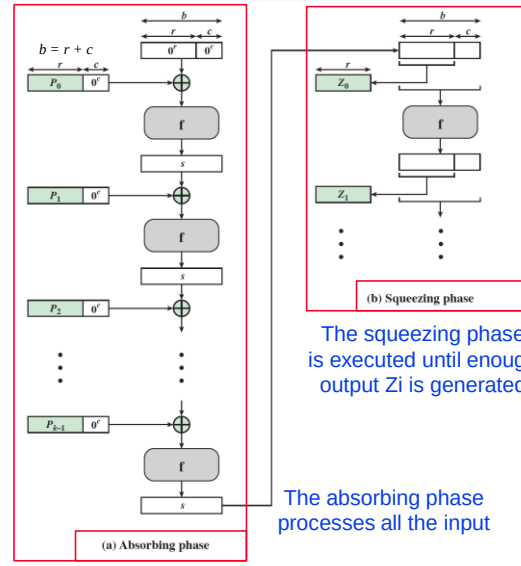
General processing

Initial preprocessing



r** is the bitrate, and also the **size of the block

b = r + c**, where **r is the bitrate**, and **c is the capacity



The squeezing phase is executed until enough output Zi is generated

The absorbing phase processes all the input

SHA-3 Parameters

- A sponge function is defined by three parameters
  pad: the padding algorithm
  f : internal function used to process each block
  r : the size in bits of the input blocks (also called the *bitrate*)

Note: All sizes and security levels are measured in bits

Message Digest Size	224	256	384	512
Message Size	no maximum	no maximum	no maximum	no maximum
Block Size (bitrate *r*)	1152	1088	832	576
Capacity *c*	448	512	768	1024
Word Size	64	64	64	64
Number of Rounds	24	24	24	24
Collision resistance	2112	2128	2192	2256
Second preimage resistance	2224	2256	2384	2512

MAC
Message Authentication Functions

(Some) Solutions for message authentication

- Symmetric message encryption
  - after decrypting the message, the receiver checks it to see if it is correct
  - *difficulty* : how does one recognize automatically that the correct content was obtained? What happens if the message can have arbitrary content?
- Error detection code (EDC) + symmetric encryption
  - C = E(K, M || EDC(M))
  - *difficulty* : EDC does not have cryptographic strength
  - *alternative* : add some known fields to the message
- Public key encryption
  - encrypt with the public key of receiver
  - *difficulty* : per se does not provide authentication
- Private key encryption
  - the sender uses his private key to encrypt the message
  - *difficulty* : same as above with symmetric encryption

Solution: add a Message Authentication Code

MAC

- In a generic way, the usual methods to create MACs
  - *I. MAC = H (M || K)*
  - *II. Use an block cipher mode of encryption, such as a (modified) CBC*
  - *III. MAC = E_K(H(M))*

- Security considerations for **Y = MAC(K, M)** with size of Y and K of n and k bits
  - find key by brute force attack (knowing pairs of M and Y)
    - » if n ≥ k: the effort is *in the order of 2^k*
    - » if n < k: then $2^n < 2^k$, which means that several keys will result in the same MAC; in this case, several pairs (M,Y) have to be tested => *effort larger 2^k*
  - generate Y without finding K (*discover Y for a given M?* or *finding M that matches a given Y?*)
    - » *effort in the order of 2^n*
  - Overall, the security is *in the order of* ***min(2^k, 2^n)***

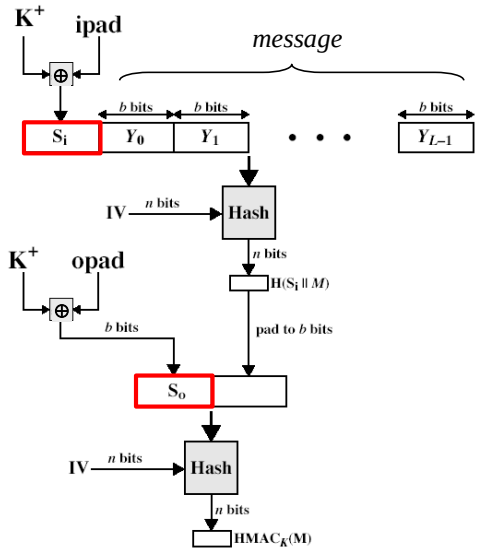
HMAC (RFC 2104 and FIPS 198)

- HMAC is an algorithm that generates MACs based on method I
- It was defined with the following objectives in mind
  - uses without modification an existing hash function
  - allows the substitution of the hash function with another one
  - performance similar to the underlying hash function
  - uses and manipulates the keys in a simple fashion
  - supports a simple cryptographic analysis (and is based on the security of the hash function)

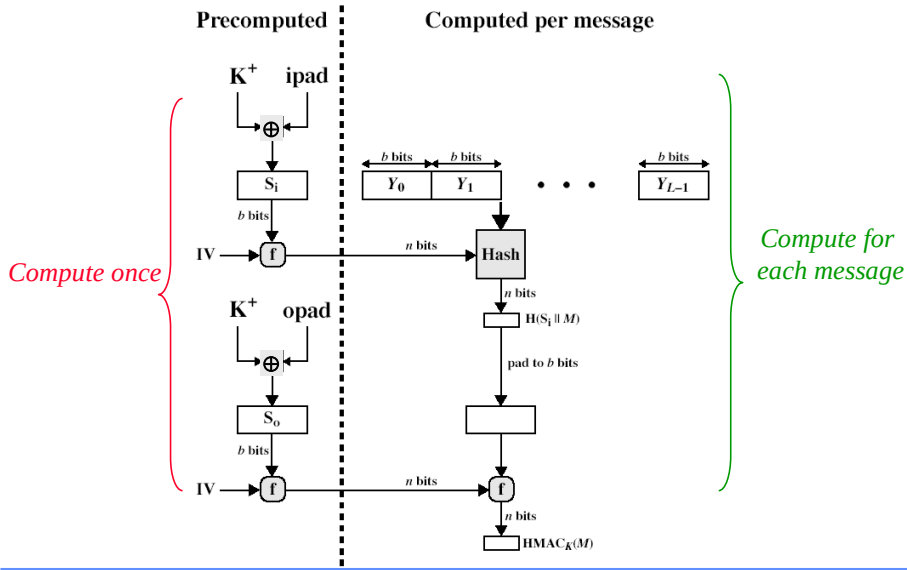
Algorithm

- K^+ is equal to key K with zeros added to the left to ensure an overall size of *b* bits
- *ipad* is 00110110 repeated *b/8* times
- *opad* is 01011100 repeated *b/8* times
- IV is the initialization vector defined by the hash function

- *ipad* and *opad* together with K^+ end up generating two pseudo-random initialization vectors (IV^*) from K



Another way to see the algorithm (more efficient)



Security of HMAC

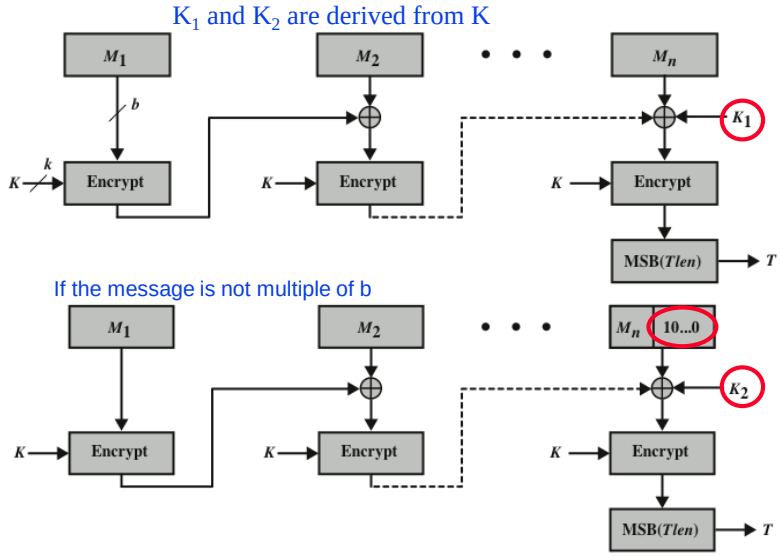
- The probability of attack to HMAC is *equivalent* to:

1. the probability that the adversary manages to **compute the result** of the compression function even if IV is random and secret
   (requires an effort in the order of 2^n, where n is the size of the IV, which means the size of the hash)

2. The probability that the adversary manages to **find a collision** in the hash function even if IV is random and secret
   (imagine a birthday attack; here the adversary cannot try different messages to see if they collide as he/she does not know the key; the adversary would need to observe at least $2^{n/2}$ blocks produced with the same key, where n is the size of the hash; for n = 128, in a 100 Gbit/s link it would take 1.500 years)

Cipher-Based Message Authentication Code (CMAC)

- *NIST 800-38B*
- *Uses AES or triple DES*
- *Works in two modes depending if the data is multiple of the block size*



Authenticated Encryption

Approaches for Encryption & Authentication

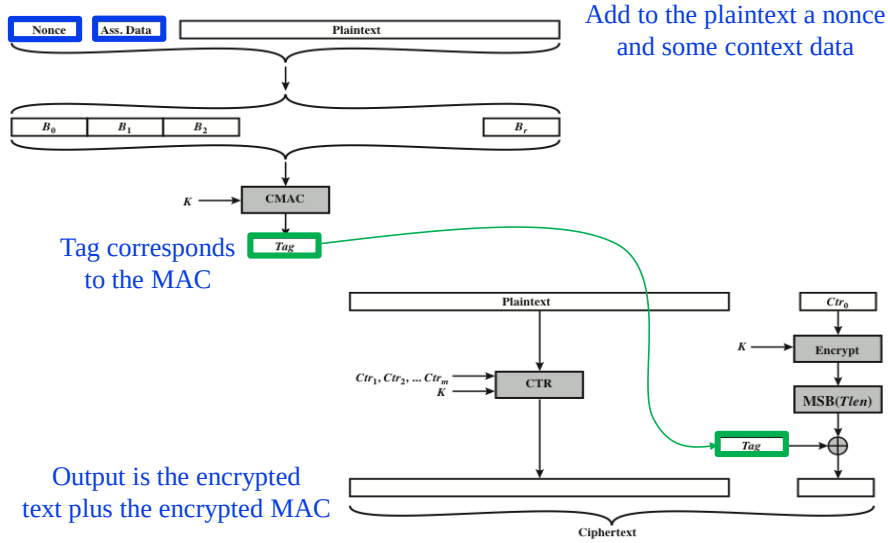
- Possible solutions
  - Hash followed by encryption: E(K, (M || h(M)))
    - » somewhat similar to approach by WEP
  - Authentication followed by encryption: E(K2, M || MAC(K1, M))
    - » used by SSL/TLS
  - Encryption followed by authentication: C = E(K2, M), *send* (C, MAC(K1, C))
    - » used by IPsec
  - Independent encrypt and authenticate: (E(K2, M), MAC(K1, M))
    - » used by SSH

  NOTE: *There are potential vulnerabilities in all of them! But the last three if properly implemented, they should be secure.*

Counter with Cipher Block Chaining-Message Authentication Code (CCM)

- CCM was standardized specifically to support the security requirements of IEEE 802.11 WiFi wireless local area networks
- For of authenticated encryption defined in NIST SP 800-38C
- Main algorithmic ingredients
  - AES encryption algorithm
  - CTR mode of operation
  - CMAC authentication algorithm (*that we saw before)*
- Single key K is used for both encryption and MAC algorithms

CCM in Operation



Add to the plaintext a nonce and some context data

Tag corresponds to the MAC

Output is the encrypted text plus the encrypted MAC

Bibliography

- Stallings, v6 :
  - RSA (274-298), DH(307-312), Elgamal (312-315)
  - Hash/MAC, chap 11 and 12