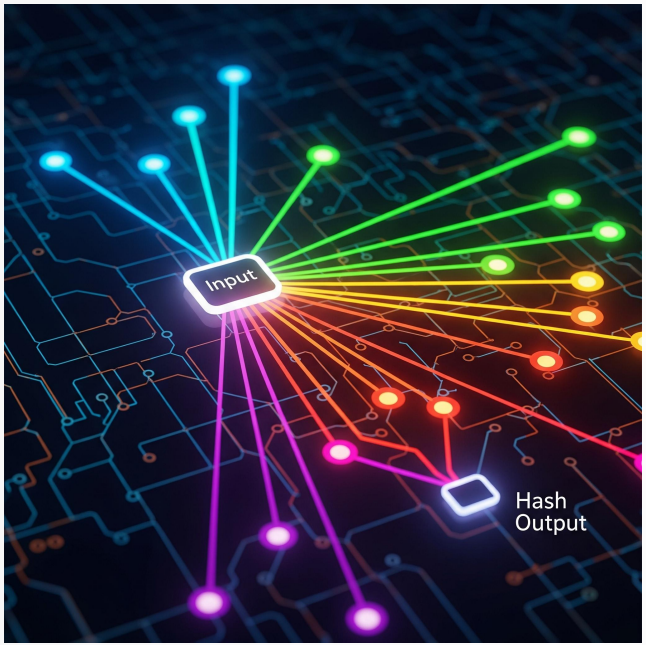
Cryptographic hashing functions



Making hashing functions stronger

done by introducing two extra-requirements on them

- collision resistance
- irreversibility

Collision resistance

A hash function $h: D \rightarrow R$ is called ***weakly collision resistant*** for $x \in D$ if it is **hard** to find $x' \neq x$ such that $h(x')=h(x)$

A function $h: D \rightarrow R$ is called ***strongly collision resistant*** if it is **hard** to find x, x' such that $x' \neq x$ but $h(x)=h(x')$

hard means computationally infeasible

Clarification

- infeasible means that computation would take too much time
    - years, or more
- birthday bound allows saying that "short" fingerprints are feasible
    - today short means 160 bits or less
    - SHA-1 was a cryptographic function, until 2014

Differences between "weak" and "strong" form

- the strong form requests that it is hard to find messages that collide
    - there is no input
    - it is pure existence, i.e., no other condition is put
- the weak form has one message in input
    - it requests to find a colliding message
- "strong" and "weak" terms used because strong ⇒ weak

Strong ⇒ weak

- proof: we show ¬weak ⇒ ¬strong (equivalent)
- given h, suppose there is poly alg. A_h: $A_h(x)$ = x' s.t. h(x) = h(x')
- we construct poly alg. B_h s.t. $B_h()$ = (x, x') s.t. h(x) = h(x'):
  1. arbitrarily choose x
  2. return (x, $A_h(x)$)

Requirements of cryptographic hashing functions

1. (strong) collision resistance
2. non-reversibility
    - given h = H(M), it is hard to determine M

Note that for no function a math proof that it is cryptographic hashing has been found, but there are several functions conjectured to be cryptographic

Terminological note

in modern cryptography terminology has been updated

- preimage resistance denotes non-reversibility
- second preimage resistance denotes what was traditionally called weak collision resistance
- collision resistance denotes what was traditionally called strong collision resistance

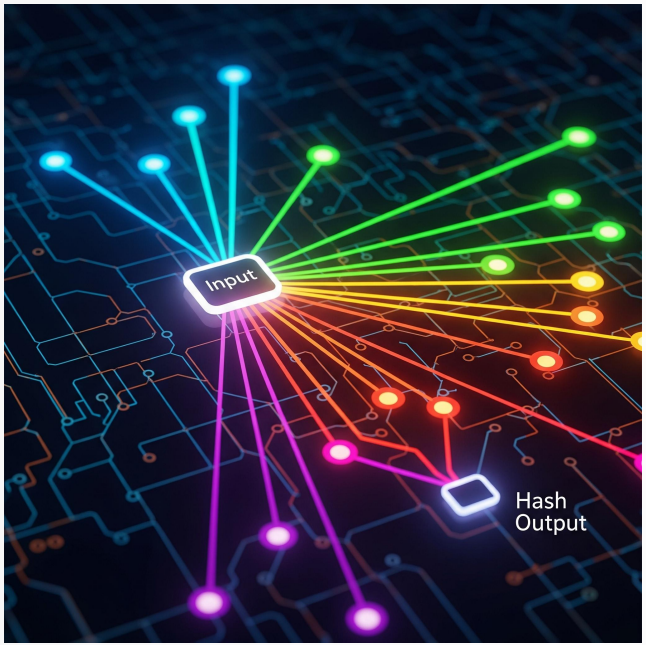
Summary on cryptographic hashing functions

- Deterministic
- Efficient
- Preimage resistance
- Second preimage resistance
- Collision resistance
- Avalanche effect
- Fixed output length

Security of cryptographic hashing functions

- Birthday bound holds even for cryptographic hashing functions
    - same number of collisions, just better distributed and random
    - birthday attack in theory is possible
- The improvement is the (much) greater difficulty in finding collisions
- Preimage resistance offers several security applications
    - Privacy, authentication, blockchain etc.
    - Used for deriving keys from passwords

Cryptographic hashing functions: design and construction



Recap on cryptographical hashing functions

- Maps input of any length to a fixed-size digest
- Properties:
  - Preimage resistance
  - Second preimage resistance
  - Collision resistance
- Applications
  - Integrity
  - Digital signatures
  - Password hashing

Design goals

- Output should be:
  - Unpredictable
  - Uniformly distributed
  - Sensitive to input changes (avalanche effect)
- Efficient computation
- Secure against known attack strategies

Historical designs

- MD5 – Broken, deprecated
- SHA-1 – Vulnerable to collisions
- SHA-2 (it's a family) – Still widely used
- SHA-3 (Keccak, a family) – Sponge-based, more secure

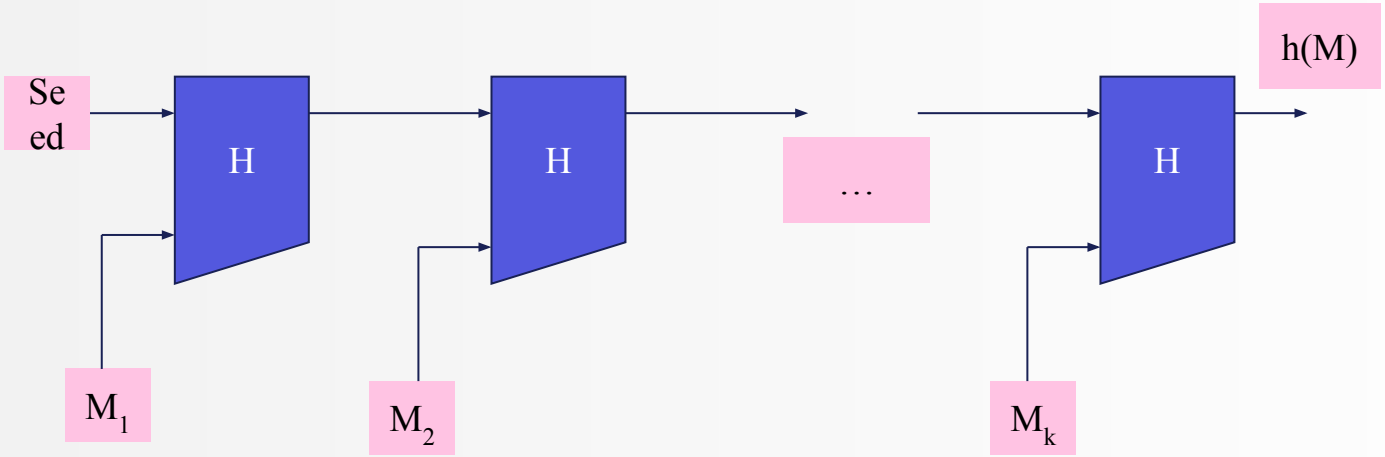
Basic info

MD5	Once widely used, but now recognized as insecure for typical security purposes due to known collision attacks. Avoid for new applications requiring cryptographic security
SHA-1	While not as easily broken as MD5, practical collision attacks have been demonstrated. It's largely considered unsuitable for new applications requiring strong collision resistance (e.g., digital signatures)
SHA-2	This is a family of hash functions (e.g., SHA-256, SHA-512). It's currently considered secure for most applications and is widely implemented in various security protocols and systems
SHA-3	Also known as Keccak, this is a family of hash functions that uses a different underlying design called a sponge construction. It was developed to offer an alternative to SHA-2 with a distinct cryptographic foundation, enhancing overall security resilience. Considered very secure

Merkle–Damgård construction

- Processes message in blocks
- Uses a compression function
- Adds padding
- Final hash = last output
- Strength: simple and modular
- Used in the design of SHA-1 and SHA-2 (and others)
- Weakness: length extension attacks (later)

Merkle–Damgård diagram



H: D ⟶ R (fixed sets, typically $\{0,1\}^n$ and $\{0,1\}^m$)

Merkle–Damgård: function H

- It is called Compression Function
- Must behave like a pseudorandom function
- Common structures
- Davies-Meyer (block cipher-based)
- Requirements
    - Non-linearity
    - Diffusion
    - Confusion

examples

- Davies-Meyer
  - denote by O_i the output of block i
  - $H_i(\cdot) = O_i = E_{M_i}(O_{i-1}) \oplus O_{i-1}$
  - E is a block cipher, and M_i (message block) is used as key
- SHA-1/SHA-2
  - operations seen in SHA-1

Merkle–Damgård: remarks 1

- The seed is usually constant
- Typically, padding (including text length of original message) is used to ensure a multiple of n (size of domain of H)
- Claim: **if the basic function H is collision resistant, then so is its extension**

Merkle–Damgård: remarks 2

- Input message length should be arbitrary. In practice it is usually up to 2^{64}, which is good enough for all practical purposes
- Block length is usually 512 bits
- Output length should be larger than 160 bits to prevent birthday attacks

Sponge construction

- The Sponge Construction (used by SHA-3/Keccak) processes data like a sponge: **absorbing input** and **squeezing out** the hash
- It uses an internal state (rate + capacity) and a permutation function (f). Data is absorbed into the rate (r) via XOR, followed by a transformation by 'f'. Output is then squeezed from r, with f applied between extractions. The capacity portion remains hidden for security
- Advantages
    - Resistant to Length Extension Attacks
    - Flexible: Creates fixed-length hashes (e.g., SHA3-256) and eXtendable Output Functions (XOFs)

SHA-3 overview

- Based on Keccak algorithm
    - not affected by length extension attack
- 1600-bit internal state
- Permutation-based rounds
- Higher diffusion per round
- Post-quantum suitability
- Available

Keccak functions

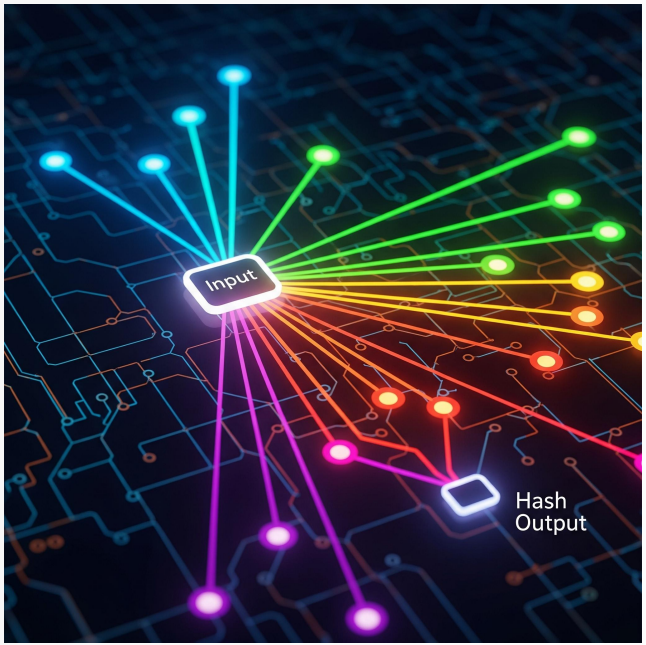
- SHA-3 family
    - SHA3-224, SHA3-256, SHA3-384, SHA3-512
- SHAKE128, SHAKE256
    - they are XOFs (eXtendable Output Functions)
    - XOFs take the desired length as input

      sometimes faster than SHA-3

The most used today

- It is SHA-256
- Member of family SHA-2, with output of 256 bits
    - SHA-224, SHA-384, SHA-512 and truncated versions
- Still secure, fast, easy
- Based on Merkle-Damgård construction
- Standardized: it is approved by FIPS 180-4 and recommended by many international security standards

Cryptographic hashing functions: applications



Recap on cryptographical hashing functions

- Maps input of any length to a fixed-size digest
- Properties
  - Preimage resistance
  - Second preimage resistance
  - Collision resistance

Applications

- In addition to integrity (currently focused and non-finished)
- Digital signatures (later)
- Passwords (later)
- Key derivation functions (KDF)
- Blockchain
- Privacy
- Disk optimization
- PRNG (later)

Key Derivation Functions

- Cryptographic algorithms used to generate one or more secret keys from a shared secret or password
- Their main purpose is to transform input data that may have low entropy or structure into cryptographically strong keys suitable for encryption or other security mechanisms

KDF definition

Algorithm that takes as input

- a base secret (e.g., a password, a key)
- optionally a salt (random value)
- possibly other context-specific parameters

It outputs

- one or more pseudorandom keys of desired length

Often hash-based (not necessarily)

very popular: get a secret key from a password

KDF properties

- Strengthen weak inputs
- Generate multiple keys from a single master secret (key separation)
- Increase entropy using salting and iteration
- Make brute-force attacks more expensive

Blockchain

- Complex and articulated subject deserving more space
- Blockchain is a distributed database or ledger that maintains a continuously growing list of records, called blocks, which are securely linked together using cryptography
- Each block typically contains
    - A list of transactions or records
    - A timestamp
    - A cryptographic hash of the previous block
    - A nonce (in proof-of-work systems)

The blockchain is immutable

- Each block's hash depends on the content of the previous block, forming an unbroken chain: if any block is altered, all subsequent blocks become invalid unless recomputed
- Computation is carried out through Proof of Work (there are other methods)
    - PoW consists in solving a difficult computational puzzle

•How it works (simplified)

1. A transaction is broadcast to the network
2. Nodes group pending transactions into a block
3. The block is validated (by PoW)
4. The validated block is added to the blockchain
5. All nodes update their copy of the chain

PoW

Generic case

- Find a nonce such that: Hash(block data || nonce) < Target
    - nonce = number used once
- This is called a hash-based puzzle. It has no shortcut, brute-force is needed
- Target is known (and slowly updated) with several mechanisms
- People (miners) work trying to solve the puzzle

Hashing functions in privacy

Use case	How hashing supports privacy
Data anonymization	Hashes can obscure identifiers (e.g. names, emails) before data sharing
Password storage	Stores only a hashed version of the password, not the password itself
Zero-Knowledge Proofs	Hashes are used to commit to a value without revealing it
Private blockchains	Transactions may include hashed identifiers to pseudonymize users

Disk optimization

- Hashing is a powerful and efficient method for detecting and eliminating duplicates, especially in large datasets or filesystems

1. Compute the hash of each item (e.g., file, string, record)
2. Compare hashes
    1. If two items have the same hash, they are very likely identical
    2. If the hashes differ, the items are definitely different
3. Keep a record of already-seen hashes to filter out repeats

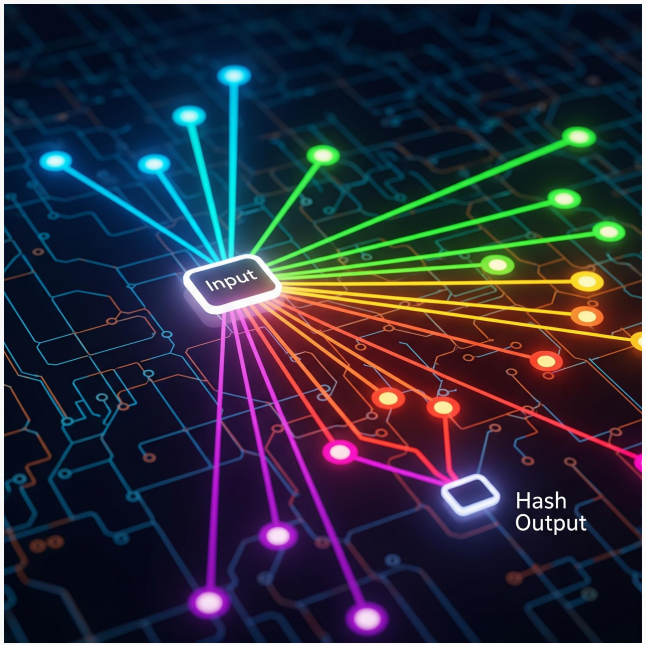
Advantages

- Fast comparison: comparing hashes is faster than comparing entire contents
- Memory-efficient: store only hashes to track duplicates
- Scales well for big data and distributed systems
- Cryptographic quality ensures protection of sensitive information in de-duplicating

A few applications

Domain	Use Case
File backups	Avoid storing same file twice (e.g., rsync)
Cloud storage	Avoid storing identical files from different users
Databases	Filter repeated entries
CDNs / web caching	Identify same content by URL or payload hash

Keyed/unkeyed hashing functions



Cryptographic hashing functions

- Cryptographic hashing functions are designed to operate without a key
- h(x) computes a fingerprint in a deterministic way and everybody can compute it, at least in the scenario that respects the Kerckhoffs's principle
- Traditional hashing functions, not having any key, are said to be **unkeyed**
- Yet, it is true that in the original definition of MAC a secret key k is proposed, and both construction and verification require the knowledge of the k
    - It is a secret key shared between the two communicating parties
- In this sense, the attempt to obtain integrity by using unkeyed hashing functions leads to incomplete results

The need for keyed hashing

- Unkeyed hashing
  - Takes any input, produces a fixed-size hash
  - Properties: deterministic, one-way, collision-resistant, avalanche effect
  - Primary use: data Integrity (verifying if data has changed)
  - Limitation: cannot provide authenticity (who sent it?). If anyone can compute the hash, anyone can forge it
- Goal of keyed hashing
  - To provide both data Integrity and data authenticity
  - Only someone with the shared secret key should be able to produce a valid hash for a given message
- Why not just encrypt?
  - Encryption provides confidentiality (hides content) and can imply authenticity, but it's not the same
  - Hashing is often faster and doesn't require decryption to verify integrity/authenticity

Keyed functions

- A very simple way to obtain a keyed function, if a key is available, is changing the domain of the function so that it considers a key
    - example: if h is an unkeyed function, we can consider h(k‖x), where the ‖ symbol denotes concatenation
    - domain is no longer the set of possible messages but still binary space
    - this function is insecure, as we'll see

Keyed hashing

- A keyed hashing function allows to fully obtain a MAC_k, used for data integrity and origin authentication (authenticity)
- This doesn't happen with unkeyed hashing
- For keyed hashing we continue talking of digest
    - fingerprint is less used
- A well-designed keyed hashing function is the best implementation of MAC

Ways to add a key

- Multiple techniques allow to include a key as an argument of h and some are more secure than others
- It is crucial that only Alice and Bob can use k, whilst the adversary cannot (hence cannot compute a MAC_k)
- Several naïve approaches are possible

Key prefix method

- h(k, M) = h(k‖M)
- Concept: concatenate the secret key before the message, then hash the combined string
- Example: MAC = SHA256("mysecretkey"‖"transfer $100 to Alice")
- Intuition: seems logical. If the key is secret, only those with the key can produce the correct hash
- Insecure, because subject to length extension attacks

Key suffix method

- h(k, M) = h(M‖k)
- Concept: concatenate the secret key after the message, then hash the combined string
- Example: MAC = SHA256("transfer $100 to Alice"‖"mysecretkey")
- Intuition: also seems plausible. The key is still secret

Key sandwich method

- $h(k, M) = h(k \| M \| k)$
- Concept: concatenate the key, then the message, then the key again, then hash
- Intuition: "double protection" might seem stronger

Length extension attack

- It affects functions built using the Merkle-Damgård construction and in particular the **key prefix method**
- An attacker by simply knowing the authentication tag of M can append data to M and forge a valid tag–without knowledge of the key
- Given a pair (M, t) known to the attacker, where t = h(k‖M), he can forge a valid tag for a longer message by guessing |k|, which determines the padding p applied to k‖M
- The attacker forges a new message M‖p‖M', where M' is arbitrary
- The attacker uses t (the internal state after hashing k‖M) to resume hashing and compute a valid tag for M‖p‖M'

Collision attack to key suffix method

If adversary manages to find M' colliding with M, then

- h(M||k) = h(M'||k)
- because of the Merkle-Damgård and the padding further details apply (h(M) = h(M') actually computed by considering padding)

Other techniques

1. h(k, M) = first bits (e.g., first half) of h(k‖M). Partially OK: blocks the length extension attack, lacks provable security guarantees
2. HMAC (later)

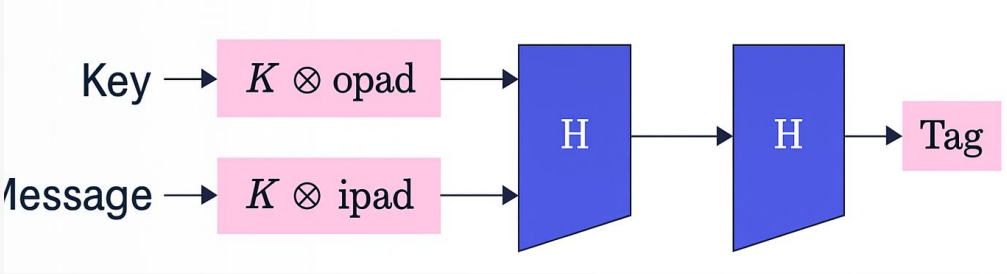
Note that a different view of keyed functions makes us considering them as unkeyed ones with a different binary input

- as key changes, for a fixed message, input changes, but still binary

Keyed hashing and the birthday paradox

- Does the birthday bound still affect keyed hashing functions?
- Yes, nothing changes in terms of domain and range, but...
- ...attacker may be able to find a collision that cannot be however checked for its correctness
    - collision could be found by using an incorrect key
    - in other words, attacker cannot check correctness, so the collision is useless
    - check is possible by knowing the key

HMAC: a complete introduction



Introduction – why HMAC?

- Keyed hashing (MACs): provides data integrity + authenticity
- Naive methods are insecure:
  - H(key || message): vulnerable to length extension attacks
  - H(message || key): vulnerable to collision attacks (if H is weak)
  - Both lack robust security guarantees
- HMAC (hash-based message authentication code):
  - Designed to overcome these weaknesses
  - Provides a robust and secure MAC
  - Industry standard for message authentication

HMAC construction: the formula

HMAC construction: key

HMAC security: addressing weaknesses

HMAC vs. naïve keyed hashing: a deeper dive

- The core difference: HMAC's strength comes from its nested structure, unlike simple concatenation
- Role of ipad and opad
  - These pads (inner and outer) are **fixed, distinct constants**
  - Two different derived keys: **K_inner = K ⊕ ipad** and **K_outer = K ⊕ opad**
  - Inner and outer hash are distinct and don't expose key or internal state
- Preventing state reconstruction
  - Inner hash **H(K_inner || M)** produces an intermediate hash value
  - Then fed into the outer hash **H(K_outer || inner_hash)**
  - Attacker knowing only the final HMAC cannot easily reconstruct K_inner or K_outer, nor the internal state of the inner hash, because the outer hash "seals" it, making length extension attacks ineffective

HMAC advantages and practical uses

- Key management and flexibility: handles various key lengths; can use any cryptographically secure hash function
- Proven security: extensively analyzed, widely adopted standard (RFC 2104, FIPS 198)
- Practical use cases
  - TLS/SSL: message authentication for secure web communications
  - IPsec: integrity and authentication for IP packets
  - JSON web tokens (JWTs): signing tokens for integrity and authenticity
  - API authentication: protecting web API requests
  - Software updates: verifying the integrity and authenticity of downloads
  - Challenge-response authentication (later)

Recommendations 1

Underlying hash function strength

- While HMAC improves resistance, using a strong underlying hash function is still crucial for overall security
- Recommendations: HMAC-SHA256, HMAC-SHA512, or HMAC-SHA3 are currently considered strong choices

Key management for HMAC

- Secure generation: use cryptographically secure random number generators for keys
- Secure storage: store keys securely, separate from the data they protect
- Key rotation: regularly change HMAC keys

Recommendations 2

HMAC is not encryption

- HMAC provides integrity and authenticity, not confidentiality
- It does not hide the message content. It is one-way

Key reuse

- Avoid reusing the same key for different cryptographic purposes (e.g., using an HMAC key as an encryption key)
- Each cryptographic primitive should ideally have its own unique key

Side-channel attacks

- Even with a secure HMAC implementation, timing or power analysis attacks can sometimes leak key information
- Proper implementation requires constant-time operations to mitigate this risk

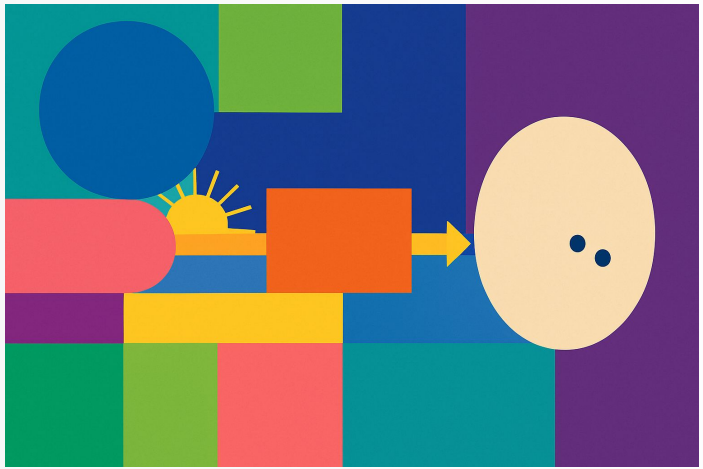
Weak underlying hash function (again)

- While HMAC improves collision resistance, using a completely broken hash (like MD5) is still not recommended for new systems
- The security of HMAC ultimately relies on the security properties of the underlying hash function

Summary

- Purpose: a robust standard for message authentication, providing both data integrity and authenticity
- A nested hash construction that leverages the underlying hash function
- Security
  - Effectively prevents length extension attacks
  - Significantly improves collision resistance compared to naive keying methods
  - Theorem: HMAC can be forged if and only if the underlying hash function is broken (collisions found). [Bellare, Canetti, Krawczyk, 1996: https://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf]
- Best practices
  - Choose a strong underlying hash function (e.g., SHA-256, SHA-3)
  - Implement secure key management (generation, storage, rotation)
  - Always rely on standardized and well-vetted libraries for implementation

HMAC and birthdays attack



Adversary model

HMAC and birthday paradox

- HMAC behaves like a pseudorandom function from the attacker’s perspective
- Even with many observed (message, HMAC) pairs
    - The attacker cannot test or verify a collision
    - He cannot generate new tags
    - Even if two messages yield the same tag, he can't confirm it's a true collision

Oracle attacks and HMAC

- In **oracle attacks**, the adversary queries a system that produces valid tags using the (unknown) key, to learn something
- HMAC is **secure against chosen-message attacks**, even if the attacker can
  - submit many messages to an oracle and observe corresponding outputs
  - attempt adaptive queries based on previous responses
- Because of its pseudorandom behaviour, HMAC reveals **no useful information** about k

HMAC and the birthday bound

- The birthday bound is a universal hash property
- It applies to HMAC output in theory, but is **not exploitable** in practice
- Shows how HMAC resists attacks even at theoretical collision levels
- In HMAC, observed collisions are meaningless without the key

Theory and practice

- Collisions **might exist** due to the birthday bound
- But the attacker cannot
  - Recognize or confirm them
  - Exploit them without the key
- HMAC output appears **random and independent** to the attacker

Example

Sandwich-style keying

- Even a naïve keyed construction like H(k||m||k) may appear safe
- The attacker still cannot practically exploit the birthday paradox
    - The key is embedded in the input
    - Without knowing the key, collisions are untestable
    - Still oracle needed
- But under some (strong) hypotheses, two colliding messages under H can still collide in the sandwich structure

Summary

- The birthday paradox has practical consequences on unkeyed hash functions, not to keyed
- HMAC hides the internal structure of the hash from the attacker
- Security of HMAC relies on the secrecy of k, not just hash function strength