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);

Piazza: piazza.com/umd/spring2020/cmsc456

ELMS: active, slides and reading posted there, **first homework is up (due midnight tonight.)**

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**)

HOMEWORK RULES AND GUIDELINES:

First homework is up (due midnight tonight.)

Rules

- collaboration ok, solutions must be written up by yourself, in your own words;
- late homeworks will not be accepted (*no exceptions*, but lowest grade will be dropped.)

Explanations and proofs

- correct answers with no explanation will get a zero score;
- explain your ideas clearly and completely;
- write in complete sentences, use correct and complete mathematical notation (as in lectures and book);
- proofs need to be rigorous, clear, and complete (consider all cases, prove counterexamples, etc.)

Suggestions

- work on your own at least some of the time for each assignment
- work in 25+ minute chunks of uninterrupted, distraction-free, device-free time
- develop intuition: try lots of examples, ask yourself questions, "play" with the concepts

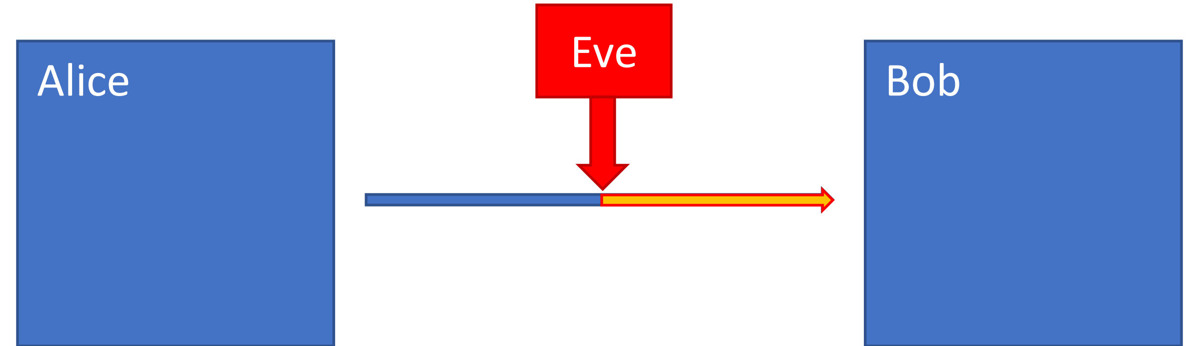
RECAP: IS CRYPTO JUST SECRECY?

Secrecy: protects against Eve learning our message.

What else could go wrong?

Eve could ***interfere***!

Is this possible? The message is encrypted!



Consider OTP:

- Eve observes a ciphertext $c = \mathbf{Enc}_k(m) = m \oplus k$;
- She flips some bits: $c \mapsto c \oplus s$;
- Bob decrypts: $\mathbf{Dec}_k(c \oplus s) = c \oplus s \oplus k = s \oplus c \oplus k = s \oplus m$.
- *Eve's attack was directly applied to the message!*

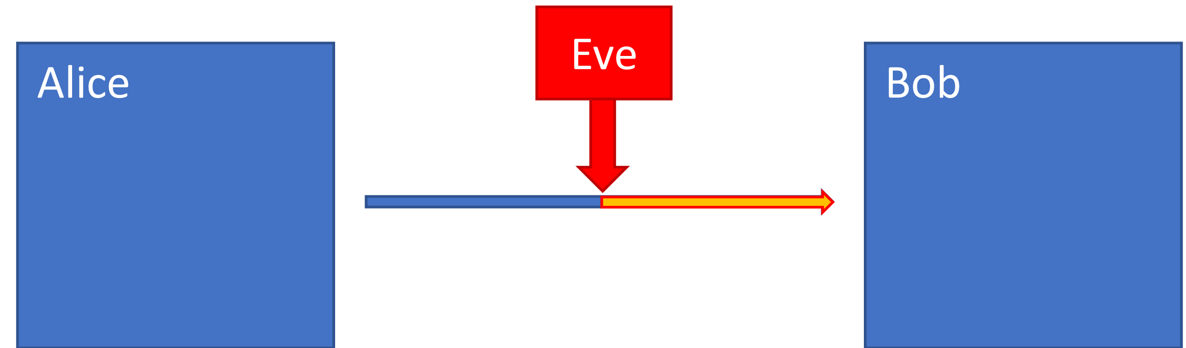
If m was a bank deposit, Eve could flip the bits that add thousands (or millions) to the amount!

RECAP: WHAT ABOUT FANCIER ENCRYPTION?

What about PRG and PRF encryption?

Both based on OTP!

So same attacks work!



For example, interference against PRF scheme:

- Eve observes a ciphertext $(r, c) \coloneqq \mathbf{Enc}_k(m) = (r, m \oplus \boldsymbol{F}_{\mathrm{k}}(r))$;
- She flips some bits: $(r, c) \mapsto (r, c \oplus s)$;
- Bob decrypts: $\mathbf{Dec}_k(r, c \oplus s) = c \oplus s \oplus \boldsymbol{F}_k(r) = s \oplus m$.
- *Eve's attack was directly applied to the message!*

All the extra ***secrecy*** protection of the PRF scheme did not help at all!

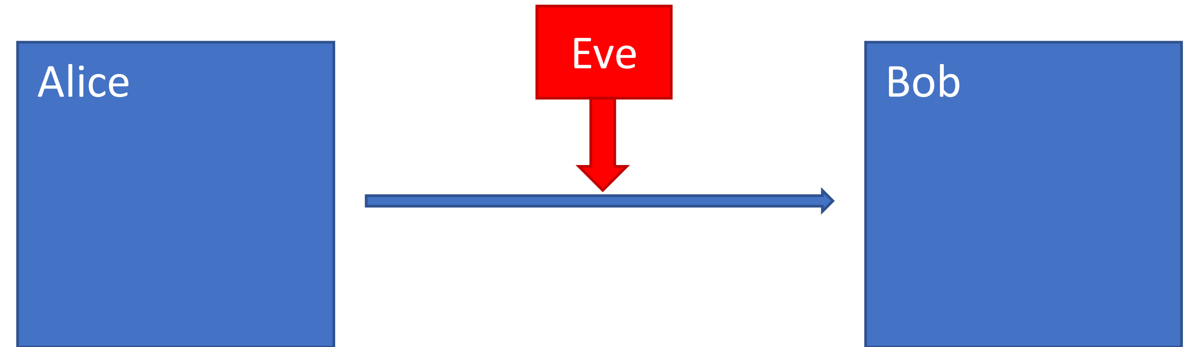
V. AUTHENTICATION

Reading: (p.107-126, 142-145)

RECAP: AUTHENTICATION

We now change tasks:

- forget *secrecy* for the moment!
- and instead consider *authenticity*.
- (we will talk about combining them later.)



The task:

- Alice wants to send a message to Bob;
- Bob's goal: make sure message is really from Alice…
- … and nobody else!

Assumptions:

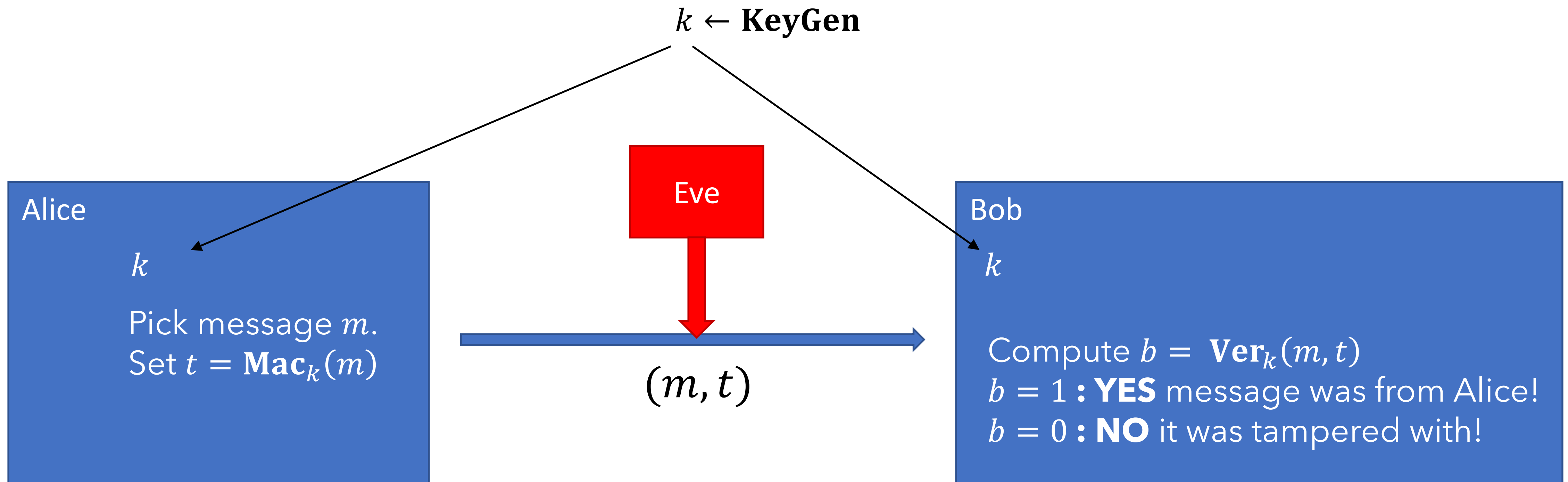
- Alice and Bob can share a secret in advance (and have private spaces);
- Alice can send only one transmission (for now);
- *Eve can change (or replace) the transmission however she likes!*
- *(… but we don't care if she can learn the message.)*

RECAP: MESSAGE AUTHENTICATION CODES

Message authentication code (MAC):

- generate key: $k \leftarrow \mathbf{KeyGen}$
- generate **tag**: $t \leftarrow \mathbf{Mac}_k(m)$
- verify (message, tag) pair: $b \leftarrow \mathbf{Ver}_k(m, t)$ [$b = 1$ (valid) or $b = 0$ (invalid)]

Correctness:
$\mathbf{Ver}_k(m, \mathbf{Mac}_k(m)) = 1.$

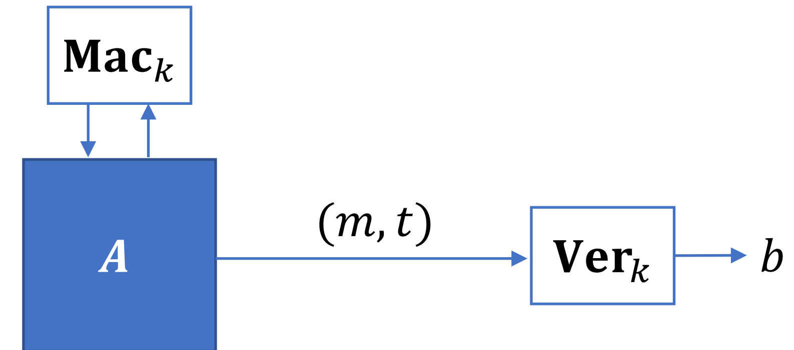


RECAP: UNFORGEABILITY

How to define security for MACs? Unforgeability.

Let's use a game: $\mathrm{MacForge}(\Pi, n)$, where Π is a MAC and n the security parameter.

1. A key is sampled: $k \leftarrow \mathbf{KeyGen}(1^{\mathrm{n}})$;
2. Adversary $\boldsymbol{A}$ is given oracle access to $\mathbf{Mac}_k$;
3. $\boldsymbol{A}$ outputs a pair (m, t); set $b = \mathbf{Ver}_k(m, t)$;



We say $\boldsymbol{A}$ wins the experiment if:

- $b = 1$ (valid), **and**
- m is not in the set of queries $\boldsymbol{A}$ made to the oracle.

Definition. A message authentication code Π is **existentially unforgeable under chosen message attack (EUF-CMA)** if, for every PPT adversary $\boldsymbol{A}$,

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

RECAP: CONSTRUCTING SECURE MACs

Definition. A keyed function family $f: K \times M \rightarrow T$ is **pairwise independent** if, for every $m \neq m'$ in M and all t, t' in T, we have

$$\Pr_{k \in K}[f_k(m) = t \wedge f_k(m') = t'] = \frac{1}{|T|^2}$$

Construction (Carter-Wegman). Let $f: K \times M \rightarrow T$ be a pairwise-independent function family. Define a MAC (with canonical verification) as follows:

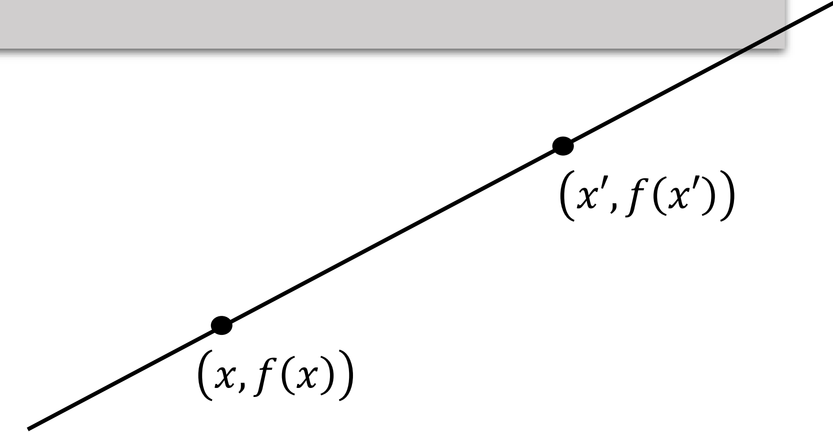
- $\mathbf{KeyGen}$: output uniformly random $k \leftarrow K$;
- $\mathbf{Mac}$: on input a key k and message $m \in M$, output tag $f_k(m)$.

Theorem. The Carter-Wegman MAC with a pairwise-independent function is 1-**EUF-CMA** *against arbitrary adversaries.*

Proof idea.

- the first pair (m, t) is the adversary's query;
- the second pair (m', t') is the adversary's claimed forgery;
- now apply definition of pairwise independent.

RECAP: CONSTRUCTING SECURE MACs



Pairwise-independent functions: random lines in $\mathbb{Z}_p$.

- Input and output spaces: $\mathbb{Z}_p = \{0,1,2,\ldots,p-1\}$ for a **prime** p.
- Key space: $\mathbb{Z}_p \times \mathbb{Z}_p$.
- All arithmetic will be modulo p.
- Recall: since p is a prime, we have multiplicative inverses (and can easily compute them.)

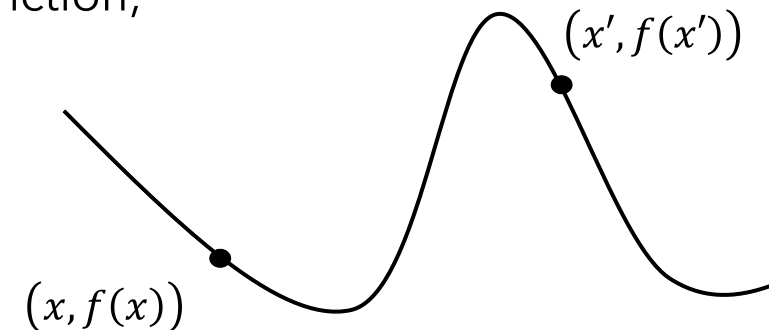
For any pair $(a, b) \in \mathbb{Z}_p \times \mathbb{Z}_p$, define

$$f_{a,b}(x) := a \cdot x + b$$

Can extend this idea…

- take random *polynomials* over $\mathbb{Z}_p$;
- keys get a bit bigger, but now you need *degree-many* points to learn the function;
- get info-theoretic q-time MACs for any fixed q.

What about arbitrary-many queries?



RECAP: PRF MAC

Construction (PRF MAC). Let $\boldsymbol{F}: \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^\ell$ be a PRF. Define a MAC (with canonical verification) as follows:

- $\mathbf{KeyGen}$: output uniformly random $k \leftarrow \{0,1\}^n$;
- $\mathbf{Mac}$: on input a key k and message $m \in \{0,1\}^m$, output tag $\boldsymbol{F}_k(m)$.

Notes.

- messages are of fixed length;
- tags are of length $\ell(n)$; we can pick this however we want (by selecting the right PRF)…
- … but careful: recall trivial tag-guessing attack, which succeeds with probability $2^{-\ell(n)}$.

Proof.

- similar to IND-CPA proof:

1. show that a scheme with a *perfectly random* function is statistically unforgeable;
2. then show that a forger for the PRF MAC would imply a distinguisher for the PRF.

PRF MAC SECURITY

Construction (PRF MAC). Let $\boldsymbol{F}: \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^\ell$ be a PRF. Define a MAC (with canonical verification) as follows:

- $\mathbf{KeyGen}$: output uniformly random $k \leftarrow \{0,1\}^n$;
- $\mathbf{Mac}$: on input a key k and message $m \in \{0,1\}^m$, output tag $\boldsymbol{F}_k(m)$.

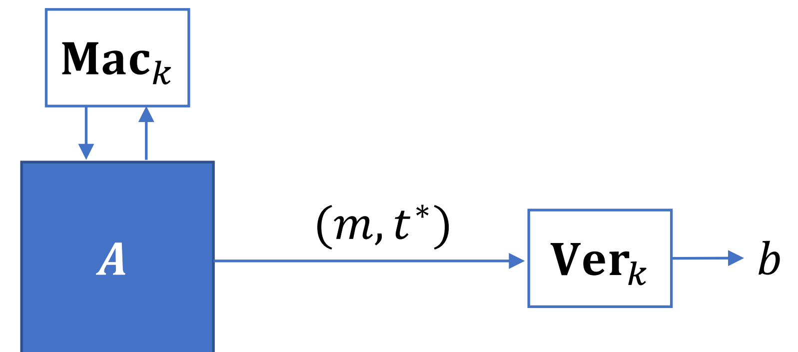
Theorem. The PRF MAC is **EUF-CMA** (against PPT adversaries.)

Proof.

- suppose we use a completely random function $\boldsymbol{R}$ in place of $\boldsymbol{F}_k$;
- recall: the candidate forgery message m has to be **fresh**;
- this means: $\boldsymbol{R}(m)$ has yet to be queried;
- it follows that $t = \boldsymbol{R}(m)$ is uniformly random;
- so $\boldsymbol{A}$ loses against random scheme: $\Pr[t = t^*] = 2^{-\ell(n)}$.

Now suppose $\boldsymbol{A}$ wins against pseudorandom scheme…

… then we build a distinguisher for $\boldsymbol{F}$ – a contradiction!



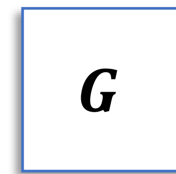
PRF MAC SECURITY

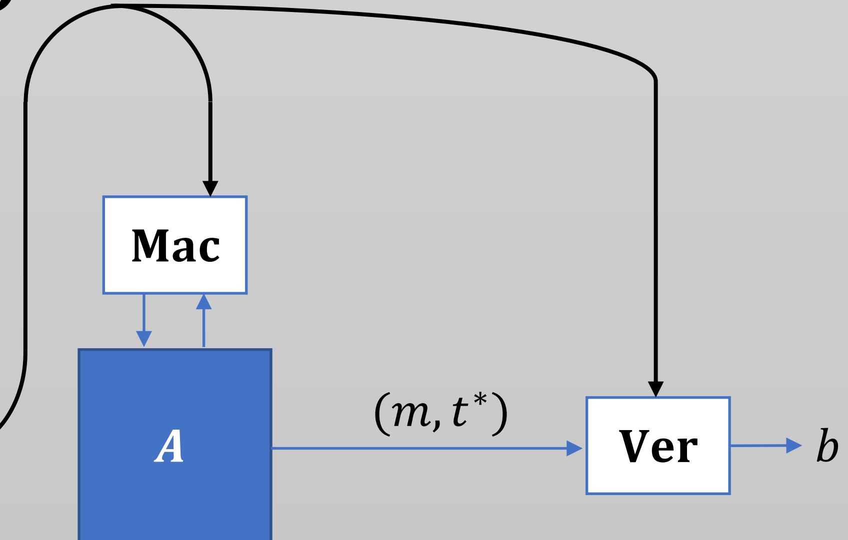
Construction (PRF MAC). Let $\boldsymbol{F}: \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^\ell$ be a PRF. D
(with canonical verification) as follows:
- **KeyGen**: output uniformly random $k \leftarrow \{0,1\}^n$;
- **Mac**: on input a key k and message $m \in \{0,1\}^m$, output tag $\boldsymbol{F}_k(m)$

Theorem. The PRF MAC is **EUF-CMA** (against PPT adversaries.)

Proof (continued.)

How to build the distinguisher? Simulate EUF-CMA!





- keep a list $L = \{m_1, m_2, \dots\}$ of all queries made;
- when $\boldsymbol{A}$ outputs (m, t^*), check that it verifies, and that $m \notin L$.

If checks pass (i.e., $\boldsymbol{A}$ won) output 1.
Otherwise output 0.

Two cases:

1. $\boldsymbol{G}$ sampled as a random function;
   (and $\boldsymbol{A}$ loses, by last slide.)
2. $\boldsymbol{G}$ sampled as $\boldsymbol{F}_k$ for random k;
   (and $\boldsymbol{A}$ wins, by assumption.)

Result: a distinguisher between case 1 and case 2. □

PRACTICAL MACs

In practice…

- one-time (or q-time) MACs are not used much, except as building blocks;
- and the PRF MAC is too inefficient;
- in general, PRFs for arbitrary input/output lengths are quite inefficient;

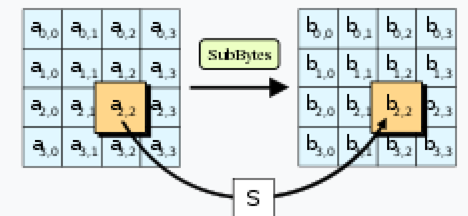
Block ciphers!

- are typically much more practical;
- these are PRFs with the same input and output length;
- *[another nice property we won’t need for now: they are invertible!]*

One of the most common MACs on the Internet…

- is the CBC-MAC, which uses block ciphers;
- the CBC stands for “cipher block chaining”;
- let’s see how it works.

Advanced Encryption Standard (Rijndael)



The SubBytes step, one of four stages in a round of AES

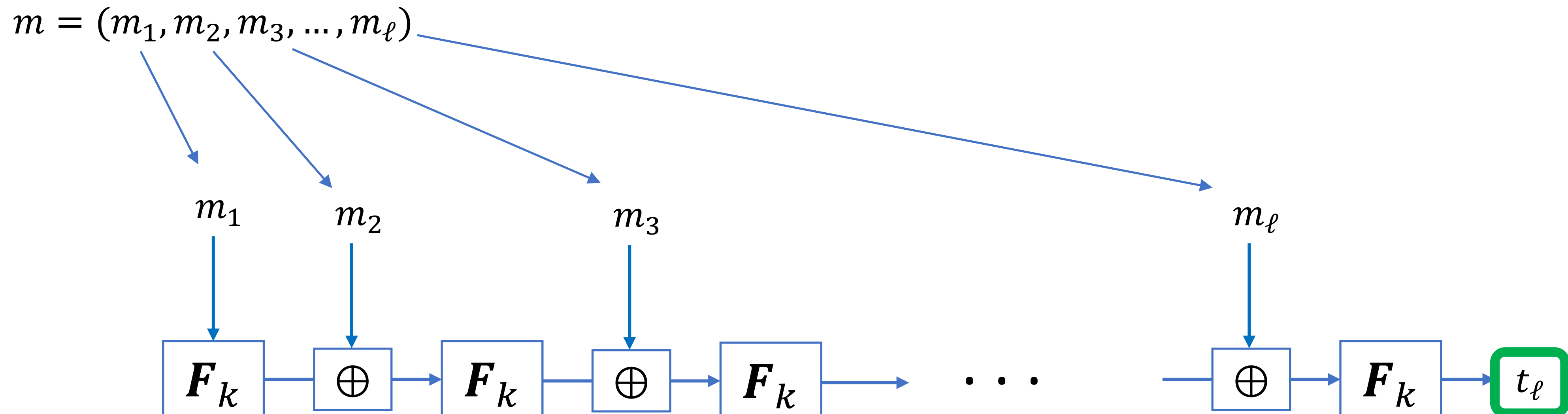
General	
Designers	Vincent Rijmen, Joan Daemen
First published	1998
Derived from	Square
Successors	Anubis, Grand Cru, Kalyna
Certification	AES winner, CRYPTREC, NESSIE, NSA
Cipher detail	
Key sizes	128, 192 or 256 bits[note 1]
Block sizes	128 bits[note 2]
Structure	Substitution–permutation network
Rounds	10, 12 or 14 (depending on key size)

CBC-MAC

Construction (CBC-MAC). Let $\boldsymbol{F}: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a PRF, and $\ell(n)$ any polynomial.
Define a deterministic MAC as follows:

- $\mathbf{KeyGen}$: output uniformly random $k \leftarrow \{0,1\}^n$;
- $\mathbf{Mac}$: on input a key k and message $m \in \{0,1\}^{\ell(n)\cdot n}$, do:
    - split m up: $m = (m_1, m_2, m_3, \dots, m_\ell)$ into chunks of length n;
    - set $t_0 \coloneqq 0^n$ and $t_i \coloneqq \boldsymbol{F}_k(t_{i-1} \oplus m_i)$ for $0 < i \leq \ell$.
    - output t_ℓ.

In pictures:



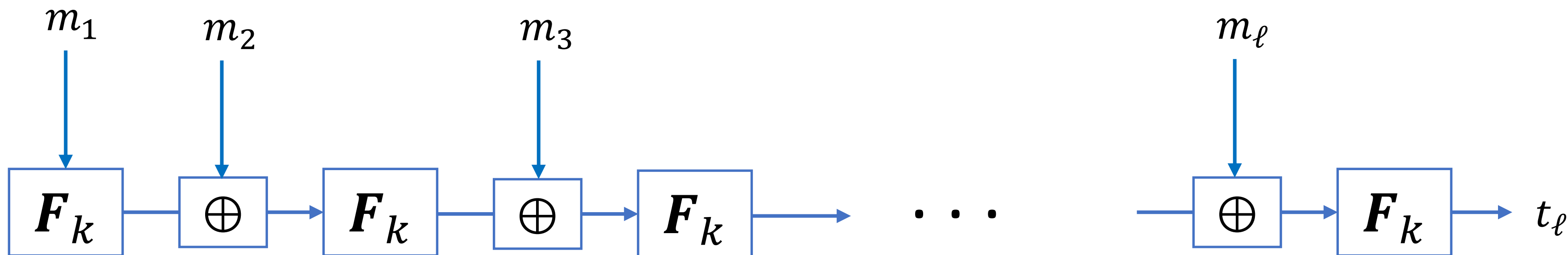
CBC-MAC

CBC-MAC is secure for fixed-length messages (see book.)

What does this mean?

- at key generation time, everyone needs to agree on a fixed length;
- for CBC-MAC, this amounts to selecting the function $\ell(n)$;
- after that point, *all messages* to be authenticated must be of length $\ell(n) \cdot n$;
- any deviation might result in an attack!

What happens if we use it to authenticate a message of different length anyway?



CBC-MAC

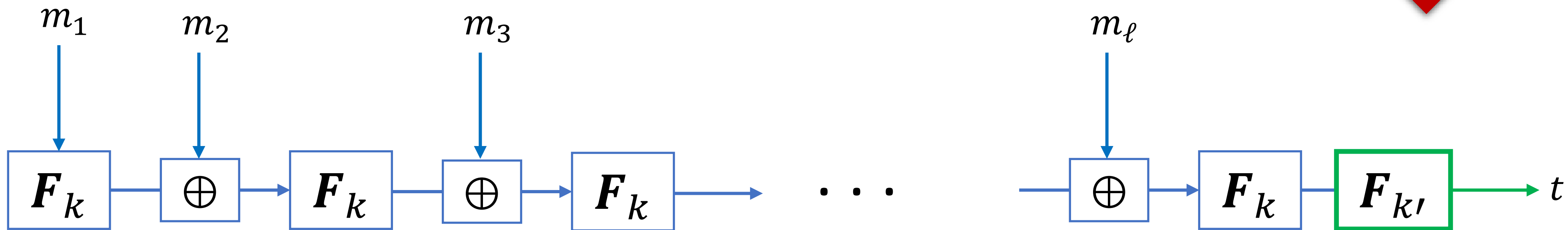
… attacks do indeed become possible.

CBC-MAC is **not** secure for variable-length messages. The trouble:

- there's nothing special about the start or the end of these chains;
- this introduces vulnerabilities.

The so-called Encrypted-CBC-MAC fixes this:

- key generation now samples two keys k, k' for the PRF;
- the chain is "capped" with an application of $\boldsymbol{F}_{k'}$.



ENCRYPTED-CBC-MAC

Theorem. The Encrypted-CBC-MAC is **EUF-CMA** for arbitrary-length messages.

Proof is somewhat involved (but mostly a matter of complicated bookkeeping.)

- ok, so now we can authenticate variable-length messages in a fairly efficient way;
- is CBC-MAC the only way? Could there be something even more efficient?
- maybe first, as a general matter: why should we care?

In general, having multiple ways to achieve the same crypto goal is helpful!

- different efficiency tradeoffs;
- different computational assumptions;
- could lead to new ideas!

Different approach: *use hash functions.*

HASH FUNCTIONS

What are hash functions?

A hash function is just a function wh[...] compresses its input:

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

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
                                                  byte.[50]
...
            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));
```

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, $a[i][j][k] \leftarrow a[i][j][k] \oplus \text{parity}(a[0...4][j-1][k]) \oplus \text{parity}(a[0...4][j+1][k-1])$

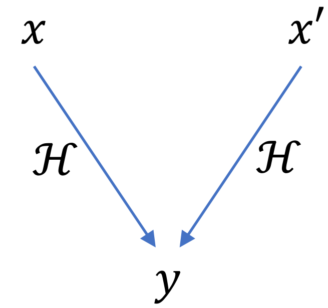
ρ (rho)
Bitwise rotate each of the 25 words by a different triangular number 0, 1, 3, 6, 10, 15, ... To be precise, a[0][0] is not rotated, and for all $0 \le t < 24$, $a[i][j][k] \leftarrow a[i][j][k-(t+1)(t+2)/2]$, where $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}^t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

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.

HASH FUNCTIONS



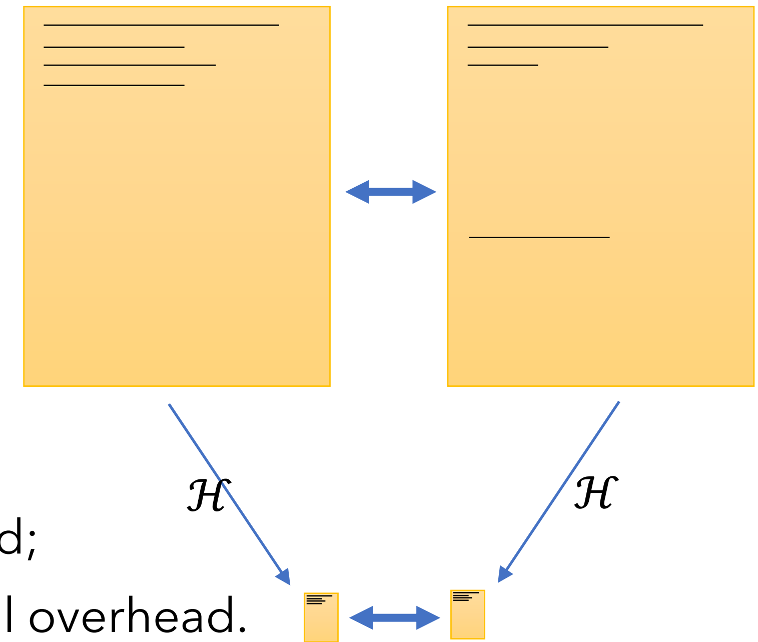
What are they good for?

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.

HASH FUNCTIONS

This should remind you of something:

- authentication!
- if comparing files (messages) is basically equivalent to comparing their hash digests…
- … why not just MAC the digest? Huge efficiency gain!
- this actually works, and is called "Hash-and-MAC."

Actually, hash functions are even crazier…

For a well-designed hash function h:

- *h seems to be indistinguishable from a random function!*
- *and the only interesting thing we know to do with them…*
- *… is just evaluate them!*

(Think back to our discussion on oracles!)

But let's slow down. This is all very informal so far.

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 $\mathcal{H}^s(x)$.

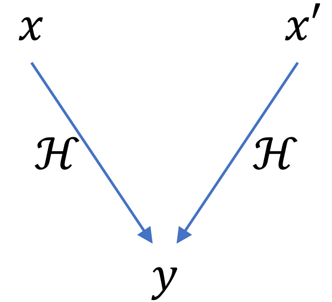
Why? In practice, anyone can look up hash function spec

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, $\Pr_{\boldsymbol{R}}[z \in X_y] \approx 2^{-n}$.

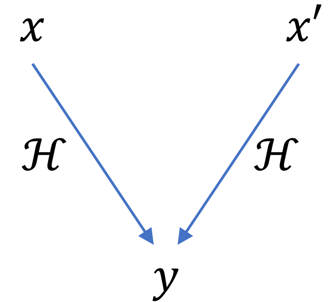
So, there are indeed functions for which it's hard to find preimages and collisions.

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

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 $\boldsymbol{A}$,

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

WEAKER PROPERTIES

We could ask for *weaker* properties.

"target-collision resistance"

- adversary has a harder task:
- given a fixed x, $\boldsymbol{A}$ must find $x' \neq x$ such that $\mathcal{H}^s(x') = \mathcal{H}^s(x)$.
- clearly implied by collision-resistance.

"preimage resistance"

- slightly different, but still harder task:
- given a random y, find x such that $\mathcal{H}^s(x) = y$.
- implied by collision-resistance:
- if you can find preimages: (i.) pick a random x; (ii.) run preimage-finding on $y := \mathcal{H}^s(x)$;
- *check:* with good probability over x, preimage-finding will yield x' such that $x' \neq x$.

WEAKER PROPERTIES

By the way:

"preimage resistance" is something like a "one-way" property:

1. Easy to evaluate;
2. Hard to invert on random inputs.

- such *"one-way functions"* are very important in the foundations of crypto;
- we will (probably) define them formally later in the course;
- you can build PRGs out of them, so by extension almost everything we've seen so far;
- … and some cool things we haven't! (next lecture)

But back to collision-resistance…

HASH-and-MAC

What is collision resistance good for?

Authentication!

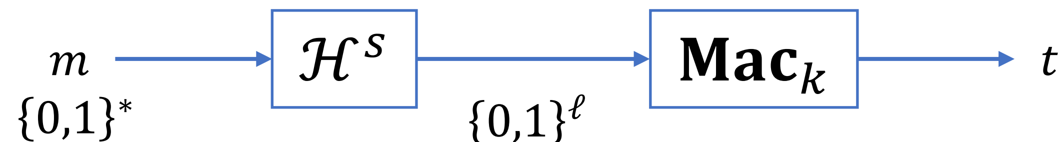
Construction (Hash-and-MAC). Let

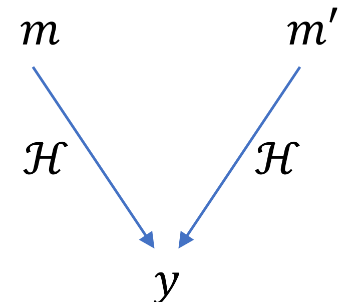
- $\Pi = (\mathbf{KeyGen}, \mathbf{Mac})$ be a fixed-length message authentication code (MAC), and
- $\Pi_{\mathrm{H}} = (\mathbf{KeyGen}_{\mathrm{H}}, \mathcal{H})$ be a hash function.

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 $t := \mathbf{Mac}_k(\mathcal{H}^s(m))$.

In pictures:





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:

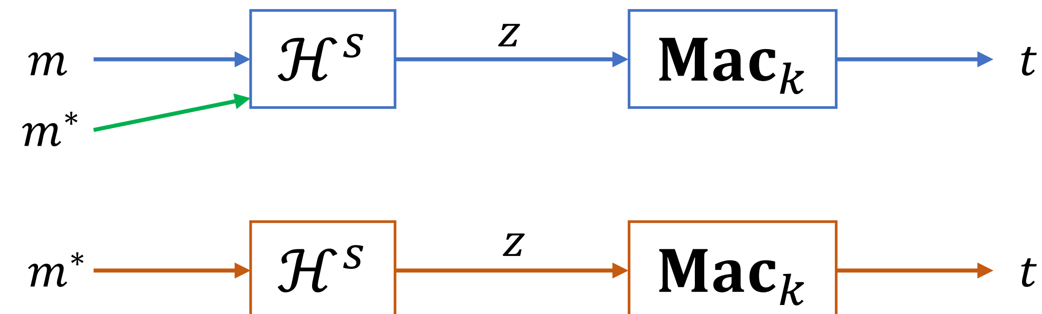
- (key generation) $\mathbf{KeyGen}'$: on input 1^n, outputs $k' \leftarrow \left(\mathbf{KeyGen}(1^{\mathrm{n}}), \mathbf{KeyGen}_{\mathrm{H}}(1^n)\right)$.
- (tag generation) $\mathbf{Mac}'$: on key (k, s) and message m, outputs $t \coloneqq \mathbf{Mac}_k(\mathcal{H}^s(m))$.

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:

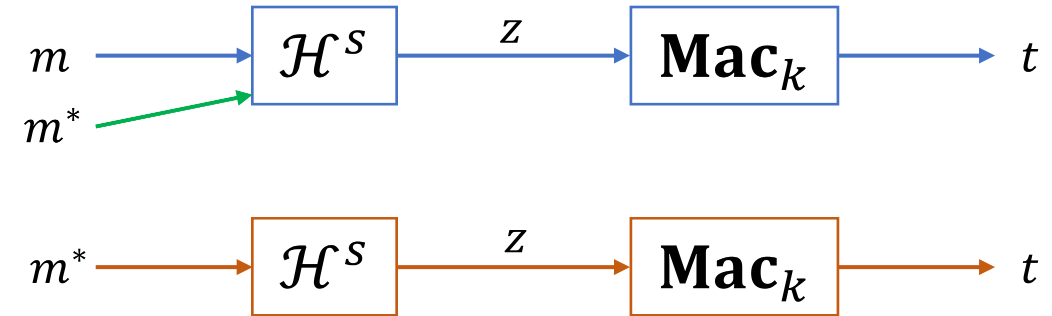
1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



HASH-and-MAC

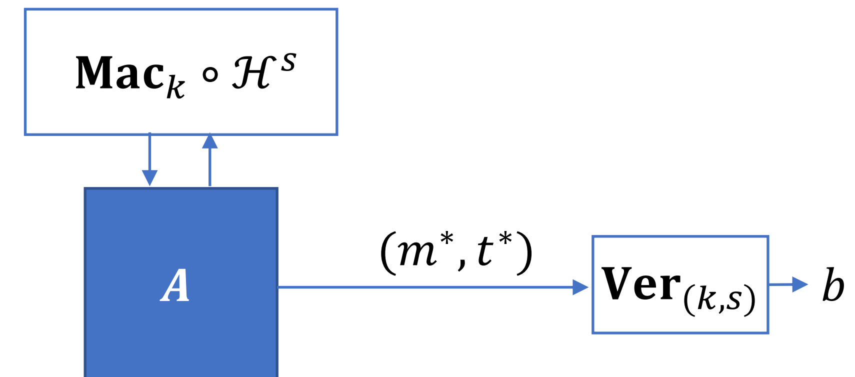
Proof idea: If forgery on m^* then either/or:

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:

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

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

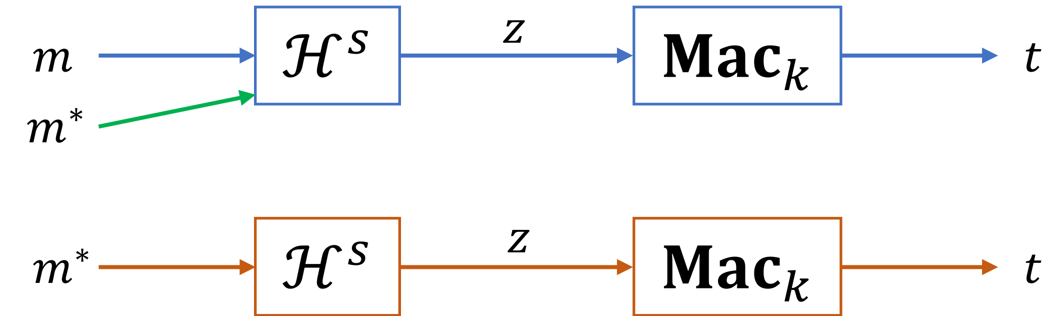
$\leq \Pr[\boldsymbol{E}] + \Pr[A \text{ wins MacForge}(\Pi') \wedge \overline{\boldsymbol{E}}]$.

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

HASH-and-MAC

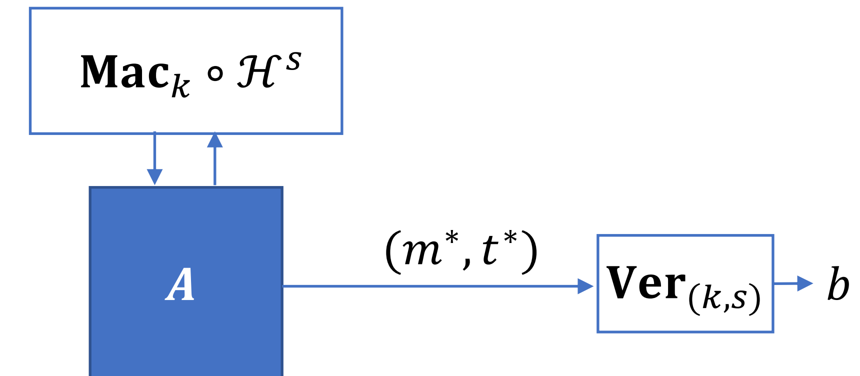
Proof idea: If forgery on m^* then either/or:

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 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:



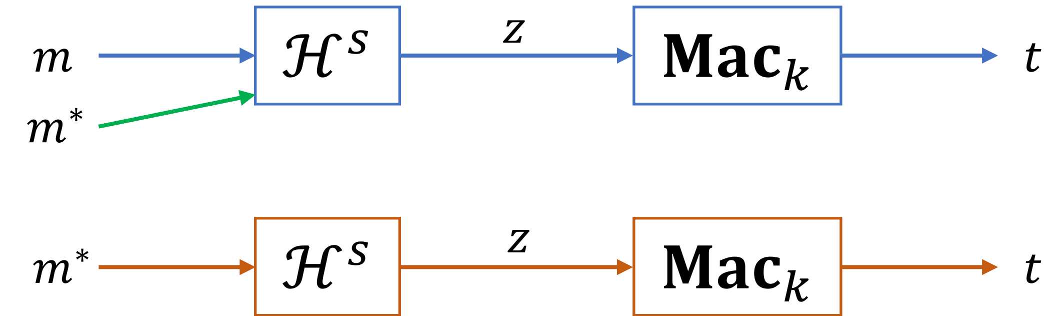
1. Receive hash key s as input. Sample $\mathbf{Mac}$ key k;
2. Run $\boldsymbol{A}$ with oracle $\mathbf{Mac}_k \circ \mathcal{H}^s$;
3. Output m^* and a random $m \in Q$.

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

HASH-and-MAC

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

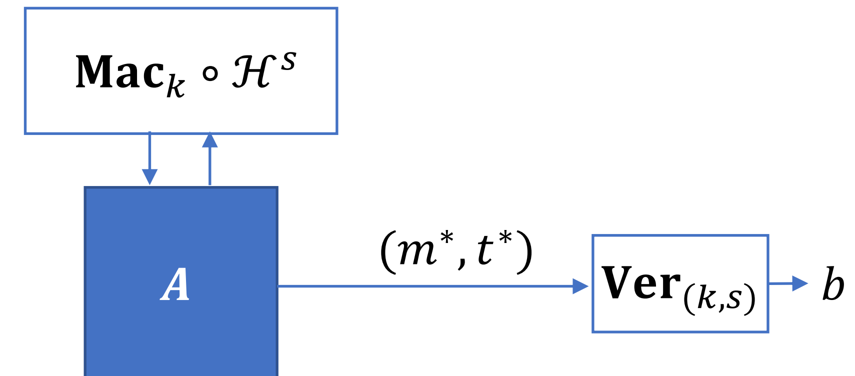
1. m^* is mapped to same z as some queried m: **collision!**
2. m^* is **not** mapped to same as any other: **forgery on Π!**



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:

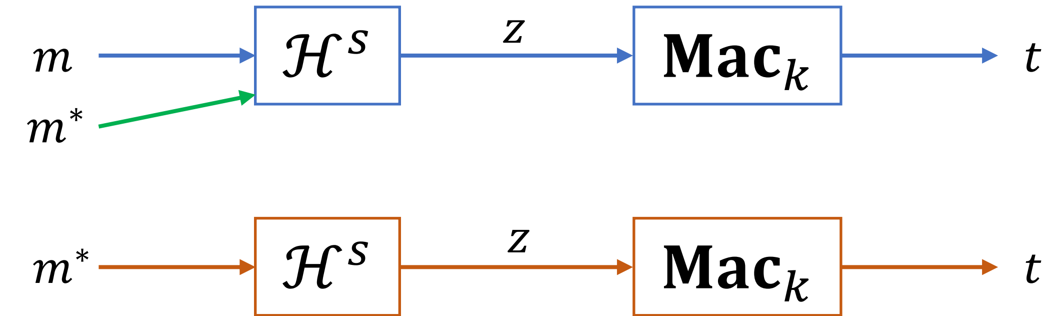
- probability that $\boldsymbol{A}$ wins the forgery game…
- … **and** for all *inputs z to $\mathbf{Mac}_k$ oracle*, $z \neq z^* := \mathcal{H}^s(m^*)$.

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

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!**



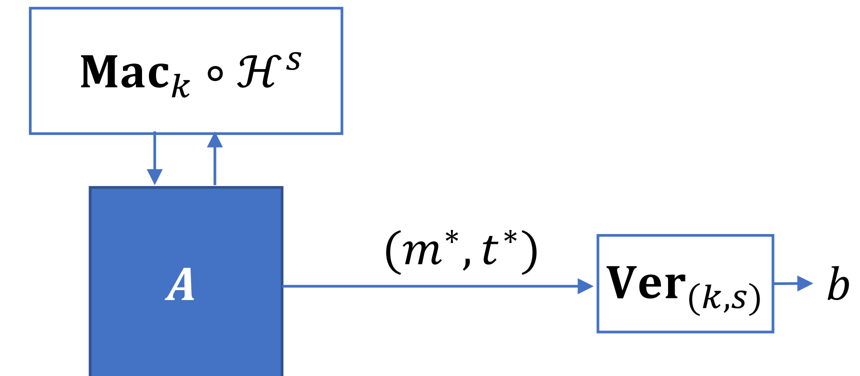
What's left:

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 $\mathcal{H}^s(m^*)$.

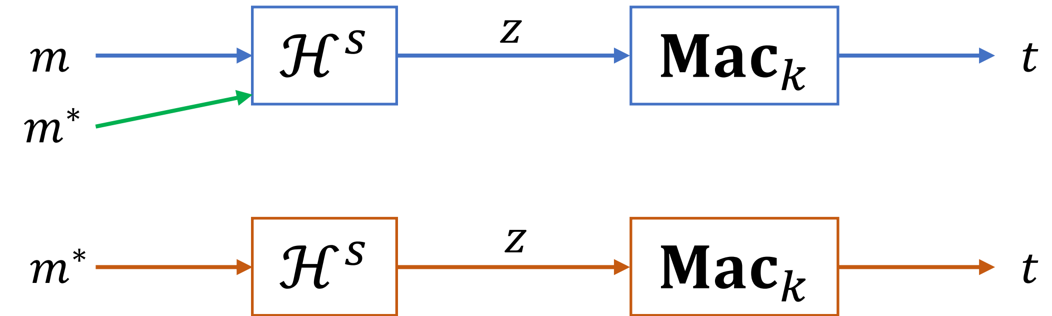


Check: probability this wins MacForge versus Π is exactly $\Pr[\boldsymbol{A} \text{ wins MacForge}(\Pi') \wedge \bar{\boldsymbol{E}}]$.

HASH-and-MAC

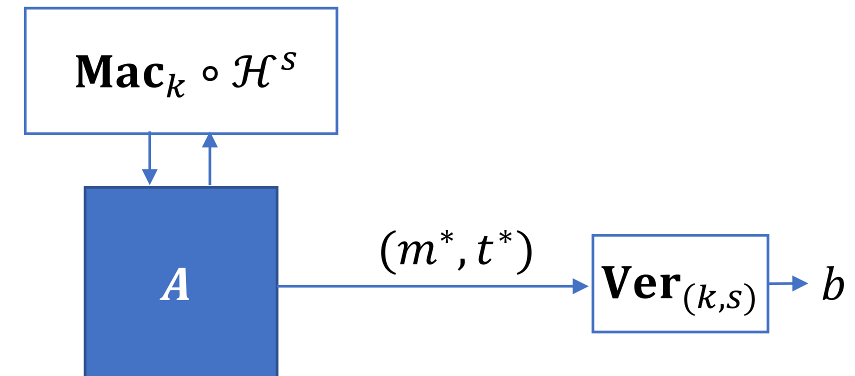
Proof idea: If forgery on m^* then either/or:

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:

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

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

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

$\leq \text{negl}(n) + \text{negl}(n) \leq \text{negl}(n).$

$\square$