

Lec 11: Hash, MAC and AE

CSED415: Computer Security
Spring 2025

Seulbae Kim



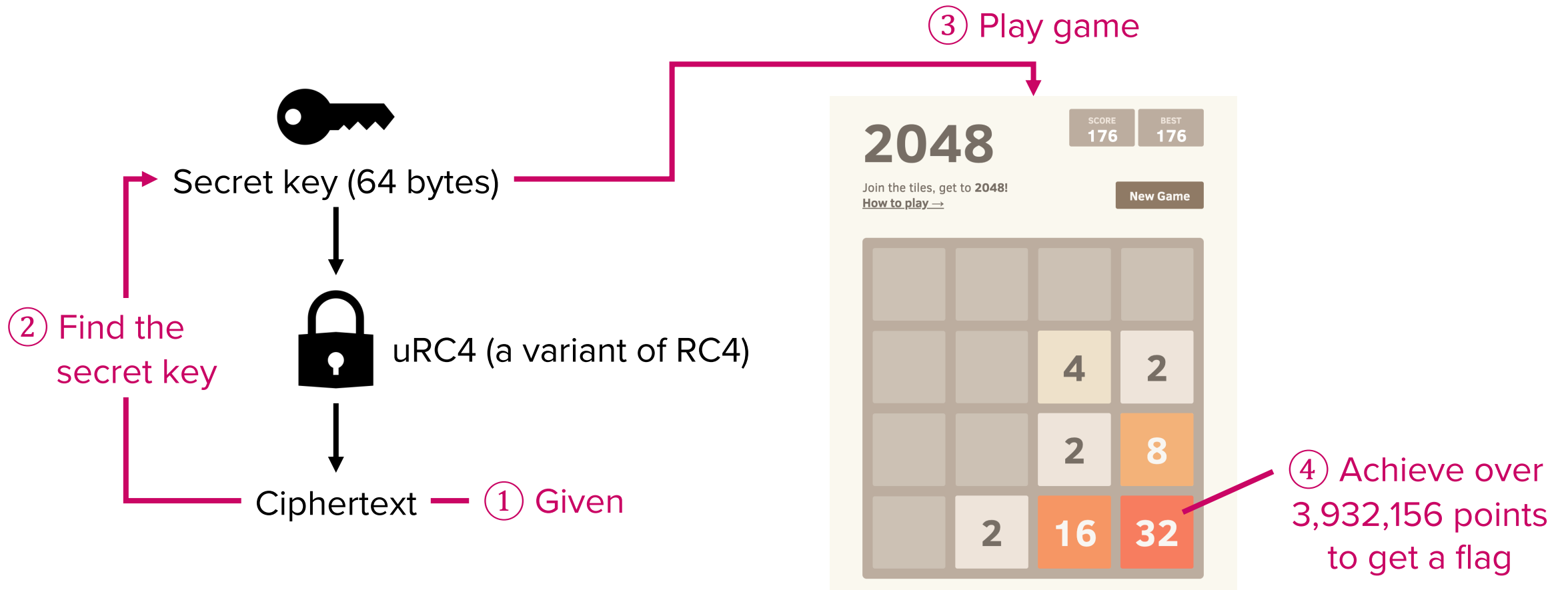
Administrivia

- No in-person class on Thursday, Mar 27
 - I will be out of town for another presentation
 - A lecture recording will be uploaded
- Lab 03 is out:
 - Due: Friday, April 4
- 10-minute proposal presentations:
 - When: Thursday, April 3, during regular class time
- Midterm exam:
 - When: Tuesday, April 8, during regular class time

Overview of Lab 03

Phase 1: uRC4 service (server.py)

Phase 2: target



Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
Confidentiality	<ul style="list-style-type: none">✓ One Time Pad (OTP)✓ Block ciphers (DES, AES)✓ Stream ciphers	<ul style="list-style-type: none">✓ DH secure key exchange✓ ElGamal encryption✓ RSA encryption
Integrity & Authentication	<ul style="list-style-type: none">• Message Authentication Code (MAC)	<ul style="list-style-type: none">• Digital signature

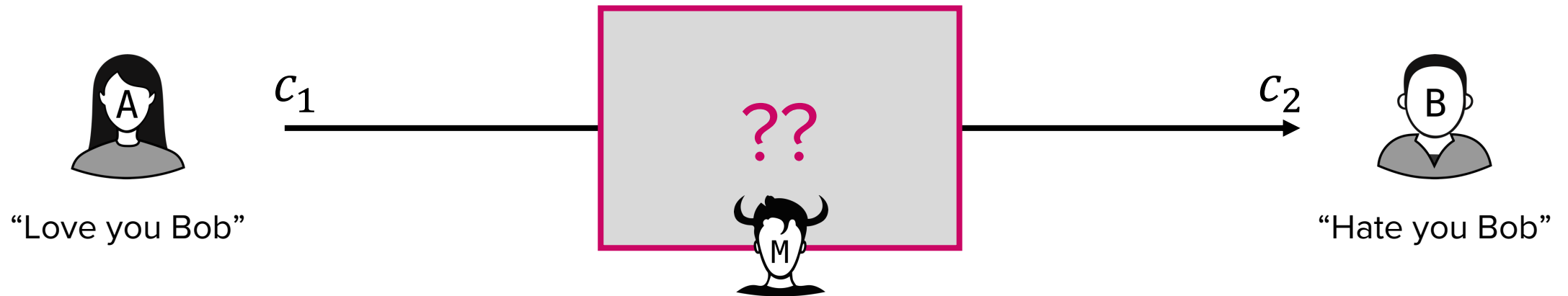
Additional Tool

- Hash functions

Hash Functions

New problem: Data integrity

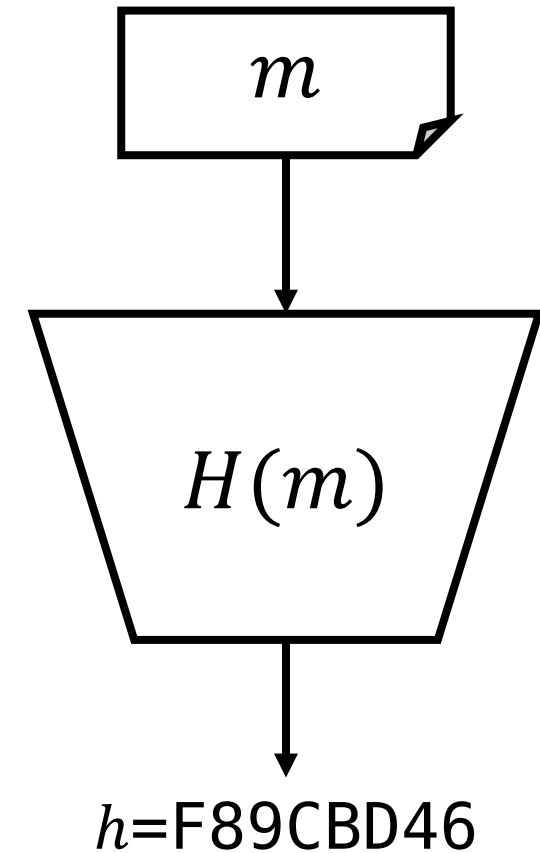
- Encryption does not guarantee integrity (Lectures 9 & 10)



- How do we ensure that Alice's message was not altered in transit?
 - In other words, how do we verify $c_1 == c_2$?

Hash functions

- Hash function H
 - Takes a message m of *arbitrary* length
 - Creates a message digest h of *fixed* length
 - h is also called hash, hash value, hash digest, ...
- Key properties
 - Correctness: $H(m)$ should be deterministic
 - Hashing m should always produce the same h
 - Efficiency: $H(m)$ should be efficient



Hash function outputs a fixed-length hash

- For example, MD5 is a “128-bit hash function”
 - Produces 16-byte hash digests, e.g.,
 - "a" → 0cc175b9c0f1b6a831c399e269772661
 - "aa" → 4124bc0a9335c27f086f24ba207a4912
 - "a"*2048 → b7ea2d21ad2ef3e28085d30247603e0b

Arbitrary-length input

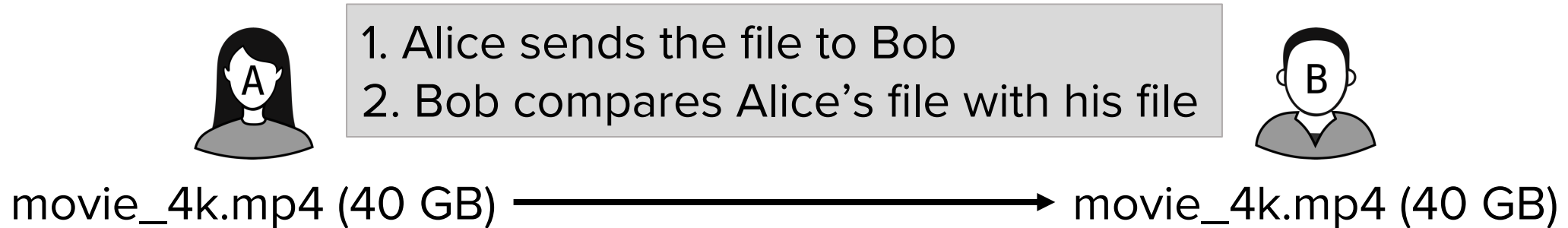
Fixed-length (16-byte) output

- Confirmation:

```
import hashlib
print(hashlib.md5(b"a").hexdigest())
print(hashlib.md5(b"aa").hexdigest())
print(hashlib.md5(b"a" * 2048).hexdigest())
```


Typical usage of hash function

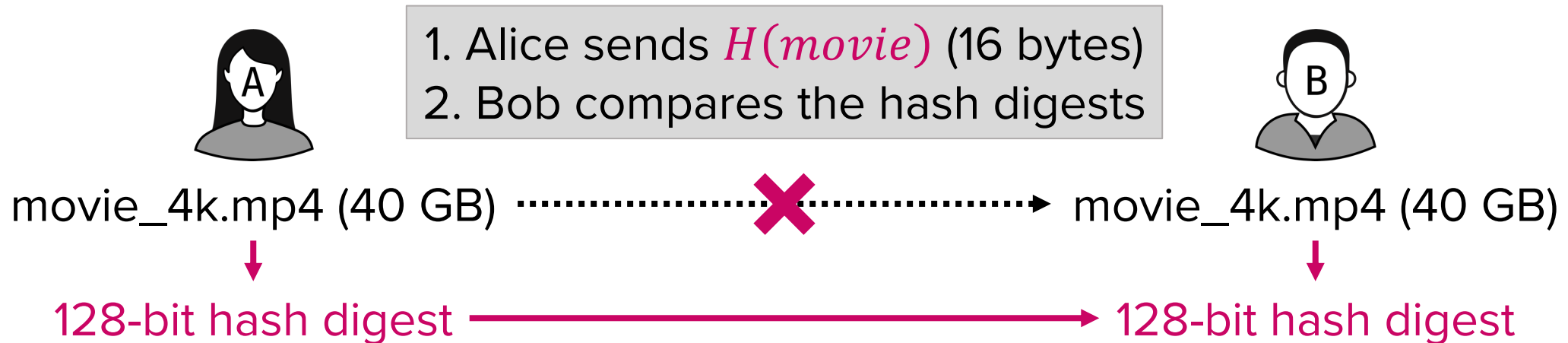
- Scenario: File integrity verification
 - Alice and Bob both downloaded a 40-GB movie file from the internet
 - They want to verify if the two files are identical
 - Naïve way:



→ Waste of network bandwidth and computational powers ☹

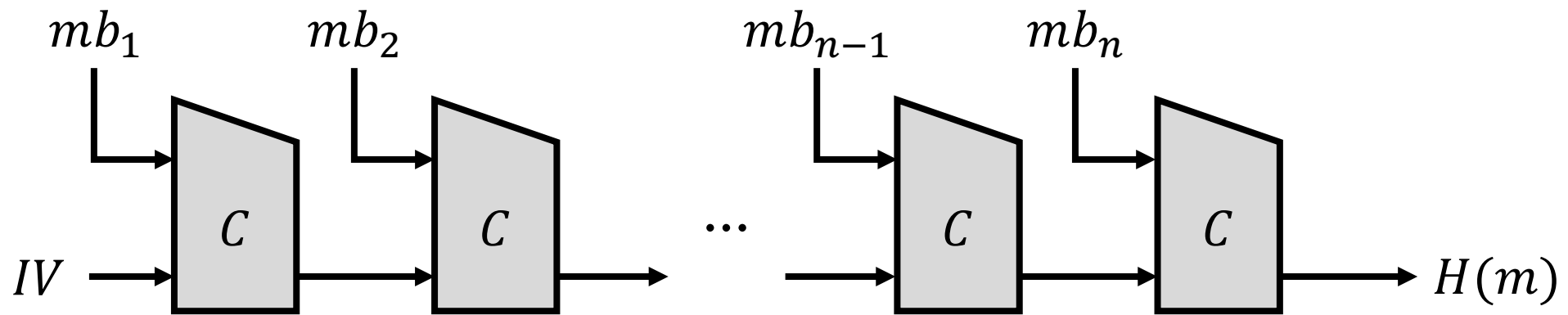
Typical usage of hash function

- Scenario: File integrity verification
 - Alice and Bob both downloaded a 40-GB movie file from the internet
 - They want to verify if the two files are identical
 - Using a hash function:



Building hash functions

- Underlying method: Merkle-Damgård construction (1979)
 - A method of building a **cryptographic** hash function H from collision-resistant one-way compression function C
 - Used by many hash functions, including MD5, SHA-1 and SHA-2 families
 - $m = mb_1 || mb_2 || \dots || mb_n$ (mb_i : i -th message block, $||$: Concatenation)



(IV: Initialization Vector,
algorithm-specific)

C always outputs a fixed length output

Cryptographic hash functions?

- Cryptographic hash functions are secure hash functions that satisfy two properties:
 - One-wayness (OW)
 - Definition: For any given hash value h , it is computationally infeasible to find m such that $H(m) = h$
 - Collision resistance (CR)
 - Definition: It is computationally infeasible to find a pair of plaintexts m_1 and m_2 such that $H(m_1) = H(m_2)$

Computationally infeasible?

- Classification of problems/computations
 - **Easy:** If there exists an algorithm that solves the problem in $poly(n)$ time, where n is the bit length of the input
 - Running time is bound by a polynomial function (e.g., n^2) of the input size
 - **Hard:** If, to the best of our knowledge, no polynomial-time algorithm exists for solving the problem
 - e.g., Best known algorithms run in exponential time (e.g., 2^n), so as n grows, the required computations quickly become impractical
 - “Computationally infeasible” falls under this category

One-wayness (OW)

- Informally:
 - Given an output h , it is infeasible to find any m such that $H(m) = h$
- Formally:
 - H is one-way, if for all polynomial-time adversary A who randomly selects m' from the plaintext domain,

$$Adv_H^{ow}(A) = Prob[H(m') = h] \text{ is negligible}$$

“Advantage”

- Negligible: Attacker’s advantage of winning decreases quickly as input size n grows. If the problem is hard (computationally infeasible), then the chance of winning is negligible

OW examples

OW: Is $Adv_H^{ow}(A) = Prob[H(m') = h]$ negligible?

- Is $H(x) = 0$ a one-way hash function?

No. A can easily find multiple m 's. $Prob[H(m') = 0] = 1$

- Is the following summation checksum one-way?

<u>Message</u>	<u>Ascii-hex format</u>
C S E D →	43 53 45 44
4 1 5 0 → +	34 31 35 30
	<hr/> <hr/>
Checksum:	77 84 7A 74

No. A can easily find “CSED4150” (and other m 's) given 77847A74

OW examples

OW: Is $Adv_H^{ow}(A) = Prob[H(m') = h]$ negligible?

- If H and G are length-preserving hash functions that are OW, is $F(x) = H(x) \oplus G(x)$ one-way? (\oplus : XOR)

No. If $H = G$, then $F(x) = 0$. Constant function is not OW

- If $H(x) = x \bmod p$ where p is a large, pre-selected prime number, is it one-way?

No. A can easily find multiple m 's that satisfy $m \bmod p = h$

Collision resistance (CR)

- Collision: Two different inputs resulting in the same output
 - $m_1 \neq m_2$ and $H(m_1) = H(m_2)$
- Can a hash function have no collision?
 - No. If the input domain is larger than 2^n for a n -bit hash function, there must exist collisions (by the pigeonhole principle)
 - Collision resistance is **not** about having no collisions!
 - It is about making it infeasible to find collisions

Collision resistance (CR)

- Informally:
 - It is computationally infeasible to find a pair of plaintexts m_1 and m_2 such that $H(m_1) = H(m_2)$
- Formally:
 - H is collision-resistant, if for all polynomial time adversary A ,

$$Adv_H^{cr}(A) = Prob[H(m_1) = H(m_2)] \text{ is negligible where } m_1 \neq m_2$$

“Advantage”

CR examples

CR: Is $Adv_H^{cr}(A) = Prob[H(m_1) = H(m_2)]$ negligible where $m_1 \neq m_2$?

- Is $H(x) = 0$ collision-resistant?

No. Every pair of $\langle m_1, m_2 \rangle$ is a collision.

- If $H(x)$ is sum of bits in $m \bmod 2$ (i.e., parity bit), is H CR?

No. A counter-example: $m_1 = 01, m_2 = 10$

CR examples

CR: Is $Adv_H^{cr}(A) = Prob[H(m_1) = H(m_2)]$ negligible where $m_1 \neq m_2$?

- Let $H: \{0, 1\}^{256} \rightarrow \{0, 1\}^{128}$ be defined by

(H is a function that maps a 256-bit input to a 128-bit output)

$$H(x) = H(x_L || x_R) = AES(x_L) \oplus AES(x_R)$$

($||$: Concatenation)

- Q) Is H collision-resistant?

No. A counter-example: $m_1 = 1010 \ 0001, m_2 = 0001 \ 1010$

CR examples

CR: Is $Adv_H^{cr}(A) = Prob[H(m_1) = H(m_2)]$ negligible where $m_1 \neq m_2$?

- Is CRC-32 (Cyclic Redundancy Check) collision-resistant?
 - CRC-32 is widely used for calculating checksum for error detection. It hashes input byte sequence to a 32-bit value

No. Because of its small output size, the birthday paradox implies that we can find collisions with about a 50% chance in only 65536 trials!

CR examples

CR: Is $Adv_H^{cr}(A) = Prob[H(m_1) = H(m_2)]$ negligible where $m_1 \neq m_2$?

- Demo: Finding hash collision of CRC-32
 - Hashing 123,565 English words found online:

```
import os, zlib

filename = "nordpass-word-list.txt"
url = "https://gist.githubusercontent.com/atoponce/4c4692940522947b8611d33d7cf3225d/raw/cd078528f2c2e5dbb8c750f7b2d1a9508c094840"
os.system(f"wget {url}/{filename}")

with open(filename, "r", encoding="utf-8") as f:
    lines = f.readlines()

crc_dict = dict()
for line in lines:
    word = line.split("\t")[-1].strip().encode("utf-8")
    c = zlib.crc32(word)
    if c not in crc_dict:
        crc_dict[c] = [word]
    else:
        crc_dict[c].append(word)
    print("collision!", crc_dict[c])
```

A generic attack for finding collisions

- Birthday problem:
 - If you choose a group of N random people, what is the probability that at least one pair of individuals have the same birthday?
- Birthday paradox:
 - If $N = 23$, the probability is 50%. Only 23 people!
 - Event E : Everyone has different b-day $\rightarrow 365 \times 364 \times \cdots \times (365 - 22) = {}_{365}P_{23}$
 - Possible outcomes: Each person has 365 choices $\rightarrow 365^{23}$
 - Prob. that no one shares the birthday: $P(E) = \frac{{}_{365}P_{23}}{365^{23}} \approx 0.492$
 - Therefore, prob. that at least two people share the b-day: $1 - P(E) \approx 50\%$

A generic attack for finding collisions

- Birthday attack
 - Similarly, the probability of detecting a hash collision via brute-forcing is much higher than we expect
 - Approximation:
 - When there are 2^n possible data, if we have $\sqrt{2^n}$ data, the probability to find a collision is $> 50\%$
 - In other words, one can find a collision for 50% chance after $\sqrt{2^n}$ trials
 - Birthday collision: 365 total days $\rightarrow n = 9$ bits to represent 365
 $\rightarrow \sqrt{2^9} = 22.6 \rightarrow$ approximately 23 trials for 50% chance

A generic attack for finding collisions

- Collision-resistance of a n -bit hash function is bounded by $\sqrt{2^n}$
- Cryptanalysis of hash functions

Function	n	Trials needed by birthday attack	Existing attacks
MD4	128	2^{64}	< sec
MD5	128	2^{64}	1 min
SHA-1	160	2^{80}	2^{69} trials (2005)
SHA-1	160	2^{80}	$2^{63.1}$ trials (2017)
SHA-256	256	2^{128}	-

Attacks requiring less trials than B-day attack are considered feasible attacks

MD5 hash

- MD5
 - An old standard hash function without collision-resistance
 - Generates 128-bit hash digests
- Severe weaknesses have been discovered
 - e.g., Chosen-prefix collisions attacks (Marc Stevens, et al.)
 - Start with two arbitrary plaintexts m_1 and m_2
 - One can compute suffixes s_1 and s_2 such that $md5(m_1||s_1) = md5(m_2||s_2)$ in only 250 trials
 - Using this approach, a pair of different files (e.g., jpeg) with the same MD5 hash value can be computed

Collision in practice – MD5 is completely broken

POSTECH

- Download ship.jpg and plane.jpg from <https://natmchugh.blogspot.com/2015/02/create-your-own-md5-collisions.html>



```
import hashlib

f1 = open("ship.jpg", "rb").read()
f2 = open("plane.jpg", "rb").read()

print(hashlib.md5(f1).hexdigest())
print(hashlib.md5(f2).hexdigest())
```

Both files are hashed to 253dd04e87492e4fc3471de5e776bc3d

CR vs OW

- Does collision-resistance imply one-wayness?

- It does not

e.g., $H(x) = x$ is CR, but not OW

- Does one-wayness imply collision-resistance?

- It does not

e.g., $H(x)$ is a secure hash function, which is one-way.

We can build $G(x) = H(x_0x_1 \dots x_{n-2})$ (ignores the last bit of x)

→ $G(x)$ is still OW because it is hard to find x from $G(x)$ if n is large

→ However, $G(x)$ is not CR. $G(x_0x_1 \dots x_{n-2}0) = G(x_0x_1 \dots x_{n-2}1)$

*Notation: $x = x_0x_1x_2 \dots x_{n-1}$ (x_i : i -th bit of x)

Using hash functions for integrity

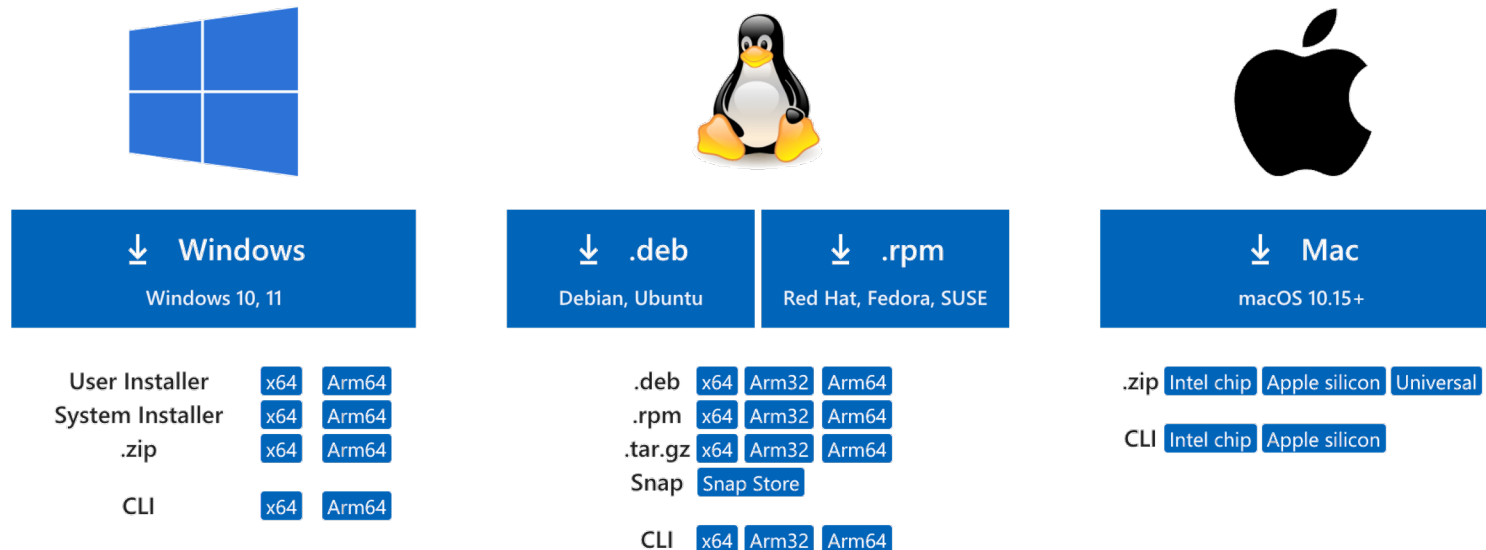
- Scenario
 - Microsoft publishes a new version of vscode
 - Alice downloads the installer
 - How does she verify that no one has tampered with it in transit?
 - In other words, how can she verify the installer's integrity?

Using hash functions for integrity

- e.g., vscode download site

Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.



The image shows the Visual Studio Code download page layout. It features three main columns for different operating systems: Windows, Linux, and Mac. Each column has a platform icon at the top, a download button with a dropdown arrow, and a list of available download options with their respective architectures.

Platform	Download Options	Architectures	
Windows	User Installer	x64, Arm64	
	System Installer	x64, Arm64	
	.zip	x64, Arm64	
	CLI	x64, Arm64	
Linux	.deb	Debian, Ubuntu	x64, Arm32, Arm64
		Red Hat, Fedora, SUSE	x64, Arm32, Arm64
	.rpm	Red Hat, Fedora, SUSE	x64, Arm32, Arm64
		Snap Store	x64, Arm32, Arm64
	.tar.gz	x64, Arm32, Arm64	
	CLI	x64, Arm32, Arm64	
Mac	.zip	Intel chip, Apple silicon, Universal	
	CLI	Intel chip, Apple silicon	

Using hash functions for integrity

- e.g., vscode download site

SHA-256 hashes

Windows User Installer (x64)	7bda1c7dfc670489155db2f8fc1f48c209b92fb6145a320d677dccf0bce921b6
Windows System Installer (x64)	c49f51562a99e19412d968a81ad653960c4861e95f7cd04e49e15c42e139a9ee
Windows .zip (x64)	564d545cc1099bcb48c7eb5b5efb292d7dea2e02a37d8bd84a907e171f3092ce
Windows CLI (x64)	c306eb45d0ef485885308090c66f1a0328aece3ccdb4cc1554a7b3ad54f639e7
Windows User Installer (Arm64)	c91bd092b71c3d948bb8f32fc5f83e454f4ec90eee7b0e9cf58decf880fea54e
Windows .zip (Arm64)	a63c75550322fca979e672d09cc46385d02d1e7a9d07f12b2b078af4f4005478
Windows System Installer (Arm64)	63178497481ddf816396566904e99b4b3a817637f1c9170255fa294babed9f79
Windows CLI (Arm64)	0d8ded98088669219b52784f48c0b4f2364dbefd104c87dcfbf048827880fe8a
Linux .deb (x64)	3340b2649e486adfde2452418599acb64c1dc3998087d715d244f10302a89b94
Linux .rpm (x64)	841f72255270b647c657f6a20728d271cf08f94a07b7625fc91b548545efac8b
Linux .tar.gz (x64)	c2e97cdc63ff1bcbfbb10c227b5398623d21f21e487108fa1d740dabe7d37985
Linux CLI (x64)	1cb4ee01e6941b369c69253f12ff0eed15071221c7f16858a49694cd981bfb6c

Using hash functions for integrity

- Method
 - Microsoft hashes the installer binary with SHA-256 and publishes the hash digests on its website
 - Alice hashes the installer binary she downloaded with SHA-256 and checks if the hash matches the hash on the website
- Security
 - If Alice downloads a malicious program, the hash would not match
 - An attacker cannot create a malicious program with the same hash as the original installer (SHA-256 is collision-resistant)

Using hash functions for integrity

- Another scenario
 - Alice and Bob want to communicate over an insecure channel and verify integrity of their messages
 - Mallory can tamper with the messages

Using hash functions for integrity

- Method

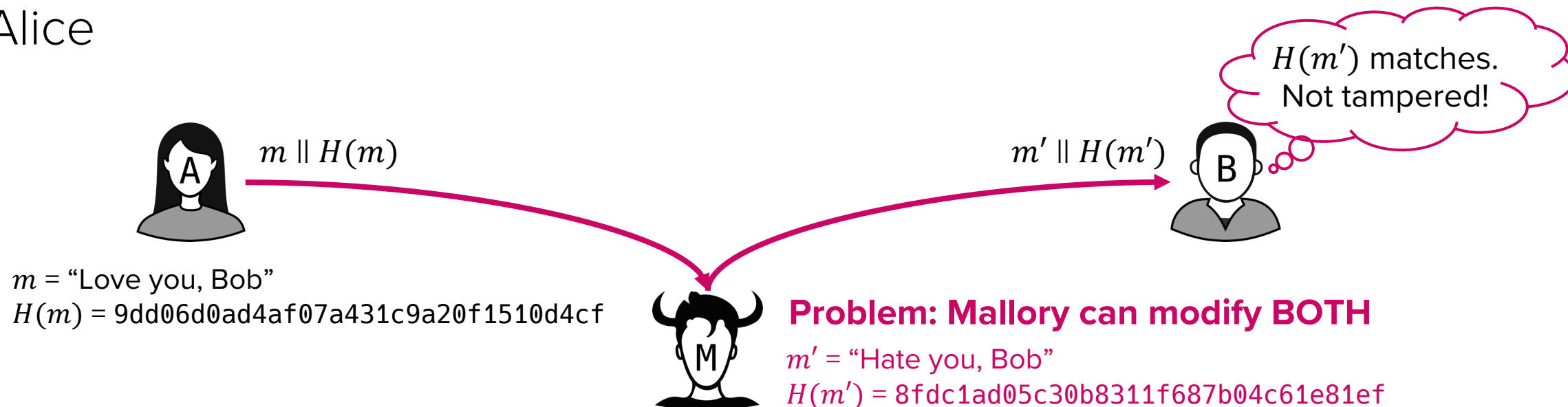
- Alice sends her message with its hash digest over the channel
- Bob receives the message and computes a hash of the message
- Bob verifies that the hash he computed matches the hash sent by Alice



Using hash functions for integrity

- Method

- Alice sends her message with its hash digest over the channel
- Bob receives the message and computes a hash of the message
- Bob verifies that the hash he computed matches the hash sent by Alice



Do hash functions provide integrity?

- Not necessarily
 - Microsoft website → Mallory can compromise Microsoft's web servers and modify $H(vscode)$
 - Communication → Mallory can modify m to m' and $H(m)$ to $H(m')$
- Main issue: Hash functions are **keyless** and deterministic
 - No secret key is used as input for hash functions, so any attacker can compute the hash of any value

Then, how can we utilize hash to design schemes that provide integrity?

Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
Confidentiality	<ul style="list-style-type: none">✓ One Time Pad (OTP)✓ Block ciphers (DES, AES)✓ Stream ciphers	<ul style="list-style-type: none">✓ DH secure key exchange✓ ElGamal encryption✓ RSA encryption
Integrity & Authentication	<ul style="list-style-type: none">• Message Authentication Code (MAC)	<ul style="list-style-type: none">• Digital signature

Additional Tool

- ✓ Hash functions

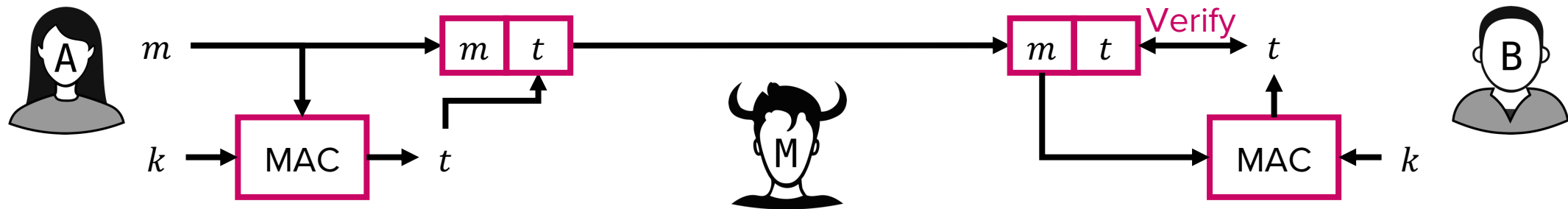
Message Authentication Code (MAC)

Goal: Providing integrity

- Reminder: We are in the symmetric-key setting
 - Alice and Bob share a secret key
 - Attacker does not know the key
- Idea: Attach some piece of information to verify that someone with the key is the sender of a message

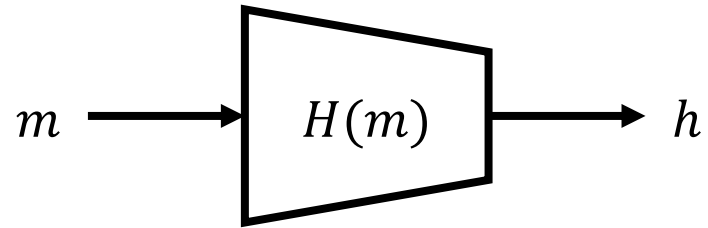
Message Authentication Code (MAC)

- Designed to provide both integrity and authenticity
- Setting
 - Alice sends message m and tag $t = MAC(k, m)$ where k : secret key
 - Bob recomputes $MAC(k, m)$ and verifies if the result matches t
 - If the MACs match, Bob is confident that m has not been altered

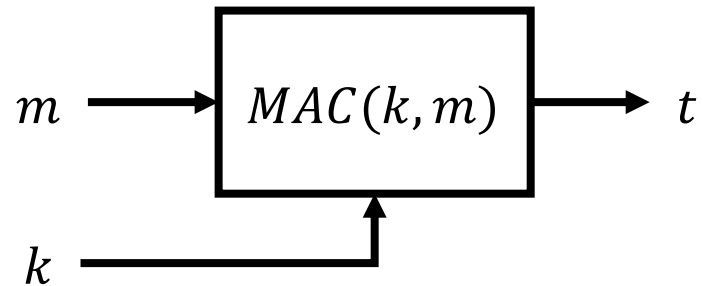


Hash function vs MAC

- Hash: Keyless



- MAC: Keyed

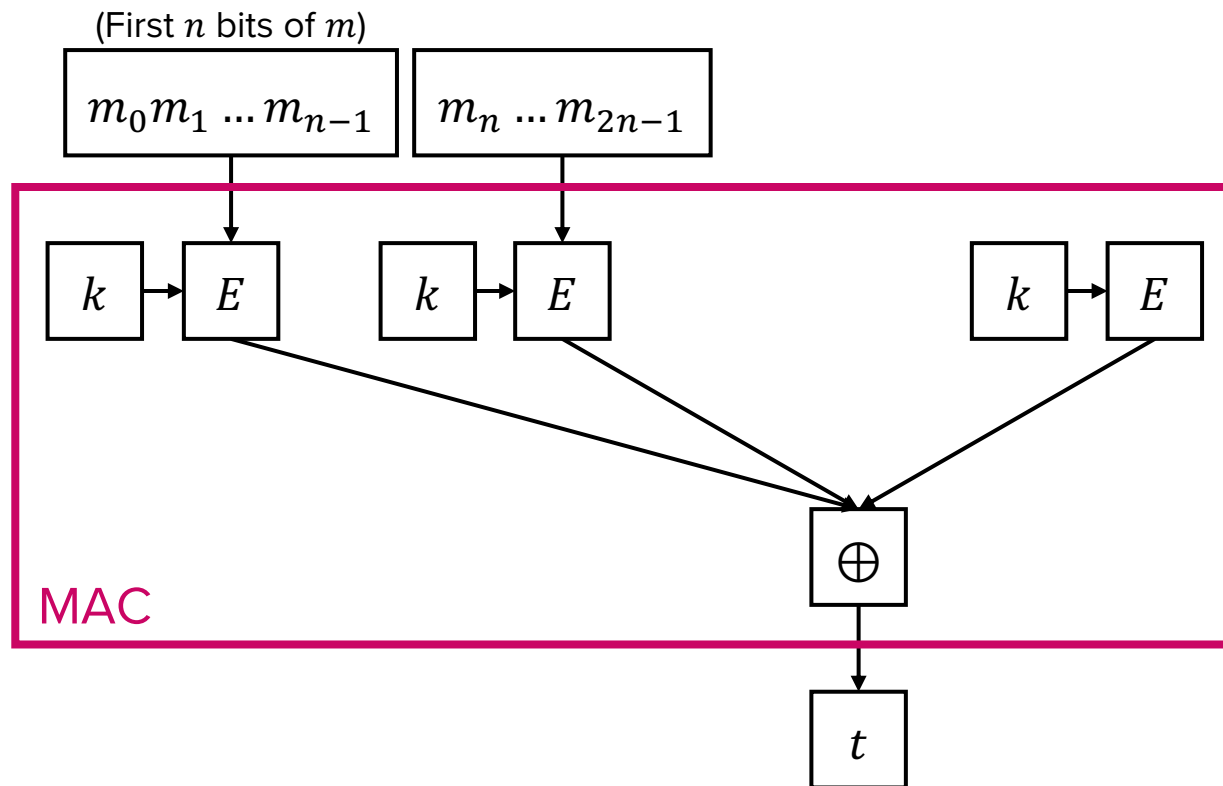


Evaluating the security of MAC

- “Unforgeability”: MAC is unforgeable under chosen msg m if
 - A polynomial time adversary can see some number of $\langle m, t \rangle$ pairs
 - Without knowing the key k ,
it is infeasible to find a message m and its MAC tag t
such that $t = \text{MAC}(k, m)$

Evaluating the security of MAC

- Example: Is block-cipher-based MAC secure?
 - E is a n -bit block cipher using key k



Is this MAC unforgeable?

- (0000 ... 01111 ... 1)
1. Adversary selects plaintext $0^n || 1^n$ and obtains $t = \text{MAC}(k, 0^n || 1^n)$
 2. Adversary found $m = 1^n || 0^n$ and its tag t such that $t = \text{MAC}(k, 1^n || 0^n)$

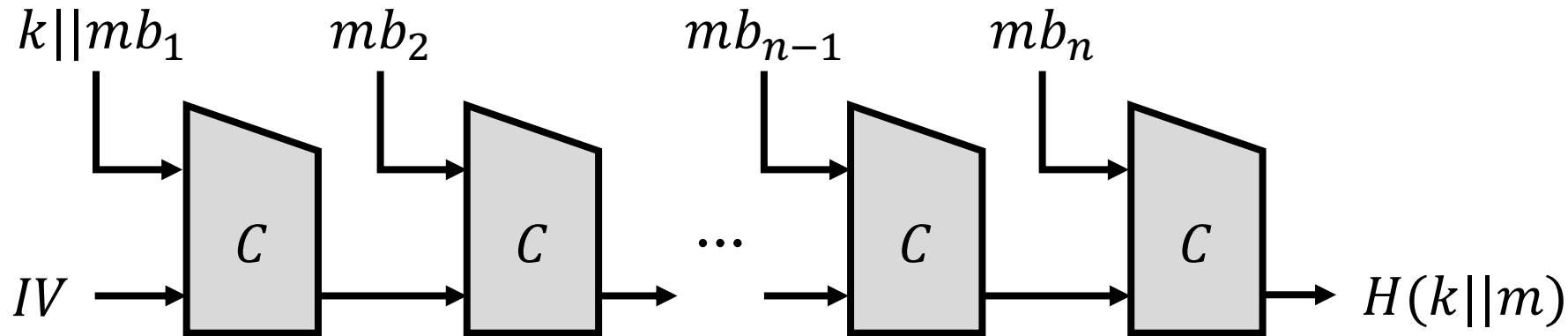
→ Not unforgeable (i.e., no integrity)

Constructing MAC using hash functions

- Secret prefix MAC:
 - $MAC_{sp} = H(k || m)$
- Secret suffix MAC:
 - $MAC_{ss} = H(m || k)$
- Nested MAC:
 - $NMAC = H(k_1 || H(k_2 || m))$
- Hash-based MAC:
 - $HMAC = H(k' \oplus opad || H(k' \oplus ipad || m))$

Constructing MAC using hash functions

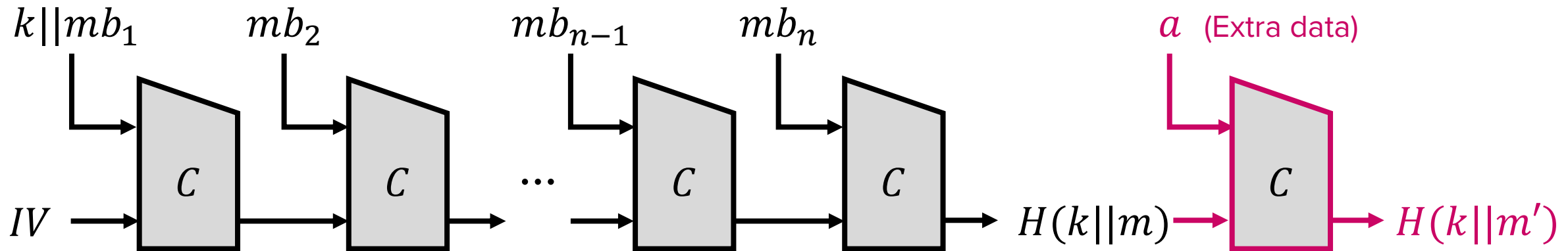
- Secret prefix MAC: $MAC_{sp} = H(k||m)$
 - Recall: Merkel-Damgård transform



Expectation: Mallory cannot compute $H(k||m)$ from m , as he does not know k

Constructing MAC using hash functions

- Secret prefix MAC: $MAC_{sp} = H(k||m)$
 - Recall: Merkle-Damgård transform

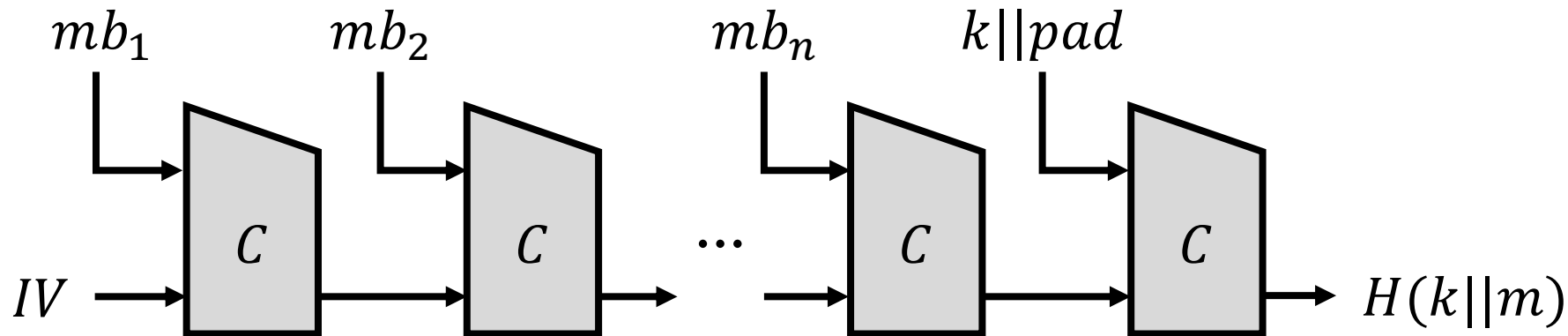


Reality: Vulnerable to length extension attack

- Given m and $H(k||m)$, Mallory can append a to m to obtain $m' = m||a$.
- Mallory can compute its MAC tag $H(k||m') = H(k||m||a)$ from the $H(k||m)$.
- Mallory successfully obtains $\langle m, t \rangle = \langle m||a, H(k||m||a) \rangle$ without knowing k

Constructing MAC using hash functions

- Secret suffix construction: $MAC_{ss} = H(m||k)$
 - There is no known attack for secret suffix construction
 - However, its unforgeability has not been proven



Constructing MAC using hash functions

- Nested MAC: $H(k_1 || H(k_2 || m))$
 - Prevents length extension attacks by hashing twice
 - It is proven that if two keys (k_1 and k_2) are different, NMAC is unforgeable
 - Proof omitted
 - Practical issues of NMAC
 - Need two different keys (weaker security)
 - Two keys need to be the same length as hash digest (constraint)

Constructing MAC using hash functions

- Hash-based MAC: $HMAC = H(k' \oplus opad || H(k' \oplus ipad || m))$
 - Improvement over NMAC
 - k' : n -bit version of k where n is the length of hash digest
 - If k is smaller than n bits, $k' = k || 0^{n-|k|}$, i.e., pad k with 0's to make it n bits
 - Otherwise, $k' = H(k)$, i.e., hash k to make it n bits
 - Two different keys can be derived from k'
 - Outer pad (*opad*): 0x5c repeated until the length becomes n bits
 - Inner pad (*ipad*): 0x36 repeated until the length becomes n bits
 - Two rounds of hashing with two keys

Evaluating the security of HMAC

- Hash-based MAC (HMAC):
 - $H(k' \oplus opad || H(k' \oplus ipad || m))$
 - HMAC is unforgeable under chosen message attack
 - A polynomial attacker cannot create m and valid $t = HMAC(k, m)$ without knowing the secret key k (proof omitted)
- HMAC is one of the most widely standardized and used cryptographic constructs

Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
Confidentiality	<ul style="list-style-type: none">✓ One Time Pad (OTP)✓ Block ciphers (DES, AES)✓ Stream ciphers	<ul style="list-style-type: none">✓ DH secure key exchange✓ ElGamal encryption✓ RSA encryption
Integrity & Authentication	<ul style="list-style-type: none">✓ Message Authentication Code (MAC)	<ul style="list-style-type: none">• Digital signature

Can we achieve both
at the same time?

Additional Tool

✓ Hash functions

Authenticated Encryption

Confidentiality and integrity/authenticity goals

- Encryption schemes provide confidentiality, but not integrity
 - MACs provide integrity/authenticity, but not confidentiality
- Can we achieve both at the same time?

Authenticated encryption (AE)

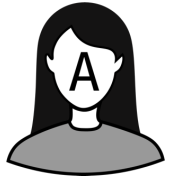
- Definition
 - A scheme that simultaneously guarantees confidentiality and integrity of a message
- Existing building blocks for AE:
 - $E(k_1, m)$ and $D(k_1, m)$
 - e.g., AES
 - $MAC(k_2, m)$
 - e.g., HMAC

Building AE from existing primitives

1. Encrypt-and-MAC

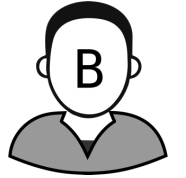
Secure?

$$c \leftarrow c' || t$$



$m,$
 $k = k_1 || k_2$

```
Algorithm  $AE(k, m)$ :  
   $c' \leftarrow E(k_1, m)$   
   $t \leftarrow MAC(k_2, m)$   
   $c \leftarrow c' || t$   
  Return  $c$ 
```



$k = k_1 || k_2$

```
Algorithm  $AD(k, c)$ :  
   $c' || t \leftarrow c$   
   $m \leftarrow D(k_1, c')$   
  If  $(t = MAC(k_2, m))$  Return  $m$   
  Else Return  $NULL$ 
```

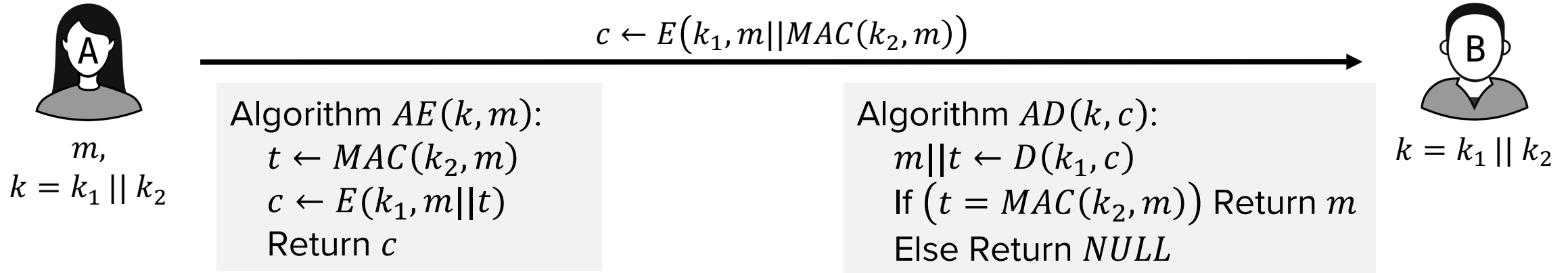
No. Vulnerable to chosen-plaintext attacks ☹

t is exposed as is. Attacker can observe t
to check the equality of messages

Building AE from existing primitives

2. MAC-then-Encrypt

Secure?



No longer vulnerable to chosen-plaintext attacks 😊

Integrity (unforgeability) is not guaranteed for some encryption schemes even if a secure MAC is used ☹️

→ Attackers can forge messages that are accepted by AD :

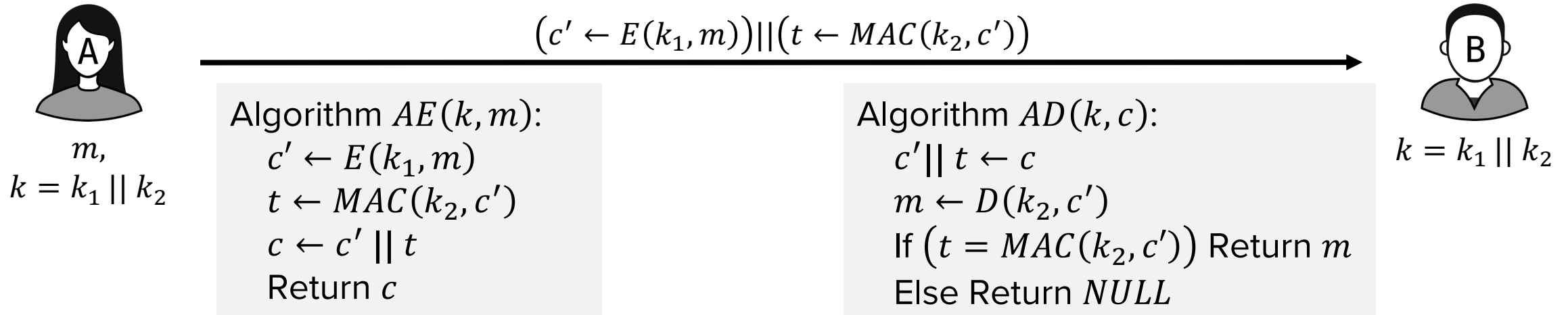
$$\text{e.g., } E'(k, m) = E(k, m) || 0 = c'$$

$$D'(k, c') = D'(k, c || 0) = D(k, c)$$

Building AE from existing primitives

3. Encrypt-then-MAC

Secure?



Not vulnerable to chosen-plaintext attacks 😊

Unforgeability is also guaranteed 😊
(proof omitted)

Can check MAC first before decrypting (efficiency!)

Cryptography roadmap

<div>Scheme</div> <div>Goal</div>	Symmetric Key	Asymmetric Key
Confidentiality	<ul style="list-style-type: none"> ✓ One Time Pad (OTP) ✓ Block ciphers (DES, AES) ✓ Stream ciphers 	<ul style="list-style-type: none"> ✓ DH secure key exchange ✓ ElGamal encryption ✓ RSA encryption
Integrity & Authentication	<ul style="list-style-type: none"> ✓ Message Authentication Code (MAC) 	<ul style="list-style-type: none"> • Digital signature
CIA at the same time	<ul style="list-style-type: none"> ✓ Authenticated encryption 	

Additional Tool

- ✓ Hash functions

Questions?