Topic 3: Authenticity (Data Origin)

3.1. Crypto Primitive: Public Key Cryptography

- 3.1.1 RSA
- 3.1.2 Security of RSA
- 3.1.3 Remarks

3.2 Data Authenticity (Hash): Unkeyed hash

3.3 Data Authenticity (Mac) : Keyed Hash

3.4 Data Authenticity (Signature): Asymmetric key.

3.5 Some attacks & pitfalls

- 3.5.1 Birthday attack on hash
- 3.5.2 Design flaw: using encryption for authenticity
- 3.5.3 Time-Space tradeoff

3.6 Other applications of Hash: password file protection (in Topic 2)

Remark:

- In Topic 2, we mentioned two modes of authentication. Topic 3: Data Origin (crypto primitive); Topic 4: Communicating Entity (authentication protocol).
- Public key encryption logically should be covered in Topic 1. However, we have it here for better flow.

Summary & takeaways

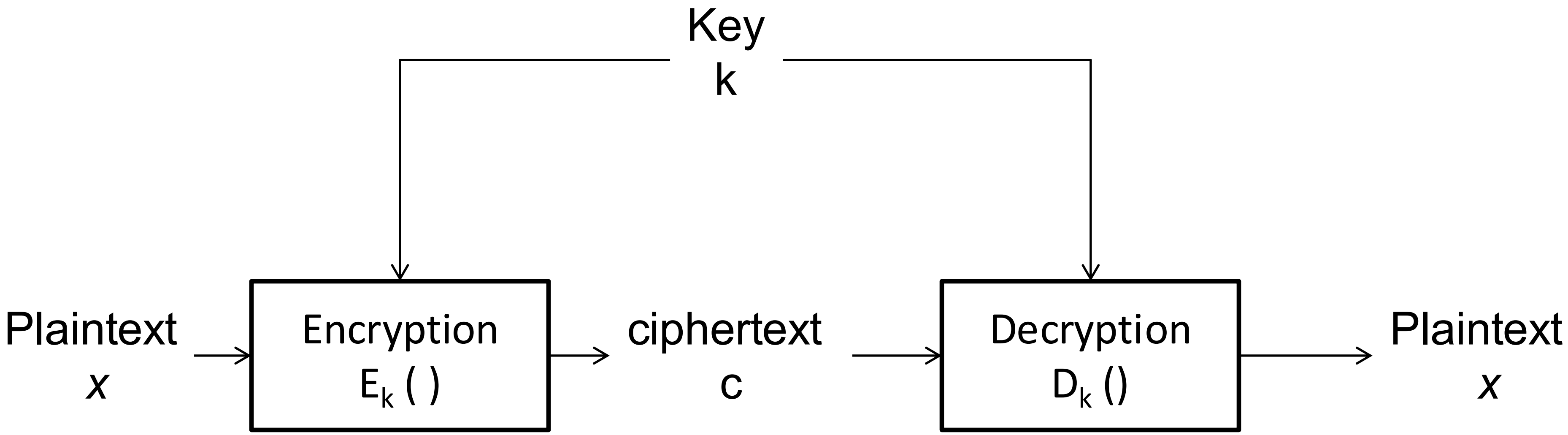
- Public Key Encryption.
  - RSA (based on integer factorization. Only integer). ElGamal (based on discrete log of “Algebraic group”. Many choices: ECC).
  - Differences with symmetric key. Implications:
    - Symmetric key requires a ***secure channel*** to distribute key.
    - Public key requires a ***secure broadcast channel*** to distribute key.
  - Post-Quantum Crypto (implication after quantum computer become available)
  - Pitfall: using RSA in symmetric key setting. Using textbook RSA.
- Authentication primitives: digest, mac, signature.
  - *Security models and application scenarios (Summary Slide 73,74,75)*
  - Digest
    - No key or other secret information in hash/digest.
    - Hash requirement: Collision resistant. Collision resistant vs 2nd pre-image attack.
    - All hash subjected to Birthday attacks.
  - Signature vs mac
    - (advantage) only require secure broadcast channel; non-repudiation.
    - (disadvantages): efficiency.
- Achieve Confidentiality $\not\Rightarrow$ Achieve Authenticity (if a scheme preserves confidentiality, it might not preserve authenticity)
- Construction:
  - hash: SHA
  - Mac: CBC-mac, HMAC
  - Signature: DSA, hash-and-sign. Special property of RSA for signature (hash-and-encrypt).
- Time-memory-tradeoff in inverting hash

3.1. Public Key Cryptography (PKC)

Public Key Crypto primitives include public key encryption and signature. The goal of Public Key Encryption is for confidentiality, while the goal of signature is for authenticity.

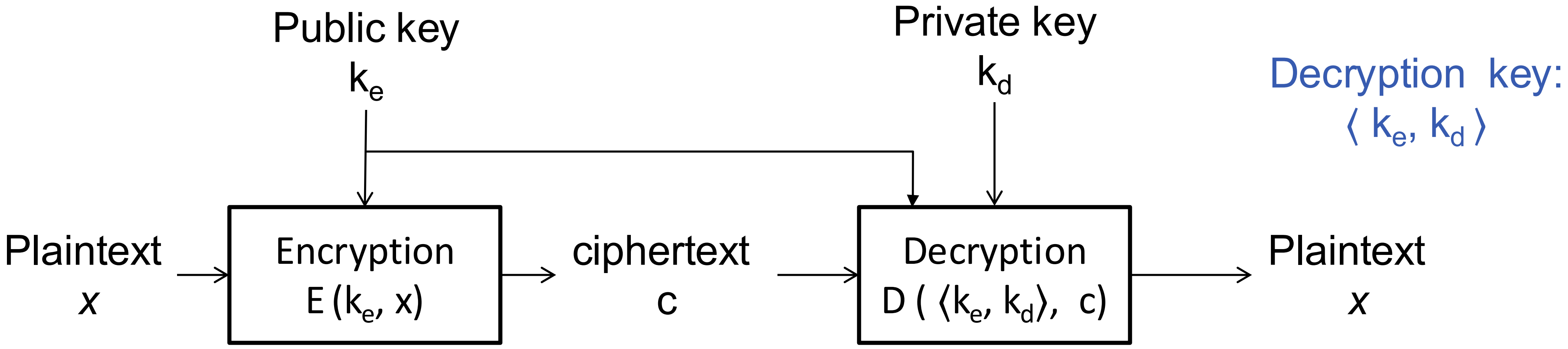
Public Key Encryption

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



Overview

A public key (aka asymmetric-key) scheme uses different keys for encryption and decryption.



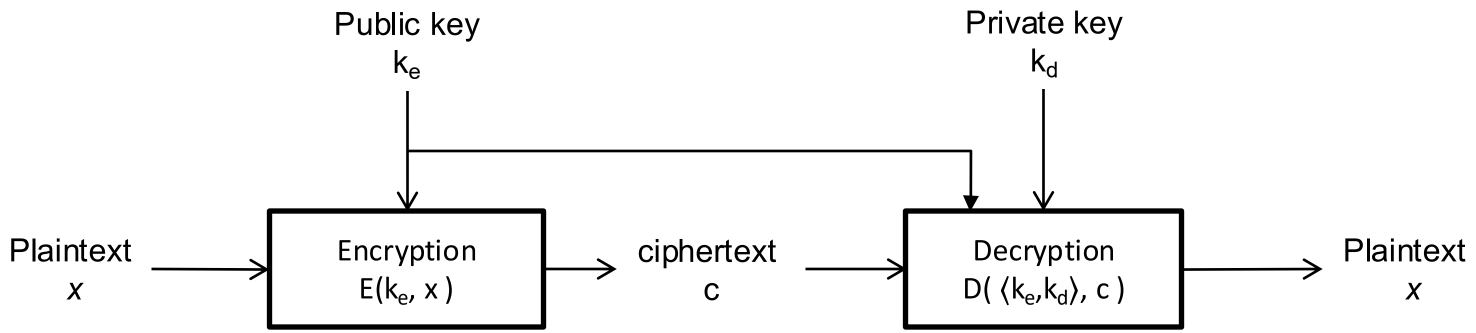
Note on definitions:

- Decryption algo also takes in the public key.
- In literature, some include k_e as part of private key, but some don't. In this lecture, we take the convention:

  Public key: k_e, Private key: k_d, Decryption key: (k_e, k_d)

  For abbreviation, many would say "using private key to decrypt". This implicitly means using both the private and public key to decrypt.

Why called “public key”?



- The owner Alice keeps the private key as a secret but tell everyone the public key (for eg. posts in her Facebook). Everyone knows the public key.

- An entity, Bob, has a ***plaintext x*** for Alice. Bob can encrypt it (using the ***public key***) and posts the ***ciphertext c*** on Alice’s Facebook.

- Another entity, Eve, obtains the public key, and ciphertext c from the Alice’s Facebook. Without the private key, Eve is unable to derive x.

- Alice, with the ***private key*** and the ***public key***, can decrypt and obtain the plaintext x.

Hello everybody, this is my public key X23kavlaem3q4?a!f+35X#;c3s%$^

314d8zf34gsfg$#VDSDF73SGDSFe

Yes, lol 😇

couldn’t get it

This example illustrates usage of public key for encryption. In practice, it is more common to employ PKC for “authentication”.

Security Requirements of a Public Key Encryption

- Threat model:

  - Given the public key and the ciphertext (but not the private key), the attacker wants to get info of the plaintext;

- To be secure under the above, it must be difficult for the attacker to get the private key from the public key.

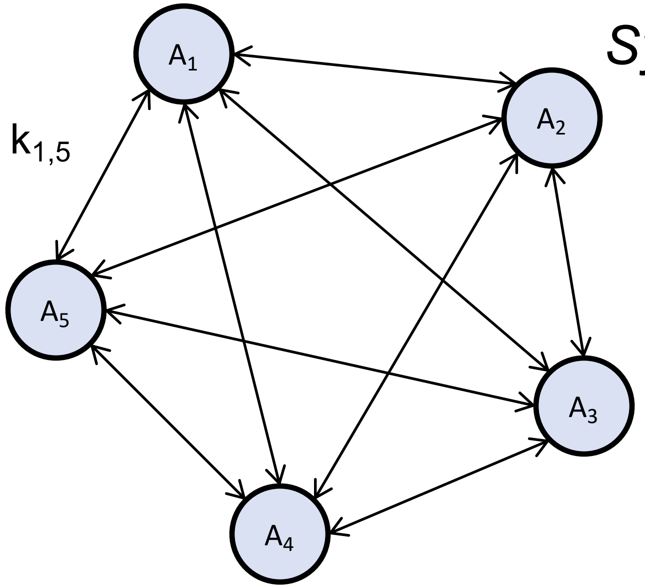
Remark:

- Recap that "indistinguishability" is the modest goal.
- ***Encryption Oracle*:** With the public key, anyone can encrypt. So, the attacker always has access to the "encryption oracle".

What so useful about public key?

- Suppose we have multiple entities, A_1 , A_2, …, A_n :
  - Each of them can compute a pair of ⟨ private key, public key ⟩
  - Each broadcast his/her public key, but keeps the private key secret
- Now, suppose A_i wants to encrypt a message *m* for A_j only:
  - 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 don't use the PKC, then, any two entities must share a symmetric key via a secure channel. Implications:
  - Many keys are required in symmetric keys;
  - Symmetric key requires both entities to know each other before the actual communication session. In contrast, PKC can handle such scenario. (e.g. You want to send a message to www.bbc.com. Does www.bbc.com know you? Likely no. If www.bbc.com doesn't know you, it is impossible to set up a secure channel for you and www.bbc.com to establish a shared symmetric key. Now, with PKC, www.bbc.com can first broadcasts its public key. Even an entity whom www.bbc.com doesn't know can get the public key.)

Keys Distribution



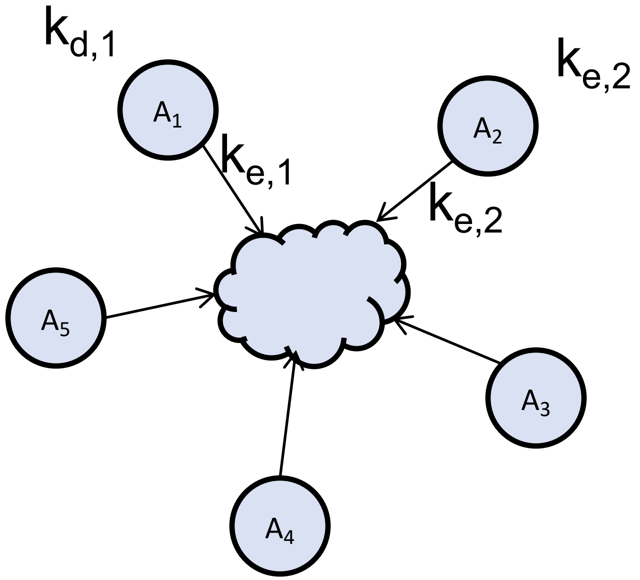
Symmetric key setting: ***Individual secure channel for each pair***

Every pair of entities requires one key. We need a secure channel to distribute that key.

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

$k_{1,2}$, $k_{1,3}$, ...

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



Public key setting: ***Secure Broadcast Channel***

Each entity publishes its public key

Entity A_i publish $k_{e,i}$ and keep $k_{d,i}$
Total number of public keys: n
Total number of private keys: n

Advantages:

- Fewer keys.
- Entities don't need to know each other before broadcasting the keys.

- **Symmetric key**: We need a secure channel to establish the secret key for any two entities.

- **Public key**: We only need a secure broadcast channel to distribute the public key.

Remark

- Although easier than setting up individual channels, setting up a secure broadcast channel for public key is still challenging.

  (whole lecture on PKI in Topic 4)

- While we introduce PKC for encryption, an important application of PKC is in authentication (e.g. HTTPS).

- PKC has limitations compare to symmetric key such as AES.
  - Efficiency: key size and compute time;
  - Security: many current popular PKC are vulnerable to quantum computer;

Popular PKC schemes

- RSA:
  - Operate on integer modulo. (optional remark. Need both multiplication and addition. i.e. need a "field")
  - Recommended Key size ≈ 2048 bits
- ElGamal:
  - ElGamal can operates in other choices of algebraic groups. (optional remark: need only one group operator)
  - E.g, Elliptic Curve Cryptography (ECC), which requires key size ~300 bits for equivalent of ~2048 in RSA.
- Paillier: homomorphic with respect to addition.
- Post-Quantum cryptography: Quantum computer breaks RSA. Crypto that is secure against quantum computer is called "post-quantum cryptography". Quantum computer not here yet.

The idea of 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

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

Rivest, Shamir and Adleman,
2002 Turing Award.

Integer representations

- We will introduce the “textbook” RSA, i.e. the basic form of RSA. We call it “textbook” because the basic form is not secure. In practice, some forms of padding required and there are special considerations in choosing the primes.

- RSA represents the data (plaintext, ciphertext, key) as integers. The integers can be represented using binary representations. Note that a *n*-bit number ranges from 0 to 2^{n-1}. (e.g. a 3-bit integer is a value from 0 to 7)

- Since the number of 1024-bit integers are 2^{1024}, hence, an algorithm that exhaustively searches all 1024-bit numbers is infeasible.

3.1.1 RSA

- Textbook RSA: algo that taught in many “non-security” modules.

In practice:

- Padded RSA. (to destroy some property)
- Choose strong primes.
- Fast and secure way to generate primes.
- Secure implementation to guard against side-channel attack.
- the “e” is fixed, typically fixed as 65537

- Quantum Computer.

"Textbook RSA" - setup

1. Owner randomly chooses 2 large primes $\boldsymbol{p}, \boldsymbol{q}$ and computes $\boldsymbol{n} = \boldsymbol{pq}$.
2. Owner randomly chooses an *encryption exponent* $\boldsymbol{e}$ s.t. gcd($\boldsymbol{e}$, **(p-1)(q-1)**) =1.
   (i.e. e does not have common factor with (p-1) or (q-1).)
3. Owner finds the *decryption exponent* $\boldsymbol{d}$

   where $\boldsymbol{d}\,\mathbf{e} \bmod (p\text{-}1)(q\text{-}1) = 1$

   There is an algorithm that finds d when given e, p and q. We won't get into the details,

   The term $(\boldsymbol{p}\text{-}1)(\boldsymbol{q}\text{-}1) = \Phi(\boldsymbol{n})$ is aka the Euler's totient function, which is the number of co-primes < n.
4. Owner publishes $\langle \boldsymbol{n}, \boldsymbol{e} \rangle$ as public key, and safe-keeps $\boldsymbol{d}$ as the private key. (note that owner doesn't need to keep p, q. The e is not required during decryption.)

Encryption, Decryption

private key d

public key $\langle n, e \rangle$

decryption key $\langle n, d \rangle$

- **Encryption,**

  Given m, the ciphertext c is

  $c = m^{e} \bmod n$

- **Decryption**

  Given c, the plaintext c is

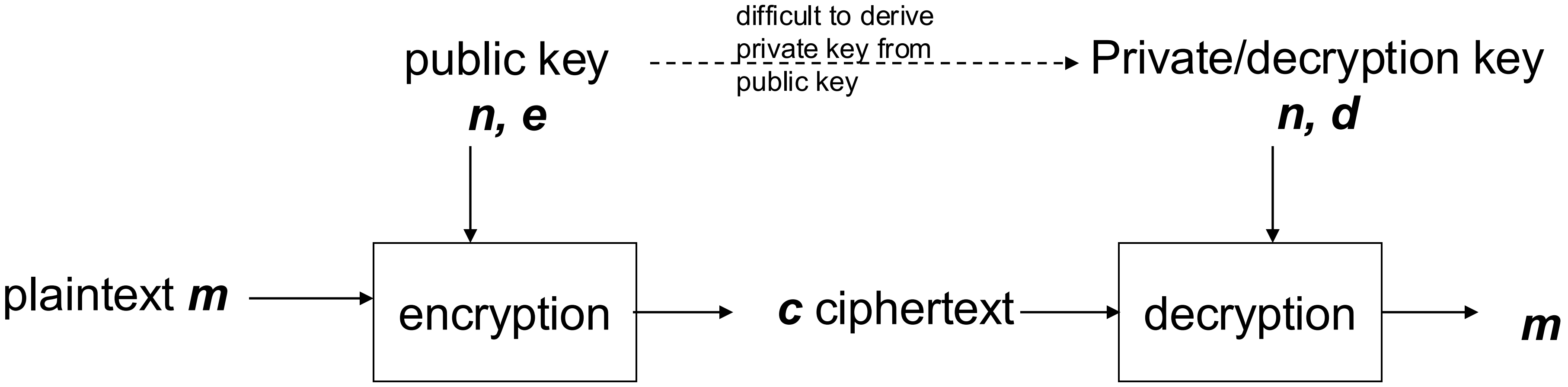
  $m = c^{d} \bmod n$

RSA

$n \quad = pq$

$\Phi(n) = (p-1)(q-1)$

$d = e^{-1} \bmod \Phi(n)$



public key

n, e

difficult to derive private key from public key

Private/decryption key

n, d

plaintext ***m***

encryption

c ciphertext

decryption

m

$c = m^e \bmod n$

$m = c^d \bmod n$

Optional remark: Consider the relationship: $c = m^e \bmod n$

- *(RSA problem) Given c, n, e, find m, (called the e-th root of c)*
- *(discrete log problem): Given c, n, m, find e, (called the discrete log of c)*

Correctness of RSA

- Note that for any positive $m < n$, and any pair of public/private keys,

Decrypt (Encrypt (m)) = m

That is,

$$(m^e)^d \mod n = m$$

Optional

Proof (sketch): The correctness depends on this property of modulo:

For any m, r, n, we have

$$m^r \mod n = m^{r \mod \Phi(\boldsymbol{n})} \mod n$$

(when n is product of two primes, p,q, then $\Phi(\boldsymbol{n}) = (p-1)(q-1)$.)

Now, combining the fact that

$$de \mod (p-1)(q-1) = 1,$$

we have $m^{de} \mod n = m^1 \mod n = m$

Example

- p=5, q=11, n=55
- (p-1)(q-1) = 40
- Suppose e = 3, then d = 27

note that indeed 3×27 mod 40 = 81 mod 40 = 1

There is an efficient algorithm that computes d from p, q, e. Details omitted.

- Suppose m = 9

Encrypt:

$c = m^e \bmod n = 9^3 \bmod 55 = 14$

Decrypt

$c^d \bmod n = 14^{27} \bmod 55$

$= 14^{8\times2+8+3} \bmod 55$

$= (36\times16\times49) \bmod 55$

$= 9$

There is an efficient algorithm that computes modulo exponentiation. Details omitted.

Algorithmic issues

- (Encryption/decryption) There is an efficient algo to compute exponentiation. Hence, there is an efficient encryption algorithm: given ***n***, ***m***, ***e***, compute m^e mod ***n***, and an efficient decryption algorithm: given ***c***, ***d***, ***n***, compute c^d mod ***n***

  (To get additional speedup, choose a small ***e*** so that encryption can be done very fast. It turns out that the value of e won't affect the security. But it can't be too small due to some attacks. It is common to fixed ***e*** =65537. In such cases, ***e*** is not a secret.)

- (step 1 in setup: Primality test) How to find a random prime? Here is a correct and secure method:

  1.Randomly and uniformly picks a number ***p***

  2.If ***p*** is a prime, outputs ***p*** and halts. Otherwise, repeats 1-2.

  Since there are many primes*, the probability that the a randomly chosen integer is prime is high. There is fast algorithm to determine whether a number is a prime.

- (step 3 in setup) The value of ***d*** can be efficiently computed from ***e*** and ***n*** using the *extended Euclidean algorithm.*

- Using Chinese Reminder Theorem, one can speedup decryption about x2 faster. Public key doesn't need to be large (but not too small). To achieve further speedup in encryption, it is common to choose a fixed small fixed public key such as 65537, which is 2^{16} - 1.

*: (optional) This is by the well-known "Prime Number Theorem". Probability of a randomly chosen 1024-bit number being a prime is around ln(2^{1024})~ (1/710). So likely takes about 710 trials to get a 1024-bit prime number. Fast but too slow for real-time application. A well-know vulnerable algorithm, instead of uniformly pick the numbers, employ a faster but unfortunately insecure algorithm to find primes.

Interchangeable role of encryption and decryption key in RSA (but may not in others)

- Textbook RSA has this property: we can also use the decryption key ***d*** to encrypt, and then the encryption key ***e*** to decrypt. In other words, we can swap the role of ***d*** and ***e**,* so that anyone in the public can decrypt, but only the owner can encrypt.

- The above property is something special about RSA. It usually does not hold in other public key schemes. For e.g. the ElGamal PKC doesn't has this property. This property is useful in designing signature scheme.

Caution: Some documents flip encryption/decryption when describing RSA, i.e. using public key to decrypt, and private key to encrypt. This can be very confusing. The root of this inconsistencies is due to how we use RSA in a signature scheme. In RSA-based signature, the decryption key is the public key.

Other PKC Schemes:

Discrete log-based PKC

- 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 Cure are often called Elliptic Curve Cryptography (ECC). Certain choices of ECC reduce the key size. E.g. ~300 bit for equivalent of 2048-bit RSA.

- Paillier Encryption

Paillier Encryption is discrete-log based.

optional remark:

- Paillier is homomorphic w.r.t. addition.
- ElGamal can be easily modified to be homomorphic w.r.t. multiplication.

Post-quantum crypto.

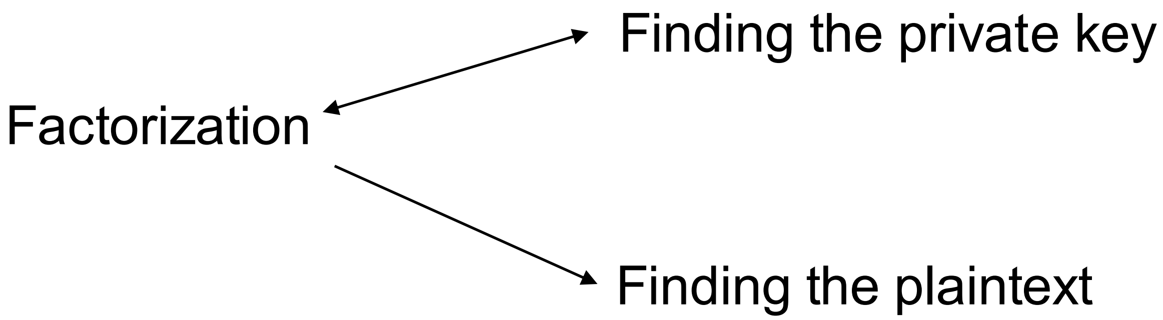
- See slides on post-quantum.

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 is as difficult as factorization.



Finding the private key

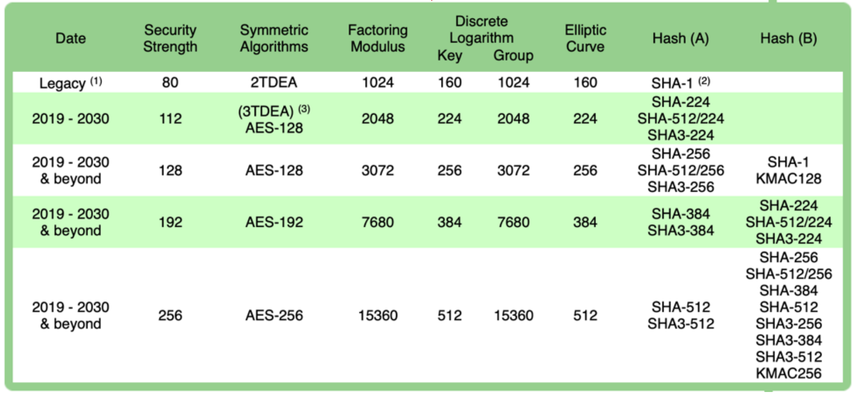
Factorization

Finding the plaintext

State-of-the-art on factorization

- more info: http://en.wikipedia.org/wiki/Integer_factorization
- A 640 bits number was successfully factored on Nov 2, 2005, using approximately 30 2.2GHz-Opteron-CPU years. It is computed over five months using multiple machines.
- A 768 bits number (RSA-768) was factored in Dec 2009, using hundreds of machines over 2 years.
- See https://en.wikipedia.org/wiki/RSA_Factoring_Challenge for latest status. (829 bits in 2020.)

Recap NIST recommendation in Topic 1.



Date	Security Strength	Symmetric Algorithms	Factoring Modulus	Discrete Logarithm		Elliptic Curve	Hash (A)	Hash (B)
				Key	Group			
Legacy (1)	80	2TDEA	1024	160	1024	160	SHA-1 (2)	
2019 - 2030	112	(3TDEA) (3) AES-128	2048	224	2048	224	SHA-224 SHA-512/224 SHA3-224	
2019 - 2030 & beyond	128	AES-128	3072	256	3072	256	SHA-256 SHA-512/256 SHA3-256	SHA-1 KMAC128
2019 - 2030 & beyond	192	AES-192	7680	384	7680	384	SHA-384 SHA3-384	SHA-224 SHA-512/224 SHA3-224
2019 - 2030 & beyond	256	AES-256	15360	512	15360	512	SHA-512 SHA3-512	SHA-256 SHA-512/256 SHA-384 SHA-512 SHA3-256 SHA3-384 SHA3-512 KMAC256

http://www.keylength.com/en/4/

- Here is the 640-bits number

  3107418240490043721350750035888567930037346022842727545720161948823206440518081504556346829671723286782437916272838033415471073108501919548529007337724822783525742386454014691736602477652346609

  =

  1634733645809253848443133883865090859841783670033092312181110852389333100104508151212118167511579

  *

  1900871281664822113126851573935413975471896789968515493666638539088027103802104498957191261465571

- Here is the 768-bits number

123018668453011775513049495838496272077285356959533479219732245215172640050726365751874
520219978646938995647494277406384592519255732630345373154826850791702612214291346167042
921431160222124047927473779408066535141959745985690214 3413

=

347807169895689878604416984821269081770479498371376856891243138898288379387800228761471
1652531743087737814467999489

*

367460436667995904282446337996279526322791581643430876426760322838157396665112792333734
171433968102700927987363089 17

from

T Kleinjung et al., *Factorization* of a *768-bit* RSA modulus, eprint 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-*qubits* quantum computer was built to factor 15, carried out by IBM using NMR.

  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 Quantum computer.

Post-Quantum Cryptography: This refers to cryptography that are secure against quantum computer.

- Code base: Base on difficulty in decoding error correcting code.
- Lattice-based cryptography: Based on this hard problem-- Given the "basis" of lattice, it is computationally hard to find the shortest (or approx) lattice point. Most promising approach.
- Multivariate polynomial: Given a multivariate polynomial, it is difficult to find the solution (under modulo p).

Although currently we don't have quantum computer with sufficient "qubits" to break ~1000 bit RSA, the threat of having such computer in near future is not negligible while the damage would be catastrophic. Hence, there are significate efforts to migrate current systems to post-quantum crypto. NIST is choosing a standard. The process started in 2016. On July 5, 2022, NIST announced the first group of winners (**see** https://en.wikipedia.org/wiki/NIST_Post-Quantum_Cryptography_Standardization details of the algo not required. Just be aware of the NIST standardization process.) More winners would be added to the standard.

There were 50 submissions for round 1.

Padding of RSA

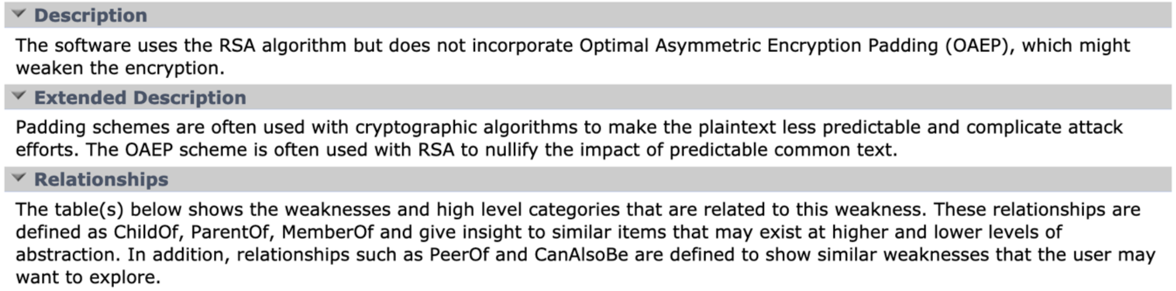
- Same as symmetric-key encryption, some forms of IV is required so that encryption of the ***same*** plaintext at different times would give different ciphertexts. Hence additional padding are required for security.

- Textbook RSA has an interesting "homomorphic" property. These properties are useful in applications (e.g. blind signature, encrypted domain processing), but they also lead to attacks. Such properties can be destroyed using padding.

- The standard (Public-Key Cryptography Standards) PKCS#1, add "optimal padding". Details omitted.
  http://en.wikipedia.org/wiki/PKCS_1

Pitfall: (CWE-780) Using Texbook RSA

https://cwe.mitre.org/data/definitions/780.html



Description

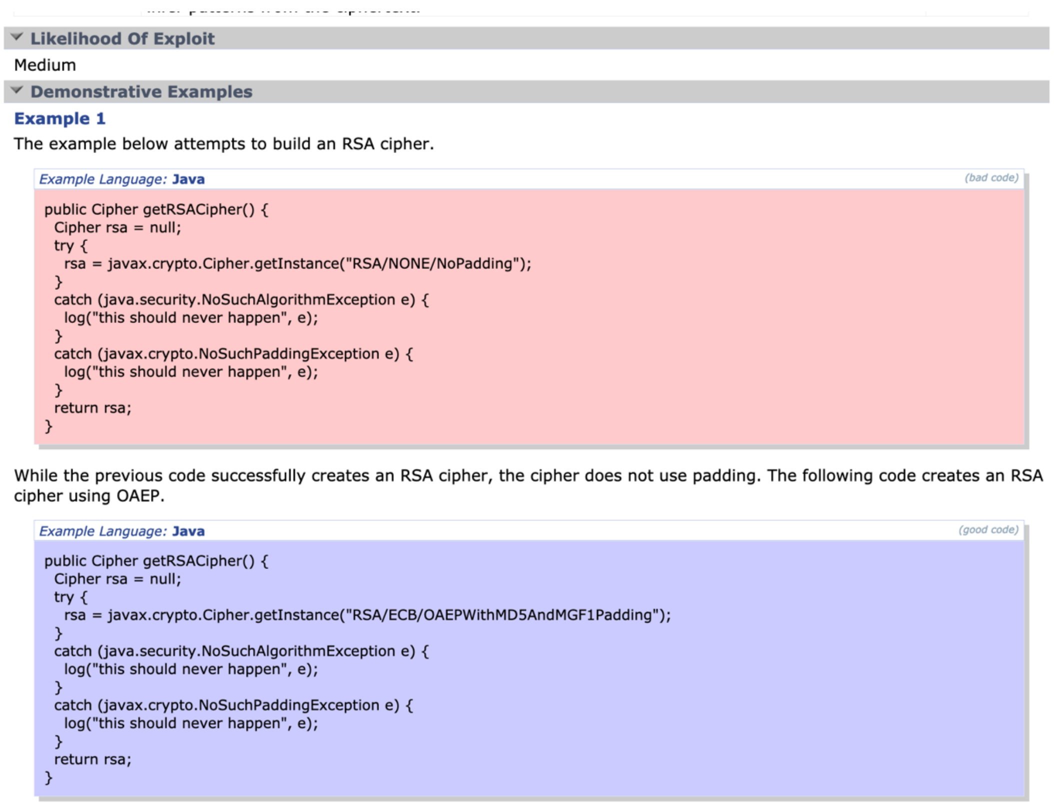
The software uses the RSA algorithm but does not incorporate Optimal Asymmetric Encryption Padding (OAEP), which might weaken the encryption.

Extended Description

Padding schemes are often used with cryptographic algorithms to make the plaintext less predictable and complicate attack efforts. The OAEP scheme is often used with RSA to nullify the impact of predictable common text.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOf and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that the user may want to explore.



infer patterns from the ciphertext.

Likelihood Of Exploit

Medium

Demonstrative Examples

Example 1

The example below attempts to build an RSA cipher.

Example Language: **Java** (bad code)

```
public Cipher getRSACipher() {
  Cipher rsa = null;
  try {
    rsa = javax.crypto.Cipher.getInstance("RSA/NONE/NoPadding");
  }
  catch (java.security.NoSuchAlgorithmException e) {
    log("this should never happen", e);
  }
  catch (javax.crypto.NoSuchPaddingException e) {
    log("this should never happen", e);
  }
  return rsa;
}
```

While the previous code successfully creates an RSA cipher, the cipher does not use padding. The following code creates an RSA cipher using OAEP.

Example Language: **Java** (good code)

```
public Cipher getRSACipher() {
  Cipher rsa = null;
  try {
    rsa = javax.crypto.Cipher.getInstance("RSA/ECB/OAEPWithMD5AndMGF1Padding");
  }
  catch (java.security.NoSuchAlgorithmException e) {
    log("this should never happen", e);
  }
  catch (javax.crypto.NoSuchPaddingException e) {
    log("this should never happen", e);
  }
  return rsa;
}
```

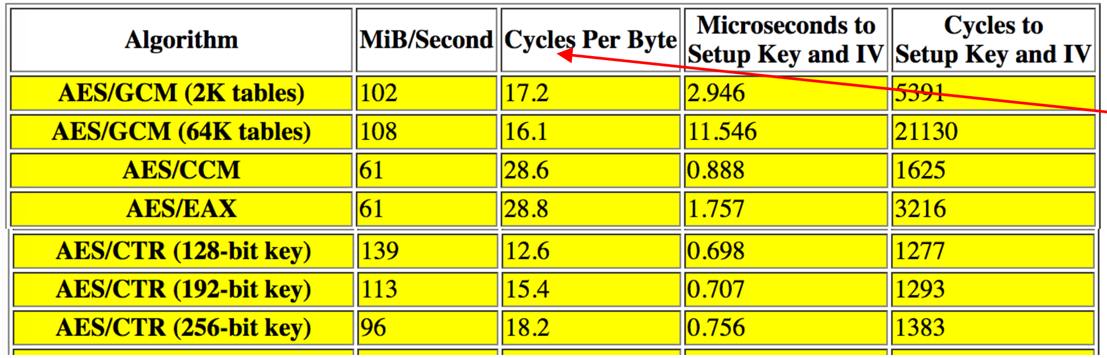
3.1.3. Efficiency

PKC is computationally expensive

- Following NIST recommendation, 128-bit AES and 3072-bit RSA has the equivalent key strength.
- RSA encryption/decryption is significantly slower than AES.

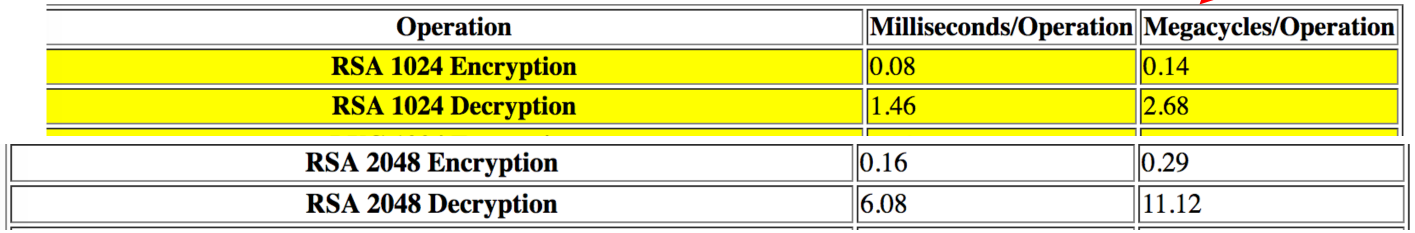
- See Crypto++ Benchmarks (crypto++ is an C++ library)

https://www.cryptopp.com/benchmarks.html



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.



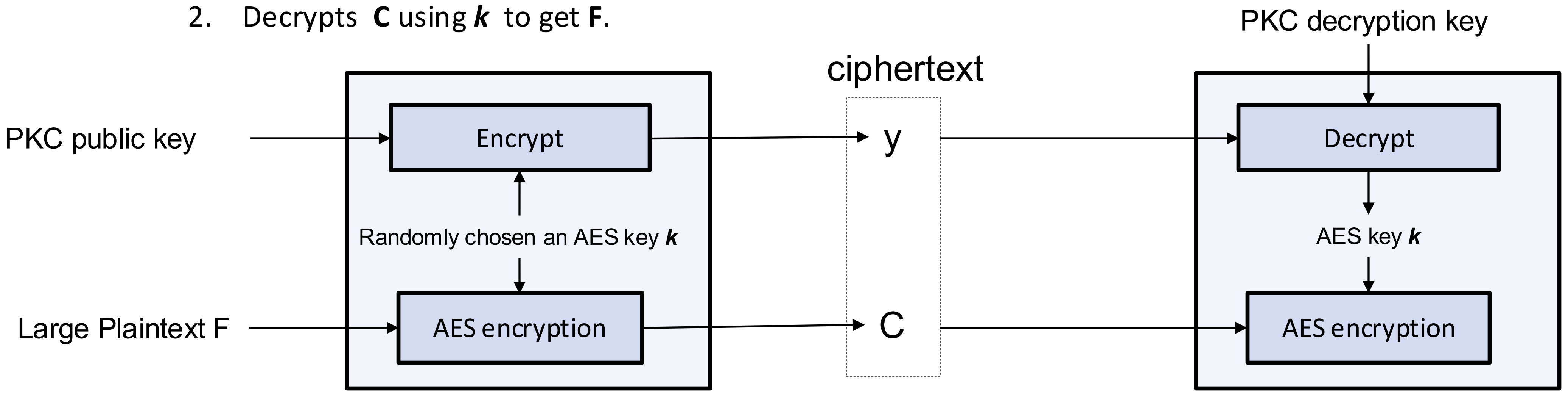
Operation	Milliseconds/Operation	Megacycles/Operation
RSA 1024 Encryption	0.08	0.14
RSA 1024 Decryption	1.46	2.68
RSA 2048 Encryption	0.16	0.29
RSA 2048 Decryption	6.08	11.12

How to efficiently encrypt using public key?

- If we want to encrypt a large file F, say a 7GB movie, it would be very slow to directly apply RSA (or other PKC) on F.

- To encrypt a large plaintext using PKC, it is typically carried out in this way:
  1. Chooses a random AES key ***k;***
  2. Encrypts ***k*** using PKC to get ***y;***
  3. Encrypts ***F*** using AES (with suitable mode, say GCM, CBC or CTR) with ***k*** as the key to get **C**.
  4. Output the (**y**, **C**) as the ciphertext.

- To decrypt,
  1. Decrypts ***y*** to get ***k*** using the PKC private key;
  2. Decrypts **C** using ***k*** to get **F**.

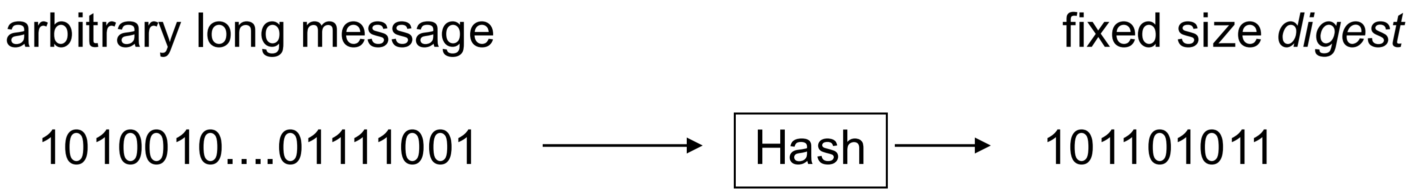


3.2. Data Authenticity (digest), unkeyed

We follow Kerchhoff’s principle. Adversary knows all the algorithms.

Hash (no secret involved!! Adversary knows the algo!!)

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



arbitrary long message

fixed size *digest*

1010010....01111001

Hash

101101011

Security requirement (collision-resistant):

- It is difficult for an attacker to find two different messages m_1 , m_2 that “hash” to the same digest. That is,

$$h(m_1) = h(m_2)$$

This is known as ***collision-resistant.***[*]

[*]: (optional) Formal security formulation of hash turns out to be complicated. A formal definition of collision resistant consists of a randomly chosen “key which is made public after chosen. Another formulation uses ”Random Oracle” which is convenient but not realizable. Attend higher-level of crypto course for more.

Security requirement of hash

Threat model:

- **Collision**

Find two different messages m_1 , m_2 s.t.

$$h(m_1) = h(m_2)$$

- **2^{nd} pre-image**

Given m_1 , find m_2 s.t.

$$h(m_1) = h(m_2)$$

- **Pre-image**

Given y, find m s.t.

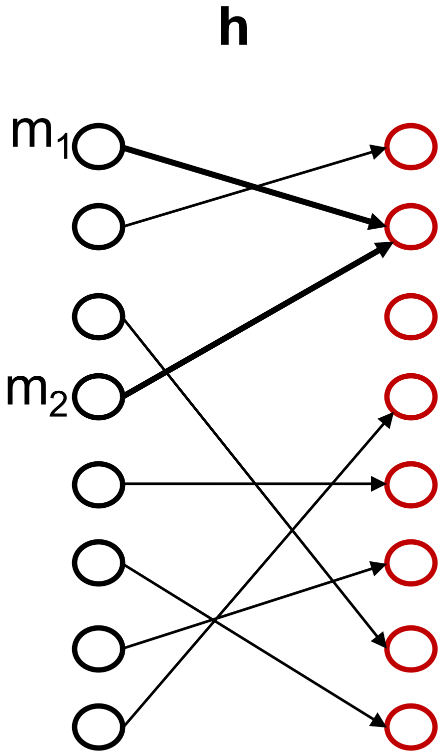
$$h(m) = y$$

Security requirement

Collision-Resistant

Second pre-image-Resistant

One-way



Example of hash algorithms that are not collision resistant

- Taking selected bits from the data.

- CRC checksum.

See https://en.wikipedia.org/wiki/Cyclic_redundancy_check

Popular Hash

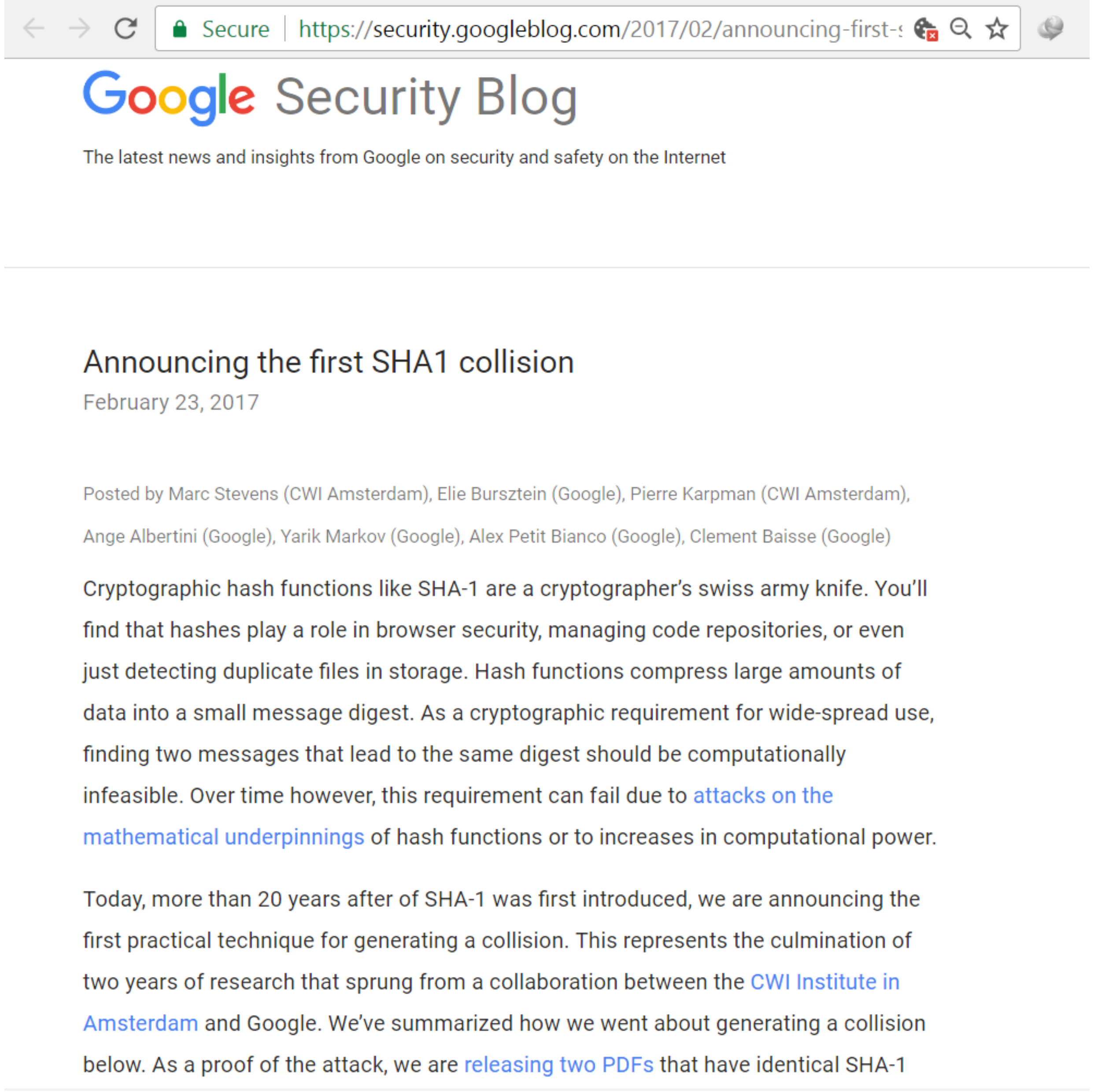
- SHA-0, SHA-1, SHA-2, SHA-3

- 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.
- SHA-1 is a popular standard. It produces 160-bits message digest. It is employed in SSL, SSH, etc.
- In 1998, an attack that finds collision of SHA-0 in 2^{61} operations was discovered. (Using the straight forward birthday attack, 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 attack that can finds collision in 2^{39} operations.
- 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-rounds SHA-256 (the full SHA-256 takes 64 rounds)
- In 2005, Xiaoyun Wang et al gave a method of finding collision of SHA-1 using 2^{69} operations, which was later improved to 2^{63} . A collision was found in 2017. It took 110 GPU years, completed 2^{63} SHA1 operations. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html
- In Nov 2007, NIST called for proposal of SHA-3. In Oct 2012, NIST announced the winner, Keccak (pronounced "catch-ack").

(NIST announcement http://www.nist.gov/itl/csd/sha-100212.cfm)

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

SHAttered
(shattered.io)
Feb 23, 2017



Secure | https://security.googleblog.com/2017/02/announcing-first-s

Google Security Blog

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

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)

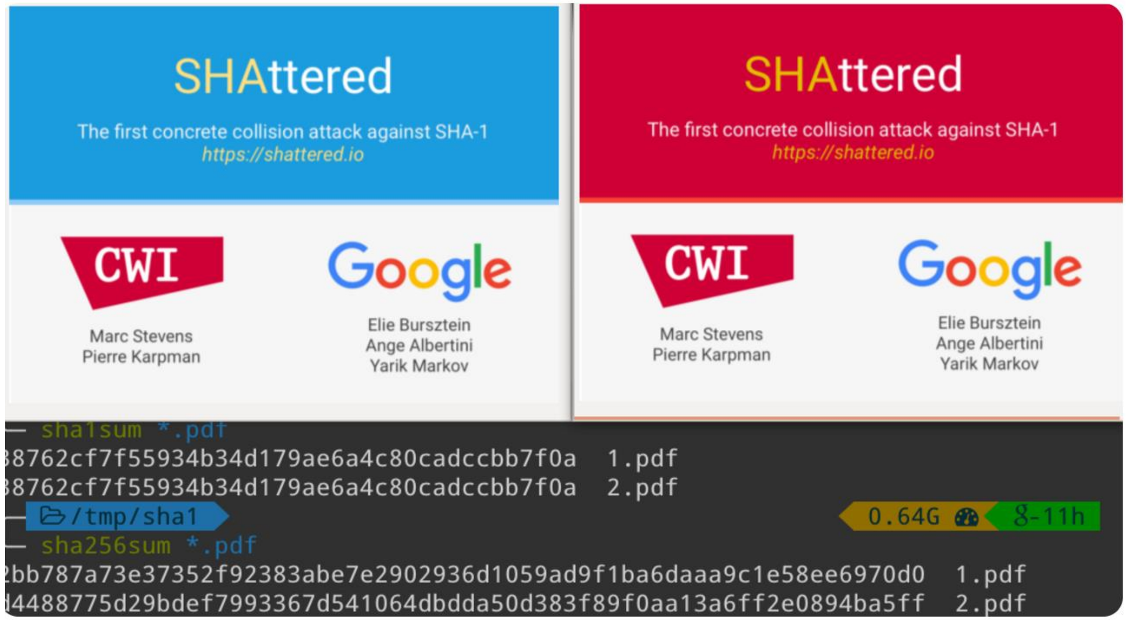
Cryptographic hash functions like SHA-1 are a cryptographer's swiss army knife. You'll find that hashes play a role in browser security, managing code repositories, or even just detecting duplicate files in storage. Hash functions compress large amounts of data into a small message digest. As a cryptographic 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

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 proof of concept:
  - https://shattered.io/static/shattered-1.pdf
  - https://shattered.io/static/shattered-2.pdf
- Defense mechanisms:
  - **Use SHA-256 or SHA-3 as replacement**

see
https://shattered.io/



```
─ sha1sum *.pdf
38762cf7f55934b34d179ae6a4c80cadccbb7f0a  1.pdf
38762cf7f55934b34d179ae6a4c80cadccbb7f0a  2.pdf
┌ 🗁/tmp/sha1                                          0.64G   8-11h
└ sha256sum *.pdf
2bb787a73e37352f92383abe7e2902936d1059ad9f1ba6daaa9c1e58ee6970d0  1.pdf
d4488775d29bdef7993367d541064dbdda50d383f89f0aa13a6ff2e0894ba5ff  2.pdf
```

- MD5

- Designed by Rivest. MD, MD2, MD3, MD4, MD5, 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.

An application scenario of unkeyed Hash (without secret keys)

Application of (unkeyed) Hash for integrity

Consider this example

- Alice downloaded a software `vlc-2.2.8-win32.exe` from the web. Is the downloaded file authentic?

  (see the "checksum" in next slide)

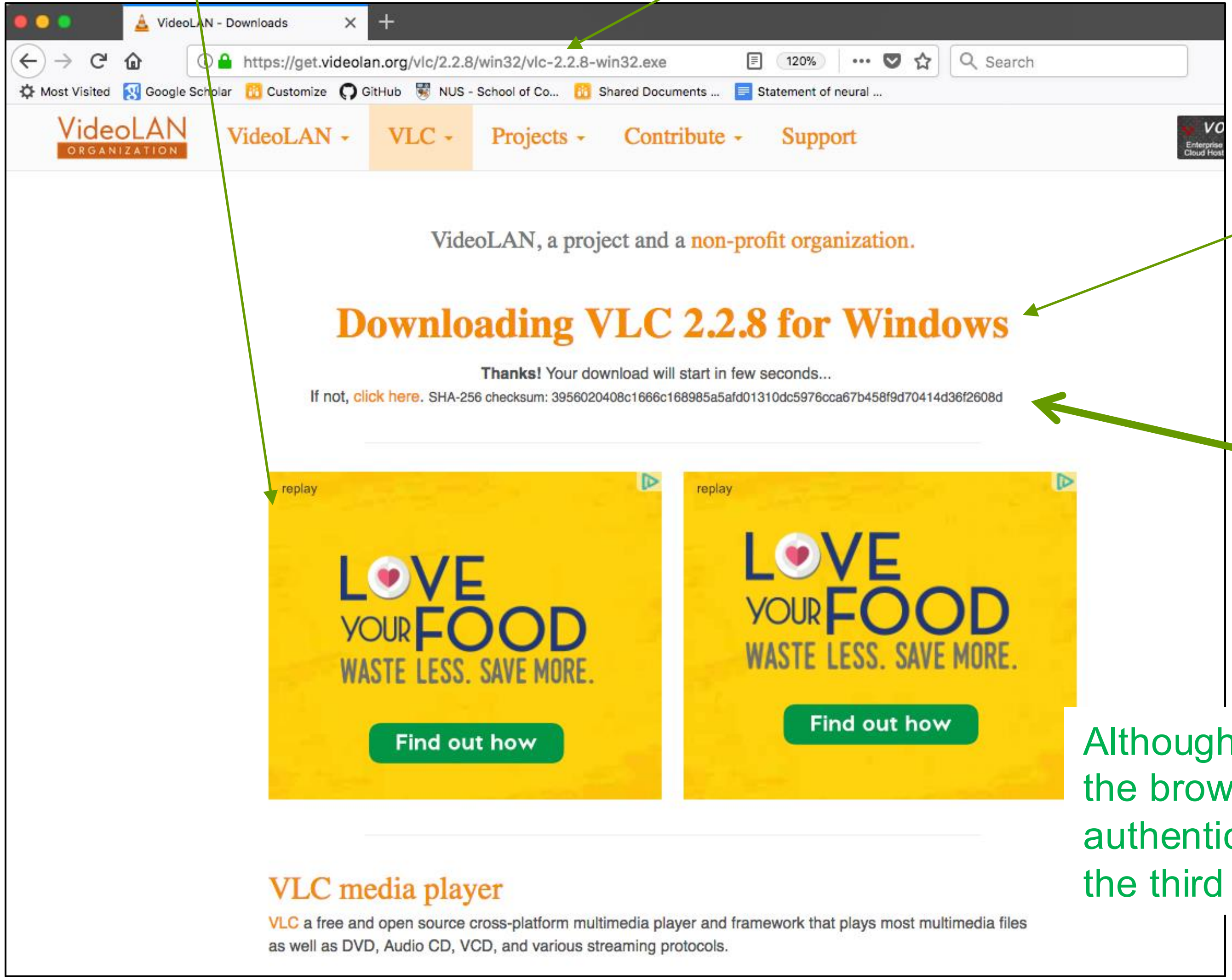
Specifically,

1. Alice visits the website of VLC.
2. Since the website is hosted with HTTPS protocol, Alice is being assured that the content displayed on the browser is from VLC and authentic. We will discuss https later and right now, let's treat all info displayed by browser as authentic.
3. However, the downloading site is a 3rd-party, i.e. the actual file `vlc-2.2.8-win32.exe` is hosted in another website. The communication channel to the 3rd party website is not secure. There is also a possibility that the 3rd party website is malicious and giving out virus infested software.
4. To verify that the file indeed is the original, after Alice downloaded the file, she can check the integrity of the file by matching the "hash" of the file with the "SHA-256 checksum" displayed in the browser. If they match, then Alice is very sure that the file is intact. If not, certainly the file is corrupted.

Ads from 3rd party.

(might not via https and videoland.org is unable to control what being shown)

https. So the content displayed here is from **videolan.org** and is authentic.



VideoLAN, a project and a non-profit organization.

Downloading VLC 2.2.8 for Windows

Thanks! Your download will start in few seconds...

If not, click here. SHA-256 checksum: 3956020408c1666c168985a5afd01310dc5976cca67b458f9d70414d36f2608d

VLC media player

VLC a free and open source cross-platform multimedia player and framework that plays most multimedia files as well as DVD, Audio CD, VCD, and various streaming protocols.

Actual file is hosted in a third party site

the digest (aka checksum).

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

Question: why "videolan.org" in the url is boldfaced? (help the user to identify the domain name, so that the user is less likely to fall prey to phishing attack.)

Application of (unkeyed) Hash for integrity

In this scenario, we assume that there is a secure channel to send short piece of information. (in VLC example, this channel is the https)

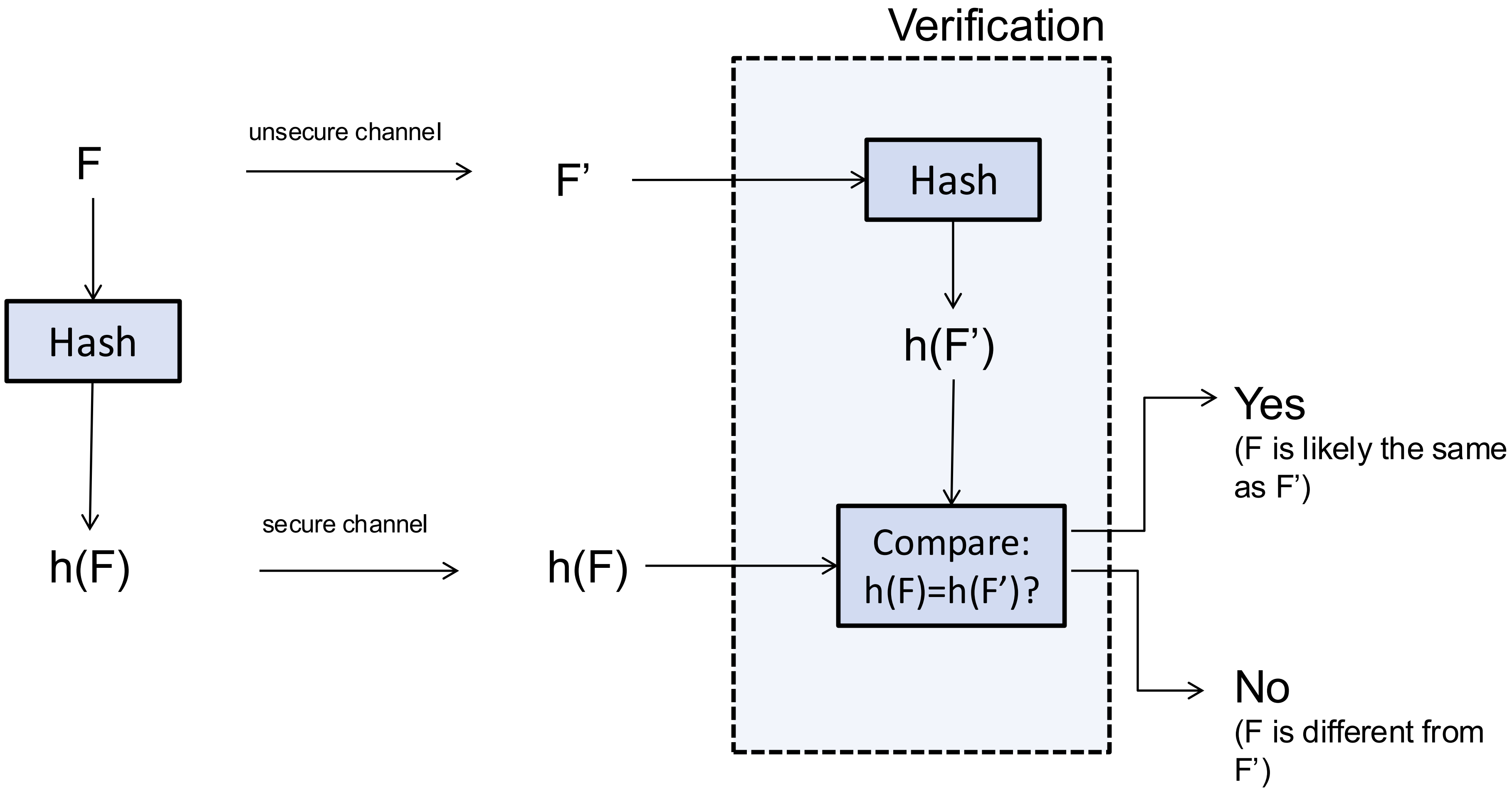
Let ***F*** be the original data. Alice obtains the digest h(***F***) from the secure channel.

Alice obtains a file, say ***F’***, whose origin claims that the file is F. Alice computes and compares the digests h(***F***), h(***F***’).

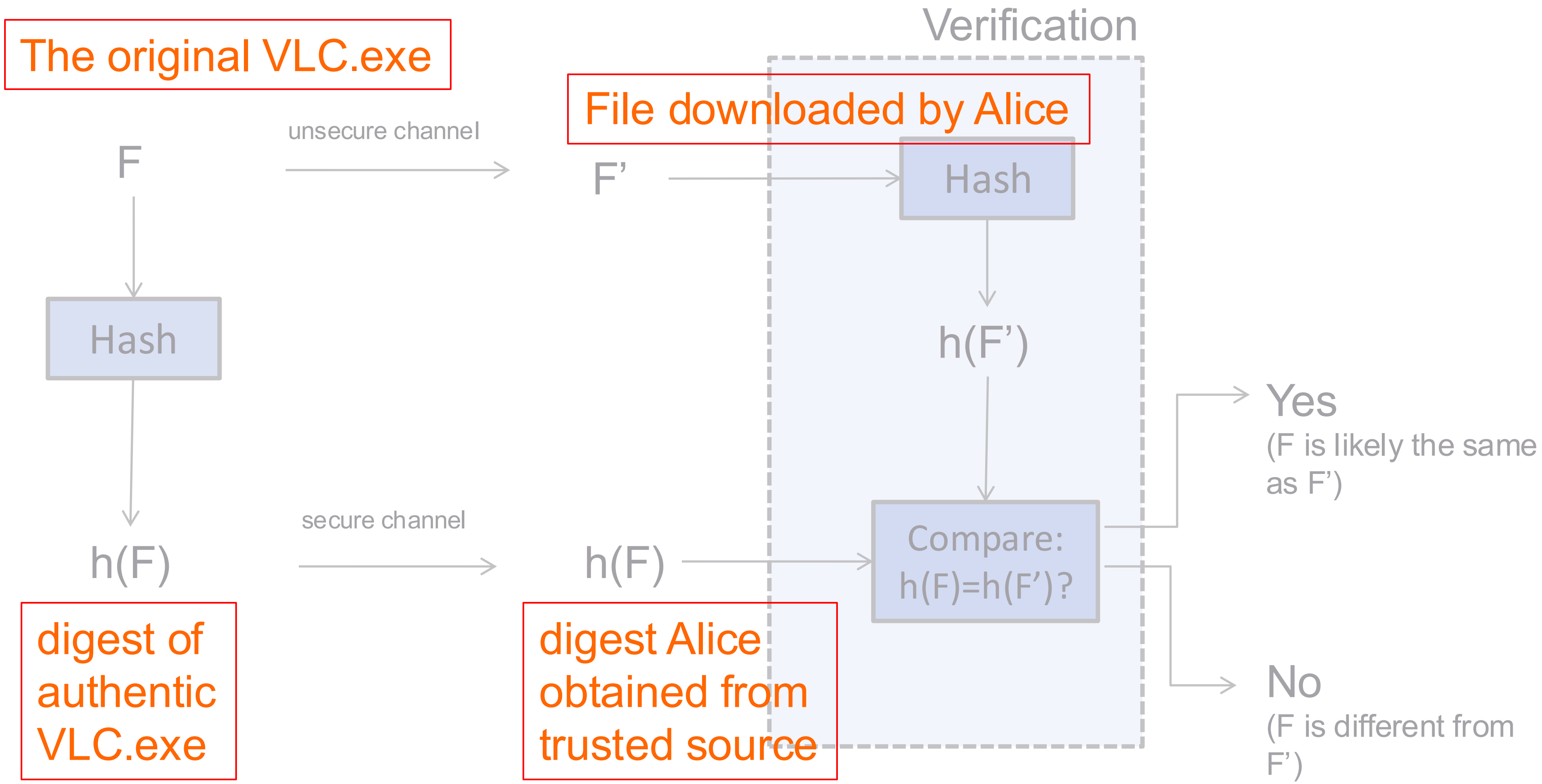
If they are the same, ***F’*** is indeed same as ***F*** with very high confidence. If they are different, then F’ must be different from F, i.e. integrity compromised.

h(***F***) = h(***F’***) ⇒ with high probability, ***F***=***F’***
h(***F***) ‡ h(***F’***) ⇒ ***F*** ‡ ***F’***

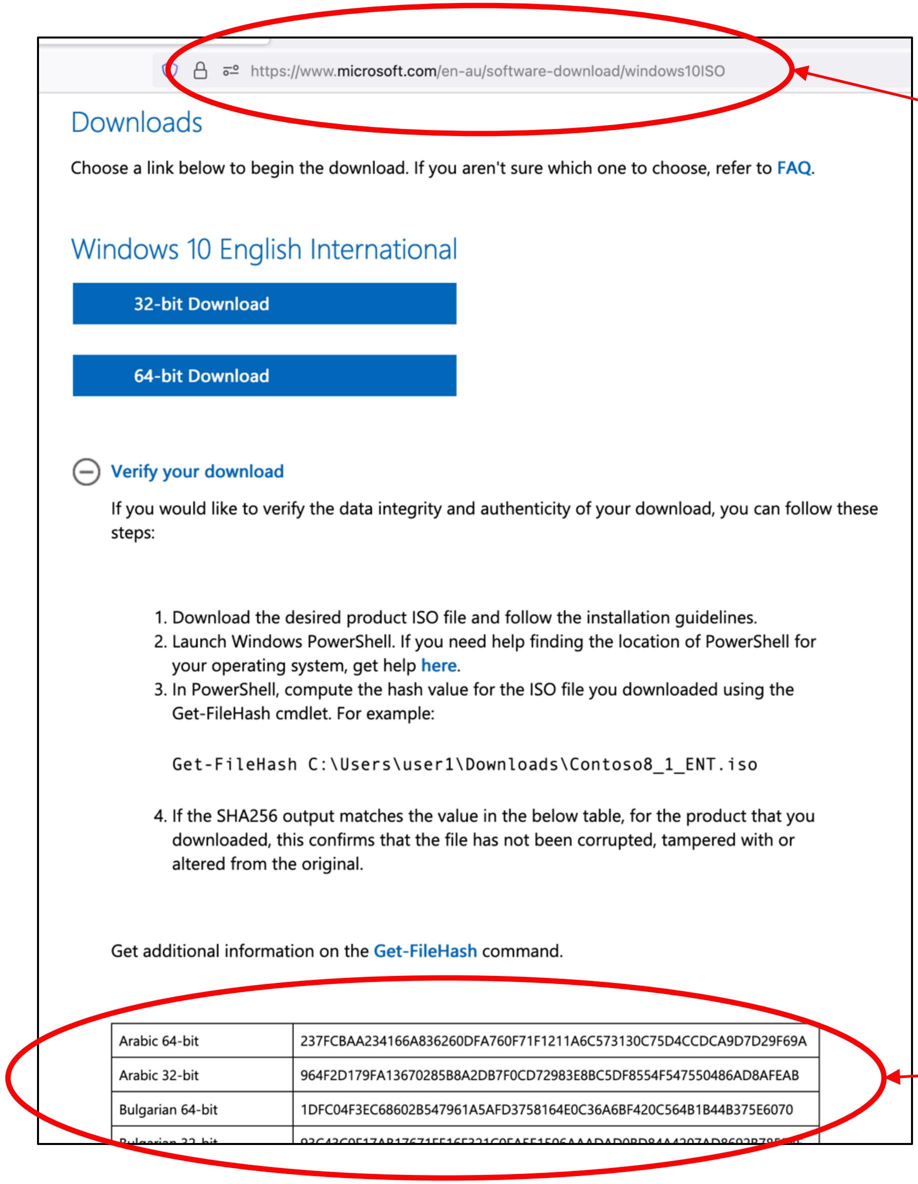
(unkeyed) hash



(unkeyed) hash



Similar setup in many other download sites..



https://www.microsoft.com/en-au/software-download/windows10ISO

Downloads

Choose a link below to begin the download. If you aren't sure which one to choose, refer to FAQ.

Windows 10 English International

Verify your download

If you would like to verify the data integrity and authenticity of your download, you can follow these steps:

1. Download the desired product ISO file and follow the installation guidelines.
2. Launch Windows PowerShell. If you need help finding the location of PowerShell for your operating system, get help here.
3. In PowerShell, compute the hash value for the ISO file you downloaded using the Get-FileHash cmdlet. For example:

```
Get-FileHash C:\Users\user1\Downloads\Contoso8_1_ENT.iso
```

4. If the SHA256 output matches the value in the below table, for the product that you downloaded, this confirms that the file has not been corrupted, tampered with or altered from the original.

Get additional information on the Get-FileHash command.

Arabic 64-bit	237FCBAA234166A836260DFA760F71F1211A6C573130C75D4CCDCA9D7D29F69A
Arabic 32-bit	964F2D179FA13670285B8A2DB7F0CD72983E8BC5DF8554F547550486AD8AFEAB
Bulgarian 64-bit	1DFC04F3EC68602B547961A5AFD3758164E0C36A6BF420C564B1B44B375E6070
Bulgarian 32-bit	[illegible]

https with padlock. So, content (other than 3rd party) displayed here are verified to be from an entity registered as microsoft.com with a "Certificate Authority" trusted by the browser, assuming that the running environment and browser is authentic.

In other words, assuming no malware in computer, the fact that we have a padlock assure us that info displayed is from microsoft.com

This course would cover the details in the above statement.

Downloading Window 10. instruction/explanation from Microsoft

The hashed value.

Security Requirement.

- What would an attacker do?
  - attacker’s goal: makes Alice accepts a file other than F.
  - Attacker knows F and can send any file F’ over. To trick the user, the attacker need to have a F’ s.t. H(F’)= H(F) and F’ ≠ F.

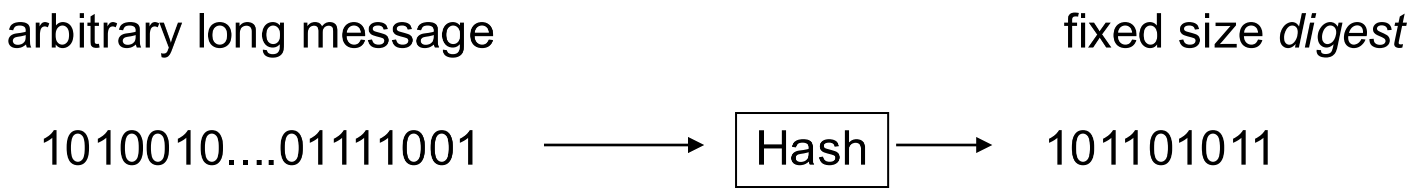
The above is an example of ***2nd preimage***:

In practical applications, for the attack to make sense, the attacker might need to find a F’ that meet certain properties. For e.g. F’ should be a workable malware. From the defender’s point of view, we want to cover as many possible attacks as possible. Hence, we focus on the modest (easiest) goal of finding any F (not necessary meeting the property), i.e. the 2nd preimage problem.

Is there a problem easier than 2nd preimage problem? Yes. “Collision”.

3.3 Data Origin Authenticity (mac), Keyed

Recap: Hash (no secret involved)



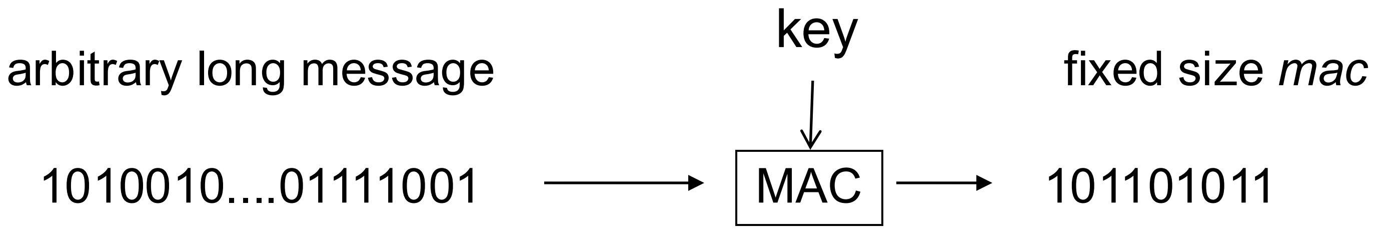
Security requirement (collision):

- ***(Collision-resistant)*** It is difficult for an attacker to find two different messages m_1 , m_2 that “hash” to the same digest. That is,

$$h(m_1) = h(m_2)$$

Keyed-Hash (aka MAC) (a secret key is involved)

A keyed-hash is a function that takes an arbitrary large message and a secret key as input, and outputs a fixed size (say 160 bits) ***mac (message authentication code)***.



- **Security requirement (forgery)**: After seen multiple valid pairs of messages and their corresponding mac, it is still difficult for the attacker to forge the mac of a message not seen before.

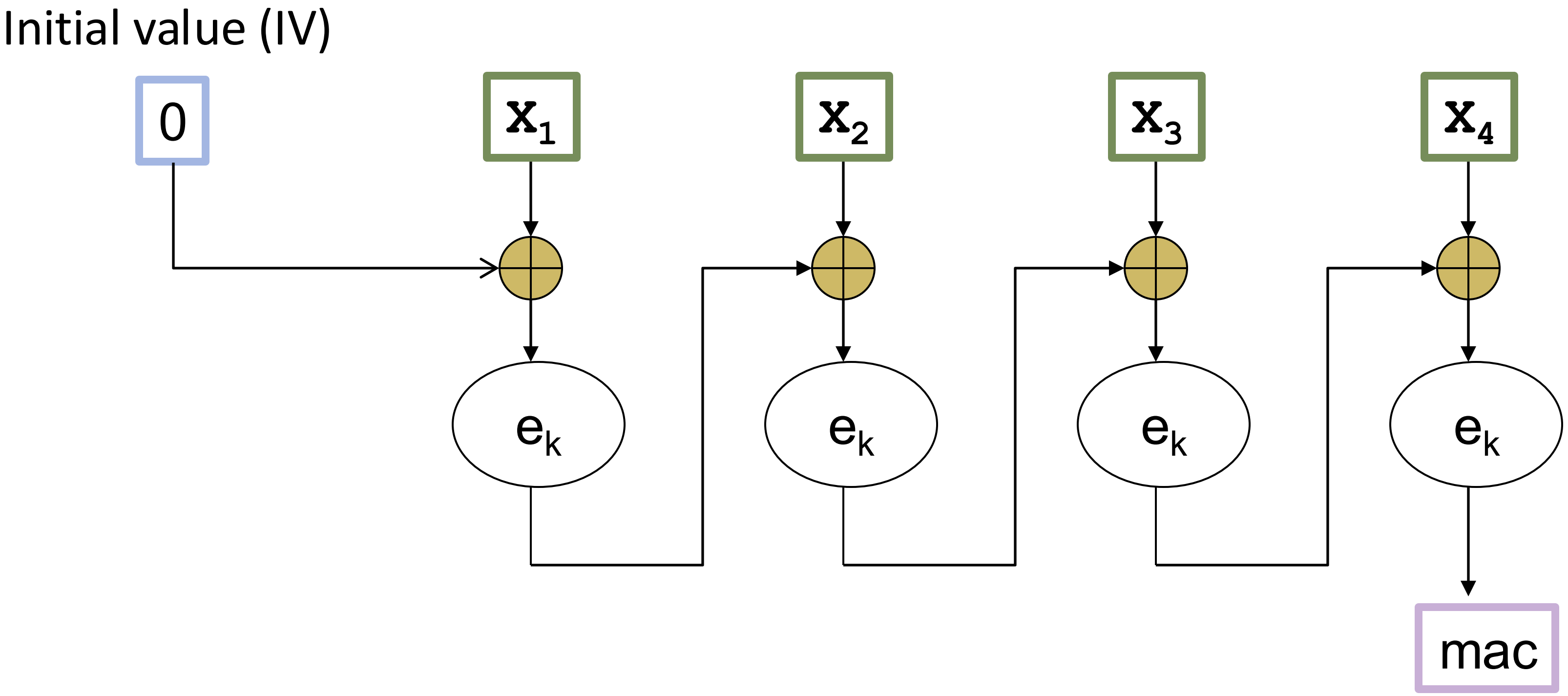
Popular keyed-hash (MAC)

- CBC-MAC (based on AES operated under CBC mode)
- HMAC (based on SHA)

  Hashed-based MAC

standard: RFC 2104. http://tools.ietf.org/html/rfc2104

CBC-mac



HMAC

Optional

$$\mathrm{HMAC}_k\,(x) = \textbf{SHA-1}\,(\,(\,K \oplus \mathit{opad}\,)\;||\;\textbf{SHA-1}\,(\,(\,K \oplus \mathit{ipad}\,)\;||\;x)\;)$$

where

opad = 3636…36 (outer pad)

ipad = 5c5c…5c (inner pad)

(the above are in hexadecimal)

An application scenario for mac

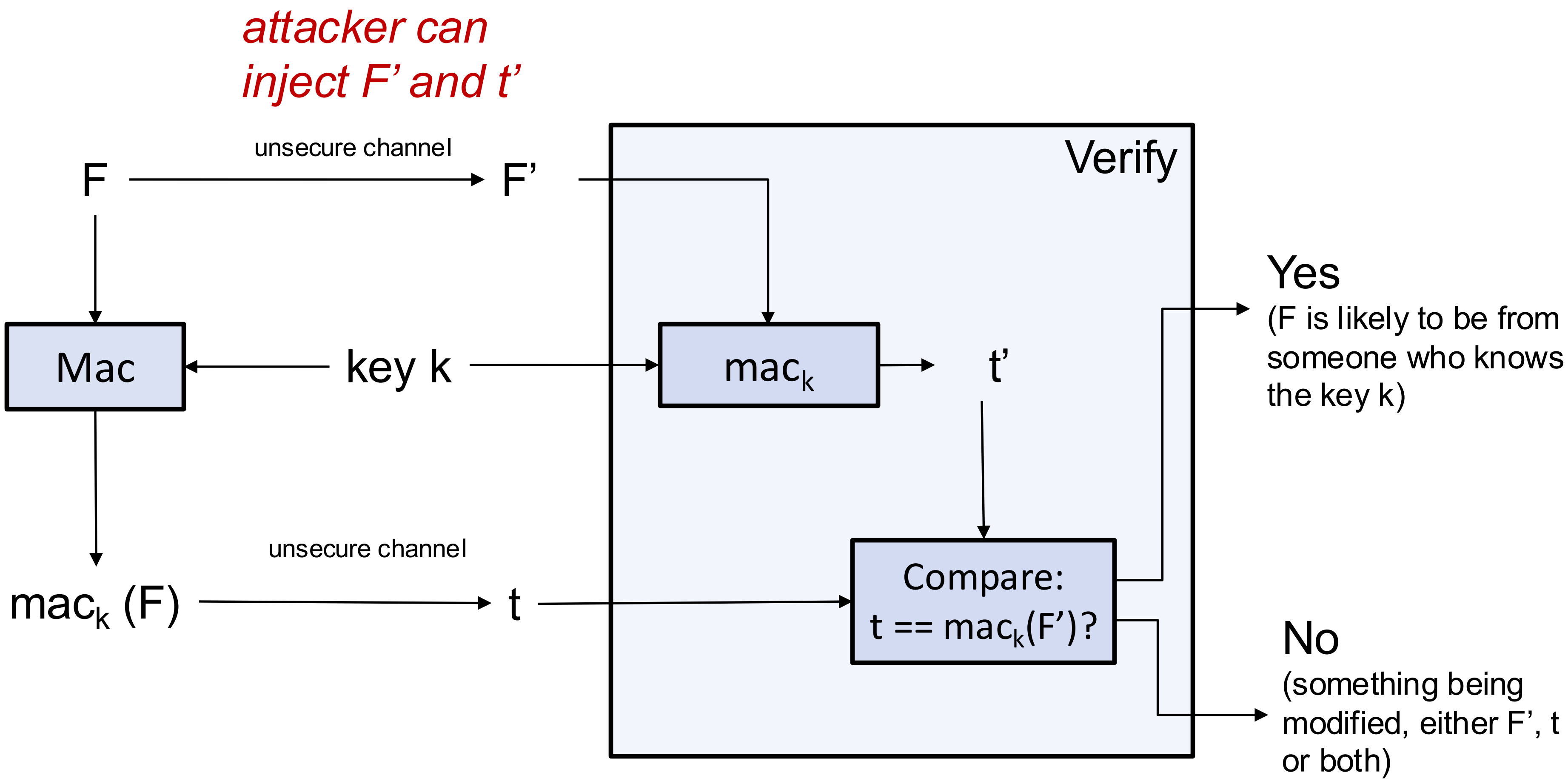
- In the previous example (on vlc), we assume that there is a secure channel to send the digest.

- There are scenarios where we don’t have a secure channel to deliver the digest. (in the vlc example, it relies on the fact that we have https. What if we don’t have https?)

- In such scenarios, we can protect the digest with the help of some secrets.
  - In the symmetric key setting, it is called the **mac.**
  - In the public key setting, it is called the **digital signature.**

mac (Message Authentication Code)

Note: Unlike the example on hash, the mac could be modified by attacker.


attacker can inject F' and t'
unsecure channel
F
F'
Verify
Mac
key k
mac_k
t'
Yes
(F is likely to be from someone who knows the key k)
unsecure channel
mac_k (F)
t
Compare:
t == mac_k(F')?
No
(something being modified, either F', t or both)

Security Requirement (forgery): After seen multiple valid pairs of messages and their corresponding mac, it is still difficult for the attacker to forge the mac of a message not seen before.

Remark

- Consider an attacker who saw F and mac(F). The attacker wants to find a F' and t' that are valid.

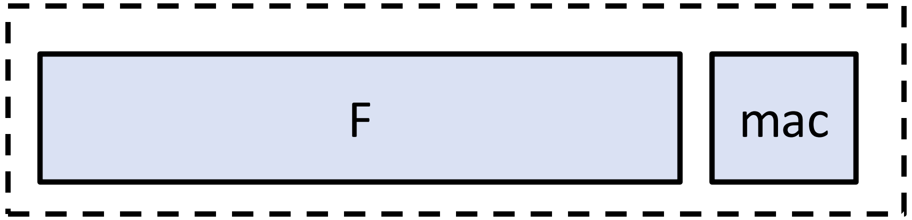
Recap that we want a method that can protect against attacker with strong capability. Hence, in the security requirement, we consider a threat model where the attacker have accesses to many pairs of message and mac:

(m_1, mac(m_1)), (m_2, mac(m_2)), …..

The attacker goal is to find a valid (m', t'), where m' is not one of the m_i.

- Note that there is no issue on confidentiality. In fact, the data F can be sent in clear.

- Typically, the *mac* is appended to F. Hence, mac is also called the *authentication tag,* or *authentication code.*



3.4 Data Origin Authenticity (Signature), Asymmetric key

Signature

- The public key version of MAC is called Signature.

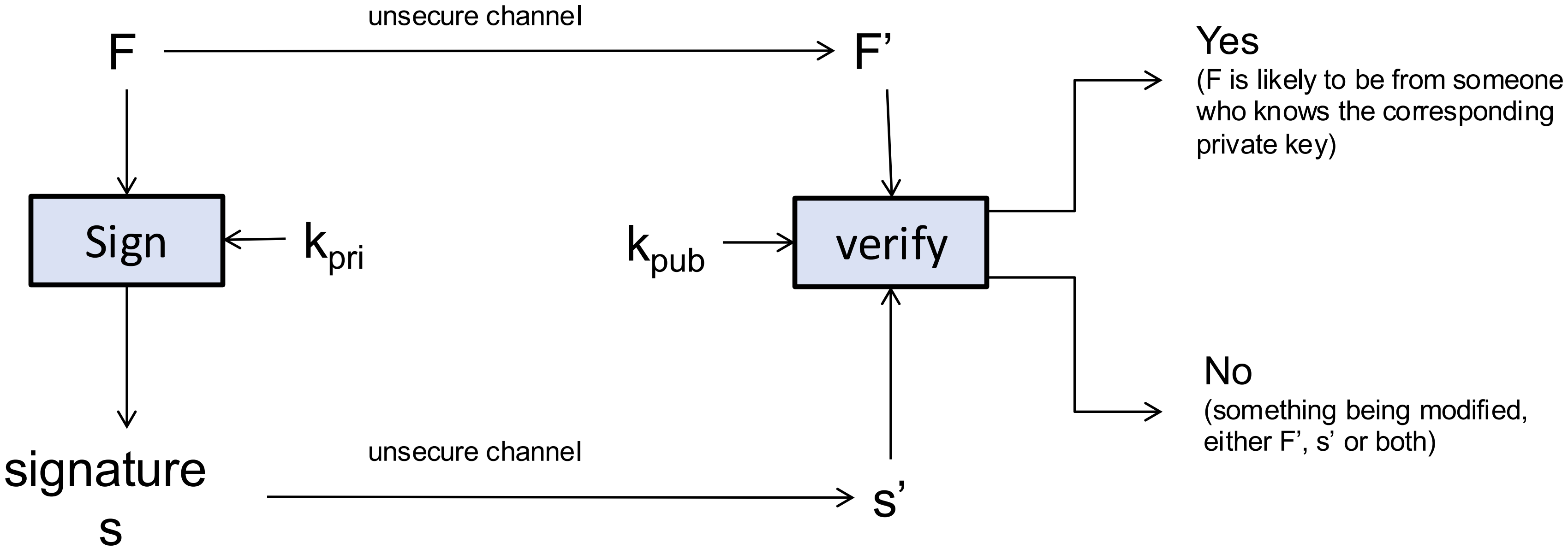
- Here, the owner uses the ***private key*** to generate the signature. The public can use the ***public key*** to verify the signature.

- So, anyone can verify the authenticity of the data, but only *the person who know the private key* can generate the signature.

Signature

Verifier and Signer using different key.

k_{pri}: private key
k_{pub}: public key

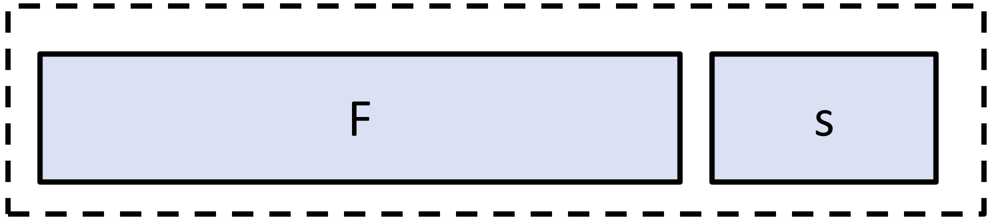


unsecure channel
F
F'
Yes
(F is likely to be from someone who knows the corresponding private key)
Sign
k_{pri}
k_{pub}
verify
No
(something being modified, either F', s' or both)
signature
s
unsecure channel
s'

Security Requirement: After seen multiple valid pairs of messages and their corresponding signature, it is still difficult for the attacker to forge the signature of a message not seen before. Attacker knows the public key.

Remark

- Likewise, the computed signature is typically appended to F.



- 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 public key. The valid signature ***can only be*** computed by someone who knows the private key. So, if it is valid, then F must be authentic.

- Anyone can verify the signature using the public key.

What so special of signature compared to mac?

- Public key has the advantage that we only need a secure broadcast channel to distribute the key.

- Beside the above, signature achieves additional security requirements.

- We can view the digital signature as handwritten signature in legal document. A legal document is authentic if it has the correct handwritten signature. No one, except the authentic signer, can generate the signature. Hence, the signer cannot repudiate, ie. deny having signed the document.

- Signature scheme achieves ***Non-repudiation.***

Non-Repudiation*: Assurance that someone cannot deny previous commitments or actions.*

Example on non-repudiation

- (using mac). Suppose Alice sent Bob a message appended with a mac. The mac is computed with a 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 signature). 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. This is because only the person who knows the private key can sign the message and only Alice knows the private key. Hence, the signature is a proof that Alice generated the message.

Popular Signature scheme

- A popular group of schemes use RSA for the sign/verify component. :

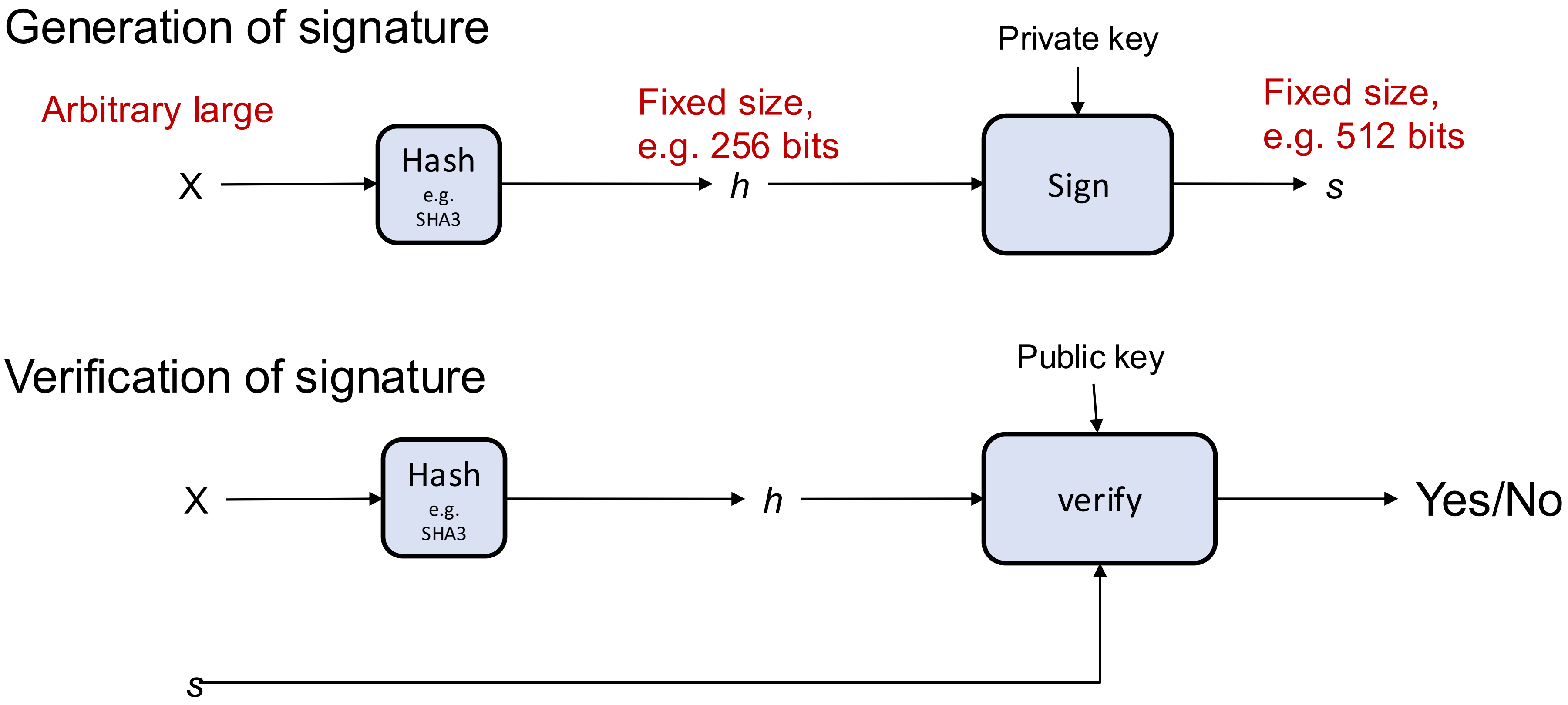
  RSASSA-PSS, RSASSA-PKCS1: signature scheme based on RSA

- DSA (Digital Signature Algorithm) is another popular standard whose security depends on discrete log.

  https://en.wikipedia.org/wiki/Digital_Signature_Algorithm#Implementations

Design of Signature scheme

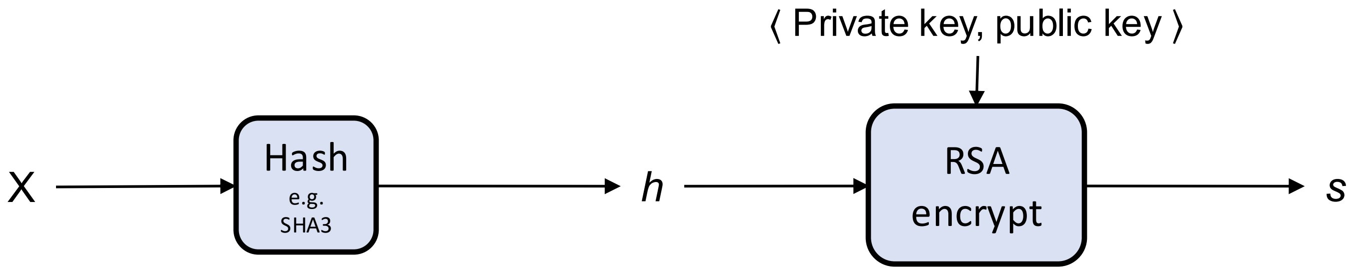
- Signature schemes typically consist of two components. An unkeyed hash, and the sign/verify algorithm.



RSA-based signature

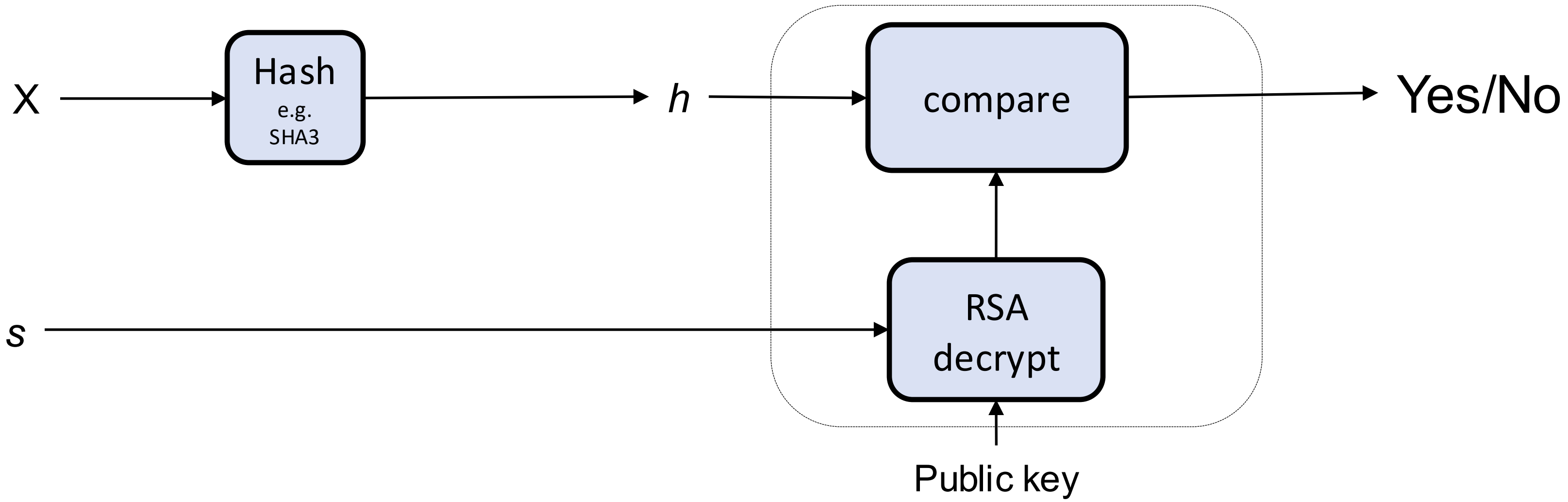
- Use RSA for signing and verification. Essentially, the signature is the “encrypted” digest. During verification, decrypt to obtain the digest and compare. *A messy notation issue: Previously, we use public key to encrypt. Here, we use private key to encrypt. Recall that for RSA, we can flip the role.*

$$\boldsymbol{s} = \text{RSA\_enc}\ (\ \langle\text{private key, public key}\rangle,\ \ \text{Hash}(X)\ )$$



- Verification of (X,***s***) is done by using the public key.

  If Hash (X) = RSA_dec (public key, s) then accept, else reject.



Not necessary to have “encryption” in signature scheme

- We employ Hash-and-Encrypt on RSA to obtain signature. Note that Hash-and-Encrypt is a special way to obtain signatures. Not all signature scheme use Hash-and-Encrypt. Also recap that RSA has a property that encrypt/decrypt can be flip.

- Popular schemes such as DSA do not “encrypt”.

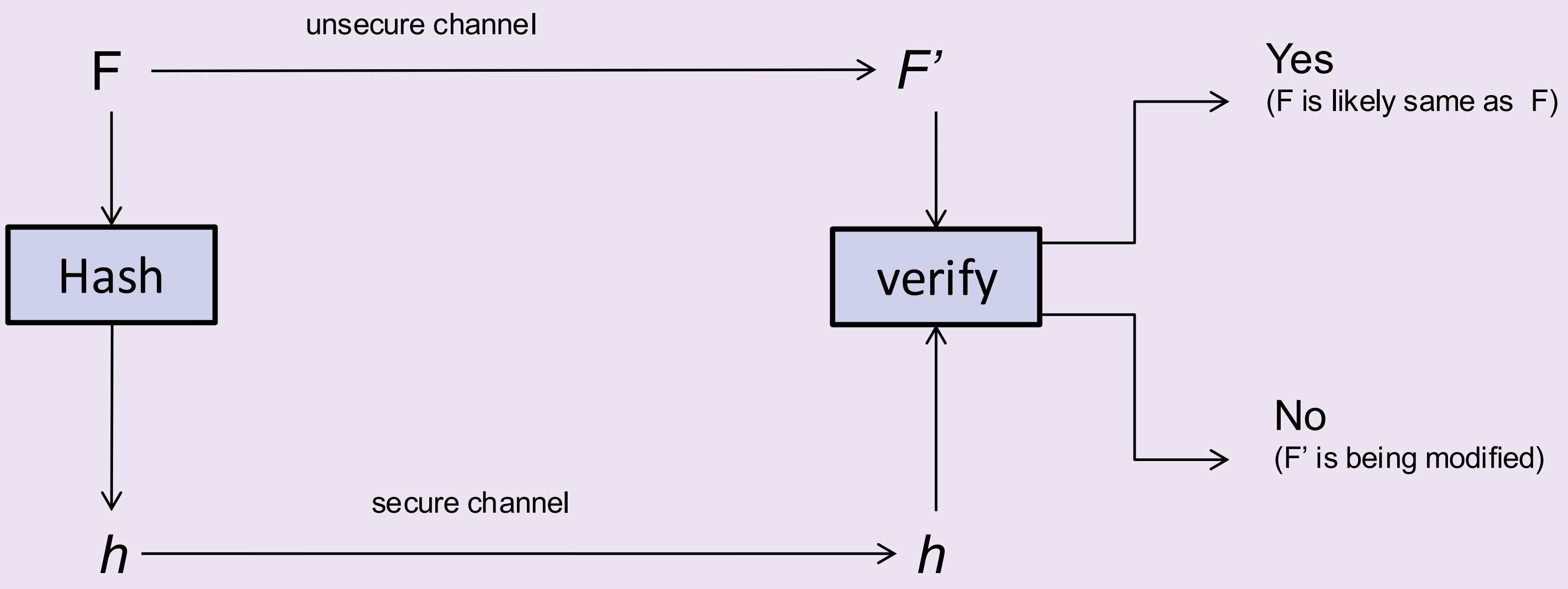
- Many books/documents describe a signature as the encryption of the hash and call all methods “hash-and-encrypt”. IMHO, this description is wrong. (due to the above point.)

- A more accurate term would be: “hash-and-sign”

Summary

Digest (Hash)

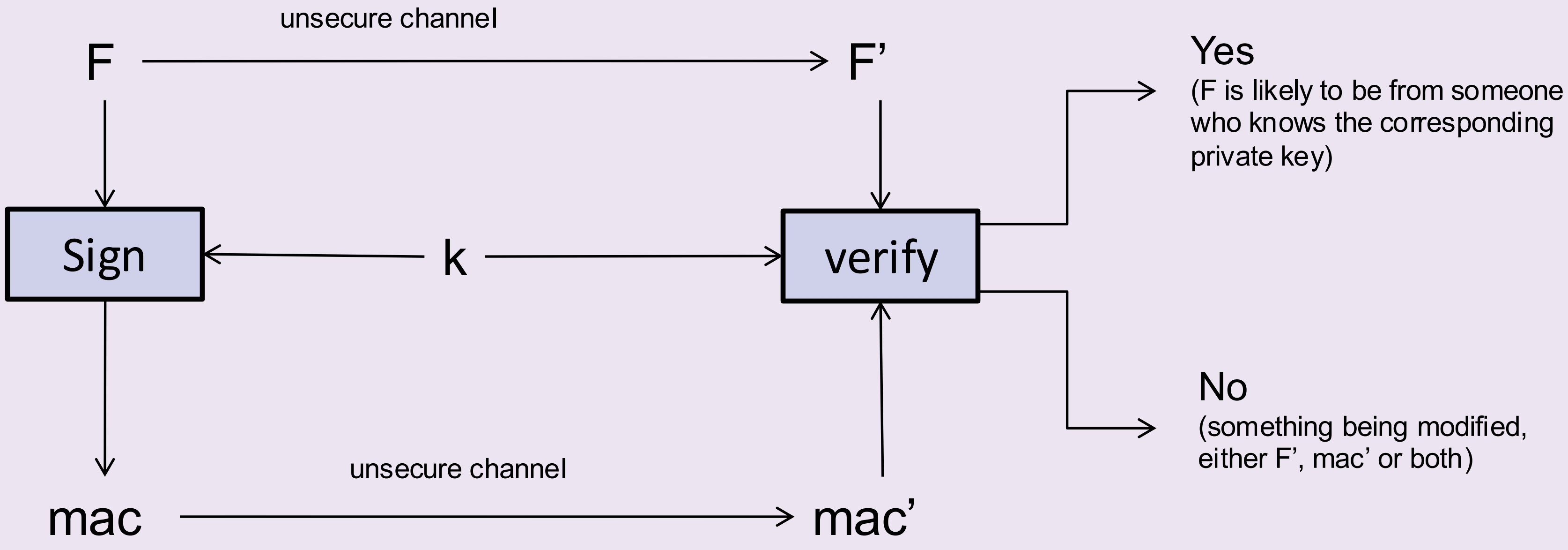
The digest must be sent through secure channel



Security Requirement: Different to find a pair F, F' with the same digest

MAC (Message Authentication Code)

The mac can be sent through an unsecure channel. Both

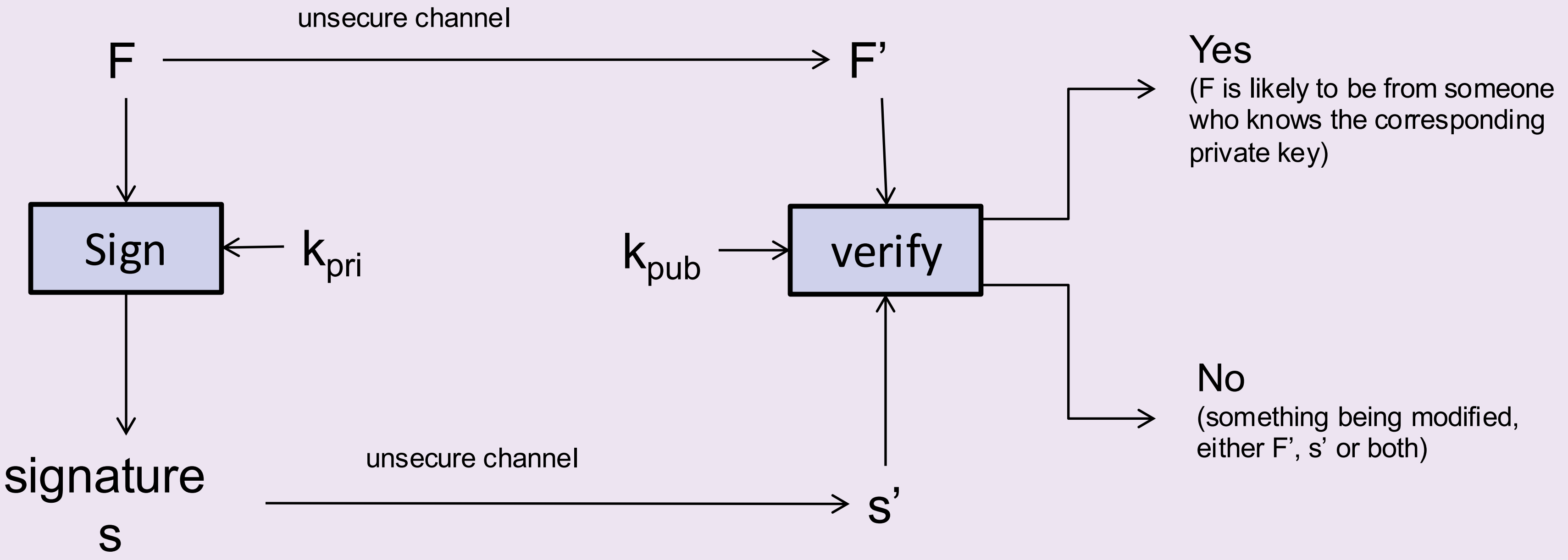

unsecure channel
F
F’
Yes
(F is likely to be from someone who knows the corresponding private key)
Sign
k
verify
No
(something being modified, either F’, mac’ or both)
unsecure channel
mac
mac’

Security Requirement: Without knowing *k*, it is difficult to forge a mac, even after seen many pairs of messages and their valid mac.

Signature

Verifier and Signer using different key.

k_{pri}: private key
k_{pub}: public key



unsecure channel
F
F'
Yes
(F is likely to be from someone who knows the corresponding private key)
Sign
k_{pri}
k_{pub}
verify
No
(something being modified, either F', s' or both)
signature
s
unsecure channel
s'

Security Requirement: Without knowing the private k_{pri}, it is difficult to forge a signature, even after seen many pairs of messages and their valid signatures.

3.5. Some attacks and pitfalls

3.5.1 Birthday attacks

This attack is like “exhaustive search” in encryption. Birthday attack can be applied to all hash functions, similar to exhaustive search on all encryption schemes.

We want to design a hash so that known attacks can’t do better than birthday attack.

This attack illustrates why 256-bit digest is required for hash, when 128-bit encryption is considered sufficient for encryption.

- Hashes are designed to make collision difficult to find. Recall that a collision consists of two different messages x_1, x_2 that give the same digest, i.e.

  $h(x_1) = h(x_2)$ and x_1 ‡ x_2

- Any hash function is subjected to **birthday attack**. (similar to exhaustive search on encryption scheme).

A straightforward method to find collision

Suppose H() is a hash with k-bit digest.

Find collision:

1. Randomly pick two messages m_1, m_2.
2. If $H(m_1) = H(m_2)$, then output m_1, m_2 and halt.
3. Repeat 1 to 3.

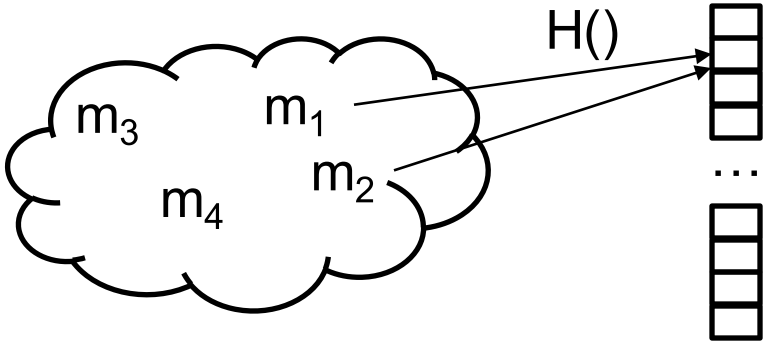
The expected number of rounds taken by the above algorithm is 2^k and thus the expected number of hashes is more than 2^k. If k =128, as analyzed in tutorial one, this is computationally infeasible. But …..

Why? Every round would have probability of 2^{-k} to halt. Let X to be the number of rounds the algorithm takes. This is "Geometric Distribution" with parameter $p=2^{-k.}$ The expected number of rounds is 1/p, which is $2^{k.}$

Birthday attack

Suppose H() is a hash giving k-bit digest.

1. Constructs a set S of $M=\lceil 1.17 * 2^{k/2} \rceil$ unique randomly chosen messages.
2. Compute the digest of each message m in S.
3. Check whether there are two messages in S having the same digest. If so, output m_1, m_2 . Otherwise, output “Fail”.



When k=128, one round would take ~ 2^{64} hashes, significantly lower than before. Next few slides show that with high probability, the above succeeds.

Birthday attack

In a class of 25 students, with probability more than 0.5, there is a pair of students having the same birthday.

- Suppose we have M messages, and each message is tagged with a value randomly chosen from {1,2,3,…,T}.

- The probability that there is a pair of messages tagged with the same value is approx.:

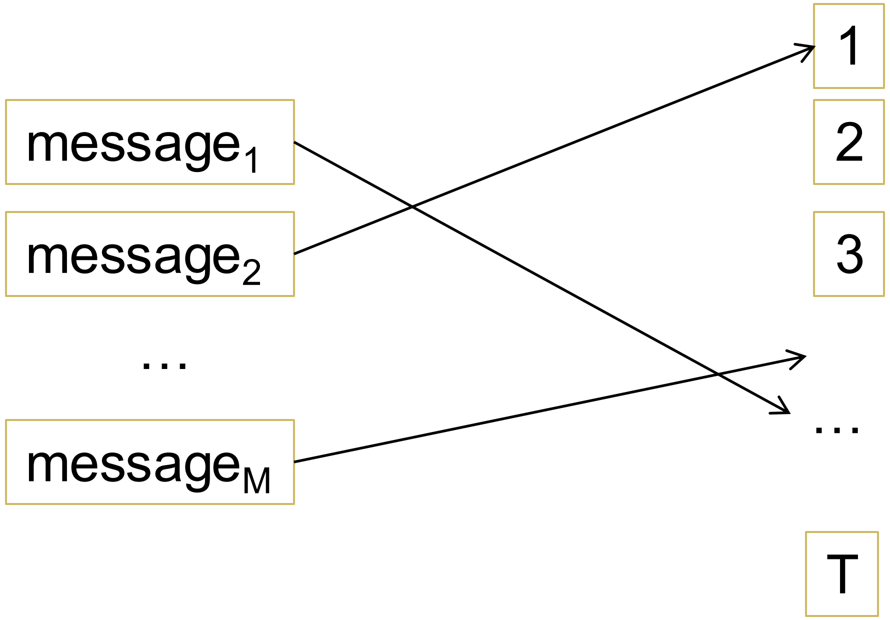
  Prob(collision) ≈ 1- exp (- M^2/ (2T))

$$\boxed{1 - e^{\left(-\frac{M^2}{2T}\right)}}$$

- In particular, when

  $M > 1.17 \quad T^{0.5}$

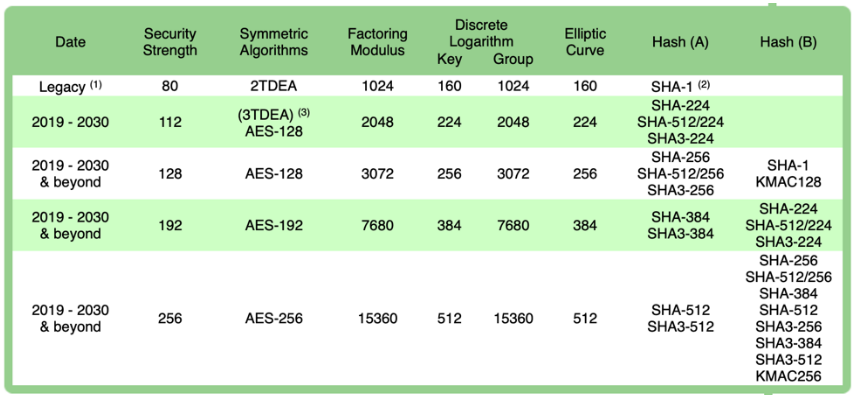
then, Prob (Collision) > 1-exp (1.17/2) > 0.5.



Relate to Hash & Birthday attack:

- Suppose the hash gives 128-bit digest.
- Then T = 2^{128}
- By choosing M = 1.17 2^{64}, with probability more than 0.5, birthday attack succeed.

Remark



Date	Security Strength	Symmetric Algorithms	Factoring Modulus	Discrete Logarithm		Elliptic Curve	Hash (A)	Hash (B)
				Key	Group			
Legacy (1)	80	2TDEA	1024	160	1024	160	SHA-1 (2)	
2019 - 2030	112	(3TDEA) (3) AES-128	2048	224	2048	224	SHA-224 SHA-512/224 SHA3-224	
2019 - 2030 & beyond	128	AES-128	3072	256	3072	256	SHA-256 SHA-512/256 SHA3-256	SHA-1 KMAC128
2019 - 2030 & beyond	192	AES-192	7680	384	7680	384	SHA-384 SHA3-384	SHA-224 SHA-512/224 SHA3-224
2019 - 2030 & beyond	256	AES-256	15360	512	15360	512	SHA-512 SHA3-512	SHA-256 SHA-512/256 SHA-384 SHA-512 SHA3-256 SHA3-384 SHA3-512 KMAC256

Recommended digest length

- Recap the NIST key-length recommendation (http://www.keylength.com/en/4/)

- When key length for symmetric-key is 112, the corresponding recommended length for digest is 224. Why the digest length is twice larger? *(birthday attack)*

(optional, for those who like CS3230) Improved birthday attack with constant memory

- Birthday attack needs large memory to store the digests. Interestingly, using ***Cycle Detection***, only constant size memory is required.

https://en.wikipedia.org/wiki/Cycle_detection#Tortoise_and_hare

3.5.2 Common pitfall: Using encryption for an application that needs authentication

Encryption schemes may provide false sense of security. Consider this design of a mobile apps from a 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 sms only consist of readable ascii characters. We assume that there is a way to encode binary string using the readable characters). The format of the instruction is:

X P

where X is an 8-bit string specifying the operation, and P is a 120-bit string specifying the parameter. So, an instruction is of size 128 bits. If an operation doesn't need a parameter, P will be ignored. There is a total of 15 valid instructions.

E.g.

00000000 P : send the GPS location to phone number P via sms. If P is not a valid phone number, ignore.

11110000 P : rings for P seconds. If P>10, ignore.

10101010 : self-destruct now!

- An instruction is to be encrypted using AES CBC-mode with 256-bit key, encoded to readable characters and sent as sms. (recap: block size of AES is 128 bits).

- After a mobile phone received a sms, it decrypts it. If the instruction is invalid, it ignores the instruction. Otherwise, it executes the instruction.

- The company XYZ claims that "*256-bit 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*".

- Something is wrong here.

(If an attacker sends a randomly chosen message to the mobile phone, what is the probability that the mobile phone self-destruct?)

Remarks

- Encryption is designed to provide confidentiality. It does not necessary guarantee integrity and authenticity.
- In the previous example, XYZ wants to achieve “authenticity”, but wrongly employed encryption to achieve that. (Some encryption schemes also provide authenticity, but not all.)
- A secure design should use schemes for authenticity instead of encryption.
- Furthermore, this application requires “communication authenticity”, not just “data-origin authenticity”.