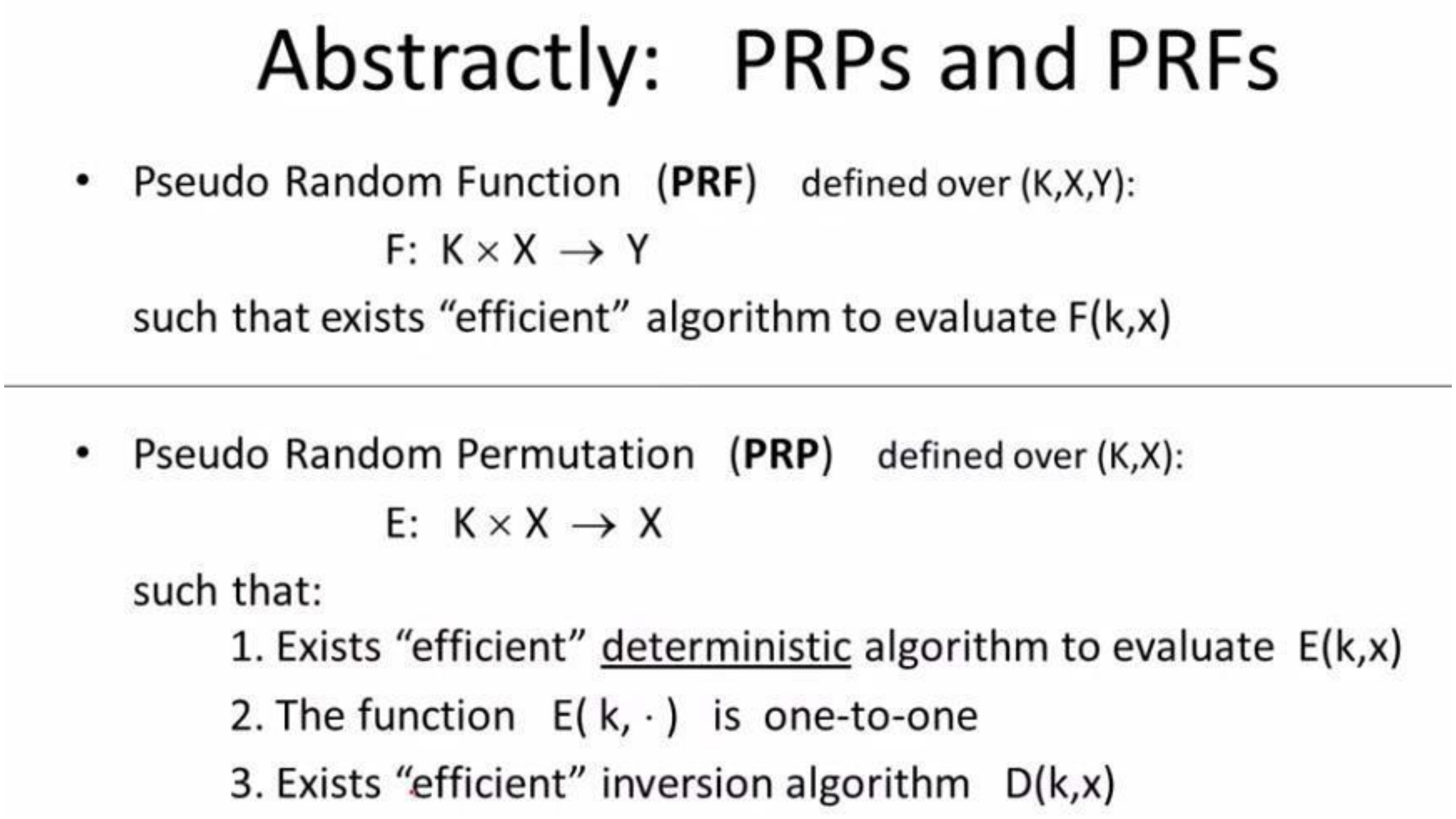
MESSAGE INTEGRITY

LECTURE CONTENT

- PRFs and PRPs
- Message integrity
- Secure MACs
- Encrypted CBC-MAC
- Hashes
- Laboratory_02: Python hashing
- Laboratory_03: Hashcat

PRFs and PRPs



Abstractly: PRPs and PRFs

- Pseudo Random Function (**PRF**) defined over (K,X,Y):

  $$F:\ K \times X \rightarrow Y$$

  such that exists “efficient” algorithm to evaluate F(k,x)

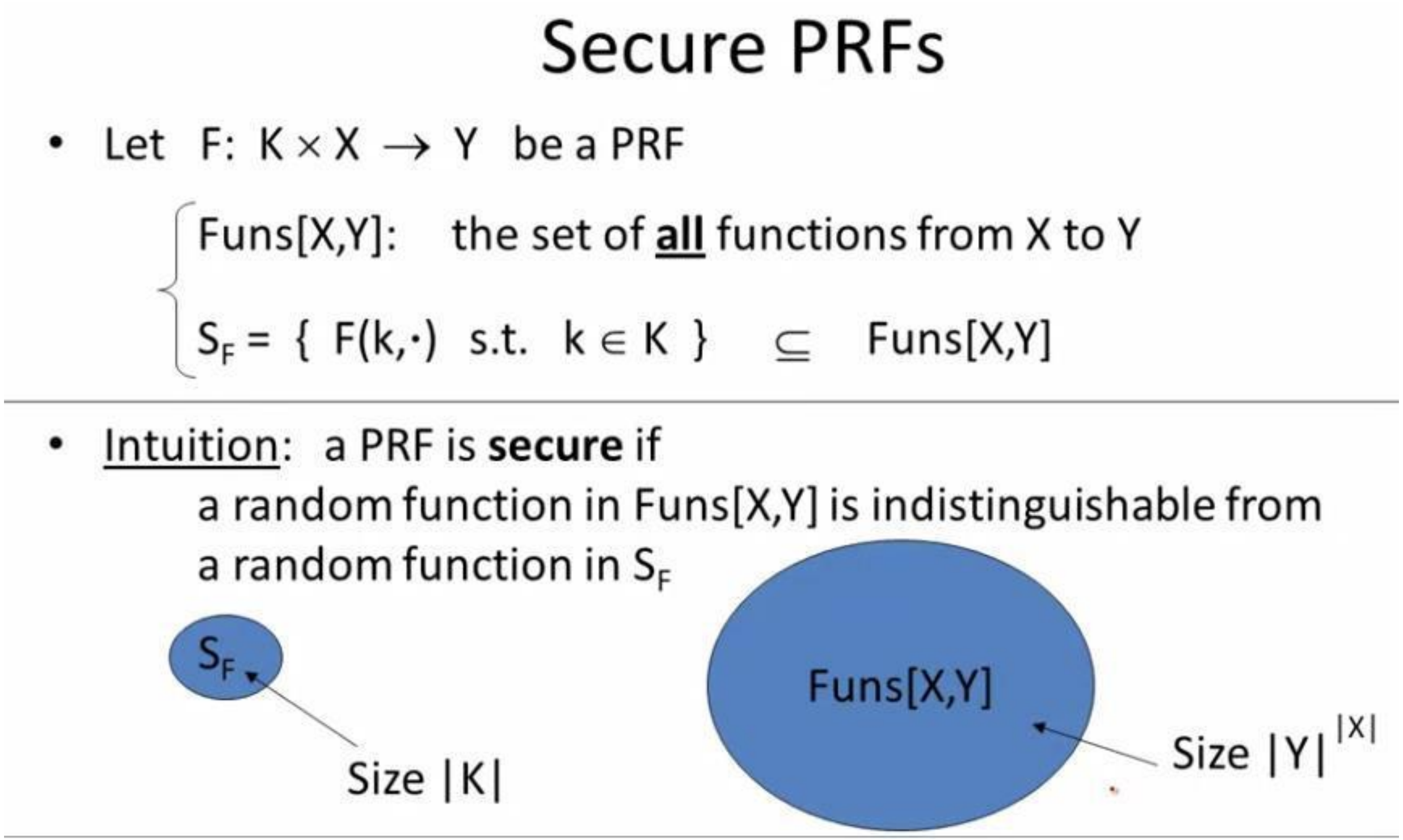
- Pseudo Random Permutation (**PRP**) defined over (K,X):

  $$E:\ K \times X \rightarrow X$$

  such that:

  1. Exists “efficient” deterministic algorithm to evaluate E(k,x)
  2. The function E(k, ·) is one-to-one
  3. Exists “efficient” inversion algorithm D(k,x)

PRFs and PRPs



Secure PRFs

- Let $F: K \times X \to Y$ be a PRF

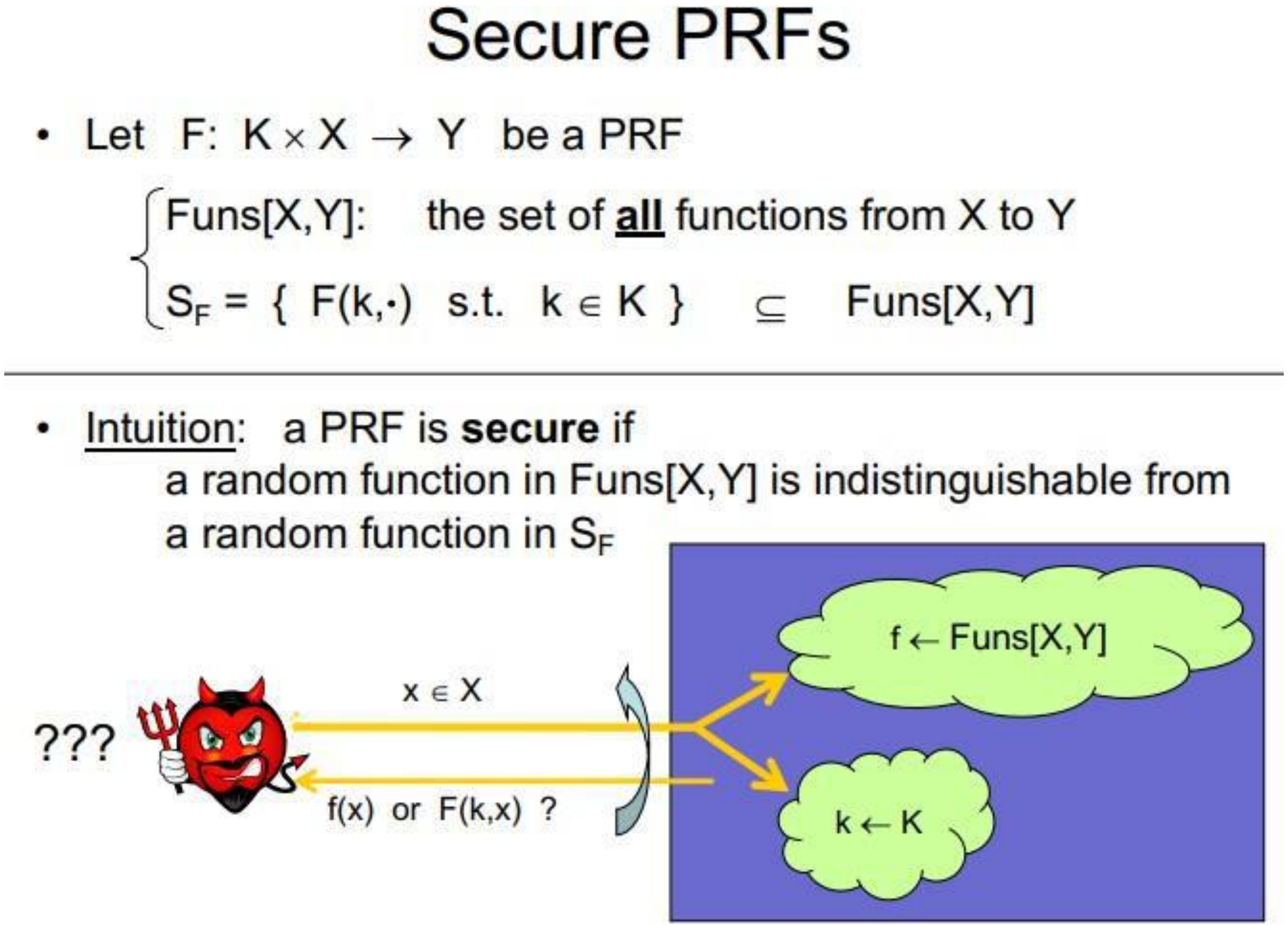
  Funs[X,Y]: the set of **all** functions from X to Y

  $S_F = \{ F(k,\cdot) \text{ s.t. } k \in K \} \subseteq \text{Funs}[X,Y]$

- Intuition: a PRF is **secure** if

  a random function in Funs[X,Y] is indistinguishable from a random function in S_F

PRFs and PRPs



Secure PRFs

- Let $F: K \times X \to Y$ be a PRF

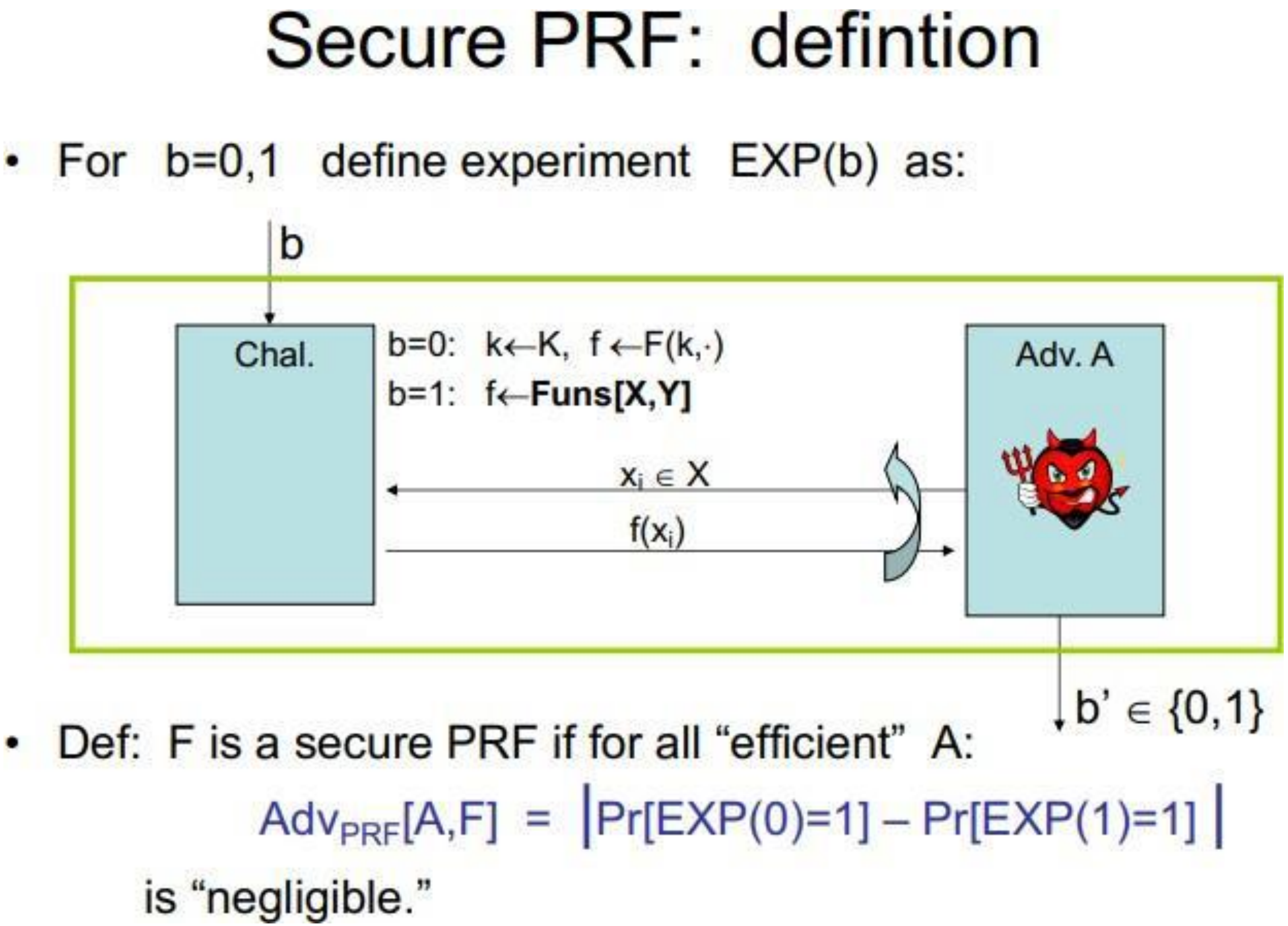
  Funs[X,Y]: the set of **<u>all</u>** functions from X to Y

  $S_F = \{ F(k,\cdot) \ \text{s.t.}\ k \in K \} \subseteq$ Funs[X,Y]

- <u>Intuition</u>: a PRF is **secure** if
  a random function in Funs[X,Y] is indistinguishable from
  a random function in S_F

???
$x \in X$
$f(x)$ or $F(k,x)$?
$f \leftarrow$ Funs[X,Y]
$k \leftarrow K$

PRFs and PRPs



Secure PRF: defintion

- For b=0,1 define experiment EXP(b) as:

b

Chal.

b=0: $k \leftarrow K$, $f \leftarrow F(k,\cdot)$

b=1: $f \leftarrow$ **Funs[X,Y]**

Adv. A

$x_i \in X$

$f(x_i)$

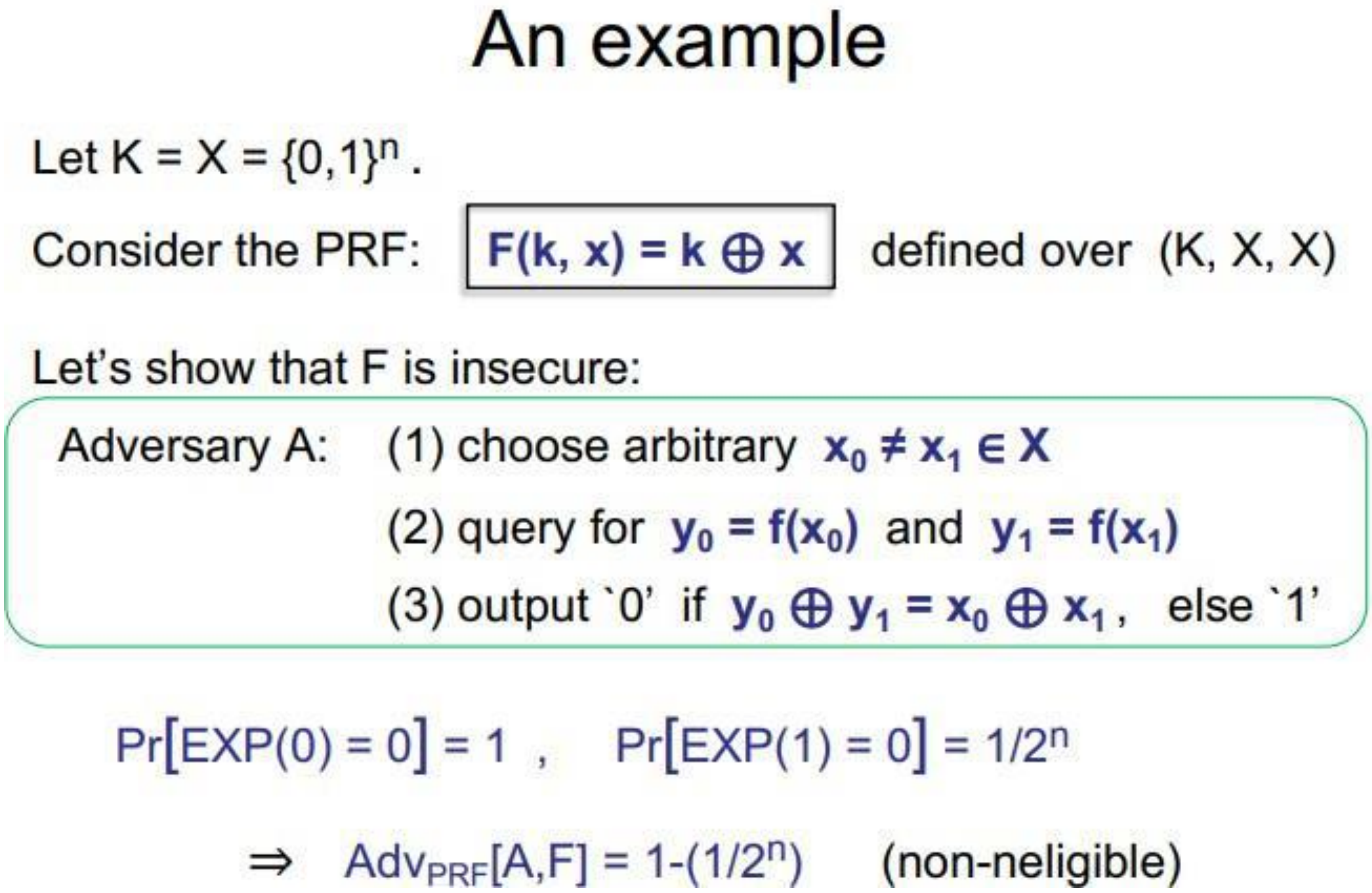
$b' \in \{0,1\}$

- Def: F is a secure PRF if for all "efficient" A:

$$\text{Adv}_{\text{PRF}}[A,F] = \left|\Pr[\text{EXP}(0)=1] - \Pr[\text{EXP}(1)=1]\right|$$

is "negligible."

PRFs and PRPs



An example

Let $K = X = \{0,1\}^n$.

Consider the PRF: $\boxed{F(k, x) = k \oplus x}$ defined over (K, X, X)

Let's show that F is insecure:

Adversary A: (1) choose arbitrary $x_0 \neq x_1 \in X$

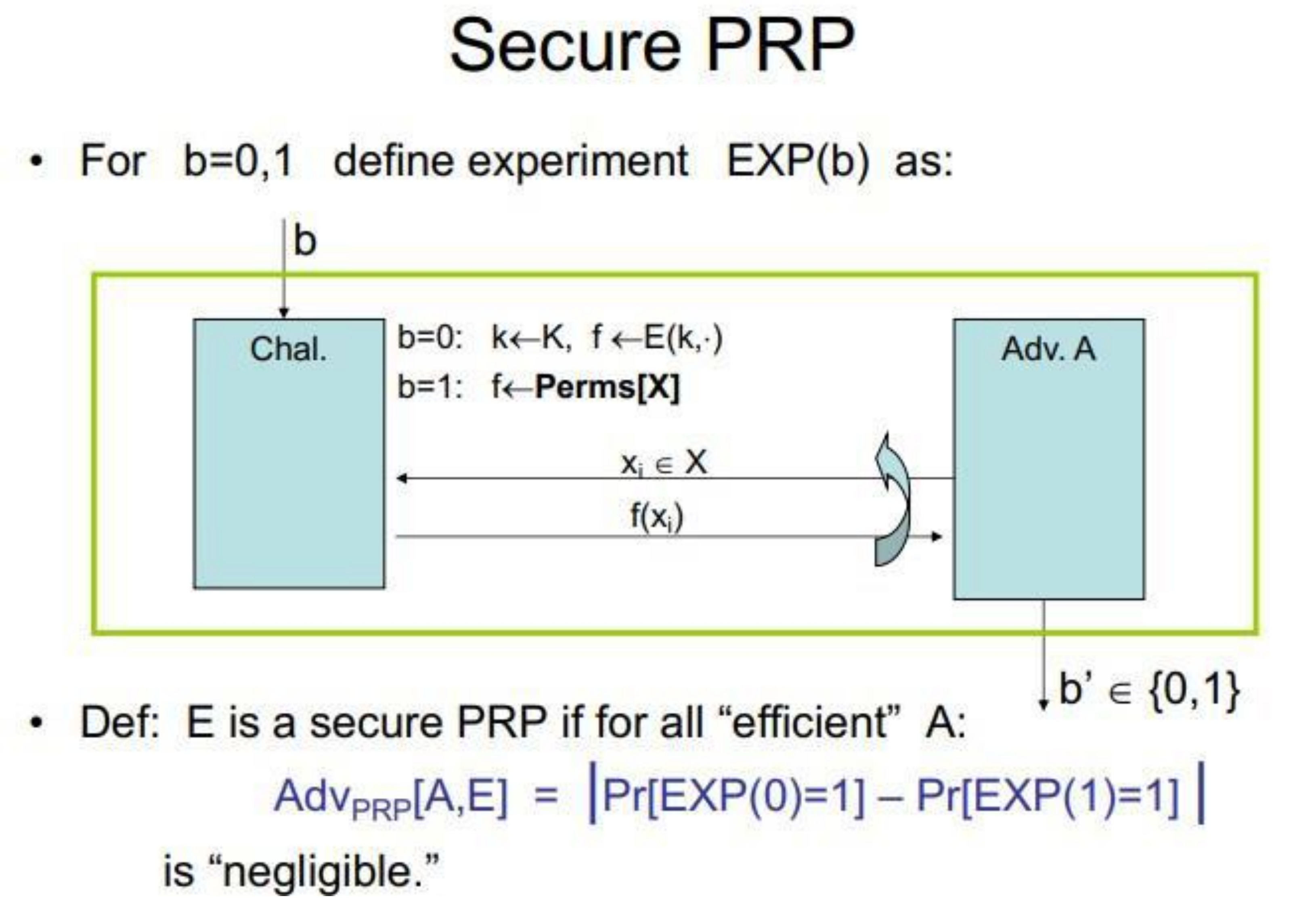
(2) query for $y_0 = f(x_0)$ and $y_1 = f(x_1)$

(3) output `0' if $y_0 \oplus y_1 = x_0 \oplus x_1$ , else `1'

$\Pr[\mathrm{EXP}(0) = 0] = 1$, $\Pr[\mathrm{EXP}(1) = 0] = 1/2^n$

$\Rightarrow$ $\mathrm{Adv}_{\mathrm{PRF}}[A,F] = 1-(1/2^n)$ (non-neligible)

PRFs and PRPs



Secure PRP

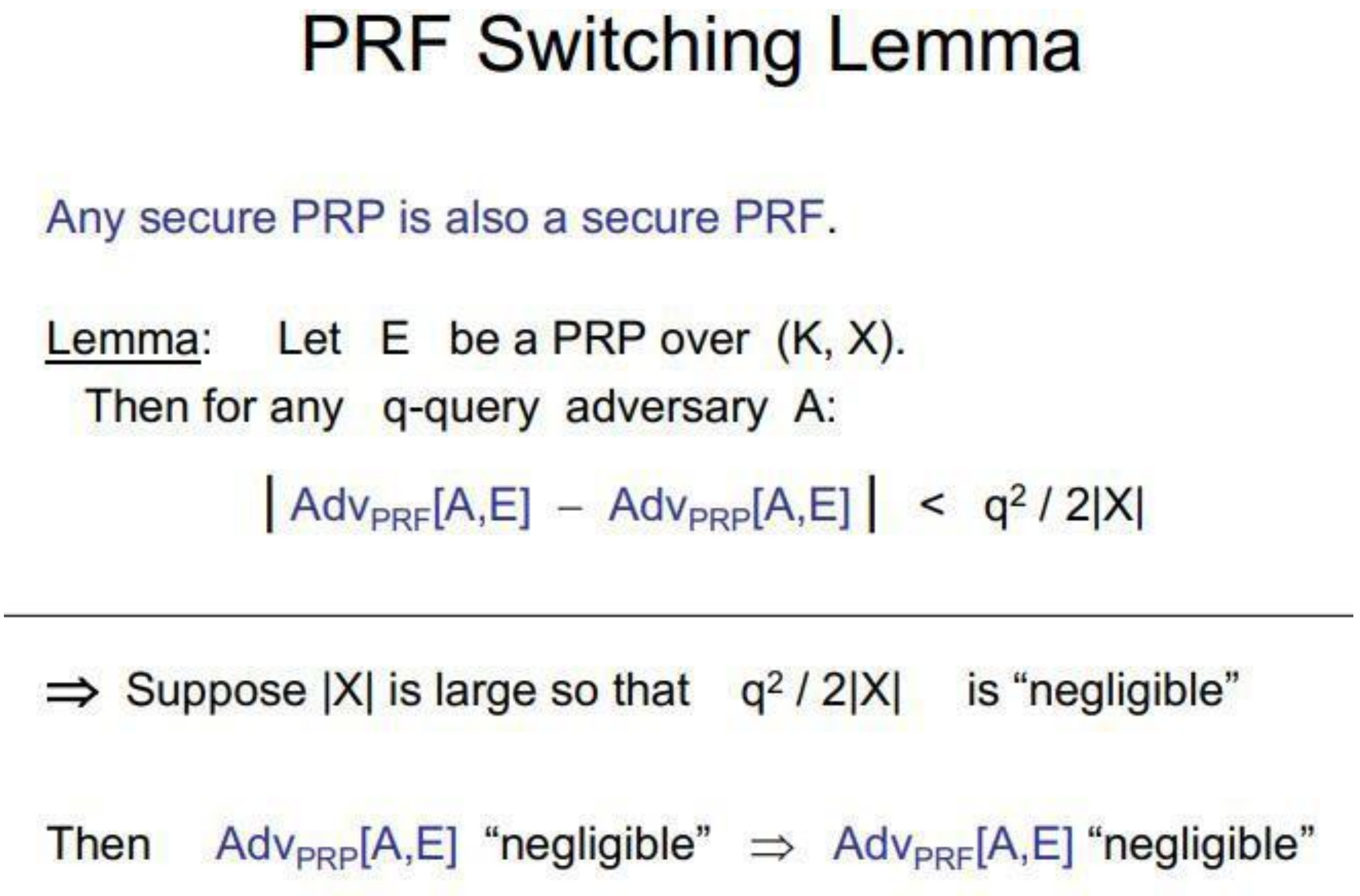
- For b=0,1 define experiment EXP(b) as:

- Def: E is a secure PRP if for all "efficient" A:

$$\mathrm{Adv}_{\mathrm{PRP}}[A,E] = \left|\Pr[\mathrm{EXP}(0)=1] - \Pr[\mathrm{EXP}(1)=1]\right|$$

is "negligible."

PRFs and PRPs



PRF Switching Lemma

Any secure PRP is also a secure PRF.

Lemma: Let E be a PRP over (K, X).
Then for any q-query adversary A:

$$\left| \mathrm{Adv}_{\mathrm{PRF}}[A,E] - \mathrm{Adv}_{\mathrm{PRP}}[A,E] \right| < q^2 / 2|X|$$

⇒ Suppose |X| is large so that $q^2 / 2|X|$ is "negligible"

Then $\mathrm{Adv}_{\mathrm{PRP}}[A,E]$ "negligible" ⇒ $\mathrm{Adv}_{\mathrm{PRF}}[A,E]$ "negligible"

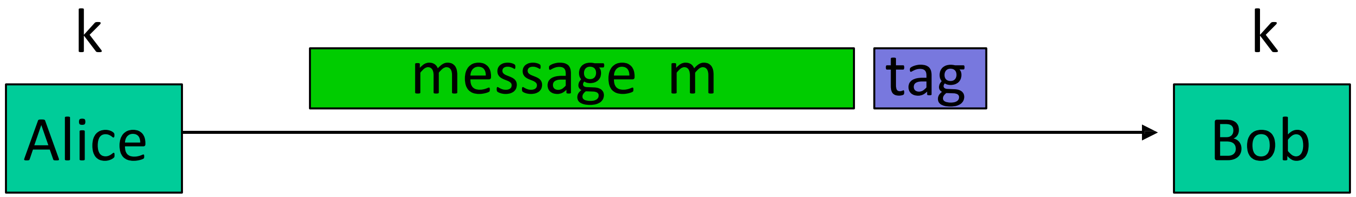
Message integrity

Goal: **integrity**, no confidentiality.

Examples:

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

Message integrity: MACs



Generate tag:

tag ← S(k, m)

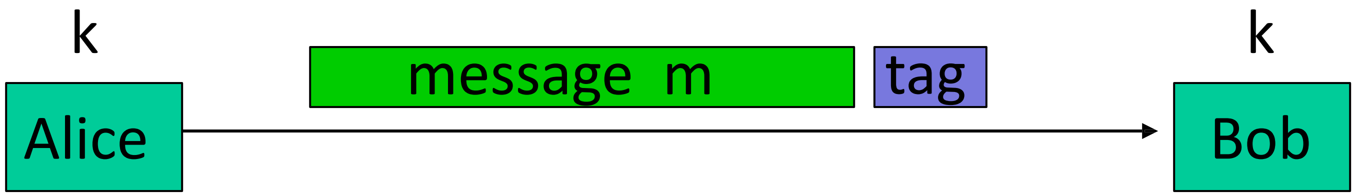
Verify tag:

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

Message integrity: MACs



k

message m

tag

Alice

Bob

k

Generate tag:

tag ← S(k, m)

Verify tag:

V(k, m, 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'

- Attacker can easily modify message m and re-compute CRC.

- CRC designed to detect **random**, not malicious errors.

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 **<u>new</u>** valid message/tag pair (m,t).

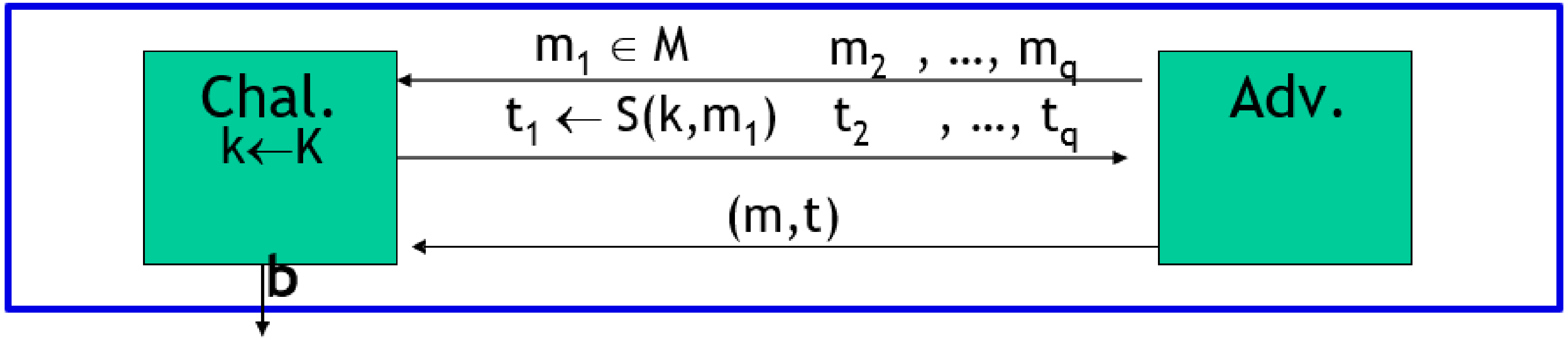
$$(m,t) \notin \left\{ (m_1,t_1), \ldots, (m_q,t_q) \right\}$$

⇒ 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:



Chal.
$k \leftarrow K$

$m_1 \in M$ m_2 , …, m_q

$t_1 \leftarrow S(k,m_1)$ t_2 , …, t_q

(m,t)

Adv.

b

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

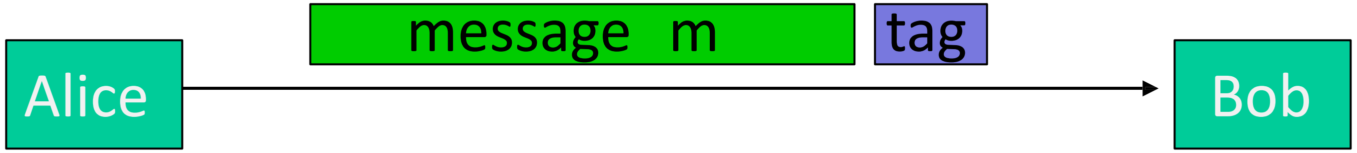
Def: I=(S,V) is a **secure MAC** if for all "efficient" A:

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

Secure MACs

For a PRF **F: K × X ⟶ Y** define a MAC $I_F = (S,V)$ as:

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



tag ⍰ F(k,m)

accept msg if

tag = F(k,m)

Secure MACs

Thm: If $F: K \times X \longrightarrow Y$ is a secure PRF and $1/|Y|$ is negligible

(i.e. $|Y|$ is large) then I_F is a secure MAC.

In particular, for every eff. MAC adversary A attacking I_F

there exists an eff. PRF adversary B attacking F s.t.:

$$\mathrm{Adv}_{\mathrm{MAC}}[A, I_F] \leq \mathrm{Adv}_{\mathrm{PRF}}[B, F] + 1/|Y|$$

$\Rightarrow$ I_F is secure as long as $|Y|$ is large, say $|Y| = 2^{80}$.

MAC 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 protocols: SSL, IPsec, SSH, …)

- Both convert a small-PRF into a big-PRF.

Truncating MACs

Easy lemma: suppose $F: K \times X \longrightarrow \{0,1\}^n$ is a secure PRF.

Then so is $F_t(k,m) = F(k,m)[1\ldots t]$ for all $1 \le t \le n$

⇒ if (S,V) is a MAC is based on a secure PRF outputting n-bit tags the truncated MAC outputting w bits is secure

… as long as $1/2^w$ is still negligible (say $w \ge 64$)

MACs for long messages

Recall: secure PRF **F** ⇒ secure MAC, as long as |Y| is large

$S(k, m) = F(k, m)$

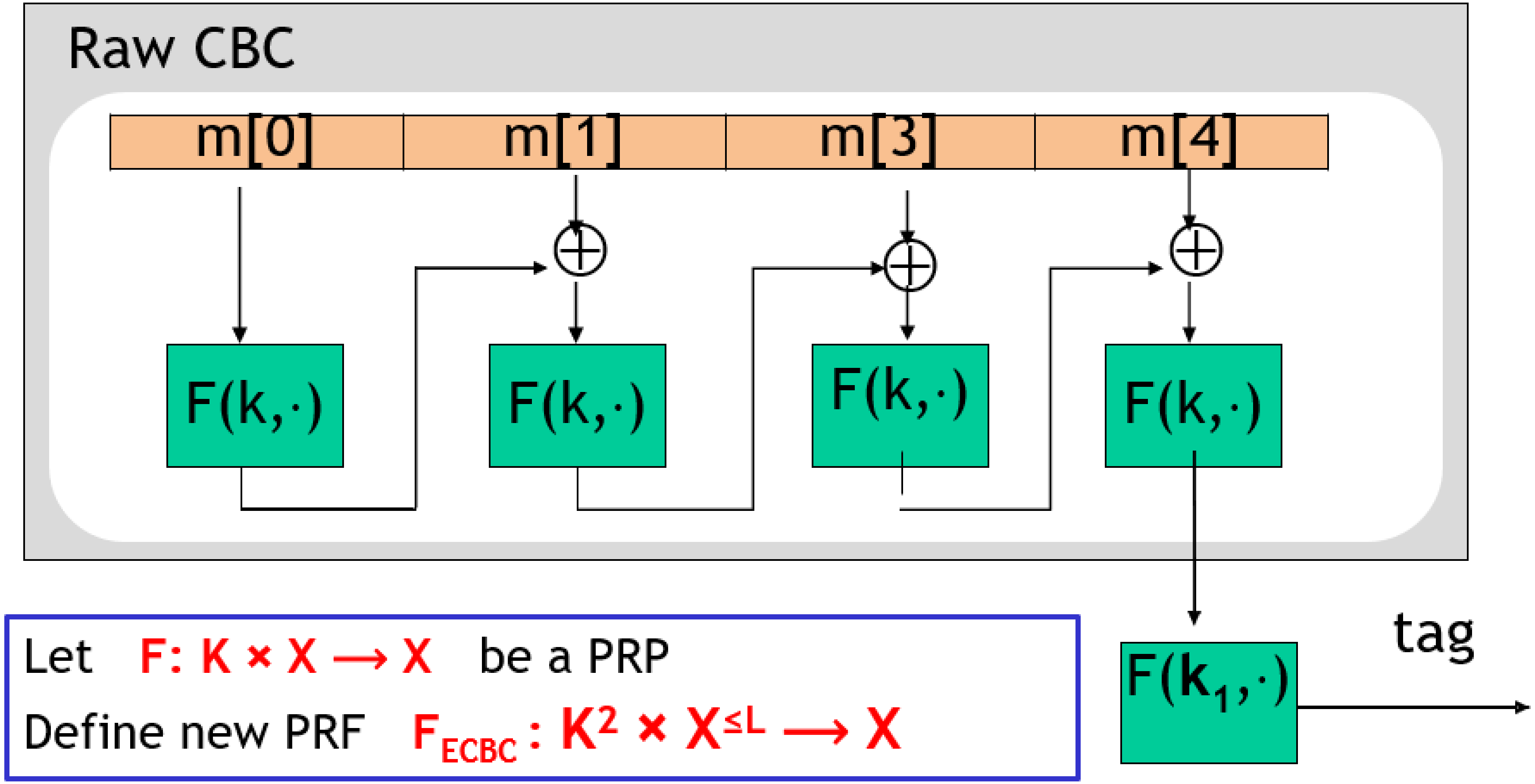
Our goal:

given a PRF for short messages (AES)

construct a PRF for long messages

From here on let $X = \{0,1\}^n$ (e.g. $n=128$)

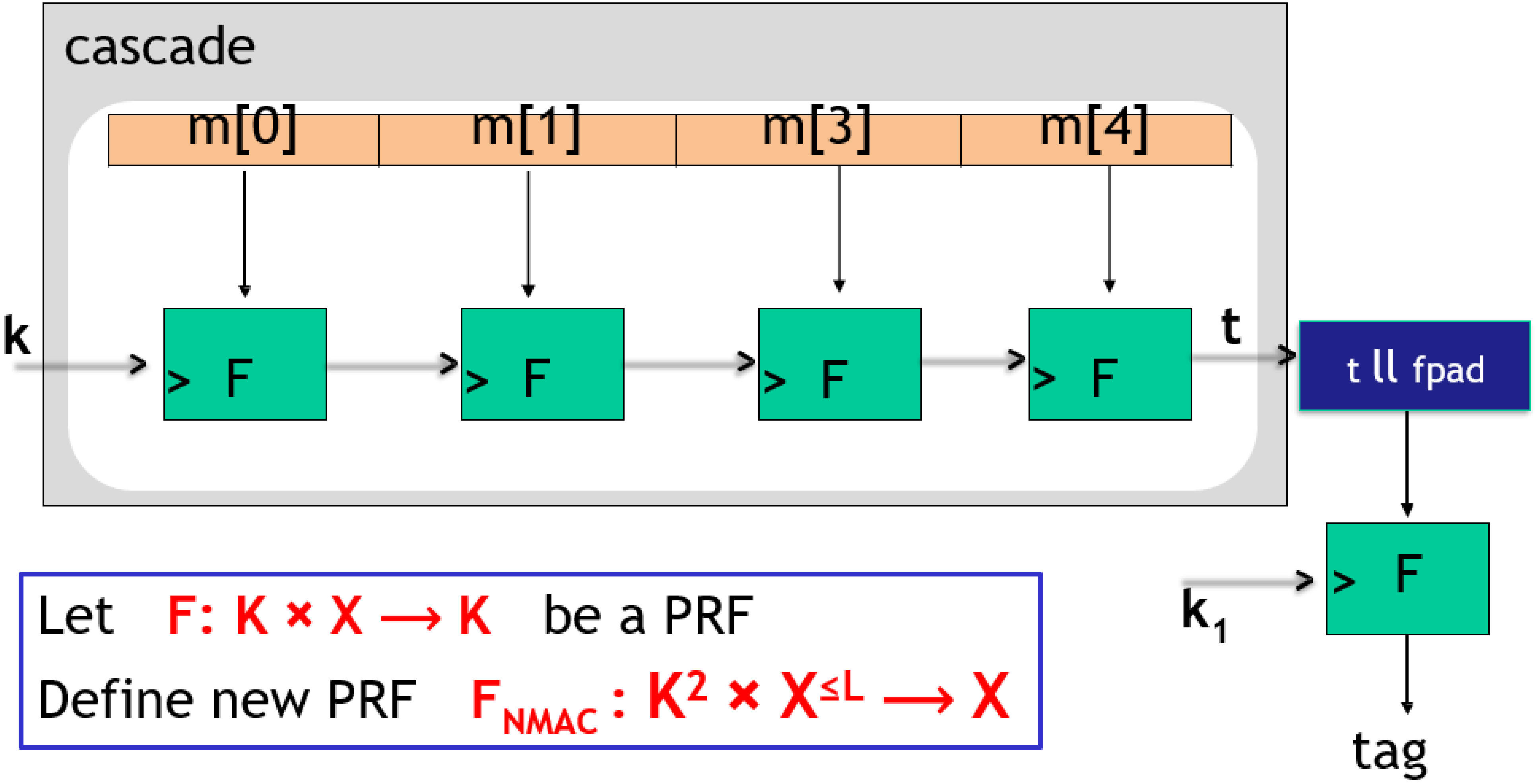
Encrypted CBC-MAC



Let $F: K \times X \longrightarrow X$ be a PRP

Define new PRF $F_{ECBC} : K^2 \times X^{\leq L} \longrightarrow X$

NMAC (nested MAC)



Let $F: K \times X \longrightarrow K$ be a PRF

Define new PRF $F_{NMAC} : K^2 \times X^{\leq L} \longrightarrow X$

eCBC-MAC and NMAC analysis

Theorem: For any $L>0$,

For every eff. q-query PRF adv. A attacking F_{ECBC} or F_{NMAC} there exists an eff. adversary B s.t.:

$$\mathrm{Adv}_{PRF}[A, F_{ECBC}] \leq \mathrm{Adv}_{PRP}[B, F] + 2\,q^2 / |X|$$

$$\mathrm{Adv}_{PRF}[A, F_{NMAC}] \leq q \cdot L \cdot \mathrm{Adv}_{PRF}[B, F] + q^2 / 2|K|$$

CBC-MAC is secure as long as $q << |X|^{1/2}$

NMAC is secure as long as $q << |K|^{1/2}$ (2^{64} for AES-128)

Suppose we want $\mathrm{Adv}_{PRF}[A, F_{ECBC}] \leq 1/2^{32}$ $\Leftarrow$ $q^2 / |X| < 1/ 2^{32}$

▯ AES: $|X| = 2^{128}$ $\Rightarrow$ $q < 2^{48}$

So, after 2^{48} messages must, must change key

▯ 3DES: $|X| = 2^{64}$ $\Rightarrow$ $q < 2^{16}$

Comparison

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

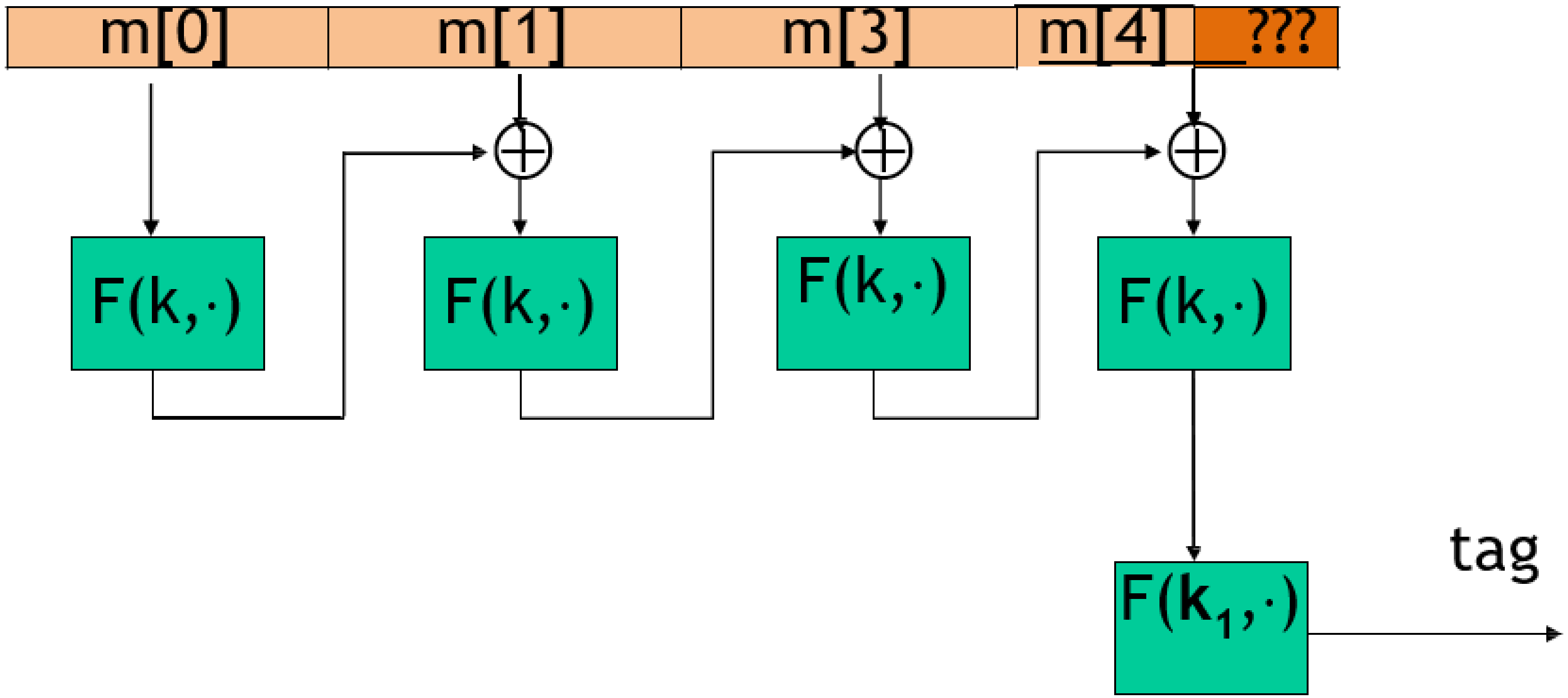
- CCM encryption mode (used in 802.11i)
- NIST standard called CMAC

NMAC not usually used with AES or 3DES

- Main reason: need to change AES key on every block

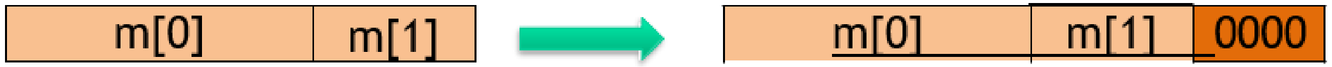
  requires re-computing AES key expansion
- But NMAC is the basis for a popular MAC called HMAC (next)

MAC padding



MAC padding

Bad idea: pad m with 0’s

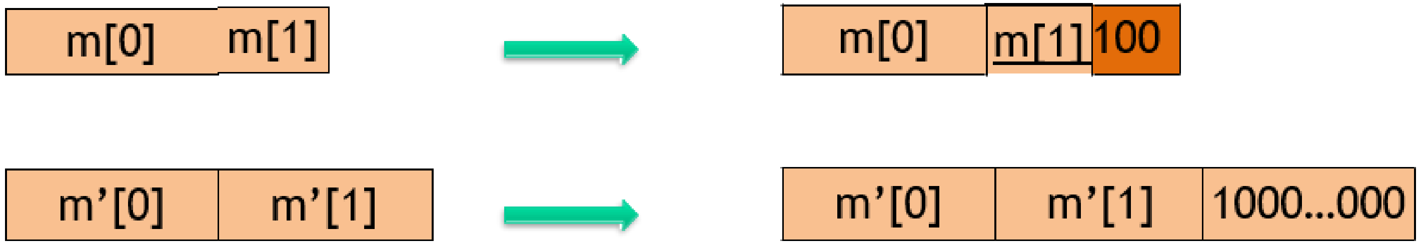


For security, padding must be invertible !

$$m_0 \neq m_1 \quad \Rightarrow \quad \text{pad}(m_0) \neq \text{pad}(m_1)$$

ISO: pad with “1000…00”. Add new dummy block if needed.

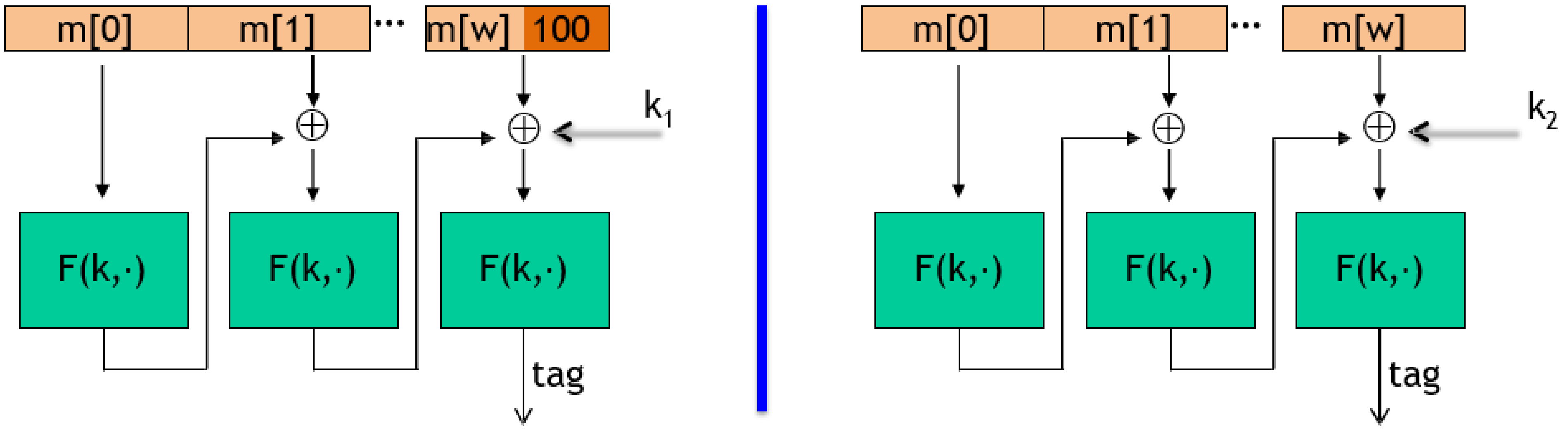
- The “1” indicates beginning of pad.



CMAC (NIST standard)

Variant of CBC-MAC where key = (k, k_1, k_2)

- No final encryption step (extension attack thwarted by last keyed xor)
- No dummy block (ambiguity resolved by use of k_1 or k_2)



HMAC……

Based on hash function.

Wait!!!

Hash

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

Hash function -> **“one-way function”**

A cryptographic hash function must be able to withstand all known types of cryptanalytic attack. In theoretical cryptography, the security level of a cryptographic hash function has been defined using the following properties:

- **Pre-image resistance**

Given a hash value h it should be difficult to find any message m such that h = hash(m). This concept is related to that of a one-way function. Functions that lack this property are vulnerable to preimage attacks.

- **Second pre-image resistance**

Given an input m1, it should be difficult to find a different input m2 such that hash(m1) = hash(m2). Functions that lack this property are vulnerable to second-preimage attacks.

- **Collision resistance**

It should be difficult to find two different messages m1 and m2 such that hash(m1) = hash(m2). Such a pair is called a cryptographic hash collision. This property is sometimes referred to as strong collision resistance. It requires a hash value at least twice as long as that required for pre-image resistance; otherwise collisions may be found by a birthday attack.

Common hash functions:

- MD5: output 128 bits X (broken) X
- SHA-1: output 160 bits X (broken) X
- SHA-256: output 256 bits
- SHA-512: output 512 bits
- Whirpool: output 512 bits

Collision Resistance

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:

$$\mathbf{Adv_{CR}[A,H] = Pr[\text{ A outputs collision for H}]}$$

is “negligible”.

Example: SHA-256 (outputs 256 bits)

The birthday paradox

Let $r_1, \ldots, r_n \in \{1,\ldots,B\}$ be indep. identically distributed integers.

Thm: when $n = 1.2 \times B^{1/2}$ then $\Pr\left[\ \exists i \neq j:\ r_i = r_j\ \right] \geq \frac{1}{2}$

Taking it as a “birthday problem” (B=365) -> n=23 then

Pr[anyone same birthday] ≥ ½

(proof in https://www.coursera.org/learn/crypto/lecture/pyR4I/generic-birthday-attack)

Generic attack

H: M ∈$\{0,1\}^n$. Collision finding algorithm:

1. Choose **$2^{n/2}$** random elements in M: $m_1, \ldots, m_{2^{n/2}}$
2. For i = 1, …, $2^{n/2}$ compute $t_i = H(m_i)$ ∈$\{0,1\}^n$
3. Look for a collision $(t_i = t_j)$. If not found, got back to step 1.

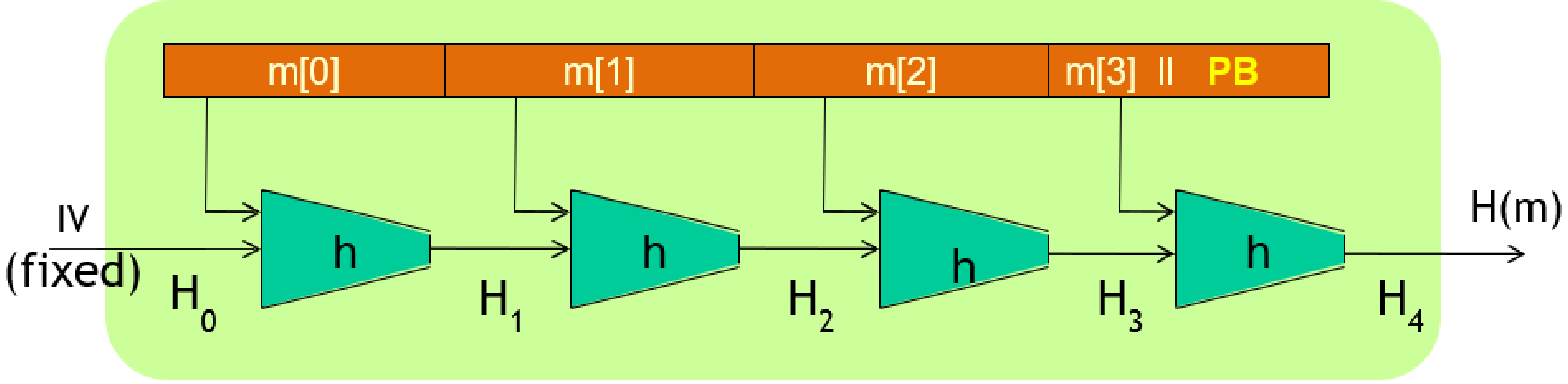
Expected number of iteration ≈ 2

Running time: **$O(2^{n/2})$** (space $O(2^{n/2})$)

AMD Opteron, 2.2 GHz (Linux)

function	digest size (bits)	Speed (MB/sec)	generic attack time
SHA-1	160	153	2^{80}
SHA-256	256	111	2^{128}
SHA-512	512	99	2^{256}
Whirlpool	512	57	2^{256}

Merkle-Damgard construction



Given $h: T \times X \longrightarrow T$ (compression function)

we obtain $H: X^{\leq L} \longrightarrow T$. H_i - chaining variables

PB: padding block

$1000\ldots0 \;\; \text{ll} \;\; \underbrace{\text{msg len}}_{\text{64 bits}}$

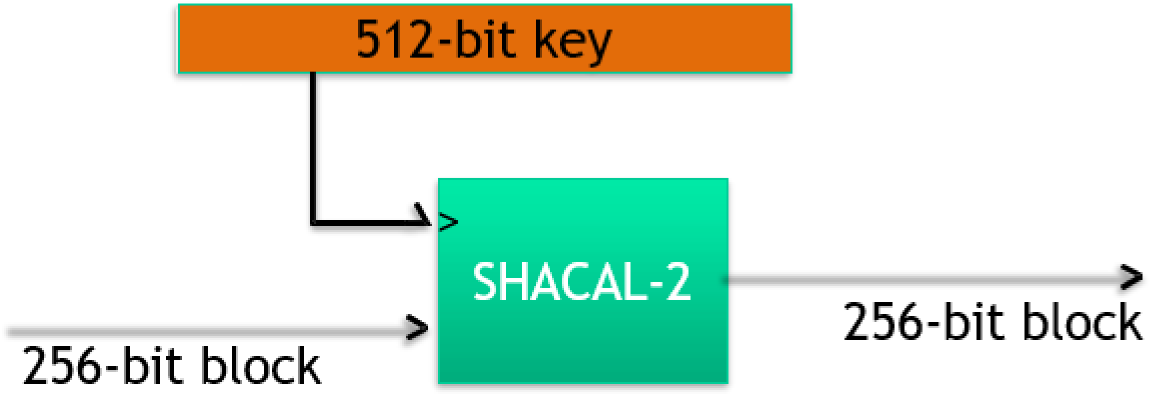
If no space for PB
add another block

Thm: if h is collision resistant then so is H.

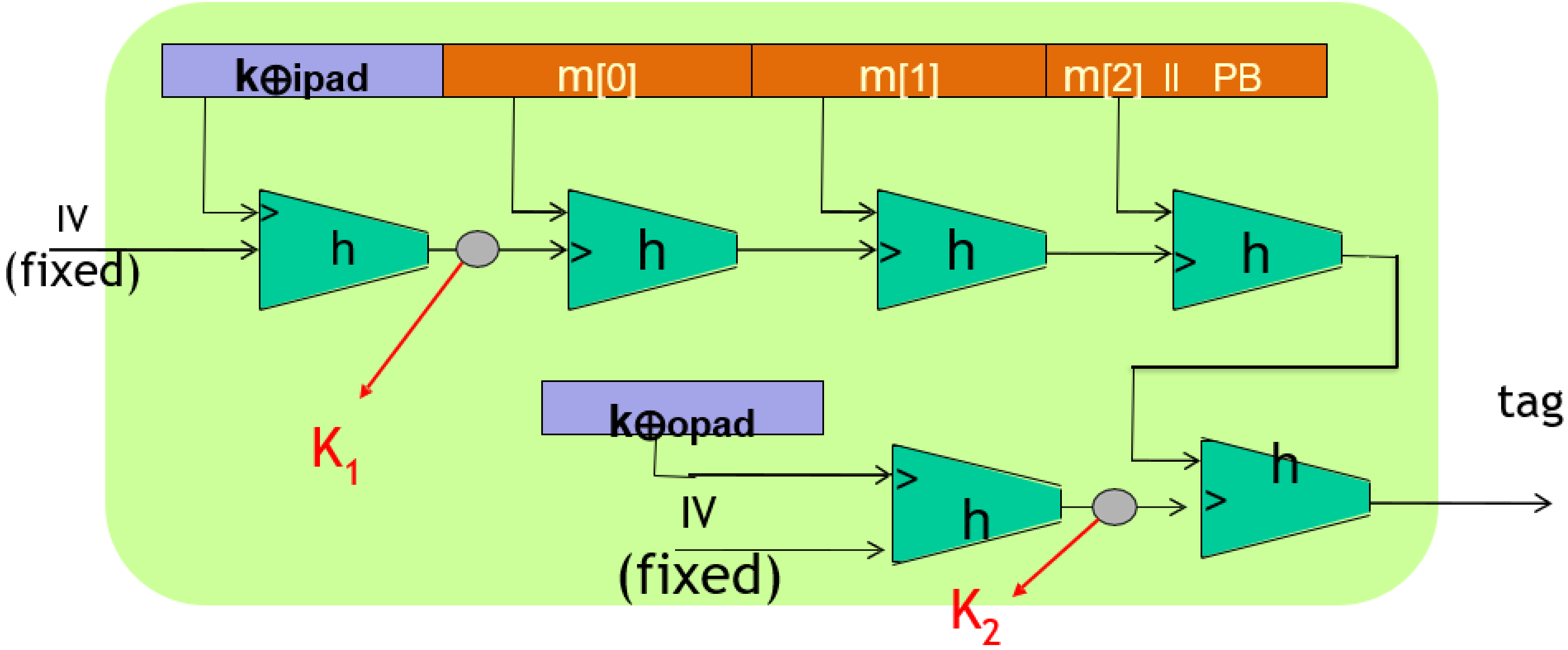
Proof in: https://www.coursera.org/learn/crypto/lecture/Hfnu9/the-merkle-damgard-paradigm

Case study: SHA-256

- Merkle-Damgard function
- Davies-Meyer compression function
- Block cipher: SHACAL-2



HMAC



Similar to the NMAC PRF.

main difference: the two keys k_1, k_2 are dependent

HMAC properties

Built from a black-box implementation of SHA-256.

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 ($q << |T|^{½}$)

In TLS: must support HMAC-SHA1-96

<u>With HMAC is a Secure MAC with the sole assumption of h(.,.) is a secure PRF.</u>

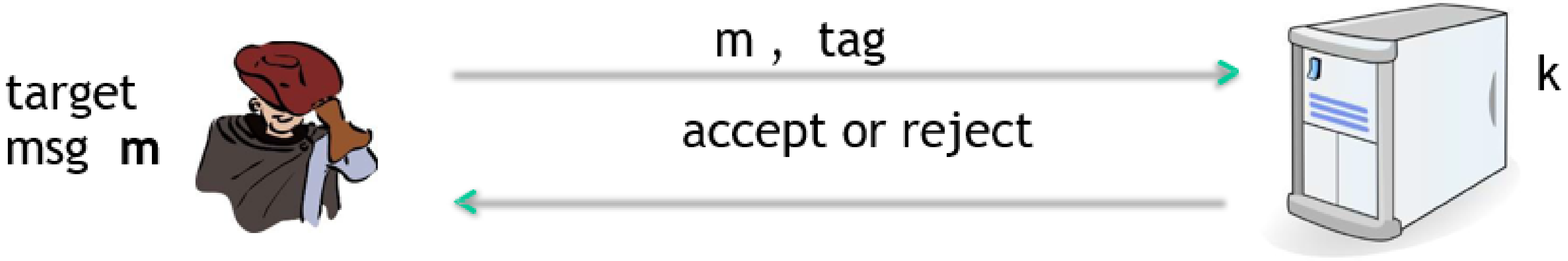
Verification timing attacks!

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



target
msg **m**

m , tag

accept or reject

k

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

Verification timing attacks!

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.

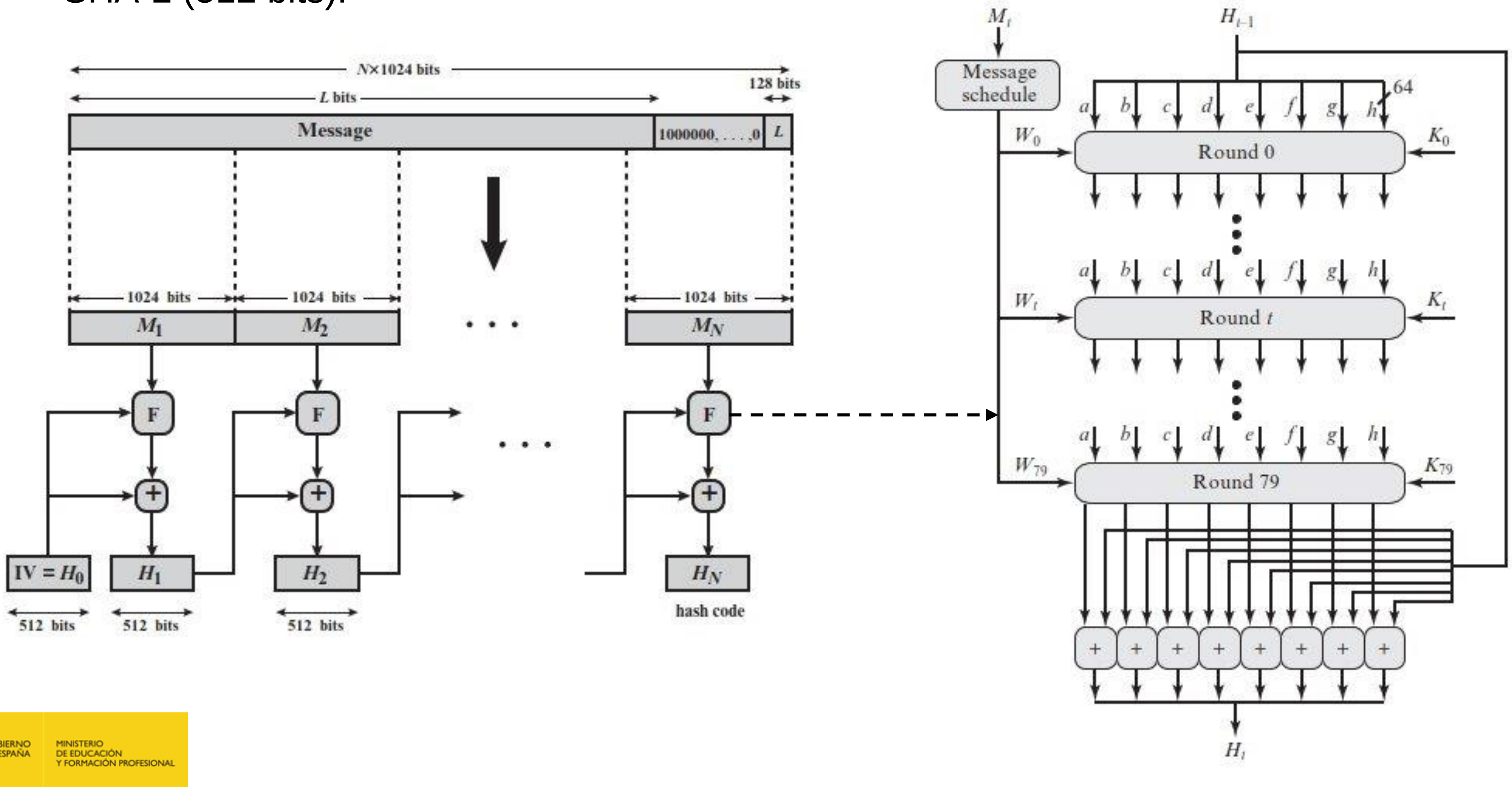
Defense 2

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

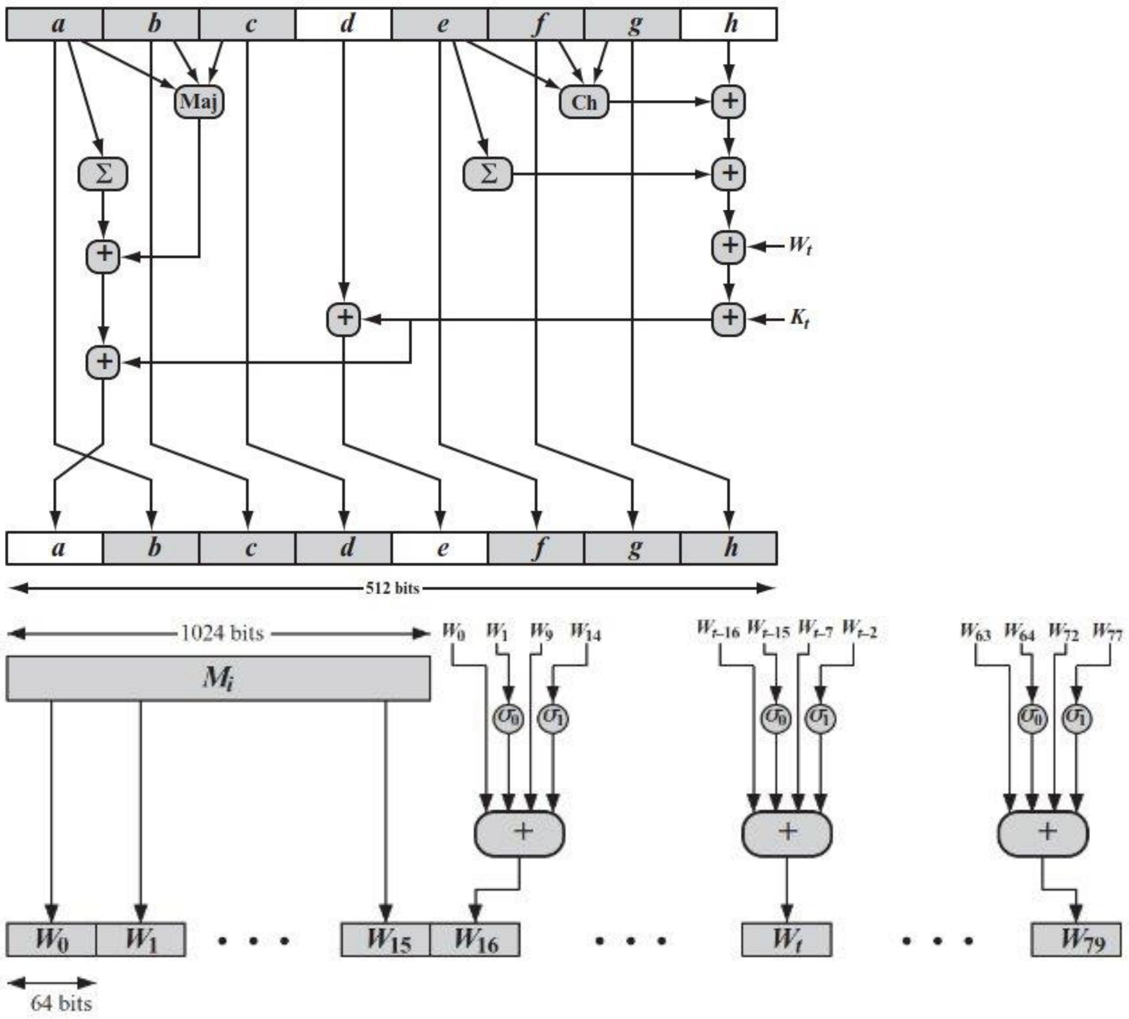
Attacker doesn’t know values being compared

Secure Hash Algorithm (SHA)

- In 1993, NIST published SHA-0 and updated it to SHA-1 in 1995, due to discovered weakness
- In 2002, NIST defined three new versions of SHA (SHA-2) as a replacement of SHA-1
- SHA-2 (512 bits):



Secure Hash Algorithm (SHA-2)


a b c d e f g h
Maj
Ch
Σ
Σ
W_t
K_t
a b c d e f g h
512 bits
1024 bits
M_i
W_0 W_1 W_9 W_{14}
W_{t-16} W_{t-15} W_{t-7} W_{t-2}
W_{63} W_{64} W_{72} W_{77}
σ_0 σ_1
+
W_0 W_1 . . . W_{15} W_{16} . . . W_t . . . W_{79}
64 bits

t = step number; $0 \le t \le 79$

$\text{Ch}(e, f, g) = (e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g)$

$\text{Maj}(a, b, c) = (a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c)$

$(\Sigma_0^{512} a) = \text{ROTR}^{28}(a) \oplus \text{ROTR}^{34}(a) \oplus \text{ROTR}^{39}(a)$

$(\Sigma_1^{512} e) = \text{ROTR}^{14}(e) \oplus \text{ROTR}^{18}(e) \oplus \text{ROTR}^{41}(e)$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

W_t = a 64-bit word derived from the current 1024-bit input block

K_t = a 64-bit additive constant

$+$ = addition modulo 2^{64}

$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

where

$\sigma_0^{512}(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x)$

$\sigma_1^{512}(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x)$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

$\text{SHR}^n(x)$ = right shift of the 64-bit argument x by n bits with padding by zeros on the left

LABORATORY

- Laboratory_02: Python hashing
- Laboratory_03: Hashcat