ECE458

Hash Functions and Message Authentication Codes

Dan Boneh

(Mods by Vijay Ganesh)

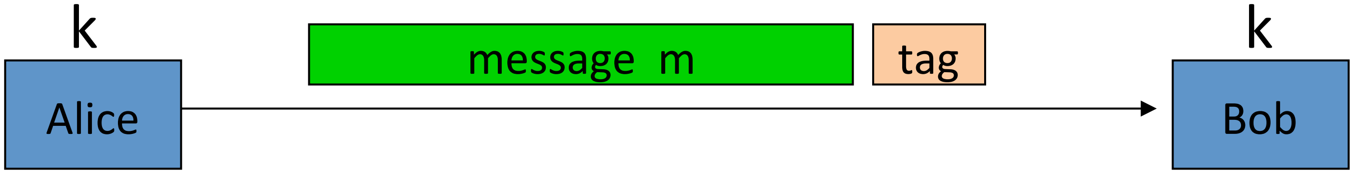
Message Integrity

Goal: **integrity**, no confidentiality.

Examples:

- Protecting public binaries on disk.
- Protecting banner ads on web pages.

Message integrity: MACs



Generate tag:

$\text{tag} \leftarrow S(k, m)$

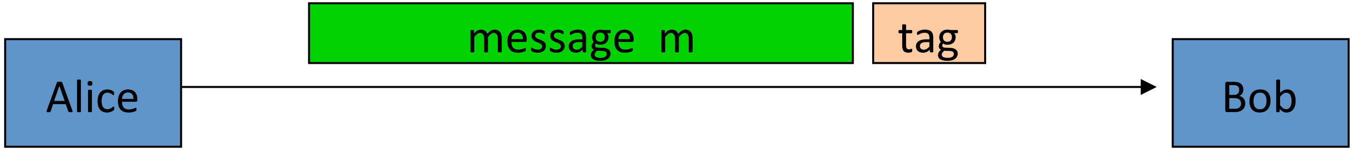
Verify tag:

$V(k, m, \text{tag}) \overset{?}{=}$ `yes'

Def: **MAC** I = (S,V) defined over (K,M,T) is a pair of algs:

- S(k,m) outputs t in T
- V(k,m,t) outputs `yes' or `no'

Integrity requires a secret key



Generate tag:

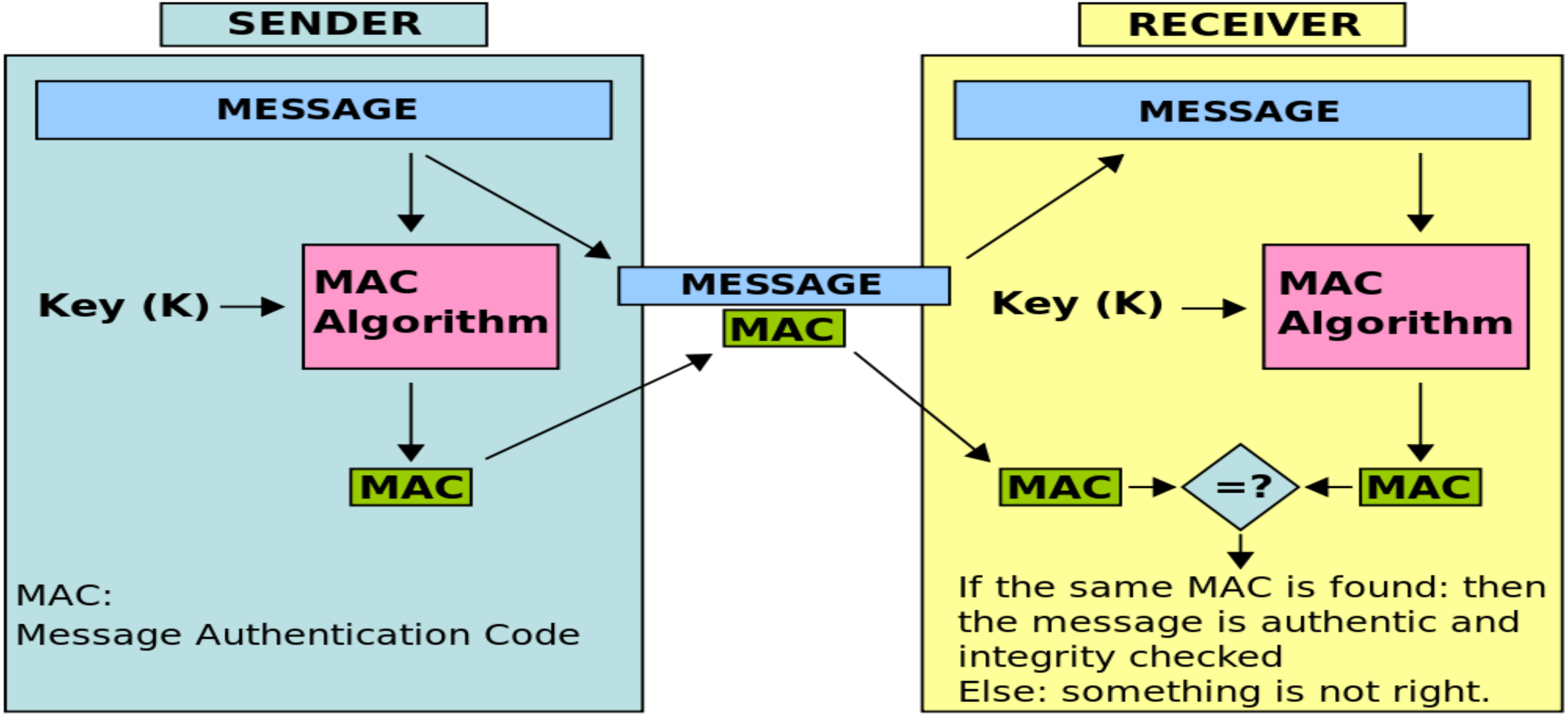
$\text{tag} \leftarrow \text{CRC}(m)$

Verify tag:

$V(m, \text{tag}) \overset{?}{=}$ `yes'

- Attacker can easily modify message m and re-compute CRC.
- CRC designed to detect **random**, not malicious errors.

Recalling MACs: Length Extension Attacks



MAC:
Message Authentication Code

If the same MAC is found: then the message is authentic and integrity checked
Else: something is not right.

Source: Wikipedia

Secure MACs

Attacker's power: **chosen message attack**

- for $m_1,m_2,\ldots,m_q$ attacker is given $t_i \leftarrow S(k,m_i)$

Attacker's goal: **existential forgery**

- produce some **new** valid message/tag pair (m,t).

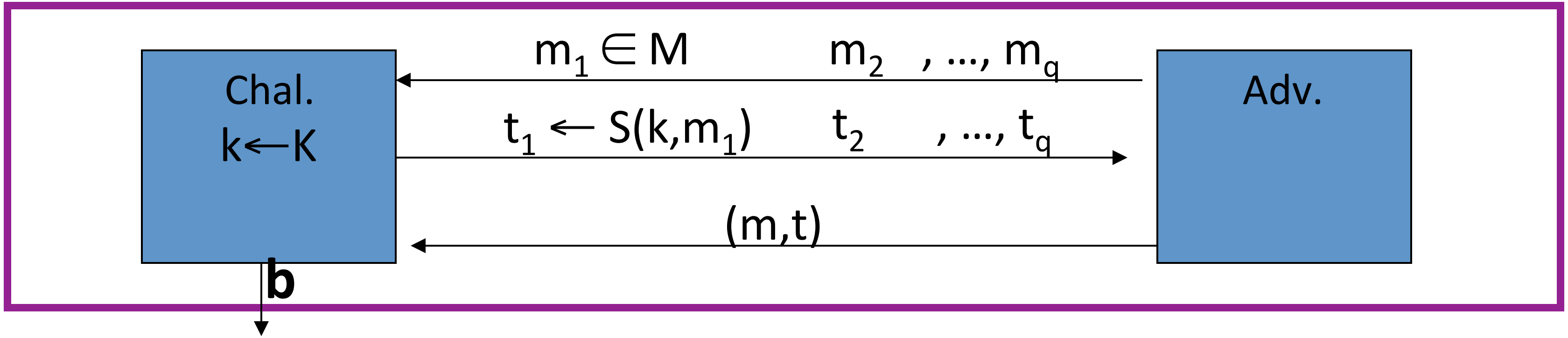
$$(m,t) \notin \{ (m_1,t_1), \ldots, (m_q,t_q) \}$$

⇒ attacker cannot produce a valid tag for a new message

⇒ given (m,t) attacker cannot even produce (m,t') for $t' \neq t$

Secure MACs

- For a MAC I=(S,V) and adv. A define a MAC game as:



$$\begin{cases} \mathbf{b}=1 & \text{if } V(k,m,t) = \text{`yes'} \text{ and } (m,t) \notin \{ (m_1,t_1), \dots, (m_q,t_q) \} \\ \mathbf{b}=0 & \text{otherwise} \end{cases}$$

Def: I=(S,V) is a **secure MAC** if for all "efficient" A:

$\text{Adv}_{\text{MAC}}[A,I]$ = Pr[Chal. outputs 1] is "negligible."

Properties of Secure MACs

1. The tag generated by a MAC should be sufficiently long. Otherwise the attacker can simply guess the tag in very few attempts.

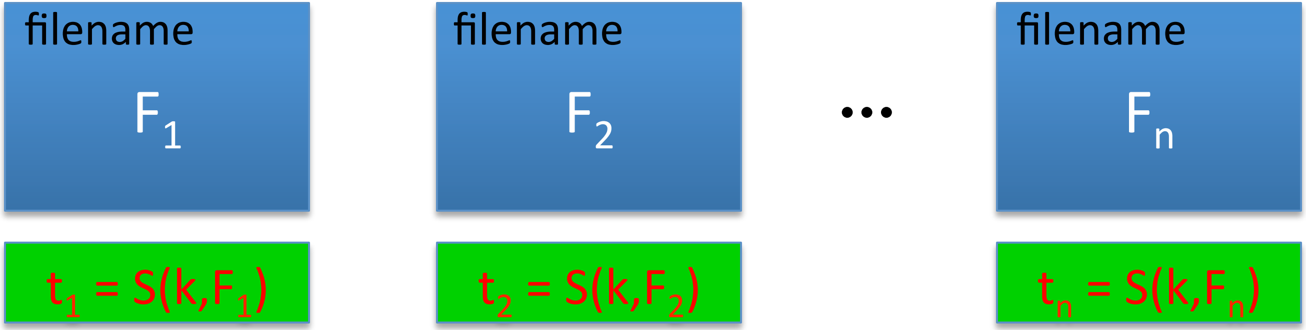
2. It should NOT be the case that the attacker can easily find a pair of messages find $m_0 \neq m_1$ such that

   $S(k, m_0) = S(k, m_1)$ for ½ of the keys k in K

3. One way to achieve property 2 above is through the use of collision-resistant hash functions (more on this later)

Example: protecting system files

Suppose at install time the system computes:



k derived from
user's password

Later a virus infects system and modifies system files

User reboots into clean OS (say, USB stick) and supplies his password

- Then: secure MAC ⇒ all modified files will be detected

How can we Construct Secure MACs?

MACs can be constructed out of pseudo-random functions (PRFs). E.g.,

PRFs

- **ECBC-MAC, CMAC** : commonly used with AES (e.g. 802.11i)
- **NMAC** : basis of HMAC (this segment)
- **PMAC**: a parallel MAC

randomized MAC

- **Carter-Wegman MAC**: built from a fast one-time MAC

Or they can be constructed out of collision-resistant hash function

Recap: Collision Resistance

Let H: M $\longrightarrow$ T be a hash function ($|M| >> |T|$)

A **collision** for H is a pair $m_0 , m_1 \in M$ such that:

$$H(m_0) = H(m_1) \quad \text{and} \quad m_0 \neq m_1$$

A function H is **collision resistant** if for all (explicit) "eff" algs. A:

$\mathrm{Adv}_{CR}[A,H]$ = Pr[A outputs collision for H]

is "negligible".

Example: SHA-256 (outputs 256 bits)

An Insecure MAC using Hash Functions

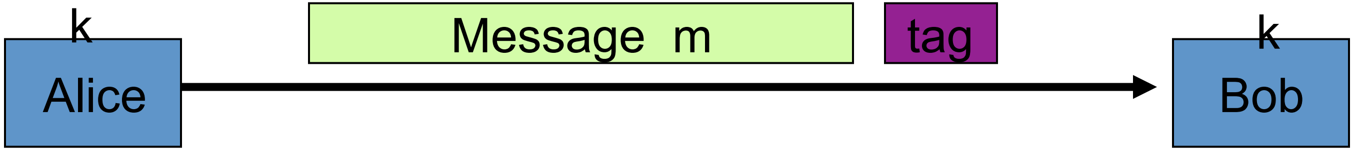
- How about we define a MAC as a simple application of hash functions? Will it be secure?

$$S(K,m) = \text{Hash}(K \,||\, m)$$

- This won't work because of length extension attack against hash functions.
- If you care about secure MACs, never use the above method!

Recalling MACs:
Hash Length Extension Attacks

- Goal: message integrity and Authenticity.
- No confidentiality.
  - ex: Protecting public binaries on disk.



k

Alice

Message m

tag

k

Bob

Generate tag:
tag ← S(k, m)

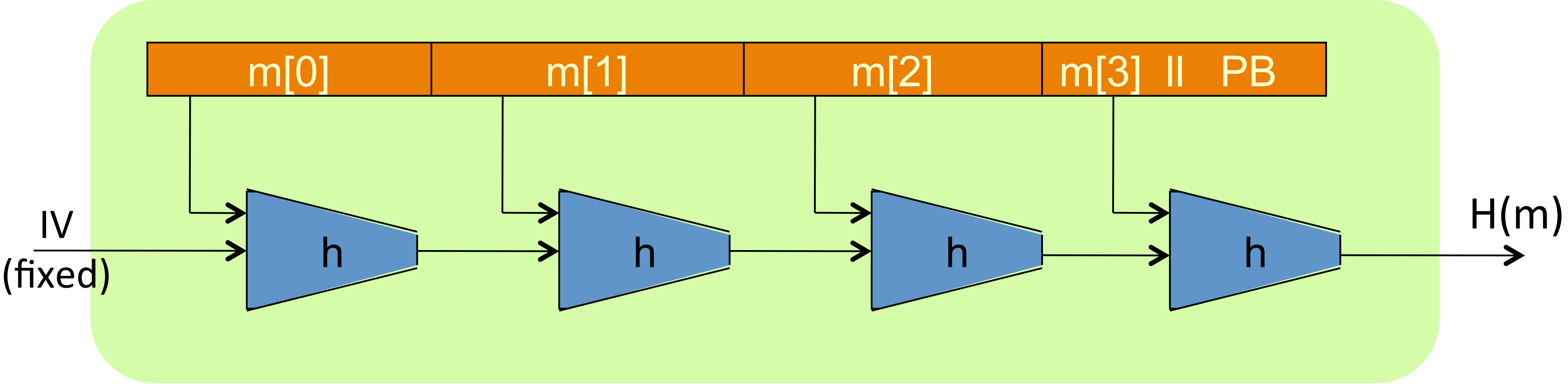
Verify tag:
V(k, m, tag) $\overset{?}{=}$ \`yes'

note: non-keyed checksum (CRC) is an insecure MAC !!

Length Extension Attacks Explained

- Alice sends "data" and "signature" (the MAC) to Bob. Recall that signature = Hash(secret || data || padding). Padding has a standard format that includes the length of "secret || data"
- Attacker intercepts "data" and "signature" (aka tag)
- Attacker's goal is to append stuff to "data" and appropriately modify signature
- The attacker sends the new "data || attacker extension" and the appropriate "signature" to Bob
- When Bob receives "data || attacker extension" verifies against the new signature, it matches. Attacker doesn't need to know the secret to launch attack. He only needs to know the length of the secret used.

Merkle-Damgard iterated construction and length-extension attack



Property: The output digest "remembers" state. I.e., if you simply appended bits to the message m as m || b, and performed another appropriate round of h in the iteration above, then the output will be H(m||b).

Length Extension Attacks Explained

- Fact:
  - When calculating *H*(*secret* || *data*), the string (*secret* || *data*) is padded with a '1' bit and some number of '0' bits, followed by the length of the string

- Fact:
  - The MD construction operates on fixed-sized blocks, and saves the output for the subsequent iteration. I.e., the digest “captures all of the input data || secret || padding”

- Fact:
  - Attacker knows the data, because Alice sent it along with the signature (MAC)

- We assume attacker knows the length of the secret

Length Extension Attacks Explained

- The Attack:
- Step 1: Compute the padding New-pad for “secret || data || pad (secret || data) || attacker extension”
  - Note that attacker already knows the length of data and length of his own “attacker extension”
  - All he needs is the length of secret to compute New-pad
- Step 2: Attacker initializes the hash function H with “signature” (i.e., uses signature as IV) and computes the H(attacker extension || New-pad)
- Step 3: He has a new verifiable signature for New-data, namely, Hash(“secret || data || pad(secret + data) || attacker extension || New-pad”)
- Step 4: Send the pair <New-data, New-signature> to Bob

Secure MAC Construction: HMAC (Hash-MAC)

Most widely used MAC on the Internet.

H: hash function.

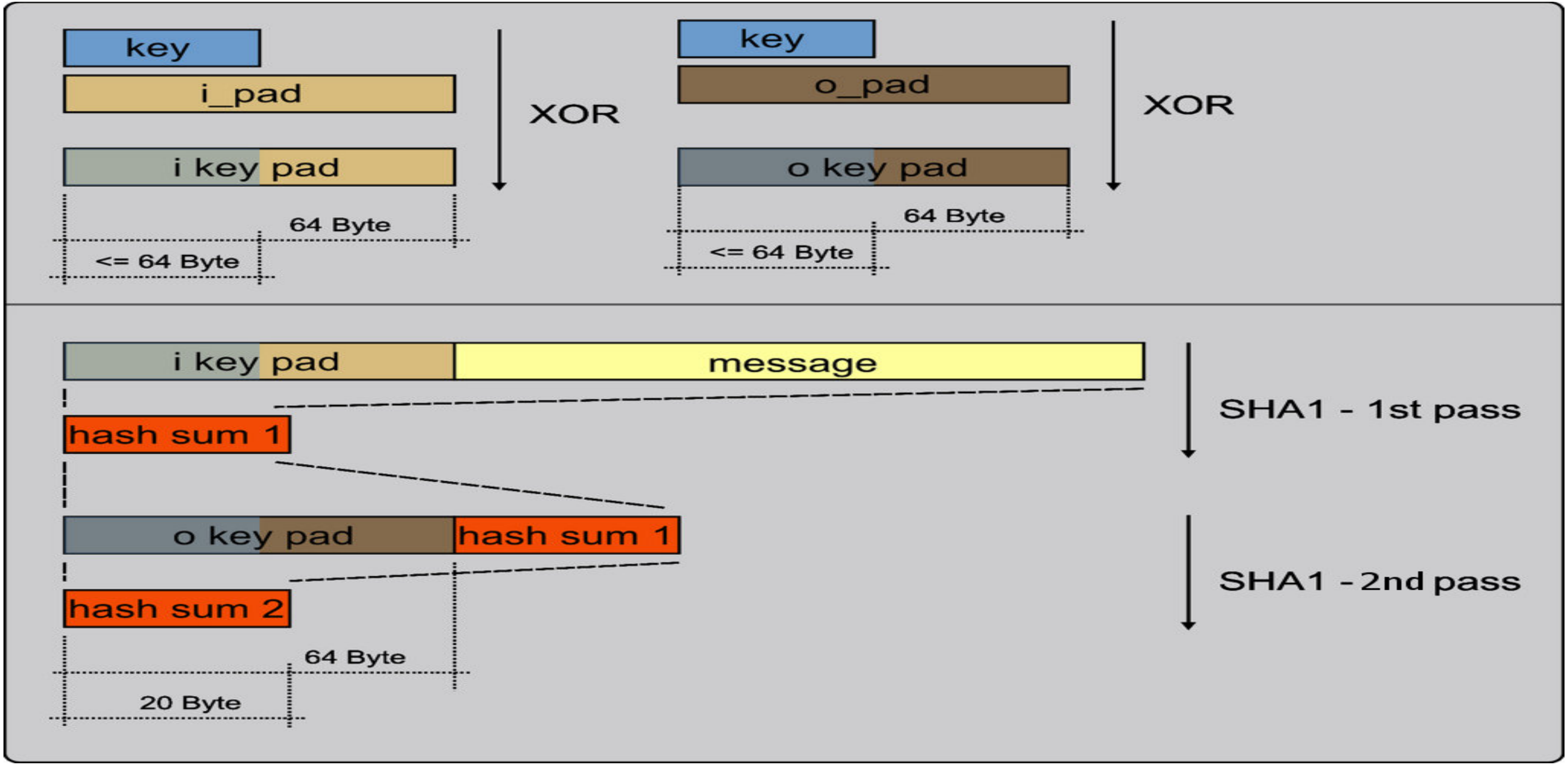
example: SHA-256 ; output is 256 bits

Building a MAC out of a hash function:

Standardized method: HMAC

$$S(k, m) = H(k \oplus opad \,\|\, \mathbf{H(k \oplus ipad \,\|\, m)})$$

Hash MAC (HMAC-SHA256)



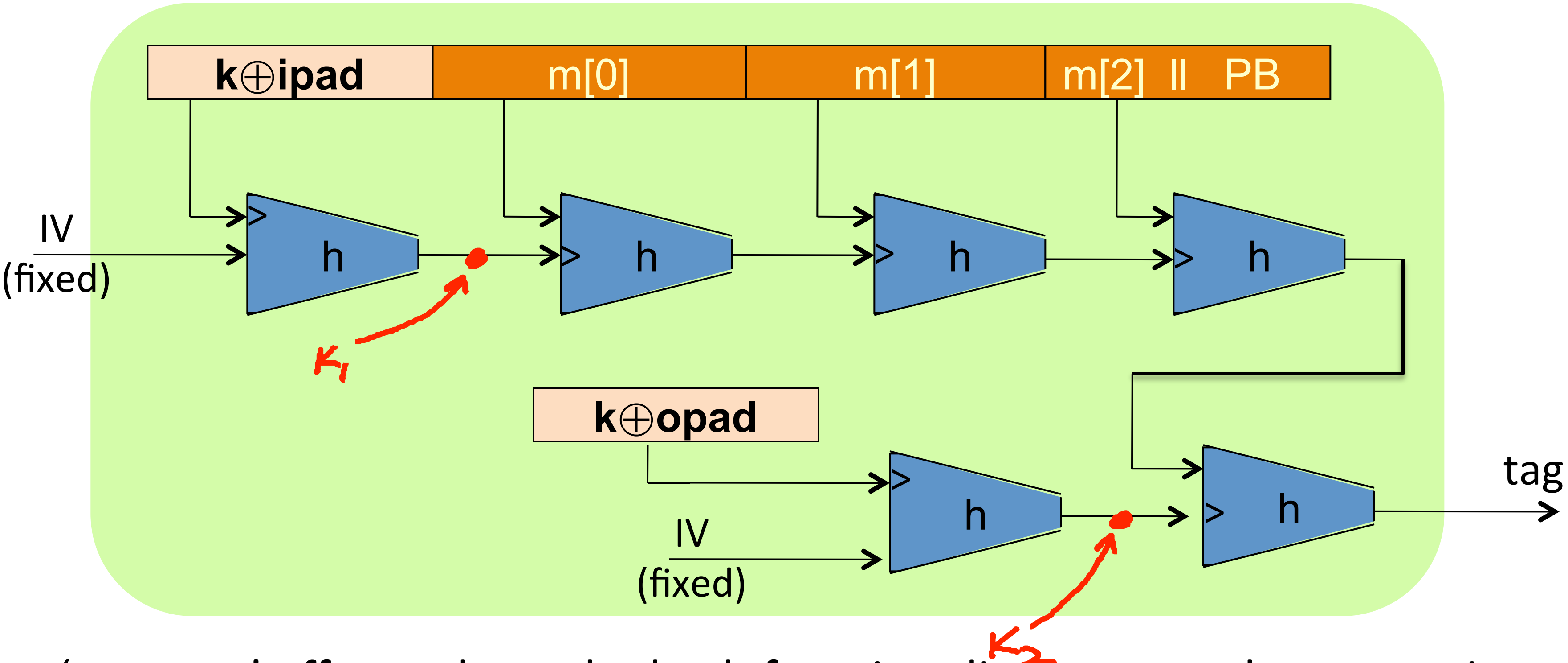
Why HMAC is Secure Against Length Extension Attack

- The Attacker knows data, length(secret), hashsum2 (aka signature), and of course his own extension

- The trouble is that he needs to know hashsum1 in order to seed the second call to hash

- Let us try a length extension attack with HMAC:
  - The attacker knows the length of the input to the second hash call, and his own extension
  - Computes new tag by seeding hash call with hashsum2, and hash(attacker extension || New-pad) = signature’
  - This won’t verify properly at Bob’s end, who is matching signature’ ≠ hash(secret || data || attacker-extension)

HMAC in pictures



The ‘memory’ effect where the hash function digest remembers state is broken by this construction. Hence, the length extension attack fails.

HMAC properties

HMAC is assumed to be a secure PRF

- Can be proven under certain PRF assumptions about h(.,.)
- Security bounds similar to NMAC
  - Need $q^2/|T|$ to be negligible ($q << |T|^{1/2}$)

In TLS: must support HMAC-SHA1-96

MACs from Collision Resistance

Let $I = (S,V)$ be a MAC for short messages over (K,M,T) (e.g. AES)

Let $H: M^{big} \longrightarrow M$

Def: $I^{big} = (S^{big}, V^{big})$ over (K, M^{big}, T) as:

$$\mathbf{S^{big}(k,m) = S(k,H(m)) \quad ; \quad V^{big}(k,m,t) = V(k,H(m),t)}$$

Thm: If I is a secure MAC and H is collision resistant

then I^{big} is a secure MAC.

Example: $S(k,m) = AES_{2\text{-block-cbc}}(k, \text{SHA-256}(m))$ is a secure MAC.

MACs from Collision Resistance

$S^{big}(k, m) = S(k, H(m))$; $V^{big}(k, m, t) = V(k, H(m), t)$

Collision resistance is necessary for security:

Suppose adversary can find $m_0 \neq m_1$ s.t. $H(m_0) = H(m_1)$.

Then: **S^{big}** is insecure under a 1-chosen msg attack

step 1: adversary asks for $t \longleftarrow S(k, m_0)$

step 2: output (m_1, t) as forgery

Use of Cryptographic Salt in Hash-based User Authentication

- Login Program:
  - Computes hash of password, and compares against stored hash. If match, the user is authenticated. Otherwise, authentication attempt is rejected.

- Stored hashes are susceptible to theft

- If passwords are easy, then they are susceptible to dictionary attacks

- Dictionary attacks:
  - Large store of easy passwords
  - Compute hash and compare against stolen stored hashes

Use of Cryptographic Salt in Hash-based User Authentication

- Login Program:
  - Computes hash of password + random seed, and compares against stored hash. If match, the user is authenticated. Otherwise, authentication attempt is rejected.

- Sometimes store even hash(intermediate hashes, password, salt)

- Forces attacker who doesn't know the salt a priori to compute all possible "easy password + salt" combinations

- Modern systems use salts upto 128 bits long
  - Infeasible for attackers to store that large a dictionary

Cryptography Module: Putting it All Together

- Which security problems can cryptography help to solve?
  - Confidentiality through encryption schemes
  - Integrity and Authenticity through MACs and digital signatures
  - User authentication through hash functions

- We studied two forms of encryption schemes
  - Symmetric: Parties must share same key
    - One-time Pad
    - Stream and Block Ciphers
  - Asymmetric: Parties need not share the same key
    - Motivation: Parties don't want to secretly share keys ahead of time
    - RSA public-key encryption

Cryptography Module: Putting it All Together

- Which security problems can cryptography help to solve?
  - Confidentiality through encryption schemes
  - Integrity and Authenticity through MACs and digital signatures
  - User authentication through hash functions

- Digital signature schemes
  - Motivation: Parties want to authenticate messages
  - RSA public-key digital signature schemes

- Hash functions
  - User authentication
  - MACs
  - Integrity

Warning: verification timing attacks [L’09]

Example: Keyczar crypto library (Python) [simplified]

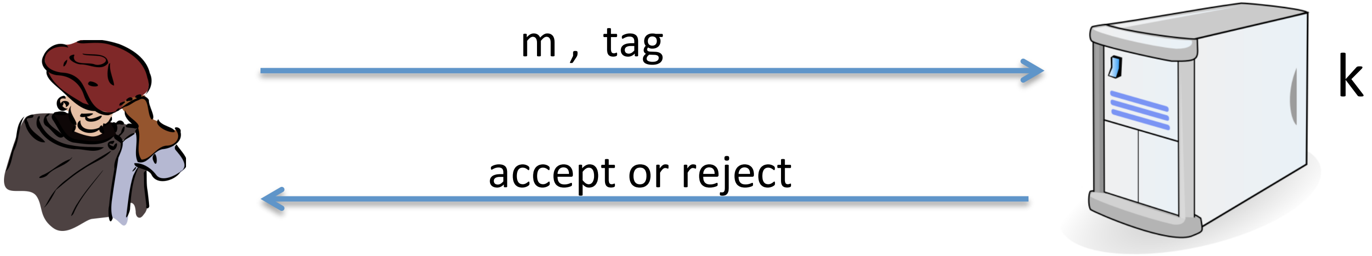
```
def Verify(key, msg, sig_bytes):
        return HMAC(key, msg) == sig_bytes
```

The problem: ‘==‘ implemented as a byte-by-byte comparison

- Comparator returns false when first inequality found

Warning: verification timing attacks [L'09]

target
msg **m**



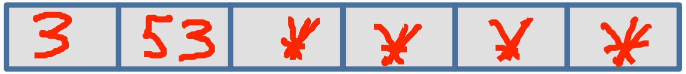
Timing attack: to compute tag for target message m do:

Step 1: Query server with random tag

Step 2: Loop over all possible first bytes 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



Defense #1

Make string comparator always take same time   (Python) :

```
return false if  sig_bytes  has wrong length
result = 0
for x, y in zip( HMAC(key,msg) , sig_bytes):
    result  |= ord(x) ^ ord(y)
return result == 0
```

Can be difficult to ensure due to optimizing compiler.

Defense #2

Make string comparator always take same time   (Python) :

```
def Verify(key, msg, sig_bytes):
    mac = HMAC(key, msg)
    return HMAC(key, mac) == HMAC(key, sig_bytes)
```

Attacker doesn't know values being compared

Lesson

Don’t implement crypto yourself !