

# Digests, Integrity Control and Key Derivation

SIO

**deti** universidade de aveiro  
departamento de eletrónica,  
telecomunicações e informática

João Paulo Barraca

Lecture notes were created by Gen AI and edited by faculty. Please report any error you may find.

# Digest Functions

## Overview

- Produce a digital **summary** of data called a **message digest**
  - Data is a text or any binary information
- The message digest **length is fixed**
  - independently of the text length
    - Both a 200 bytes and a 200 TB data items will result in a digest with the same length
- The message digest value **strongly depends** on the data
- Two digests are typically **very different**
  - Even if the original data is extremely similar

A message digest function compresses arbitrary data into a short fixed-length value, the digest, which acts as a fingerprint of the input. Any binary object, from a two-line text file to a multi-terabyte video, can be reduced to this summary, and that reduction is what makes digests a universal way to refer to data: instead of copying or comparing the data itself, systems compare its fingerprint.

The digest length depends only on the chosen algorithm, never on the input size: 200 bytes and 200 terabytes both produce a value of the same length. The mapping is deterministic, so the same input always yields the same digest, while two very similar inputs produce digests that look completely unrelated and share no visible structure. This is why a digest is a fingerprint, not a summary a human could read back.

Two caveats matter in practice. The mapping is one-way, so a digest cannot be turned back into the original data: there is no inverse operation, which is why we say a digest cannot be decrypted. And shorter digests are easier to collide than longer ones, which is why modern designs use 256-bit digests and abandon the older 128-bit and 160-bit ones, whose collision attacks are already practical.

Version control systems such as git identify every object in a repository by a fingerprint of its content, so renaming a file does not change its identity, and software download sites publish digests next to their files so users can confirm they obtained the exact, unaltered artifact. Both are everyday examples of data addressed by fingerprint.

# Digest Functions

## Properties

- Preimage resistance
  - Given a digest, it is unfeasible to find an original text producing it
  - That is: we **cannot go back** from a digest to the data (we cannot “decrypt” it)
- 2nd-preimage resistance
  - Given a text, it is unfeasible to find another one with the same digest
  - That is: if we **have a text**, we cannot find another one with the same digest
- Collision resistance
  - It is unfeasible to find any two texts with the same digest
  - That is: given two unique texts, they will result **in a different digest**
    - Relates to the Birthday paradox: Collision probability  $P = 2^{-(n/2)}$  where the typical  $n$  is  $\geq 256$

Digest security rests on three mathematical properties: preimage resistance, second-preimage resistance, and collision resistance. Together they make the function behave like a one-way door: easy to cross forward, infeasible to cross back, and the three properties correspond to three different directions of attack that an adversary might try.

Preimage resistance means that given only a digest, finding any input that produces it is computationally infeasible. Second-preimage resistance means that given one input, finding a different input with the same digest is infeasible. Collision resistance is the strongest requirement: finding any two inputs with the same digest at all is infeasible. In practice the first two cost about  $2^n$  operations each, while collisions are the cheaper target.

Collision resistance is the binding constraint because of the birthday paradox: for an  $n$ -bit digest, a collision can be found in roughly  $2^{n/2}$  operations, so a 128-bit digest only offers about 64 bits of collision security. This bound, not raw hash speed, is why digests are at least 256 bits today, and it is why 256-bit designs are said to give about 128 bits of security.

These properties are exactly what file integrity checks rely on. An attacker who cannot forge a matching digest cannot silently alter a published file, which is why an integrity scheme is only as strong as the collision resistance of the hash it uses. When a hash is later shown to be weak, every system built on it must be re-evaluated against these three properties.

# Digest Functions

## Lets check: Size independence

- Considering the similar, yet different texts:
  - T1: “Hello User\_A!”
  - T2: “Hello User\_XPTO! Welcome to this lecture”
- Different algorithms will create digests with different lengths, but **independent** from the dimension of the text
  - MD5 (128 bits):
    - T1: 70df836fdaf02e0dfc990f9139762541
    - T2: 18f12f09c45d880ce738afe4780c2f3e
  - SHA-1 (160 bits):
    - T1: f591aa1eabcc97fb39c5f422b370ddf8cb880fde
    - T2: 622f7832e204f2d70161cf42480c4bf0f13e7324
  - SHA-256 (256 bits):
    - T1: 9649d8c0d25515a239ec8ec94b293c8868e931ad318df4ccd0dfd67aff89905
    - T2: 6453be3f643d0a7e9b5890eed76bb63df8b6b071b30d5f97269a530c289b9839

The first hands-on demonstration checks size independence: two different strings are hashed with three algorithms, and the output width stays constant no matter how long the input is. The strings are deliberately similar, a short greeting and a longer welcome message, so the effect is visible in the comparison rather than just claimed.

MD5 always outputs 128 bits, shown as 32 hex characters; SHA-1 outputs 160 bits as 40 hex characters; SHA-256 outputs 256 bits as 64 hex characters. Whether the input is a short greeting or a long welcome message, each algorithm returns the same width, and the hex values printed are the genuine digests of those exact strings, reproducible on any machine that implements the algorithm.

The values are illustrative, not a security endorsement: MD5 and SHA-1 are deprecated because practical collision attacks exist, so their outputs should not be trusted for integrity in any new design. SHA-256 is the current workhorse, and its 256-bit output is the width to aim for. The demonstration is about the width property; the choice of algorithm is a separate decision covered later. The three widths also show the design trade-off: wider digests cost a little more to compute and store, which is why the deprecated 128-bit and 160-bit functions were popular in an era when storage was dearer.

Fixed output width is what lets a system store a small checksum next to a huge file and compare digests instead of data, the basis of every download verification page and content-addressed storage system. A 64-character line can stand in for a multi-terabyte object, which is why digests are so central to infrastructure.

# Digest Functions

## Lets check: Content dependency

- Considering the similar, yet different texts (1 bit difference 'B' -> 'C'):
  - T1: "Hello User\_B!", [0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x55, 0x73, 0x65, 0x72, 0x5f, 0x42, 0x21]
  - T2: "Hello User\_C!", [0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x55, 0x73, 0x65, 0x72, 0x5f, 0x43, 0x21]
- A small difference in the text (1 bit) results in a **completely different digest**
  - MD5:
    - T1: c32e0f62a7c9c815063d373acac80c37
    - T2: 324a1bfc3041259480c6ad164cf0529f
  - SHA-1:
    - T1: bab31eb62f961266758524071a7ad8221bc8700b
    - T2: bd758d82899d132cd2af66dc3402b948d98de62d
  - SHA-256:
    - T1: e663a01d3bec4f35a470aba4baccece79bf484b5d0bfffa88b59a9bb08707758a
    - T2: 69f78345da90c6b8d4785b769cd6ae09e0531716fe5f5a392fde1bdc70a2bb7d

The second demonstration checks content dependence: a one-character change in the input should scramble the entire digest, a property called the avalanche effect. It is the reason a digest is a fingerprint rather than a summary a human could approximate, and it is what makes 'almost the same' meaningless for digests.

Changing a single character in the greeting, one bit difference in the byte sequence, flips roughly half the output bits in every algorithm. The MD5, SHA-1, and SHA-256 digests of the two strings share no visible pattern: no prefix survives, no alignment remains, and the outputs look like independent random strings, with no hint that the inputs differed by a single bit. The effect is a direct consequence of the internal structure: each round of compression mixes the input through a permutation, so a one-bit difference is amplified and spread across the entire output.

Averaging or comparing digests of nearly identical inputs is therefore meaningless; digest equality is all or nothing, and there is no such thing as almost the same digest. That also means diff tools cannot use digests to measure how similar two files are, and any tool that tries to use digest similarity as a signal of content similarity is fundamentally wrong.

Blockchain nodes exploit this property to detect tampering: altering any byte of a block changes its digest and breaks the chain of hashes linking the blocks, so a single corrupted byte is immediately visible. The same property is what makes content-addressed storage trustworthy: the address is a promise about every byte.

# Applications of digest functions

- File and data integrity verification
  - Check downloaded files against published digests (kernel.org, ISOs)
  - Detect accidental corruption from storage or transmission errors
  - Digital Signatures (To be addressed later)
- Password storage
  - Store salted digests instead of plaintext passwords
  - A slow hash (or KDF) resists offline guessing
- Data deduplication and change detection
  - Compare digests to find duplicates or changed data
  - Used in versioning, backups and intrusion detection

SIO

Digests are a workhorse of systems security, far beyond academic examples. Their four main roles are integrity checking, password storage, building MACs and digital signatures, and deduplication or change detection. Each role uses a different strength of the same basic function, which is why the same digest appears in so many unrelated systems.

For integrity, a published digest lets anyone verify a download bit-for-bit against random corruption from disk or transfer errors. For passwords, systems store a salted digest or derived key instead of plaintext and deliberately use a slow function so offline guessing is expensive. For signatures and MACs, the short digest is what actually gets signed or authenticated, keeping the expensive asymmetric or keyed operation cheap. For deduplication, equal digests mean equal content, which powers versioning, backup deduplication, and change detection.

The pitfall is matching the speed of the hash to the wrong job: a fast digest makes password guessing trivial on modern hardware, and a deprecated algorithm such as SHA-1 is unsafe for any new integrity purpose. Choose the tool per role, not once for everything, and revisit the choice when the underlying algorithm ages.

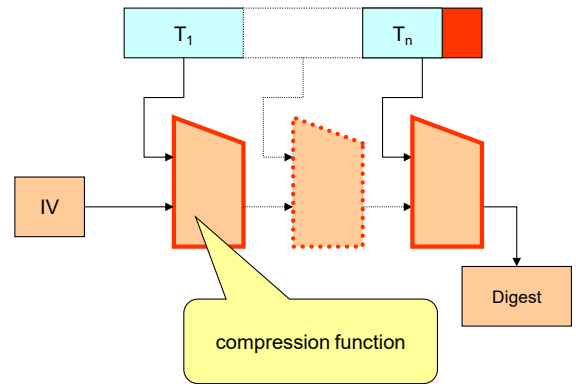
Package managers, intrusion detection systems, and backup software all compare digests daily to catch corruption, detect duplicates, and flag files that changed on disk without an update. If a system you use could not work without digests, it is almost certainly doing at least one of these four jobs.

# Digest Functions

## Approaches

- Merkle-Damgård construction

- Collision-resistant, one-way compression functions
  - Can be a block cipher!
- Iterative compression
- Length padding
- Digest size is the last block
- Can be resumed!
  - Digest is the state at  $T_n$
- Algorithms: MD5, SHA1, SHA2



Merkle-Damgård is the iterative construction behind MD5, SHA-1, and SHA-2. It processes the input in fixed-size blocks, chaining the output of each compression step as the running state for the next block, so every block is processed in the context of everything that came before it.

The message is padded so its length is a multiple of the block size, then an initial value, the IV, seeds the first block. Each block is fed to a collision-resistant compression function, which can even be built from a block cipher, and the digest is simply the final state after the last block is processed. The compression function is the heart of the design: its collision resistance is what the whole hash inherits.

Because the digest is just the running state, the computation can be resumed from a checkpoint, which is convenient when hashing huge files in a stream. The same chaining is also what enables length-extension attacks on naive keyed variants, a problem covered later in this topic: the strength and the weakness of the design come from the same structural decision.

Streaming hashers in operating systems and download managers exploit exactly this resumable property to compute digests as files arrive, without loading the entire file into memory at once. The construction is simple enough to implement by hand, which is also why its subtle failure modes are worth understanding. It is also the structure that makes the construction easy to analyze: the security of the whole hash can be reduced to the security of a single compression step.

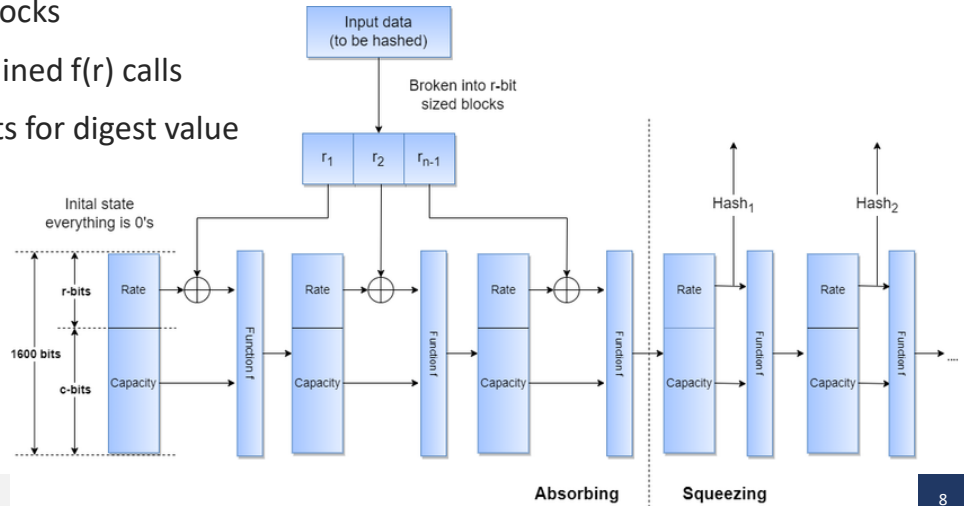
# Digest Functions

## Approaches

- Sponge functions

- Data split in  $r$  sized blocks
- Absorbing phase: chained  $f(r)$  calls
- Squeezing: extract bits for digest value
- Algorithms: SHA3

- Key point: Output is not the internal state, only part of it.



João Paulo Barraca, André Zúquete

8

Sponge functions form the other main family of digest constructions and underlie SHA-3, also known as Keccak. Instead of chaining blocks one at a time, they absorb the input into a large internal state and then squeeze output out of it, which is why the picture on the slide shows a sponge being soaked and then wrung out.

The input is split into blocks of  $r$  bits and absorbed by repeated calls to a permutation function  $f$  over the state; once the input is exhausted, output bits are squeezed out until the digest is complete. The state width and the rate  $r$  together control both the security level and the throughput, and the key point on the slide is that the output is only part of the internal state, not the state itself. The permutation is the security core: in Keccak it is a fixed sequence of bit-level operations applied to the full state, chosen so that no shortcut around the full number of rounds is known.

The practical difference from Merkle-Damgård designs is structural diversity: sponge constructions do not expose the running state the same way, so they are not vulnerable to length-extension attacks in the same manner. That independence is one reason SHA-3 was standardized as a genuinely different alternative, not just a faster variant of the same idea.

SHA-3 is already deployed in TLS stacks and blockchain systems as a hedge against weaknesses that might later be found in the older iterative designs, giving protocols a second, structurally different option. Having two families in production is exactly the redundancy that keeps the ecosystem safe when one family falls.

# Length extension attack

- The problem with naive keyed digests
  - Appending/prepending a key to the data is weak
  - Attacker can extend the message without knowing the key
- How the attack works (Merkle-Damgård)
  - Reuse the internal state as a new IV
  - Append padding and extra blocks, recompute the digest
- Practical impact
  - Breaks naive keyed hashes and some MAC designs
  - Never build a MAC by simply prepending or appending a key

SIO

Length extension is the classic attack on naive keyed digests built on Merkle-Damgård hashes, and it explains precisely why HMAC exists (Discussed later). It is one of those attacks that is easy to describe, hard to anticipate, and completely avoidable if you use the standard construction instead of inventing your own.

Such a hash chains blocks, so its final output is the internal state after processing the key and the message. An attacker who sees that digest can reuse it as a starting state, append extra blocks with correctly computed padding, and produce a valid digest for the extended message without ever knowing the key. The extended message is usually crafted so that the added part carries attacker-controlled content.

This attack broke several real-world schemes, from naive file integrity tools to early API signing formats, and it is the reason every modern MAC construction specifies HMAC rather than a homegrown keyed hash. It is also a good example of why 'it works on my inputs' is not a security argument.

# Choosing a digest algorithm

- **Deprecated:** MD5 and SHA-1
  - MD5: practical collisions since 2004-2008; do not use
  - SHA-1: collision attacks practical (SHAttered, 2017)
- **Current standard:** SHA-256 / SHA-512
  - Widely supported, ~256-bit security
  - Basis of TLS, certificates, git, and many protocols
- **Modern alternatives:** SHA-3, BLAKE2
  - SHA-3: sponge-based, different internal structure
  - BLAKE2: fast, with keyed and tree variants
- **Match the digest size to the security need**
  - Preimage  $\sim 2^n$ , collision  $\sim 2^{(n/2)}$
  - For ~128-bit security use at least 256-bit digests

SIO

Choosing a digest today is mostly a matter of knowing which algorithms are safe. The rule is simple: MD5 and SHA-1 are dead, SHA-256 is the default, and SHA-3 or BLAKE2 are strong modern alternatives for new designs. The decision is about the algorithm's age and structure, not about its speed.

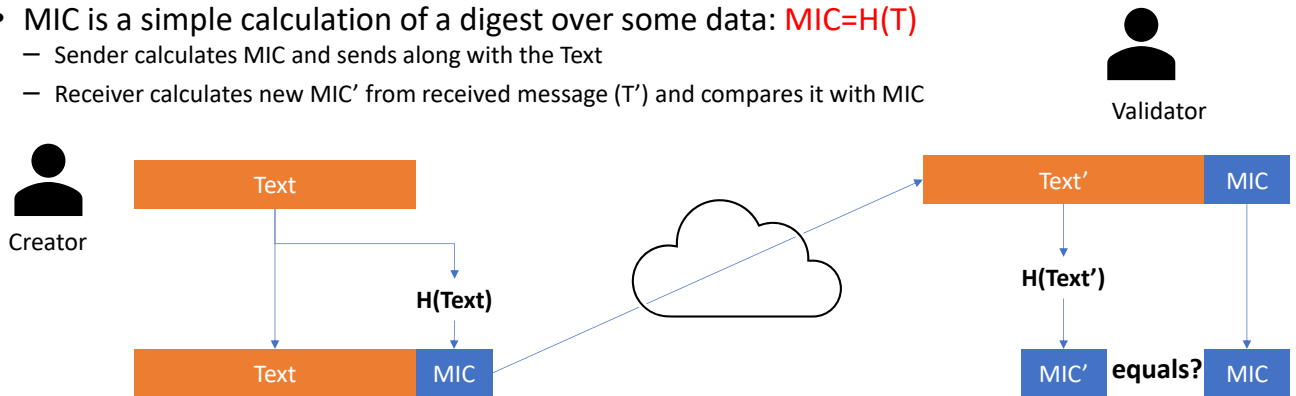
MD5 has had practical collisions since the mid-2000s and SHA-1 since the SHAttered attack of 2017, so both are deprecated for anything security-relevant, even if they are still present in legacy formats. SHA-256 offers about 128 bits of collision security and underpins TLS, code-signing certificates, and git. SHA-3 brings a sponge structure, while BLAKE2 is fast and ships in keyed and tree-based variants.

The common sizing mistake is to match digest length to the wrong target: preimage resistance costs about 2 to the  $n$  operations, but collisions only 2 to the  $n/2$ , so 128 bits of security needs a 256-bit digest, not a 128-bit one. Picking the width by security target, not by habit, avoids this trap, and it is the same birthday-bound arithmetic from the properties material earlier.

Most protocols quietly standardize on SHA-256, so in day-to-day development the real decision is less about picking an algorithm and more about refusing the legacy ones whenever they still appear in a configuration. When you see MD5 or SHA-1 in a new design, treat it as a red flag, not a tradition. Legacy support is why MD5 and SHA-1 still appear in file formats and old protocols: existing data keeps the old hashes alive, but new data should never join them.

# Message Integrity Code (MIC)

- Provide the capability to detect **arbitrary** changes to data
  - Communication/storage errors from a random process or without integrity control
  - Humans/Attackers can change the Text and calculate a new MIC!
- MIC is a simple calculation of a digest over some data:  $MIC = H(T)$ 
  - Sender calculates MIC and sends along with the Text
  - Receiver calculates new  $MIC'$  from received message ( $T'$ ) and compares it with MIC



A Message Integrity Code is the simplest integrity tool in cryptography: it is just the unkeyed digest of the data,  $MIC = H(T)$ . It detects accidental changes but offers no protection against an adversary who wants to alter the data, and that distinction between error and attack is the whole point of the slide.

The sender computes the digest of the text and transmits both together; the receiver recomputes the digest over the received text and compares the two values. Any mismatch means the data was altered or corrupted in transit or at rest, so the message is rejected or re-requested. The comparison is exact: one changed bit changes the whole digest, so there are no near misses. The mechanism is the same whether the data travels over a network or sits on disk: compute, store, recompute, compare, and the digest is the only thing that needs to match exactly.

Because no secret is involved, anyone who modifies the text can simply recompute the MIC and attach the new value, and the receiver has no way to tell. A MIC therefore only guards against random errors, not deliberate tampering, a frequent misconception when it is used to secure a file: a published digest protects against corruption, not against an attacker who can also publish.

Download pages that publish digests of release files rely on exactly this mechanism to catch corruption during transfer; the trust is placed in the published value, which is why it must come from an authoritative channel that the attacker cannot modify in the same pass.

# Example usage at kernel.org to validate file integrity

| patch-6.7.9.xz  | 06-Mar-2024 15:09 | 703K |
|-----------------|-------------------|------|
| patch-6.7.xz    | 08-Jan-2024 06:00 | 8M   |
| patch-6.8.1.xz  | 15-Mar-2024 19:04 | 5992 |
| patch-6.8.10.xz | 17-May-2024 10:24 | 738K |
| patch-6.8.11.xz | 25-May-2024 14:46 | 748K |
| patch-6.8.12.xz | 30-May-2024 07:59 | 878K |
| patch-6.8.2.xz  | 27-Mar-2024 05:24 | 241K |
| patch-6.8.3.xz  | 03-Apr-2024 13:44 | 374K |
| patch-6.8.4.xz  | 04-Apr-2024 18:39 | 366K |
| patch-6.8.5.xz  | 10-Apr-2024 14:49 | 461K |
| patch-6.8.6.xz  | 13-Apr-2024 11:27 | 498K |
| patch-6.8.7.xz  | 17-Apr-2024 09:38 | 537K |
| patch-6.8.8.xz  | 27-Apr-2024 15:28 | 583K |
| patch-6.8.9.xz  | 02-May-2024 14:54 | 643K |
| patch-6.8.xz    | 10-Mar-2024 21:45 | 7M   |
| patch-6.9.1.xz  | 17-May-2024 10:28 | 3336 |
| patch-6.9.10.xz | 18-Jul-2024 11:37 | 693K |
| patch-6.9.11.xz | 25-Jul-2024 08:15 | 647K |
| patch-6.9.12.xz | 27-Jul-2024 09:48 | 652K |
| patch-6.9.2.xz  | 25-May-2024 14:54 | 16K  |
| patch-6.9.3.xz  | 30-May-2024 07:55 | 151K |
| patch-6.9.4.xz  | 12-Jun-2024 09:49 | 263K |
| patch-6.9.5.xz  | 16-Jun-2024 12:04 | 306K |
| patch-6.9.6.xz  | 21-Jun-2024 12:54 | 388K |
| patch-6.9.7.xz  | 27-Jun-2024 12:04 | 465K |
| patch-6.9.8.xz  | 05-Jul-2024 07:53 | 521K |
| patch-6.9.9.xz  | 11-Jul-2024 11:08 | 572K |
| patch-6.9.xz    | 13-May-2024 05:20 | 7M   |
| sha256sums.asc  | 10-Oct-2024 11:05 | 102K |

|                                                                   |                  |
|-------------------------------------------------------------------|------------------|
| 7d9d4db55cea8270d51e9c3b7c3e045896da8bc218bb788d36231da050e8dfcfa | patch-6.9.2.xz   |
| f82cdfa0f94327f9f9fbbdbde5efc560ef4cf5f2269718c9be58440aebff3d3   | patch-6.9.3.xz   |
| ee37e61dfb1cc4e638043bb411a38f742e945a9f7e54519f835d62c10052795   | patch-6.9.4.xz   |
| 93293c1618f13d003a3815f77a188e091857cc1b61ac20e24a8c07a0e5de104   | patch-6.9.5.xz   |
| 119c49942462d9d9d67dccc2d746cf1e5d773e06b9f5fa23c18dc0680bda9209  | patch-6.9.6.xz   |
| f51cf3888051bcd94cc209973e0e32dda87696d47ba5f31856c8d40571584f1   | patch-6.9.7.xz   |
| cf7dbb88fa35557195f8cde7fd05ec873db95af91bcbae7472a1d39bc9c5cbaa  | patch-6.9.8.xz   |
| 0a988fea75294155d7e44505e95bdc56304d5650b6910fc5e50847ae16e77e7   | patch-6.9.9.xz   |
| a99364d3ca23bf7bed62532c160d252ab157ea7b549cb315e9861e6192380358  | patch-6.9.10.xz  |
| 8f649680158104c8a255030e3e476e2f908c70fc470a463a0be62c561bf32     | patch-6.9.11.xz  |
| 27d5981423079215cd4e0403f80508927c3239f8008bac9c25d3c1407c612     | patch-6.9.12.xz  |
| de113d55da1846e717ed420ae5e6e6277d6f54abe03b70fb11c6cf197f613a    | patch-6.10.xz    |
| 101689f5d5d98d1adaf90ec3f334524848513d53dc40341f1f5c4809f0390549  | patch-6.10.1.xz  |
| f3166b9b9f6a7dbae9ed7e92e373c8dbd672c5b5d2da3991207aa30f52ceda7fa | patch-6.10.2.xz  |
| ccde554363cfbd3d2533d1cc2506f397dd1dd278809b1041deb929c5534e8b4f  | patch-6.10.3.xz  |
| 4c6e823a3ed73089b958cb2d2974982f769435d5aa7750cbdf4932c92188eca0  | patch-6.10.4.xz  |
| 26478afe830672dc6daedafcf6c9ab9b1683c039a3ace95aac88d32d4ba364f5  | patch-6.10.5.xz  |
| 8f480892bc3f38080e0a7665e9c189699af35979d2ff1da85dedebd2f1ca97f2  | patch-6.10.6.xz  |
| 221b40140dabf32b1dc1cece374d1731e4566e52da5313909d7080baa8ead9c0  | patch-6.10.7.xz  |
| d554f93e19038ff702a9679aac1aa491be737e98c22f8d5307c5c09ef6d11903  | patch-6.10.8.xz  |
| 90bc50b9106c4e0b796cf70473dd830073789bb9896664ff8617a1203d17527   | patch-6.10.9.xz  |
| bb50ad317a90bf40846284fa4d3cc8a4065bdde9357be4ab50c74a12b2f2ff5   | patch-6.10.10.xz |
| b0d817a60609b41bdee44e3e3e8dd077ad64cd6e22818e21fe1b8b97a6adb8    | patch-6.10.11.xz |
| 85994d53de093fff1217962232b49629b6c5607eae03fa4e234fed740a16f665  | patch-6.10.12.xz |
| c46ec7c0663f79e957fc82226d03ea5416367b5582916a8bb4e83d3510796c7   | patch-6.10.13.xz |
| 28d575921f079c7ff449e9066904c27d041851342a9f08164ba8853f0a37ae    | patch-6.10.14.xz |
| 65c5274a457a8757ac543dd9561f4264e4db6c37b9de9716e6aa0928f18e      | patch-6.11.xz    |
| e209cd7f59dd57a6c5c36ce5bc7c494401a695e323e35e2a62c70807095c0     | patch-6.11.1.xz  |
| 2b269f51bafb89937206a0efcac6f93c94cdf2f24d7e54eb304a93cf9e4929    | patch-6.11.2.xz  |
| 4c808f6dd8814ab55a343649a2e2b2925895bf7f9044d15fa3424eac6f9349c3e | patch-6.11.3.xz  |

A practical example of MIC-based integrity checking is the workflow on kernel release pages, where digests are published next to every downloadable file, one line per artifact, in a format that is easy to copy or parse.

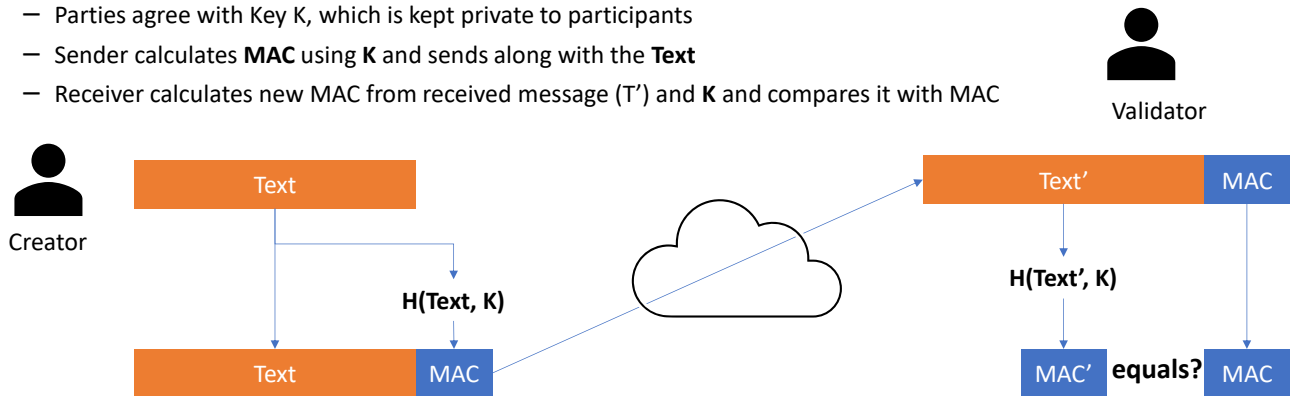
A user downloads a tarball, runs a local digest command over the file, and compares the 64-hex-character output with the published value. An exact match is strong evidence that the file arrived intact; any difference, even one character, points to corruption in transfer or on the storage medium, and the file should be re-downloaded rather than used.

The subtlety is where the reference digest comes from: a MIC only works if the published value was obtained through a channel the attacker cannot alter, such as a signed announcement page, rather than a plain file listing that could be tampered with in the same pass. The digest proves the file matches the claim; it cannot prove the claim itself.

This exact workflow of download, hash, compare is the daily routine for anyone installing operating system images, kernel releases, or large scientific datasets, and it takes only seconds to run, which is why it is considered basic hygiene rather than advanced security.

# Message Authentication Code (MAC)

- Provide the capability to detect **deliberate** changes to data
  - Any change to data, even if from attackers!
- MAC is a keyed calculation of a digest over some data:  $MAC = H(T, K)$ 
  - Parties agree with Key  $K$ , which is kept private to participants
  - Sender calculates **MAC** using  $K$  and sends along with the **Text**
  - Receiver calculates new MAC from received message ( $T'$ ) and  $K$  and compares it with MAC



A Message Authentication Code upgrades the digest from error detection to tamper detection by adding a secret key:  $MAC = H(T, K)$ . Only parties who share  $K$  can produce a valid code or verify one, which is what separates a MAC from a MIC: the secret turns a public checksum into a private one.

The sender computes the keyed digest and sends it with the text; the receiver recomputes it with the same key and compares. If an attacker modifies the text without knowing  $K$ , the recomputed MAC will not match and the tampering is detected, regardless of how cleverly the message was altered, because the attacker cannot predict what the keyed digest of the altered text will be.

The key must be kept secret and scoped to the communication context; reusing one MAC key across unrelated systems is a common source of subtle failures, because a compromise in one system becomes a forgery tool in the others. And how the key is mixed into the hash matters a great deal, a point that will come up when we discuss why naive keyed digests fail.

Web sessions, API tokens, and signed browser cookies all rely on keyed digests to prove that a message really came from the party that holds the secret, without revealing the secret itself. A MAC is the difference between 'this data is consistent' and 'this data is consistent and it is from them'. In protocol design the MAC key and the encryption key are separate values, which means a compromise of one does not automatically break the other, a small but important layer of defense in depth.

# Example usage in JWT

## Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ._syti9TdagS1-vSnVExnCud460QVKX7BxQR1YomY9cA
```

Cookie provided  
in webpage to  
Clients

Clients cannot change  
Cookie due to MAC

## Decoded

EDIT THE PAYLOAD AND SECRET

### HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  
  "typ": "JWT"  
}
```

Algorithm

### PAYLOAD: DATA

```
{  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

Data in cookie

### VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret_key  
)  
[x] secret base64 encoded
```

MAC calculated  
with secret\_key.  
Key is private to server

[https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.\\_syti9TdagS1-vSnVExnCud460QVKX7BxQR1YomY9cA](https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ._syti9TdagS1-vSnVExnCud460QVKX7BxQR1YomY9cA)

A JSON Web Token carried in a browser cookie is a MAC deployment you can inspect with the naked eye: three parts separated by dots, each with a clear role, which makes it a good concrete example of the abstract definition from before.

The header, base64-encoded, names the signing algorithm; the payload, also base64-encoded, carries the user data such as the subject and the name; and the signature is a keyed digest computed over the first two parts, using a secret known only to the server. A client can read the cookie but cannot modify it without invalidating the signature, because any change breaks the MAC over the first two parts. The base64 encoding is for transport convenience, not for secrecy: anyone can decode the payload in seconds, which is why the payload must never contain sensitive data.

The pitfalls are operational: the server must pin the expected algorithm, because a header claiming a weaker or unsigned scheme would otherwise be honored, and tokens must always be re-verified, never cached as trusted. Expired or altered tokens fail the signature check and are rejected, so the check is cheap enough to run on every request.

Single sign-on systems use exactly this pattern to let many services share one authenticated session without each service storing the user's password or holding its own secret. The token is self-contained proof that the holder was authenticated by the issuing server. It is the same pattern as signed messages and signed updates, scaled to a browser cookie: a self-contained claim plus a check that only the issuer can produce.

# Message Authentication Code (MAC)

## Approaches

- Encryption of an ordinary digest (e.g. from SHA3)
  - Using, for instance, a symmetric block cipher
- Using encryption with feedback & error propagation
  - CBC-MAC or GCM
- Adding a key to the hashed data
  - Keyed-MD5 (128 bits)
    - $\text{MD5}(K, \text{keyfill}, \text{text}, K, \text{MD5fill})$
  - HMAC (output length depends on the function H used)
    - $H(K, \text{opad}, H(K, \text{ipad}, \text{text}))$
    - $\text{ipad} = 0x36 \text{ B times}$     $\text{opad} = 0x5C \text{ B times}$     $\text{B} = \text{size of H input block}$ 
      - HMAC-MD5, HMAC-SHA-1, etc.

There are several ways to add a key to a hash, and they are not equally safe. The options range from encrypting the digest, to block-cipher modes such as CBC-MAC or GCM, to mixing the key directly into the hashed data, and the slide organizes them by where the key enters the computation.

The direct approach comes in two forms. A naive keyed-MD5 style scheme just feeds key, filler, and text into the hash, which is weak and is the exact construction broken by length-extension attacks. The standard construction is HMAC, computed as  $H(K \text{ opad}, H(K \text{ ipad}, M))$ , where the key is combined with an inner pad of 0x36 bytes and an outer pad of 0x5C bytes, B being the block size of the inner hash. The output length follows the underlying function, giving HMAC-MD5, HMAC-SHA-1, and so on.

The trap is rolling your own keyed hash: without understanding the structure, it is easy to build something that looks right but is vulnerable, and the two-pad structure of HMAC is precisely what blocks the length-extension path. The rule of thumb is to use HMAC with a safe underlying hash such as SHA-256, and never to invent a variant, even a small one.

TLS, IPsec, and most token formats use HMAC, so its two-pass structure is one of the most deployed cryptographic patterns in everyday networking, and it is worth memorizing: inner pad 0x36, outer pad 0x5C, two passes, one key. The two constants, 0x36 and 0x5C, look arbitrary; they are fixed public values, and the security comes from the double application of the key, not from keeping anything secret.

# Message Authentication Code (MAC)

## When used with encryption

- **Encrypt-then-MAC: MAC is computed from ciphertext:  $M = C \parallel \text{MAC}(C, K_2)$ ,  $C = E(T, K_1)$** 
  - Allows verifying integrity before decryption
  - MAC calculation is frequently faster than decryption
- **Encrypt-and-MAC: MAC is computed from plaintext:  $M = E(T, K_1) \parallel \text{MAC}(T, K_2)$** 
  - May give information regarding original text (if similar to other text)
  - Receiver will find that text was manipulated **only after decryption plus MAC calculation (slower)**
  - Manipulated ciphertext may attack the decryption algorithm without detection
- **MAC-then-Encrypt: MAC is computed from plaintext:  $M = E(T \parallel \text{MAC}(T, K_2), K_1)$** 
  - MAC is encrypted (which is not bad)
  - Receiver will find that text was manipulated **only after decryption plus MAC calculation (slower)**
  - Manipulated ciphertext may attack the decryption algorithm without detection

BAD

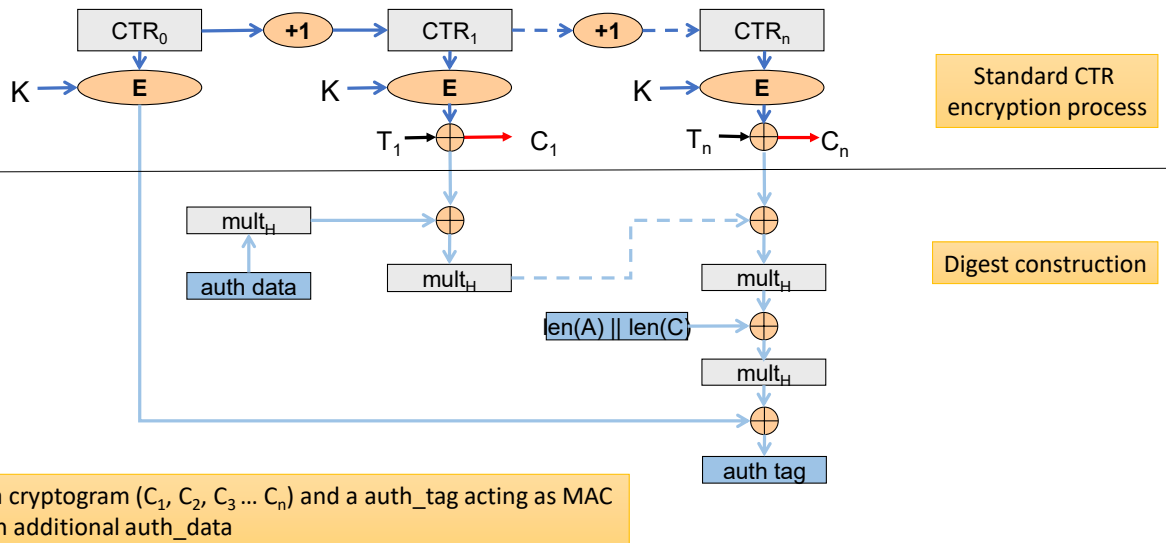
When a message is both encrypted and authenticated, the order in which encryption and the MAC are applied matters for security. Three orders are possible, and only one of them is safe, which is surprising because the difference is a matter of where the MAC is computed, not of which algorithms are used.

Encrypt-then-MAC computes the MAC over the ciphertext: the message is  $C = E(T, K_1)$  followed by  $\text{MAC}(C, K_2)$ . The receiver can verify authenticity before decrypting, and a tampered ciphertext is rejected without ever reaching the cipher. The other two orders MAC the plaintext, which leaks information about it and lets an attacker probe the decryption routine with crafted ciphertexts, all undetected until after decryption.

The common pitfall is choosing Encrypt-and-MAC because it is slightly cheaper and simpler to implement; it is convenient, not safe. MAC-then-Encrypt, where the MAC is encrypted inside the ciphertext, has the same late-detection problem and is marked bad for the same reason. In both bad orders, the decryption algorithm sees attacker-chosen input before any integrity check can reject it.

Modern authenticated encryption modes implement exactly the encrypt-then-MAC principle, verifying the authentication tag before any plaintext is released, which is the pattern every new protocol should copy. When a protocol is found to be vulnerable to padding oracles, the root cause is almost always a MAC ordering or verification mistake of this kind. The lesson generalizes: in any pipeline that combines protection steps, verify before you process, so that malformed or malicious input never reaches the expensive operation.

## Example: GCM (Galois Counter Mode)



GCM is an authenticated encryption mode that combines the CTR stream of a block cipher with a keyed digest called GMAC, producing both the ciphertext and a 16-byte authentication tag in one pass. It is the standard answer to the question of how to get confidentiality and integrity together in a single mode.

The plaintext is encrypted in counter mode by XORing it with successive encryptions of a counter. In parallel, a digest built over the ciphertext and any additional authenticated data, using Galois field multiplications keyed by  $H$ , accumulates into the final tag, which acts as the MAC over the whole cryptogram, including the lengths of the authenticated data and the ciphertext, so even length changes are detected.

The pitfalls are operational rather than theoretical: the nonce must never be reused with the same key, and the tag must be verified before the plaintext is accepted. A mode that looks like plain encryption becomes unsafe the moment the tag check is skipped or the nonce is repeated, and nonce reuse in particular can leak both the authentication key and the plaintext. Both mistakes are silent: the receiver gets plausible output either way, which is why the tag check belongs in the protocol specification, not in application code that could forget it.

GCM is the default cipher mode in TLS 1.3 and in many disk and storage encryption schemes precisely because it delivers confidentiality and integrity in a single fast operation, without a separate MAC step, and because it parallelizes well on modern hardware.

# Key derivation

## Motivation

- Ciphers require fixed dimension keys
  - 56, 128, 256... bits
- Original source may have low entropy
  - Reduces the difficulty of a brute force attack
  - Although we must have some strong relation into a useful key
- Derive keys from multiple sources
  - Shared secrets
  - Passwords generated by humans
  - PIN codes and small length secrets
- Multiple keys from the same material
  - While not allowing to find the material (a password, another key) from the new key

Key derivation turns material that is not a proper key, such as a password, a PIN, or a short shared secret, into a key of the size and quality that an algorithm actually requires. It is the bridge between human secrets, which are short and low in entropy, and machine keys, which are long and uniformly random.

Ciphers need fixed-size keys such as 128 or 256 bits, but human secrets are short and low in entropy, which makes brute force easy if they are used directly. A derivation function imposes a strong, one-way relationship: the original material should not be recoverable from the derived key, while the same material can seed multiple different keys for different purposes, each in a distinct context. The one-way direction is what allows the same password to seed keys on a phone, a laptop, and a server without ever transmitting the password itself: everyone derives, no one sends.

The pitfall is skipping derivation entirely and using a raw password as a key: the security collapses to the entropy of the password, and an attacker can brute force it offline at full machine speed, with no network in the loop. Derivation is what converts a human secret into something a machine can safely use, and its cost is a feature, not a bug.

Every password login, full-disk encryption, and VPN handshake is silently running a key derivation function before the first real encryption happens; the user never sees it, but its absence, or its use with too few rounds, is a serious vulnerability that auditors check for first.

# Key derivation

## Purposes

- **Key reinforcement:** increase the security of a password
  - Usually defined by humans
  - To make dictionary attacks impractical
- **Key expansion:** increase/decrease the length of a key
  - Expansion to a size that suits an algorithm
  - Eventually derive other related keys for other algorithms (e.g. MAC)

Key derivation serves two main purposes: reinforcement and expansion, and a real system usually needs both at the same time, which is why serious key derivation functions do the two in one pass.

Reinforcement takes a weak, human-chosen password and stretches it into a stronger key by making each guess expensive, which is what defends against dictionary attacks: the attacker pays the full derivation cost for every candidate password they try. Expansion resizes a key to fit an algorithm and can also derive several related keys from one secret, for example one for encryption and another for a MAC, so the two purposes never share the same byte sequence.

The pitfall is deriving everything from one password without separating the derived keys: if the same derived value is used for both encryption and authentication, a weakness or leak in one use can bleed into the other. Distinct contexts should get distinct derived keys, usually by mixing in a context label, which also prevents one service's key material from being replayed in another. A context label, usually a short string naming the purpose, is mixed into the derivation input so that the bytes for encryption and the bytes for a MAC can never coincide, even for the same password.

Full-disk encryption systems derive a separate encryption key and a separate integrity key from the user's passphrase, so a compromise of one purpose does not hand over the other. The same pattern appears in protocols that need many keys from one secret. The same idea appears in hardware security modules, which derive per-session keys from a master key combined with a session label.

# Key derivation

- Key derivation requires the existence of:
  - A **Salt** which makes the derivation unique
  - A difficult problem
  - A chosen level of complexity
- Computational difficulty
  - Transformation requires relevant computational resources
- Memory difficulty
  - Transformation requires relevant storage resources
  - Limits attacks using dedicated hardware accelerators

A serious key derivation function needs three ingredients: a salt, a deliberately hard problem, and a tunable level of complexity. Missing any one of the three leaves a hole that the other two cannot fill.

The salt makes each derivation unique, so identical passwords produce different outputs and precomputed tables become useless: the attacker must redo the work per account instead of buying a generic table. The difficulty can be computational, requiring large amounts of CPU work, or memory-hard, requiring a large block of storage to complete. Memory-hardness is especially valuable because it limits attacks that use specialized hardware accelerators, which are easy to build for pure hashing. A unique salt per account also means a leaked database cannot be attacked with one precomputed effort: every row forces fresh work, which multiplies the attacker's bill by the number of accounts.

The pitfall is underestimating the tuning: a derivation with too few rounds or too little memory is no stronger than a plain hash, and the cost must be revisited as hardware gets faster and cheaper. The parameter is a moving target, not a constant to set once, and the cost is usually chosen so a single derivation takes a fraction of a second on a server but minutes on a cracking rig.

Password managers and operating systems bake a salt and a high cost into every stored credential, which is why cracking one leaked hash no longer unlocks the other accounts of the same user, and why two users with the same password produce two unrelated stored values.

# Key derivation

## Simple Approach: A Digest function

- Arguments:
  - Salt = A random value
  - Password = a secret (provided by humans)
  - H = An adequate Digest Function

$$\text{key} = H(\text{password}, \text{salt})$$

- Advantages:
  - Key has a large length, and can be truncated to the adequate length
  - Two passwords will result in different keys
  - Finding the key will not lead to the password
- Issues: simple, enabling brute force/dictionary attacks

The simplest key derivation is a plain digest of the password and a salt:  $\text{key} = H(\text{password}, \text{salt})$ . It is a useful starting point that makes the structure clear, but on its own it is not a defense against a determined attacker.

The salt, a random per-user value, stops identical passwords from producing identical keys and defeats precomputed rainbow tables, because the table would have to be rebuilt for every salt. The output is a long digest that can be truncated to the needed key length, and recovering the password from the key remains computationally infeasible, which is the one-way property we want from any digest. The truncated key length is another quiet decision: taking more bytes from the digest than the algorithm needs wastes nothing, but taking fewer throws away security for no benefit.

The fatal flaw is speed: a fast digest means a GPU can try millions of candidate passwords per second, so dictionary and brute-force attacks stay practical for an attacker with a leaked database, and the salt, while it blocks tables, does nothing to slow the guessing itself. This approach is acceptable only for low-sensitivity or legacy data, and it is exactly what the next slides improve upon. The salt and the digest each solve half of the problem; together with a cost parameter, they become the full recipe that the slower functions below implement.

Many older systems still store salted SHA-1 or MD5 password hashes, and security teams routinely flag them for replacement with slow, memory-hard derivation functions. Seeing a salted fast hash in a credential store is a standard finding in any audit.

# Key derivation

## Password Based Key Derivation Function (PBKDF2)

- Produces a key from a password, with a chosen difficulty
- **$K = \text{PBKDF2}(\text{PRF}, \text{Salt}, \text{rounds}, \text{dim}, \text{password})$** 
  - PRF: Pseudo-Random-Function: a digest function
  - Salt: a random value
  - Rounds: the computational cost (hundreds of thousands)
  - Dim: the size of the result required
- Operation: calculate  $\text{ROUNDS} \times \text{DIM}$  operations of the PRF using the SALT and Password
  - Higher number of rounds will increase the cost of brute force/dictionary attacks

PBKDF2 is the standard way to turn a password into a key with a chosen difficulty. It wraps a pseudo-random function, typically a keyed digest such as HMAC-SHA-1 or HMAC-SHA-256, in a deliberately slow loop, which is the simplest way to buy security with computation.

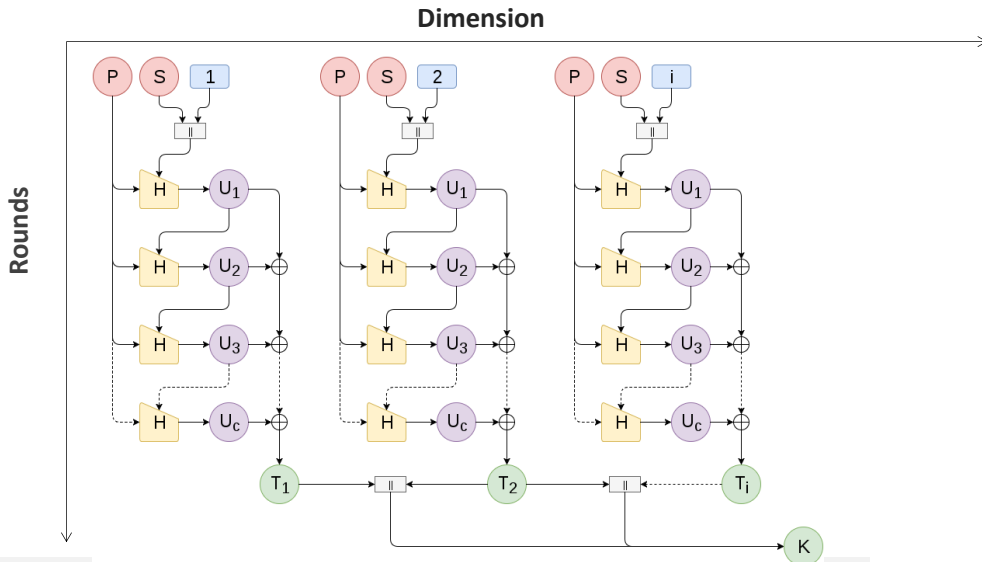
The call  $K = \text{PBKDF2}(\text{PRF}, \text{salt}, \text{rounds}, \text{dim}, \text{password})$  takes the salt, the number of rounds, in practice hundreds of thousands, and the desired output length dim, then applies the PRF repeatedly over that material. Each extra round multiplies the attacker's cost per guess, which is the whole point of the rounds parameter: the honest user pays it once, the attacker pays it for every candidate password. Because the construction is a loop over a standard PRF, two implementations that agree on the parameters always produce the same key, which is what makes PBKDF2 portable across languages and platforms.

The pitfalls are choosing too few rounds, reusing the same salt or password across services, and forgetting that the cost parameter must grow over time as attacking hardware improves. A PBKDF2 setup that was strong a decade ago can be cheap to attack today, which is why the round count is a policy setting that gets revisited, not a constant.

PBKDF2 is built into virtually every database, file format, and protocol that derives keys from a user's password, which is why its round count is one of the first numbers a security auditor checks, and why configurations publish it openly: the cost is part of the design, not an implementation secret.

# Key derivation

## Password Based Key Derivation Function (PBKDF2)



The two knobs of PBKDF2 are the round count, which sets the computational cost, and the output dimension, which sets the final key length, and the diagram on the slide shows how the two are produced by two different loops.

Each round runs the PRF over the salt and password and feeds the result back in, so the cost scales linearly with the round count, while the final key is stretched to the requested number of bytes by chaining further blocks of PRF output. The round loop and the expansion chain are the two halves of the diagram: one makes it slow, the other makes it the right size.

The practical pitfall is treating the diagram as a fixed recipe: the round count is a policy decision, not a constant, and it must be revisited as attacking hardware gets cheaper, while the dimension must match exactly what the consuming algorithm expects, not one byte more or less, because a key of the wrong length is simply rejected.

When a service says it stretches passwords with a high work factor, this round loop is exactly the mechanism being described, and the work factor is nothing more than the rounds parameter made public, which is why you can compare the strength of two services by reading the number.

# Key derivation

## scrypt

- Produces a key with a chosen computation and storage cost
- **$K = \text{scrypt}(\text{password}, \text{salt}, n, p, \text{dim}, r, \text{hLen}, \text{Mflen})$** 
  - Password: a secret
  - Salt: a random value
  - N: the cost parameter
  - P: the parallelization parameter.  $p \leq (2^{32} - 1) * \text{hLen} / \text{Mflen}$
  - Dim: the size of the result
  - R: the size of the blocks to use (default is 8)
  - hLen: the size of the digest function (32 for SHA256)
  - Mflen: bytes in the internal mix (default is  $8 \times R$ )

scrypt extends password derivation with a memory-hard component: besides CPU work, it forces the attacker to allocate and repeatedly sweep a large block of memory, which is far harder to scale on GPUs and ASICs, because those devices are fast at computing but poor at moving large amounts of data.

The call  $K = \text{scrypt}(\text{password}, \text{salt}, n, r, p, \text{dim}, \text{hLen}, \text{Mflen})$  takes a cost parameter  $n$ , a block size  $r$  with a default of 8, a parallelism parameter  $p$  bounded by 2 to the 32 minus 1 times  $\text{hLen}$  divided by  $\text{Mflen}$ , and the output length  $\text{dim}$ . Internally it first derives a key with a PBKDF2-style step and then runs a mixing function over the memory block, so the storage requirement scales with  $n$  times  $r$  times 8 bytes.

The pitfall is picking parameters that are affordable for the server but trivial for a GPU farm: if the memory block is small, the memory-hardness is nominal and the function collapses toward a plain hash. The whole benefit of scrypt is that the memory requirement cannot be parallelized away as easily as raw CPU work, so the cost stays high even on specialized hardware. A useful calibration is to measure the derivation time on your own server: a few hundred milliseconds is strong for a login, while a few seconds is appropriate for encrypting an archive that must stay protected for years.

scrypt was designed precisely because GPU and ASIC cracking had outpaced CPU-only functions like PBKDF2, and it is still widely used in file formats and blockchains where Argon2 is not yet available. Recognizing its parameter names in a configuration tells you the system is paying for memory, not just CPU.

## Key derivation: scrypt

- Produces a key with a chosen storage cost
- $K = \text{scrypt}(\text{password}, \text{salt}, n, p, \text{dim}, r, \text{hlen}, \text{mflen})$ 
  - Password: a secret
  - Salt: a random value
  - N: the cost parameter
  - P: the parallelization parameter.  $p \leq (2^{32} - 1) * \text{hlen} / \text{Mflen}$
  - Dim: the size of the result
  - R: the size of the blocks to use (default is 8)
  - hLen: the size of the digest function (32 for SHA256)
  - Mflen: bytes in the internal mix (default is  $8 \times R$ )

SIO

This is the same scrypt construction recapped from the parameter side: a password-based key derivation that charges the attacker for both computation and storage at the same time, which is the property that separates it from the pure CPU work of PBKDF2.

The parameters  $n$ ,  $r$ , and  $p$  control the cost:  $n$  sets the number of mixing iterations,  $r$  the size of each memory block, and  $p$  the number of parallel branches, bounded by the output and hash sizes. The salt keeps each derivation unique so precomputed tables do not apply, and the output dimension fixes the resulting key length, which must match the consuming algorithm. None of the parameters is a secret; they are published configuration, and the security comes from the cost they impose, not from hiding them.

As a recap, the storage-cost requirement is the differentiator: an attacker who wants to try many guesses must keep a large working memory per guess, which is far more expensive to scale than raw CPU speed, and that is exactly the asymmetry the design buys. The server pays a modest cost once; the attacker pays a storage cost per guess, and the two curves diverge as hardware improves. The per-guess memory cost also caps the parallelism of an attack: a farm of accelerators cannot each hold a full working block, so the speedup is bounded by the memory budget.

File formats and blockchains that predate Argon2 commonly use scrypt as their password or key-stretching layer, so its parameter names are worth recognizing in configuration files and protocol documents, where the memory size is often the single number that tells you how seriously the system takes credential security.

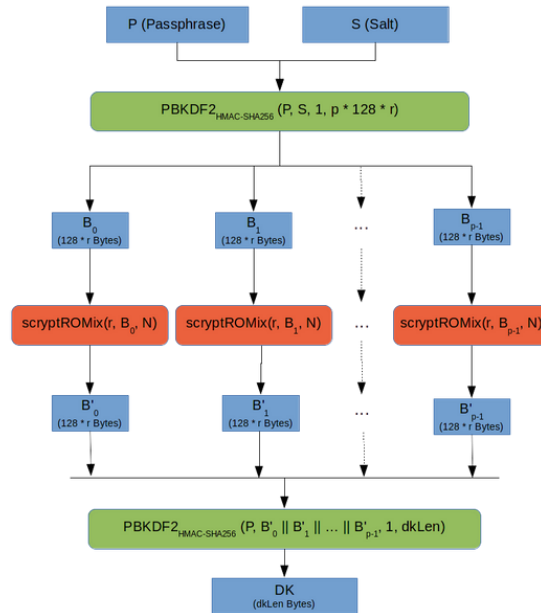
# Key derivation

## script

script (P, S, N, r, p, dkLen)

Parameters:

N (CPU/Memory Cost Parameter)  
r (Block Size)  
p (Parallelization Parameter)  
dkLen (Output Length)



The script pipeline starts by stretching the password and salt with a PBKDF2-style step, then expands the result into a large memory block that is mixed repeatedly until the key is produced. The diagram separates the two phases: an initial key setup, and the memory-intensive mixing that follows.

The mixing step is where the memory-hardness lives: each pass reads and rewrites the entire block in a data-dependent order, so the work cannot be done in a few small registers and must actually touch storage, which is what punishes hardware that is fast at hashing but poor at moving data. The access pattern depends on the data being processed, so the work cannot be shortcut or precomputed.

The pitfall is reading the pipeline as if more passes always meant more security without checking the memory size: if the block is small, the memory-hardness is nominal and the function behaves like a plain iterative hash, no matter how many rounds are configured. The memory block size, not the round count, is the number that decides whether the design earns its name. Checking the memory size is a one-line configuration review that catches most misconfigured deployments.

This two-phase structure, cheap setup plus expensive mixing, is the same shape as the PBKDF2 loop but with a storage cost bolted on, and it is why script configurations always quote both the memory size and the iteration count: either number alone tells you nothing about the real cost. It is the same reasoning as choosing the round count for PBKDF2: the published numbers define the cost, and the cost, not the secrecy, is what protects the password.

# Modern key derivation: Argon2

- Argon2: winner of the Password Hashing Competition
  - Memory-hard, resistant to GPU and ASIC attacks
  - Variants: Argon2d (data-dependent) and Argon2i (data-independent)
- Relevant Parameters
  - MemorySize, iterations, parallelism, password, salt, extraData and output length
  - Tune memory and iterations to the target platform
- Argon2 vs PBKDF2 and scrypt
  - PBKDF2: CPU-only, cheap to attack on GPUs/ASICs
  - scrypt: memory-hard;

SIO

Argon2 is the current recommendation for password hashing. It won the Password Hashing Competition, an organized contest that pitted candidate designs against each other under scrutiny, and it is memory-hard by design, which makes it resistant to attacks from GPUs and purpose-built ASICs.

It comes in three variants: Argon2d uses a data-dependent memory access pattern, Argon2i a data-independent one that resists timing side channels, and Argon2id a hybrid of both that is the common default. Its parameters, memory size, number of iterations, parallelism, salt, and output length, are all tunable and should be set against the target platform, with the memory size usually dominating the cost.

Compared with the alternatives, PBKDF2 is CPU-only and cheap to attack on modern hardware, while scrypt is memory-hard but less flexible in its cost model. The practical rule is to prefer Argon2id for new password storage and to re-tune the memory and iteration settings as hardware improves, rather than leaving them fixed for years, because the cost parameter is a moving target like the others.

Major operating systems, password managers, and web platforms have been moving new credential storage to Argon2, making it the de facto standard for anything password-derived going forward. If a system you manage still uses a fast salted hash, the migration path is usually to Argon2id with a memory size chosen for your servers. The migration is low-risk in the common case: existing hashes are re-derived on the next successful login, so the switch happens gradually across the user base.