Message Integrity

Goal: Message Integrity

Alice wants to send message ***m*** to Bob

- don’t fully trust the messenger or network carrying the message
- want to be sure what Bob receives is actually what Alice sent

Threat model:

- Mallory can see, modify, forge messages
- Mallory wants to trick Bob into accepting a message Alice didn’t send

Solution:
Message Authentication Code (MAC)

One approach:

1. Alice computes **v** := ***f***(**m**)

2. 

e.g. "Attack at dawn", 628369867…

3. Bob verifies that **v'** = ***f***(**m'**), accepts message iff this is true

Function ***f*** ?

Easily computable by Alice and Bob; <u>not</u> computable by Mallory

(Idea: Secret only Alice & Bob know)

We're sunk if Mallory can learn ***f***(**x**) for any **x** ≠ **m**!

Candidate ***f***:
Random function

Input: Any size up to huge maximum

Output: Fixed size (e.g. 256 bits)

Defined by a giant lookup table that's filled in by flipping coins

0 → 0011111001010001...
1 → 1110011010010100...
2 → 0101010001010000...
⋮ ⋮

Completely <u>impractical</u> [Why?]

Provably <u>secure</u> [Why?]

(Mallory can't do better than randomly guessing)

Want a function that's practical
but "looks random"…

Pseudorandom function (PRF)

Let's build one:

Start with a big *family of functions*
$\boldsymbol{f}_0, \boldsymbol{f}_1, \boldsymbol{f}_2, \ldots$ all known to Mallory

Use $\boldsymbol{f}_{\mathbf{k}}$, where **k** is a secret value
(or "key") known only to Alice/Bob

k is (say) 256 bits, chosen randomly

Kerckhoffs's Principle

Don't rely on secret functions

Use a secret key, to choose from
a function family [Why?]

Formal definition of a secure **PRF**:

Game against Mallory

1. We flip a coin secretly to get bit **b**
2. If **b**=0, let $\boldsymbol{g}$ be a random function
   If **b**=1, let $\boldsymbol{g} = \boldsymbol{f}_{\mathbf{k}}$, where **k** is a randomly chosen secret
3. Repeat until Mallory says “stop”:
   Mallory chooses **x**; we announce $\boldsymbol{g}(\mathbf{x})$
4. Mallory guesses **b**

We say $\boldsymbol{f}$ is a *secure PRF* if Mallory can’t do better than random guessing*

i.e., $\boldsymbol{f}_{\mathbf{k}}$ is indistinguishable in practice from a random function, unless you know **k**

Important fact: There’s an algorithm that always wins for Mallory

[What is it?] [How to fix it?]

A solution for Alice and Bob:

1. Let $\boldsymbol{f}$ by a secure PRF
2. In advance, choose a random **k** known only to Alice and Bob
3. Alice computes $\mathbf{v} := \boldsymbol{f}_k(\mathbf{m})$
4. 

5. Bob verifies that $\mathbf{v'} = \boldsymbol{f}_k(\mathbf{m'})$, accepts message iff this is true

[Important assumptions?]

What if Alice and Bob want to send more than one message?

[Attacks?] [Solutions?]

Annoying question:
Do PRFs actually exist?

Annoying answer:
We don’t know.

So how do we get a MAC?

Well-studied functions where we haven’t spotted a problem yet (e.g. HMAC-SHA256)

New Approach: Hash-based MAC (HMAC)

HMAC-SHA256

see RFC 2104

HMAC$_\mathbf{k}$(**m**) =

$$\mathrm{SHA256}\left(k \oplus c_1 \,\|\, \mathrm{SHA256}\left(k \oplus c_2 \,\|\, m \right) \right)$$

XOR 0x3636…

Concatenation

0x5c5c…

SHA256 function

takes arbitrary length input,

returns 256-bit output

What is SHA256?

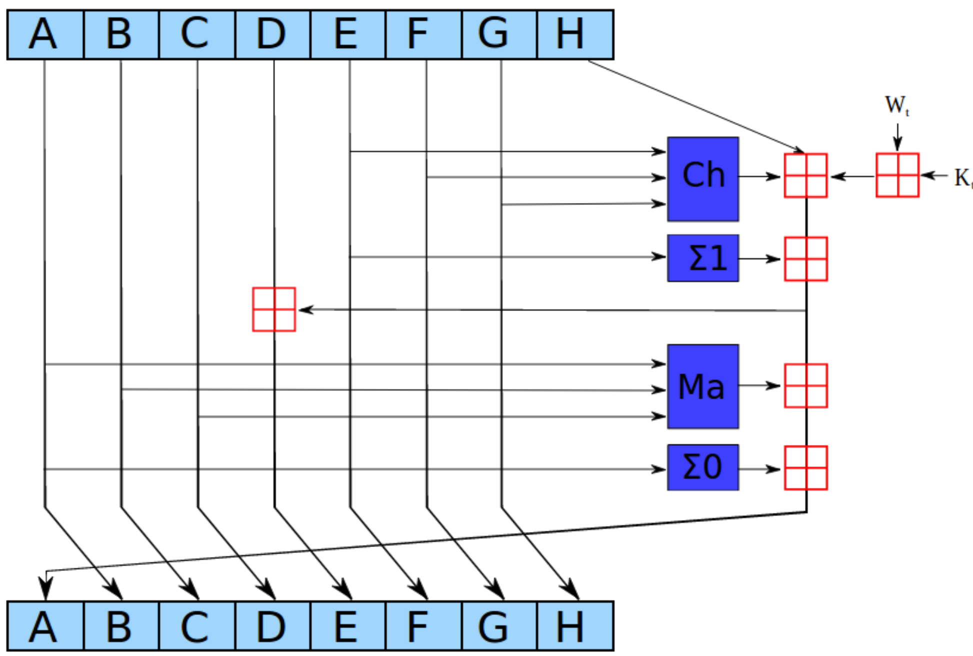
"Cryptographic hash function"

Input: arbitrary length data (<u>No key</u>)
Output: 256 bits

Built with "compression function" *h*

(256 bits, 512 bits) in → 256 bits out

Designed to be really hairy (64 rounds of this):

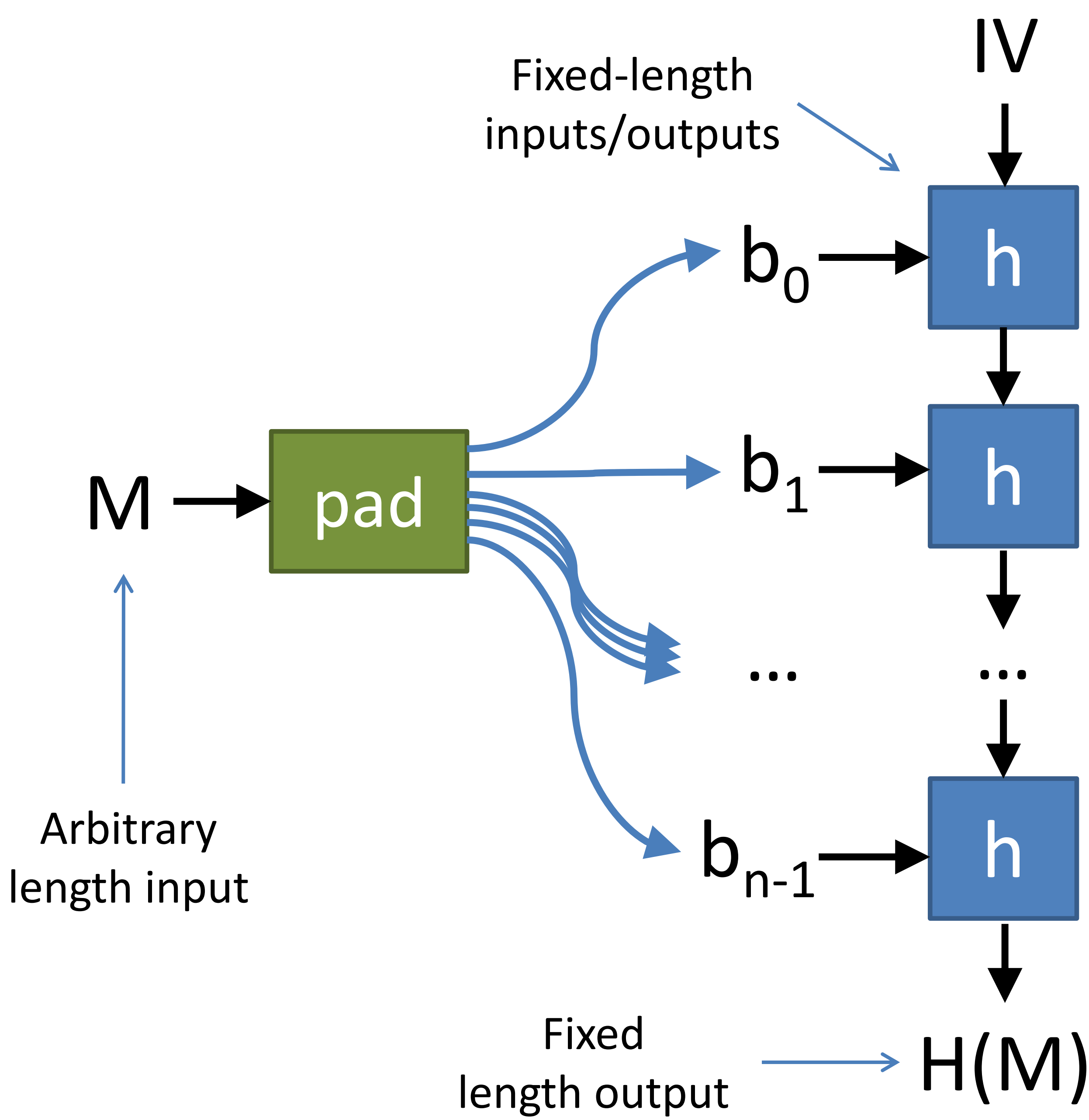


Entire algorithm:

1. Pad input to multiple of 512 bits (using a fixed algorithm [Why?])
2. Break into 512-bit blocks $\mathbf{b}_0$, $\mathbf{b}_1$, … $\mathbf{b}_{n-1}$
3. $\mathbf{y}_0$ = const,
   $\mathbf{y}_1 = \boldsymbol{h}(\mathbf{y}_0,\mathbf{b}_0)$,
   …,
   $\mathbf{y}_i = \boldsymbol{h}(\mathbf{y}_{i-1},\mathbf{b}_{i-1})$
4. Return $\mathbf{y}_n$

Merkle–Damgård Construction

- Arbitrary-length input
- Fixed-length output
- Built from fixed-size “compression function”



Hash function properties

Good hash functions should make it difficult to find …

First pre-image:

given $h(m)$, find m

Second pre-image:

given m_1, find m_2 s.t. $h(m_1) = h(m_2)$

Collision:

find *any* $m_1 \; != m_2$ s.t. $h(m_1) = h(m_2)$

Other hash functions:

MD5

Once ubiquitous

Broken in 2004

Turns out to be easy to find ***collisions***
(pairs of messages with same MD5 hash)

You'll investigate this in Project 1

SHA1

Currently widely used

Suspected to be weak

Don't use in new applications

SHA3

Different construction: "Sponge"

Not susceptible to length-extension

Message Authentication Code (MAC)

e.g. HMAC-SHA256

vs.

Cryptographic hash function

e.g. SHA256

not a strong PRF

Used to think the distinction didn't matter, now we think it does

e.g., ***length extension attacks***

Better to use a MAC/PRF (not a hash)

```
$ openssl dgst -sha256 -hmac <key>
```

[What if you don't need a key?]

So Far

The Security Mindset

Message Integrity

Next time …

The classic problem in crypto:

How can Alice send Bob a message,
with **confidentiality**?