

Lecture 3: Authenticity

(Data Origin: MAC & Signature)

3.1 Crypto background 1: Public Key Cryptography (PKC)

3.1.1 RSA

3.1.2 Security of RSA

3.1.3 Improper RSA usage

3.2 Crypto background 2: Hash and keyed-hash

3.2.1 Hash

3.2.2 Keyed hash

3.3 Data integrity (Hash)

3.4 Data authenticity (MAC, signature)

3.4.1 MAC

3.4.2 Signature

3.5 Some attacks & pitfalls:

3.5.1 Birthday attack on hash

3.5.2 Using encryption for authenticity

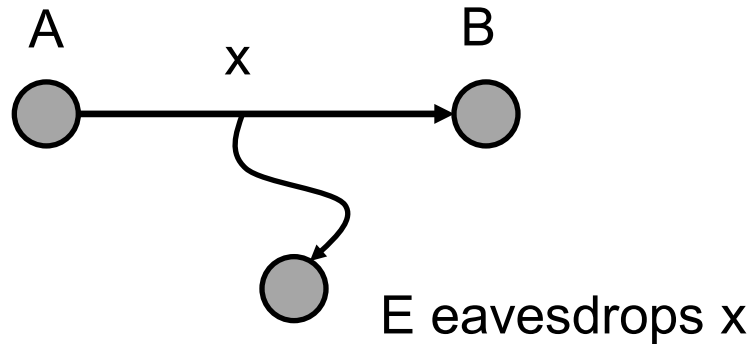
3.6 Application of hash: Password file protection (revisited)

Authentication (Review)

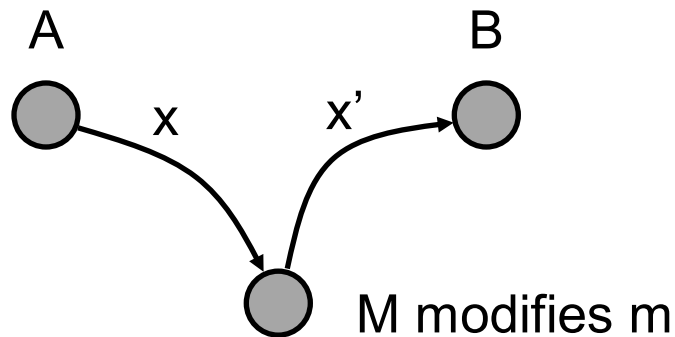
- ***Authentication***: the process of assuring that the communicating entity, or the origin of a piece of information, is the one that it claims to be
- **Two types** of authentication:
 - ***Entity authentication***:
 - For connection-oriented communication
 - Communicating entity is *an entity involved in a connection*
 - Mechanisms: password, challenge and response, biometrics
 - ***Data-origin authentication***:
 - For connection/less communication
 - Communicating entity is *the origin of a piece of information*
 - Data-origin authenticity implies data integrity (see next slides)
 - Mechanisms: **MAC** or **digital signature**

Threats to Confidentiality, Integrity & Authenticity: Illustration

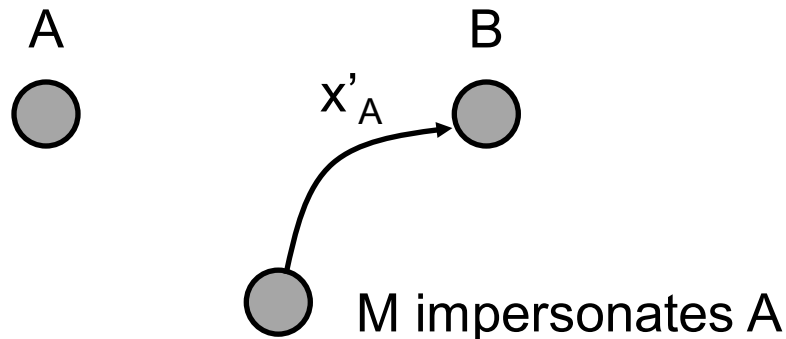
- Confidentiality:



- Integrity:



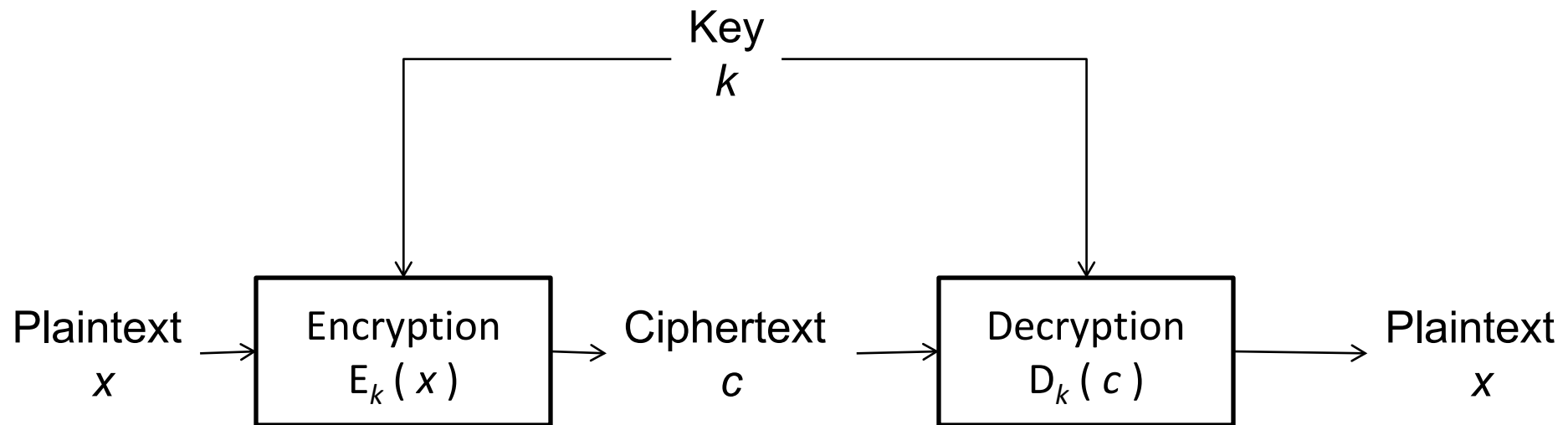
- Authenticity:**



3.1 Crypto Background 1: Public Key Cryptography (PKC)

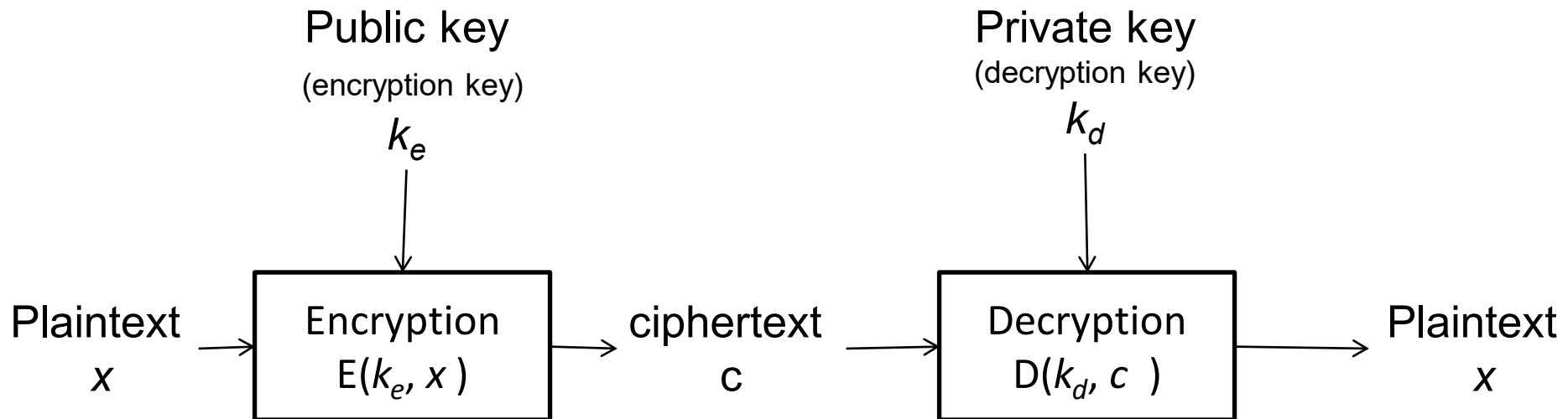
Overview: Symmetric-Key Scheme

A symmetric-key encryption scheme uses the same key for both encryption and decryption



Overview: Public-Key Scheme

A **public-key (asymmetric-key)** scheme uses **two different keys** for encryption and decryption



Note:

Very often, the *decryption key* consists of two parts $\langle k_1, k_2 \rangle$, where k_1 is part of the encryption/public key, whereas k_2 is kept secret.

When it is clear in the context, we also call k_2 the private key, although part of the private key may actually be made public.

Why is It Called “Public Key”?

- In a typical usage, the owner Alice **keeps her private key secret**, but **tells everyone of her public key** (e.g. by posting it in her Facebook page)
- A following **usage scenario** is possible
- Suppose Bob has a **plaintext x** for Alice.
Bob can encrypt it using **Alice’s *public key***, and post the **ciphertext c** on Alice’s Facebook
- Another entity, Eve, also obtains Alice’s public key and the posted ciphertext c from Alice’s Facebook page: yet, without Alice’s private key, Eve is unable to derive x
- Alice, with her secret ***private key***, can decrypt and obtain the plaintext x

Public-Key Encryption Scheme: More Formal Definition

- Can be more formally defined as **algorithms** (**G**, **E**, **D**) over **sets** (K_e , K_d , M , C):
 - K_e : set of all **public** keys (= public-key space)
 - K_d : set of all **private** keys (= private-key space)
 - M : set of all plaintexts (= plaintext/message space)
 - C : set of all ciphertexts (= ciphertext space)
- **G** (key-generation algorithm): generates **a key pair** (k_e, k_d), where k_e is publicly known whereas k_d is kept secret
- **E** (encryption algorithm): $K_e \times M \rightarrow C$
- **D** (decryption algorithm): $K_d \times C \rightarrow M$

Public-Key Encryption Scheme: More Formal Definition

- **Requirements:**
 - **Correctness:** For all $m \in \mathbf{M}$ and all $(k_e, k_d) \in \mathbf{K}_e \times \mathbf{K}_d$ outputted by G:
 $D(k_d, E(k_e, m)) = m$
 - **Efficiency:** G, E & D are “fast”, i.e. they run in polynomial time
 - **Security:** E() is a “one-way” function:
it can be efficiently evaluated (using the publicly-known k_e),
but its **inverse D() is computationally infeasible *without* k_d**
- Encryption is a ***trapdoor function* E()**:
 - Its inverse can only be efficiently evaluated **with** the ***trapdoor* k_d**
 - Without the trapdoor k_d , there’s an **asymmetric** computational requirement between E() and its inverse D()
- Public-key encryption scheme as a secure ***trapdoor permutation***

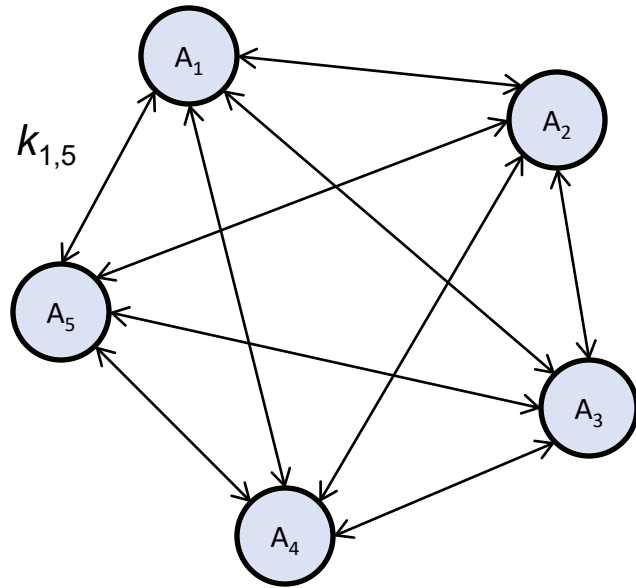
Security Requirements of a Public Key Scheme

- From the previous slide: the plaintext can be recovered ***only with*** the availability of the private key k_d
- In other words, the ***security requirement*** is: given the public key and the ciphertext, but not the private key, it is difficult to determine the plaintext
- (A more *refined formulation* of the requirement: without a knowledge of the private key, the ciphertext resembles a sequence of random values)
- The requirement **implies** that: it must be difficult to get the private key from the public key
- **Note:** Why can't we just require that "it must be difficult to get the private key from the public key"?
(See Tutorial 4)

Public-Key Scheme and Advantages in Key Management

- Suppose we have n entities, A_1, A_2, \dots, A_n :
 - Each of them can compute a pair of <private key, public key>
 - Each announces his/her public key, but keeps the private key secret
- Now, suppose A_i wants to encrypt a message m for A_j :
 - A_i can use A_j 's public key to encrypt m
 - By the property of PKC, **only A_j** can decrypt it
- If we use SKC, then any two entities must share a **secret key**:
 - Many keys are required especially if the no of entities is large (see the next slide)
 - Establishing all the secret keys is also difficult

Number of Keys Required (with n Entities)

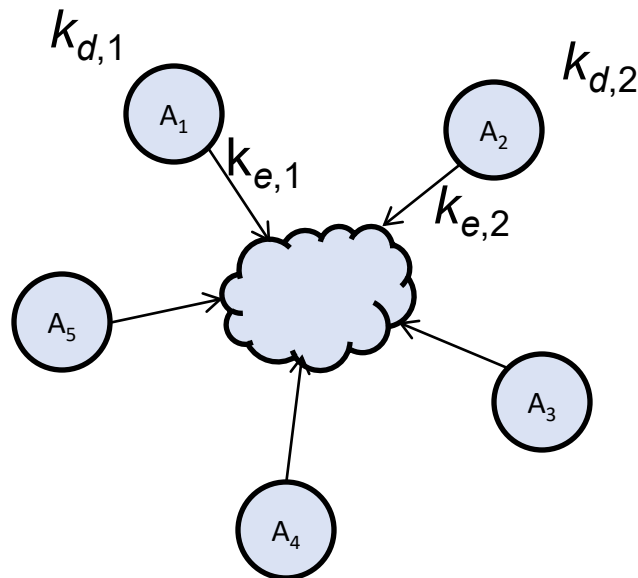


Symmetric-key setting:

Every pair of entities requires one key

Let $k_{i,j}$ be the key shared by A_i and A_j

Total number of keys needed: $n(n-1)/2$



Public-key setting:

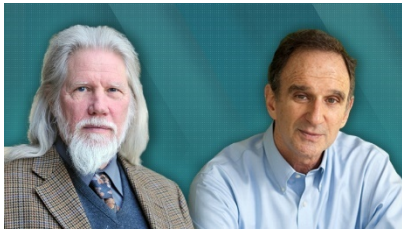
Each entity A_i publish its public key $k_{e,i}$
and keep its private key $k_{d,i}$

Total number of public keys: n

Total number of private keys: n

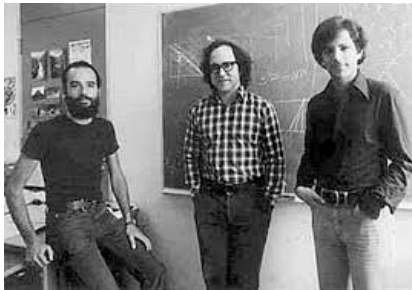
Popular PKC Schemes

- RSA: Key size $\sim 2,048$ bits (256 bytes)
- ElGamal: ElGamal can exploit techniques in Elliptic Curve Cryptography (ECC), thus reducing the key size to ~ 300 bits
- Historical notes:



The idea of an asymmetric public-private key cryptosystem is attributed to Whitfield Diffie and Martin Hellman, who published the concept in 1976

Diffie and Hellman, 2015 Turing Award winners



Ron Rivest, Adi Shamir, and Leonard Adleman proposed RSA algorithm in 1977

Rivest, Shamir and Adleman, 2002 Turing Award winners

3.1.1 RSA

Notes on Integer Representations

- We will introduce the “*classroom/textbook*” RSA:
 - The basic form of RSA
 - The variant used in practice is different: with padding, special considerations in choosing the parameters, etc.
- Many PKC systems represent the data (plaintext, ciphertext, key) as **integers**, and the algorithms are **arithmetic operations** on integers
- The integers can be represented using their **binary representations**
- When we say that a **key is 1024 bits**, we mean that it can be represented using 1024 bits **under binary representation**:
e.g. a 3-bit integer is a value from 0 to 7
- Note that the total number of 1024-bit integers is 2^{1024} :
an algorithm that exhaustively searches 1024-bit numbers is **infeasible**

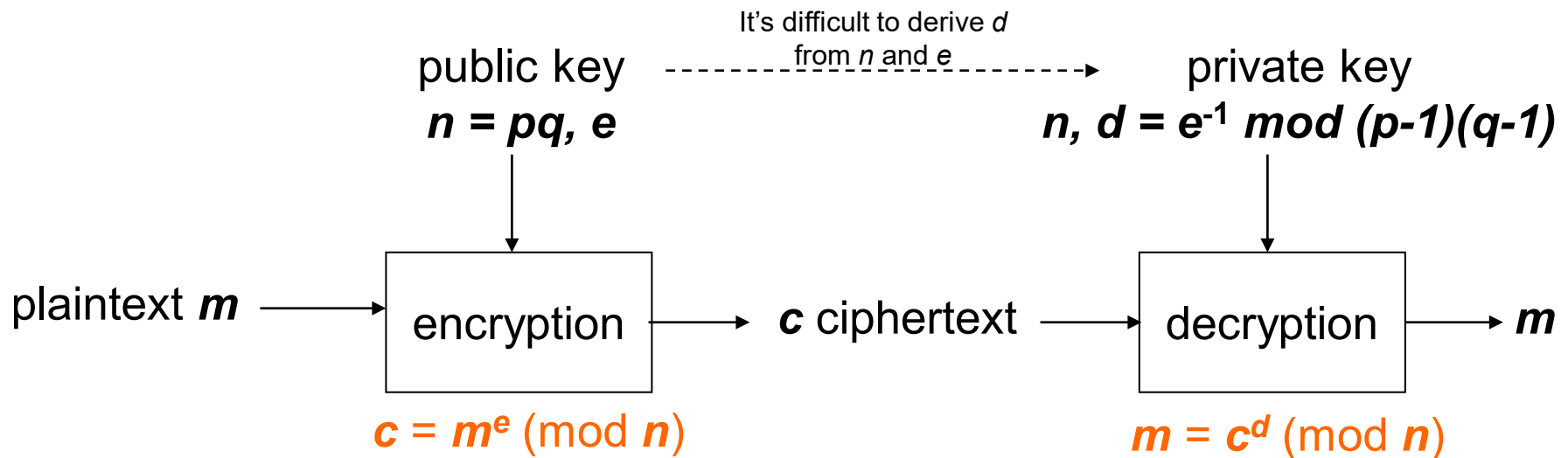
“Classroom” RSA: The Set-up

1. The owner randomly chooses 2 large primes p, q and computes $n = pq$ as the public **composite modulus**
(Note that the values of p and q must **not** be revealed to the public)
2. The owner randomly chooses an **encryption exponent** e s.t.
 $\gcd(e, \Phi(n)) = 1$, i.e. $e < \Phi(n)$ and e is relatively prime to $\Phi(n)$
(The term $\Phi(n) = (p-1)(q-1)$ is the **Euler’s totient function**, which is the number of integers $< n$ that are relatively prime to n .)
3. The owner determines the **decryption exponent** d , where:
$$de = 1 \pmod{\Phi(n)}, \text{ i.e. } d = e^{-1} \pmod{\Phi(n)}$$

(There is an **efficient algorithm** to find d when given e, p and q , but we won’t get into the details
see: http://shell.cas.usf.edu/~wclark/elem_num_th_book.pdf.
After determining d , the owner doesn’t need to keep p and q .)
4. The owner publishes (n, e) as her public key, and keep (n, d) as her private key

“Classroom” RSA: Encryption and Decryption

- Given the **public key** (n, e) and corresponding **private key** (n, d) , the public encryption & private decryption can be done as follows
- Encryption:** Given a message m , the ciphertext c is: $c = m^e \pmod{n}$
- Decryption:** Given a ciphertext c , the plaintext m is: $m = c^d \pmod{n}$



Note that given g and x , there's an efficient way of calculating $g^x \pmod{n}$, e.g. by using the **successive squaring** technique

(see: <https://www.math.lsu.edu/~adkins/m4023/SuccessivePowers.pdf>)

Correctness of RSA

- Note that for any positive $m < n$, and any pair of e and d :

$$\text{Decrypt}(d, \text{Encrypt}(e, m)) = m$$

- That is:

$$(m^e)^d = m \pmod{n}$$

Proof (sketch of):

The correctness depends on this property of modular arithmetic called **Euler's Theorem**:
if $n > 0$ and m is relatively prime to n , then: $m^{\Phi(n)} = 1 \pmod{n}$

This is the generalization of **Fermat's Little Theorem**, which says that:

if p is prime and m is relatively prime to p , then: $m^{p-1} = 1 \pmod{p}$

Recall that when $n = pq$, then $\Phi(n) = (p-1)(q-1)$

Now, recall also that $ed = 1 \pmod{\Phi(n)}$. Hence: $ed - 1 = k \cdot \Phi(n)$ or $ed = k \cdot \Phi(n) + 1$

We thus can show that, for all m relatively prime to n :

$$(m^e)^d = m^{ed} = m^{k \cdot \Phi(n) + 1} = m^{k \cdot \Phi(n)} \cdot m^1 = (m^{\Phi(n)})^k \cdot m = m \pmod{n}$$

In rare cases when m is not relatively prime to n , i.e. either $m = 0 \pmod{p}$

or $m = 0 \pmod{q}$, the congruence above is still true

(See also: [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)#Proof_using_Euler's_theorem](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Proof_using_Euler's_theorem))

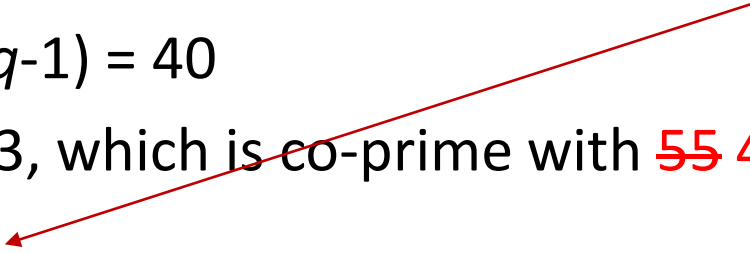
(Note: For more RSA mathematical background, see: [PF12.3];

Susanna Epp, "Discrete Mathematics with Applications", 4th ed, Chapter 8, Section 4;

or http://shell.cas.usf.edu/~wclark/elem_num_th_book.pdf)

Optional

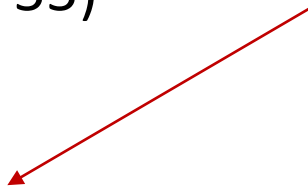
“Classroom” RSA: Example

- $p = 5, q = 11, n = 5 \times 11 = 55$
- $\Phi(n) = (p-1)(q-1) = 40$
- Suppose $e = 3$, which is co-prime with ~~55~~ 40
- Then $d = 27$ 
[you can check that $3 \times 27 = 81 = 1 \pmod{40}$]

There is an efficient algorithm to compute d from p, q, e (details are omitted)

- Suppose $m = 9$
- The encryption derives:
$$c = m^e \pmod{n}$$
$$= 9^3 = 14 \pmod{55}$$

There is an efficient algorithm that computes modular exponentiation (details are omitted)

- The decryption derives:
$$m = c^d \pmod{n}$$
$$= 14^{27} = 14^{16+8+2+1} = 9 \pmod{55}$$
 

Interchangeable Role of Encryption & Decryption Keys

- RSA has this **special property**:
we can also use the decryption key d to encrypt,
and then the encryption key e to decrypt
- Note that such a property may **not** hold in other public key scheme:
e.g. the ElGamal PKC (not covered in this module) doesn't have this property

Notes:

Some documents flip the role of encryption/decryption when describing RSA, i.e. using public key to decrypt. This could be very confusing.

Take a special note of this.

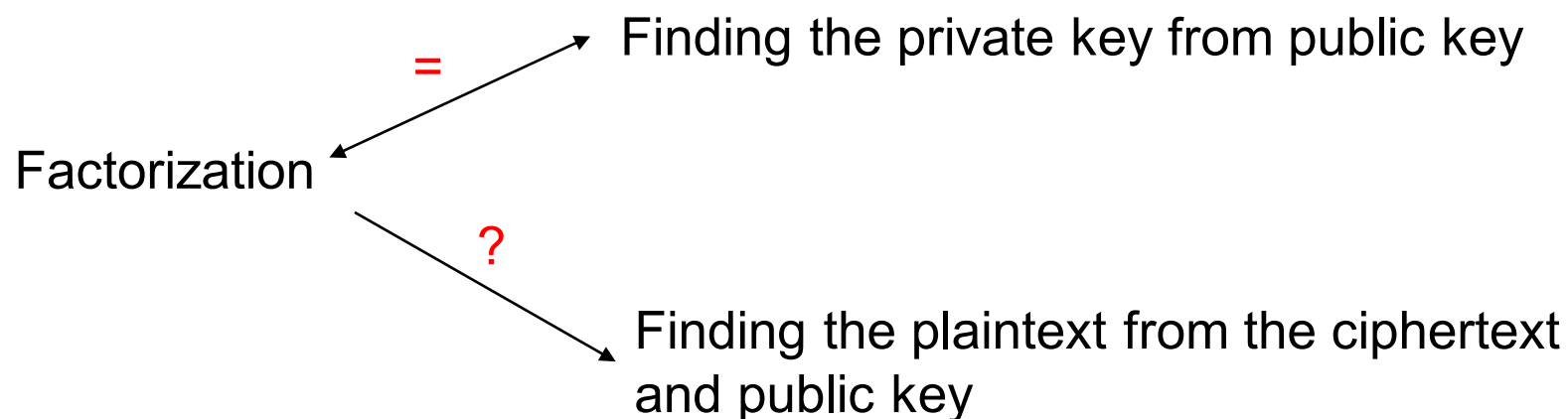
Algorithmic (Efficiency) Issues

- How can we know that **all the steps** in RSA are efficient (except performing a decryption without d)?
- How to find a **random prime** (Step 1 in the set-up)?
 - Randomly pick a number, and test whether it is a prime
 - **Primality testing** can be done efficiently
 - Since there are **many primes**, the step is likely to find one
 - Note that, as we shall see later, some primes are weak and lead to an attack
 - Hence, use “**strong**” **primes** to avoid this issue
- How to efficiently compute d from e , p and q (Step 3 in the set-up)?
 - We can use the *Extended Euclidean algorithm*
 - See: <http://shell.cas.usf.edu/~wclark/elementarybook.pdf>
- How to easily perform the **encryption** and **decryption** operations?
 - We need to perform **modular exponentiation** operations: $m^e \pmod{n}$, $c^d \pmod{n}$
 - There is an **efficient algorithm** to compute a modular exponentiation
 - In practice, either one of e or d (i.e. both can't be small) is intentionally chosen to be **small**, so that the respective encryption or decryption can be done very fast

3.1.2 Security of RSA

Security of RSA

- It can be shown that, the problem of getting the RSA private key from public key is *as difficult as* the problem of factorizing n
- However, it **not known** whether the problem of getting the plaintext from the ciphertext and public key (the **RSA problem**) is *as difficult as* factorization, i.e. it **could be easier**



State-of-the-Art on Factorization

- For more info:
http://en.wikipedia.org/wiki/Integer_factorization
- A 640-bit number (RSA-640) was successfully factored in Nov 2005 using approximately 30 2.2GHz-Opteron-CPU years:
 - It is computed over five months using multiple machines
 - See also <https://mathworld.wolfram.com/news/2005-11-08/rsa-640/>
- A 768-bit number (RSA-768) was factored in Dec 2009 using hundreds of machines over 2 years
- See NIST recommendation about the key length for RSA

State-of-the-Art on Factorization

- Here is the 640-bit number:

31074182404900437213507500358885679300373460228427
27545720161948823206440518081504556346829671723286
78243791627283803341547107310850191954852900733772
4822783525742386454014691736602477652346609

=

16347336458092538484431338838650908598417836700330
92312181110852389333100104508151212118167511579

*

1900871281664822113126851573935413975471896789968
515493666638539088027103802104498957191261465571

State-of-the-Art on Factorization

- Here is the 768-bit number:

12301866845301177551304949583849627207728535695953347921973224521
51726400507263657518745202199786469389956474942774063845925192557
32630345373154826850791702612214291346167042921431160222124047927
4737794080665351419597459856902143413

=

34780716989568987860441698482126908177047949837137685689124313889
82883793878002287614711652531743087737814467999489

*

36746043666799590428244633799627952632279158164343087642676032283
815739666511279233373417143396810270092798736308917

Source:

T Kleinjung et al., *Factorization of a 768-bit RSA modulus*, 2010, <https://eprint.iacr.org/2010/006.pdf>

Post-Quantum Cryptography

- A quantum computer can factorize and perform “*discrete log*” in **polynomial time**:
 - In 2001, a 7-*qubit* quantum computer was built to factor 15, carried out by IBM using NMR
(See: http://domino.watson.ibm.com/comm/pr.nsf/pages/news.20011219_quantum.html)
- Hence, both RSA and “discrete log” based PKC will be **broken** with a quantum computer:
 - See Shor's algorithm (https://en.wikipedia.org/wiki/Shor%27s_algorithm)
- ***Post-Quantum Cryptography:***
PKC schemes that are secure against quantum computer
- ***Lattice-based cryptography:***
 - Based on this hard problem:
given the “basis” of lattice, it is computational hard to find a short lattice point
 - No good candidate yet
- ***Multivariate cryptography:***
 - Given a multivariate polynomial, it is difficult to find the solution (under modulo p)
 - No secure construction yet

Padding of RSA

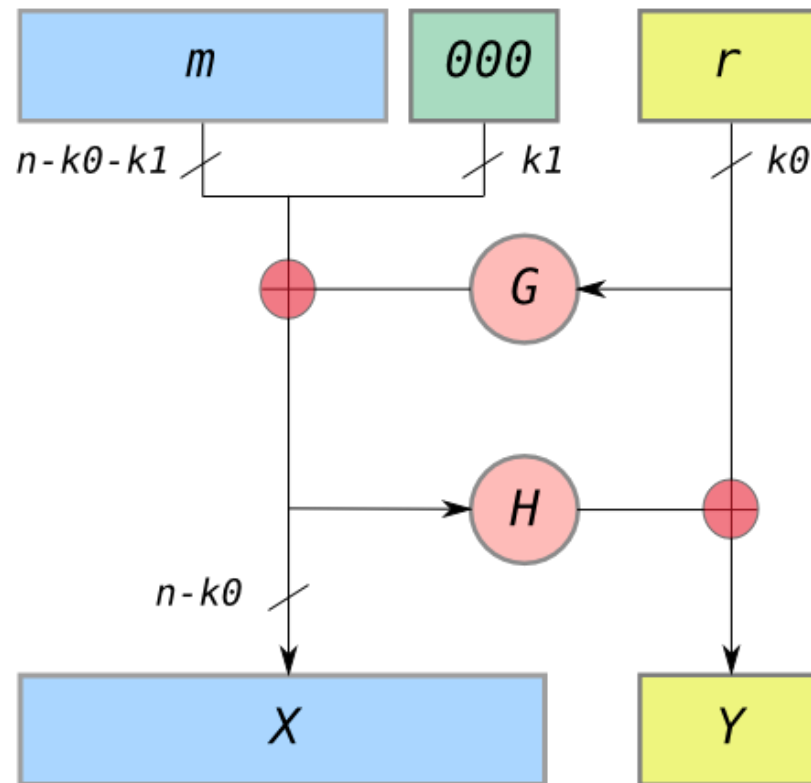
- Similar to symmetric-key encryption, some form of IV is required so that encryption gets **randomized**
- Hence, additional mechanisms are added to “classroom” RSA
- Furthermore, RSA has some interesting properties, e.g. *homomorphic property*:
 - These properties are useful in some applications (e.g. blind signature, encrypted domain processing)
 - But they lead to **attacks** and **information leakage**
 - Such properties have to be destroyed for security
- The Public-Key Cryptography Standards standard PKCS#1 adds “optimal padding” to achieve the above
 - See http://en.wikipedia.org/wiki/PKCS_1

RSA with Padding (Just to Illustrate How Complicated It Could be)

Optional

Optimal Asymmetric Encryption Padding (OAEP) in PKCS#1 v2:

- The encoded message can then be encrypted with RSA
- The deterministic property of RSA is now avoided by the encoding



Source: Wikipedia

- It provides the **basic definitions** of and **recommendations** for implementing the RSA algorithm for public-key cryptography
- The latest version is **2.2** (2012-10-27)
- ***Primitives:***
 - **I2OSP** - Integer to Octet String Primitive:
Converts a (potentially very large) non-negative integer into a sequence of bytes (octet string)
 - **OS2IP** - Octet String to Integer Primitive:
Interprets a sequence of bytes as a non-negative integer
 - **RSAEP** - RSA Encryption Primitive: Encrypts a message using a public key
 - **RSADP** - RSA Decryption Primitive: Decrypts ciphertext using a private key
 - **RSASP1** - RSA Signature Primitive 1:
Creates a signature over a message using a private key
 - **RSAVP1** - RSA Verification Primitive 1:
Verifies a signature is for a message using a public key

- **Schemes:** define higher level algorithms or uses of the primitives so they achieve certain security goals
- Schemes for **encryption & decryption:**
 - **RSOAES-OAEP:** improved encryption/decryption scheme based on the Optimal Asymmetric Encryption Padding scheme
 - **RSOAES-PKCS1-v1_5:** older encryption/decryption scheme as first standardized in version 1.5 of PKCS #1
- Schemes for dealing with **signatures:**
 - **RSASSA-PSS:** improved Probabilistic Signature Scheme with Appendix
 - **RSASSA-PKCS1-v1_5:** old Signature Scheme with Appendix as first standardized in version 1.5 of PKCS #1
- **Encoding methods** for two signature schemes:
 - **EMSA-PSS:** encoding for signature appendix, probabilistic signature scheme
 - **EMSA-PKCS1-v1_5:** encoding for signature appendix as first standardized in version 1.5 of PKCS #1

3.1.3 Improper RSA Usage

Pitfall: Using RSA in the Symmetric-Key Setting

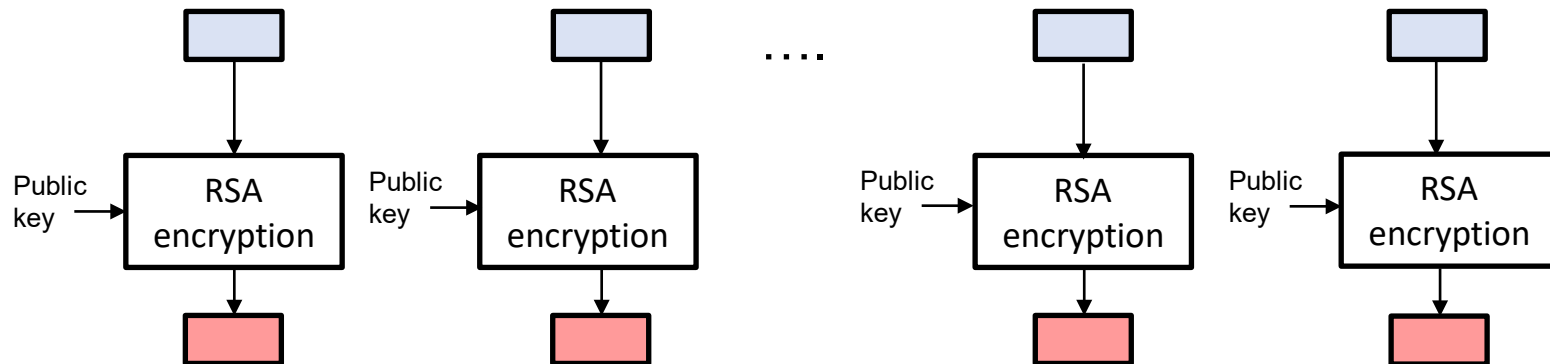
Consider a sample scenario:

“For his Final Year Project, Bob is tasked to write an app that employs an end-to-end encryption to send images from a mobile phone to another mobile phone over WiFi. The sender and recipient use SMS to establish the required symmetric key.”

- Bob feels that RSA is cool 😊
- Instead of employing AES, Bob employs RSA as the encryption scheme: the public/secret key pair is treated as the symmetric key
- Bob claims that RSA is “more secure” than AES: it can be proved to be as difficult as factorization, which is believed to be hard

Pitfall: Using RSA in the Symmetric-Key Setting

- Bob's RSA implementation:
To encrypt a large image (say 1MB),
Bob divides the file into chunks of 128 bytes,
and then apply RSA to each chunk
- Implementation diagram:



Issue 1: Efficiency/Performance

RSA is significantly slower than AES:

- Comparing with the same key strength (e.g. by following NIST recommendation): 128-bit AES \approx 3072-bit RSA
- Crypto++ Benchmarks
(<https://www.cryptopp.com/benchmarks.html>): a C++ library

Algorithm	MiB/Second	Cycles Per Byte	Microseconds to Setup Key and IV	Cycles to Setup Key and IV
AES/GCM (2K tables)	102	17.2	2.946	5391
AES/GCM (64K tables)	108	16.1	11.546	21130
AES/CCM	61	28.6	0.888	1625
AES/EAX	61	28.8	1.757	3216
AES/CTR (128-bit key)	139	12.6	0.698	1277
AES/CTR (192-bit key)	113	15.4	0.707	1293
AES/CTR (256-bit key)	96	18.2	0.756	1383

In different order of magnitude!

Operation	Milliseconds/Operation	Megacycles/Operation
RSA 1024 Encryption	0.08	0.14
RSA 1024 Decryption	1.46	2.68
RSA 2048 Encryption	0.16	0.29
RSA 2048 Decryption	6.08	11.12

Solution

When a large file F is to be **encrypted** under the public-key setting, for efficiency, the following steps are carried out:

1. Randomly choose an AES key k
2. Encrypt F using AES with k as the key, to produce the ciphertext C
3. Encrypt k using RSA to produce the ciphertext q
4. The final ciphertext consists of two components: (q, C)

Question: What about decryption steps?

Remarks on PKC Strengths

- The main strength of RSA and PKC is the “public-key” setting, which allows an entity in the public to perform an encryption **without** a (pre-established) pair-wise **secret key**
- This “secret-key-less” feature is also very useful for providing **authentication**: more in our module later
- In practice, PKC is rarely used to encrypt a large data file (as already discussed earlier)

Question:

Assuming that a PKC encryption scheme on average takes 1M cycles to decrypt 16 bytes.

How long does it take to decrypt a 4 GBytes movie on a single-core 4GHz chip?

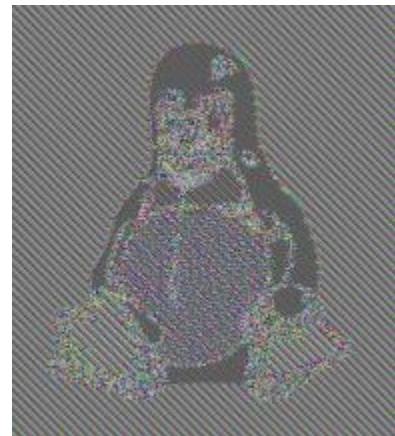
(For comparison, note that RSA-OAEP takes 42M cycles per invocation [Gollmann] pg272.)

Issue 2: ECB with RSA

- Dividing the plaintext into pieces, and independently apply encryption to each of them **could leak** important information!
- Suppose the image below is divided into blocks, and encrypted with some **deterministic** encryption scheme using the same key
- Since it is deterministic, any two plaintext blocks that are exactly the same (e.g. from the white background) will be encrypted into the same ciphertext
- Note that this issue applies to symmetric-key encryption too



Plaintext



Ciphertext

Issue 3: Security of RSA

- RSA is not necessary “more secure” than AES
- It can be shown that, getting ***the private key from the public key*** is as difficult as factorization
- But, it is not known whether the problem of getting ***the plaintext from the ciphertext and public key*** (i.e. RSA Problem) is as difficult as factorization
- In a certain sense, similar to AES, there is no *rigorous proof* that RSA is “secure” in protecting the ciphertext
- As mentioned before, the “classroom” RSA has to be **modified** so that different encryptions of the same plaintext lead to different ciphertexts (i.e. probabilistic)
- Such modification is not straightforward (e.g. PKCS#1)
- Factorization can be efficiently done by a quantum computer:
 - RSA is broken by a quantum computer
 - It is not clear how a quantum computer can be used to break AES

Other PKCs: Some Additional Notes

- ElGamal Encryption:
 - ElGamal is a ***discrete log based*** encryption, whereas RSA is ***factorization-based***
 - There are many choices of algebraic groups for discrete log-based encryption, e.g. Elliptic Curve
 - Those using Elliptic Curve are often called **Elliptic Curve Cryptography (ECC)**
 - Certain choices of ECC reduce the key size, e.g. ~300 bits for the equivalent of 2048-bit RSA
- Paillier Encryption:
 - Paillier Encryption is also discrete-log based
 - ElGamal can be easily modified so that it is homomorphic w.r.t. multiplication, whereas Paillier is homomorphic w.r.t. addition
- If an algorithm can efficiently solve the ***Discrete Log*** problem, then it can easily break discrete-log based encryptions:
 - In this module, we omit details of the discrete log problem

3.2 Crypto Background 2: Hash and Keyed-Hash

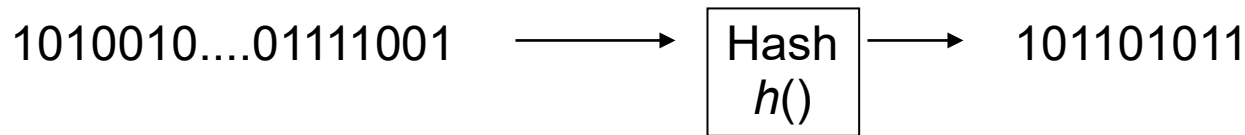
3.2.1 (Unkeyed) Hash

Hash (With No Secret Key Involved)

- A **(cryptographic) hash** is a function that takes an *arbitrarily long* message as input, and outputs a *fixed-size* (say 160 bits) **digest**

Arbitrarily long *message*

Fixed size *digest*



- Note that a hash function takes **no** key, i.e. it is **key-less**
- A cryptographic hash must meet some **security requirements** (listed on the next slide)
- Do not use **non-cryptographic** hash functions, such as:
 - Taking selected bits from the data
 - CRC checksum
(see https://en.wikipedia.org/wiki/Cyclic_redundancy_check)

Cryptographic Hash: More Formal Definition

- A hash function takes a message m and produces n -bit digest:
 $H : \{0,1\}^* \rightarrow \{0,1\}^n$
- Requirements:
 - **Efficiency:** Given m , it is computationally efficient to compute $y = h(m)$
 - **Pre-image resistance (“one-way”):**
Given y , it is computationally infeasible to find a m such that $h(m) = y$
 - **Collision-resistance:**
It is computationally infeasible to find *any* pair of messages (m_1, m_2) where $m_1 \neq m_2$, such that $h(m_1) = h(m_2)$,
i.e. both messages have the same digest
- Collision resistance \rightarrow pre-image resistance:
 - This also means: If we have an algorithm to attack the pre-image problem, then we can produce collisions
 - See Tutorial 4

Some Examples (SHA-1)

- **SHA-1** with 160-bit digest
- SHA1("The quick brown fox jumps over the lazy dog"):
 - Hexadecimal: 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
 - **Base64** binary to ASCII text encoding:
L9ThxnotKPzthJ7hu3bnORuT6xl=
- SHA1("The quick brown fox jumps over the lazy **c**og"):
 - Hexadecimal: de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3
 - Base64 binary to ASCII text encoding:
3p8sf9JeGzr60+haC9F9mxANtLM=
- Source: Wikipedia

Base64 Encoding

Base64: a **binary-to-text encoding** to represent binary data in an ASCII string

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Source: Wikipedia

Base64 Encoding

Source	Text (ASCII)	M								a								n							
	Octets	77 (0x4d)								97 (0x61)								110 (0x6e)							
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Base64 encoded	Sextets	19								22								5							
	Character	T								W								F							
	Octets	84 (0x54)								87 (0x57)								70 (0x46)							

Source	Text (ASCII)	M								a															
	Octets	77 (0x4d)								97 (0x61)															
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	0						
Base64 encoded	Sextets	19								22								4							
	Character	T								W								E							
	Octets	84 (0x54)								87 (0x57)								69 (0x45)							

Source: Wikipedia

Popular Hash

- SHA-0, SHA-1, SHA-2, SHA-3

Some historical notes:

- **SHA-0** was published by NIST in **1993**.
It produces a 160-bits digest.
It was **withdrawn** shortly after publication and superseded by the revised version SHA-1 in **1995**.
- In **1998**, an attack that finds collision of SHA-0 in 2^{61} operations was discovered.
(Note that simply using the straight forward birthday attack, a collision can be found in $2^{160/2} = 2^{80}$ operations.)
In **2004**, a collision was found, using 80,000 CPU hours.
In **2005**, Wang Xiaoyun et al. (Shandong University) gave an attack that can find collision in 2^{39} operations.

Popular Hash (Historical Notes)

- **SHA-1** is a popular standard. It produces 160-bits message digest. It is employed in SSL, SSH, etc.
- In **2005**, Xiaoyun Wang et al. gave a method of finding collision in SHA-1 using 2^{69} operations, which was later improved to 2^{63} .
In Feb **2017**, researchers from CWI and Google announced the first successful SHA1 collision (<https://shattered.io>): see the next 2 slides
- In **2001**, NIST published SHA-224, SHA-256, SHA-384, SHA-512, collectively known as **SHA-2**.
The number in the name indicates the digest length.
- No known attack on full SHA-2 but there are known attacks on “**partial**” SHA-2, for e.g. attack on a 41-round SHA-256 (whereas the full SHA-256 takes 64 rounds).
- In Nov **2007**, NIST called for proposal of **SHA-3**.
In Oct **2012**, NIST announced the winner, Keccak (pronounced “catch-ack”).
(See: NIST announcement <http://www.nist.gov/itl/csd/sha-100212.cfm>)

Recent Successful Collision Attacks on SHA-1 (Feb 2017)



Google Security Blog

The latest news and insights from Google on security and safety on the Internet

SHattered
(shattered.io)
Feb 23, 2017



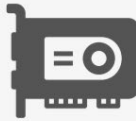
Announcing the first SHA1 collision

February 23, 2017

Posted by Marc Stevens (CWI Amsterdam), Elie Bursztein (Google), Pierre Karpman (CWI Amsterdam), Ange Albertini (Google), Yarik Markov (Google), Alex Petit Bianco (Google), Clement Baisse (Google)

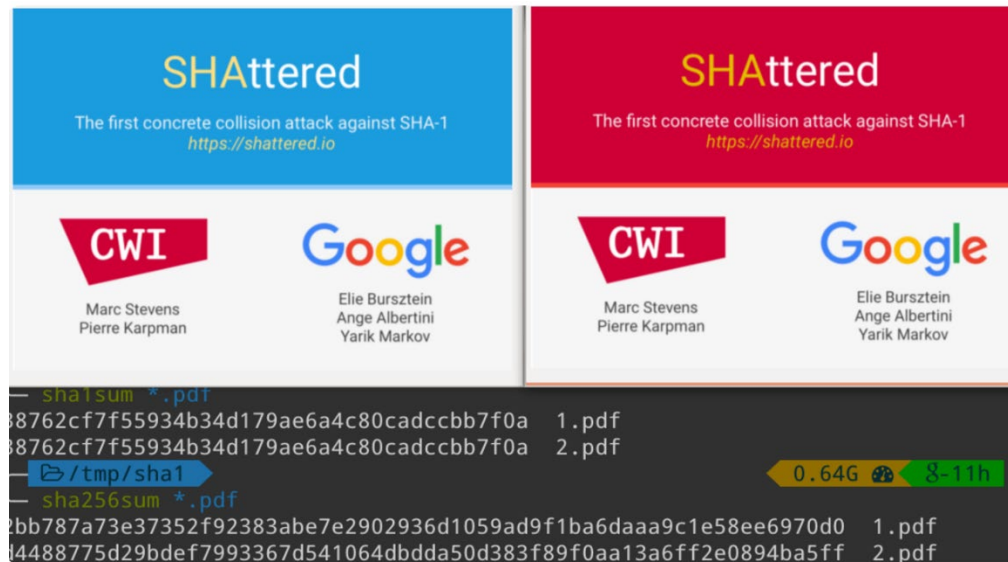
Cryptographic hash functions like SHA-1 are a cryptographer's swiss army knife. You'll find that hashes play a role in browser security, managing code repositories, or even just detecting duplicate files in storage. Hash functions compress large amounts of data into a small message digest. As a cryptographic requirement for wide-spread use, finding two messages that lead to the same digest should be computationally infeasible. Over time however, this requirement can fail due to [attacks on the mathematical underpinnings](#) of hash functions or to increases in computational power.

Today, more than 20 years after of SHA-1 was first introduced, we are announcing the first practical technique for generating a collision. This represents the culmination of two years of research that sprung from a collaboration between the [CWI Institute in Amsterdam](#) and Google. We've summarized how we went about generating a collision below. As a proof of the attack, we are [releasing two PDFs](#) that have identical SHA-1

 MD5 1 smartphone 30 sec	 SHA-1 Shattered 110 GPU 1 year	 SHA-1 Bruteforce 12,000,000 GPU 1 year
--	---	---

Recent Successful Collision Attacks on SHA-1 (Feb 2017)

- Done by a team from CWI and Google
- Two PDF files with the same hash values as attack proof:
 - <https://shattered.io/static/shattered-1.pdf>
 - <https://shattered.io/static/shattered-2.pdf>



- Defense mechanisms:
 - **Use SHA-256 or SHA-3 as replacement**
 - Visit shattered.io to test your PDF (Read <https://shattered.io/>)

Other Popular (but Obsolete) Hash

- MD5

Some historical notes:

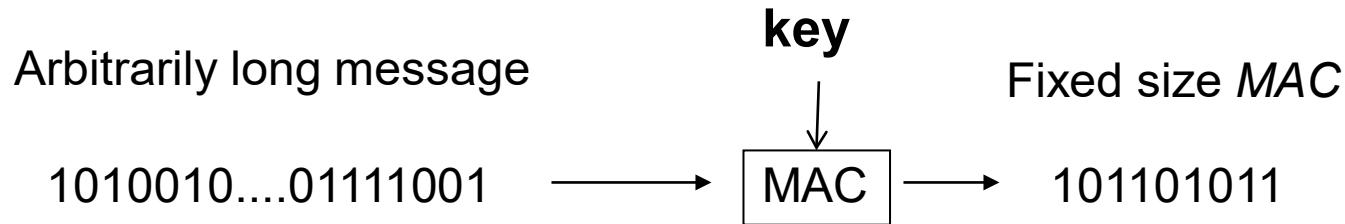
- Designed by Rivest, who also invented MD, MD2, MD3, MD4, MD6.
- MD6 was submitted to NIST SHA-3 competition, but did not advance to the second round of the competition.
- MD5 was widely used. It produces 128-bit digest.
- In **1996**, Dobbertin announced a collision of the compress function of MD5.
- In **2004**, collision was announced by Xiaoyu Wang et al.
The attack was reported to take one hour.
- In **2006**, Klima give an algorithm that can find collision within one minute on a single notebook!

Security implication: ***Do not*** use MD5!

3.2.1 Keyed Hash

MAC (a.k.a Keyed-Hash, with a Secret Key *Involved*)

- A ***keyed-hash*** is a function that takes an arbitrarily long message and a **secret key** as input, and outputs a fixed-size (say 160 bits) ***MAC (Message Authentication Code)***



- Note that now a correct MAC can only be generated by someone who **knows** the key

MAC: More Formal Definition

- A MAC is defined by (**Gen**, **Mac**, **Vrfy**):
 - **Gen**: Outputs a key k
 - **Mac**: Takes as input key k and message $m \in \{0,1\}^*$; and outputs tag $t = \text{Mac}_k(m)$
 - **Vrfy** (Verification): Takes key k , message m , and tag t as input; outputs 1 ("true"/"accept") if $\text{Mac}_k(m) = t$, or 0 ("false"/"reject") otherwise
- **Correctness** requirement:
for all k outputted by Gen and all m , $\text{Vrfy}_k(m, \text{Mac}_k(m)) = 1$
- **Security** requirement:
 - **Threat model**: adaptive chosen message attack
 - **Security goal**: "*existential forgery*", i.e. attacker cannot forge a valid tag on *any* message not authenticated by the sender
 - That is, after seeing multiple valid pairs of messages and their corresponding MACs, it is difficult for the attacker to forge the MAC of an *unseen* message (The precise formulation is quite involved, and with many variants. A higher level crypto module CS4236 would cover this.)
- How about *replay attacks*? Are they prevented by MAC?

Optional

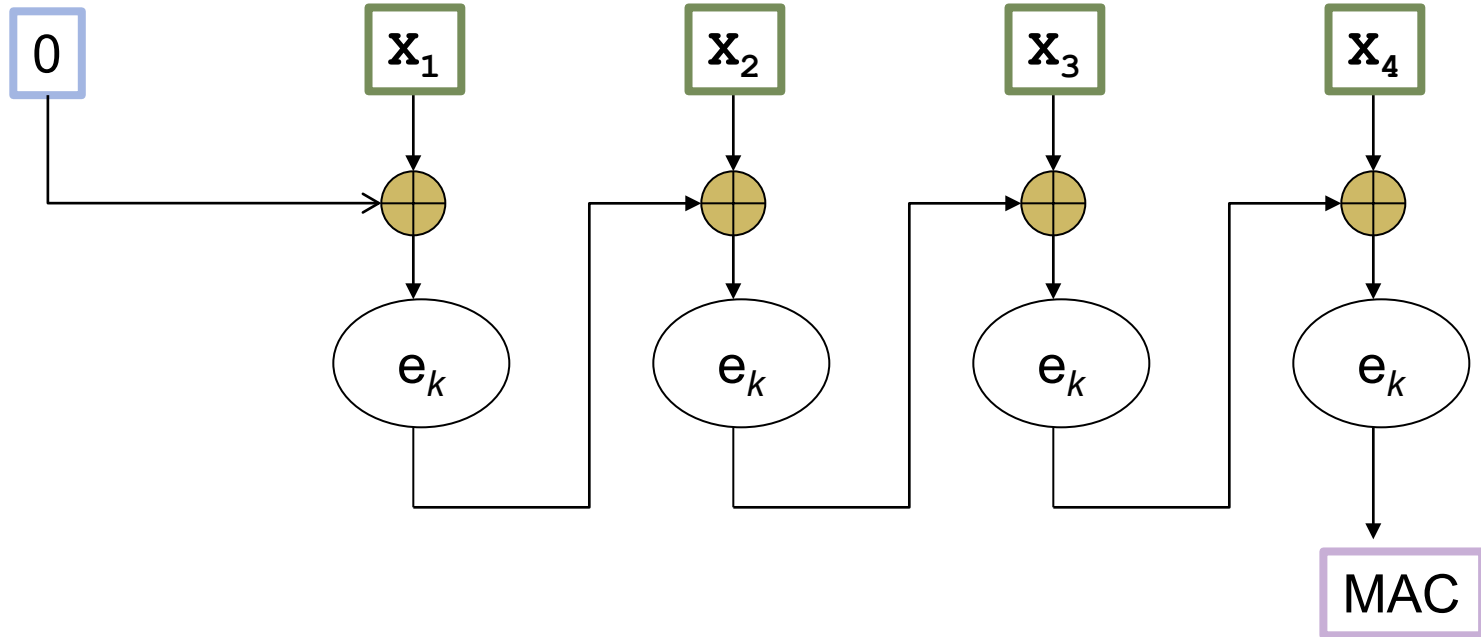
Popular Keyed-Hash (MAC)

- **CBC-MAC:**
 - Based on AES, a **block cipher**, operated under CBC mode
- **HMAC:**
 - Based on any iterative cryptographic **hash function** (e.g. SHA-1, SHA-2)
 - Hashed-based MAC
 - Standardized under RFC 2104 (<http://tools.ietf.org/html/rfc2104>)

CBC-MAC

Optional

Initial Value (IV)



$$\text{HMAC}_k(x) = \text{SHA-1}((K \oplus \text{opad}) || \text{SHA-1}((K \oplus \text{ipad}) || x))$$

where:

$\text{opad} =$ 3636...36 (outer pad)

$\text{ipad} =$ 5c5c...5c (inner pad)

(**Note**: the values above are in hexadecimal)

3.3 Data Integrity: Hash (Without Secret Keys)

Authenticity and Integrity Problems

Recall the examples given in last lecture:

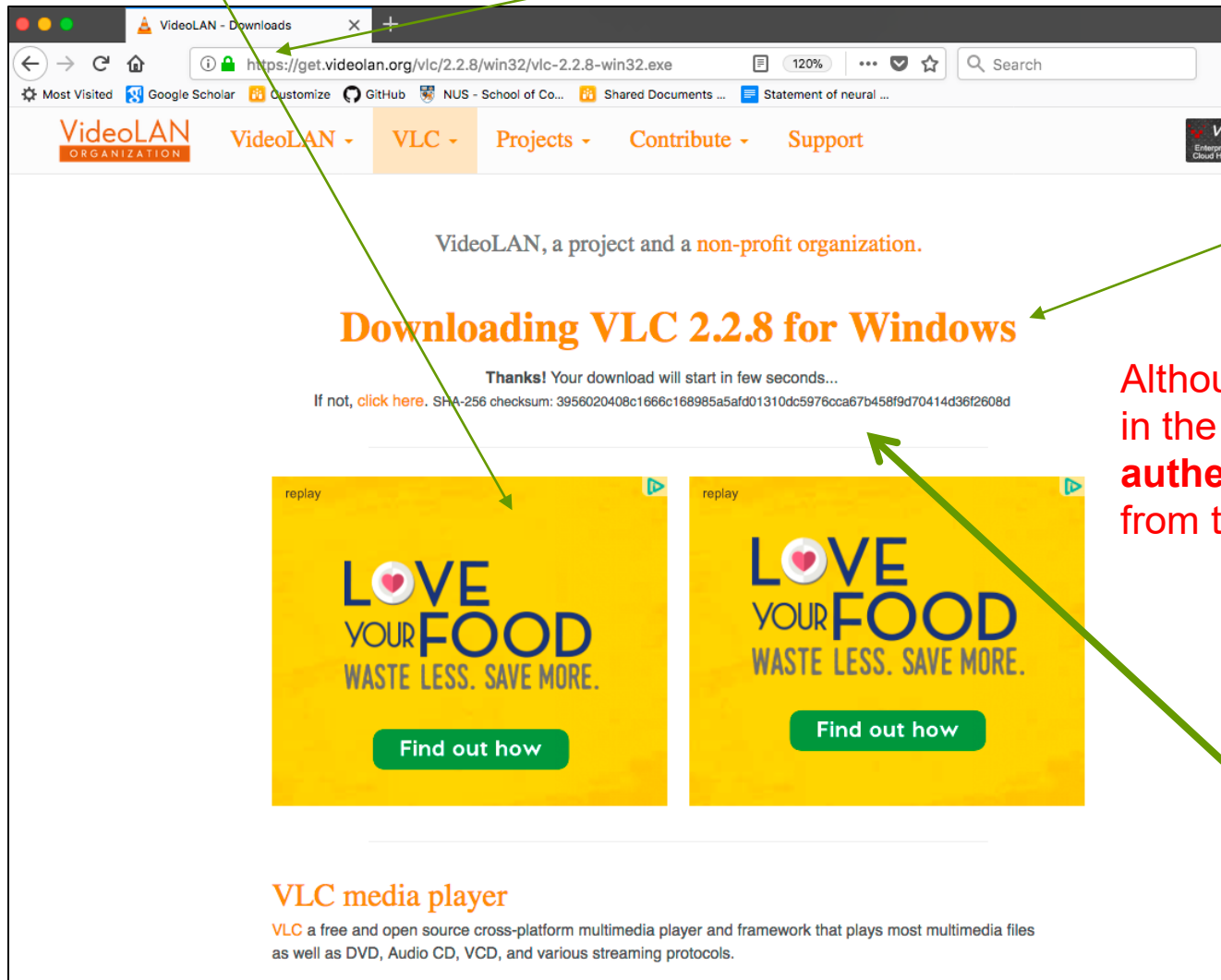
- Bob submitted a **medical certificate** to the lecturer, indicating that he was unfit for exam.
Was the certificate **authentic**: i.e. was it issued by the clinic?
Or had Bob altered the date?
- Alice received an **email** from the lecturer.
Is the email **authentic**: i.e. sent by the lecturer?

Another **important example**:

- Alice downloaded a **software** vlc-2.15-win32.exe from the Web.
Is the downloaded file **authentic**?
(See the “checksum” in <https://get.videolan.org/vlc/2.2.8/win32/vlc-2.2.8-win32.exe>)

ads from 3rd party

https: so the content displayed is from **videolan.org** and is authentic



The actual file is hosted by a **third party site**

Although the information displayed in the browser is verified to be **authentic**, what about **the file** from the third party site??

The digest (crypto checksum)

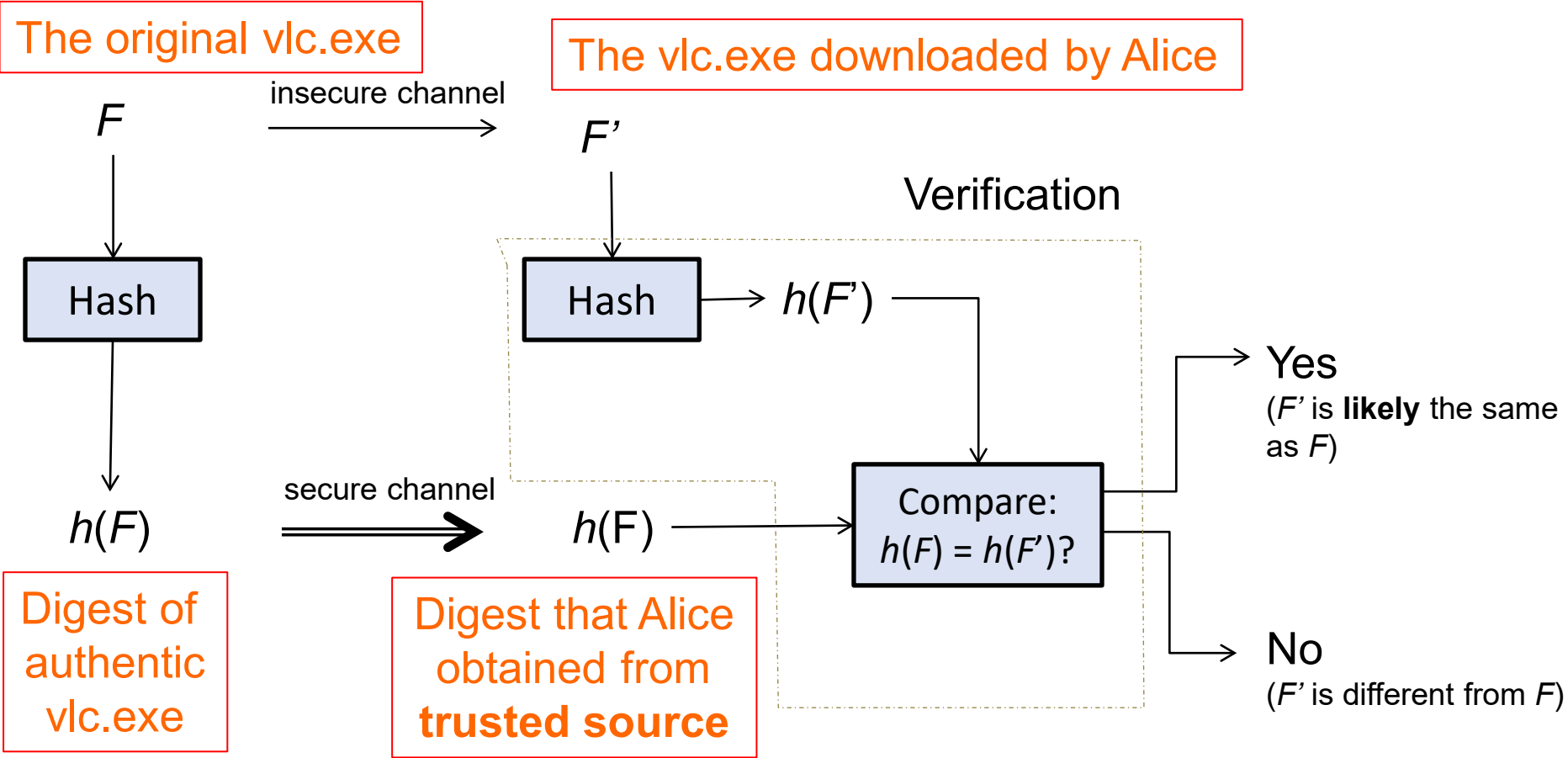
Question: Why is "videolan.org" in the URL is shown in bold? (to be discussed later)

Using (Unkeyed) Hash for *Integrity* Protection

Assuming that **there is a secure channel** to send a short piece of information, **the steps** are as follows:

- Let F be the *original* data file
- Alice obtains the digest $h(F)$ from the secure channel
- Alice then obtains a file, say F' , whose origin claims that the file is F
- Alice computes and compares the two digests $h(F)$ and $h(F')$:
 - If $h(F) = h(F')$, then $F = F'$ (with a very high confidence)
 - If $h(F) \neq h(F')$, then $F \neq F'$, i.e. the file integrity is compromised

Using (Unkeyed) Hash for *Integrity* Protection



Some Remarks

- What would an attacker try to do?
*To find the **second pre-image**, i.e. another $F' \neq F$ such that $h(F')=h(F)$*
- We may argue that with the digest, the verifier can be assured that the data is “**authentic**”, and thus the “authenticity” of the data origin is achieved
- Nevertheless, in many literatures/documents, when there is no secret key involved, we refer to the problem as an “**integrity**” issue
- There is a requirement for a **secure channel**, e.g. digest posted on a webpage which is sent over HTTPS
- How can provide (data-origin) authenticity?
Use **MAC** or **digital signature**

3.4 Data-Origin Authenticity

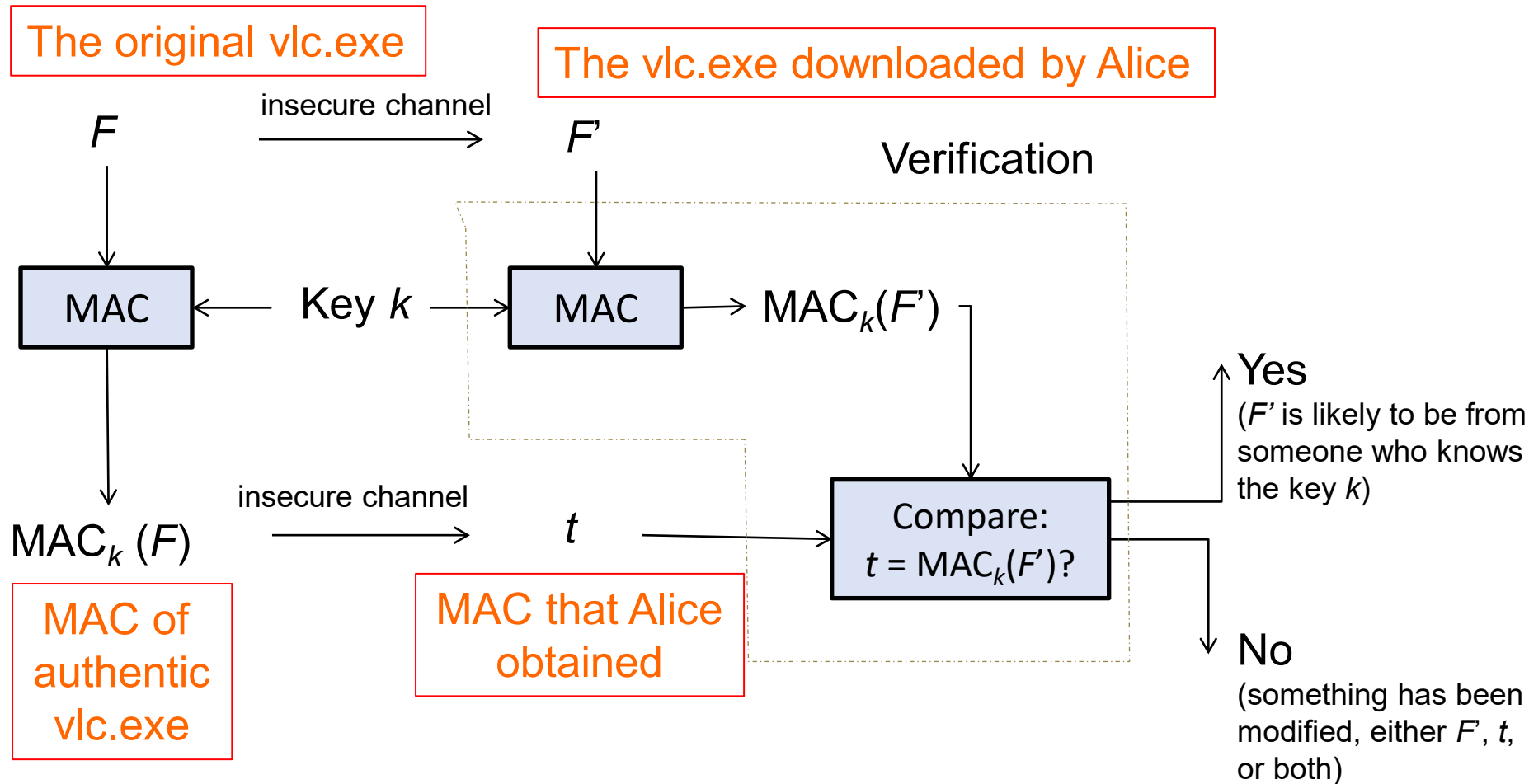
MAC and Signature

- In the previous vlc.exe example, the digest **could be forged**:
 - E.g. the server hosting the web page could be compromised
 - Or in the first place, it was obtained from a “spoofed” webpage
- In other words, we **don’t have** a secure channel to deliver the digest
- In such scenarios, we can protect the digest with the help of some secrets:
 - In the symmetric-key setting: **MAC**
 - In the public-key setting: **digital signature**

3.4.1 Data Origin Authenticity: MAC

MAC (Message Authentication Code)

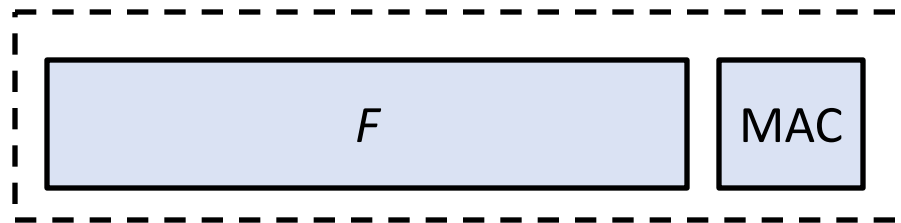
- In this setting, the MAC *might* also be modified by attacker
- If such a case happened, it can be **detected** with a high probability



Security requirement: without knowing k , it is difficult to forge the MAC

Some Remarks

- What would an attacker try to do?
To forge a valid pair of (file, MAC)
- Note that there is no issue with confidentiality:
the **data F** can be sent **in clear**
- Typically, the MAC is **appended** to F ,
then they are stored as a **single file**,
or **transmitted together** through a communication channel:
hence, MAC is also called the ***authentication tag***



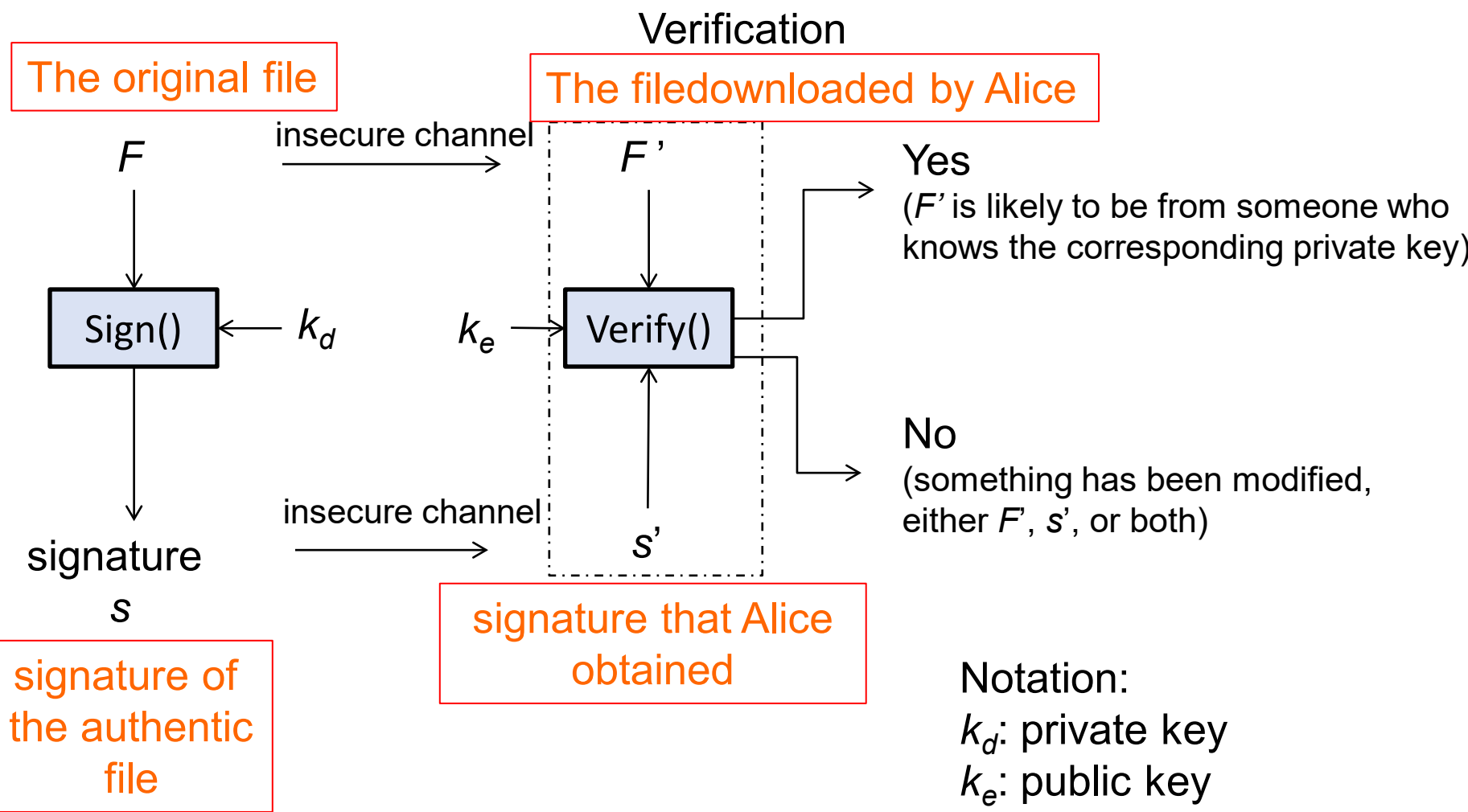
- Later, an entity who wants to verify the *authenticity* of F ,
can carry out the verification process using the secret key

3.4.1 Data Origin Authenticity: Signature

Digital Signature

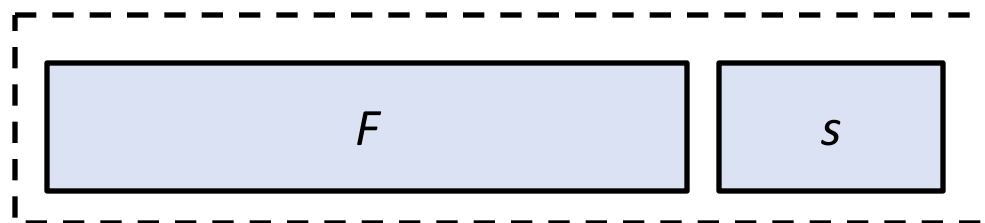
This is essentially the asymmetric-key version of MAC

Security requirement: without knowing k_d , it is difficult to forge s



Some Remarks

- Note that the signature is computed using the **private key** and F
- Likewise, the computed signature is typically **appended** to F and stored as a single file or transmitted together



- When we say that “***Alice signs the file F*** ”, we mean that Alice computes the signature s , and then appends it to F
- Later, the authenticity of F can be verified by ***anyone*** who knows the signer’s public key
- Why? The valid signature can be computed **only** by someone knowing the private key:
if the signature is valid, then F must be authentic

What's so Special about Signature (Compared to MAC)?

- We can view the digital signature as the counterpart of the **handwritten signature** in a legal document:
 - A legal document is authentic or certified if it has the **correct** handwritten signature
 - **No one**, except the authentic signer, can forge the signature
- In addition to authenticity, signature scheme also achieves ***non-repudiation***:
 - *Assurance that someone cannot deny his/her previous commitments or actions*

Two Scenarios of Non-Repudiation Provisioning

Using only **MAC** (non-repudiation is ***not*** achieved):

- Suppose Alice sent Bob a message appended with a MAC, which is computed with a secret key **shared by** Alice and Bob
- Later Alice denied that she had sent the message
- When confronted by Bob, Alice claimed that Bob generated the MAC

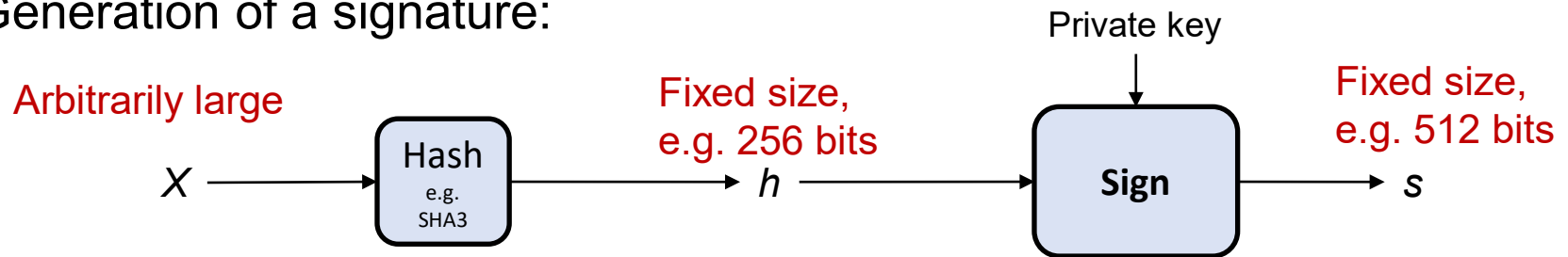
Using **digital signature** (non-repudiation ***is*** achieved):

- Suppose Alice sent Bob a message signed using her private key
- Later, she wanted to deny that she had sent the message
- However, she was unable to so: only the person who knows the **private key** (i.e. Alice) can sign the message
- Hence, the signature is a **proof** that Alice is aware of the sent message

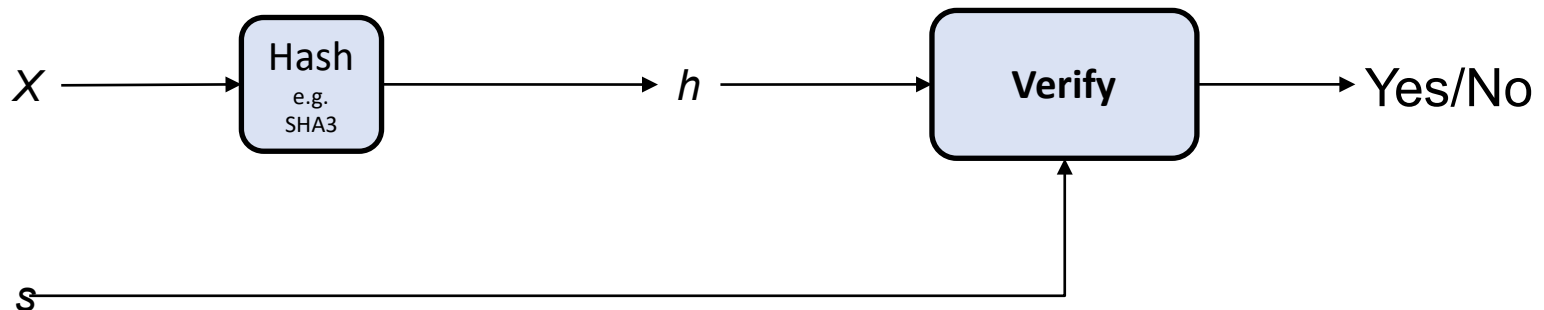
Design of Signature Scheme

- Most signature scheme consists of two components: an unkeyed **hash**, and the **sign/verify** algorithms

Generation of a signature:

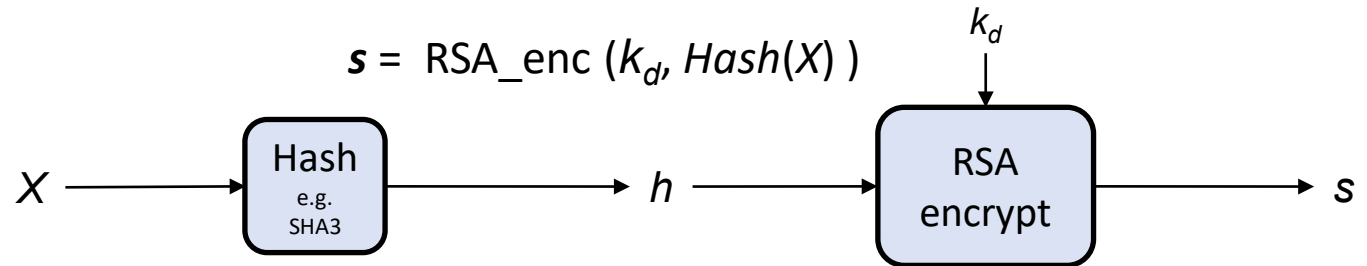


Verification of the signature:

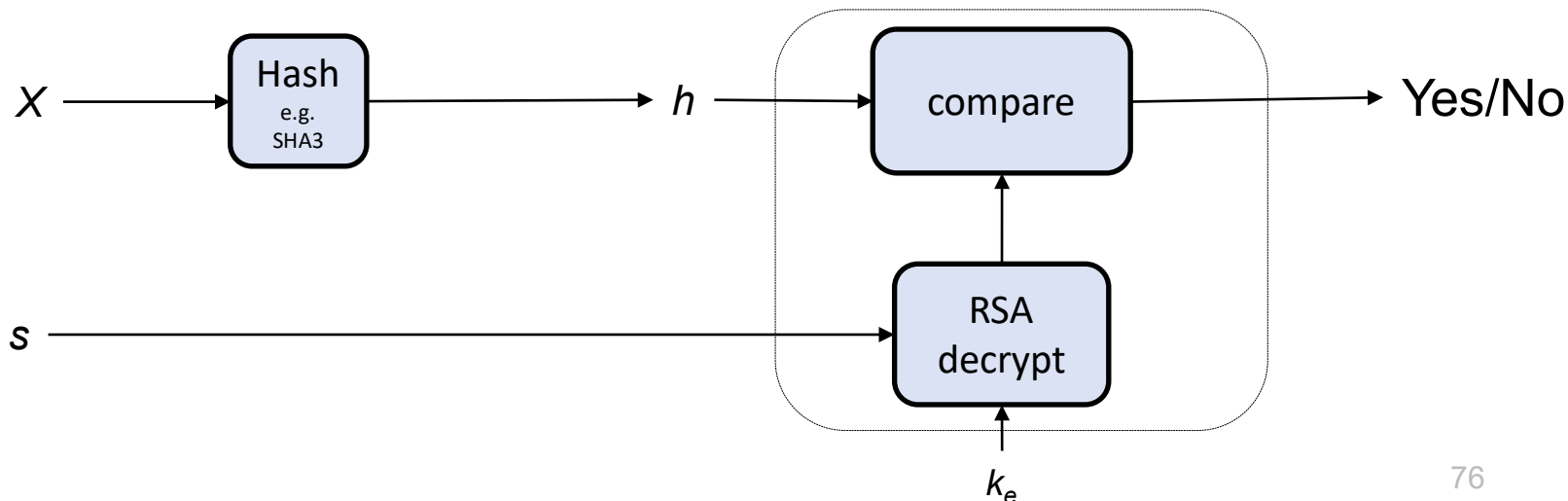


RSA-based Signature

- When RSA is used for signing and verification:
 - The **signature** is essentially the “**encrypted**” digest done using the private key
 - During verification, decrypt using the public key to obtain the digest and compare
 - A messy notation issue: we previously used the public key to encrypt, but here we use the private key to encrypt; recall that for RSA, we can flip the role of both keys*



If $\text{Hash}(X) = \text{RSA_dec}(k_e, s)$ then accept, else reject



Popular Signature Schemes

- The algorithm ***sign()*** usually corresponds to encryption using the private key; and the algorithm ***verify()*** corresponds to decryption using the public key: the details are omitted here
- However, do refer to the operations as ***sign()*** and ***verify()***
- For examples:
 - RSASSA-PSS, RSASSA-PKCS1:
a signature scheme based on RSA
 - DSA (Digital Signature Algorithm):
a standard by FIPS that is based on ElGamal

3.5 Some Attacks and Pitfalls

3.5.1 Birthday Attack on Hash

Hash and Birthday Attack

- Hash functions are designed to make a **collision** difficult to find (i.e. to be **collision resistant**)
- Recall that a *collision* involves two different messages m_1 , m_2 that give the same digest, i.e.:

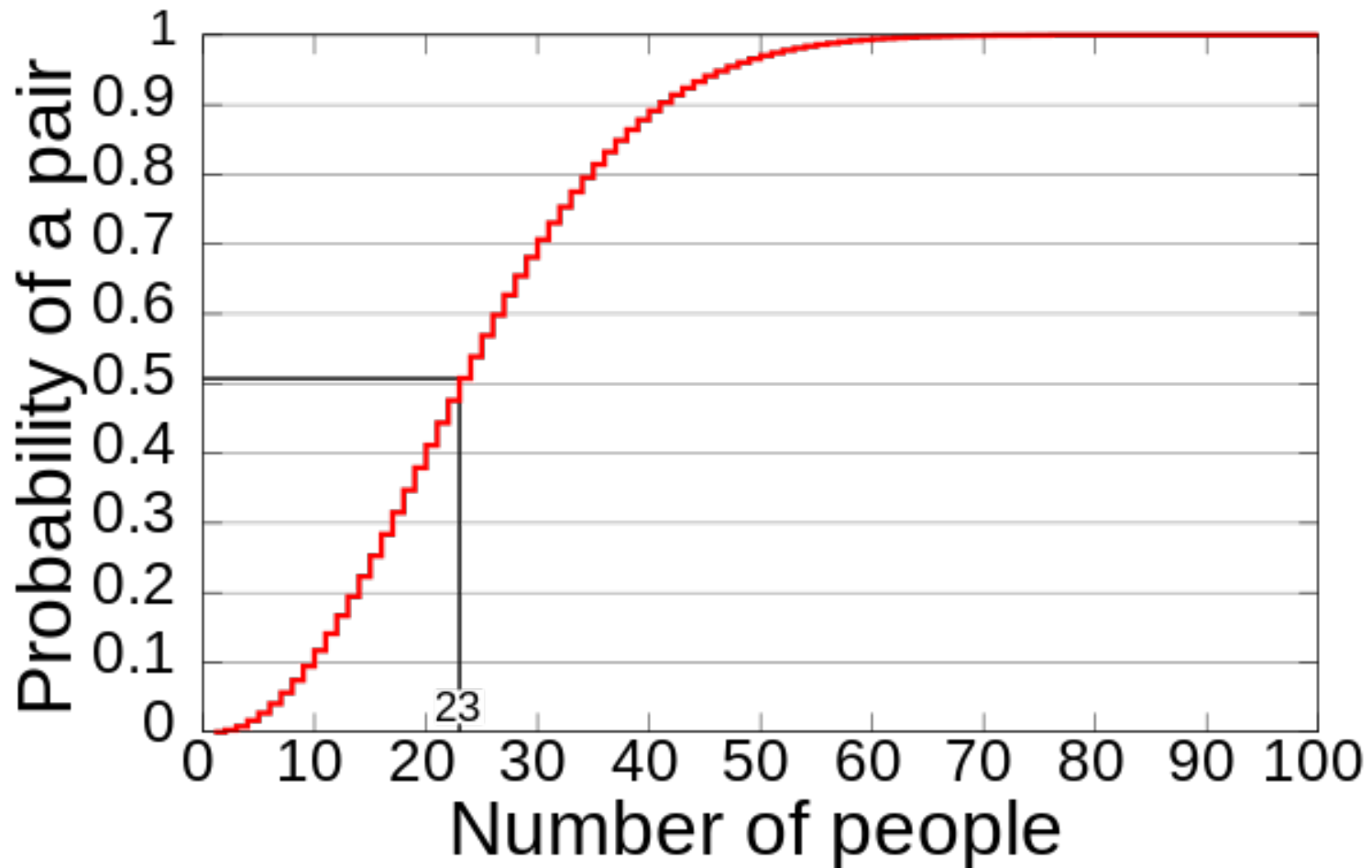
$$h(m_1) = h(m_2) \quad \text{and} \quad m_1 \neq m_2$$

- Nevertheless, all hash functions are subjected to **birthday attack** (similar to exhaustive search on encryption schemes)

Birthday Paradox/Problem

- “How many students need be randomly selected so that, with probability more than 0.5, there is *a pair of students* having the same birthday”
- Answer: **23**
- The (low) number needed may surprise many people!
- See: https://en.wikipedia.org/wiki/Birthday_problem
- *Why is that possible?*
- There are **366** possible birthdays (including 29 February)
- By **the pigeonhole principle**, the probability reaches 100% when the number of students reaches $366+1=367$
- This is *not* about fixing on one individual and comparing his/her birthday to everyone else's birthday
- But about comparing between *every possible pair* of students = $23 \times 22 / 2 = 253$ comparisons

Birthday Paradox/Problem



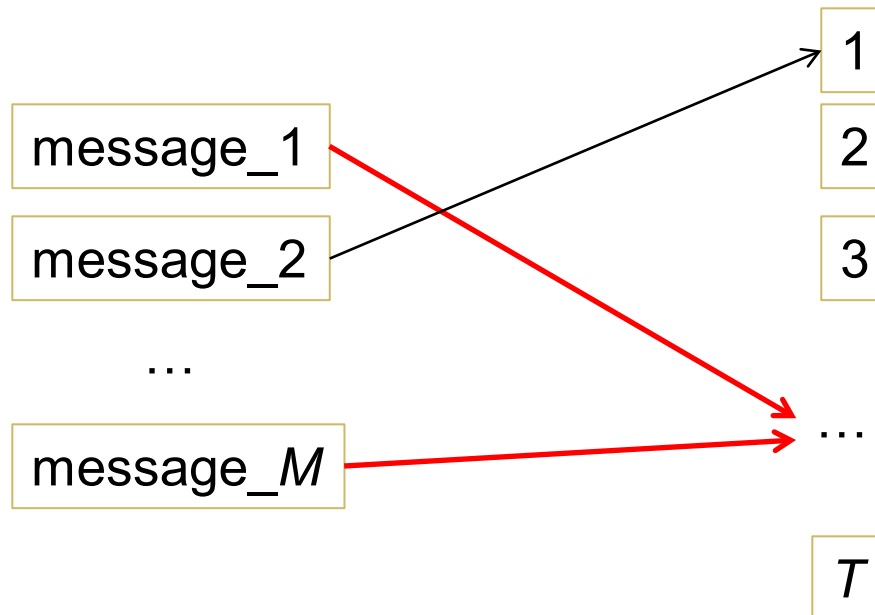
Ref: https://en.wikipedia.org/wiki/Birthday_problem

Fun Facts on Birthday Paradox

- Assume that a person can have at most 100,000 hair glands
- By **pigeon hole principle**, there exists two persons in Singapore who have the same number of hair glands
- By **birthday paradox**, with probability more than $\frac{1}{2}$, 2 in 1,000 under-graduate students in SoC have the same number of hair glands
- *See Tutorial 4*

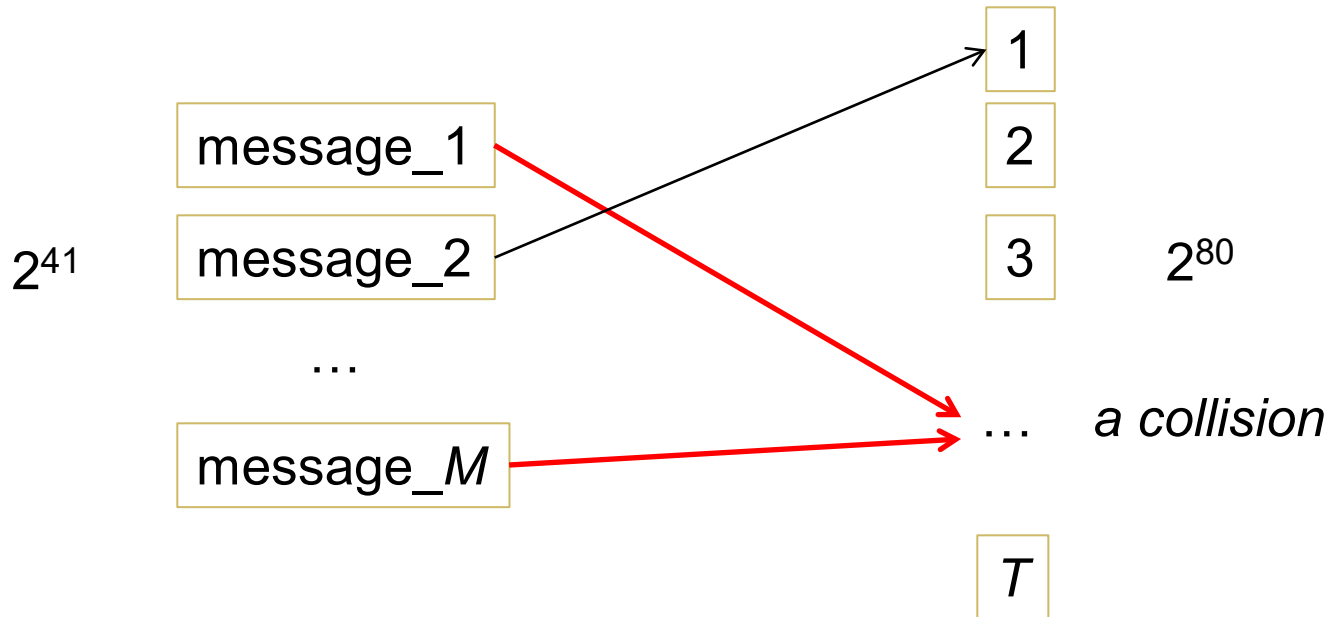
Birthday Paradox/Problem

- Birthday paradox can be applied to hash function
- Suppose we have M messages, and each message is tagged with a value randomly chosen from $\{1, 2, 3, \dots, T\}$
- If $M > 1.17 T^{0.5}$, then with probability more than 0.5, there is *a pair of messages* tagged with the same value
- This result leads to **birthday attack** on a hash function: to reduce the complexity/work-factor of finding a collision



Birthday Attack on Hash

- Suppose the digest of a hash is 80 bits: $T = 2^{80}$
- Now, an attacker wants to find a collision
- If the attacker randomly generates 2^{41} messages ($M = 2^{41} = 2^{1 + 80 \cdot 0.5}$), then $M > 1.17 T^{0.5}$
- Hence, with probability more than 0.5, among the 2^{41} messages, ***two of them give the same digest!***



In general, the **probability that a collision occurs** $\approx 1 - \exp(-M^2 / (2T))$

Implication of Birthday Attack for Digest Length

- Birthday attack has a serious consequence on the **digest length** required by a hash function to be collision resistant
- When the key length for a symmetric-key is **112**, the corresponding recommended length for digest is at least **224**
- Why must the digest length be significantly larger? (Answer: birthday attack, of course)
- Read the NIST cryptographic key-length recommendation, including on several popular hash functions:
<http://www.keylength.com/en/4/>

3.5.2 Using Encryption for Authenticity

Misuse of Encryption: A Case Study

- Encryption schemes may provide *false sense* of security
- Consider the following case involving **the design of a mobile app** from company XYZ:
- The mobile phone and a server share a secret 256-bit key k . The server can send **instructions** to the mobile phone via SMS. *(Note that an SMS consists of only readable ASCII characters. We assume, however, that there is a way to **encode** a binary string into readable characters, e.g. base-64).*
- Suppose **the format** of the instruction is:

$X \ P$

where: X is a **8-bit** string specifying the type of operation, and P is a **248-bit** string specifying the parameter.

- If an operation doesn't take in parameter, P will be ignored

Misuse of Encryption: A Case Study

- There is a total of **16 valid instructions**, such as:
 - 00000000 P : adds P into the contact list
 - 11110000 P : rings for P seconds
 - 10101010 : self-destruct (brick) the phone!**
- An instruction is to be **encrypted** using as 256-bit AES, encoded to readable characters, and then sent as an SMS
- After a mobile phone received the SMS, it **decrypts** it. If the instruction is invalid, it ignores the instruction. Otherwise, it executes the instruction.
- The company XYZ claims that: *“256-bits AES provides high level of security, and in fact is classified as Type 1 by NSA. Hence the communication is secure. Even if the attackers have compromised the base station, they are still unable to break the security”*.
- **What’s wrong with this?**

Misuse of Encryption: A Case Study (Concluding Remarks)

- Encryption is designed to provide **confidentiality**
- It does **not**, however, guarantee integrity and authenticity
- In the case above, the XYZ company actually wants to achieve “authenticity” (or “integrity”??)
- But it wrongly employs encryption to achieve that
- A secure design could use **MAC** instead of encryption

Notes:

- There are still some details being omitted in this case
- Simply adding MAC to the instructions is not secure, for example against “replay attacks”
- To prevent replay attacks, a “nonce” is required

The Need for Integrity

- Some may argue that an encryption is “**sufficient**” to protect a message:
 - The attacker doesn’t know the plaintext and obtains only the ciphertext
 - A change on the ciphertext will result into a change in the plaintext
- What an attacker can possibly do on an encrypted message?
 - ***Ciphertext blocks re-ordering*** say on CBC: See Tutorial 4
 - Integrity attack on **One-Time-Pad**:
what happen if **1 bit** of the ciphertext gets flipped?
 - Modification has a predictable impact on the plaintext!
 - Modification to the ciphertext can be undetected
- Unmet “***non-malleability***” security goal:
(Informal) a cipher is ***malleable*** if it is possible to modify a ciphertext and cause a predictable change to the plaintext

3.6 Password File Protection (Revisited)

Hashed Password (From Lecture 2)

- Passwords should be “hashed” and stored in the password files.
(Textbook ([PF]pg 46) uses the term “**encrypted**”. Note that this is **inaccurate**. For encryption, there is a way to recover the password from the ciphertext. For cryptographically secure hash, it is infeasible to recover the password from its hashed value.)
- During authentication, the password entered by the entity is hashed, and compared with the the value stored in the password file

Password in clear

Alice	OpenSesaMe
Bob	123456
Ali	SesameOpen
Charles	SesameOpen

Hashed password

Alice	X3lad=3adfv
Bob	3Dv6usgawer
Ali	da5DGDSDFd3
Charles	da5DGDSDFd3

Hash(“**SesameOpen**”) = “**da5DGDSDFd3**”

Hashed Password (From Lecture 2)

- It is desired that the same password will be hashed into *two different values* for two different userid.
Why? (see tutorial)
- This can be achieved by using **salt**

Password in clear

Alice	OpenSesaMe
Bob	123456
Ali	SesameOpen
Charles	SesameOpen

Salted-hashed password

Alice,	Adf3,	39Gkaj10Dmf
Bob,	a3gh,	d978bjklDFD
Ali,	f8ad,	DJk34hoaev7
Charles,	10vd,	K108ELvio2B

$\text{Hash}(\text{"f8adSesameOpen"}) = \text{"DJk34hoaev7"}$
 $\text{Hash}(\text{"10vdSesameOpen"}) = \text{"K108ELvio2B"}$