MATH/CMSC 456 :: UPDATED COURSE INFO

Instructor: Gorjan Alagic (galagic@umd.edu); ATL 3102, office hours: by appointment

Textbook: *Introduction to Modern Cryptography*, Katz and Lindell;

Webpage: alagic.org/cmsc-456-cryptography-spring-2020/ (slides, **reading posted here**);

Piazza: piazza.com/umd/spring2020/cmsc456

ELMS: active, slides and reading posted there, **homework 2 due midnight Thursday.**

Gradescope: active, access through ELMS.

TAs (Our spot: shared open area across from **AVW 4166**)

- Elijah Grubb (egrubb@cs.umd.edu) 11am-12pm TuTh (AVW);
- Justin Hontz (jhontz@terpmail.umd.edu) 1pm-2pm MW (AVW);

Additional help:

- Chen Bai (cbai1@terpmail.umd.edu) 3:30-5:30pm Tu (2115 ATL – inside **JQI**)
- Bibhusa Rawal (bibhusa@terpmail.umd.edu) 3:30-5:30pm Th (2115 ATL – inside **JQI**)

RECAP: HASH FUNCTIONS

What are hash functions?

A hash function is just a function w

$\mathcal{H}: \{0,1\}^m \to \{0,1\}^\ell$ for $\ell < m$.

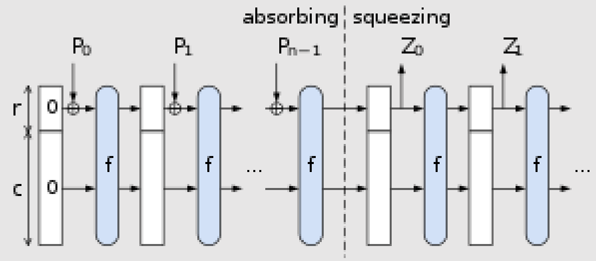
In practice:

- $\mathcal{H}$ is implementable with a very f
- this algorithm is completely pub
- ℓ is a fixed constant (e.g., 128) w

How do you design them?

- a bit like PRGs: part art, part science;
- analysis is difficult.

SHA3
2015



The basic block permutation function consists of $12 + 2\ell$ rounds of five steps:

θ (theta)
Compute the parity of each of the $5w$ (320, when $w = 64$) 5-bit columns, and exclusive-or that into two nearby columns in a regular pattern. To be precise, $a[i][j][k] \leftarrow a[i][j][k] \oplus \text{parity}(a[0...4][j-1][k]) \oplus \text{parity}(a[0...4][j+1][k-1])$

ρ (rho)
Bitwise rotate each of the 25 words by a different triangular number 0, 1, 3, 6, 10, 15, ... To be precise, $a[0][0]$ is not rotated, and for all $0 \le t < 24$, $a[i][j][k] \leftarrow a[i][j][k-(t+1)(t+2)/2]$, where $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}^t \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

π (pi)
Permute the 25 words in a fixed pattern. $a[j][2i+3j] \leftarrow a[i][j]$.

χ (chi)
Bitwise combine along rows, using $x \leftarrow x \oplus (\neg y \,\&\, z)$. To be precise, $a[i][j][k] \leftarrow a[i][j][k] \oplus (\neg a[i][j+1][k] \,\&\, a[i][j+2][k])$. This is the only non-linear operation in SHA-3.

ι (iota)
Exclusive-or a round constant into one word of the state. To be precise, in round n, for $0 \le m \le \ell$, $a[0][0][2^m-1]$ is XORed with bit $m + 7n$ of a degree-8 LFSR sequence. This breaks the symmetry that is preserved by the other steps.

MD5
1992

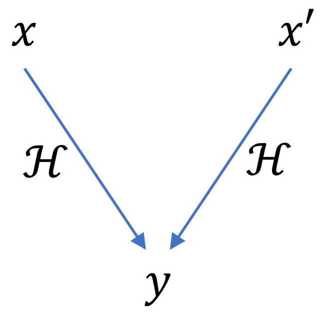
```
// Note: All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int s[64], K[64]
var int i

// s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }

// Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63 do
    K[i] := floor(2^32 × abs(sin(i + 1)))
end for
// Initialize variables:
var int a0 := 0x67452301   // A
var int b0 := 0xefcdab89   // B
var int c0 := 0x98badcfe   // C
var int d0 := 0x10325476   // D

// Pre-processing: adding a single 1 bit
append "1" bit to message

        F := C xor (B or (not D))
        g := (7×i) mod 16
      // Be wary of the below definitions of a,b,c,d
      F := F + A + K[i] + M[g]  // M[g] must be a 32-bits block
      A := D
      D := C
      C := B
      B := B + leftrotate(F, s[i])
    end for
    // Add this chunk's hash to result so far:
    a0 := a0 + A
    b0 := b0 + B
    c0 := c0 + C
    d0 := d0 + D
end for

var char digest[16] := a0 append b0 append c0 append d0 // (Output is in little-endian)

// leftrotate function definition
leftrotate (x, c)
    return (x << c) binary or (x >> (32-c));
```

RECAP: HASH FUNCTIONS

What are they good for?

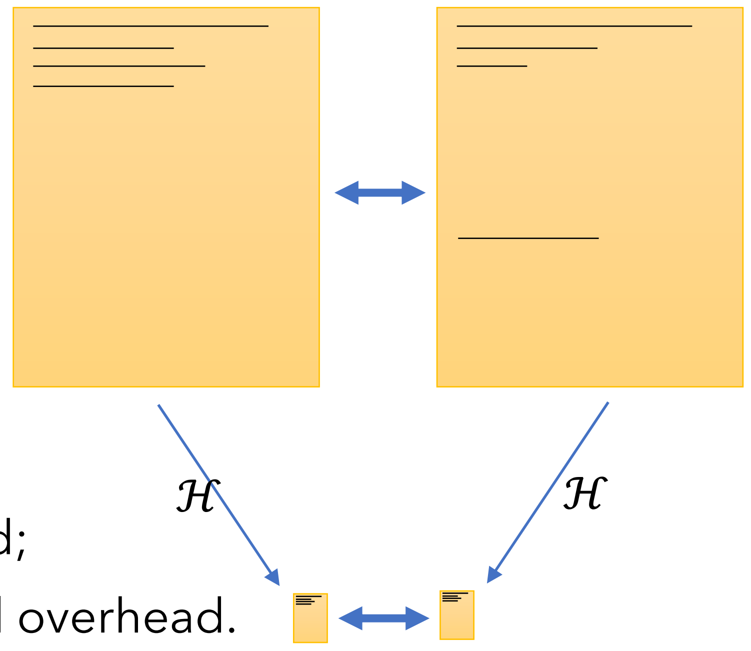
They compress their input: $\mathcal{H}: \{0,1\}^m \to \{0,1\}^\ell$ for $\ell < m$.

So obviously, some $y \in \{0,1\}^\ell$ have a *lot* of preimages: at least $2^{m-\ell}$.



But, for a well-designed hash function:

- h *seems to be* 1-to-1;
- *typically* hard to find two inputs x, x' with the same **digest** $\mathcal{H}(x)$;
- *typically* also hard: given a digest y, find an input x such that $\mathcal{H}(x) = y$.



This is why they are used, e.g., in **git**:

- files are not compared directly;
- instead, a hash (digest) of each file is stored, and the hashes are compared;
- this allows for all sorts of integrity checks without a massive computational overhead.

They're also used, e.g., in **blockchains** (e.g., in Bitcoin) for similar reasons.

RECAP: HASH FUNCTIONS, FORMALLY

We will think about *keyed* hash functions.

Definition. A hash function $\mathcal{H}$ is a polynomial-time computable function family

$$\mathcal{H}: \{0,1\}^d \times \{0,1\}^* \to \{0,1\}^\ell$$

equipped with a PPT algorithm $\mathbf{KeyGen}$ which, on input 1^n, outputs a key $s \in \{0,1\}^d$.

We write $\mathcal{H}^s(x) \coloneqq \mathcal{H}(s, x)$.

How to use it?

Typically:

1. Sample $s \leftarrow \mathbf{KeyGen}(1^n)$;
2. Make s public to everyone;
3. Now anyone can evaluate $\mathcal{H}^s$ on any string x and get the hash digest $\mathcal{H}^s(x)$.

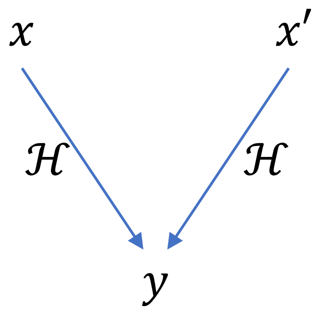
Why? In practice, anyone can look up hash function spec

RECAP: COLLISION-RESISTANCE

What security properties do we want?

There are many. An important one: *collision-resistance*.

- as we saw, every hash function is necessarily *many-to-one*;
- but in a **good** hash function, it should be hard to find inputs with the same digest.



If this sounds impossible:

Think about a random function $\boldsymbol{R}: \{0,1\}^{2n} \rightarrow \{0,1\}^n$

- it's true that each $y \in \{0,1\}^n$ has (roughly) 2^n preimages;
- let $X_y = \{x \in \{0,1\}^{2n} : \boldsymbol{R}(x) = y\}$ be the set of preimages of y;
- Note: X_y is a random subset of size 2^n in a set of size 2^{2n};
- In other words: for any z, $\Pr_{\boldsymbol{R}}[z \in X_y] \approx 2^{-n}$.

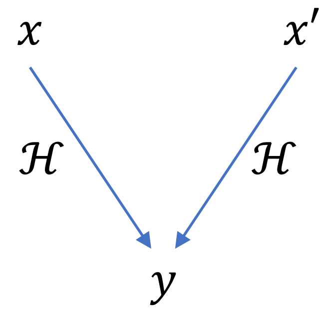
So, there are indeed functions for which it's hard to find preimages and collisions.

(Actually, in a certain sense, *most* functions have this property.)

RECAP: COLLISION-RESISTANCE

How to define?

As usual: with a game!



Let $\Pi = (\mathbf{KeyGen}, \mathcal{H})$ be a hash function, and $\boldsymbol{A}$ an algorithm.

The game HashColl$(\Pi, \boldsymbol{A})$ proceeds as follows:

1. Generate key: $s \leftarrow \mathbf{KeyGen}$;
2. $\boldsymbol{A}$ receives s and outputs $x, x' \in \{0,1\}^*$;

We say $\boldsymbol{A}$ wins if $\mathcal{H}^s(x) = \mathcal{H}^s(x')$ and $x \neq x'$.

Definition. A hash function $\Pi = (\mathbf{KeyGen}, \mathcal{H})$ is **collision-resistant** if, for every PPT adversary $\boldsymbol{A}$,

$$\Pr[\boldsymbol{A} \text{ wins HashColl}(\Pi, \boldsymbol{A})] \leq \text{negl}(n).$$

RECAP: HASH-and-MAC

What is collision resistance good for?

Authentication!

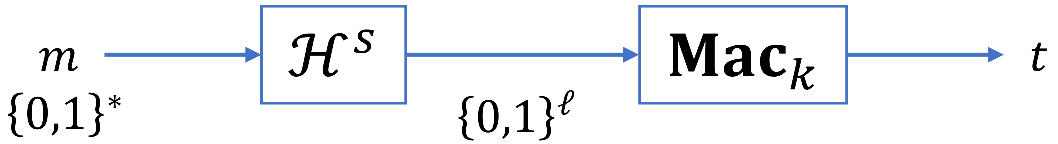
Construction (Hash-and-MAC). Let

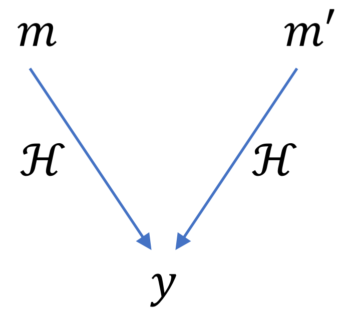
- $\Pi = (\mathbf{KeyGen}, \mathbf{Mac})$ be a fixed-length message authentication code (MAC), and
- $\Pi_{\mathrm{H}} = (\mathbf{KeyGen}_{\mathrm{H}}, \mathcal{H})$ be a hash function.

Define an arbitrary-length deterministic MAC $\Pi' = (\mathbf{KeyGen}', \mathbf{Mac}')$ as follows:

- (key generation) $\mathbf{KeyGen}'$: on input 1^n, outputs $k' \leftarrow \left(\mathbf{KeyGen}(1^n), \mathbf{KeyGen}_{\mathrm{H}}(1^n)\right)$.
- (tag generation) $\mathbf{Mac}'$: on key (k, s) and message m, outputs $t \coloneqq \mathbf{Mac}_k(\mathcal{H}^s(m))$.

In pictures:





RECAP: HASH-and-MAC

Construction (Hash-and-MAC). Let

- $\Pi = (\mathbf{KeyGen}, \mathbf{Mac})$ be a fixed-length message authentication code (MAC), and
- $\Pi_{\mathrm{H}} = (\mathbf{KeyGen}_{\mathrm{H}}, \mathcal{H})$ be a hash function.

Define an arbitrary-length deterministic MAC $\Pi' = (\mathbf{KeyGen}', \mathbf{Mac}')$ as follows:

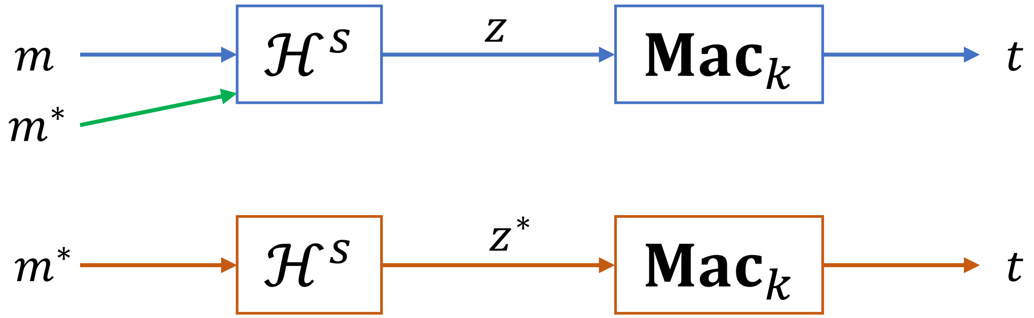
- (key generation) $\mathbf{KeyGen}'$: on input 1^n, outputs $k' \leftarrow \left(\mathbf{KeyGen}(1^{\mathrm{n}}), \mathbf{KeyGen}_{\mathrm{H}}(1^n)\right)$.
- (tag generation) $\mathbf{Mac}'$: on key (k, s) and message m, outputs $t \coloneqq \mathbf{Mac}_k(\mathcal{H}^s(m))$.

Theorem. If Π is an EUF-CMA fixed-length MAC, and Π_{H} is a collision-resistant hash function, then the Hash-and-MAC construction Π' is an EUF-CMA arbitrary-length MAC.

Proof idea:

If adversary forges on message m^* then either/or:

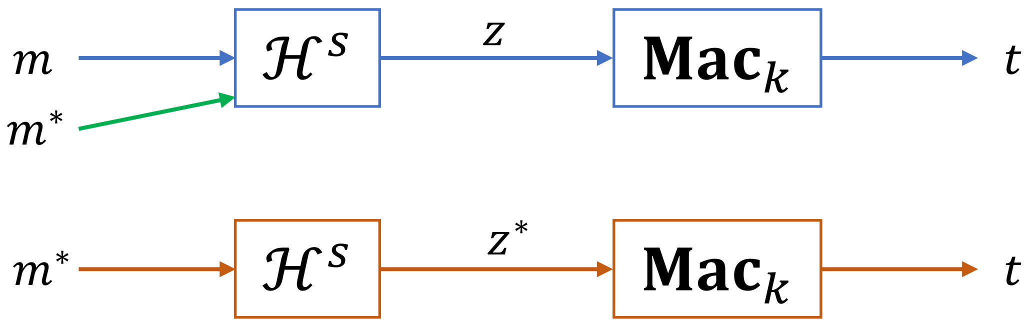
1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



RECAP: HASH-and-MAC

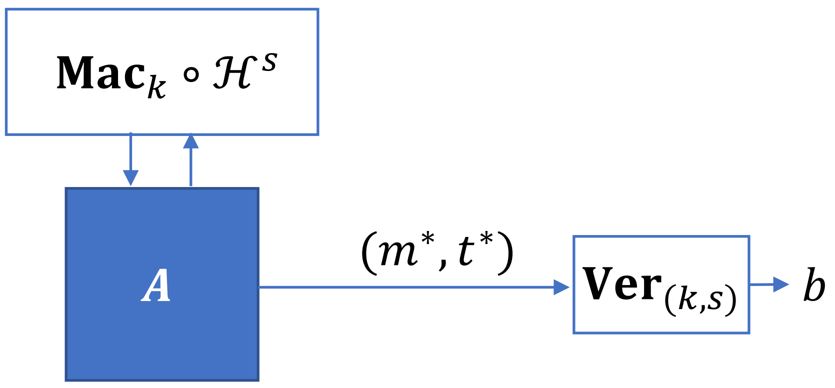
Proof idea: If forgery on m^* then either/or:

1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on** Π**!**



Recall EUF-CMA and MacForge experiment.

- let Q be the set of queries made by $\boldsymbol{A}$, and (m^*, t^*) its output;
- let $\boldsymbol{E}$ be the green event: $\exists m \in Q$ such that $\mathcal{H}^s(m) = \mathcal{H}^s(m^*)$;



Calculate:

$\Pr[A \text{ wins MacForge}(\Pi')] =$

$= \Pr[A \text{ wins MacForge}(\Pi') \wedge \boldsymbol{E}] + \Pr[A \text{ wins MacForge}(\Pi') \wedge \bar{\boldsymbol{E}}]$

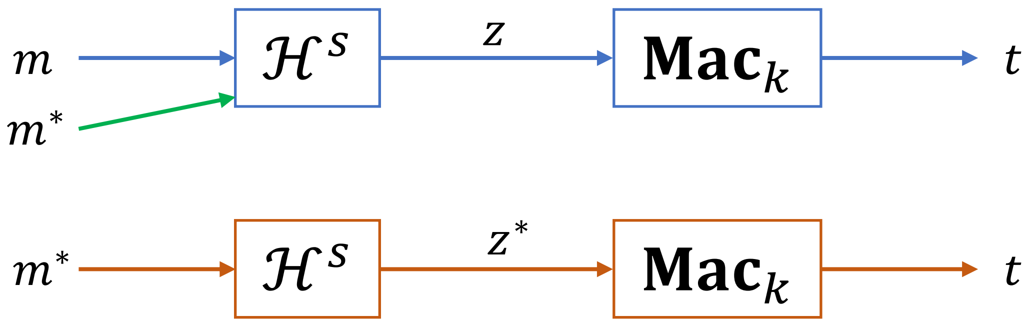
$\leq \Pr[\boldsymbol{E}] + \Pr[A \text{ wins MacForge}(\Pi') \wedge \bar{\boldsymbol{E}}]$.

We will show that both of these terms are negligible. How?

RECAP: HASH-and-MAC

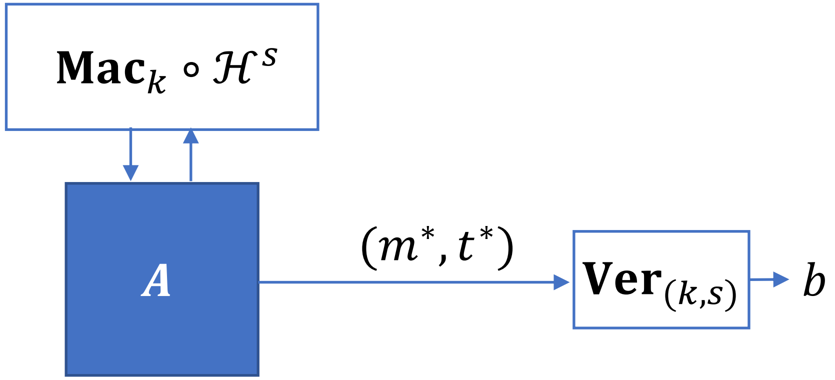
Proof idea: If forgery on m^* then either/or:

1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



Controlling probability of $\boldsymbol{E}$:

- $\boldsymbol{E}$ is the green event: $\exists m \in Q$ such that $\mathcal{H}^s(m) = \mathcal{H}^s(m^*)$;
- want to show: $\Pr[\boldsymbol{E}] \leq \mathrm{negl}(n)$.
- how? Well, suppose it's not, and consider this algorithm:



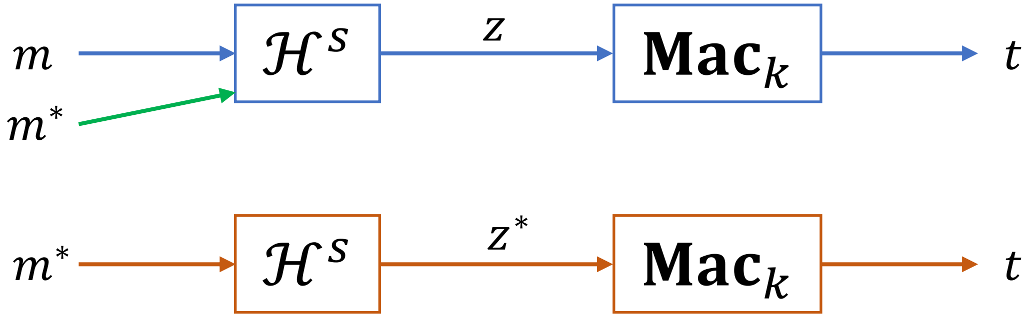
1. Receive hash key s as input. Sample $\mathbf{Mac}$ key k;
2. Run $\boldsymbol{A}$ with oracle $\mathbf{Mac}_k \circ \mathcal{H}^s$;
3. Output m^* and a random $m \in Q$.

Check: the probability that this algorithm finds a collision in $\mathcal{H}^s$ is at least $\Pr[\boldsymbol{E}]\,/|Q|$.

RECAP: HASH-and-MAC

Proof idea: If forgery on m^* then either/or:

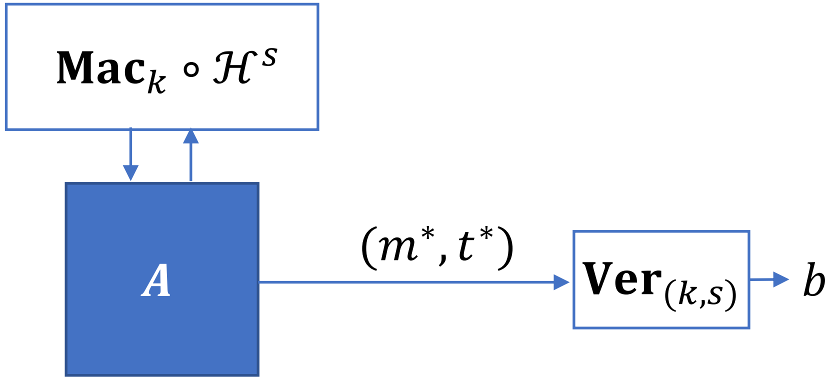
1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



What's left:

Control $\Pr[\boldsymbol{A} \text{ wins MacForge}(\Pi') \wedge \overline{\boldsymbol{E}}]$.

- what is this quantity?
- probability that $\boldsymbol{A}$ wins the forgery game…
- … **and** for all queried m, $\mathcal{H}^s(m) \neq \mathcal{H}^s(m^*)$.



Stated a bit differently:

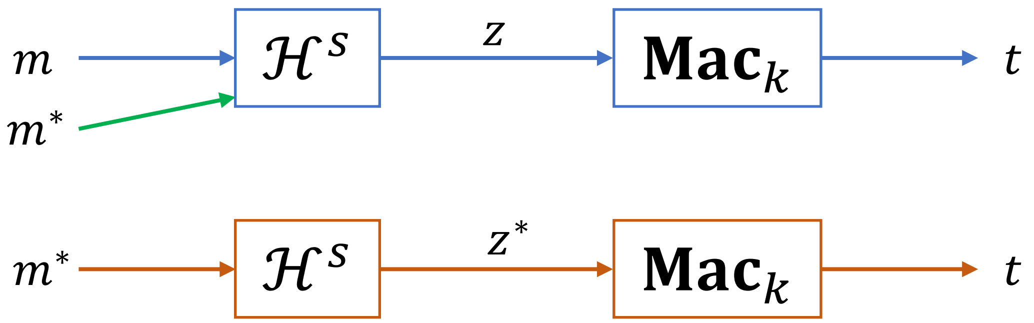
- probability that $\boldsymbol{A}$ wins the forgery game…
- … **and** for all *inputs z to* $\mathbf{Mac}_k$ *oracle*, $z \neq z^* := \mathcal{H}^s(m^*)$.

Point: in this case, we should be able to win a MacForge game against Π!

RECAP: HASH-and-MAC

Proof idea: If forgery on m^* then either/or:

1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



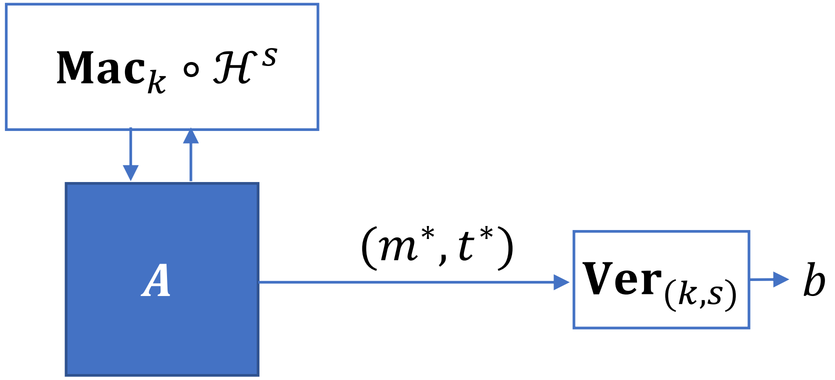
What's left:

Control $\Pr[\boldsymbol{A} \text{ wins MacForge}(\Pi') \wedge \bar{\boldsymbol{E}}]$. If it's large…

… then we should be able to win a MacForge game against Π!

Here's how:

1. Receive $\mathbf{Mac}_k$ oracle. Sample hash key s;
2. When queried with $m \in \{0,1\}^*$ …
   i. Hash it: $z := \mathcal{H}^s(\mathrm{m})$;
   ii. MAC it (using oracle): $t := \mathbf{Mac}_k(\mathrm{z})$; return t.
3. When $\boldsymbol{A}$ outputs m^*, output $\mathcal{H}^s(m^*)$.

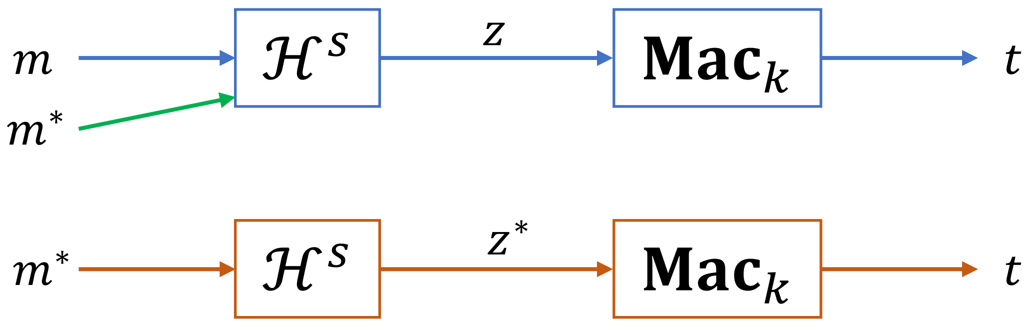


Check: probability this wins MacForge versus Π is exactly $\Pr[\boldsymbol{A} \text{ wins MacForge}(\Pi') \wedge \bar{\boldsymbol{E}}]$.

RECAP: HASH-and-MAC

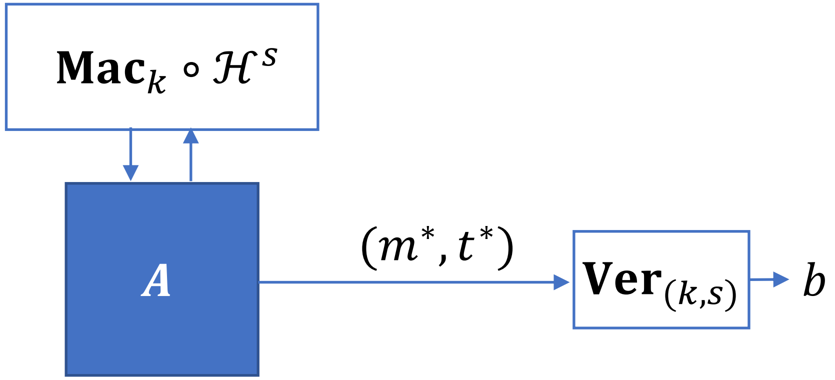
Proof idea: If forgery on m^* then either/or:

1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



Recall EUF-CMA and MacForge experiment.

- let Q be the set of queries made by $\boldsymbol{A}$, and (m^*, t^*) its output;
- let $\boldsymbol{E}$ be the green event: $\exists m \in Q$ such that $\mathcal{H}^s(m) = \mathcal{H}^s(m^*)$;



Calculate:

$$\Pr[A \text{ wins MacForge}(\Pi')] =$$
$$= \Pr[A \text{ wins MacForge}(\Pi') \wedge \boldsymbol{E}] + \Pr[A \text{ wins MacForge}(\Pi') \wedge \overline{\boldsymbol{E}}]$$
$$\leq \Pr[\boldsymbol{E}] + \Pr[A \text{ wins MacForge}(\Pi') \wedge \overline{\boldsymbol{E}}]$$
$$\leq \text{negl}(n) + \text{negl}(n) \leq \text{negl}(n).$$

$\square$

HASH FUNCTIONS continued

Reading: (p.156-160, 174-181)

HASH FUNCTIONS continued

More on hash functions:

1. Keyed **vs** unkeyed;

2. Arbitrary-length inputs **vs** fixed-length inputs;

3. Hash functions as random oracles.

HASH FUNCTIONS, FORMALLY

Keyed vs unkeyed hash functions.

I did something funny…

- I first cited some public hash functions (like MD5 and SHA3)…
- … clearly, these hashes do not have a "key." They are fixed algorithms!
- but then I defined a hash function to have a key!
- and that key seems pretty critical!

This is pretty standard in cryptography. Why?

Why do we use keyed hash functions in formal reasoning?

HASH FUNCTIONS, FORMALLY

Why keyed hash functions?

1. There actually do exist keyed hash functions, and they are interesting!
   (maybe we will get to them later in the course.)
2. An annoying technicality:

- technically, an unkeyed hash is a *completely fixed (and hence known) object.*
- this means that, if we were to strictly follow our theoretical formalism…
- … algorithms could have *hard-coded* properties of the hash: collisions, preimages, etc.

This is analogous to this juxtaposition:

Fix a boolean formula φ of size 10^{200}. Is it NP-hard to determine if φ is satisfiable? **NO**.

Is it NP-hard to determine, given an arbitrary formula ψ as input, if ψ is satisfiable? **YES**.

In spirit: a keyed hash models the fact that, in reality,

… nobody really "knows" anything about SHA3 except a few of its values!

HASH FUNCTIONS continued

More on hash functions:

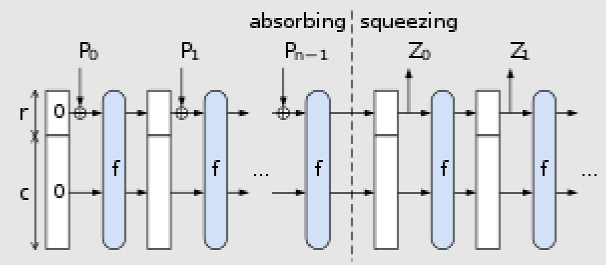
1. Keyed **vs** unkeyed;

2. Arbitrary-length inputs **vs** fixed-length inputs;

3. Hash functions as random oracles.

HASH FUNCTIONS: ARBITRARY-LENGTH INPUTS

SHA3
2015



The basic block permutation function consists of $12 + 2\ell$ rounds of five steps:

θ (theta)
Compute the parity of each of the $5w$ (320, when $w = 64$) 5-bit columns, and exclusive-or that into two nearby columns in a regular pattern. To be precise, $a[i][j][k] \leftarrow a[i][j][k] \oplus \text{parity}(a[0...4][j-1][k]) \oplus \text{parity}(a[0...4][j+1][k-1])$

ρ (rho)
Bitwise rotate each of the 25 words by a different triangular number 0, 1, 3, 6, 10, 15, ... To be precise, $a[0][0]$ is not rotated, and for all $0 \le t < 24$, $a[i][j][k] \leftarrow a[i][j][k-(t+1)(t+2)/2]$, where $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}^t \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

π (pi)
Permute the 25 words in a fixed pattern. $a[j][2i+3j] \leftarrow a[i][j]$.

χ (chi)
Bitwise combine along rows, using $x \leftarrow x \oplus (\neg y \,\&\, z)$. To be precise, $a[i][j][k] \leftarrow a[i][j][k] \oplus (\neg a[i][j+1][k] \,\&\, a[i][j+2][k])$. This is the only non-linear operation in SHA-3.

ι (iota)
Exclusive-or a round constant into one word of the state. To be precise, in round n, for $0 \le m \le \ell$, $a[0][0][2^m-1]$ is XORed with bit $m + 7n$ of a degree-8 LFSR sequence. This breaks the symmetry that is preserved by the other steps.

We've been assuming…

… hash functions can take in arbitrary strings.

Pretty important for Hash-and-MAC!

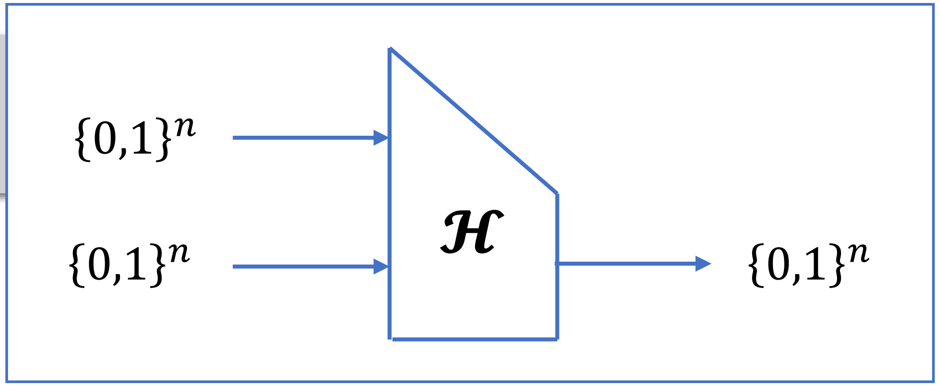
- in practice, that doesn't come for free;
- we first construct a fixed-length hash, like this:

$$\mathcal{H}: \{0,1\}^{2n} \to \{0,1\}^n$$

- maybe for a fixed n (e.g., $n = 128$)…
- … and then apply a transformation that enables arbitrary-length inputs.

The simplest is the **Merkle-Damgård transform**.

HASH FUNCTIONS: ARBITRARY-LENGTH INPUTS



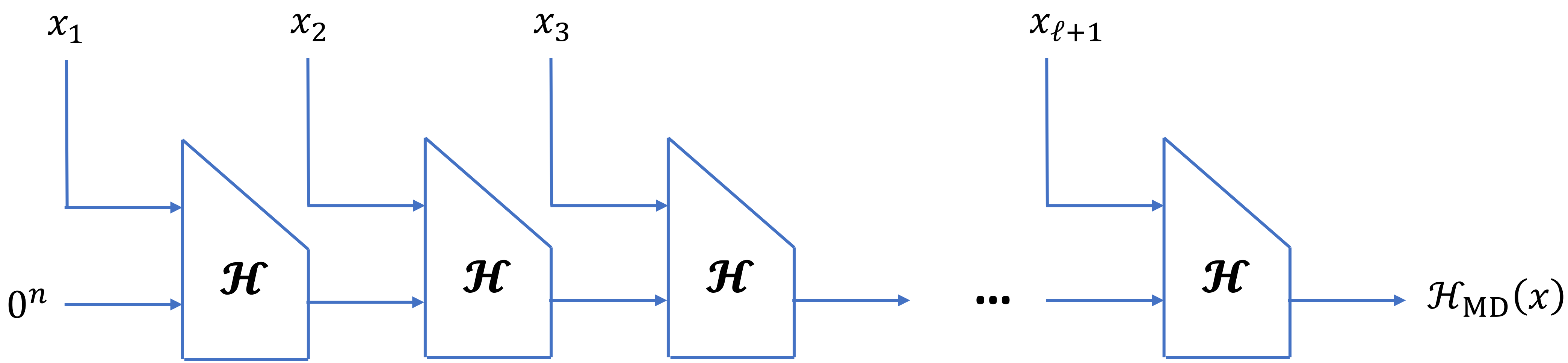
Merkle-Damgård transform

What does it do?

- transforms one algorithm into another;
- <u>input algorithm</u>: computes a fixed-length *compression function* (e.g., $\mathcal{H}: \{0,1\}^{2n} \to \{0,1\}^n$);
- <u>output algorithm</u>: computes an arbitrary-input-length hash function $\mathcal{H}_{\mathrm{MD}}: \{0,1\}^* \to \{0,1\}^n$;

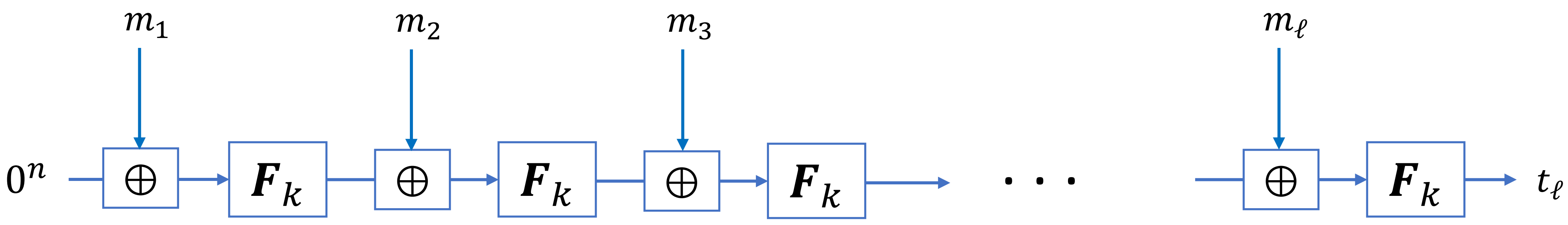
How does it work?

Split up input: $x = (x_1, x_2, \dots, x_\ell)$ so each $x_i \in \{0,1\}^n$. Set $x_{\ell+1} := |x|$.

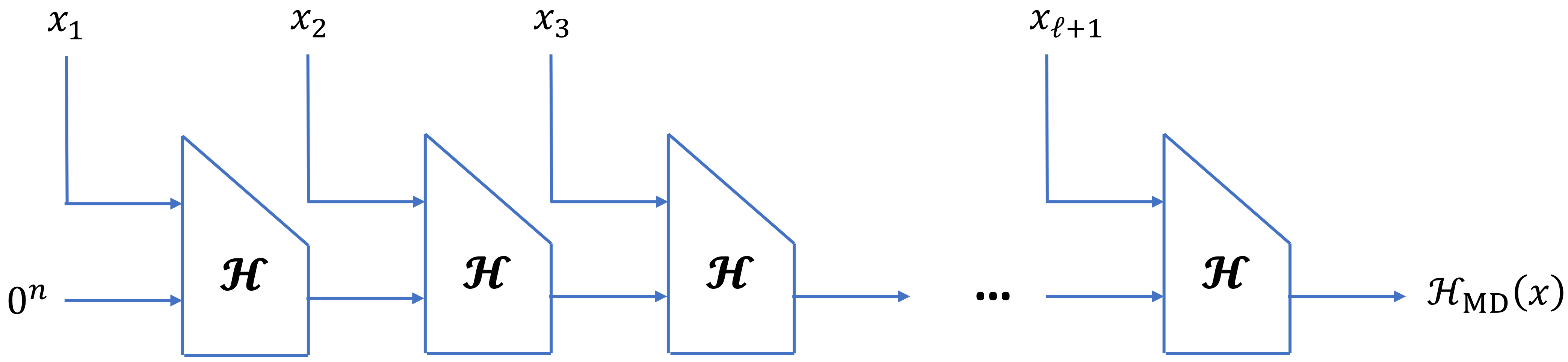


CBC-MAC vs Merkle-Damgård

Compare: CBC-MAC



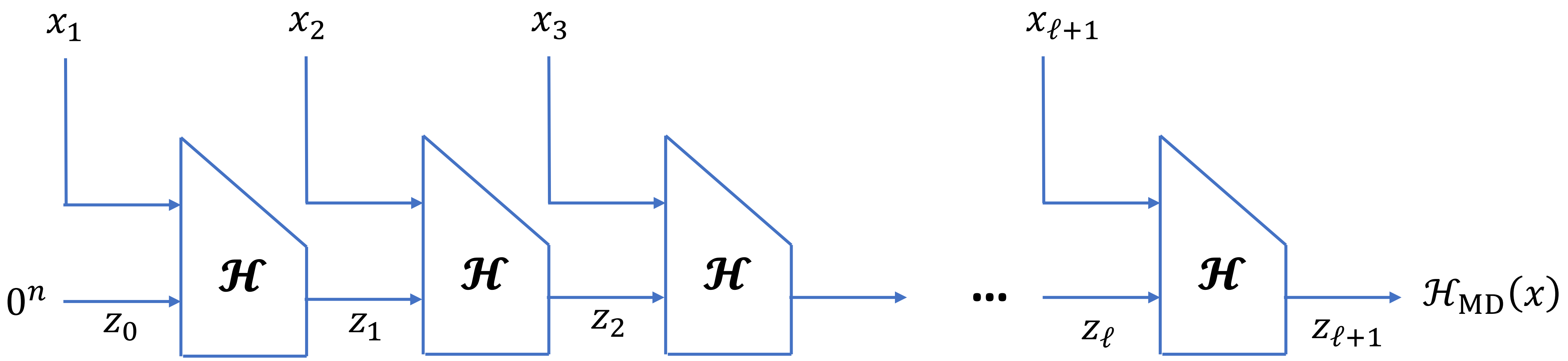
Compare: Merkle-Damgård transform



Merkle-Damgård transform

Construction (Merkle-Damgård).
Let $(\mathbf{KeyGen}_{\mathrm{H}}, \mathcal{H})$ be a hash function, and suppose $\mathcal{H}: \{0,1\}^{2n} \to \{0,1\}^n$. Define a new hash function:

- $\mathbf{KeyGen}_{\mathrm{H_{MD}}}$: same as $\mathbf{KeyGen}_{\mathrm{H}}$;
- $\mathcal{H}_{\mathrm{MD}}: \{0,1\}^* \to \{0,1\}^n$ defined as follows, on input x:
  1. assume length $|x|$ of x is divisible by n (otherwise pad with 0s);
  2. split x as $x = (x_1, x_2, \dots, x_\ell)$ and set $x_{\ell+1} \coloneqq |x|$.
  3. set $z_0 = 0^n$; compute $z_i = \mathcal{H}(x_i, z_{i-1})$;
  4. output $z_{\ell+1}$.



Merkle–Damgård transform

Remember:

- critical property we needed for integrity checks…
- … and for Hash-and-MAC…
- … was collision-resistance!

What happens when we apply MD?

Theorem. If $\mathcal{H}$ is a collision-free hash function, then so is its Merkle-Damgård transform $\mathcal{H}_{\mathrm{MD}}$.

See book for proof. It's fairly straightforward.

HASH FUNCTIONS continued

More on hash functions:

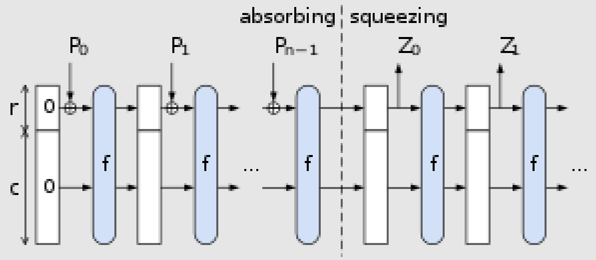
1. Keyed **vs** unkeyed;

2. Arbitrary-length inputs **vs** fixed-length inputs;

3. Hash functions as random oracles.

RANDOM ORACLES

SHA3
2015



The basic block permutation function consists of $12 + 2\ell$ rounds of five steps:

θ (theta)

Compute the parity of each of the $5w$ (320, when $w = 64$) 5-bit columns, and exclusive-or that into two nearby columns in a regular pattern. To be precise, $a[i][j][k] \leftarrow a[i][j][k] \oplus \text{parity}(a[0...4][j-1][k]) \oplus \text{parity}(a[0...4][j+1][k-1])$

ρ (rho)

Bitwise rotate each of the 25 words by a different triangular number 0, 1, 3, 6, 10, 15, ... To be precise, $a[0][0]$ is not rotated, and for all $0 \le t < 24$, $a[i][j][k] \leftarrow a[i][j][k-(t+1)(t+2)/2]$, where $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}^t \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

π (pi)

Permute the 25 words in a fixed pattern. $a[j][2i+3j] \leftarrow a[i][j]$.

χ (chi)

Bitwise combine along rows, using $x \leftarrow x \oplus (\neg y \,\&\, z)$. To be precise, $a[i][j][k] \leftarrow a[i][j][k] \oplus (\neg a[i][j+1][k] \,\&\, a[i][j+2][k])$. This is the only non-linear operation in SHA-3.

ι (iota)

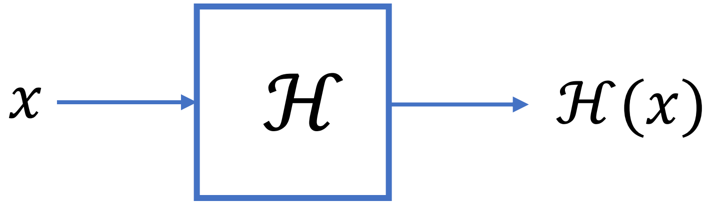
Exclusive-or a round constant into one word of the state. To be precise, in round n, for $0 \le m \le \ell$, $a[0][0][2^m-1]$ is XORed with bit $m + 7n$ of a degree-8 LFSR sequence. This breaks the symmetry that is preserved by the other steps.

Recall:

We constructed stuff from the collision-resistant property.

But in practice, for good hash functions:

- nobody knows what to do except evaluate;
- and, when they evaluate…
- … they can't distinguish output from random!
- (can you come up with a "security game" for this task?)



This is **much stronger** than just collision-resistance!

RANDOM ORACLES

So really…

It seems like hash functions behave like **random oracles!**

It's **as if** someone sampled a uniformly random function $\boldsymbol{R}$…

… and then put it in an oracle!

A strong hash function (like SHA-3) is developed and standardized.

LOOKS LIKE

1. Define $\boldsymbol{R}: \{0,1\}^{\mathrm{n}} \rightarrow \{0,1\}^{n}$ by setting $\boldsymbol{R}(x) \leftarrow \{0,1\}^{n}$ for each x.
2. Put $\boldsymbol{R}$ "into a box" so **everyone** can query it, but only as an oracle.

$$x \longrightarrow \boxed{\boldsymbol{R}} \longrightarrow \boldsymbol{R}(x)$$

RANDOM ORACLES

Some caveats:

- this is just a general impression;
- for example:

1. A hash function is a fixed, deterministic object;
2. A random oracle is drawn from a distribution of functions.

So we should be careful!

But what can we do in such a model?

It's called "The Random Oracle Model" (ROM.)

Let's assume that it's real. What does it get us?

1. Define $\boldsymbol{R}: \{0,1\}^{n} \rightarrow \{0,1\}^{n}$
   by setting $\boldsymbol{R}(x) \leftarrow \{0,1\}^{n}$ for each x.

2. Put $\boldsymbol{R}$ "into a box" so **everyone**
   can query it, but only as an oracle.

$$x \longrightarrow \boxed{\boldsymbol{R}} \longrightarrow \boldsymbol{R}(x)$$

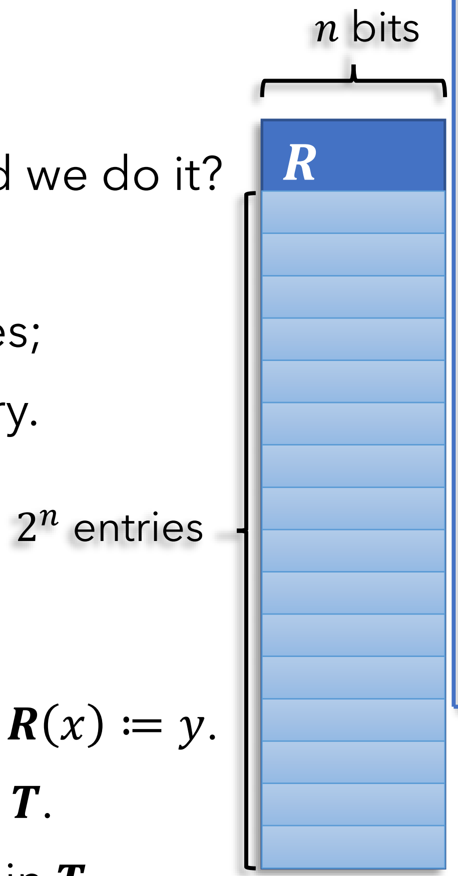
Random Oracle Model (ROM).

A useful observation:

Suppose it is OUR job to sample $\boldsymbol{R}$. How could we do it?

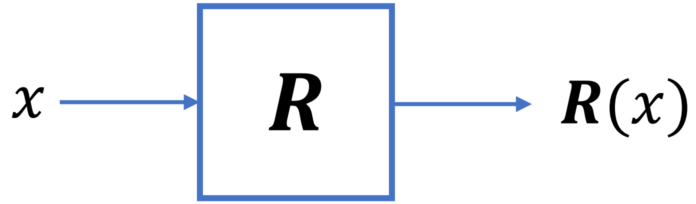
I. generate a huge lookup table with 2^n entries;
put a random string from $\{0,1\}^n$ in each entry.

II. be lazy about it!
wait until someone asks a question x…
generate a random value y and output it as $\boldsymbol{R}(x) \coloneqq y$.
… *and* store the pair (x, y) in a lookup table $\boldsymbol{T}$.
for future questions x': (i.) check if $\exists(x', y')$ in $\boldsymbol{T}$.
(ii.) if yes, return y'; if no, generate fresh y' and add (x', y') to $\boldsymbol{T}$.



1. Define $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$
by setting $\boldsymbol{R}(x) \leftarrow \{0,1\}^n$ for each x.

2. Put $\boldsymbol{R}$ "into a box" so **everyone**
can query it, but only as an oracle.



RANDOM ORACLES

Random Oracle Model (ROM).

Lazy sampling:

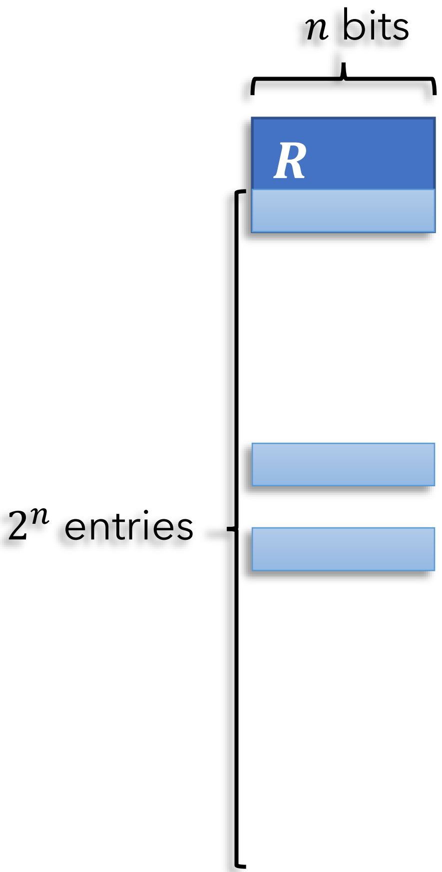
wait until someone asks a question x…

generate a random value y and output it as $\boldsymbol{R}(x) \coloneqq y$.

… *and* store the pair (x, y) in a lookup table T.

for future questions x': (i.) check if $\exists(x', y')$ in T.

(ii.) if yes, return y'; if no, generate fresh y' and add (x', y') to T.



Important takeaways:

1. In any ROM situation:
   - if <u>nobody</u> has asked a question x yet…
   - … then $\boldsymbol{R}(x)$ is still uniformly random (and independent of everything!)
2. We can simulate the random oracle in reductions!

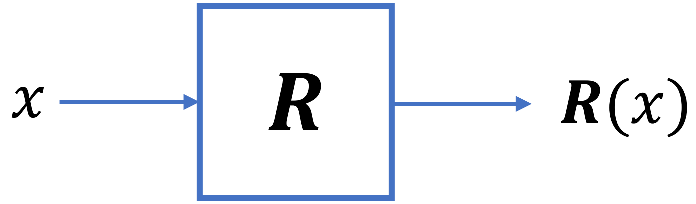
RANDOM ORACLES ⇒ collision-resistant hashing

Random Oracle Model (ROM).

What crypto can we build in this model?

1. Define $\boldsymbol{R}: \{0,1\}^n \rightarrow \{0,1\}^n$
   by setting $\boldsymbol{R}(x) \leftarrow \{0,1\}^n$ for each x.

2. Put $\boldsymbol{R}$ "into a box" so **everyone**
   can query it, but only as an oracle.



Collision-resistant hash:

- recall: random functions are collision-resistant;
- (because preimages are uniformly distributed)
- so $\boldsymbol{R}$ itself serves as a collision-resistant hash;
- if we want small outputs, can discard bits of output.

Note:

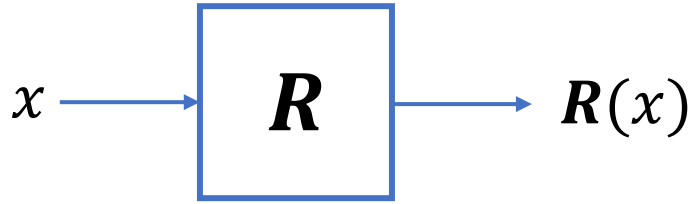
- this is now *statistical collision-resistance;*
- for normal hash functions, it was *computational* (i.e., against PPT adversaries.)
- remember: *collision-resistant* ⇒ *one-way*. So we also get **one-way functions!**

RANDOM ORACLES ⇒ PRFs

Random Oracle Model (ROM).

What crypto can we build in this model?

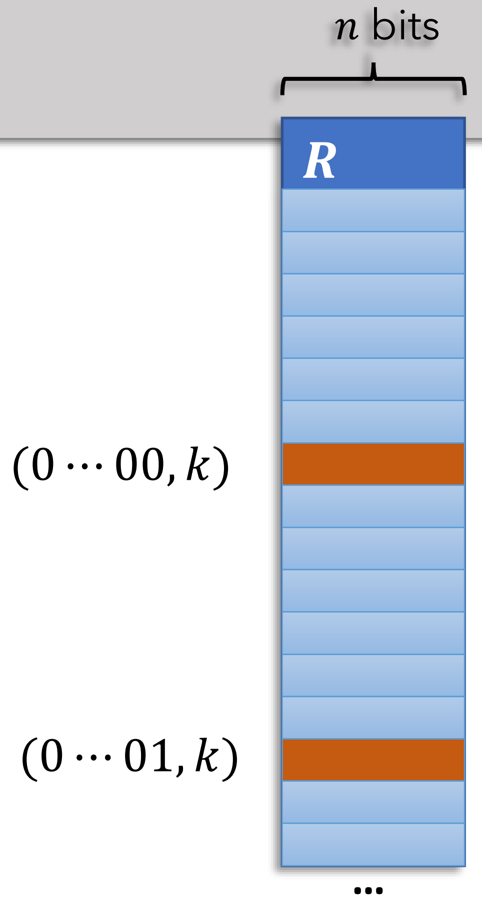
1. Define $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$
   by setting $\boldsymbol{R}(x) \leftarrow \{0,1\}^n$ for each x.
2. Put $\boldsymbol{R}$ "into a box" so **everyone**
   can query it, but only as an oracle.



Pseudorandom functions:

- sample a key: $k \leftarrow \{0,1\}^{n/2}$;
- define

$$\boldsymbol{F}_k: \{0,1\}^{n/2} \to \{0,1\}^{n/2}$$
$$\boldsymbol{F}_k(x) := \boldsymbol{R}(x, k)$$



Why is it pseudorandom? Note: $\boldsymbol{A}$ knows $\boldsymbol{R}$!

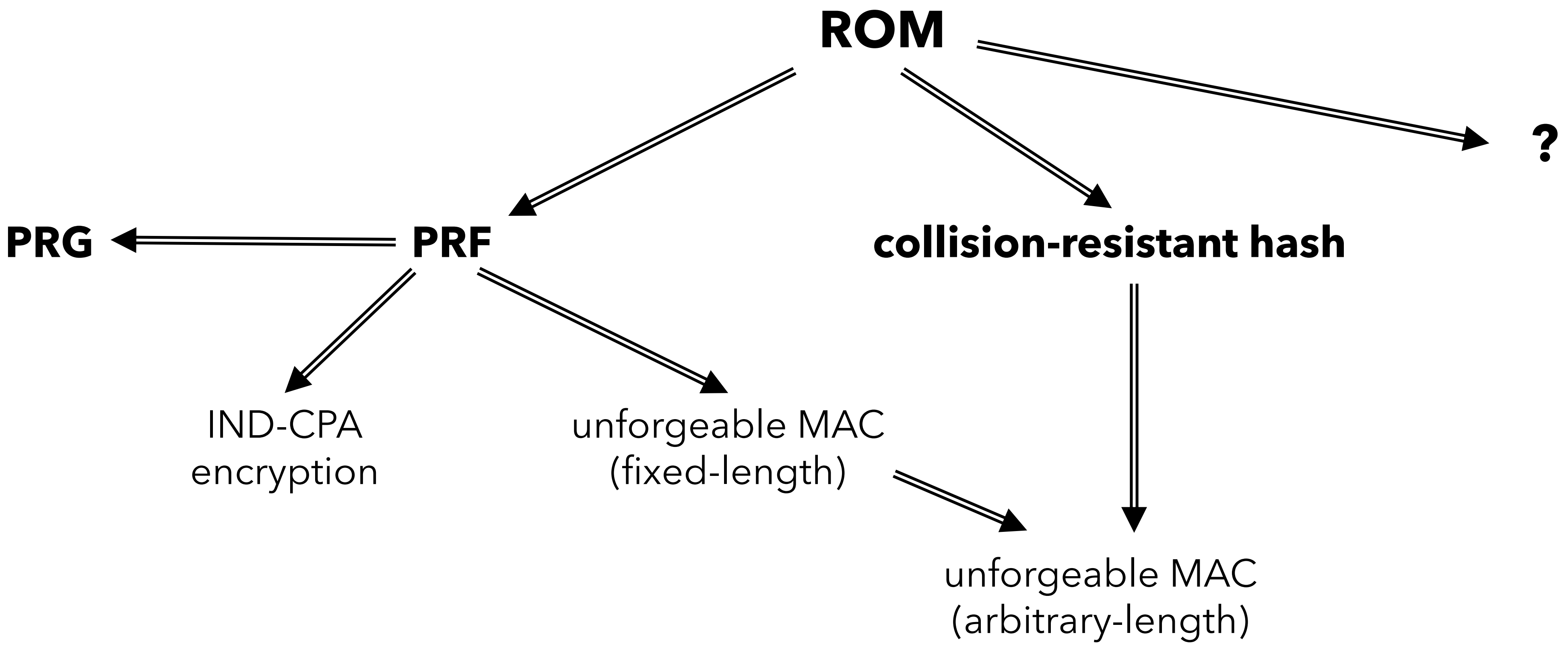
1. take any algorithm $\boldsymbol{A}^{\boldsymbol{F}_k}$. It makes some query x_1;
2. $\Pr[x_1 = (z, k)] = 2^{-n/2}$ for any z; so response is uniformly random in $\{0,1\}^{n/2}$;
3. in particular, $\boldsymbol{A}^{\boldsymbol{F}_k}$ learned nothing with the first query.
4. so we can repeat the argument starting from 1.

so $\boldsymbol{F}_k$ is oracle indistinguishable from a random function!

RANDOM ORACLES ⇒ lots of stuff

Random Oracle Model (ROM).

What crypto can we build in this model?



RANDOM ORACLES ⇒ one-time authentication

IMPORTANT! NEW IDEAS!

Lamport scheme. One-time MAC for messages of length ℓ.

Let $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$ be a random oracle.

KeyGen:

I. Sample 2ℓ random inputs to $\boldsymbol{R}$:

- $x_1^0,\ x_2^0,\ x_3^0, \dots, x_\ell^0.$
- $x_1^1,\ x_2^1,\ x_3^1, \dots, x_\ell^1.$

Note each $x_j^b \in \{0,1\}^n$.

II. Now compute, for each j, b:

$$y_j^b \coloneqq \boldsymbol{R}(x_j^b);$$

III. Output key consisting of two parts:

1. $x_1^0,\ x_2^0,\ x_3^0, \dots, x_\ell^0$ and $x_1^1,\ x_2^1,\ x_3^1, \dots, x_\ell^1$;
2. $y_1^0,\ y_2^0,\ y_3^0, \dots, y_\ell^0$ and $y_1^1,\ y_2^1,\ y_3^1, \dots, y_\ell^1$;

0	x_1^0	x_2^0	x_3^0	…	x_ℓ^0
1	x_1^1	x_2^1	x_3^1	…	x_ℓ^1

$$\downarrow \boxed{\boldsymbol{R}} \downarrow$$

0	y_1^0	y_2^0	y_3^0	…	y_ℓ^0
1	y_1^1	y_2^1	y_3^1	…	y_ℓ^1

RANDOM ORACLES ⇒ one-time authentication

Lamport scheme. One-time MAC for messages of length ℓ.

Let $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$ be a random oracle.

Mac:

On input a message $m \in \{0,1\}^\ell$:

Output tag $t \in \{0,1\}^{n\ell}$ like this:

For each bit position $j = 1, 2, \dots, \ell$

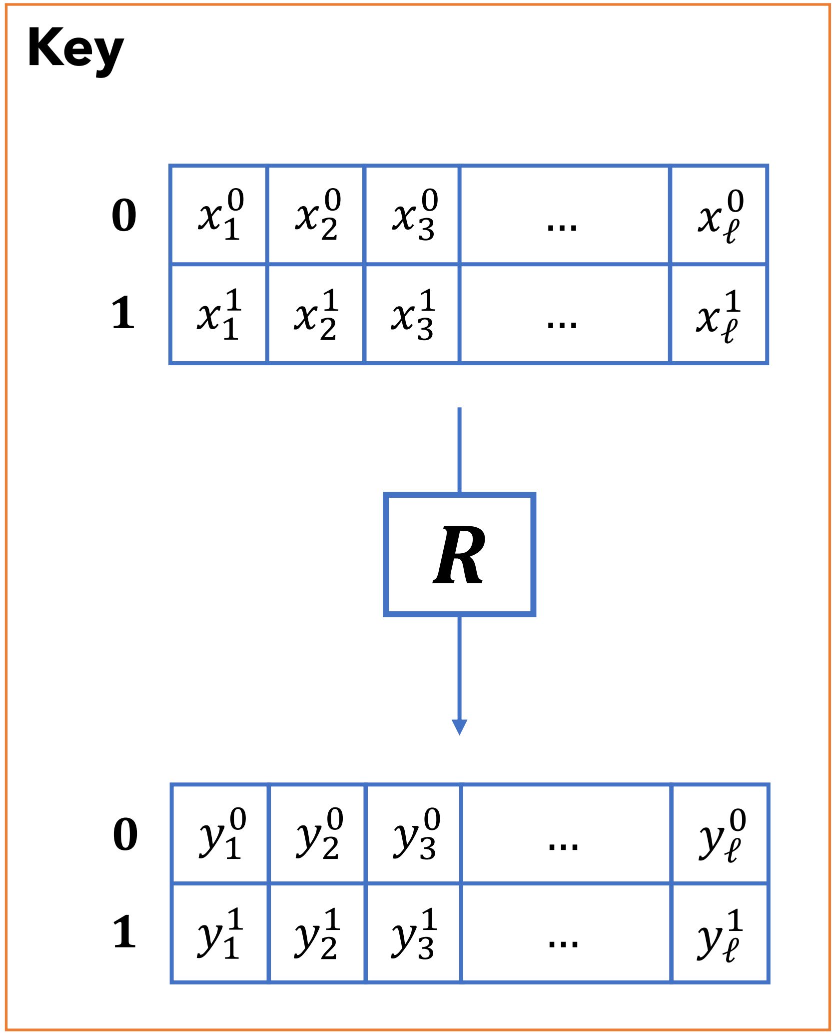
output $x_j^{m_j}$.

Example:

Suppose $m = 010110$.

x_1^0		x_3^0			x_6^0
	x_2^1		x_4^1	x_5^1	

So tag is $(x_1^0, x_2^1, x_3^0, x_4^1, x_5^1, x_6^0)$.



RANDOM ORACLES ⇒ one-time authentication

Lamport scheme. One-time MAC for messages of length ℓ.

Let $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$ be a random oracle.

Ver:

On input $m \in \{0,1\}^\ell$ and tag $(t_1, t_2, \dots, t_\ell)$:

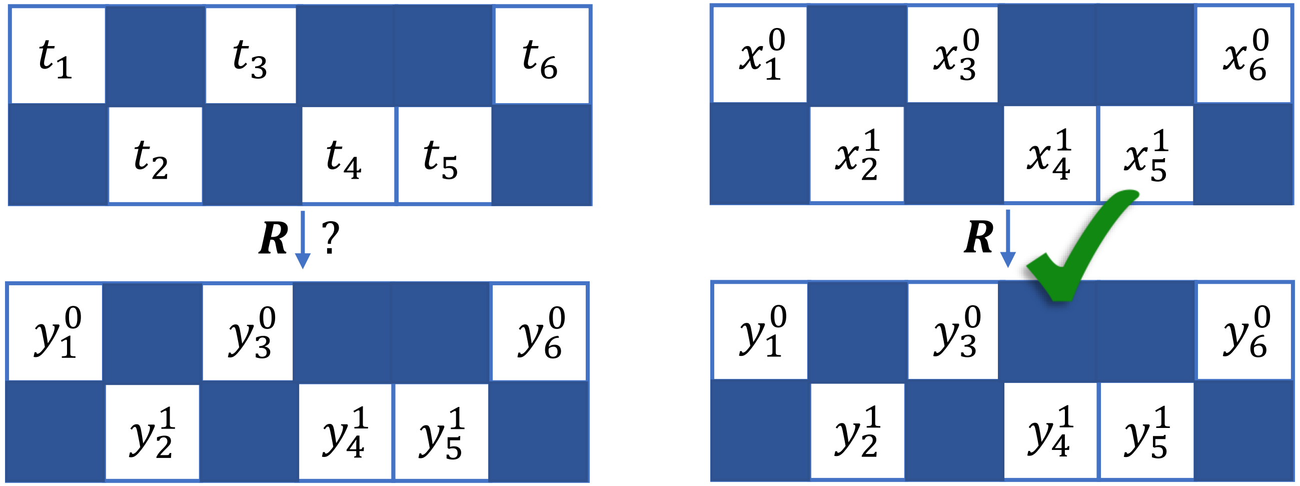
For each bit position $j = 1, 2, \dots, \ell$:

If $\left(R(t_j) \neq y_j^{m_j}\right)$ output **reject;**

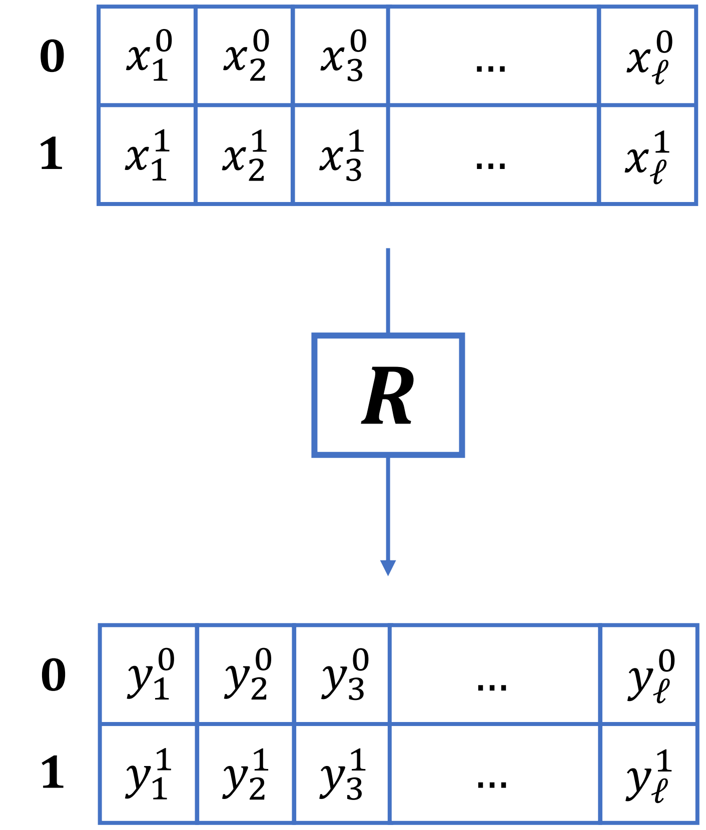
output **accept.**

Example: Suppose $m = 010110$.

Honestly generated



Key



RANDOM ORACLES $\Rightarrow$ one-time authentication

Lamport scheme. One-time MAC for messages of length ℓ.

Let $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$ be a random oracle.

Check correctness:

- for message $m \in \{0,1\}^\ell$ …
- … tag is $\left(x_1^{m_1}, x_2^{m_2}, x_3^{m_3}, \dots, x_\ell^{m_\ell}\right)$;
- at the verification stage, we do this check for each j:

$$\boldsymbol{R}\left(x_j^{m_j}\right) = y_j^{m_j}$$

- but in **KeyGen** this is exactly how we defined y_j^b for $b \in \{0,1\}$.
- so verification succeeds.

So scheme is correct. Is it unforgeable?

Key

0	x_1^0	x_2^0	x_3^0	…	x_ℓ^0
1	x_1^1	x_2^1	x_3^1	…	x_ℓ^1

$$\boxed{\boldsymbol{R}}$$

0	y_1^0	y_2^0	y_3^0	…	y_ℓ^0
1	y_1^1	y_2^1	y_3^1	…	y_ℓ^1

RANDOM ORACLES ⇒ one-time authentication

Lamport scheme. One-time MAC for messages of length ℓ.

Let $\boldsymbol{R}: \{0,1\}^n \to \{0,1\}^n$ be a random oracle.

So scheme is correct. Is it unforgeable?

Let's look at the adversary's view. It has two things:

$$m = m_0 m_1 m_2 \dots m_\ell$$

$t =$

x_1^0		x_3^0			x_6^0
	x_2^1		x_4^1	x_5^1	

Now adversary tries to forge on $m^* \neq m$.

There's a bit j where m^* differs from m. Say $j = 2$. Then…

$$m^* = m_0 m_1^* m_2 \dots m_\ell$$