Digests, Integrity Control and Key Derivation

SIO

João Paulo Barraca

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

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

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 >=256

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: 9649d8c0d25515a239ec8ec94b293c8868e931ad318df4ccd0dffd67aff89905
    - T2: 6453be3f643d0a7e9b5890eed76bb63df8b6b071b30d5f97269a530c289b9839

Digest Functions

Lets check: Content dependency

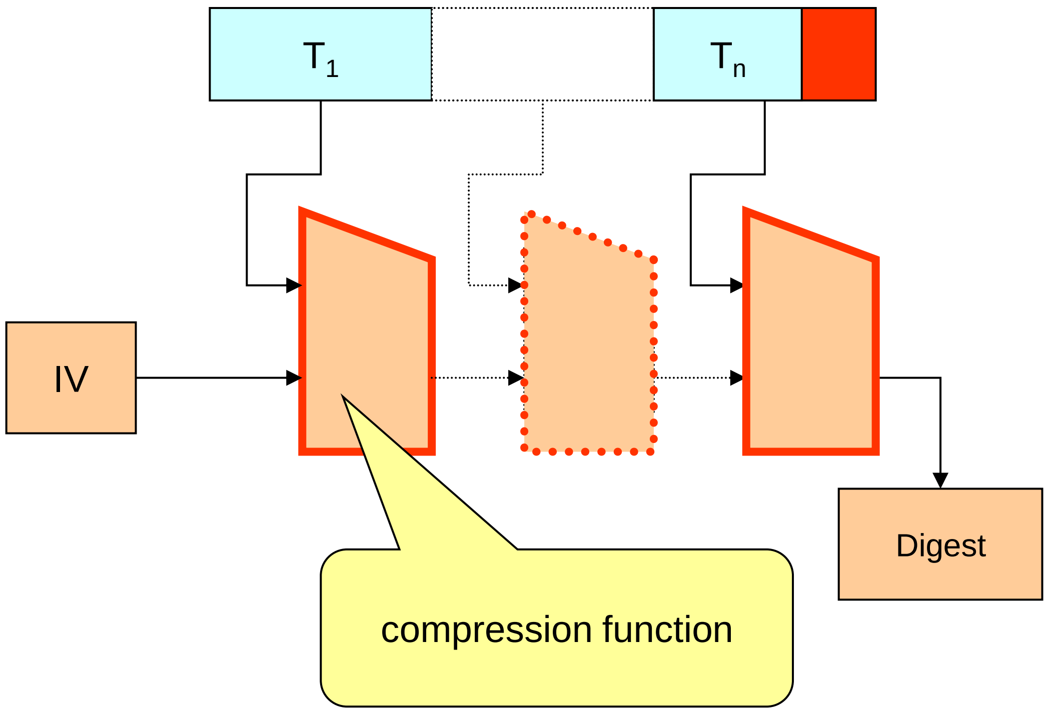
- Considering the similar, yet different texts (1 bit difference 'B' -> 'C'):
  - T1: "`Hello User_B!`", `[0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x55, 0x73, 0x65, 0x72, 0x5f,` **`0x42`**`, 0x21]`
  - T2: "`Hello User_C!`", `[0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x55, 0x73, 0x65, 0x72, 0x5f,` **`0x43`**`, 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: `e663a01d3bec4f35a470aba4baccece79bf484b5d0bffa88b59a9bb08707758a`
    - T2: `69f78345da90c6b8d4785b769cd6ae09e0531716fe5f5a392fde1bdc70a2bb7d`

Digest Functions

Approaches

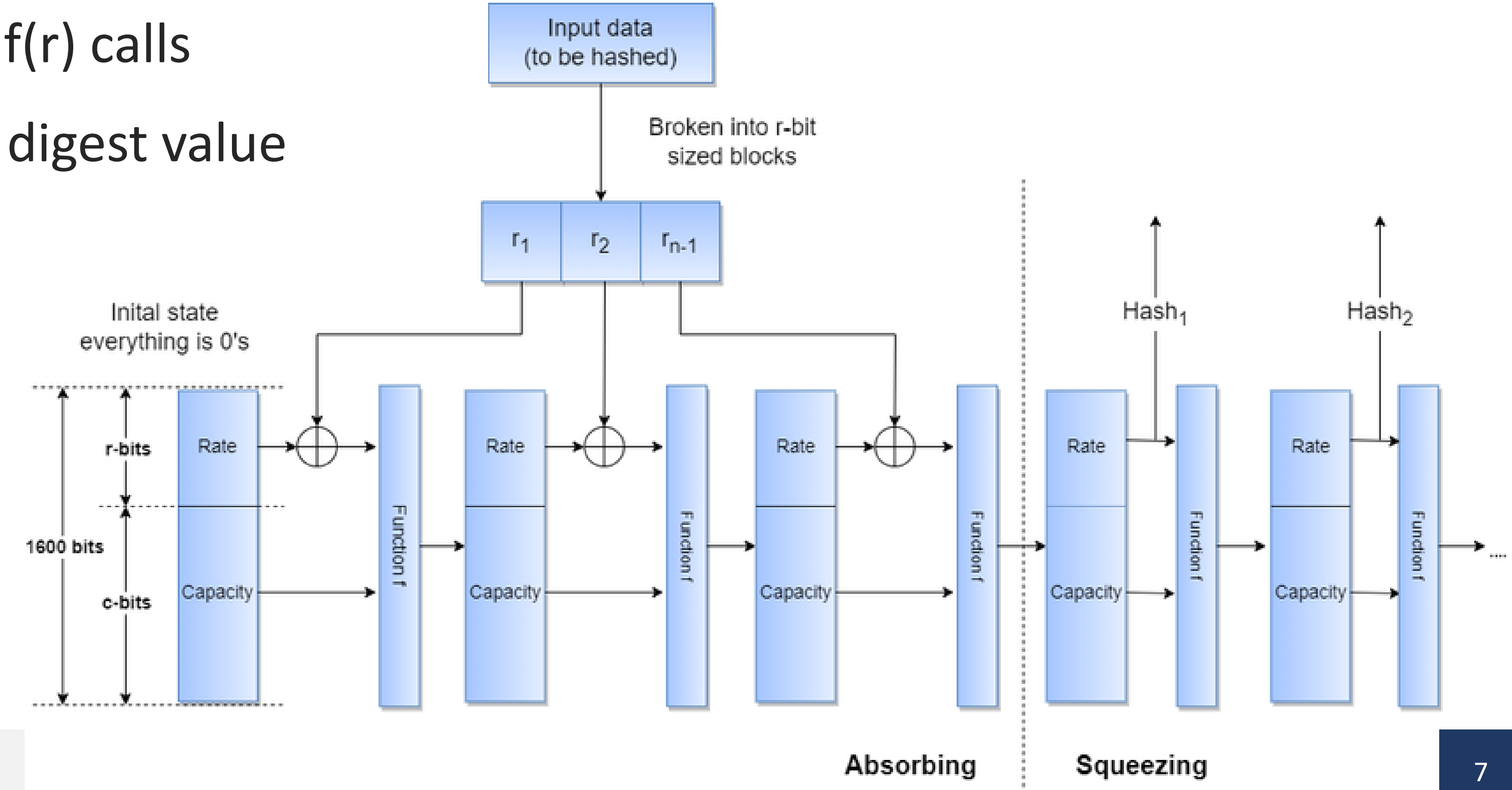
- Merkle-Damgård construction
  - Collision-resistant, one-way compression functions
    - Can be a block cipher!
  - Iterative compression
  - Length padding
  - Digest size is the last block
  - Can be resumed!
    - Digest is the state at T_n
  - Algorithms: MD5, SHA1, SHA2



Digest Functions

Approaches

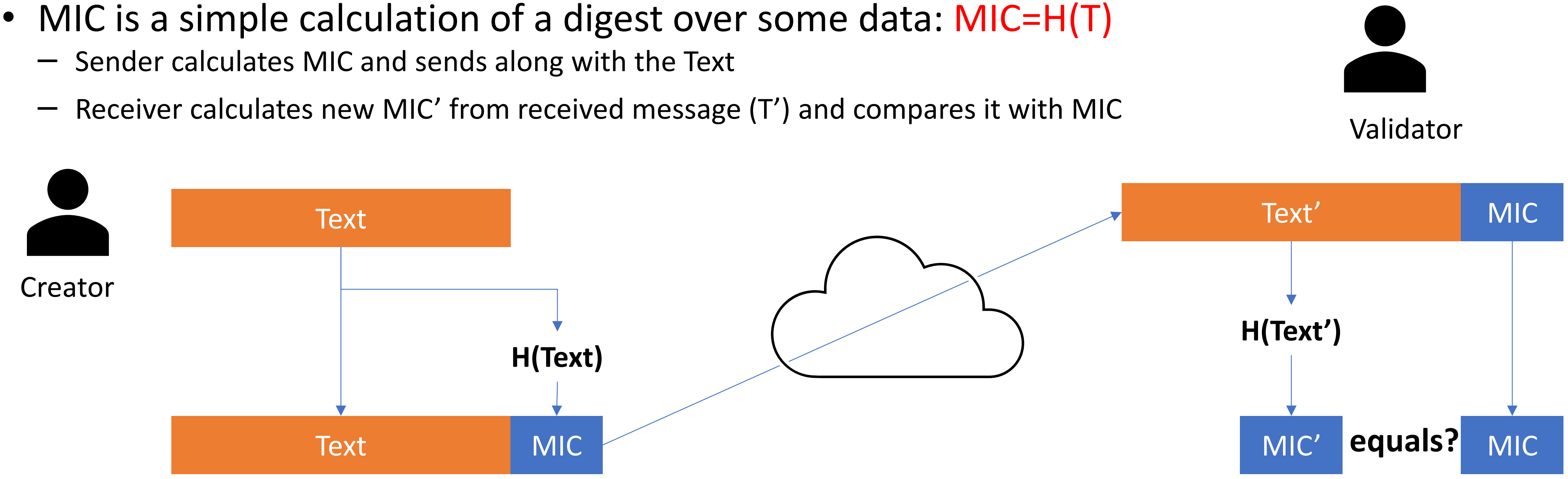
- Sponge functions
  - Data split in *r* sized blocks
  - Absorbing phase: chained f(r) calls
  - Squeezing: extract bits for digest value
  - Algorithms: SHA3



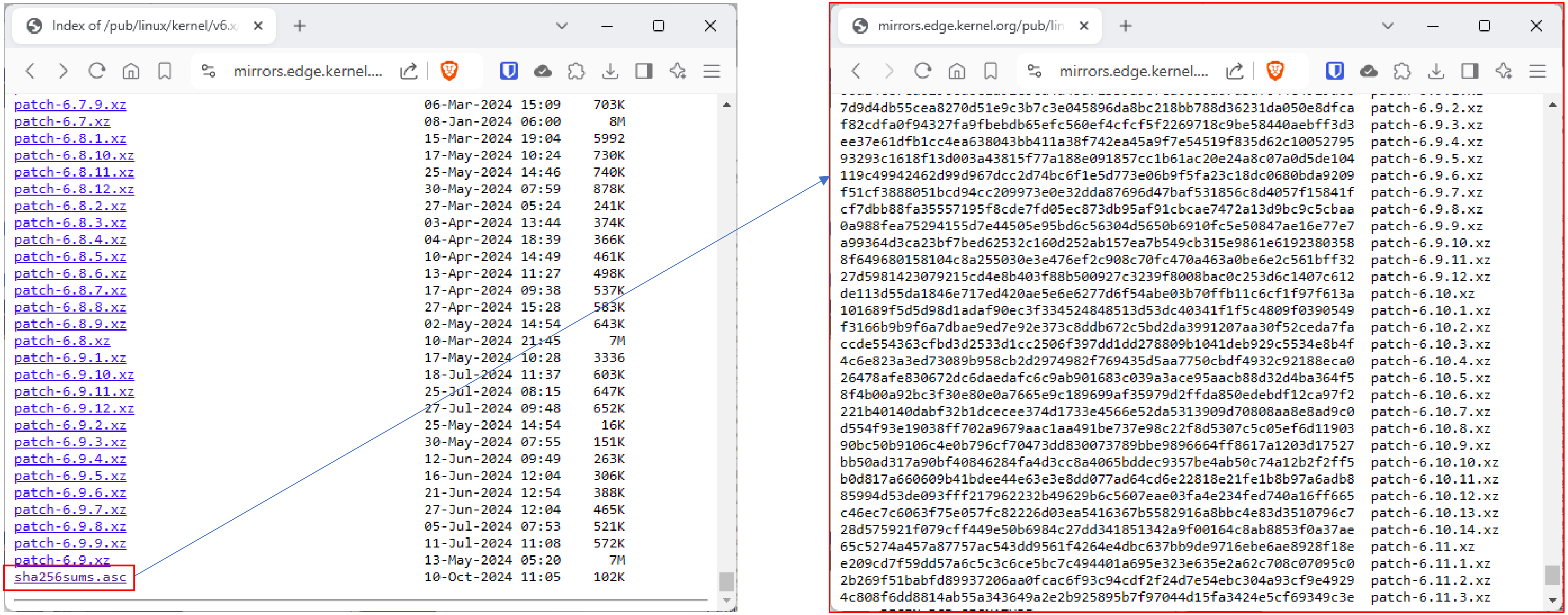
Message Integrity Code (MIC)

- 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



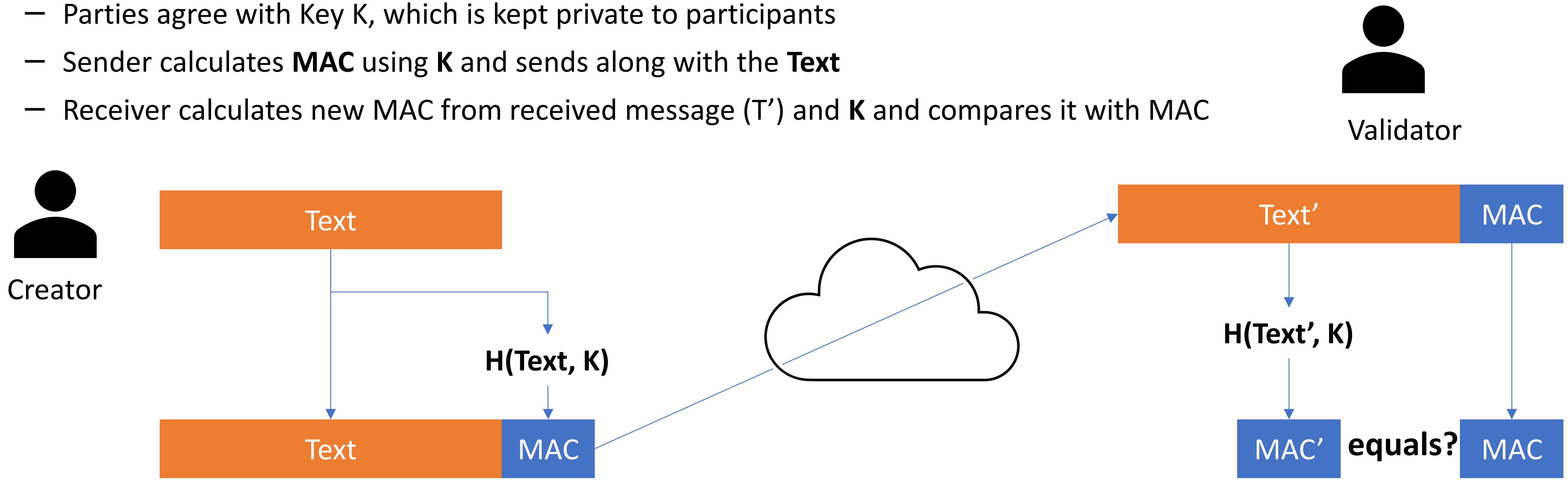
Example usage at kernel.org to validate file integrity



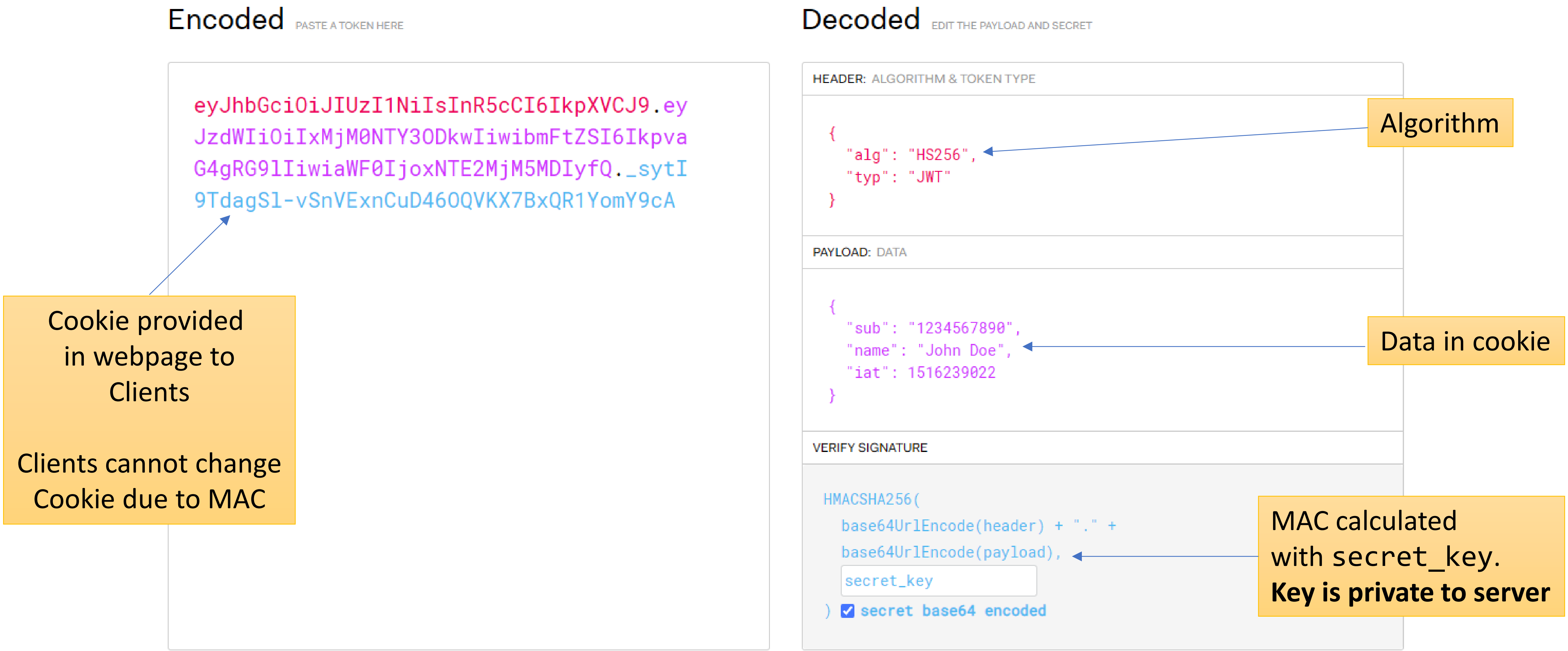
Message Authentication Code (MAC)

- 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: MIC=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



Example usage in JWT



https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ._sytI9TdagSl-vSnVExnCuD46OQVKX7BxQR1YomY9cA

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)
    - MD5(K, keyfill, text, K, MD5fill)
  - HMAC (output length depends on the function H used)
    - H(K, opad, H(K, ipad, text))
    - ipad = 0x36 B times opad = 0x5C B times B = size of H input block
      - HMAC-MD5, HMAC-SHA-1, etc.

Message Authentication Code (MAC)

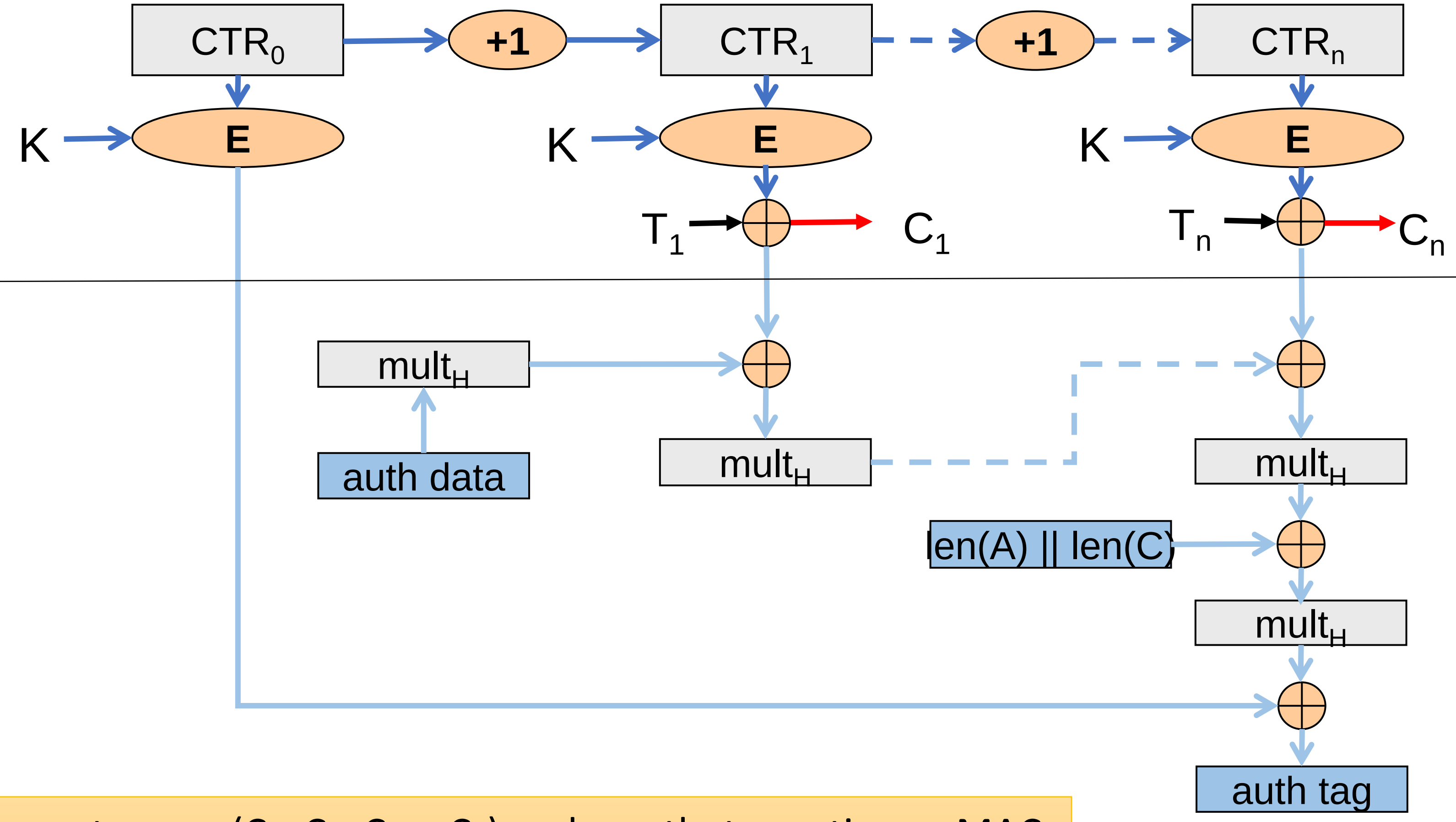
When used with encryption

- Encrypt-then-MAC: MAC is computed from cryptogram: $M = C \mid MAC(C, K_2)$, $C=E(T, K_1)$
  - Allows verifying integrity before decryption
  - MAC calculation is frequently faster than decryption

BAD

- **Encrypt-and-MAC: MAC is computed from plaintext: $M = E(T, K_1) \mid MAC(T, K_2)$**
  - 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: $M = E(\ T \mid MAC(T, K_2), K_1)$**
  - 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

Example: GCM (Galois Counter Mode)



Standard CTR encryption process

Digest construction

Results in a cryptogram (C_1, C_2, C_3 … C_n) and a auth_tag acting as MAC
Requires an additional auth_data

Key derivation

Motivation

- Cipher algorithms require fixed dimension keys
  - 56, 128, 256… bits
- We may need to derive keys from multiple sources
  - Shared secrets
  - Passwords generated by humans
  - PIN codes and small length secrets
- 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
- Sometimes we need multiple keys from the same material
  - While not allowing to find the material (a password, another key) from the new key

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

- 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

Key derivation

Simple Approach: A Digest function

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

key = H(password, salt)

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

Key derivation

Password Based Key Derivation Function (PBKDF2)

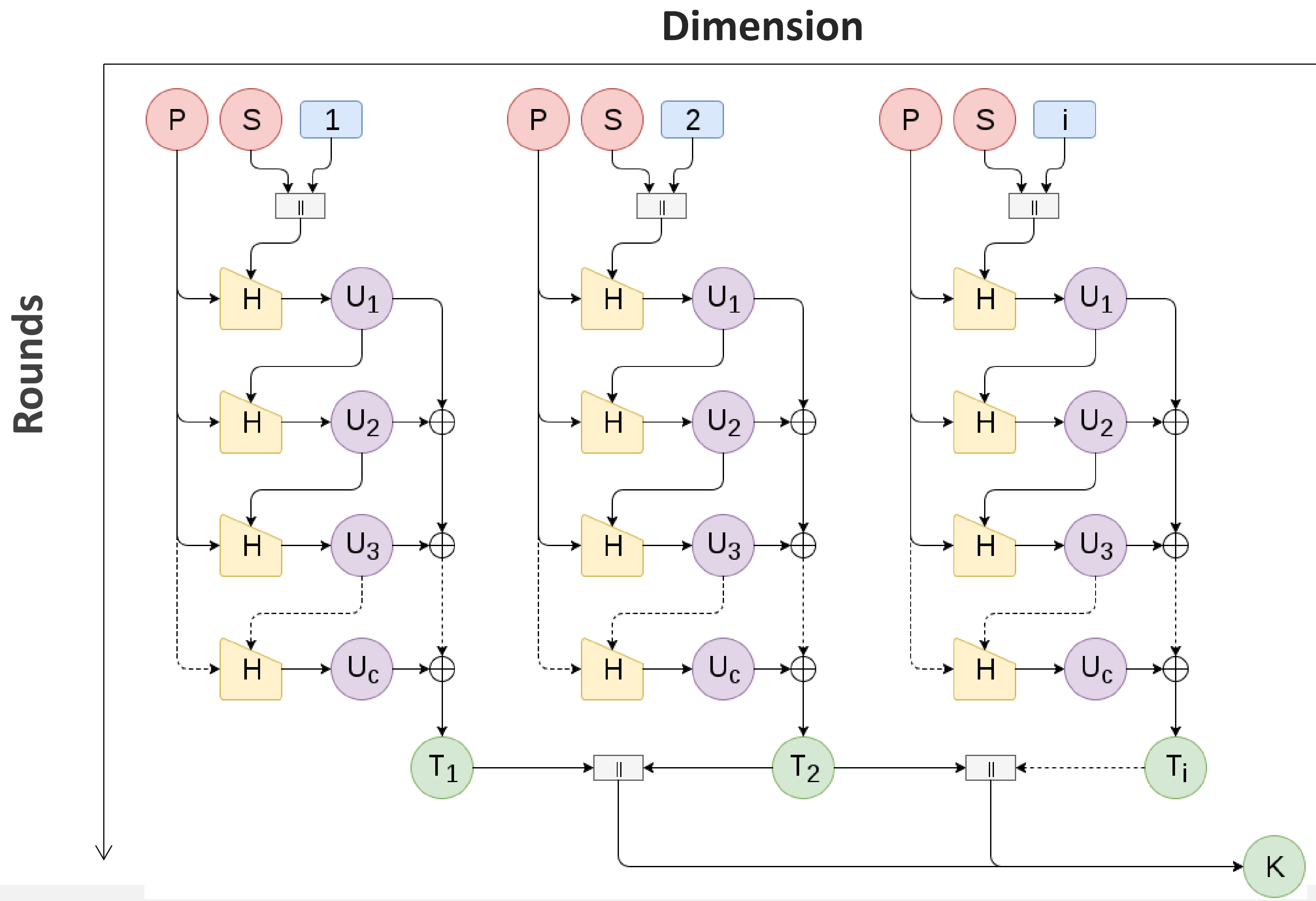
- Produces a key from a password, with a chosen difficulty

- **K = PBKDF2(PRF, Salt, rounds, dim, 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 ROUNDS x DIM operations of the PRF using the SALT and Password
  - Higher number of rounds will increase the cost of brute force/diccionary attacks

Key derivation

Password Based Key Derivation Function (PBKDF2)



Key derivation

scrypt

- Produces a key with a chosen computation and storage cost

- **K = scrypt(password, salt, n, p, dim, r, hLen, Mflen)**
  - Password: a secret
  - Salt: a random value
  - N: the cost parameter
  - P: the parallelization parameter. p ≤ (232− 1) * hLen / 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 x R)

Key Derivation: scrypt

- Produces a key with a chosen storage cost

- K = scrypt(password, salt, n, p, dim, r, hLen, Mflen)
  - Password: a secret
  - Salt: a random value
  - N: the cost parameter
  - P: the parallelization parameter. $p \leq (2^{32} - 1) * hLen / MFLen$
  - 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 x R)

Key derivation

scrypt