# Message Authentication Codes

## Message Integrity

The goal is integrity and not confidentiality (public binaries on disk or banner ads on web pages)

Integrity is provided by MACs (Message Authentication Codes)

MAC signing algorithm:

Definition: MAC I = (S,V) defined over (K,M,T) is a pair of algorithms:

- S(k,m) outputs t in T → Signing algorithm

- V(k,m,t) outputs ‘yes’ or ‘no’ → Verification algorithm

For all k in K and m in M: V(k,m,S(k,m)) = ‘yes’

Don’t confuse with CRC. CRC is designed to detect random nor malicious errors.

**Integrity requires a secret key.**

## Secure MACs

For m1,m2,...,mq attacker is given ti← S(k,mi)

Attacker’s goal: **existential forgery**

The attacker wants to produce some new valid message/tar pair (m,t)

(m,t) not in { (m1,t1),...,(mq,tq) }

The requirements to be secure are:

- attacker cannot produce a valid tag for a new message

- given (m,t) attacker cannot even produce (m,t’) for t != t’

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

The Adversary send q messages to the Challenger: m1,...,mq in M

The Challenger responds with q tags: t1,...tq s.t. ti ← S(k,mi)

The Advesary sends a new pair (m,t)

The Challenger produces an answer b such that:

- b = 1 if V(k,m,t) = ‘yes’ and (m,t) not in { (m1,t1),...,(mq,tq) }

- b = 0 otherwise

Definition: I=(S,V) is a secure MAC if for all “efficient” A:

AdvMAC[A,I] = Pr[Challenger outputs 1] is “negligible”

The tag length must be long enough for the MAC to be secure. If it is not then an attacker can simply guess the tag.

For example for a tag of 5 bits long the probability of guessing the tag is only 1/25 = 1/32 and the AdvMAC will be 1/32 which is not negligible

Tag lengths of 64, 96, 128 bits

# MACs based on PRFs

## A secure PRF ⇒ a secure MAC

For a PRF: F: KxX → Y define a MAC IF = (S,V) as:

- S(k,m) := F(k,m)

- V(k,m,t): output ‘yes’ if t = F(k,m) and ‘no’ otherwise

## Security

Theorem: If F: KxX → Y is a secure PRF and 1/|Y| is negligible (i.e. |Y| is large) the IF is a secure MAC.

In particular, for every efficient MAC adversary A attacking IF there exists an efficient PRF adversary B attacking F such that:

AdvMAC[A,IF] =< AdvPRF[B,F] + 1/|Y|

Resume: IF is secure as long as |Y| is large, say |Y| = 280

## Examples

AES: a MAC for 16-byte messages

Main question: how to convert Small-MAC into a Big-MAC?

Two main constructions used in practice:

- CBC-MAC (banking - ANSI X9.9, X9.19, FIPS 186-3)

- HMAC (Internet protocol: SSL, IPsec, SSH,...)

Both convert a small-PRF into a big-PRF

## Truncating MACs based on PRFs

Easy lemma: suppose F: KxX → {0,1}n is a secure PRF

Teh so is Ft(k,m)=F(k,m)[1..t] for all 1=<t=<n

if (S,V) is a MAC is based on a secure PRF outputtin n-bit tags the truncated MAC outputting w bits is secure as log as 1/2w is still negligible (say w>=64)

# CBC-MAC and NMAC

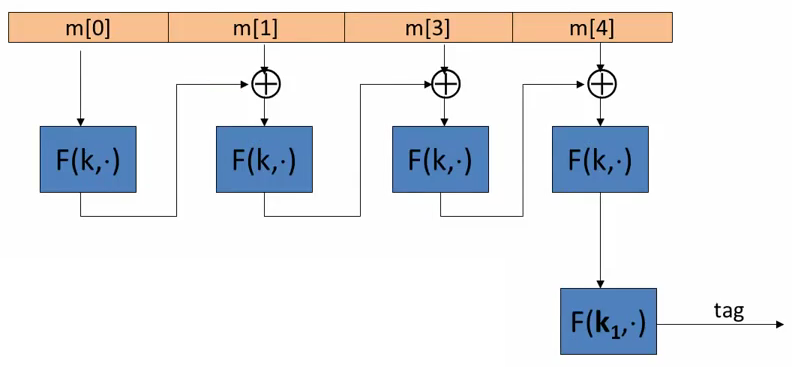
Given a PRF for short message (AES 16-bits) construct a PRF for long messages

From here on let X={0,1}n e.g. n=128

## Construction 1: encrypted CBC-MAC (ECBC)

Let F: KxX → X be a PRP

Define new PRF FECBC:K2xX<=L→ X



At the end of the process an extra encryption is made with another independent key k1. That makes the “raw CBC” function to become a secure MAC (raw CBC is insecure)

Suppose we define a MAC IRAW=(S,V) where

S(k,m) = rawDBD(k,m)

Without the last step ECBC-MAC is insecure. It is easily broken using a 1-chosen msg attack.

Adversary works as follows:

1.- Choose an arbitrary one-block message m in X

2.- Request the tag for m. Get t=F(k,m)

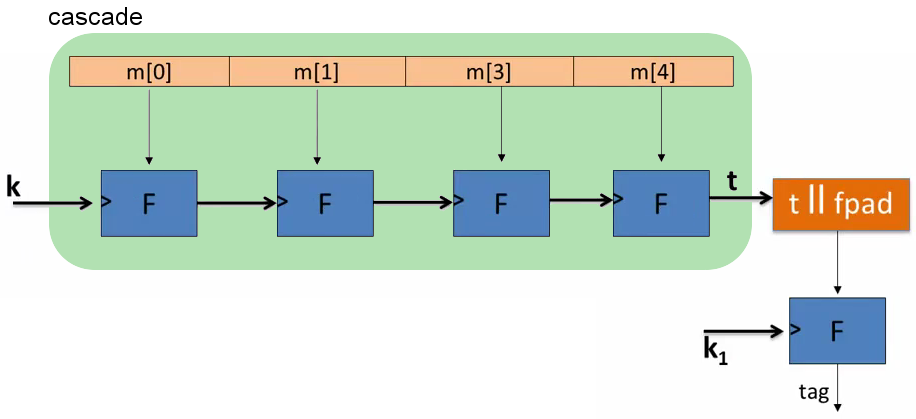
3.- Output t as MAC forgery for the 2-block message m’=(m, t xor m). That’s true because:

rawCBC(k,(m, t xor m)) = F(k, F(k,m) xor (t xor m) ) = F(k,t xor (t xor m)) = t

## Construction 2: NMAC (nested MAC)

Let F: KxX→ K be a PRF

Define new PRF FNMAC: K2xX<=L→ K



“cascade” will output an element in K. This will not be a secure MAC.

fpad = fix pad.

t || fpad is an element in X

Without the last step the MAC is insecure. Explanation:

After one tag: cascade(k,m) an attacker can extend the initial message with one more block w. The attacker can then calculate the tag for the new extended message m||w because he knows the entries for the las F-block:

- The previous tag = cascade (k,m)

- The new block = w

## ECBC-MAC and NMAC analysis

Theorem: For any L>0,

For every efficient q-query PRF adversary A attacking FECBC or FNMAC there exists an efficient adversary B such that:

AdvPRF[A, FECBC] <= AdvPRP[B,F] + 2 q2 / |X|

AdvPRF[A, FNMAC] <= q L AdvPRF[B,F] + q2 / 2|K|

CBC-MAC is secure as long as q << |X|½

NMAC is secure as long as q << |K|½ (264 for AES-128)

Example:

Suppose we want AdvPRF[A,FECBC] <= 1/232 ⇐ q2/|X| < 1/232

AES: |X| = 2128 ⇒ q<248 → So, after 248 messages must must change key

3DES: |X| = 264 ⇒ q<216

After signing |X|½ message with ECBC-MAC or |K|½ messages with NMAC the MACs become insecure

Suppose the underlying PRF F is a PRP (e.g. AES)

Then both PRFs (ECBC and NMAC) have the following extension property:

For all x,y,w: FBIG(k,x) = FBIG(k,y) ⇒ FBIG(k,x||w) = FBIG(k,y||w)

These means that if there is a collision then the extend with the same “w” block will generate a collision too.

Remember that the function F is a PRP, so it is a one-to-one function. Then the value before applying the F function at the last step is the same for x and y. So, if the result of the raw CBC is the same on both cases, the result extending the messages will be the same too ⇒ the result of the CBC will be the same.

### Attack

1.- issue |Y|½ message queries for random messages in X

obtain (mi,ti) for i=1,...,|Y|½

2.- find a collision tu=tv for u!=v (one exists with high probability by birthday paradox)

3.- choose some w and query for t:=FBIG(k,mu||w)

4.- output forgery (mv||w,t). Indeed t:=FBIG(k,mv||w)

### Comparison

ECBC-MAC is commonly used as an AES-based MAC:

- CCM encryption mode (used in 802.11i)

- NIST standard called CMAC

NMAC not usually used with AES or 3DES

- Main reason: need to change AES key on every block → requires re-computing AES key expansion

- However, NMAC is the basis for a popular MAC called HMAC

# MAC padding

For security the padding function must be revertible

Also m0 != m1 ⇒ pad(m0) != pad(m1)

ISO: pad with “100...00”. Add new dummy block if needed.

- The “1” indicates the beginning of the pad

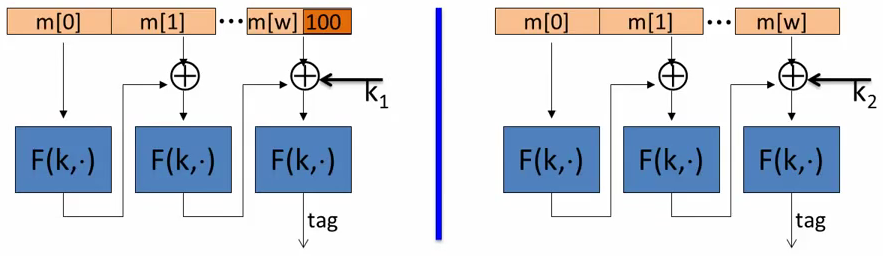
## CMAC (NIST standard)

Variant of CBC-MAC where key = (k,k1,k2)

k1 and k2 are derived from the key k through a PRG

No final encryption step (extension attack thwarted by las keyed xor)

No dummy block (ambiguity resolved by use of k1 or k2)

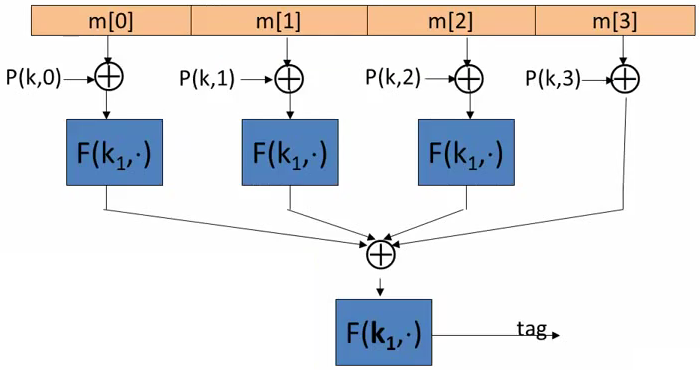


# PMAC and the Carter-Wegman MAC

PMAC - Parallel MAC

Let F: K x X → X be a PRF

Define new PRF FPMAC: K2 x X<=L → X



The function P make the MAC secure. If the P function is eliminated the blocks can be swapped between them and the result will be the same.

P(k, i) is a very easy function to compute

Padding similar as PMAC

## PMAC Theorem:

For any L>0

If F is a secure PRF over (K,X,X) then FPMAC is a secure PRF over (K2,X<=L, X)

For every efficient q-query PRF adversary A attacking FPMAC there exists an efficient PRF adversary B such that:

AdvPRF[A, FPMAC] <= AdvPRF[B, F] + 2 q2 L2 / |X|

PMAC is secure as long as qL << |X|½

## PMAC is incremental

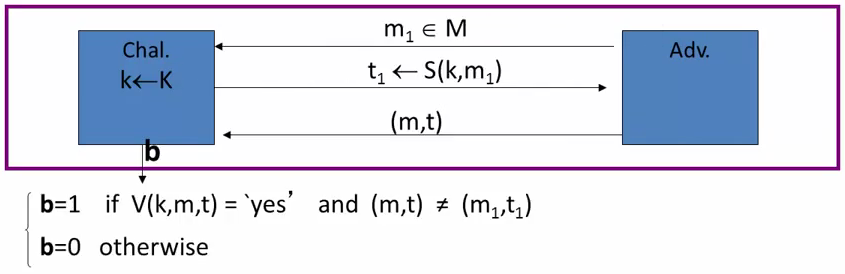
Suppose F is a PRP (we can invert it)

When m[1] → m’[1] can we quickly update tag?

We can do F-1(k1,tag) xor F(k1,m[1]xorP(k,1)) xor F(k1,m’[1] xor P(k,1)) and then apply F(k1,·)

## One time MAC (analog of one time pad)

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



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

AdvMAC[A,I] = Pr[Challenger outputs 1] is “negligible”

## One time MAC: an example

Can be secure against all adversaries and faster than PRF-based MACs

Let q be a large prime (e.g. q = 2128+51)

key = (k, a) in {1...q}2 (two random ints in [1,q])

msg = ( m[1],...,m[L]) where each block is 128 bit integer

S(key,msg) = Pmsg(k) + a (mod q)

where Pmsg(x) = m[L] xL +...+ m[1] x is a polynomy of degree L

Fact: given S(key,msg1) adversary has no info about S(key,msg2)

## One time MAC ⇒ Many-time MAC

Let (S,V) be a secure ont-time MAC over (KI,M,{0,1}n)

Let F:KF x {0,1}n → {0,1}n be a secure PRF.

**Carter-Wegman MAC**: CW( (k1,k2), m) = ( r ,F(k1,r) xor S(k2,m) ) for random r← {0,1}n

Theorem: If (S,V) is a secure ont-time MAC and F a secure PRF then CW is a secure MAC outputting tags in {0,1}2n.

Verification of a CW tag (r,t) on message m.

We can use V(k2,m,·) from the one-time MAC such that:

V(k2,m, F(k1,r) xor t )

Construction 4: HMAC (Hash-MAC) → Next lesson

# Collision resistance: Introduction

Let H: M→ T be a hash function ( |M| >> |T| )

A collision for H is a pair m0, m1 in M such that:

H(m0) = H(m1) and m0 != m1

A function H is collision resistant if for all (explicit) “efficient” algorithms A:

AdvCR[A,H] = Pr[ A outputs collision for H ] is negligible

Example: SHA-256 (outputs 256 bits)

## MACs from Collision Resistance

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

Let H: Mbig → M

Def IBIG = (SBIG, VBIG) over (K,MBIG, T) as:

SBIG(k,m) = S(k,H(m)) ; Vbig(k,m,t) = V(k,H(m), t)

Theorem: If I is a secure MAC and H is collision resistant then Ibig is a secure MAC

Example: S(k,m) ) AES2-block-cbc(k,SHA-256(m)) is a secure MAC

Collision resistance is necessary for security:

Suppose adversary can find m0 != m1 such that H(m0) = H(m1)

Then: Sbig is insecure under a 1-chosen message attack

- step 1: adversary asks for t← S(k,m0)

- setp 2: output (m1, t) as forgery

# Generic birthday attack

## Generic attack on Collision Resistant functions

Let H: M → {0,1}n be a hash function (|M| >> 2n)

Generic algorithm to find a collision in time O(2n/2) hashes

1.- Choose 2n/2 random messages in M: m1,...,m2n/2 (distinct with high probability)

2.- For i = 1,...,2n/2 compute ti = H(mi) in {0,1}n

3.- Look for a collision (ti = tj). If not found, got back to step 1

The bitthday paradox

Let r1,...,rn in {1,...,B} be independent identically distributed integers

Theorem: when M=1.2 x B1/2 then Pr[ exists i!=j: ri=rj] >= ½

Proof: (for uniform independent r1,...,rn)

Pr[ exists i!=j: ri=rj] = 1 - Pr[for all i!=j: ri!=rj] = 1 - ((B-1) / B) ( (B-2) / B )...( (B-n+1) / B ) =>

We take r1 → r1 doesn’t collide with nothing

We take r2 → The Pr[r2 != r1] = (B-1) / B

We take r3 → The Pr[r3 != (r1,r2)] = (B-2) / B

…

When n = 1.2 x B1/2 → n2/2 = 0.72 B so:

Pr[exists i!=j: ri = rj] >= 1 - e-0.72 = 0.53 > ½

Generic attack

H: M→ {0.1}n. Collision finding algorithm:

1.- Choose 2n/2 random elements in M: m1,..,m2n/2

2.- For i = 1,..,2n/2 compute ti = H(mi) in {0,1}n

3.- Look for a collision (ti=tj). If not found, got back to step 1

Expected number of iteration aprox = 2

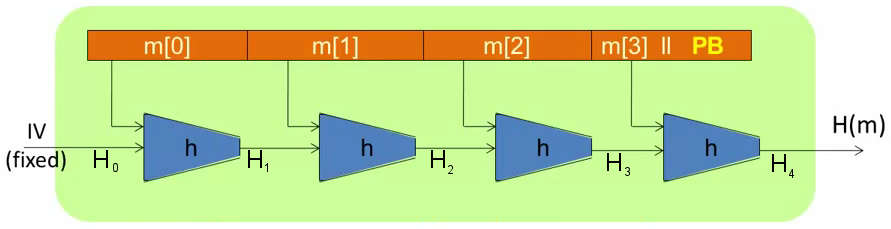
Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Digest size (bits)** | **Speed (MB/sec)** | **Generic Attack Time** |
| SHA-1 | 160 | 153 | 280 |
| SHA-256 | 256 | 111 | 2128 |
| SHA-512 | 512 | 99 | 2256 |
| Whirlpool | 512 | 57 | 2256 |

Best known collision finder for SHA-1 requires 251 hash evaluations → don’t use on projects

# The Merkle-Damgard Paradigm

The Mergle-Damgard iterated construction



Given h: T x X → T (compression function)

we obtain H: X<=L → T Hi - chaining variables

IV is fixed, is part of the definition of the function

PB = padding block

Padding block = 100...0 || msg length (64 bits)

The message is limited by 264 bits length

If there is no space for the PB add another block

Theorem: if h is collision resistant then so is H

Proof: Counterpositive → collision on H ⇒ collision on h

Suppose H(M) = H(M’). We build collision for h

IV = H0 , H1 , … , Ht , Ht+1 = H(M)

IV = H’0 , H’1 , … , H’r , H’r+1 = H(M’)

If there is a collision H(M) = H(M’)

Let assume the Padding Block fits in the last block

h (Ht, Mt, || PB) = Ht+1 = H’r+1 h (H’r, M’r || PB ‘ )

if Ht != H’r OR Mt != M’r OR PB != PB’ ⇒ Collision for h

Suppose Ht = H’r and Mt = M’r and PB = PB’

if PB = PB’ then r = t (same PB implies same message length)

Then h(Ht-1,Mt-1) = Ht = H’t = h(H’t-1, M’t-1)

If Ht-1 != H’t-1 OR Mt-1!=M’t-1 ⇒ Collision for h

If there is no collision we apply the same to Ht-1=H’t-1 and Mt=M’t and Mt-1=M’t-1 …

We iterate to the begin of message:

1.- Find Collision for h, or

2.- For all i: Mi=M’i ⇒ M=M’

The second fact contradicts the starting assumption, so there have to be a collision, so:

If there is a collision on H ⇒ there is a collision on h

so

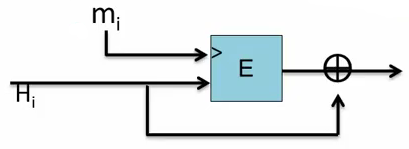
If there is no collision on h ⇒ there is no collision on H

# Constructing compression functions

## Compression function from a block cipher

Let E: Kx{0,1}n→ {0,1}n a block cipher

The Davies-Meyer compression function: h(H,m) = E(m,H) xor H



Theorem: Suppose E is an ideal cipher (collection of |K| random permutations), then finding a collision such that

h(H,m)=h(H’,m’)

takes O(2n/2) evaluations of (E,D)

It’s the same as the generic birthday algorithm on collision resistance functions. It’s the best possible result!!

## Example of a not collision resistance variation:

Suppose we define h(H,m) = E(m,H) → ignore the last xor step

The the resulting h(.,.) is not collision resistant.

To build a collision (H,m) and (h’,m’) we can choose a random (H.m.m’) and construct H’ as follows:

**H’=D(m’, E(m,H))** ⇒ E(m’,H’) = E(m’, D(m’,E(m,H))) ⇒ E(m’,H’) = E(m,H)

Remember that the encryption and decryption with m’ cancel out.

## Other block cipher constructions

Let E: {0,1}n x {0,1}n → {0,1}n for simplicity

Miyaguchi-Preneel:

h(H.m) = E(m,H) xor H xor m (Whirlpool)

h(H,m) = E(H xor m, m) xor m .

…

Total of 12 variants like this

Other natural variants are insecure:

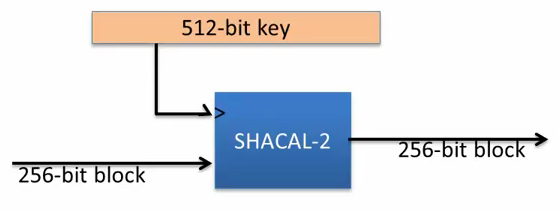
h(H,m) = E(m,H) xor m

## Case study: SHA-256

Merkle-Damgard function

Davies-Meyer compression function

Block cipher: SHACAL-2



## Provable compression functions

Choose a random 2000-bit prime p and random 1 <= u, v <= p

For m,h in {0,...,p-1} define

h(H,m) = uH vm (mod p)

Fact: finding collision for h(.,.) is as hard as solving “discrete-log” modulo p

Problem: slow

# HMAC: a MAC for SHA-256

Let H(m) be a Merkle-Damgard large hash function built from a small compression function h(H,m).

The question is: can we use H(.) to construct a MAC directly without having to rely on a PRF?

## MAC from a Merkle-Damgard Hash Function

Let H: X<=L → T a Collision Resistance Merkle-Damgard Hash Function

### Attempt 1: S(k,m) = H(k || m)

The MAC is insecure because:

Given H(k||m) we could compute H(k||m||PB||w) for any w

At the end of the Merkle-Damgard we have: H(k || m || PB). Then we can only add another block w and compute another step on the Mergle-Damgard structure with: h( Hn, w)

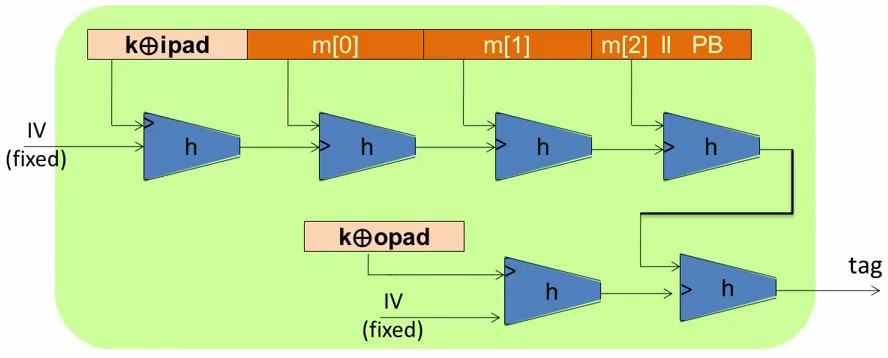
Standardized method: HMAC (Hash-MAC)

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:

HMAC: S(k,m) = H( k xor opad, H(k oxr ipad || m ) )



ipad = inner pad (fixed constant)

opad = outer pad (fixed constant)

Similar to the NMAC PRF. Main difference: the two keys k1, k2 are dependent

HMAC is assumed to be a secure PRF

- Can be proven under certain PRF assumptions about h(.,.)

- Security bounds similar to NMAC

- Need q2 / |T| to be negligible (q << |T|½ )

In TLS: must support HMAC-SHA1-96

it uses SHA1 because the requirement for HMAC is the compression function to be a PRF, it is not necessary to be collision resistant.

# Timing attacks on MAC verification

## Verification timing attacks

Example: Keyczar crypto library (Python)

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

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