# Cryptography 3

## CyberChallenge 2025

*Maxim Kovalkov*

# Hash functions

# Goals of cryptography

We know about some;
let's concentrate on:

- **message integrity**: how do we know that the message was neither *corrupted* nor *modified* by a malicious actor?
- **message authentication**: how do we know that the message actually came from the declared sender? (no impersonation; see Man-in-the-middle)

We know about asymmetric cryptography, key exchange, etc...
**Hash functions** are another ingredient we need to achieve those goals

# Hash functions

A hash function is a function that maps strings of **arbitrary length** to strings of a **fixed length $\mathbf{n}$**

$$H : \{0,1\}^* \rightarrow \{0,1\}^n$$

In order to be generic the above definition works for *strings of bits*
...but mostly we care about *bytes*, so from now on feel free to think of *character strings*

The output is usually called the **digest** of the input string

# Hash functions - Examples

To understand the technicalities, let's sort through some "non-useful" examples:

- The following **are** valid as hash functions.
  - $H(s) = 1$
  - $H(s) = s[0]$
  - $H(s) = s^e \mod n$
- The following **are not** valid hash functions.
  - $H(s) = s$
  - $H(s) = s^e$

## Hash functions - The interesting ones

To see hashes that are actually widely used, let's define what are **cryptographic hash functions**.

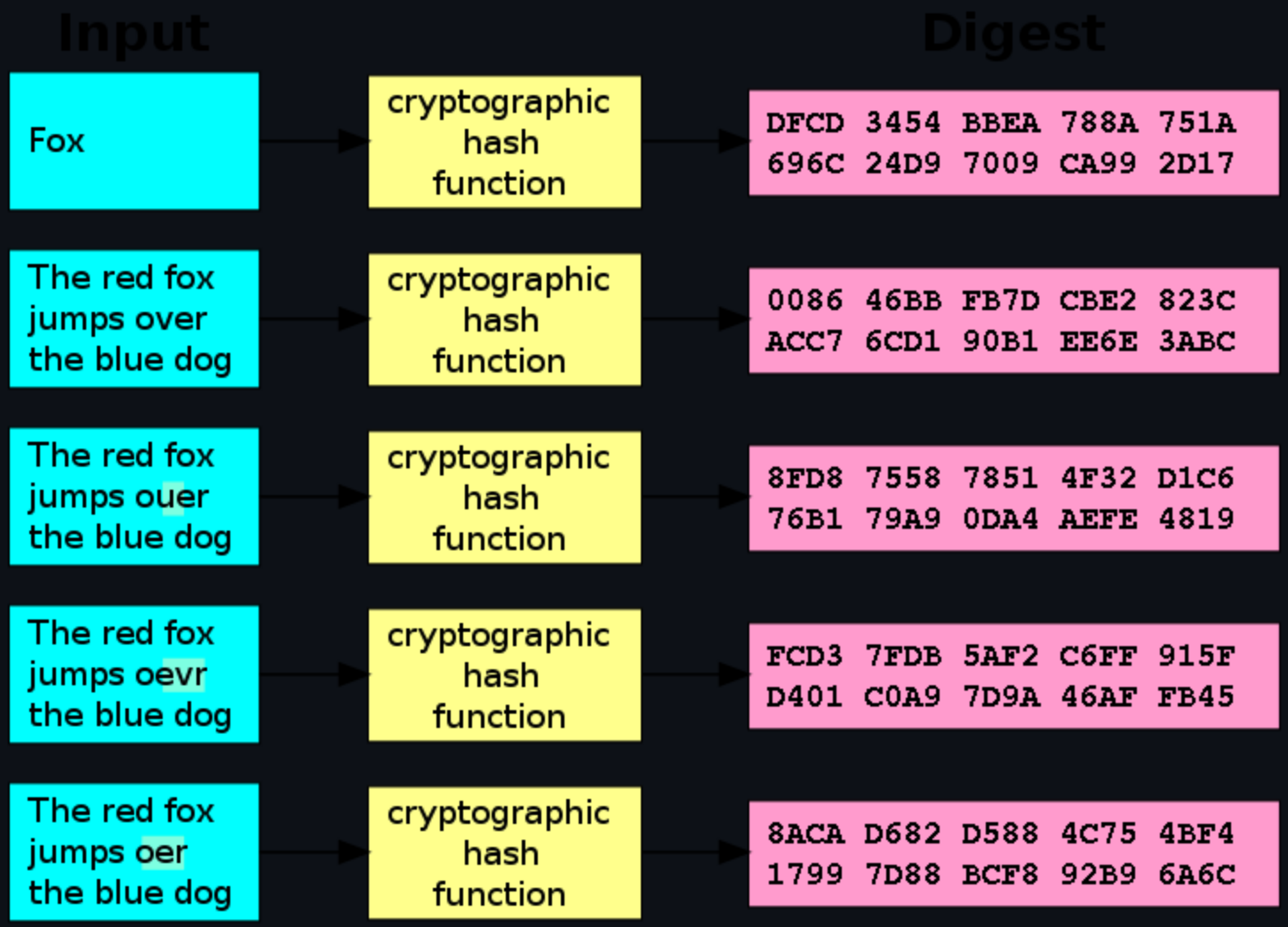
They are members of a family of hash functions with these useful traits:

- they are **one-way**, meaning that it is *not feasible* to get back to the message from its digest
- they are **collision resistant**, meaning they avoid to output the same digest for different messages
  - ...as much as possible: collisions **will** happen by definition, since the output size is *fixed* and *smaller that the input*
  - they *resist* message forgery

# Hash function resistance

- In a cryptographic function, a *small change* in the input *drastically changes* the digest
- This is called **avalanche effect**
- Though we can't avoid collisions altogether, we try to guarantee **no collisions for similar messages**

# Hash function resistance



## Collision resistance

We say that $H$ is **collision resistant** if there is no (1) **explicit** (2) **efficient** algorithm to find collisions for $H$

- **efficient**: feasible to execute in terms of time complexity
- **explicit**: allows for finding multiple different collisions

  ...in mathematical terms: the collision search algorithm $C(m, t)$ is dependent on some parameter $t$ so that you can run the same algorithm to produce different messages: like $c_1 = C(m, 1), c_2 = C(m, 2) \dots$

  ...then you have $c_1 \neq c_2 \neq \cdots \neq c_i$ while $H(c_1) = H(c_2) = \cdots = H(m)$

## Collision resistance

- Every hash function has *infinite* inputs that map to the same digest
- Inevitable... how else would you be able to map *arbitrary-length* strings to fixed-length ones?
- We want to find hash functions for which collisions are **less meaningful** and **more difficult to find**

## Three levels of resistance

- **Preimage resistance**:
  given $h$, it is hard to find a message $m$ such that $H(m) = h$
  (This is the property of *one-way functions*!)
- ***Second* preimage resistance**
  given $m$, it is hard to find a $m' \neq m$ such that $H(m) = H(m')$
  This is also called *weak collision resistance*
- **Strong collision resistance**
  it is hard to find $(m_1, m_2)$ (free to pick any!) such that $H(m_1) = H(m_2)$

Now let's focus on how to find collisions.

# Birthdays!

How many people do we need to put in a room to **guarantee** that there's at least two that share the same birthday?

Yep, 367 is correct.
(nope, leap years are not the point)

# Birthdays!

How many people should there be in a room to have an expected **probability of ≥50%** of finding at least two with the same birthday?

About half of the previous number?
More than 183? (how much more?)
Less than 183? (how much less?)

The answer: **only 23**!

This is the so-called *birthday paradox*

## Birthday attack

General attack against an $n$-bit hash function $H$ to find collisions

- Choose $2^{n/2}$ distinct random messages
  - For each of them, hash it and store the digest in a table
- Look for collisions in that table
- If no collisions occur, repeat the process

Similar to the birthday problem, we would need $2^n$ attempts to **guarantee** a collision, ...but only $1.2 \cdot 2^{n/2}$ are enough for a probability of success of 50%+

# Birthday attack - Analysis

The expected amount of iterations of the above algorithm is slightly above 2

So the birthday attack has a runtime of $O(2^{n/2})$

- still exponential, but now we know how a secure $n$ should be chosen

# Hash collisions and exploitations

A veeery nice online resource:
basic to advanced explanations & exploitation techniques!

https://github.com/corkami/collisions

## The Merkle-Damgård Paradigm

- A standard construction for hash functions
- Exploits collision-resistant *compression functions* (fixed $\times$ fixed $\longrightarrow$ fixed), to build collision-resistant *hashes* (arbitrary $\longrightarrow$ fixed)

## Compression function

A compression function takes *two* $x$-bit strings and outputs *one* new $x$-bit string

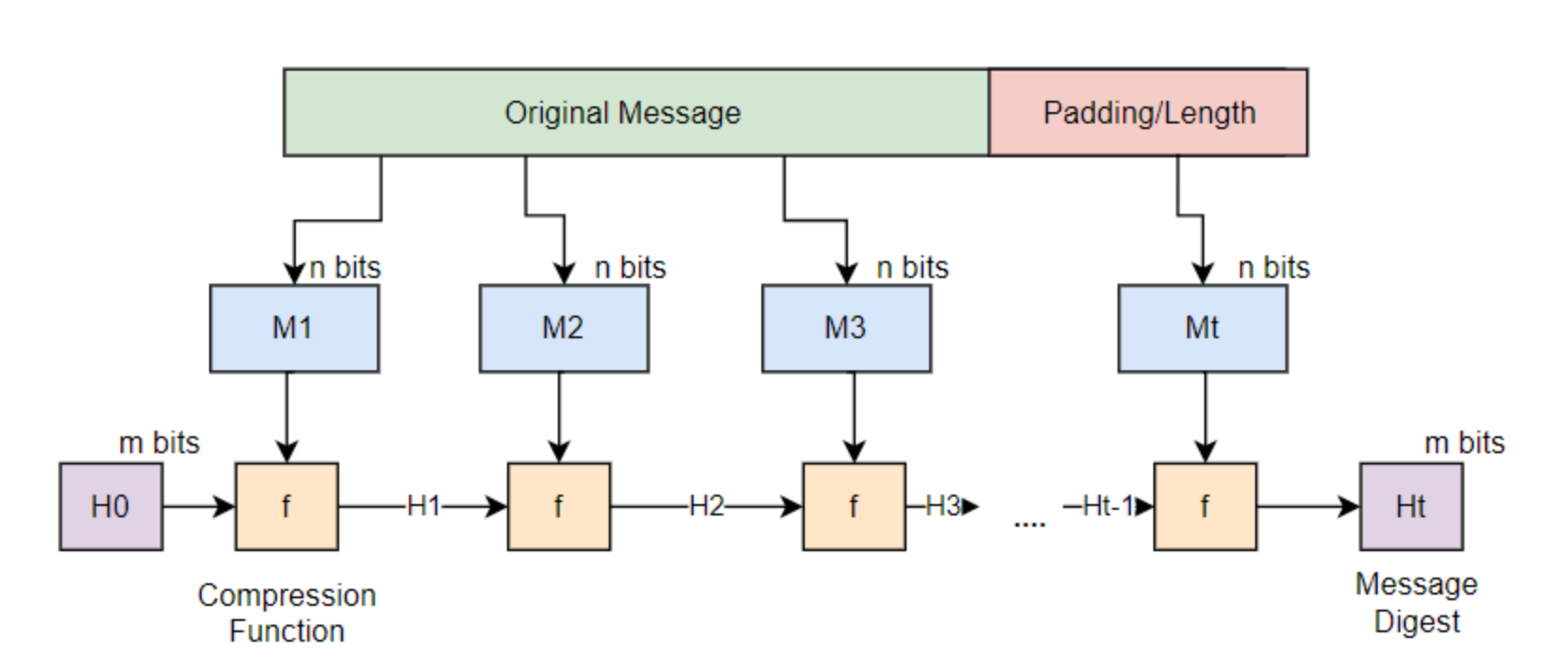
$$f : \{0,1\}^x \times \{0,1\}^x \to \{0,1\}^x$$
$$f(A, B) \mapsto C$$

# The Merkle-Damgård Paradigm

How to obtain a hash function from an $x$-bit compression function

- Pad the message $m$ to a multiple of $x$: obtaining $(m||pad)$
  different algorithm: append '1', fill with '0's, encode length in last 64 bits
- Split the message into $k$ blocks of size $x$
- Start with a fixed Initialization Vector (IV)
- Chain applications of the compression function to each block, from first to last

# The Merkle-Damgård Paradigm



## Theorem on Collision Resistance

What Merkle & Damgård proved is that if the compression function is collision-resistant, then the whole hash is collision-resistant.

Now it is enough to find a good compression function!

## The Davies-Meyer construction

A popular way of building compression functions, using building blocks we know!

- Start from a secure block cipher $E$ (CR_1.3 - Block Ciphers)
- The current block $m_i$ is used as the **key**
- The previous intermediate result $H_i$ (with $H_0 = IV$) is the **plaintext**
- The output is then XORed with the previous result

$$H_{i+1} = E(_{key=}m_i, _{pt=}H_i) \oplus H_i$$

## Comments on Davies-Meyer

Other variants of the construction are possible
...e.g. Miyaguchi and Preneel proposed 12 different and secure variants

- The most widely used hash functions (MD5, SHA-1, SHA2) use the Merkle-Damgård construction
- Davies-Meyer is used in most of the hashes with Merkle-Damgård structure
- "Natural" variants of Davies-Meyer are insecure:
  ...omitting the XOR → no
  ...XOR-ing with $m_i$ instead of $H_i$ → no
  All of those are relevant for the security of the compression function

# Standard hash functions

# MD5

Based on the above M.-D. paradigm; designed in 1991 by Ronald Rivest

- Widely used hash function producing 128-bit (16-byte) digest
- Initially designed to be a cryptographic hash function
- But **extensive vulnerabilities** have since been found
- Still used for non cryptographically-critical purposes
    - e.g. file integrity check

# SHA

A **standard** similar to DES and AES. (all hail NSA)

The Secure Hash Algorithm standard is a suite of 3 hash function families:

- **SHA-1**: a 160-bit hash. Usage no longer approved since 2010
- **SHA-2**: a set of 6 hash functions with different digest lengths. (choose from 224, 256, 384, 512 bits)
- **SHA-3**: introduced in 2012, supports the same lengths as SHA-2 with different internal structures.

# SHA-1

Based on Merkle-Damgård like MD5, introduced in 1995.

Widely used in standard protocols, like **TLS**, **SSL**, **PGP**, **SSH**.
Also used by versioning systems like **git** & **Mercurial**

- Known to be vulnerable since 2005!
- Organizations started phasing it out in 2010
- On 2017 the *SHAttered* attack went public: shattered.io (big deal!)
  Practical and dangerous attack: forgery of PDF files!
- Since 2017, applications (incl. all major browsers) stopped accepting SHA-1 based SSL certificates
- Since 2020, it is recommended to *always* use SHA-x(≥2) variants

# SHA-2

SHA-2 is a set of cryptographic hash functions designed by National Security Agency in 2001 and published by the NIST

The 6 hash functions mentioned before allow for 4 different output lengths:

- SHA-256 and SHA-512 use different shifting and additive constants, but their structures are virtually identical, differing only in the number of rounds
- SHA-224 and SHA-384 are truncated versions of SHA-256 and SHA-512, calculated with different initial values
- SHA-512/224 and SHA-512/256 are also truncated versions of SHA-512, but the initial values are generated differently

# Applications of hash functions

Let's come back to the utility of hash functions:

- Message integrity
- Message authentication (*)
- Digital signatures (*)
- Password verification
- Pseudo-random number generators

## Message integrity

Easy: send the message $M$ together with the digest $H(M)$

If $M$ has been altered (faulty communication or intentional forgery), the hash calculated hash would be (completely) different

Of course the attacker can not control both $M$ and $H(M)$

## Password verification

How do websites (or any service at all) store passwords?

Just taking the user's password and shoving into a database column is a ***BAD IDEA***

- Upon registration, put the password through a hash function and *store only the digest*
- Upon login, calculate the digest of the input password and compare with the stored one
- see also: **salting**, **rainbow tables**

## PRNG

Good cryptographic hash functions approximate a "*true random oracle*" (despite some flaws and gotchas!)

- A true random oracle should produce *uncorrelated* outputs $H(A), H(B)$ if $A$ and $B$ are distinct inputs
- $n = H(seed)$
- With a good seed and a suitable hash function, the output bits should be *uniformly distributed*

# MAC: Message Authentication Code

# Encrypted message authentication

In order to guarantee both integrity and authentication:

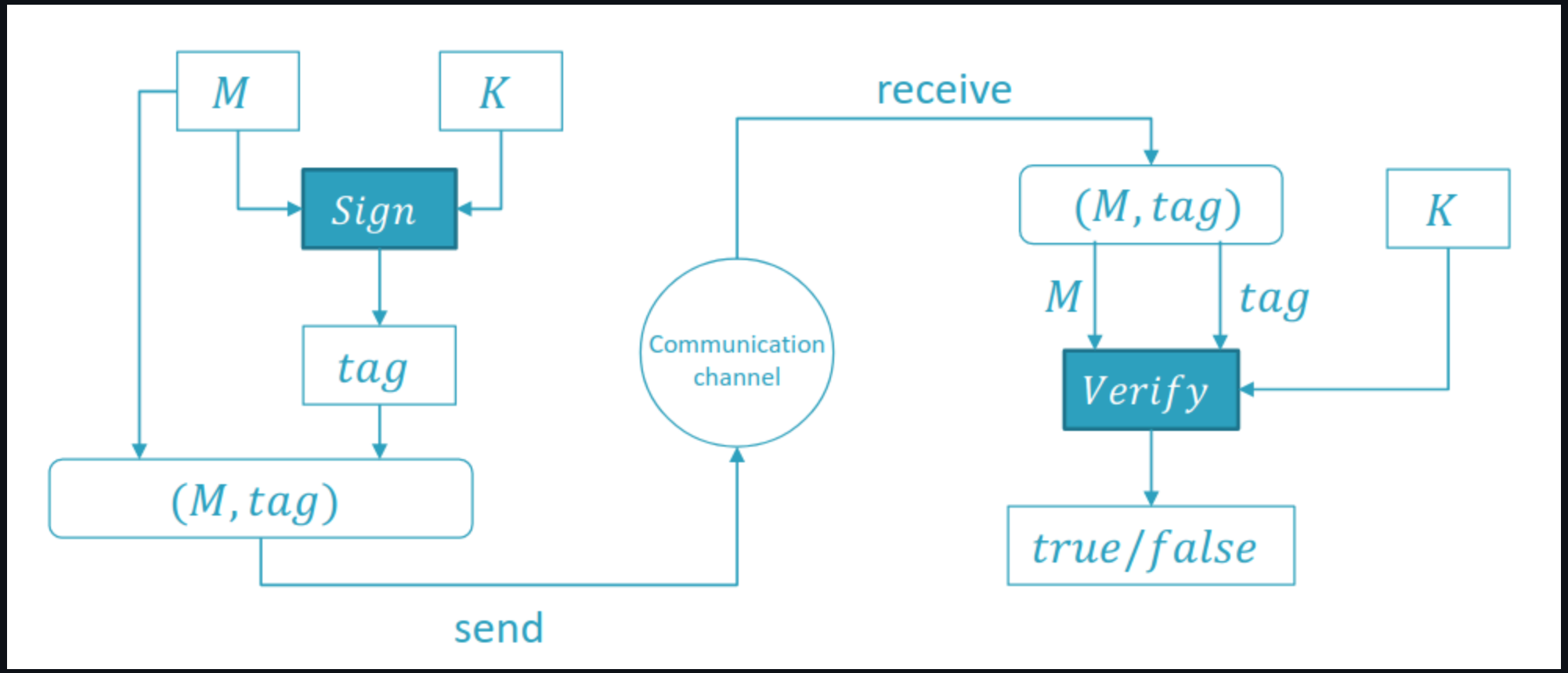
- A shared secret key can be generated using for example the Diffie-Hellman algorithm
- Cryptographic hash functions ensure integrity
- If the digest is encrypted, authentication is reached
- If also the message is encrypted, confidentiality is reached

# Message Authentication Code (MAC)

A Message Authentication Code (MAC) is a pair of functions, Sign and Verify, such that:

- Sign takes a message $M$ of arbitrary length and a key $k$ and produces a fixed-length string, called $tag$
- Verify takes the message $M$, the key $k$ and the $tag$, and outputs *true* if the tag is valid and *false* otherwise

# Message Authentication Code (MAC)



# Hash vs MAC

With hash functions we can reach integrity but not authentication

With MACs we can reach both integrity and authentication

## Attacks on MAC

A MAC can be subject to several types of attacks by external attackers who do not know the key

- Existential Forgery Attack: The attacker can create a valid message M and a tag for M without knowing the key. The attacker defines both M and the corresponding tag.
- Selective Forgery Attack: Given a message M, the attacker is able to produce a valid tag for M.
- Universal Forgery Attack: The attacker can create a valid tag for any possible message M. This attack is the most powerful and implies the total break of the MAC scheme.

# MAC from symmetric ciphers

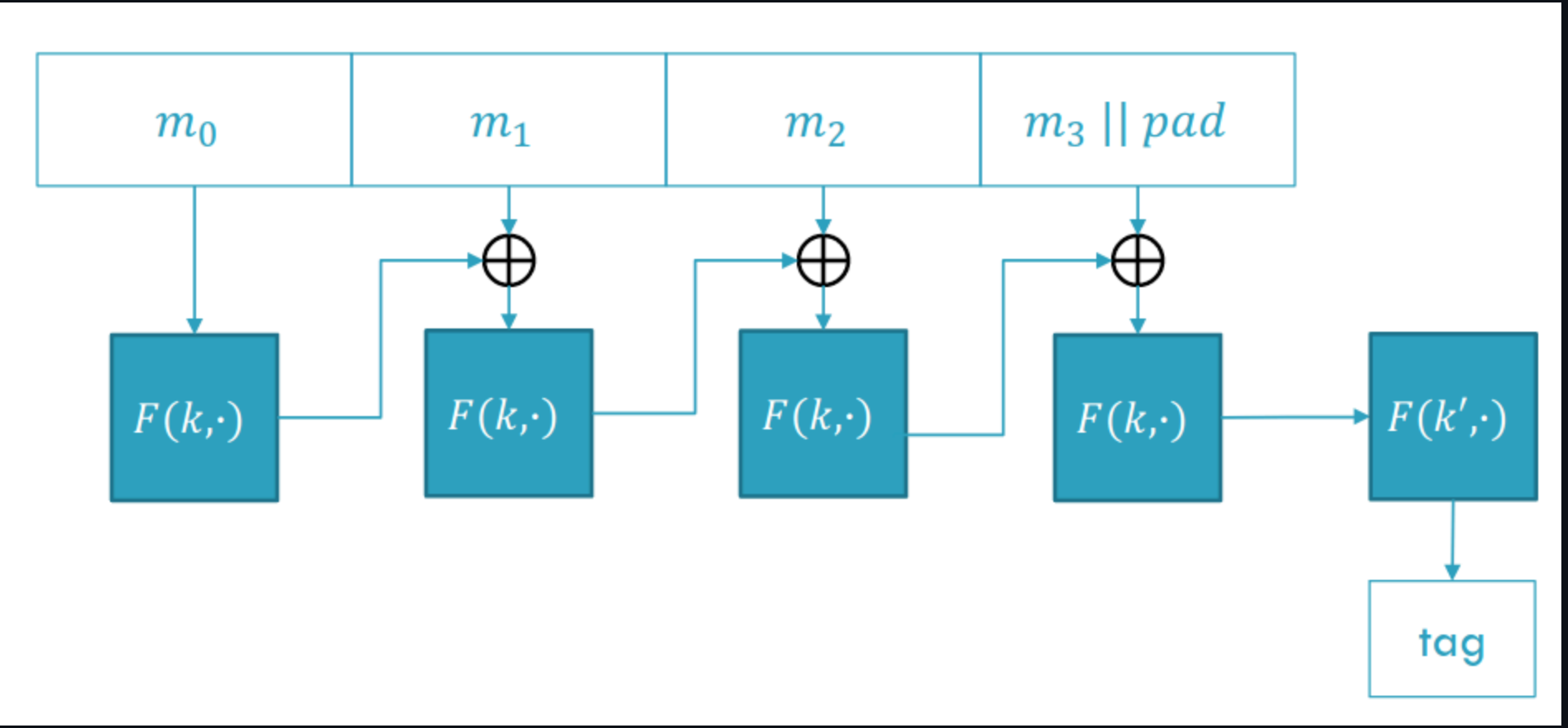
In this section we introduce two implementations of a secure MAC

- The CBC-MAC, built from the CBC mode of operation
- The Nested-MAC (NMAC), a more natural construction

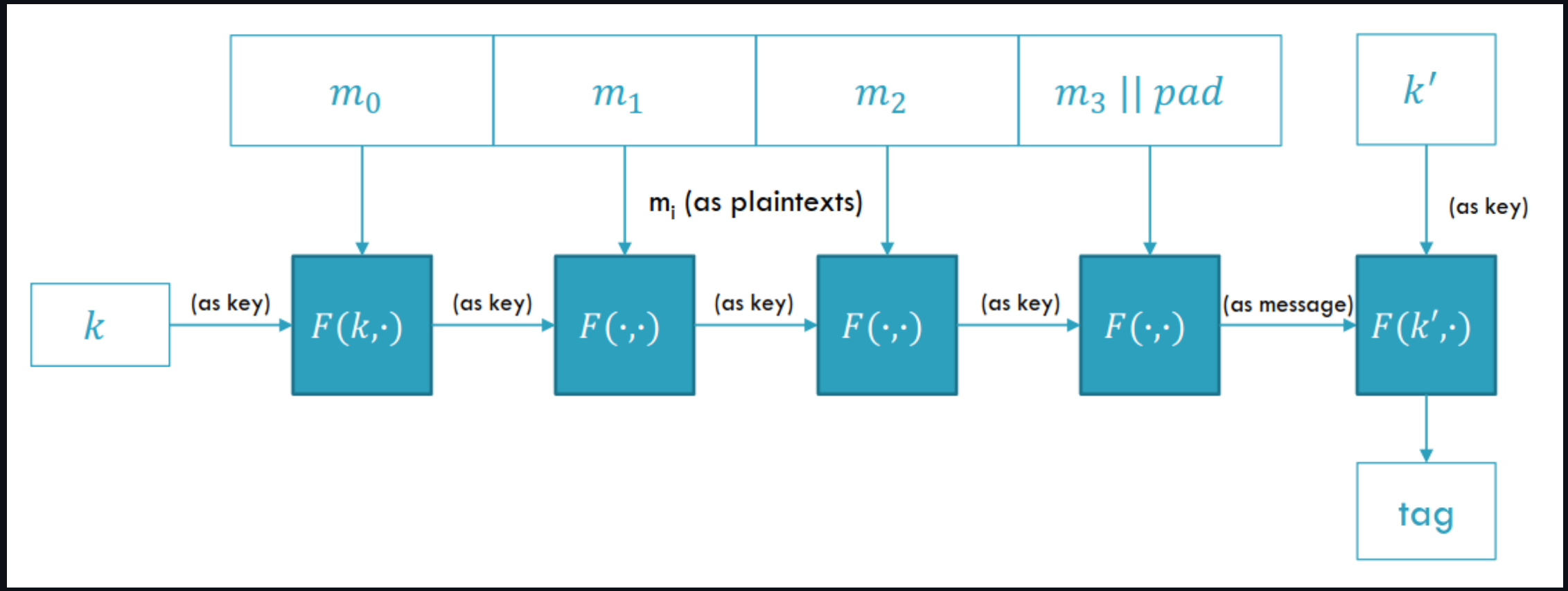
Both constructions need:

- A secure block cipher $F(key, message)$
- A pair of keys $(k, k')$ for the block cipher

# CBC-MAC scheme



# NMAC scheme


$m_0$
$m_1$
$m_2$
$m_3 \,||\, pad$
$k'$
$m_i$ (as plaintexts)
(as key)
$k$
(as key)
$F(k,\cdot)$
(as key)
$F(\cdot,\cdot)$
(as key)
$F(\cdot,\cdot)$
(as key)
$F(\cdot,\cdot)$
(as message)
$F(k',\cdot)$
tag

## One-chosen-message attack

Without the last encryption block we can perform a so called 1-chosen message attack

- For example, in CBC-MAC:
- Choose an arbitrary one-block message $m$
- Get the associated tag $t$
- Now $t$ is also the tag for the 2-block message $(m, t \oplus m)$

# CBC-MAC / NMAC Comparison

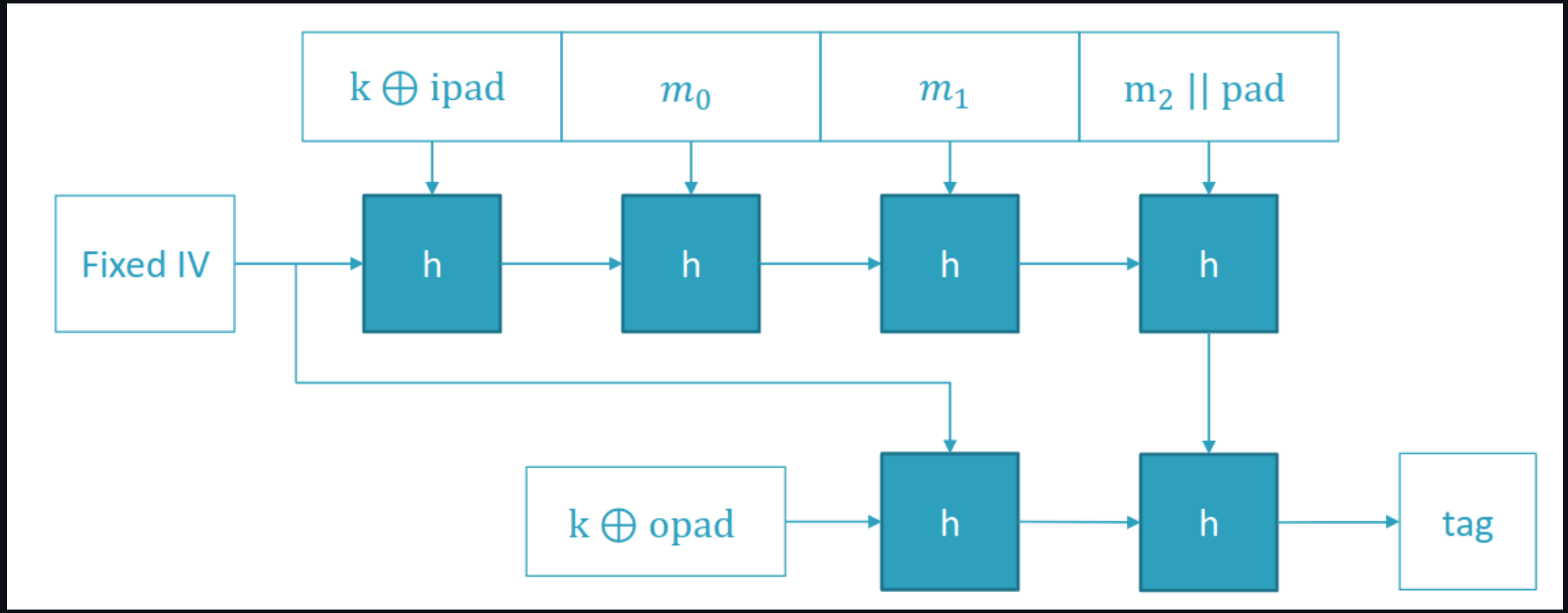
CBC-MAC is usually used with AES:

- CCM encryption mode used in 802.11i (WPA2)
- NIST standard called CMAC

NMAC is not used with AES:

- Changing the key each time is too slow (because of the key schedule of AES)
- It is instead used with hash functions (HMAC)

# MAC from Hash functions: HMAC



## Naïve approach

We want to build a MAC using Hash functions.

First idea:

- Take a key $k$, a message $m$ and a collision resistant hash function $H$
- Build the MAC as $\mathsf{MAC}(k, m) = H(k||m)$

Issues?

## Length extension attack

This is vulnerable to a length extension attack.

We can forge MACs for new messages:

- Take a message $m'$ and $S(k, m)$
- Then $S(k, m||m') = H(k||m||pad||m')$
- Since $H(k||m||pad) = H(k||m)$ we simply need to compute the compression for the new blocks, not caring about the key!

## A better idea

We now explain a popular strategy, called HMAC
(hash message authentication code)

Ingredients:

- A collision resistant* hash function $H$
  (requirement can be relaxed, but would be difficult to explain)
- Two padding strings $ipad$, $opad$ (fixed and known)
- A secret key $k$ and a message $m$

## HMAC scheme

HMAC can be used to verify both integrity and authentication of a message, at the same time.

Any available hash function can be used, like SHA-1 or SHA-2, without having to modify the scheme.

The resulting version of the MAC is called HMAC-X, where X is the used hash function.

## HMAC scheme

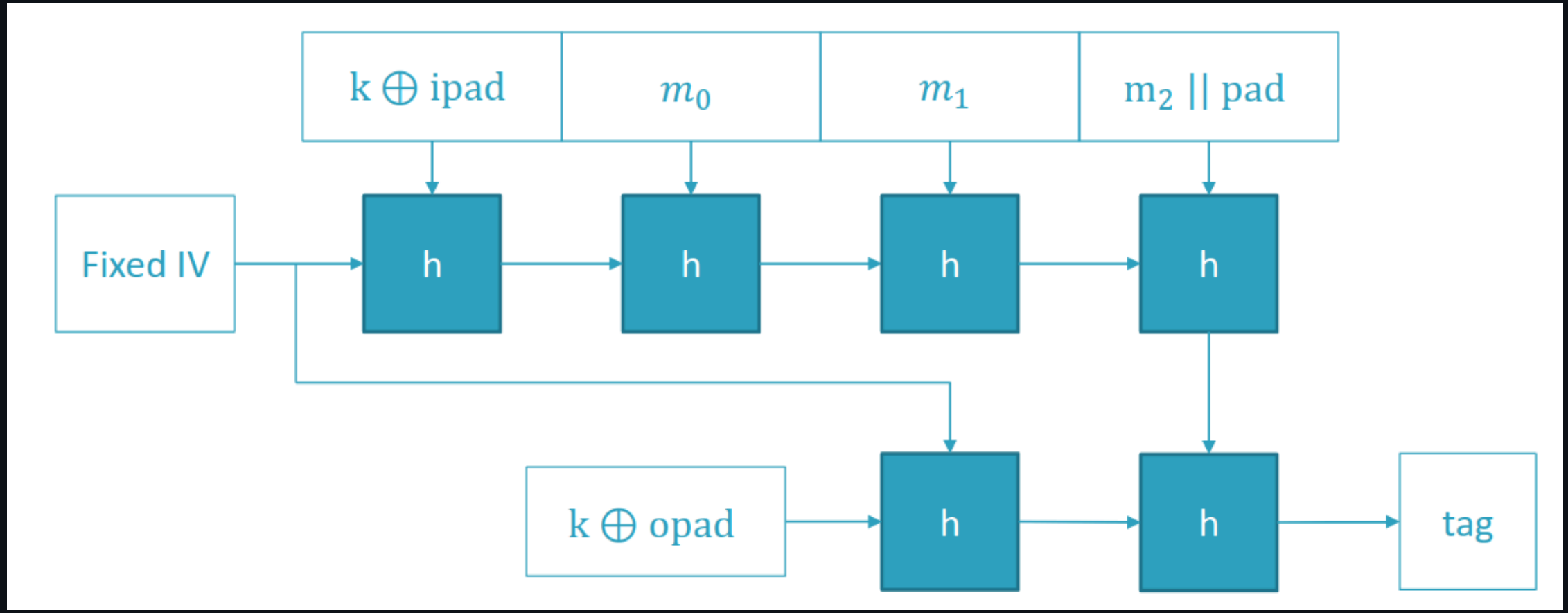
We define HMAC as:

$$\mathsf{HMAC}_k(m) = H\left((k \oplus opad)||H\left((k \oplus ipad)||m\right)\right)$$

Note:

- $(k \oplus opad)$ and $(k \oplus ipad)$ are the two "secret keys"
- The construction is very similar to NMAC, except the two keys are *related*
- Length extension attack cannot be performed, since it needs the knowledge of the result of internal call of the hash function $H$

# HMAC scheme



# Digital signatures

# Another security criterion

We know *Integrity*, we know *Authentication*...

Another desirable property is **non-repudiation**:

- Protection against an actor falsely denying the performance of an action
- Provides the capability to determine whether a given individual took a particular action such as creating information, sending a message, approving information, and receiving a message

# Digital signatures

Informally: a digital signature is like a MAC, but with public key cryptography:

No need for a shared secret key!

# Hash vs MAC vs Signatures

Pros of digital signatures

- No need of sharing a key
- Non-repudiation property

Cons of digital signatures

- Slow compared to MACs

## Signatures in practice

In practice, a digital signature is a pair of functions, Sign and Verify, such that:

- Sign takes a hash of a message of arbitrary length and a key and produces a fixed-length string, called signature
- Verify takes the hash of the message, the key and the signature, and outputs true if the signature is valid and false otherwise

Hashes help avoid the times when messages are either too short or too long!

# Digital signatures from RSA

We already talked about this veeery briefly:
The RSA encryption and decryption keys can be *switched around* for use with digital signatures!

- The Sign function for a message $m$ is: $s = m^d \mod n$
- The Verify function for a message $m$ is: $m = s^e \mod n$

Issues?

# Forgery

Some signatures are independent from the value of d:

- The signature of 0 is always 0
- The signature of 1 is always 1
- The signature of n-1 is always n-1

→ Avoid them!!!

## Blinding

Using the homomorphic properties of RSA, we can sign an arbitrary message M without asking directly to the oracle to sign it:

- Select a value R
- Ask to sign $(R^e M) \rightarrow \quad \textsf{Sign}(R^e M) \equiv (R^e M)^d \equiv RM^d \pmod{n}$
- Use the multiplicative inverse of $R$ to get a signature for $M$ from the signature of $(R^e M)$:
- $\textsf{Sign}(M) \equiv \textsf{Sign}(R^e M) R^{-1} \equiv RM^d R^{-1} \equiv M^d \pmod{n}$

# Digital Signature Algorithm (DSA)

- In 1982, the US government asked for proposals for digital signature standards
- In 1991, the Digital Signature Algorithm (DSA) was proposed by NIST and standardized
- Since 2019, DSA is no longer recommended by NIST, and it has been mostly replaced by its elliptic curve-based equivalent algorithm (ECDSA)

# DSA Overview

DSA is based on 4 algorithms:

- Parameters generation
- Key generation
- Sign algorithm
- Verify algorithm

## Parameters generation

- Pick a cryptographic hash function $H$ (usually SHA1)
- Pick a prime number $q$
- Pick a prime number $p$ such that $p-1$ is multiple of $q$
- Pick a number $h$ in $\{2, 3, \ldots, p-2\}$ (usually $h = 2$) and $g = h^{(p-1)/q} \mod p$
- The values $(H, p, q, g)$ are the (publicly shared)parameters of the DSA instance

## Key generation

Each user generates a key as follows:

- Pick $x$ in $\{1, 2, \ldots, q-1\}$
- Set $y = g^x \mod p$
- $x$ is the private key, $y$ is the public one

## Signing

A signature of a message $m$ is made as follows:

- Pick a random value $k$ (called the *nonce*) in $\{1, \ldots, q-1\}$
- Compute $r = (g^k \mod p) \mod q$
- Compute $s = (k^{-1}(H(m) + xr)) \mod q$
- The pair $(r, s)$ is the signature of $m$

## Verifying

Given a signature $(r, s)$ and a message $m$, the verification is made as follows:

- Compute $u_1 = H(m)s^{-1} \mod q$
- Compute $u_2 = rs^{-1} \mod q$
- $v = (g^{u_1}y^{u_2} \mod p) \mod q$
- signature is valid? $\iff r = v$

# Nonce reuse in DSA

The main problem for DSA stems from the choice of the nonce $k$

- In the next slide, we show just what happens if $k$ is used more than once
- In general, using not random (or biased) nonces is a bad idea

## Nonce reuse in DSA

Suppose to have two messages $m_1, m_2$ signed by the same user with the same nonce $k$

Let's call the signatures $(r_1, s_1)$ and $(r_2, s_2)$

We can simply recover the private key $x$ as follows:

$$x = (s_2 H(m_1) - s_1 H(m_2)) (r_2 s_1 - r_1 s_2)^{-1} \mod q$$

# Attacks against PRNGs

## Pseudo-random numbers

A Random Number Generator (RNG) is a utility or device that produces a *sequence of numbers* within some interval `[min, max]` while guaranteeing that values appear **unpredictable**
(Randomness can have multiple meanings, but we won't go into this)

A *Pseudo-Random Number Generator* (PRNG) is an **algorithm** or **hardware device** that generates a sequence of random bits / numbers

- NB: SW and pure-HW solutions will *always* be "pseudo"!
- PRNGs usually take a **seed**, which defines the entire random sequence

# PRNGs in cryptography

Not every RNG needs to be cryptographically secure!

Most of them are designed to work well for *simulations*
PRNGs are generally **not** cryptographically secure
However, researchers have tackled this problem: Cryptographically Secure PRNGs (CSPRNG) have since been designed

# Attacks on PRNGs

The types of PRNG that we will consider:

- Linear congruential generator (LCG)
- Mersenne Twister
- Linear-feedback shift register (LFSR)

Attacks consist in observing the generated numbers (to then predict the next numbers in the sequence)
Knowing internals of the PRNG algorithm can help break it

# Linear Congruential Generator

Simplest known PRNG!

Three **parameters**: $n, a, b$ called *modulus*, *multiplier* and *increment* respectively

The **seed** is taking as the starting value $x_0$

Every next value is produced as:

$$x_{i+1} = (ax_i + b) \mod n$$

## Issues of LCGs

LCGs can be easily broken: it is enough to take some observations

Let's entertain different scenarios of "increasing difficulty":

- $n, a$ and $b$ are known
- $n$ and $a$ are known
- only $n$ is known
- nothing is known

## Break LCGs: n, a and b known

In all these scenarios, we may have access to observations $x_1, x_2, \ldots, x_k$, and we want to predict $x_{k+1}$

- $n, a, b$ are known
- only observation $x_k$ is needed
- To break: simply compute $x_{k+1} = (ax_k + b) \mod n$

## Break LCGs: n and a known

- $n, a$ known
- observations $x_{k-1}, x_k$ are needed
- To break:
  - First find $b$

$$\begin{aligned} x_k &= (ax_{k-1} + b) \mod n \\ \Rightarrow \quad b &= (x_k - ax_{k-1}) \mod n \end{aligned}$$

  - With $n, a, b$ known, solve as in previous example

# Break LCGs: only n is known

- Only $n$ is known
- We require observations $x_{k-2}, x_{k-1}, x_k$
- To break:
  - Find $a$:

$$\begin{cases} x_k = ax_{k-1} + b \pmod n \\ x_{k-1} = ax_{k-2} + b \pmod n \end{cases}$$

$$\overline{x_k - x_{k-1} = a(x_{k-1} - x_{k-2})}$$

$$a = (x_k - x_{k-1})(x_{k-1} - x_{k-2})^{-1} \pmod n$$

  - Solve as before with $n, a$ known

# Break LCGs: nothing is known

- We only know that the RNG type is LCG, and have observations $x_k, x_{k-1}, x_{k-2}, \ldots$
- From LCG generation expression, we define $s_i$:

$$\begin{aligned} x_k &\equiv a x_{k-1} + b \pmod{n} & s_k &= x_k - (a x_{k-1} + b) \\ x_{k-1} &\equiv a x_{k-2} + b \pmod{n} & s_{k-1} &= x_{k-1} - (a x_{k-2} + b) \\ &\vdots & &\vdots \end{aligned}$$

- Modular congruence relations say that every $s_i$ is a whole multiple of $n$
- So $\gcd(s_1, \ldots, s_k) = n$ with high probability (more likely the more observations we have!)

# Break LCGs: nothing is known

- We can not compute the $s_i$ terms as we have missing information
- We define another sequence that produces multiples of $n$, and use $\gcd$ to recover $n$
- Let $t_i = x_i - x_{i-1}$. Note that this way, also $t_i \equiv a t_{i-1} \pmod{n}$
- It can be proven that $t_{i-1} \cdot t_{i+1} - t_i^2 \equiv 0 \pmod{n}$
- We take as many values of $(t_{i-1} \cdot t_{i+1} - t_i^2)$ as needed, and apply $\gcd$ to recover modulus
- Now we solve as before, with $n$ known

## Mersenne Twister

Another famous PRNG is the Mersenne Twister (MT)

- It is by far the most widely used PRNG in practice
- Its name derives from the Mersenne prime numbers, primes in the form of $2^n - 1$, used in the algorithm
- It is usually used in its MT19937 version, where 19937 means that you need $2^{19937}$ calls of the function to obtain a duplicate number from the PRNG

## Issues of Mersenne Twister

- Despite its ubiquity, it's **not** cryptographically secure
- By observing enough iterations, it is possible to recover the internal *state vector* from which future iterations are produced and allows one to predict all future iterations
- In case of the MT19937, you need to observe 624 words of 32 bits to crack it

# Break (also) the Mersenne Twister

**Untwister** is a seed recovery tool for common PRNGs

github.com/altf4/untwister

Supported RNGs include:

- MT19937 version of Mersenne Twister
- `rand()` from glibc; `php_mt_rand` in PHP stdlib
- Ruby's MT variant `DEFAULT::rand()`
- Java's `Random` class

# Linear Feedback Shift Register

LFSRs are used as PRNG with application for example in stream ciphers

A LFSR is defined by:

- A bit size
- A characteristic polynomial
- An initial state

## Recovering internal state

Assume that the characteristic polynomial of a $n$-bit LFSR is known

It is possible to completely recover the internal state given a binary output sequence of length $n$

With the internal state it is possible to go backward and forward and recover all the output sequence

# Berlekamp Massey algorithm

Given some binary observation of a LFSR is it possible to recover its characteristic polynomial

The *Berlekamp–Massey algorithm* is an algorithm that will find the shortest linear feedback shift register (LFSR) for a given binary output sequence
*(See Weisstein, Eric W. "Berlekamp-Massey Algorithm." From MathWorld - A Wolfram Web Resource)*
https://mathworld.wolfram.com/Berlekamp-MasseyAlgorithm.html

An online calculator of Berlekamp-Massey Algorithm: http://bma.bozhu.me

# That's all, folks!

CyberChallenge.IT 2025 — Cryptography 3 — Maxim Kovalkov