



# Message Authentication Code

Gianluca Dini  
Dept. of Ingegneria dell'Informazione  
University of Pisa  
Email: [gianluca.dini@unipi.it](mailto:gianluca.dini@unipi.it)  
Version: 07/04/2025

1

Message Authentication Codes (MACs)

# PRINCIPLES OF MACS

apr. '25

Hash functions

2

2

## Message Authentication Code

- Synonyms
  - Cryptographic checksum
  - Keyed hash function
- Similarly to digital signatures, MACs provide message authentication and integrity
- Unlike digital signatures, MACs are symmetric schemes and do not provide nonrepudiation
- MACs are much faster than digital signatures

apr. '25

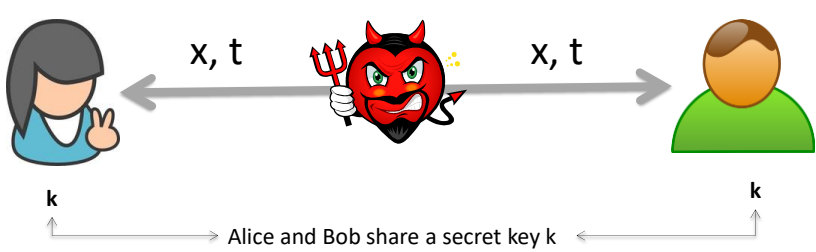
Hash functions

3

3

You cannot use them if mutual trust doesn't hold

## Communication model



MAC Generation  $S: \{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^m \rightarrow$

MAC Verification  $V: \{0,1\}^* \times \{0,1\}^k \times \{0,1\}^m \rightarrow \text{Boolean} \rightarrow$

MAC generation

$$t = S(k, x)$$

Alice and Bob want to be assured that any manipulations of a message  $m$  in transit are detected

MAC verification

$$t' = \text{MAC}(k, x)$$
$$\text{return } (t == t')$$

Recompute the key and compare with the one received

apr. '25

Hash functions

2

## Message Authentication Code (MAC)

- A MAC is defined by  $(\text{Gen}, S, V)$ 
  - $\text{Gen}$  takes as input  $1^n$  and outputs a key  $k$
  - $S$  takes as input a key  $k$  and a message  $x \in \{0, 1\}^*$  and outputs a tag  $t$ , s.t.  $t = S_k(x)$
  - $V$  takes as input a key  $k$ , a message  $x$  and a tag  $t$  and returns true or false
- Consistency property
  - For all key  $k$  and message  $x$ ,  $V_k(x, S_k(x)) = \text{true}$

apr. '25

Hash functions

5

5

## Properties of MACs ( $\rightarrow$ )

- Cryptographic checksum
  - A MAC generates a cryptographically secure authentication tag for a given message.
- Symmetric
  - MACs are based on secret symmetric keys. The signing and verifying parties must share a secret key.
- Arbitrary message size
  - MACs accept messages of arbitrary length.
- Fixed output length
  - MACs generate fixed-size authentication tags.

apr. '25

Hash functions

6

6

# Properties of MACs

- Message integrity
  - MACs provide message integrity: Any manipulations of a message during transit will be detected by the receiver.
- Message authentication
  - The receiving party is assured of the origin of the message.
- No nonrepudiation
  - Since MACs are based on symmetric <sup>keys</sup> principles, they do not provide nonrepudiation

7

# Security

- Threat model
  - Adaptive chosen-message attack → No assume they can do this
  - Assume the attacker can induce the sender to authenticate messages of the attacker's choice
- Security goal
  - Existential unforgeability
  - Attacker should be unable to forge a valid tag on any message not authenticated by the sender

8

# Security

- **Computation-resistance (chosen message attack)**
  - For each key  $k$ , given zero or more  $(x_i, t_i)$  pairs, where  $t_i = S_k(x_i)$ , it is computationally infeasible to compute  $(x, t)$ , s.t.  $t = S_k(x)$ , for any new input  $x \neq x_i$  (including possible  $t = t_i$  for some  $i$ )

For adversary it must also be difficult to find a collision (new  $x$  that maps to  $t_i$ )

9

# Replay

- Mac **does not prevent replay**
  - No stateless mechanism can
- Replay attack **can be a significant real-world concern**
- Need to **protect** against replay **at a higher layer**

10

## Types of forgery

- **Selective forgery** – Attacks whereby an adversary is able to produce a new text-MAC pair for a text of his choice (or perhaps partially under his control)
  - Note that here the selected value is the text for which a MAC is forged, whereas in a chosen-text attack the chosen value is the text of a text-MAC pair used for analytical purposes (e.g., to forge a MAC on a distinct text).
- **Existential forgery** – Attacks whereby an adversary is able to produce a new text-MAC pair, but with no control over the value of that text.

11

In the context of MACs (Message Authentication Codes), when it talks about a **chosen-text attack**, it's referring to a situation where an attacker can pick specific texts (messages) and see what their corresponding MACs are. The idea is: "Hey, I give you a message, and you give me the MAC for it." The attacker collects this info and tries to use it to understand the MAC algorithm or forge a MAC for a different message later.

## Implications of a secure MAC

- **FACT 1** – Computation resistance <sup>IMPLIES</sup> key non-recovery (but not vice versa)
  - It must be computationally infeasible to compute  $k$  from  $(x_i, t_i)$ s
  - However, it may be possible to forge a tag without knowing the key

↓  
In some ways you could still compute a tag w/o Key.

12

## Implications of a secure MAC

- **FACT 2** - Attacker cannot produce a valid tag for any new message
  - Given  $(x, t)$ , attacker cannot even produce  $(x, t')$  – a collision – for  $t' \neq t$ 

Must not be able to generate even a collision.

13

## Implications of a secure MAC

- **FACT 3** - For an adversary not knowing  $k$ 
  - $S$  must be 2nd-preimage and collision resistant;
  - $S$  must be preimage resistant w.r.t. a chosen-text <sup>\*</sup>attack; ?
- **FACT 4** - Secure MAC definition says nothing about preimage and 2nd-preimage for parties knowing  $k$ 
  - Mutual trust model
- Neither Alice/Bob can generate a collision, that is not a requirement.

<sup>\*</sup> Adv. must not be able to find a preimage even given a number of input collected as result.

14

# How to use MACs in practice

- In combination with encryption
  - **x**: plaintext; **t**: tag; **y**: ciphertext; **z**: transmitted message;  
**ek**: encryption key; **ak**: MAC key (authentication key)
  - Option 1 (SSL)
    - $t = S_{ak}(x); y = E_{ek}(x || t), z = y$
  - Option 2 (IPsec) *hash on ciphertext*
    - $y = E_{ek}(x); t = S_{ak}(y); z = y || t$
  - Option 3 (SSH)
    - $y = E_{ek}(x); t = S_{ak}(x); z = y || t$

How can we generate keys?  
 $K = \text{shared secret}$   
 $ek = h(K || 0)$   
 $ak = h(K || 1)$

## Reasons for 2 Keys

- Good practice to use 2 keys for 2 purposes
- MAC can be made with *hash functions* that are made with ciphers, if you use the same key, ciphers could interfere with encryption.

apr. '25

15

# Other uses

- One-time password
  - Based on time-synchronization
  - Based on challenge-response

apr. '25

Hash functions

16



Message Authentication Codes (MACs)

HOW TO BUILD A MAC

apr. '25

Hash functions

17

17

How to build a MAC

• From Block Ciphers (more in general from PRF)

– CBC-MAC

– NMAC

– PMAC

• From a hash functions

– HMAC

apr. '25

Hash functions

18

18

# HMAC

## How to build a MAC from a hash function

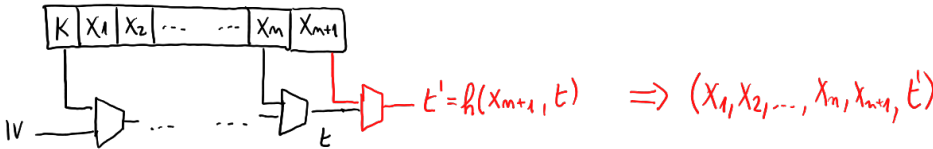
- Insecure constructions
  - Secret prefix scheme
    - $S(k, x) = H(k || x)$ , H hash function
  - Secret suffix scheme
    - $S(k, x) = H(x || k)$ , H hash function
  - Forgery is possible in both cases
  - HMAC construction is necessary

19

# Insecurity of prefix scheme

- Let  $x = (x_1, x_2, x_3, \dots, x_n)$
  - Let  $t = S(k, x) = H(k || x_1, x_2, \dots, x_n)$
  - Existential forgery attack
    - **Objective:** construct  $t'$  of  $x' = x_1, x_2, \dots, x_n, x_{n+1}$  without knowing  $k$  ( $x_{n+1}$ : additional block)
    - **Assumption:** H follows the Merkle-Damgard scheme
    - **The attack:**  $t' = h(x_{n+1}, t)$  with  $h$  compression function
    - Forging  $(x', t')$  only needs the previous hash output  $t$  (but not  $k$ )
- we take identical message and just add identical block*
- Output of hash function is output of the last compression function.  $t$  is the value of chaining variable.*

20



REMEMBER DIFFERENCE!

## Insecurity of the suffix scheme

- Let  $t = S(k, x) = H(x || k)$
- Existential forgery attack
  - Objective: Construct  $t'$  of a  $x'$  without knowing the key  $k$
  - Assumption:  $H$  follows the Merkle-Damgard scheme
  - The attack
    - Assume the adversary can find a collision  $H(x) = H(x')$
    - Then,  $t = h(H(x), k) = h(H(x'), k)$ , thus  $t' = t$ , where  $h$  compression function

You can show how to forge on the other way. Here assumption is stronger but still.

If  $x$  is a block and we compute  $h(H(x), k)$ , then the result will be the same:  $x$  analyzed, the  $k$ .



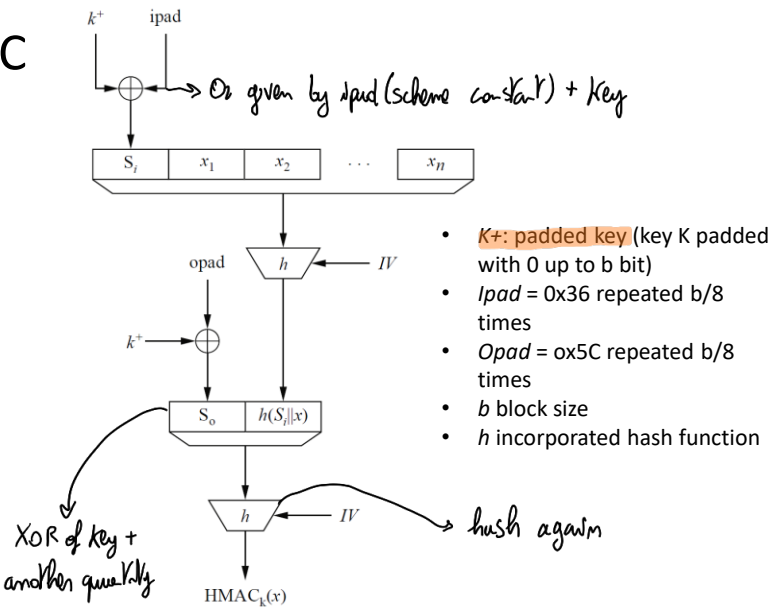
apr. '25

Hash functions

21

21

## HMAC



apr. '25

Hash functions

22

22

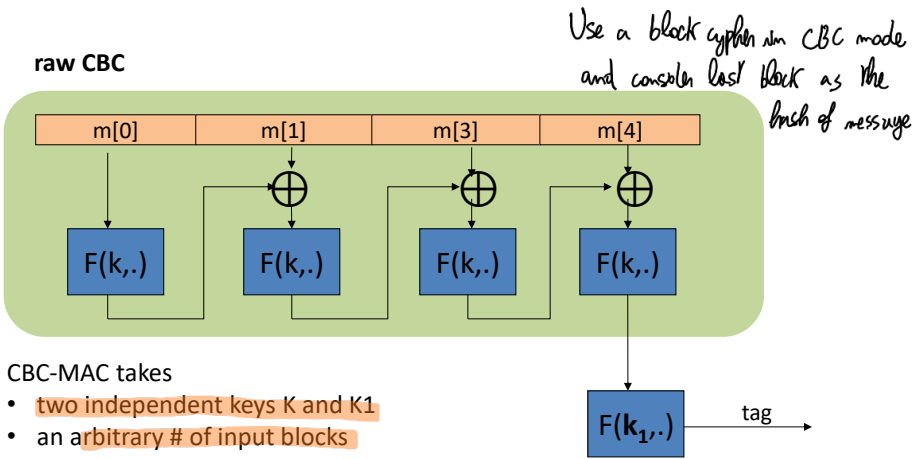
$HMAC_K(x) = H(K \oplus opad || H(K \oplus ipad || x))$   
Outermost hash adds little performance penalty, because the hash that makes the dirty work is the inner hash so we have eventually a smaller output. Outermost always works on the same amount of bits.

# HMAC

- Computational efficiency
  - The message is hashed in the inner hash
  - The outer hash only hashes two blocks
- Security
  - There exists a proof of security in HMAC
  - THM - If an attacker can break HMAC then (s)he can break H

23

# CBC-MAC: construction



24

It is inherently sequential

## CBC-MAC: security

- Normally CBC-MAC does not use the last encryption  
→ rawCBC
- rawCBC is insecure
  - Proof (An existential forgery attack)
    1. The adversary chooses a one-block message  $x$
    2. The adversary requests  $t = \text{rawCBC}(k, x)$  where  $t = E(k, x)$
    3. The adversary outputs  $t' = t$  as MAC forgery of the two-block message  $x' = x, (t \oplus x)$

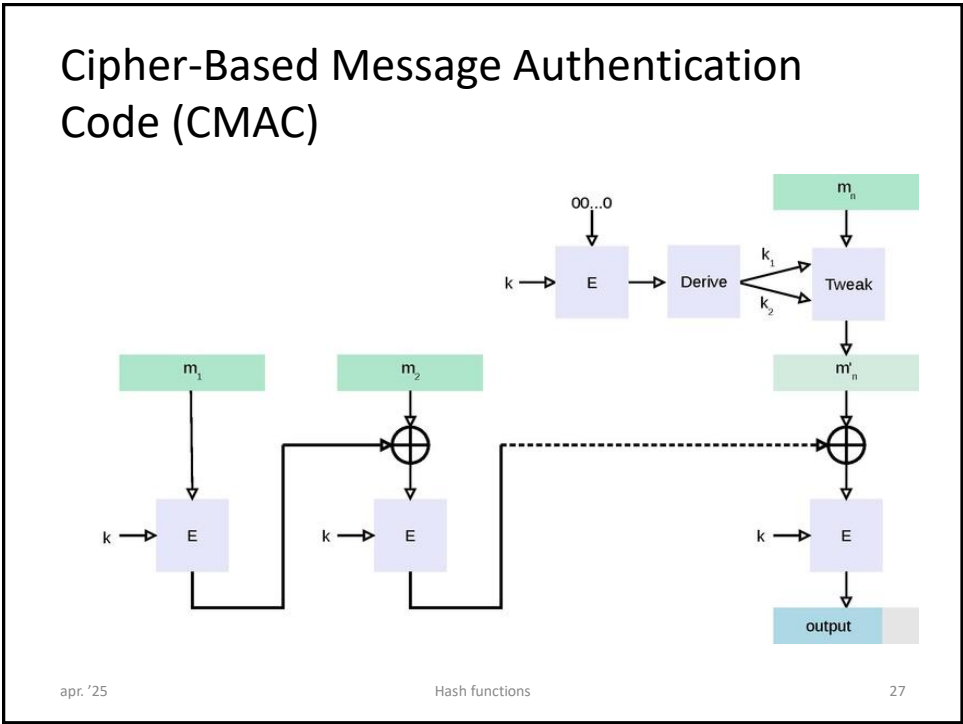
25

## CBC-MAC: security

- Proof (for brevity  $\text{rawCBC} = H$ )
  - Let  $t' = H(k, x') = H(k, (x, (t \oplus x))) =$   
 $E(k, (E(k, x) \oplus (t \oplus x))) =$   
 $E(k, t \oplus (t \oplus x)) =$   
 $E(k, x) = t$ , where  $E$  is the cipher

Q.E.D

26



27

### CMAC

- CMAC overcomes the problems of CBC-MAC
- CMAC uses three keys  $K, K_1, K_2$ 
  - $K$  is  $k$  bit,  $K_1$  and  $K_2$  are  $n$  bit
  - $K_1$  and  $K_2$  can be derived from  $K$  (NIST 800-38B)
    - $L = E_K(0^n)$
    - $K_1 = L \cdot x$  (if  $\text{len}(x)$  is an integer multiple of  $n$ )
    - $K_2 = L \cdot x^2$  (if  $\text{len}(x)$  is not an integer multiple of  $n$ )
    - Polynomials  $x, x^2 \in GF(2^n)$ , multiplication  $\cdot$  in  $GF(2^n)$
- $E = \text{AES}, 3\text{DES}$

apr. '25 Hash functions 28

28

# CBC-MAC & CMAC drawbacks

- CBC-MAC and CMAC are not suitable for high-speed implementations because they are neither pipelineable nor parallelizable

apr. '25

Hash functions

29

29

Message Authentication Code (MAC)

## PADDING

Padding should be done in a special way

NOTE: hash fct. w/o Keys must be used with additional methods to prove authenticity and integrity

30

## MAC Padding

- Pad by zeroes is insecure
  - $\text{pad}(m)$  and  $\text{pad}(m \parallel 0)$  have the same MAC
- Padding must be an invertible function
  - $m_0 \neq m_1 \Rightarrow \text{pad}(m_0) \neq \text{pad}(m_1)$
- Standard padding (ISO)
  - Append “100...00” as needed
    - Scan right to left
    - “1” determines the beginning of the pad
  - Add a dummy block if necessary
    - When the message is a multiple of the block
    - The dummy block is necessary or existential forgery arises

apr. '25

Hash functions

31

31

## Padding by 0es is a bad idea

- Proof
    - Let  $x = x_1, x_2, x_3$  where  $x_3$  is shorter than a block
    - Let's pad  $x_3$  as follows  $x_3^* \leftarrow x_3 \parallel 000$  (for example)
    - Let  $t$  be the tag outputted.
    - Consider now a message  $x' = x \parallel 0$ .
      - $x'$  would be composed of three blocks  $x'_1 = x_1, x'_2 = x_2$ , and  $x'_3 = x_3 \parallel 0$ .
      - $x'_3$  needs padding and becomes  $x_3'^* = x_3 \parallel 0 \parallel 00 = x_3 \parallel 000$ .
      - So,  $x$  and  $x'$  after padding are equal and thus have the same tag.
- QED

apr. '25

Hash functions

32

32



## On dummy block

- Without dummy block, existential forgery arises
- Proof
  - Let  $x = x_1, x_2$  which needs padding
  - Build  $x^* = x_1, x_2 || 100$ , where  $x^*$  is the padded message
  - Consider now  $x' = x_1, x_2 || 100$ 
    - Since  $x'$  is a multiple of the block we don't pad it
  - It follows that  $x' = x^*$  and thus  $x$  and  $x'$  have the same tag

QED

apr. '25

Hash functions

33

33

## TIMING ATTACK

apr. '25

Hash functions

34

34

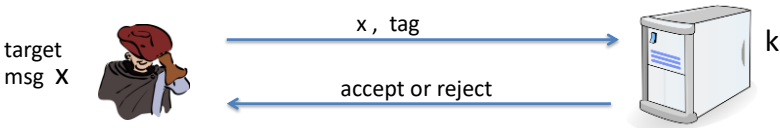
# Timing Attack

- Example: Keyczar crypto library (Python) [simplified]

```
def Verify(key, msg, tag):  
    return HMAC(key, msg) == tag
```
- The problem: operator '==' is implemented as a byte-by-byte comparison
  - It returns false when first inequality found

35

# Timing attack



- Timing attack: to compute tag for target message do:
- Step 1: Query server with random tag
  - Step 2: Loop over all possible first bytes of tag and query server.  
Stop when verification takes a little longer than in step 1
  - Step 3: Repeat for all tag bytes until valid tag found

3	47	*	*	*	*
---	----	---	---	---	---

36

## Defense – solution #1 🤔

- Make string comparator always take same time
- Solution 1:

```
return false if tag has wrong length
result = 0
for x, y in zip( HMAC(key,msg) , tag):
    result |= ord(x) ^ ord(y)
return result == 0
```
- Can be difficult to ensure due to optimizing compiler

apr. '25

Hash functions

37

37

## Defense – solution #2 😊

- Make string comparator always take same time
- Solution 2

```
def Verify(key, msg, tag):
    mac = HMAC(key, msg)
    return HMAC(key, mac) == HMAC(key, tag)
```
- Attacker doesn't know values being compared

apr. '25

Hash functions

38

38

