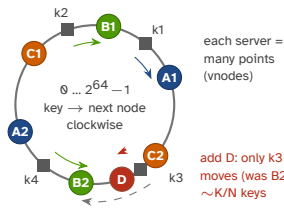
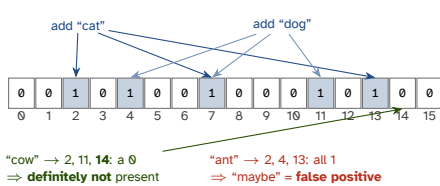


In one line: A hash maps arbitrary input to a fixed-size value. **Table hashes** want speed + uniformity and a **secret seed** against flooding (Swift: SipHash, reseeded per process); **cryptographic hashes** want one-wayness and collision resistance (SHA-256); **password hashes** want to be **slow and memory-hard**; a **MAC** needs a key. Choosing the wrong family is the classic mistake.

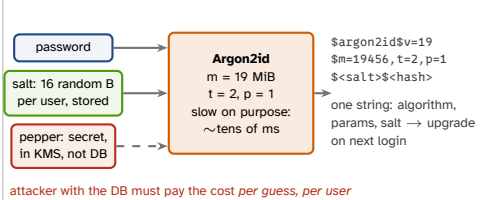
Consistent hashing + virtual nodes



Bloom filter: m bits, k hashes



Storing a password



What makes a hash good

- Deterministic** (within its seed), **uniform** (keys spread evenly over buckets), **avalanche** (flip 1 input bit $\rightarrow$ $\sim$ half the output bits flip, so near-identical keys don't cluster), **fast** ($O(\text{key length})$).
- Table hashes:** FNV, Murmur, xxHash (fast, unkeyed $\rightarrow$ floodable); **SipHash** (Aumasson & Bernstein 2012) is a fast keyed PRF built to stop flooding. Swift's Hasher uses SipHash-1-3, keyed by a **random per-process seed** (SWIFT_DETERMINISTIC_HASHING=1 to pin it in tests).
- HashDoS** (28C3, 2011): an attacker who knows an unkeyed hash posts thousands of keys that collide $\rightarrow$ every insert walks one chain $\rightarrow O(n^2)$ CPU from one request. The secret seed makes the collision set unpredictable. Cost: hashValue and Set order change every launch.

Collisions — normal, then resolved

- Birthday bound:** with m possible values, a collision is likely after $\sim \sqrt{m}$ items: $P \approx 1 - e^{-n^2/2m}$, 23 people $\rightarrow$ 50%; a 64-bit hash $\rightarrow$ 2^{32} (4 billion) items; UUIDv4 (122 random bits) $\rightarrow$ 2.7×10^{18} , SHA-256 collision work = 2^{128} , not 2^{256} .
- Chaining** (list per bucket; Java HashMap turns a chain of ≥ 8 into a red-black tree).
- Open addressing:** linear probing (cache-friendly, primary clustering), quadratic, double hashing, **Robin Hood** (steal the slot from a key nearer its home — evens probe lengths), **cuckoo** (2 tables, $O(1)$ worst-case lookup). Deletion needs **tombstones** or backward shift.
- Rehash:** at load $\alpha \approx 0.75$ (Swift: 3/4) double and re-insert all — $O(n)$ once, $O(1)$ amortised, a visible *latency spike*. Redis rehashes **incrementally** (two tables, move a few buckets per op).
- Consistent hashing:** $\text{hash}(\text{key}) \% N$ remaps $\sim$ all keys when N changes; the ring moves $\sim K/N$. 100–200 vnodes per server smooth load and spread a dead node's keys over all survivors. Alternatives: **rendezvous** (highest random weight), **jump hash** (no memory, numbered buckets only).

Example — Hashable right, and a stable hash

```
struct User: Hashable {
    let id: Int; var displayName: String // not identity
    static func == (a: User, b: User) -> Bool { a.id == b.id }
    func hash(into h: inout Hasher) { h.combine(id) } // SAME fields as ==
}
// hashValue changes per launch: never persist it. A stable key:
import CryptoKit
let name = SHA256.hash(data: Data(url.absoluteString.utf8))
.map { String(format: "%02x", $0) }.joined() // cache file name
```

Cryptographic hashes

property	attacker cannot...	SHA-256 cost
preimage	find x from $H(x)$	2^{256}
second preimage	given x , find $x' \neq x$, same H	2^{256}
collision	find any pair $x \neq y$	2^{128} (birthday)

MD5: collisions in seconds; **SHA-1:** public collision 2017 (SHAttered). Both dead for signatures/integrity; fine only as checksums. SHA-256 / SHA-512 are Merkle-Damgård $\rightarrow$ **length extension:** from $H(k||m)$ anyone computes $H(k||m||pad||m')$. SHA-3 and BLAKE2/3 don't have it.

MAC: HMAC vs a plain hash

A plain hash proves integrity only against *accidents* — anyone can recompute it. $\text{HMAC}(K, m) = H((K \oplus \text{opad}) || H((K \oplus \text{ipad}) || m))$: only a key holder can make or check the tag, and the nesting defeats length extension. Compare tags in **constant time** (CryptoKit isValidAuthenticationCode), never with ==.

Password hashing — slow on purpose

function	OWASP minimum	note
Argon2id	m=19 MiB, t=2, p=1	PHC winner; memory-hard; first choice
scrypt	$N=2^{17}$, r=8, p=1	memory-hard
bcrypt	cost ≥ 10	CPU only; input cut at 72 bytes
PBKDF2	$600000 \times \text{HMAC-SHA256}$	FIPS; GPU-friendly, weakest

SHA-256 does $\sim 10^{10}$ guesses/s on one GPU; Argon2id a handful per core — that ratio is the defence. **Salt** (unique, public) kills rainbow tables and same-password-same-hash; **pepper** (secret, outside the DB) makes a DB-only leak uncrackable. Hash on the server.

Bloom filter

m bits, k hash functions. No false negatives, tunable false positives: $p \approx (1 - e^{-kn/m})^k$; best $k = (m/n) \ln 2$; **1% $\approx$ 9.6 bits per item, $k = 7$** . No delete (counting Bloom filter can). Use: skip a disk/network lookup for keys that surely aren't there (LSM-tree databases, CDN caches).

Interview traps

- ! Persisting hashValue / asserting Set order — changes per launch.
- ! `hash(into:)` using a field == ignores, or mutating a key after insertion $\rightarrow$ lost or duplicate entries.
- ! $\text{SHA-256}(\text{password})$, even salted — too fast. And $H(\text{key}||\text{msg})$ is not a MAC.
- ! "A hash is encryption" — no key, no inverse. "Collision = 2^{256} " — it's 2^{128} .

Remember

Tables: fast + seeded. Integrity: SHA-256. Authenticity: HMAC. Passwords: Argon2id + salt. Membership at scale: Bloom.

Likely questions

- Why is Swift's hash seeded? — HashDoS; so never persist it.
- Why vnodes? — even load; a dead node's keys spread over all.
- Bloom says yes? — maybe; says no — definitely no.
- SHA-256 for a Dictionary? — too slow; seeded SipHash already stops flooding.
- Why HMAC, not $H(k||m)$? — length extension forges the latter.
- Store passwords? — Argon2id, per-user salt, PHC string; re-hash on login when params rise.