Fundamental Concepts (CS-352)

Basic Terminology

Some Basic Terminology

- Plaintext: original message
- Ciphertext: coded message
- Enciphering (encryption): converting plaintext to ciphertext
- Deciphering (decryption): restoring the plaintext from ciphertext
- Cryptography: the area of study of encryption principles/methods
- Cipher: an algorithm for performing encryption
- Cryptanalysis: the area of study of principles/ methods of deciphering ciphertext without knowing key – breaking the cipher

Some Basic Terminology

- Cryptology: areas of cryptography and cryptanalysis together
- Secret key: the input of encryption algorithm. The key is independent of the plaintext and the algorithm

Cryptography

- Cryptographic system is characterized by:
  - The type of encryption operations used
    - Substitution: each element (a bit or a letter) in the plaintext is mapped into another element
    - Transposition: elements in the plaintext are rearranged.
    - Product: multiple stages of substitutions and transpositions
  - The number of keys used
    - Symmetric, Single-key encryption
    - Asymmetric, Two-key or Public-key encryption
  - The way in which plaintext is processed
    - Block: process one block of elements at a time, producing an output block for each input block
    - Stream: process the input elements continuously, producing one element at a time.

Symmetric Key Cryptography Overview

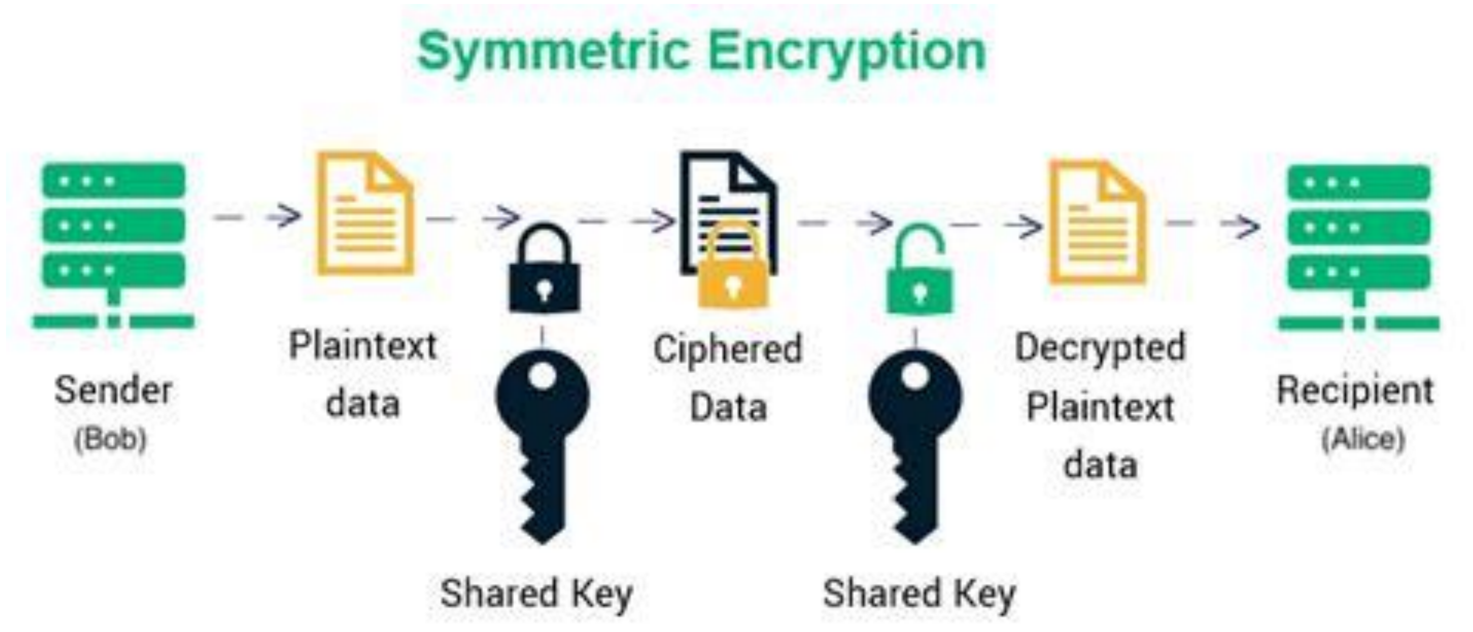
Symmetric Encryption

- A form of cryptosystem in which encryption and decryption are performed using the same key – single-key encryption
- Was only type prior to invention of public-key in 1970's, and by far most widely used

Symmetric Encryption

- Two parties share the same secret key

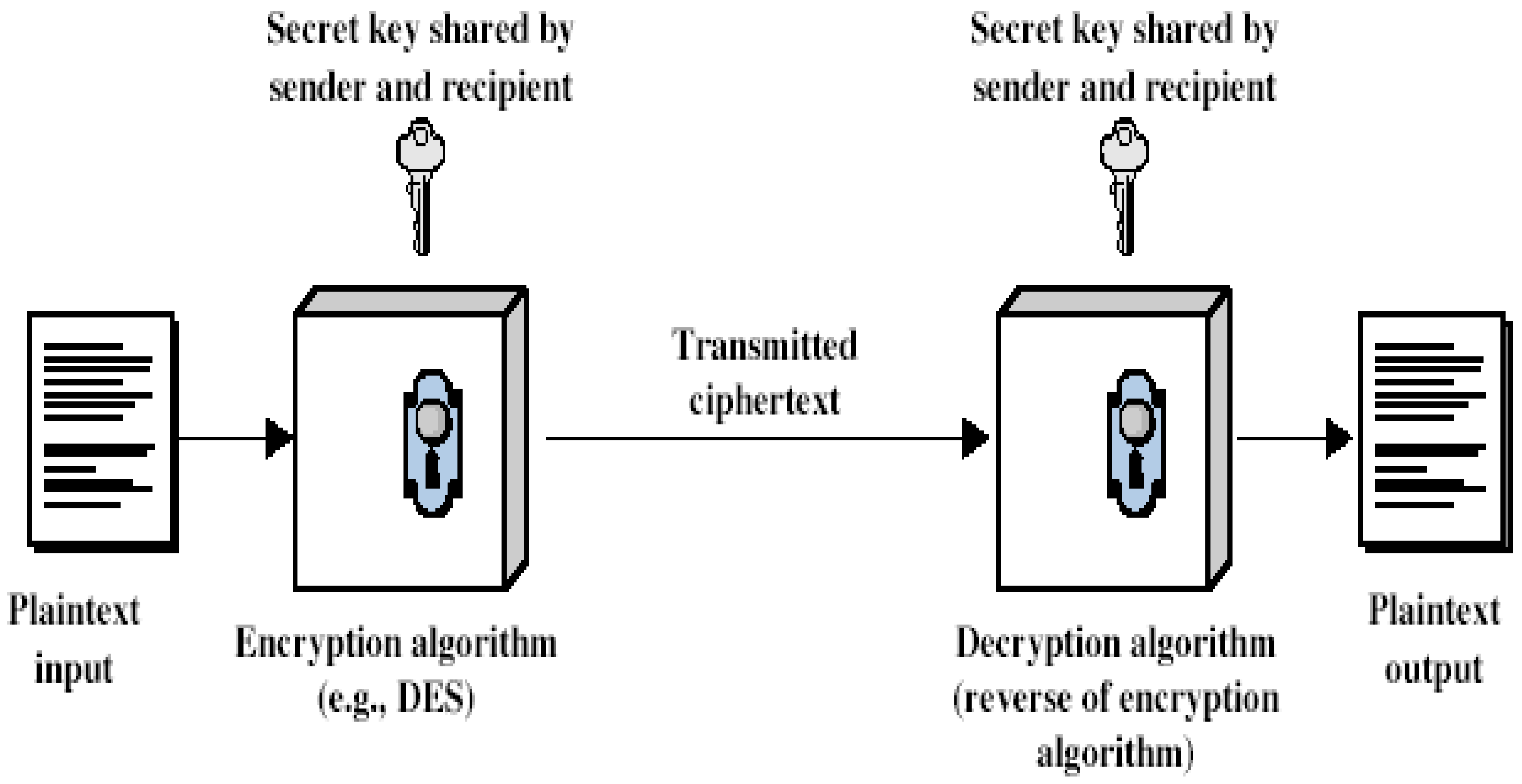
- The same key is used for encryption and decryption



- Popular algorithms: Advanced Encryption Standard (AES), TwoFish, IDEA, 3DES.

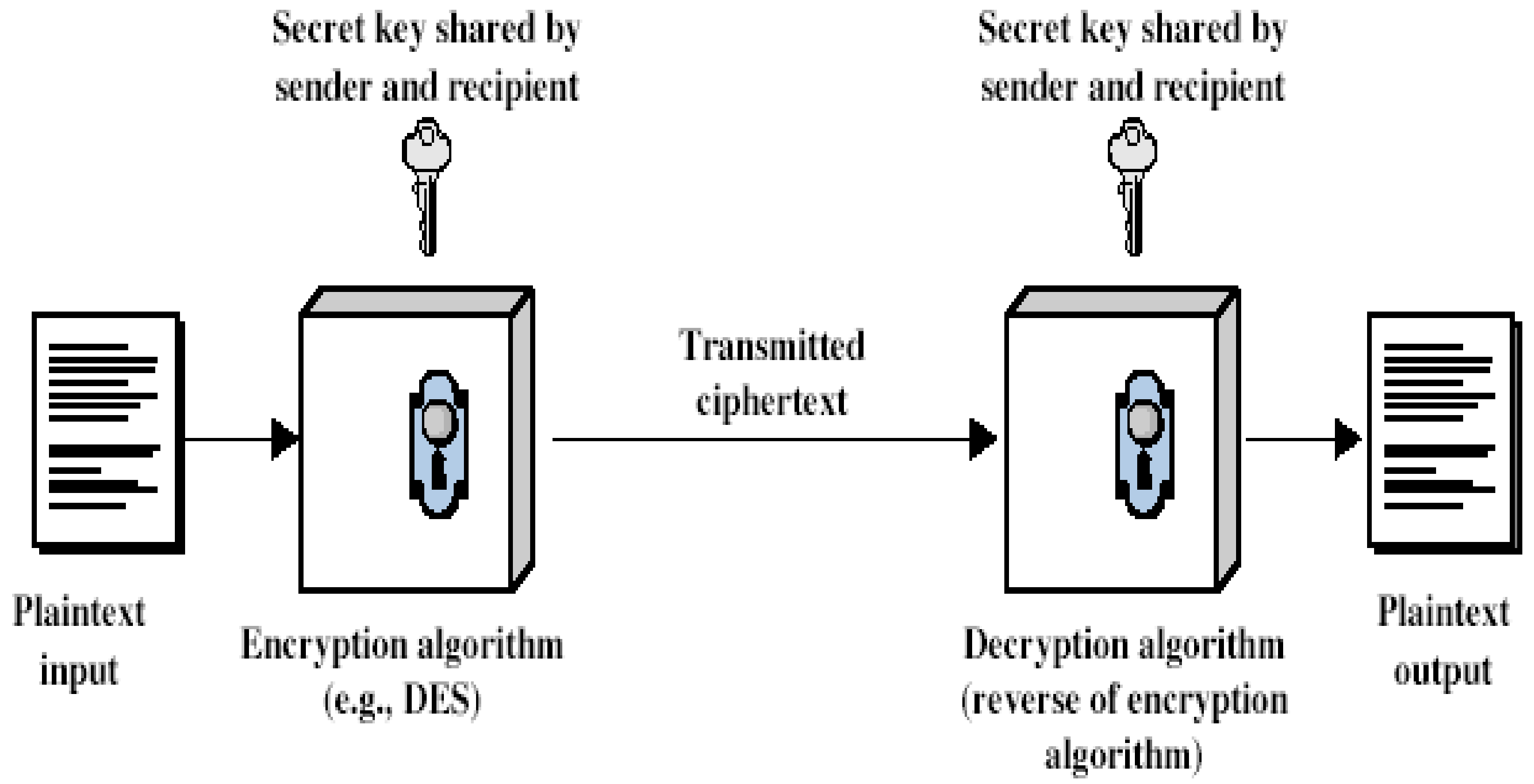
Symmetric Cipher Model

- Encryption algorithm: performs various substitutions and transformation on the plaintext.
- Secret key: the input of encryption algorithm. The key is independent of the plaintext and the alg..


Secret key shared by sender and recipient
Secret key shared by sender and recipient
Transmitted ciphertext
Plaintext input
Encryption algorithm (e.g., DES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Symmetric Cipher Model

- Decryption algorithm: the ciphertext and the secret key and produces the original plaintext


Secret key shared by sender and recipient
Secret key shared by sender and recipient
Transmitted ciphertext
Plaintext input
Encryption algorithm (e.g., DES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Requirements

- Requirements for secure use of symmetric encryption:
    - A strong encryption algorithm: the opponent should be unable to decrypt ciphertext or discover the key even if he/she has a number of ciphertexts and the plaintext that produced each ciphertext

Requirements

- Requirements for secure use of symmetric encryption:
    - Must assume that the opponent knows the algorithm. However, that knowledge must not suffice in deducing the secret key or the message contents (Kerckhoff’s Principle; next slide).

Requirements

- Requirements for secure use of symmetric encryption:
  - A third party that can securely distribute the key to the two communicating parties

Requirements

- Formalization of symmetric key security:

  X: message, K: encryption key, Y: ciphertext E: encryption function D: decryption function

  $$Y = E(K,X)$$

  $$X = D(K,Y)$$

  An opponent can observe Y, but do not have access to K or X.

Requirements: Kerckhoff's Principle

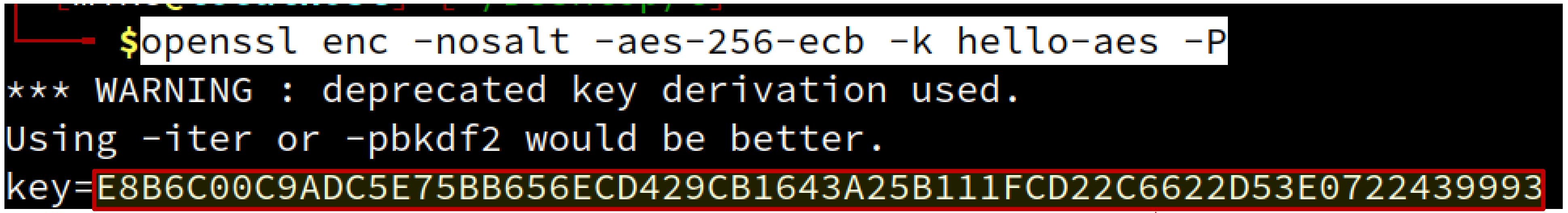
- ***A cryptographic system must remain secure even if everything about it except the secret key is known***
- Was first stated in the 19th century mathematician Auguste Kerckhoffs
- All modern cryptographic algorithms and protocols abide by it
- Is ***the opposite of "security by obscurity" (flawed) principle*** where security depends on the attacker not knowing the details of the cryptographic system function

Auguste Kerckhoff

Image source: https://www.jungfrauzeitung.ch/gosimg100E015a01ad806680b300001201041r.jpg

Symmetric Cryptography

- **Example:** encrypting a file using AES from Linux command line
- Generating a (symmetric) AES 256-bit cryptographic key:
  - **openssl enc -nosalt -aes-256-ecb -k hello-aes -P**
    - openssl: a popular *nix system cryptographic utility
    - -aes-256-ecb: the encryption algorithm
    - -k: the passphrase used for generating the key ("hello-aes" in this example)



```
$openssl enc -nosalt -aes-256-ecb -k hello-aes -P
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
key=E8B6C00C9ADC5E75BB656ECD429CB1643A25B111FCD22C6622D53E0722439993
```

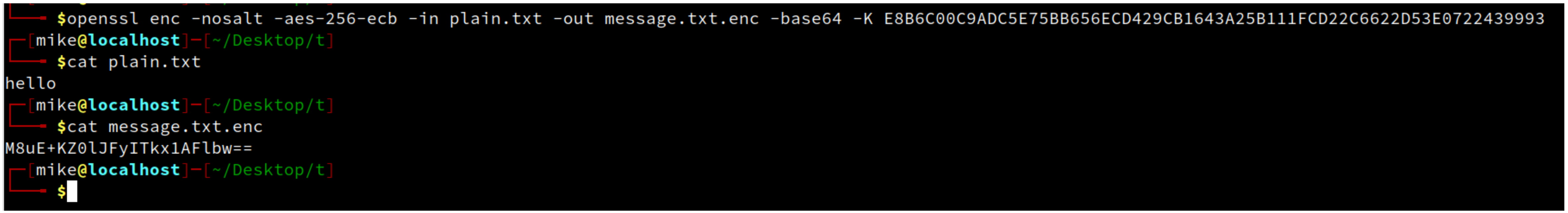
They generated key (in 256-bit hex; AES keys can be either 128, 192, or 256-bit keys. More later)

Symmetric Cryptography

- **Example:** encrypting a file using AES from Linux command line
- Encrypting using the key generated in the previous slide
  - **openssl enc -nosalt -aes-256-ecb -in plain.txt -out message.txt.enc -base64 -K E8B6C00C9ADC5E75BB656ECD429CB1643A25B111FCD22C6622D53E0722439993**
    - enc: tells the program to perform encryption
    - openssl: a popular *nix system cryptographic utility
    - -nosalt: do not use salting (more later; don't worry about this now)
    - -aes-256-ecb: encrypt using the AES symmetric key encryption algorithm with a 256-bit key
    - -in plain.txt: specifies the name of the file to encrypt (plain.txt in this example)
    - -out message.txt: the name of the file where to save the encrypted contents (message.txt.enc in this example)
    - -base64 –K: the encryption key to use (we used the key generated in the previous slide)

Symmetric Cryptography

- **Example:** encrypting a file using AES from Linux command line
- Encrypting using the key generated in the previous slide
  - **openssl enc -nosalt -aes-256-ecb -in plain.txt -out message.txt.enc -base64 -K E8B6C00C9ADC5E75BB656ECD429CB1643A25B111FCD22C6622D53E0722439993**
    - Running the above command and then printing out the contents of the original file and the encrypted file:



```
└──╼ $openssl enc -nosalt -aes-256-ecb -in plain.txt -out message.txt.enc -base64 -K E8B6C00C9ADC5E75BB656ECD429CB1643A25B111FCD22C6622D53E0722439993
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $cat plain.txt
hello
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $cat message.txt.enc
M8uE+KZ0lJFyITkx1AFlbw==
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $
```

Symmetric Cryptography

- **Example:** encrypting a file using AES from Linux command line
- Decrypting using the key generated in the previous slide
  - **openssl enc -nosalt -aes-256-ecb –d -in message.txt.enc -out message.txt.dec -base64 -K** E8B6C00C9ADC5E75BB656ECD429CB1643A25B111FCD22C6622D53E0722439993
    - In order to decrypt the file in the previous slide (message.txt.enc), we simply:
      - Add a –d argument after –aes-256-ecb as shown above
      - Provide the name of the encrypted file as input (message.txt.enc; the file we created in the previous slide)
      - Provide the name of the file to store the decrypted contents (message.txt.dec)

NOTE: Please notice that we used the SAME KEY for both encryption and decryption (example in the next slide)

Symmetric Cryptography

- Symmetric cryptography provides confidentiality

- Also provides a limited degree of authenticity – if you are able to decrypt the message successfully, you know it could only have been generated by you or the other party who knows the same symmetric key

Public Key Cryptography Overview

Public Key Cryptography

- **The Basics:**
  - Each communicator generates two mathematically linked keys
  - The keys are called public and private, respectively
  - The public key is made available to the public
  - The private key is kept private

Public Key Cryptography

- **The Basics:**
  - What is encrypted using the public key can decrypted using ***only*** the private key
  - What is encrypted using the private key can decrypted using ***only*** the public key
  - Encryption with public key provides confidentiality (explained on the next slide)
  - Encryption with private key provides digital signature (explained on the slides that follow)

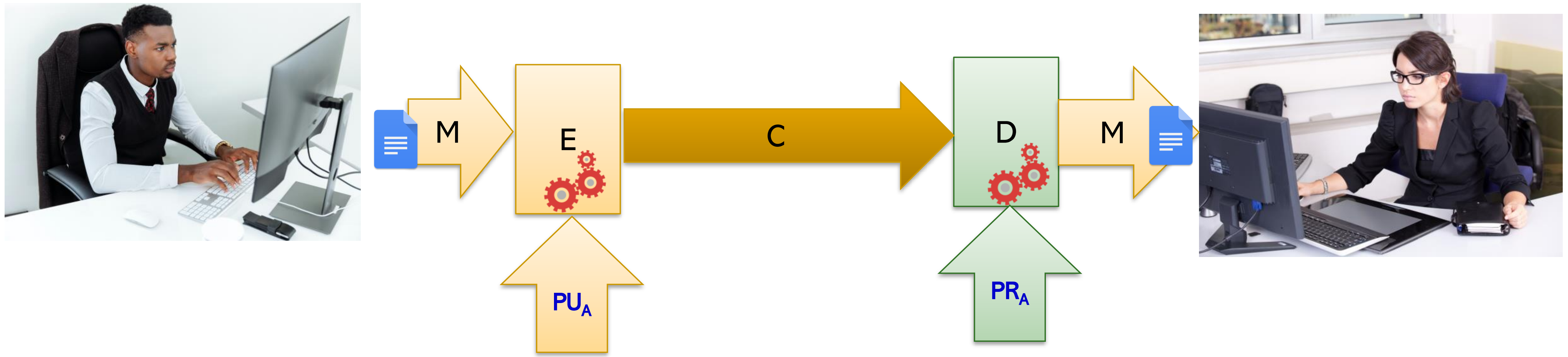
Public Key Cryptography

- Confidentiality Example: Bob wants to send a message confidentially to Alice
  - Let Bob and Alice be the two communicators
  - Let PU_B represent Bob's public key
  - Let PR_B represent Bob's private key
  - Let PU_A represent Alice's public key
  - Let PR_A represent Alice's private key
  - Let $C = E(M,K_e)$ represent the encryption function where
    - M is the plaintext
    - K_e is the encryption key
    - C is the ciphertext
  - Let $M = D(C,K_d)$ represent the encryption function where C is the ciphertext and K_d is the decryption key

Public Key Cryptography

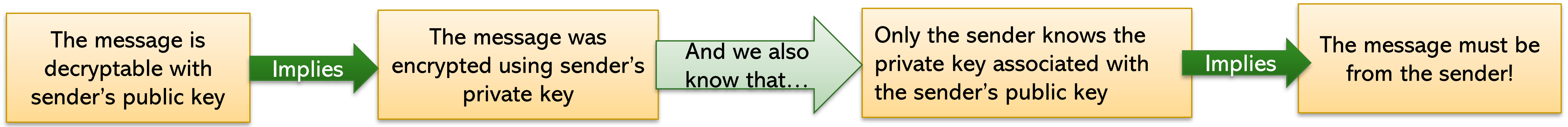
- Confidentiality Example: Bob wants to send a message (M) confidentially to Alice
  - Bob obtains Alice's public key (PU_A) and uses it to encrypt the message
  - Alice decrypts the message Alice's private key (PR_A)

Bob (in possession of PU_B, PR_B, and PU_A) **Alice (in possession of PU_A, PR_A, and PU_B)**



Public Key Cryptography

- Can also encrypt the message using the sender's private key to provide what is called a ***digital signature:***
  - Assures the receiver that the message is indeed from the sender and was untampered with:
  - **Logic:**


The message is decryptable with sender's public key
Implies
The message was encrypted using sender's private key
And we also know that…
Only the sender knows the private key associated with the sender's public key
Implies
The message must be from the sender!

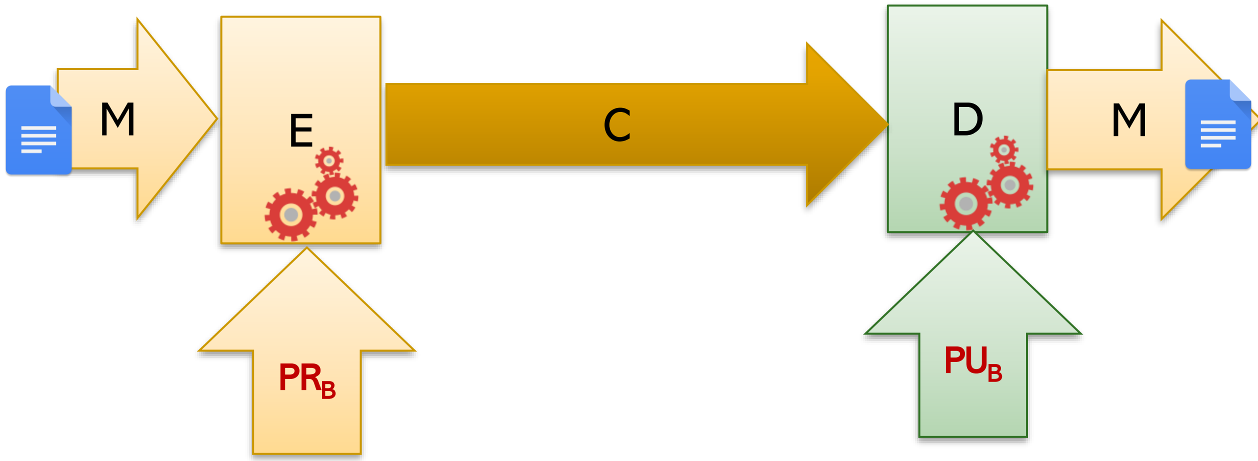
NOTE: This does ***NOT*** provide confidentiality! Anybody (and we can assume everybody) who has the sender's public key will be able to decrypt the message.

Public Key Cryptography

- **Digital Signature Example:** Bob wants to send a digitally signed message (M) to Alice
  - Bob encrypts the message using his private key (PR_B)
  - Alice obtains Bob's public key (PU_B) and uses it to decrypt the message
  - If the decryption succeeds, the message must be from Bob

Bob (in possession of PU_B, PR_B, and PU_A)

Alice (in possession of PU_A, PR_A, and PU_B)



Public Key Cryptography

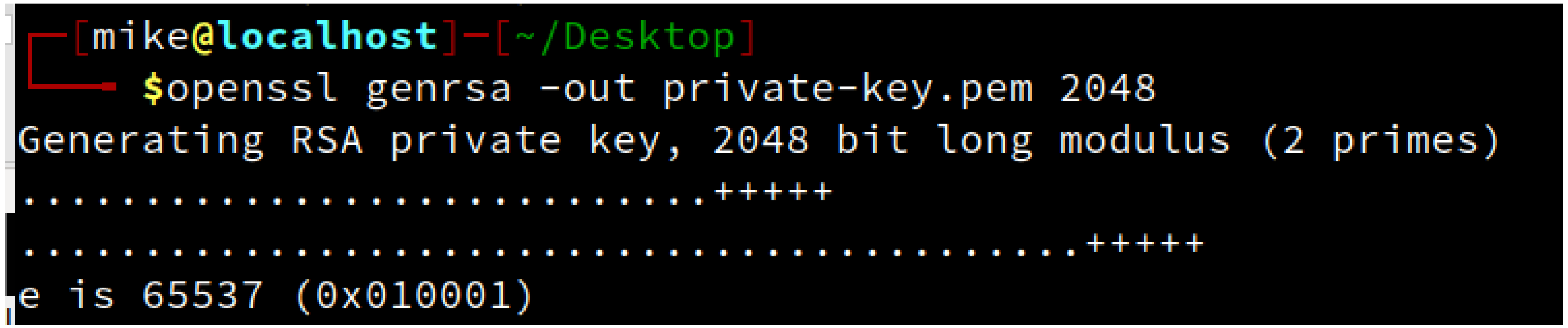
- Example: Bob can provide **both** confidentiality and digital signature if:
  - Bob encrypts the message using his private key (**PR_B**): $C_1 = E(M, PR_B)$
    - Provides digital signature
  - Encrypt the resulting message with Alice's public key: $C = E(C_1, PU_A)$
    - Provides confidentiality

Public Key Cryptography

- Example: Bob can provide **both** confidentiality and digital signature if:
  - Alice then first decrypts the message using Alice's private key: $C_1 = D(C, PR_A)$
  - Alice then obtains Bob's public key (PU_B) and uses it to decrypt the resulting message again: $M = D(C_1, PU_B)$
    - If the decryption succeed, the digital signature is considered valid

Public Key Cryptography

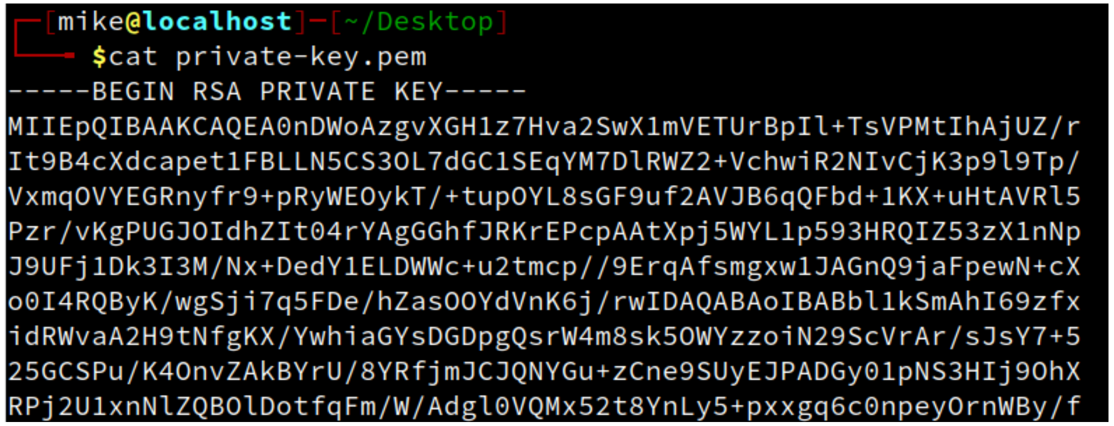
- **Example:** generating a public/private key pair:
  - First, we need to generate the private key
  - openssl genrsa –out private-key.pem 2048
    - Generates the private key
    - genrsa: generate a private key for the RSA algorithm – a popular public key encryption algorithm
    - Specifies to save the generated private key in the file called private-key.pem
    - 2048: the size of the key to generate (in bits)
    - Running the command:



```
┌─[mike@localhost]─[~/Desktop]
└──╼ $openssl genrsa -out private-key.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
............................+++++
...............................................+++++
e is 65537 (0x010001)
```

Public Key Cryptography

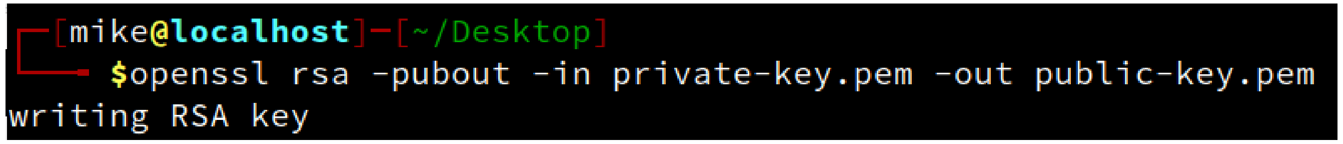
- **Example:** generating a public/private key pair:
  - First, we need to generate the private key
  - openssl genrsa –out private-key.pem 2048
    - Generates the private key
    - genrsa: generate a private key for the RSA algorithm – a popular public key encryption algorithm
    - Specifies to save the generated private key in the file called private-key.pem
    - 2048: the size of the key to generate (in bits)
    - Partial contents of private-key.pem:



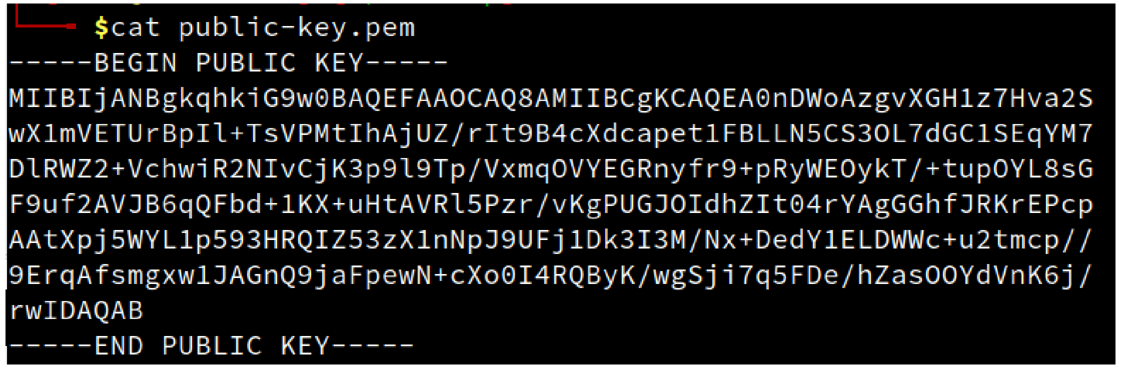
```
┌─[mike@localhost]─[~/Desktop]
└──╼ $cat private-key.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEA0nDWoAzgvXGH1z7Hva2SwX1mVETUrBpIl+TsVPMtIhAjUZ/r
It9B4cXdcapet1FBLLN5CS3OL7dGC1SEqYM7DlRWZ2+VchwiR2NIvCjK3p9l9Tp/
VxmqOVYEGRnyfr9+pRyWEOykT/+tupOYL8sGF9uf2AVJB6qQFbd+1KX+uHtAVRl5
Pzr/vKgPUGJOIdhZIt04rYAgGGhfJRKrEPcpAAtXpj5WYL1p593HRQIZ53zX1nNp
J9UFj1Dk3I3M/Nx+DedY1ELDWWc+u2tmcp//9ErqAfsmgxw1JAGnQ9jaFpewN+cX
o0I4RQByK/wgSji7q5FDe/hZasOOYdVnK6j/rwIDAQABAoIBABbl1kSmAhI69zfx
idRWvaA2H9tNfgKX/YwhiaGYsDGDpgQsrW4m8sk5OWYzzoiN29ScVrAr/sJsY7+5
25GCSPu/K4OnvZAkBYrU/8YRfjmJCJQNYGu+zCne9SUyEJPADGy01pNS3HIj9OhX
RPj2U1xnNlZQBOlDotfqFm/W/Adgl0VQMx52t8YnLy5+pxxgq6c0npeyOrnWBy/f
```

Public Key Cryptography

- **Example:** generating a public/private key pair:
  - Next, we generate the public key linked to the private key we have previously generated (i.e., the key in private-key.pem)
  - openssl rsa -pubout -in private-key.pem -out public-key.pem
    - Generates a public and private keys
    - rsa: specifies to use the RSA algorithm – a popular public key encryption algorithm
    - Specifies to save the generated public key in the file called public-key.pem
    - Specifies to save the generated private key in the file called private-key.pem
    - Running the command (left) and partial contents of public-key.pem (right):



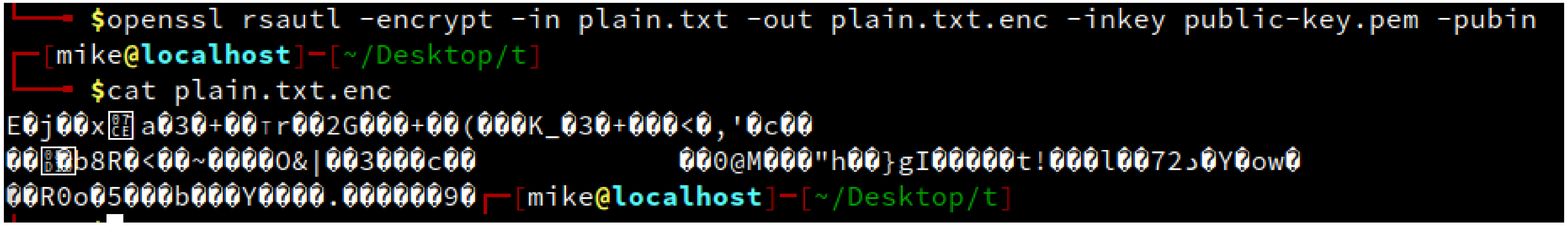
```
┌─[mike@localhost]─[~/Desktop]
└──╼ $openssl rsa -pubout -in private-key.pem -out public-key.pem
writing RSA key
```



```
└──╼ $cat public-key.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0nDWoAzgvXGH1z7Hva2S
wX1mVETUrBpIl+TsVPMtIhAjUZ/rIt9B4cXdcapet1FBLLN5CS3OL7dGC1SEqYM7
DlRWZ2+VchwiR2NIvCjK3p9l9Tp/VxmqOVYEGRnyfr9+pRyWEOykT/+tupOYL8sG
F9uf2AVJB6qQFbd+1KX+uHtAVRl5Pzr/vKgPUGJOIdhZIt04rYAgGGhfJRKrEPcp
AAtXpj5WYL1p593HRQIZ53zX1nNpJ9UFj1Dk3I3M/Nx+DedY1ELDWWc+u2tmcp//
9ErqAfsmgxw1JAGnQ9jaFpewN+cXo0I4RQByK/wgSji7q5FDe/hZasOOYdVnK6j/
rwIDAQAB
-----END PUBLIC KEY-----
```

Public Key Cryptography

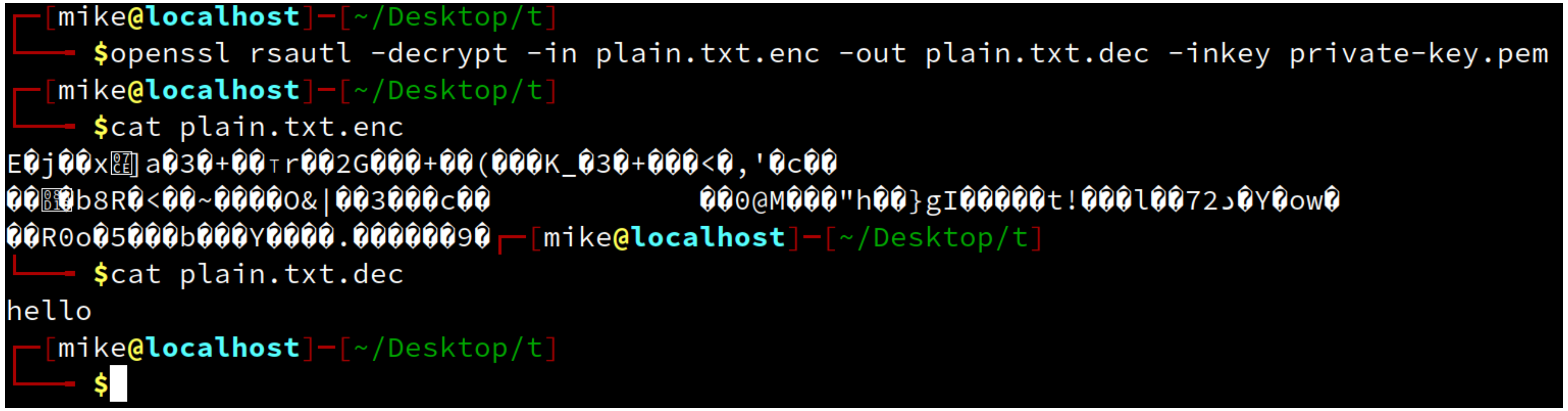
- Example: Confidentiality: Encrypting a file using the public key
  - openssl rsautl -encrypt -in plain.txt -out plain.txt.enc -inkey public-key.pem –pubin
    - Encrypt file plain.txt using RSA and public key stored in file public-key.pem (the one we generated in the previous example)
    - Store the encrypted file contents in plain.txt.enc
    - Running the command and printing the contents of the original file and the encrypted file:



```
└──╼ $openssl rsautl -encrypt -in plain.txt -out plain.txt.enc -inkey public-key.pem -pubin
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $cat plain.txt.enc
E�j��x a�3�+��┬r��2G���+��(���K_�3�+���<�,'�c��
����b8R�<��~����O&|��3���c��              ��0@M���"h��}gI�����t!���l��72¸�Y�ow�
��R0o�5���b���Y����.�������9�┌─[mike@localhost]─[~/Desktop/t]
```

Public Key Cryptography

- Example: Confidentiality: Decrypting a file using the private key:
  - openssl rsautl -decrypt -in plain.txt.enc -out plain.txt.dec -inkey private-key.pem
    - -decrypt: decrypt the file
    - plain.txt.enc: the file to decrypt using RSA and private key stored in file private-key.pem (the one we generated in the previous example)
    - Store the decrypted file contents in plain.txt.enc
    - Running the command and printing the contents of the original file and the decrypted file:



```
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $openssl rsautl -decrypt -in plain.txt.enc -out plain.txt.dec -inkey private-key.pem
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $cat plain.txt.enc
E�j��x a�3�+��┬r��2G���+��(���K_�3�+���<�,'�c��
��b8R�<��~����0&|��3���c��            ��0@M���"h��}gI�����t!���l��72.�Y�ow�
��R0o�5���b���Y����.�������9�┌─[mike@localhost]─[~/Desktop/t]
└──╼ $cat plain.txt.dec
hello
┌─[mike@localhost]─[~/Desktop/t]
└──╼ $
```

Public-key cryptography: Misconceptions

- Misconception 1: Public-key encryption is more secure from cryptanalysis than symmetric encryption
  - The security depends on the length of the key and the computational work involved in breaking a cipher.

Public-key cryptography: Misconceptions

- Misconception 2: Public-key encryption is a general-purpose technique that has made symmetric encryption obsolete.
  - Computation overhead of public-key encryption is significantly higher

Overview of Cryptoanalysis

Cryptanalysis

- Objective of attacking an encryption system: recover key rather than simply to recover the plaintext of a single ciphertext.
- General approaches:
  - Cryptanalytic attack:
    - Rely on the nature of the algorithm + some knowledge of the plaintext (e.g. English or French) or some sample plaintext-ciphertext pairs.
  - Brute-force attack
    - Try every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.

Cryptanalysis

- Objective of attacking an encryption system: recover key rather than simply to recover the plaintext of a single ciphertext.
- General approaches:
  - Cryptanalytic attack:
    - Rely on the nature of the algorithm + some knowledge of the plaintext (e.g., English or French) or some sample plaintext-ciphertext pairs.
  - Brute-force attack
    - Try every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.

Brute-Force Attacks in Practice

- Simply try every key until an intelligible translation of the ciphertext into plaintext is obtained
- On average, half of all possible keys must be tried to achieve success - proportional to key size
- DES: 56-bit, triple DES: 168-bit, AES: > 128 bits
- Time required for various key spaces:

Key Size (bits)	Number of Alternative Keys	Time required at 1 decryption/μs	Time required at 10^6 decryptions/μs
32	$2^{32} = 4.3 \times 10^9$	2^{31} μs= 35.8 minutes	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	2^{55} μs= 1142 years	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	2^{127} μs= 5.4×10^{24} years	5.4×10^{18} years
168	$2^{168} = 3.7 \times 10^{50}$	2^{167} μs= 5.9×10^{36} years	5.9×10^{30} years

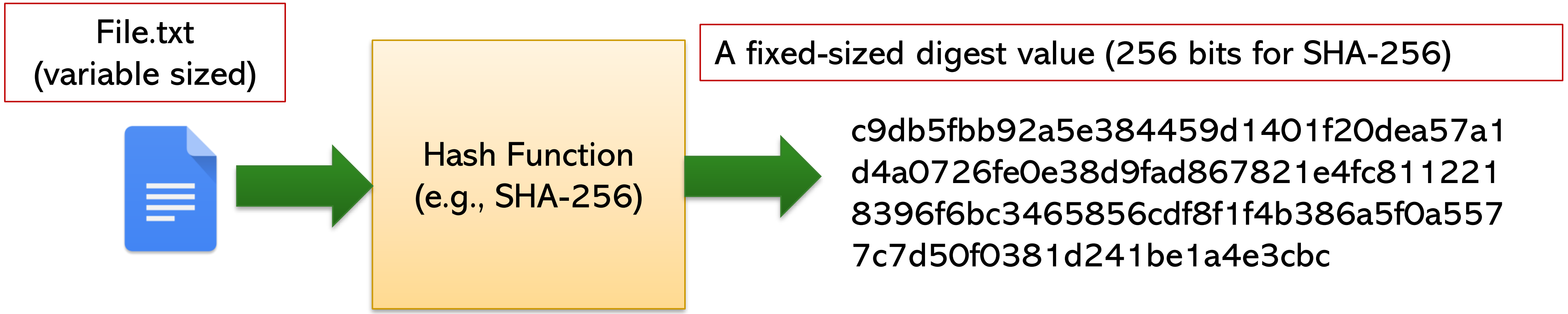
Cryptanalysis

- Types of cryptoanalytic attacks:
  - Ciphertext-only:
    - Attacker only has access to the ciphertext.
    - Attempts to use statistical techniques to discover the key and/or plaintext.
  - Known-plain text attack:
    - Attacker has access to the ciphertext and knows some properties of the plaintext
    - Example: the plaintext is known to contain a certain sentence.
  - Chosen-plaintext attack:
    - Attacker can encrypt any plaintext using the target encryption scheme
    - Uses analysis in order to figure out the function of the cryptographic system (and hopefully the secret key!).

Overview of Hashing

Hash Functions

- Hash function: a one-way function that accepts a variable-size message M as input and produces a fixed-size output, referred to as a hash code H(M)
  - Hash code: message digest or hash value


File.txt
(variable sized)
Hash Function
(e.g., SHA-256)
A fixed-sized digest value (256 bits for SHA-256)
c9db5fbb92a5e384459d1401f20dea57a1
d4a0726fe0e38d9fad867821e4fc811221
8396f6bc3465856cdf8f1f4b386a5f0a557
7c7d50f0381d241be1a4e3cbc

- Hash code is used to detect changes to message
- A change to any bit(s) in the message results in a *significant* change in the hash value
- Can use in various ways with message
- Most often to create a digital signature, verify file integrity, etc

Requirements for Hash Functions

- The purpose of a hash is to produce a fingerprint of a file, message, etc
- A hash function must have the following properties:
  - Can be applied to any sized message M
  - Produces fixed-length output hc
  - Is easy to compute $hc=H(M)$ for any message M
  - Given hc is infeasible to find x s.t. $H(x)=hc$
  - Given x is infeasible to find y s.t. $H(y)=H(x)$
  - It's infeasible to find any x,y s.t. $H(y)=H(x)$

Requirements for Hash Functions

- There are many hashing algorithms: e.g.,
  - MD5 algorithm computes a 128-bit hash (not considered very secure anymore)
  - SHA-512 algorithms computes a 512-bit hash

Requirements for Hash Functions: Birthday Attack

- Birthday Attack: a bruteforce attack technique that exploits the probability theory behind the birthday problem, to reduce the complexity of finding the collision between hash functions by comparing the hashes of $2^{n/2}$ different messages.

Requirements for Hash Functions: Birthday Attack

- Birthday Problem: What is the probability that of n randomly chosen people, that at least two will have the same birthday?
  - Birthday Paradox: states that if 23 people chosen, the probability will exceed 50%.

Explanation: We need to compare the birthday of each pair of people. There are **(22 * 23)** / 2 = 253 possible pairs of people. This exceeds the number of days in the year (**365**)!

Requirements for Hash Functions

- Birthday Attack: The number of hashes is limited by the length of the hash. Therefore, attackers can exploit this property to find to generate a colliding hashes…how (next slide)?

- Let M be a message of n-bit length.
- Let S be set of hashes (initially empty)
- The algorithm works as follows:

Step 1: S = empty

Step 2: Randomly choose $2^{n/2}$ messages derived from M

Step 3: For i = 1, 2, …, $2^{n/2}$ compute hash **h** = H(m_i)

Step 4: Does **h** match any hash in **S**? If so, $\Longrightarrow$ collision detected!

Step 5: If h does not match, then add **h** to **S**, and go back to step 2

Requirements for Hash Functions

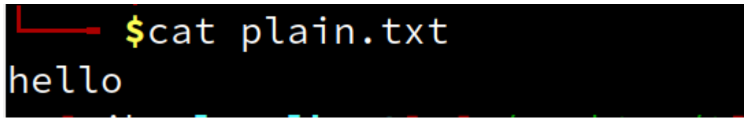
- Birthday Attack: The number of hashes is limited by the length of the hash. Therefore, attackers can exploit this property to find to generate a colliding hashes…how (next slide)?

- The probability of finding a collision is

$$P(H)=\left(\frac{1}{H}\right)^{N}\cdot\frac{H!}{(H-N)!}$$

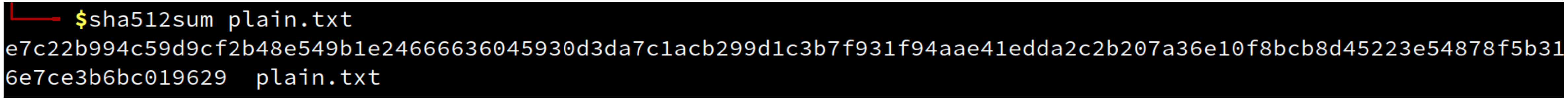
Requirements for Hash Functions

- Example: compute a hash of the file in Linux using SHA-512 hash:
  - Command: sha512sum <filename>
  - Let's first print contents of the original file plain.txt:



```
└── $cat plain.txt
hello
```

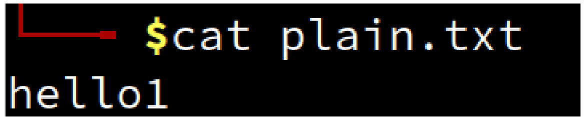
  - Next, let's compute the SHA-512 hash (the digest value is 512-bit long)



```
└── $sha512sum plain.txt
e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36e10f8bcb8d45223e54878f5b31
6e7ce3b6bc019629  plain.txt
```

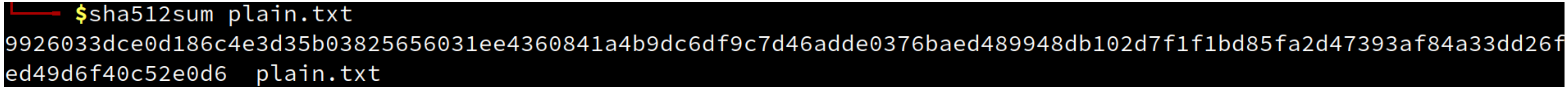
Requirements for Hash Functions

- **Example:** compute a hash of the file in Linux using SHA-512 hash:
  - Now, let’s add “1” to the end of the file, and print its contents:



```
└──╼ $cat plain.txt
hello1
```

  - Compute the SHA-512 hash of the file:

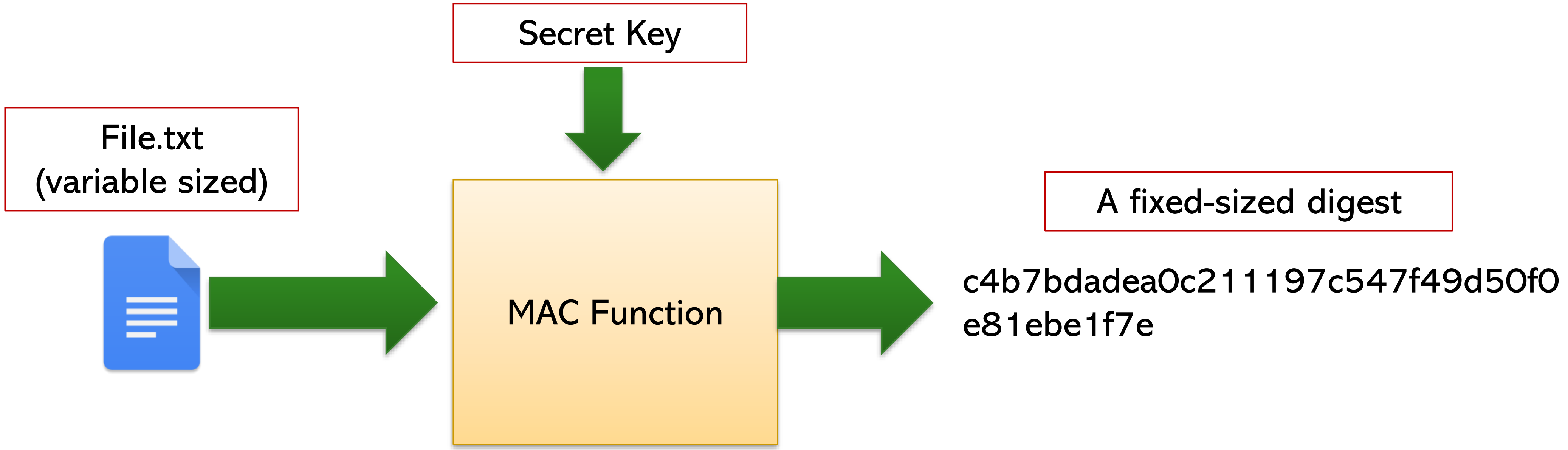


```
└──╼ $sha512sum plain.txt
9926033dce0d186c4e3d35b03825656031ee4360841a4b9dc6df9c7d46adde0376baed489948db102d7f1f1bd85fa2d47393af84a33dd26f
ed49d6f40c52e0d6  plain.txt
```

Message Authentication Codes (MACs)

Message Authentication Code (MAC)

- Similar to a hash function, the MAC function is a one-way function that accepts variable input and produces a fixed-sized output, ***but also requires a secret key***



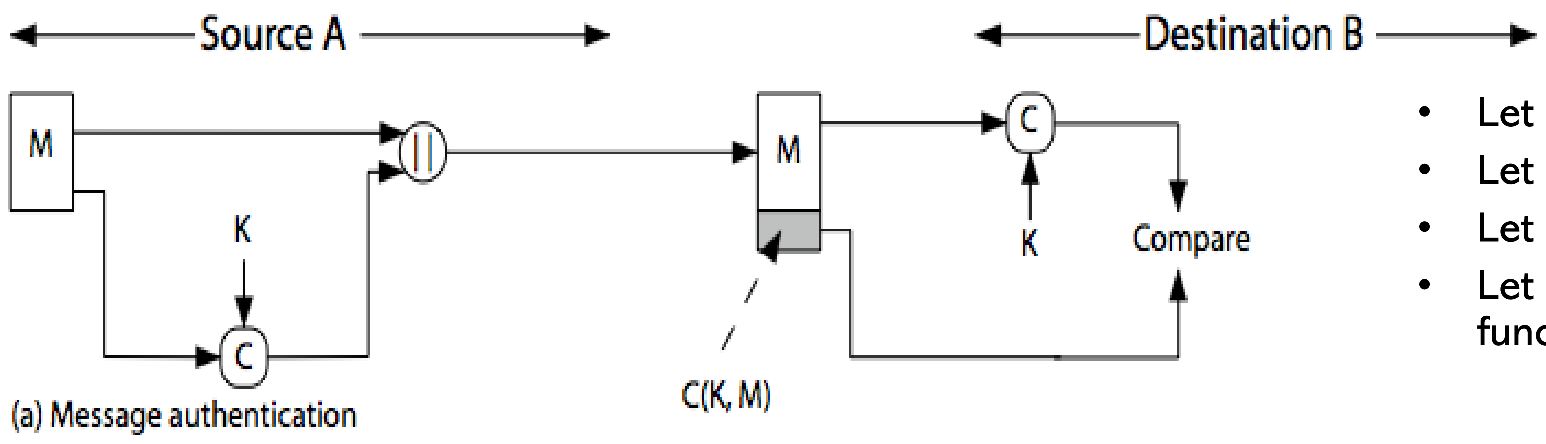
- Can be used for checking integrity of e.g., messages and files, and provide a degree of authentication similar to symmetric cryptography:

Message Authentication Code (MAC)

- Example: how MACs are used to verify the message integrity:
  - Let M be the message
  - Let C be the MAC function
  - Let K be the secret MAC key
  - Let || represent the appending function (literally appending data)

Message Authentication Code (MAC)

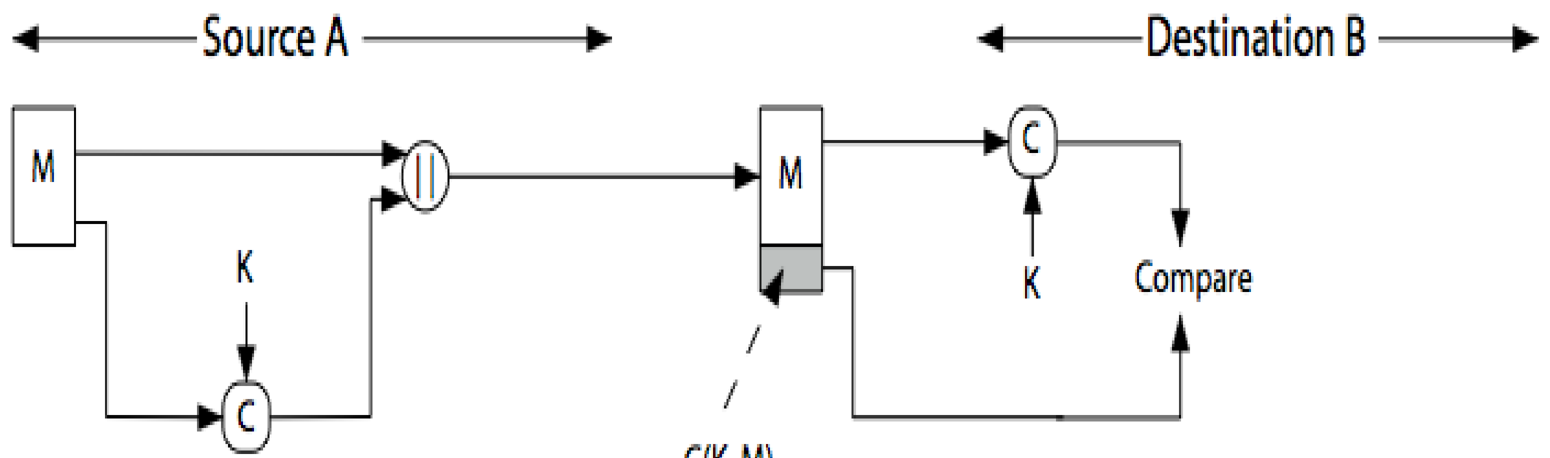
- Example: how MACs are used to verify the message integrity:



- Let M be the message
- Let C be the MAC function
- Let K be the secret MAC key
- Let || represent the appending function (literally appending data)

- Use a secret key to generate a small fixed-size block of data MAC = C(K,M)
  - M: input mesg. C: MAC function. K: shared secret key
- The message + MAC are transmitted.
- Receiver performs same computation on message and checks it matches the MAC

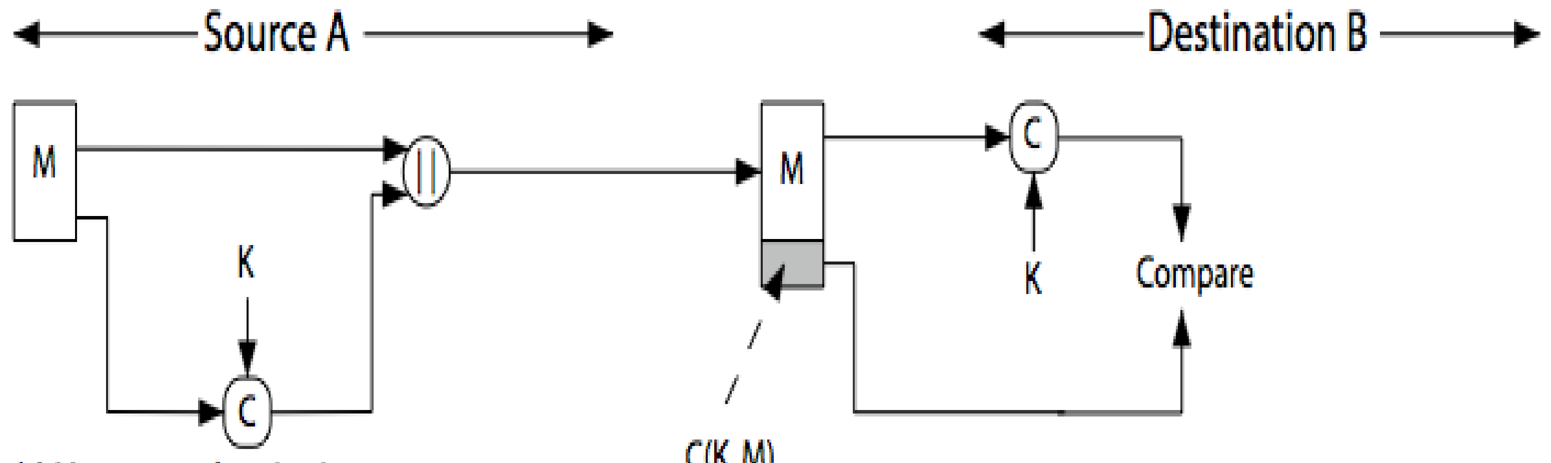
Message Authentication Code (MAC)



(a) Message authentication

- Does this assure that message is unaltered and comes from sender?

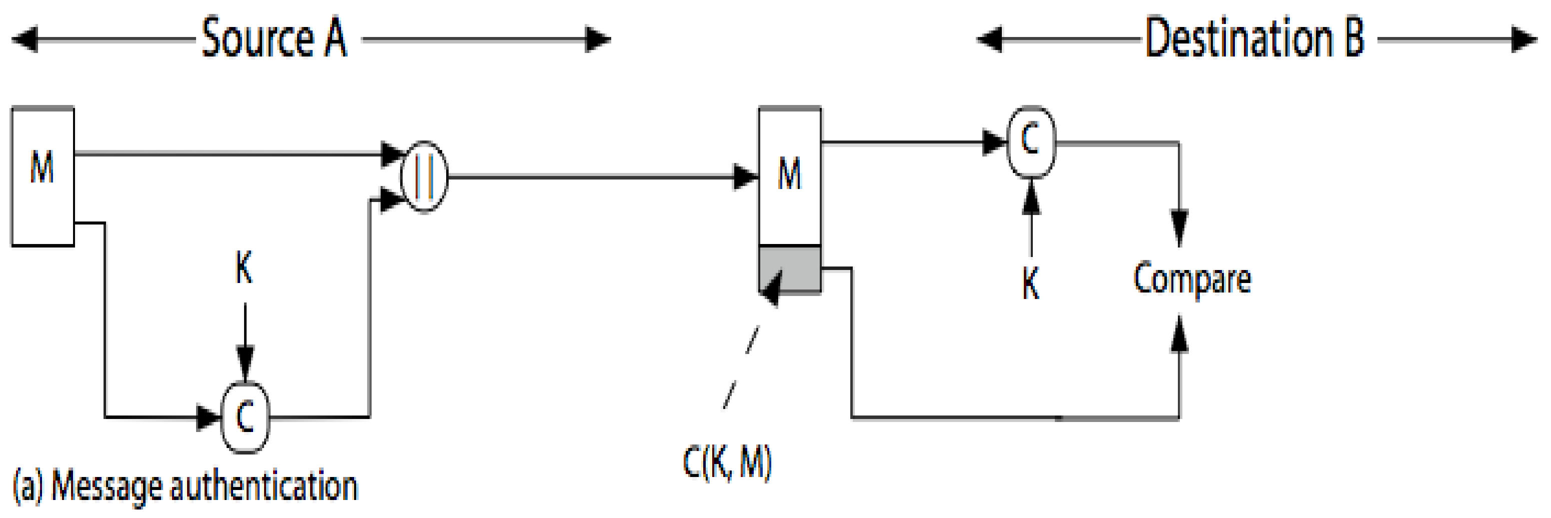
Message Authentication Code (MAC)



(a) Message authentication

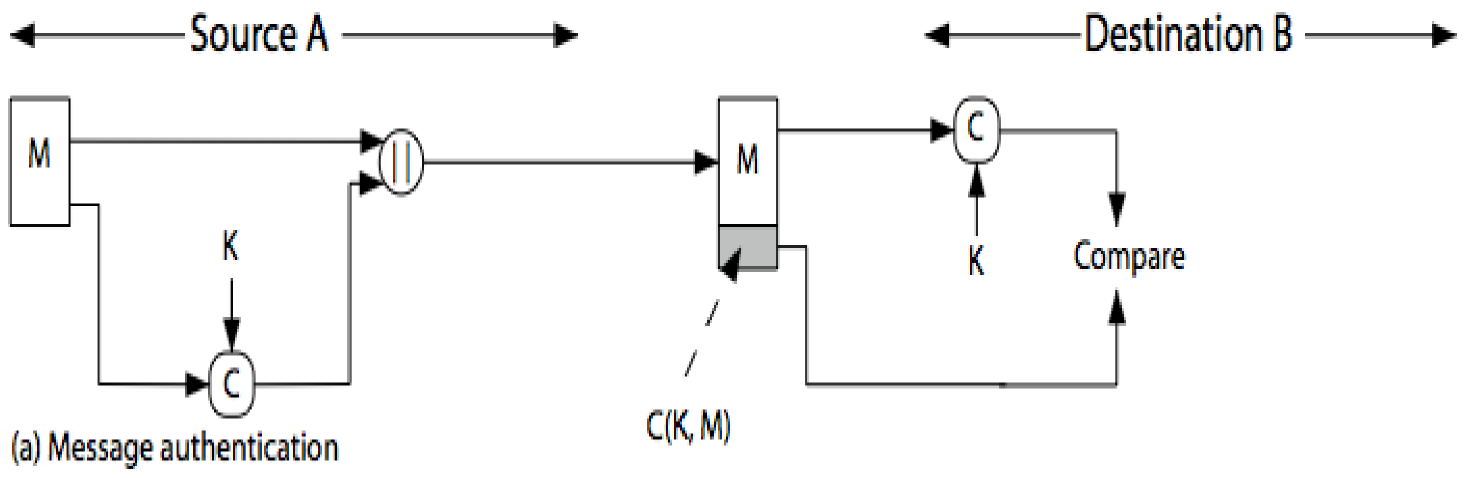
- Does this assure that message is unaltered and comes from sender?
  - Yes

Message Authentication Code (MAC)



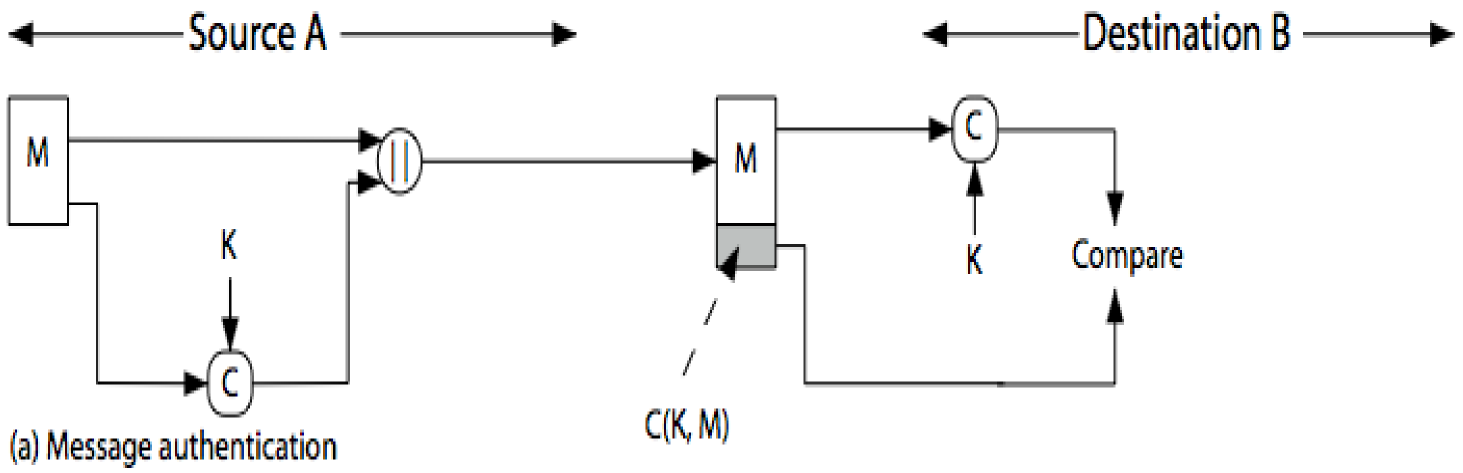
- Does this assure that message is unaltered and comes from sender?
  - Yes
  - If an attacker alters the message, then the receiver's calculation of the MAC will differ from the received MAC.
  - No one else knows the secret key, no one else could prepare a message with a proper MAC.

Message Authentication Codes



- Provides authentication but not confidentiality
  - because the message is transmitted in the clear.

Message Authentication Codes



- A MAC function is similar to encryption
  - Difference: MAC alg. needs not be reversible, as it must for decryption. In general, MAC function is a many-to-one function - many different messages may generate the same MAC value.

Message Authentication Codes

- Symmetric encryption will provide authentication. Why use a MAC?

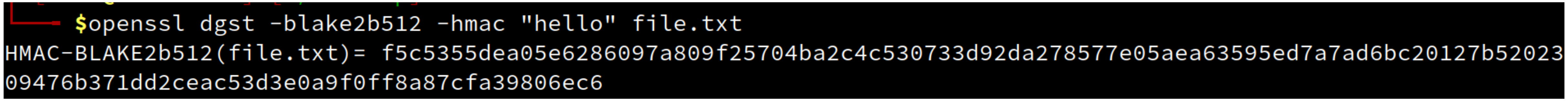
Message Authentication Codes

- Symmetric encryption will provide authentication. Why use a MAC?
  - Sometimes only authentication is needed.
  - One side has a heavy load and cannot afford the time to decrypt all incoming messages – messages are chosen at random for checking
- MAC does not provide a digital signature because both sender and receiver share the same key.

Message Authentication Codes

- Example: Computing MACs from the command line:

- Computing a MAC of a file:
  - openssl –dgst <MAC Algorithm> -hmac <Key> <File Name>
  - Example: openssl dgst -blake2b512 -hmac "hello" file.txt
    - Compute the MAC of file file.txt using the blake2b512 MAC algorithm and key “hello”



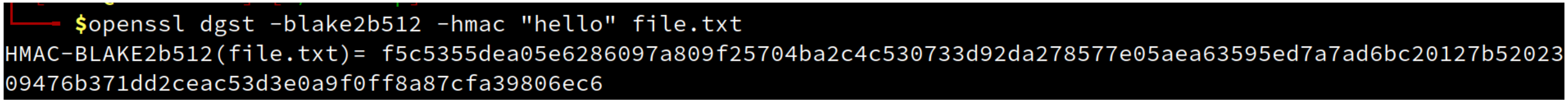
```
└──── $openssl dgst -blake2b512 -hmac "hello" file.txt
HMAC-BLAKE2b512(file.txt)= f5c5355dea05e6286097a809f25704ba2c4c530733d92da278577e05aea63595ed7a7ad6bc20127b52023
09476b371dd2ceac53d3e0a9f0ff8a87cfa39806ec6
```

Message Authentication Codes

- Example: Computing MACs from the command line:

- Computing a MAC of a file:
  - NOTE: Even minor changes to the key or the message/file will result in major changes to the MAC
  - For example, let's change the key from the previous slide to "hello1":

BEFORE (key "hello")



```
└──╼ $openssl dgst -blake2b512 -hmac "hello" file.txt
HMAC-BLAKE2b512(file.txt)= f5c5355dea05e6286097a809f25704ba2c4c530733d92da278577e05aea63595ed7a7ad6bc20127b52023
09476b371dd2ceac53d3e0a9f0ff8a87cfa39806ec6
```

AFTER (key "hello1")

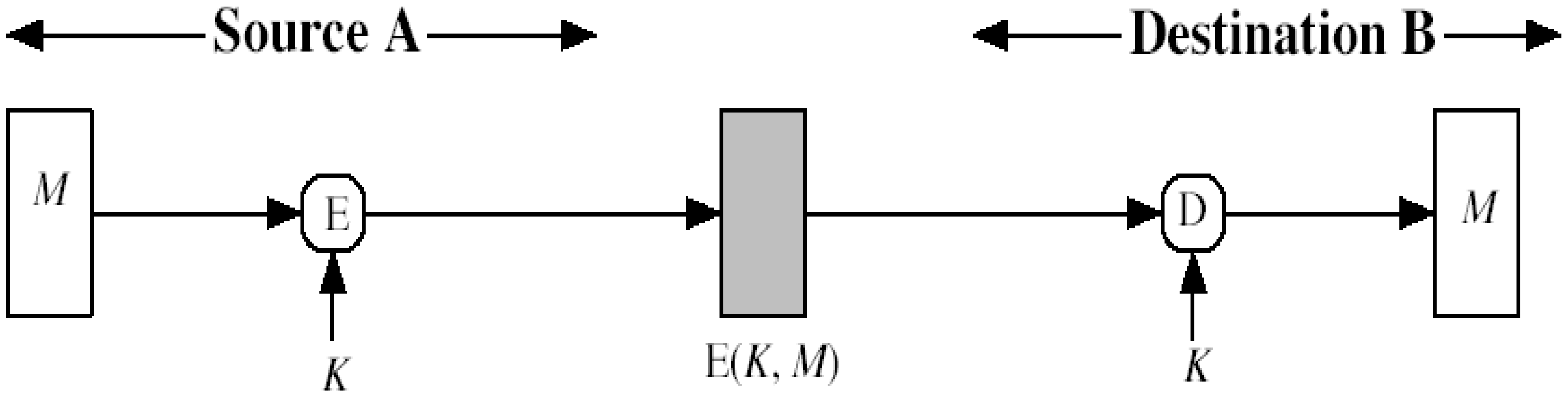
Basic Cryptographic Techniques for Authentication and Confidentiality

Basic Cryptographic Techniques for Authentication and Confidentiality

- We will now use the building blocks of cryptography (symmetric, public, hashing, MACs, etc) to see how they can be used for authentication, confidentiality, integrity, etc.

- Let's start with symmetric cryptography…

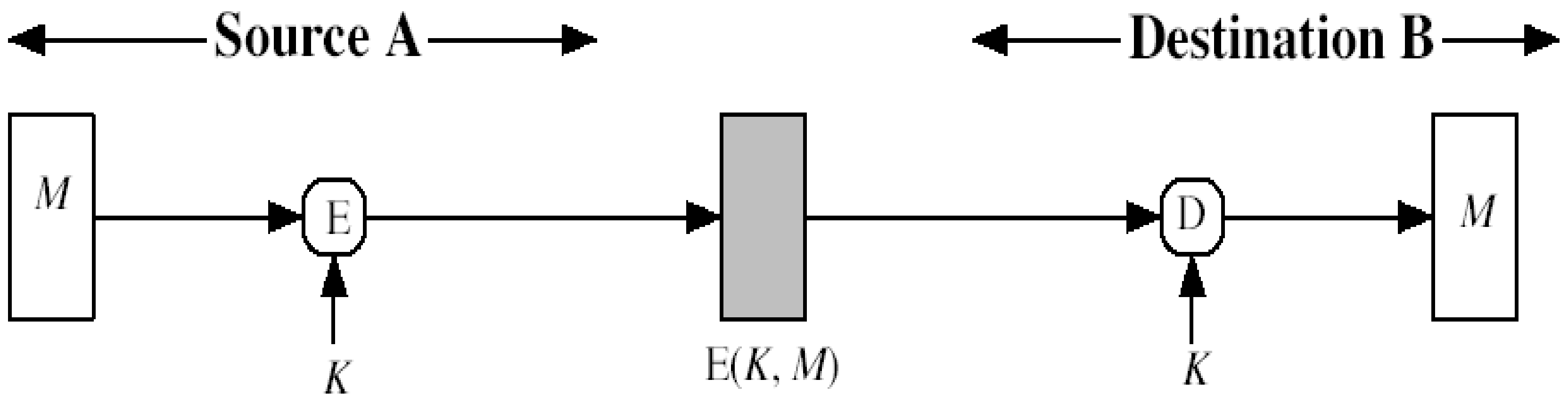
Message Encryption: Symmetric Encryption



(a) Symmetric encryption: confidentiality and authentication

- Provides confidentiality!

Message Encryption: Symmetric Encryption

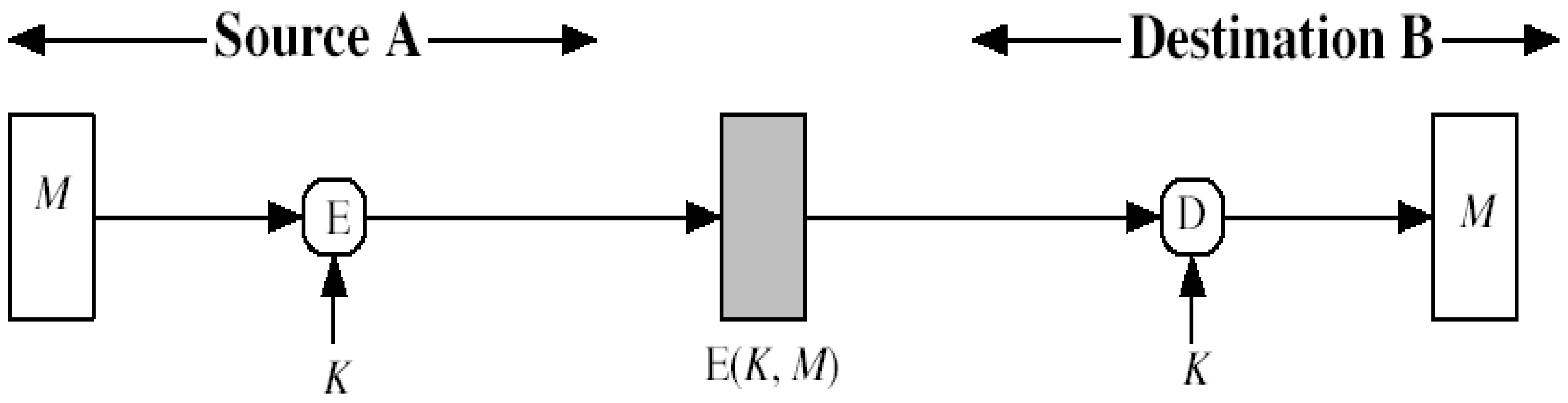


(a) Symmetric encryption: confidentiality and authentication

Also…

- B knows A must have created it
    - Since only sender and receiver know key used
- Knows content cannot be altered

Message Encryption: Symmetric Encryption



(a) Symmetric encryption: confidentiality and authentication

- B knows A must have created it
  - Since only sender and receiver know key used
- Knows content cannot be altered

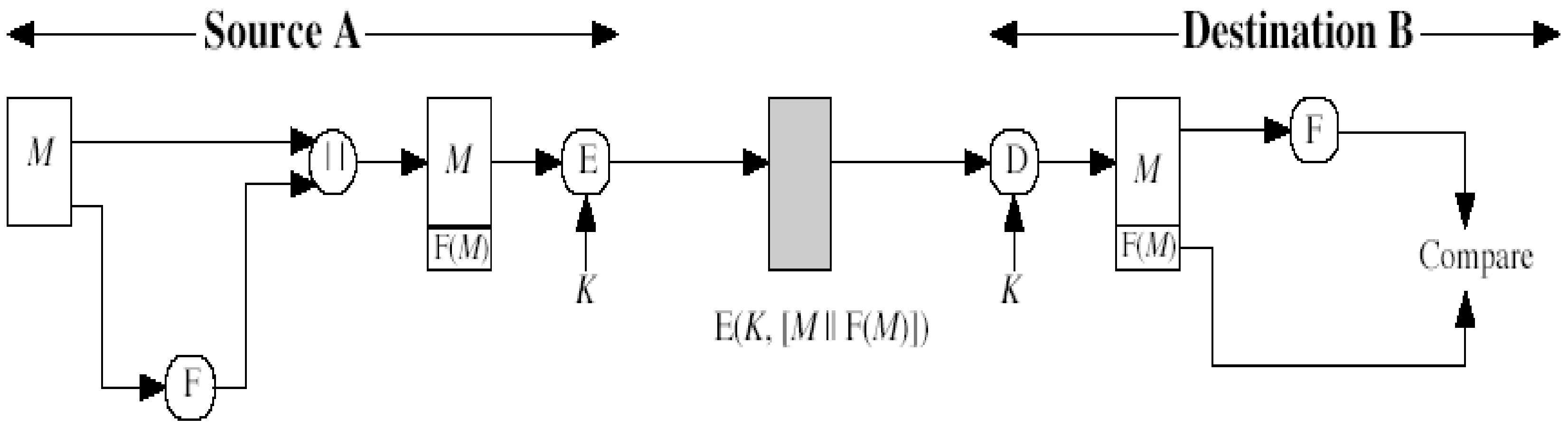
However, for these to be true we also need a mechanism that ***B can use to verify that the decryption actually succeeded***…(next slide)

Message Encryption: Symmetric Encryption

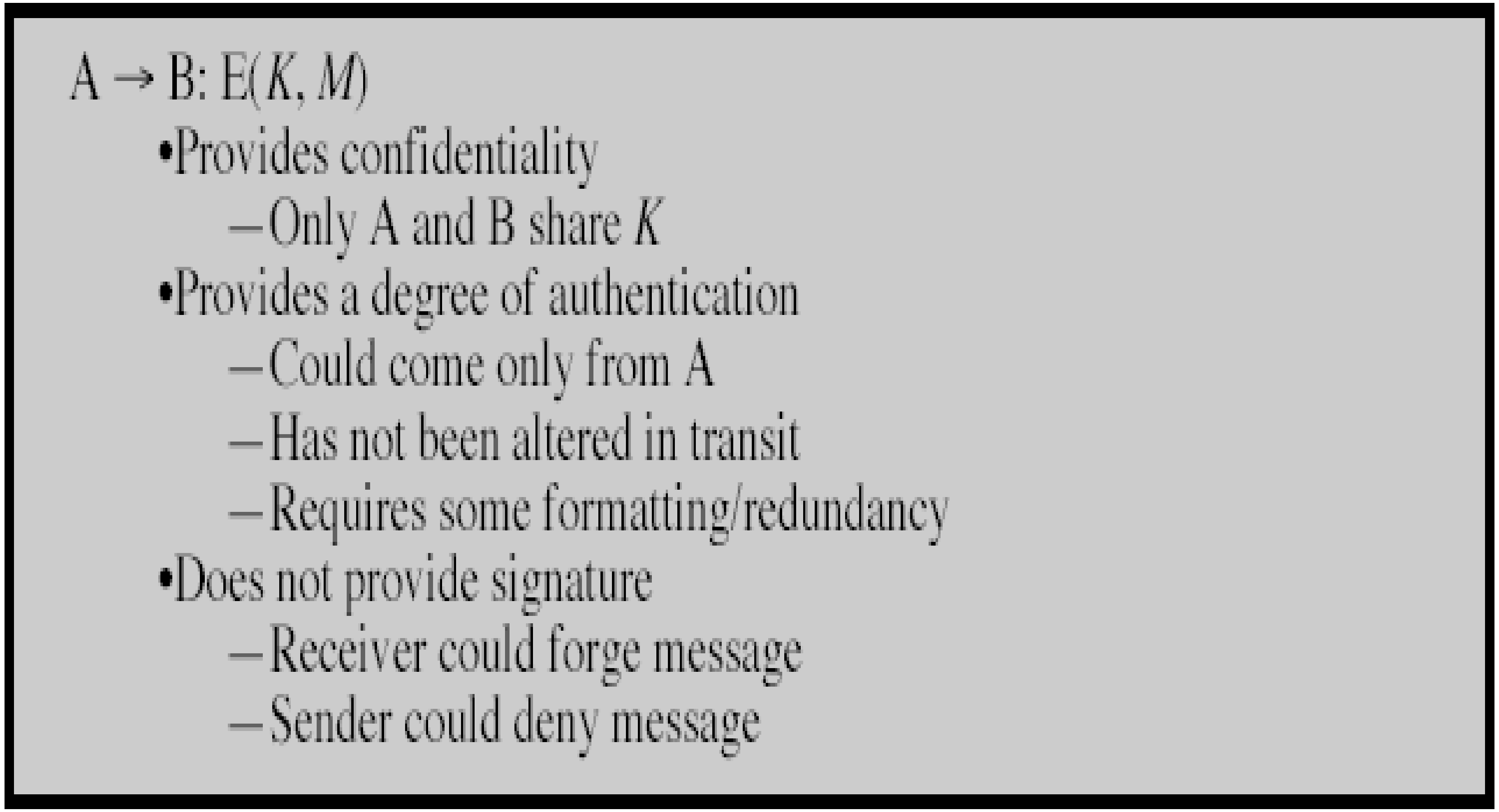
- Problem: given a decryption function D and a secret key K, the receiver will accept any input X and produce output $Y=D(K,X)$.
  - If X is the ciphertext of a legitimate message M, then Y is M, otherwise, meaningless sequence of bits.

Symmetric Encryption: Internal Control

- Solution: force the plaintext to have some structure
  - E.g. append an error-detecting code as a frame check sequence (FCS) to each message before encryption.
    - Sender: prepares a plaintext megs M, provides M as input to a function F that produces an FCS, appends FCS to M and encrypts entire block.
    - Receiver: decrypts the block, treats the result as a message with an appended FCS, and applies F to reproduce FCS. If the calculated FCS = incoming FCS, then the message is considered authentic.



Summary: Message Encryption (Symmetric)

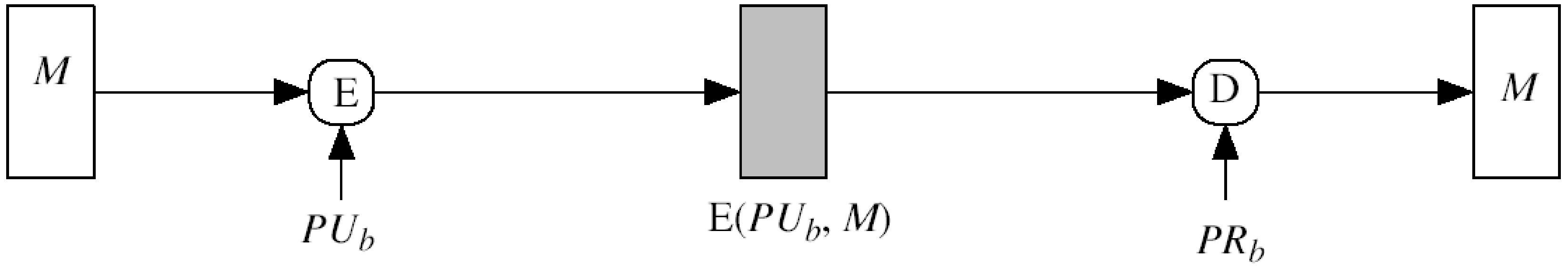


A → B: E(*K*, *M*)

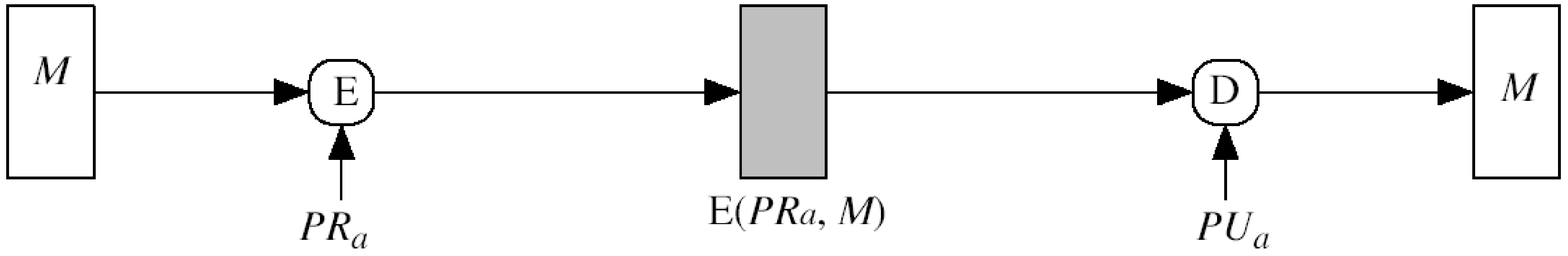
- Provides confidentiality
  - Only A and B share *K*
- Provides a degree of authentication
  - Could come only from A
  - Has not been altered in transit
  - Requires some formatting/redundancy
- Does not provide signature
  - Receiver could forge message
  - Sender could deny message

(a) Symmetric encryption

Public-key Encryption

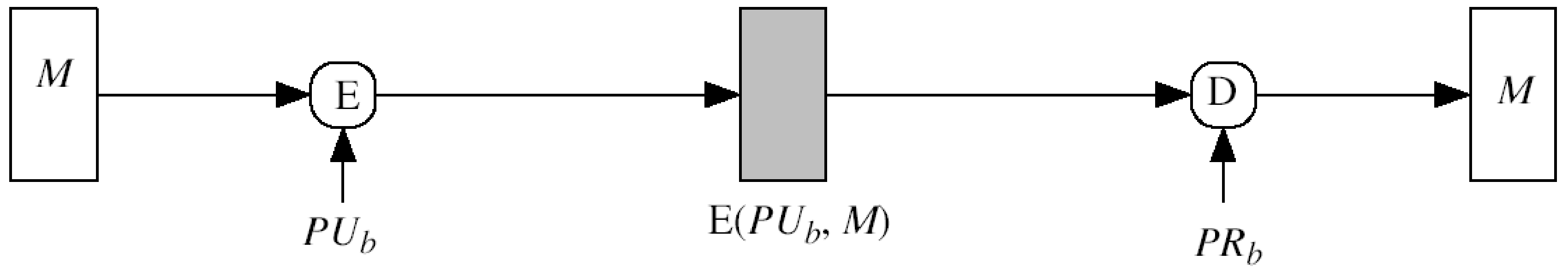


(b) Public-key encryption: confidentiality

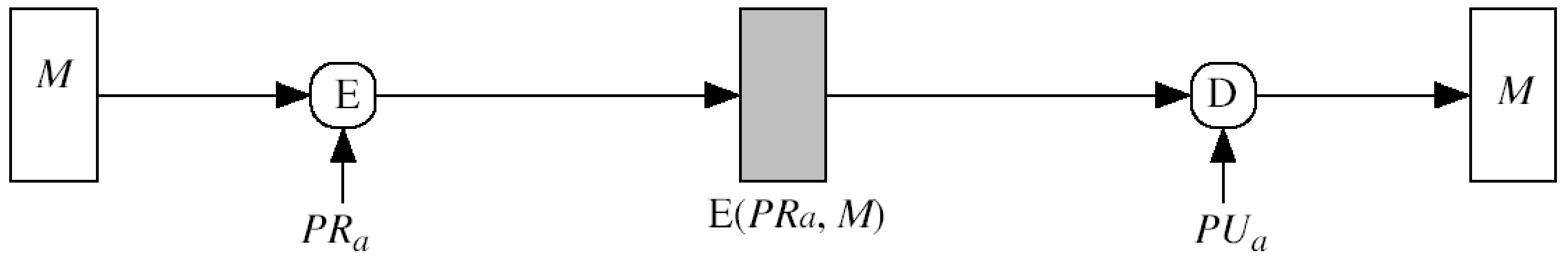


(c) Public-key encryption: authentication and signature

Public-key Encryption

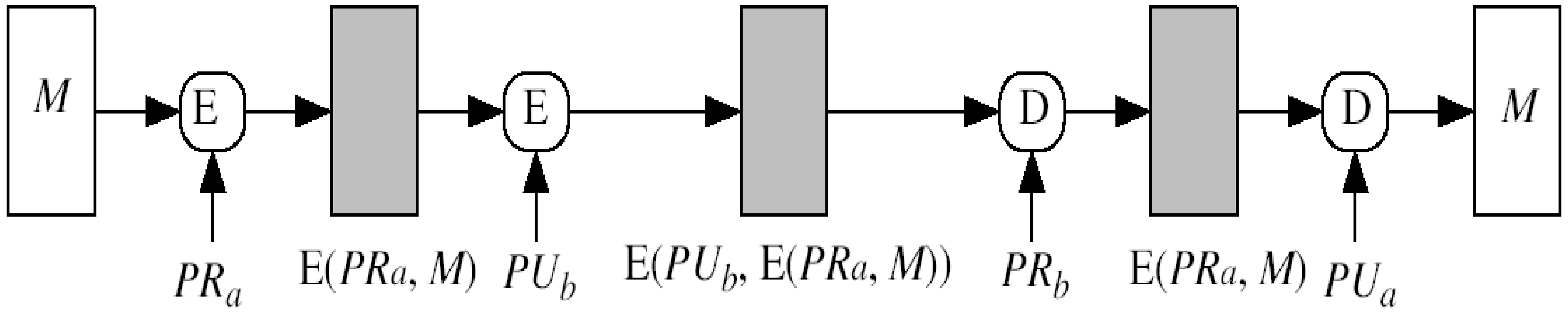


(b) Public-key encryption: confidentiality



(c) Public-key encryption: authentication and signature

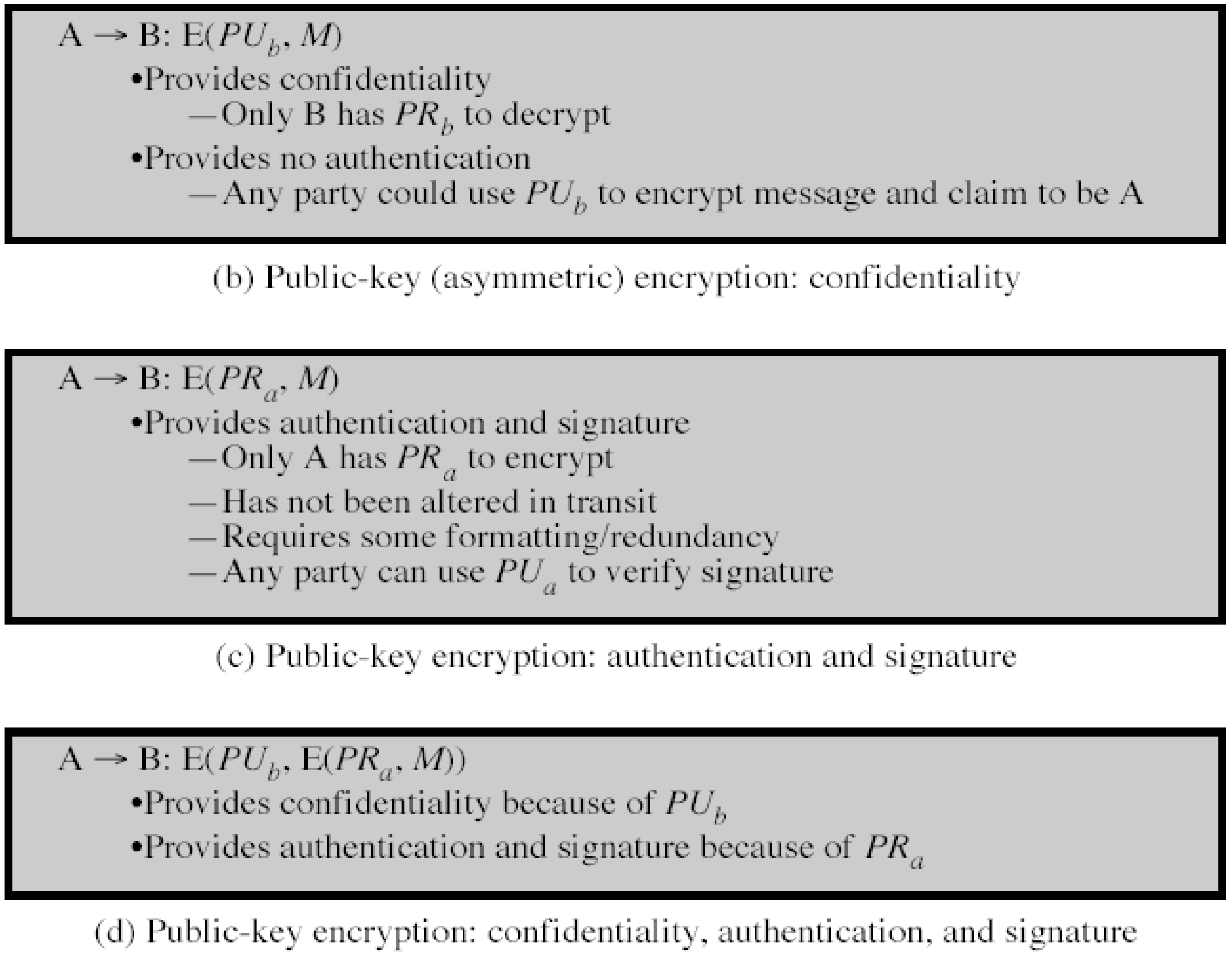
Public-key Encryption



(d) Public-key encryption: confidentiality, authentication, and signature

- Sender signs message using their private-key
- Then encrypts with recipients public key
- Have both secrecy and authentication
- Again, plaintext needs to have some internal structure so that the receiver can distinguish between well-formed plaintext and random bits.

Summary: Message Encryption (Public-key)



$A \rightarrow B$: $E(PU_b, M)$

- Provides confidentiality
  - Only B has PR_b to decrypt
- Provides no authentication
  - Any party could use PU_b to encrypt message and claim to be A

(b) Public-key (asymmetric) encryption: confidentiality

$A \rightarrow B$: $E(PR_a, M)$

- Provides authentication and signature
  - Only A has PR_a to encrypt
  - Has not been altered in transit
  - Requires some formatting/redundancy
  - Any party can use PU_a to verify signature

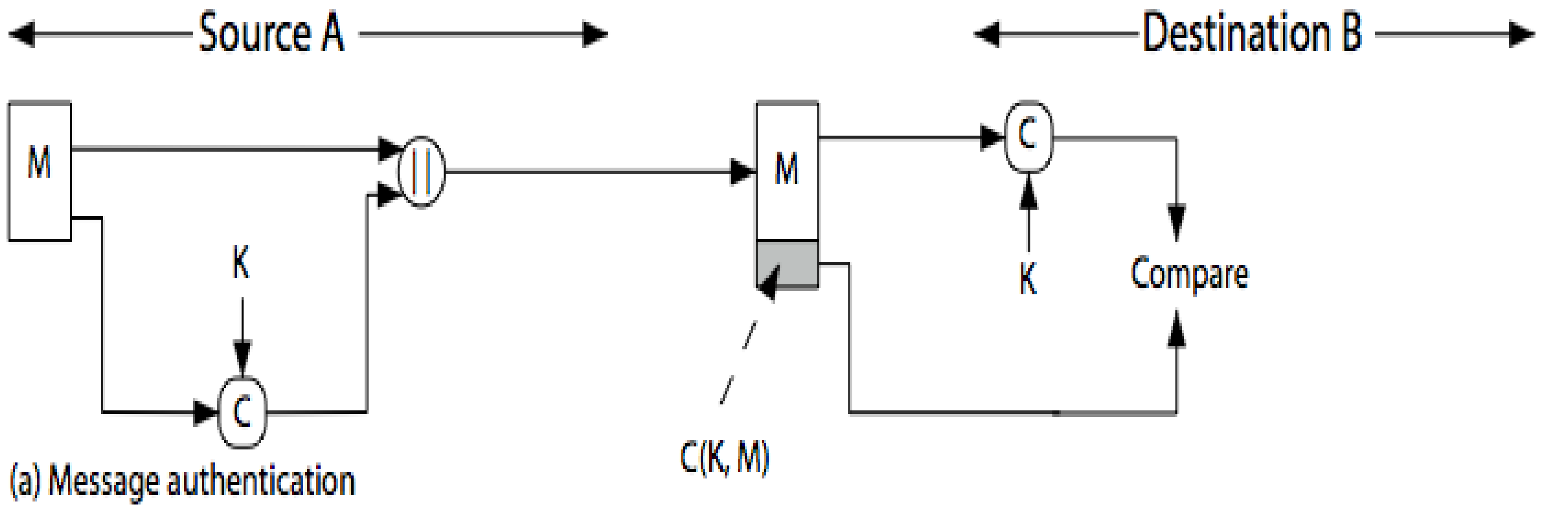
(c) Public-key encryption: authentication and signature

$A \rightarrow B$: $E(PU_b, E(PR_a, M))$

- Provides confidentiality because of PU_b
- Provides authentication and signature because of PR_a

(d) Public-key encryption: confidentiality, authentication, and signature

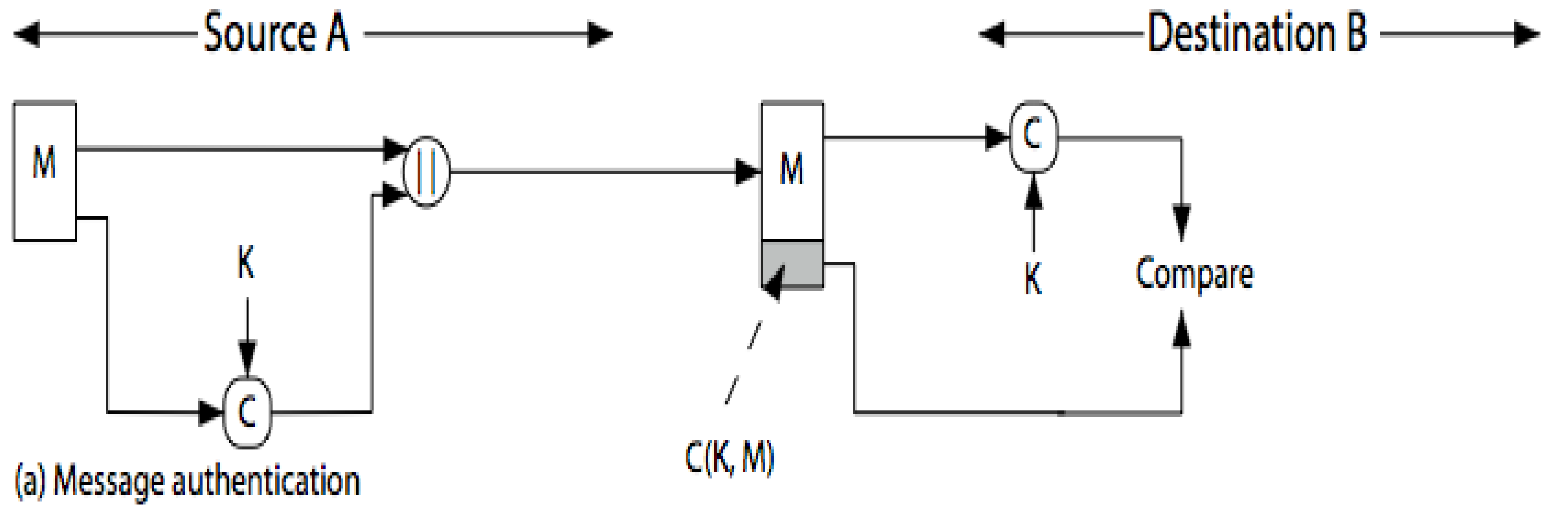
Message Authentication Code (MAC)



(a) Message authentication

- Ensures that the message is unaltered

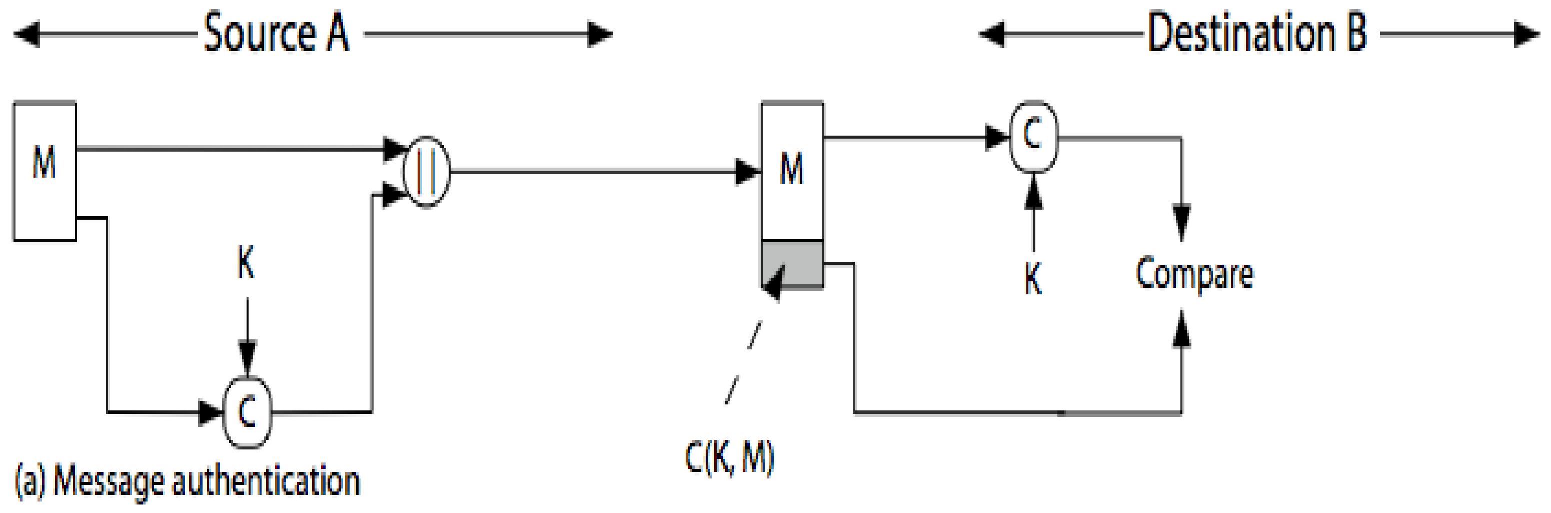
Message Authentication Code (MAC)



(a) Message authentication

- Provides a limited degree of authenticity (A and B trust each other, and if B knowns they did not generate the message, B knows the message came from A, since A is the only other party that has the encryption key)

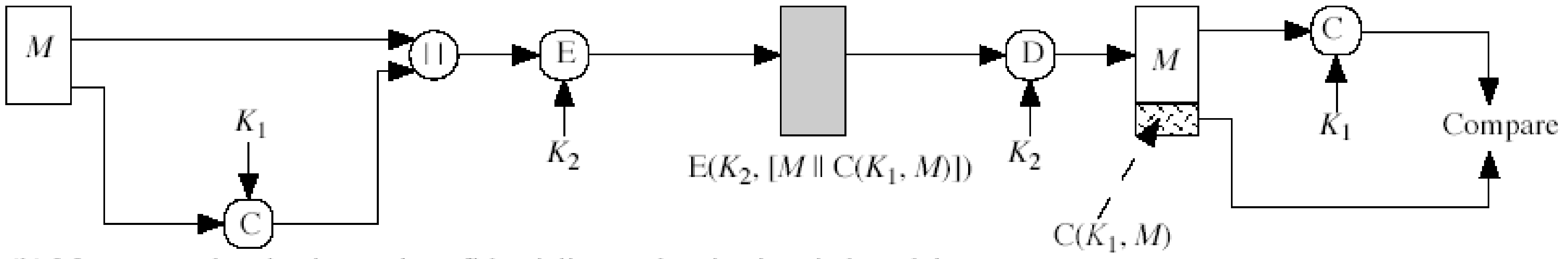
Message Authentication Code (MAC)



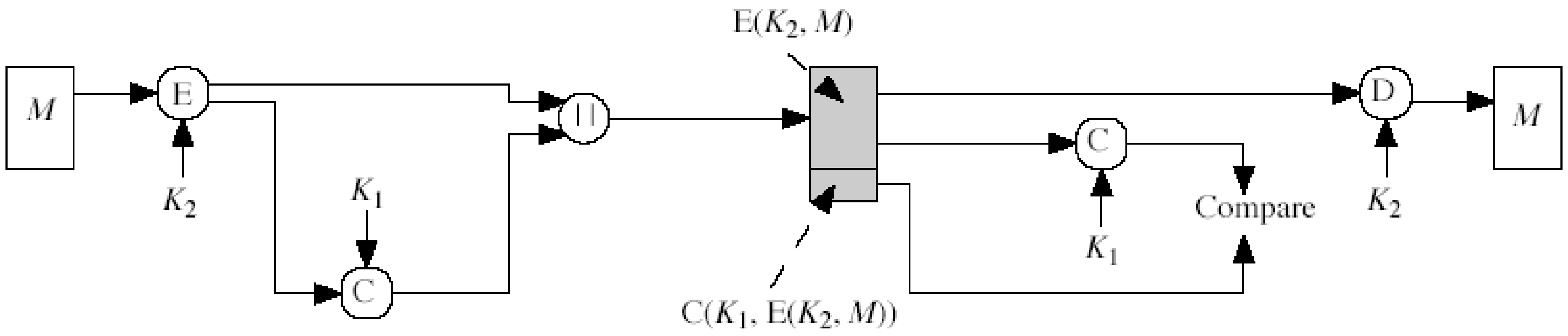
- Does not provide confidentiality

Message Authentication Codes

- Providing both authentication and confidentiality
  - Performing symmetric encryption after or before the MAC algorithm.



(b) Message authentication and confidentiality; authentication tied to plaintext

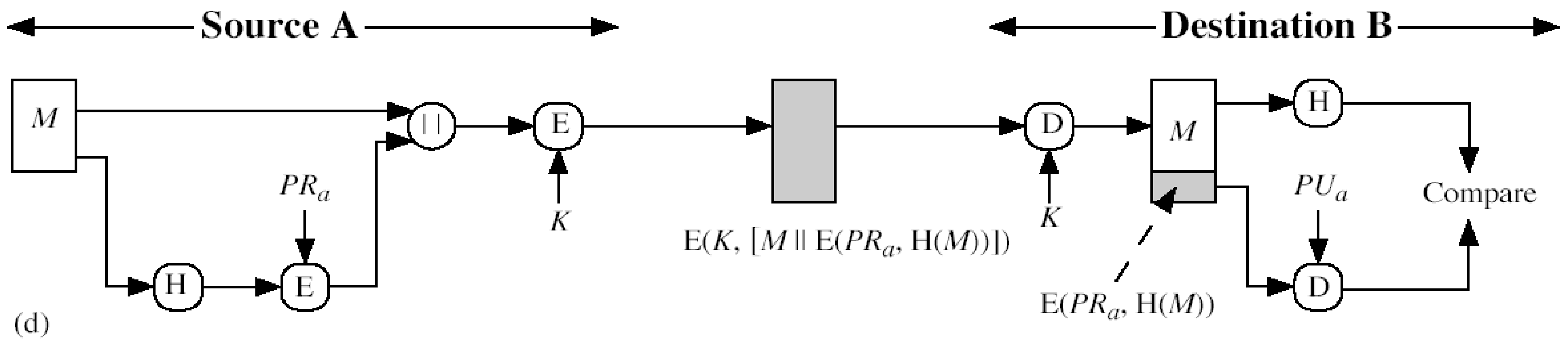


(c) Message authentication and confidentiality; authentication tied to ciphertext

Digital Signatures

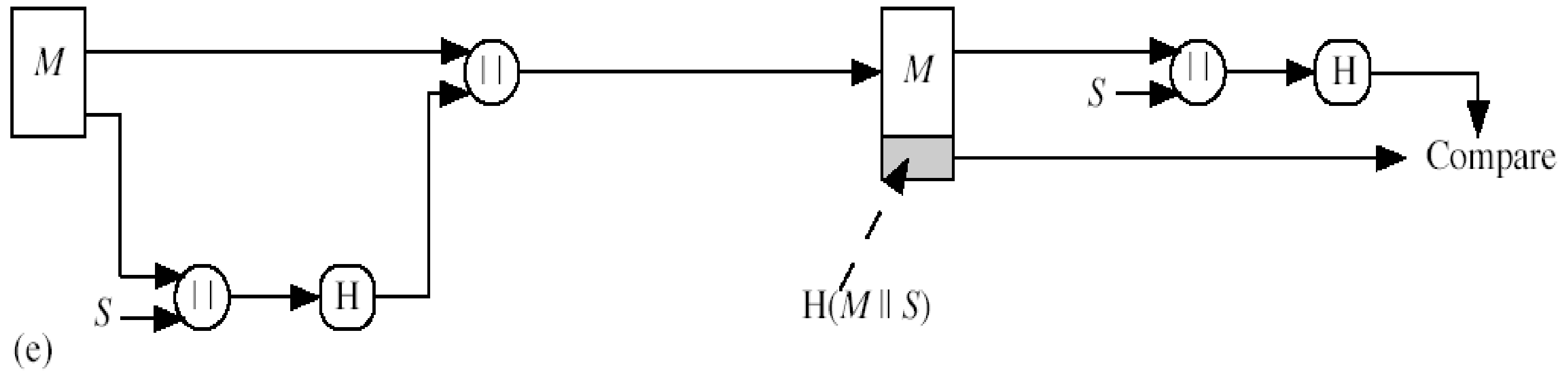
- When two communicating parties do not trust each other, symmetric encryption or MACs do not deliver the sufficient degree of authenticity

Using Hash Code (4)



- If confidentiality as well as a digital signature is desired, then the message plus the private-key-encrypted hash code can be encrypted using a symmetric secret key.

Using Hash Code (5)

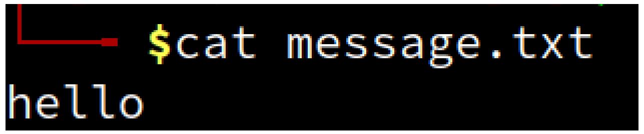


- It is possible to use a hash function but no encryption for message authentication
- Two parties share a common secret value S.
- A computes the hash value over the concatenation of M and S, and appends the resulting hash value to M
- B has S, so can re-compute the hash value to verify.
- Because S is not sent, an attacker cannot modify a message or generate a false message.

Using Hash Code (6)

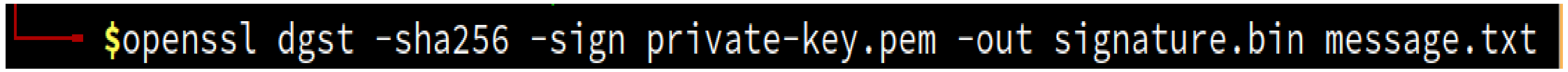
- Example: **Generating** a digital signature using public key cryptography and SHA-256 hash:
  - openssl dgst -sha256 -sign private-key.pem -out signature.bin message.txt
    - dgst -sha256: the hashing algorithm to use
    - -sign private-key.pem: the private key to use for the signature
    - -out signature.bin: the file in which to store the signature
    - message.txt: the file for which to generate the signature

The contents of message.txt



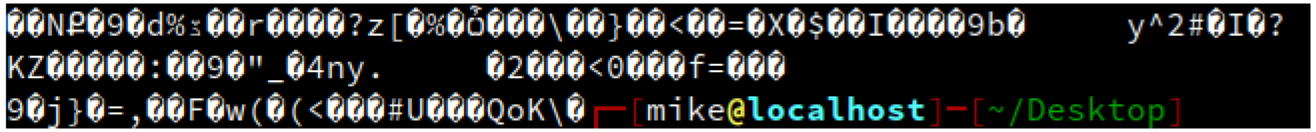
```
└──╼ $cat message.txt
hello
```

Running the command



```
└──╼ $openssl dgst -sha256 -sign private-key.pem -out signature.bin message.txt
```

The contents of signature.bin (it's in binary)

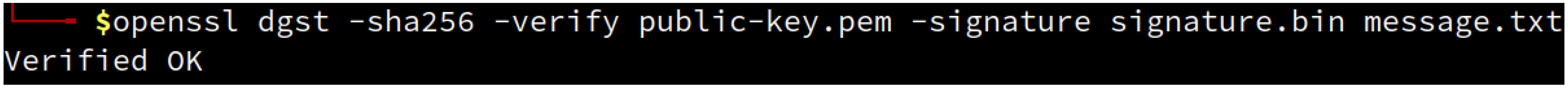


```
��N₽��9�d%₃��r����?z[�%�ở���\��}��<��=�X�$��I�����9b�     y^2#�I�?
KZ�����:��9�"_�4ny.     �2���<0���f=���
9�j}�=,��F�w(�(<���#U���QoK\�┌─[mike@localhost]─[~/Desktop]
```

Using Hash Code (7)

- Example: **Verifying** a digital signature using public key cryptography and SHA-256 hash:
  - openssl dgst -sha256 -verify public-key.pem -signature signature.bin message.txt
    - dgst -sha256: the hashing algorithm to use
    - -verify public-key.pem: the public key to use for validating the signature (linked to the private key used for creating the signature).
    - -signature signature.bin: the file containing the signature to verify
    - message.txt: the file for which to verify the signature (same as on the previous slide)

When the signature matches:

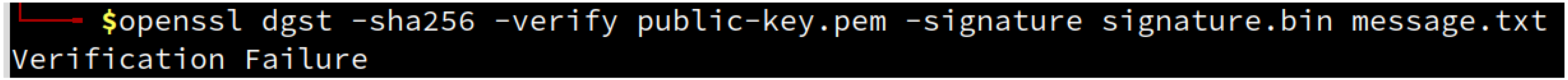


```
└─$ openssl dgst -sha256 -verify public-key.pem -signature signature.bin message.txt
Verified OK
```

Using Hash Code (7)

- Example: **Verifying** a digital signature using public key cryptography and SHA-256 hash:
  - openssl dgst -sha256 -verify public-key.pem -signature signature.bin message.txt
    - dgst -sha256: the hashing algorithm to use
    - -verify public-key.pem: the public key to use for validating the signature (linked to the private key used for creating the signature).
    - -signature signature.bin: the file containing the signature to verify
    - message.txt: the file for which to verify the signature (same as on the previous slide)

When we change the string in message.txt to "hello1"

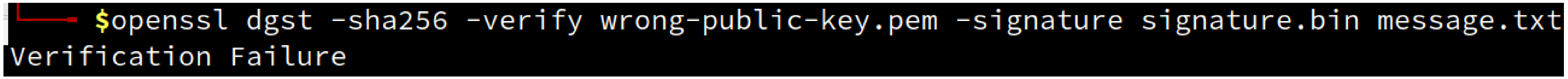


```
└──╼ $openssl dgst -sha256 -verify public-key.pem -signature signature.bin message.txt
Verification Failure
```

Using Hash Code (7)

- Example: **Verifying** a digital signature using public key cryptography and SHA-256 hash:
  - openssl dgst -sha256 -verify public-key.pem -signature signature.bin message.txt
    - dgst -sha256: the hashing algorithm to use
    - -verify public-key.pem: the public key to use for validating the signature (linked to the private key used for creating the signature).
    - -signature signature.bin: the file containing the signature to verify
    - message.txt: the file for which to verify the signature (same as on the previous slide)

When we use the wrong public key to verify the signature of the unmodified message.txt (i.e., the public key not linked to the private key used to generate the signature)



```
└──╼ $openssl dgst -sha256 -verify wrong-public-key.pem -signature signature.bin message.txt
Verification Failure
```

Cryptology Fundamentals

Key Clustering

- When two different keys produce the same plaintext

Unconditional Security

- Unconditional security
  - No matter how much computer power or time is available, the cipher cannot be broken since the ciphertext provides insufficient information to determine the corresponding plaintext

Unconditional Security

- Unconditional security
  - No matter how much computer power or time is available, the cipher cannot be broken since the ciphertext provides insufficient information to determine the corresponding plaintext
  - No encryption algorithm is unconditionally secure except the one-time pad

Computational Security

- Computational security
  - Given limited computing resources (e.g., time needed for calculations is greater than age of universe), the cipher cannot be broken
    - The cost of breaking the cipher exceeds the value of the encrypted information
    - The time required to break the cipher exceeds the useful lifetime of the information

Additional References

- http://www.eventid.net/docs/desexample.htm
- http://www.iusmentis.com/technology/encryption/des/
- http://www.research.ibm.com/journal/rd/383/coppersmith.pdf

Acknowledgement

- Some slides are borrowed from Dr. Ping Yang