Cryptography Engineering

- Lecture 2 (Oct 29, 2026)

- Today:
  - More on symmetric primitives

Recap

- Hash functions
- Symmetric-key encryption
- MAC
- KDF
- AE(AD)

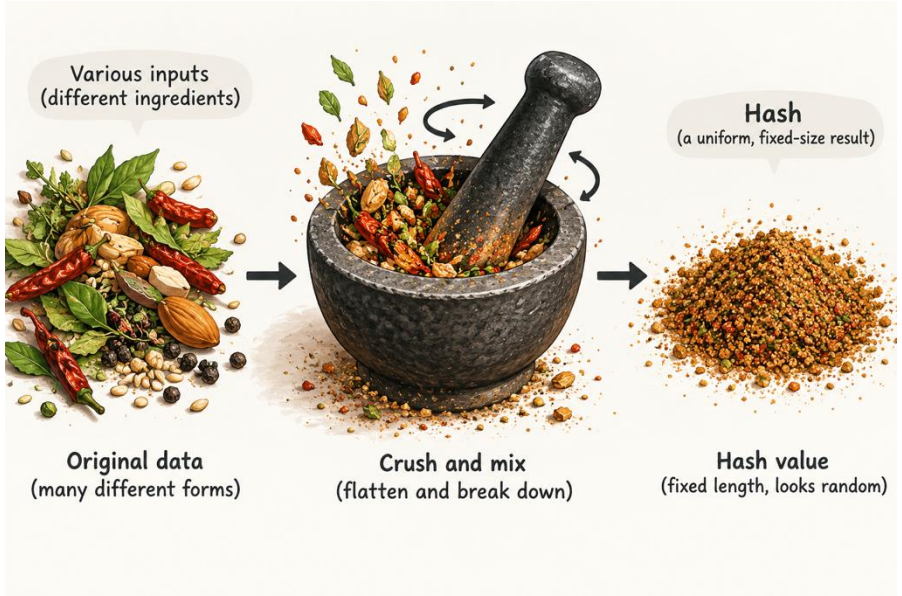
Today's Topics

- Hash Functions:
  - ~~MD5, SHA1,~~ **SHA2, SHA3, SHAKE**
- Message authentication code (MAC):
  - **HMAC**
- Key Derivation Function (KDF):
  - **HKDF**
- Symmetric-key encryption (SKE):
  - **AES, AES-CBC, AES-CTR**
- Authenticated Encryption (AE):
  - **Encrypt-then-MAC**

Hash Function

- Hash function:

  H(“…arbitrary-length string…”) = a fixed length (e.g., 32) byte string **(digest)**



Generated by ChatGPT-6 Sol

SHA2 Family

- SHA-224, **SHA-256**, SHA-384, SHA-512
- The number at the end tells the output length (28, 32, 48, 64 bytes)

SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes

SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes

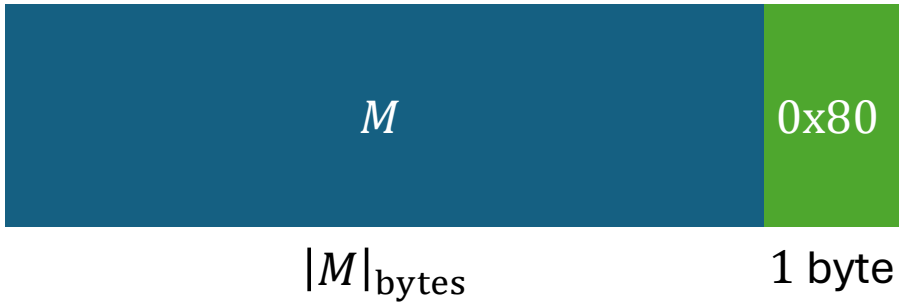


Merkle–Damgård padding

SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes



SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes



SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes



$$|M|_{\text{bytes}} + 1 + k + 8 \equiv 0 \ (mod\ 64)$$

SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes

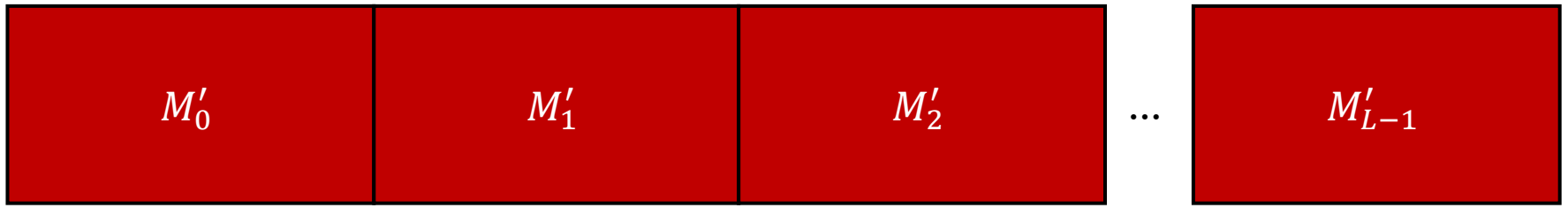


$$|M'|_{\text{byte}} = 64 \cdot L$$

SHA2 Family

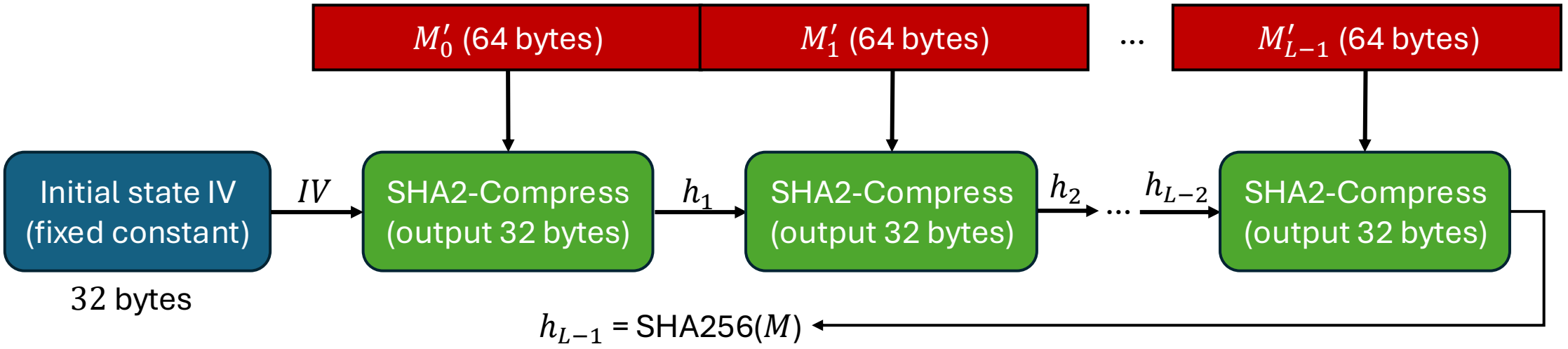
- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes

- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes
- Split the padded M into L blocks (64 bytes each)



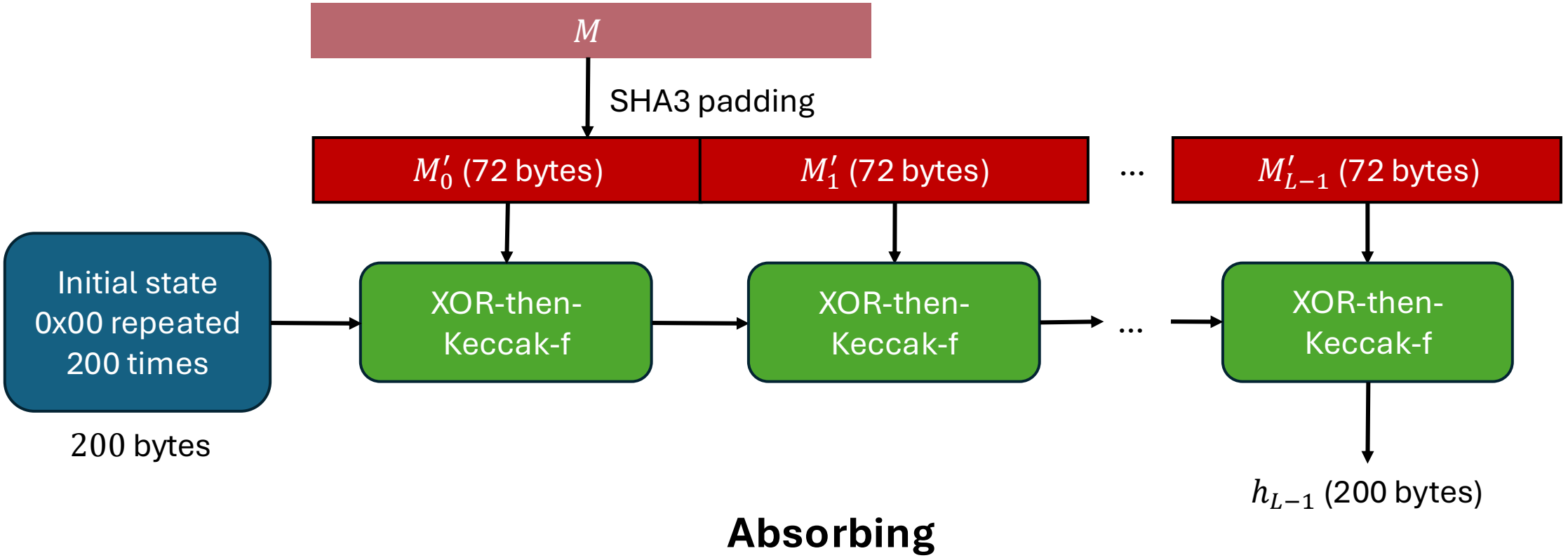
SHA2 Family

- **SHA-256**
  - Output length: 32 bytes
  - Block size: 64 bytes
- Let M be a a message. SHA-256 first pads M so that its length is a multiple of 64 bytes
- Split the padded M into L blocks



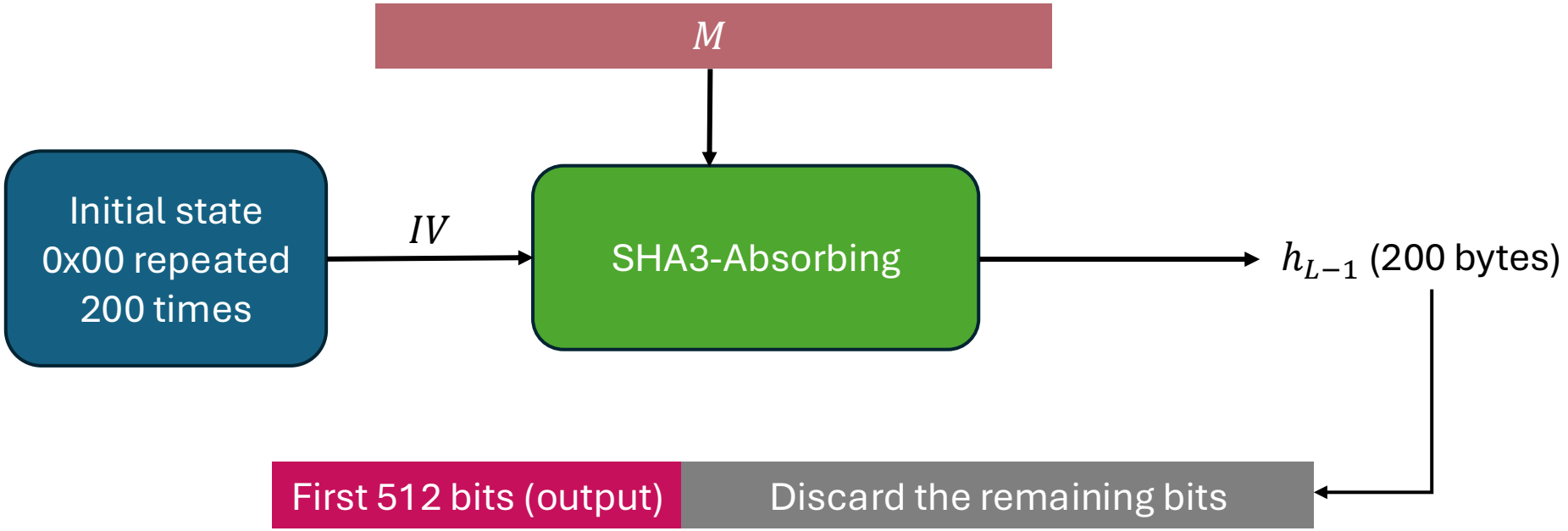
SHA3 Family

- **SHA3-512**



SHA3 Family

- **SHA3-512**



Squeezing

SHA3-512(M) = the first 512 bits (64 bytes) of the final state h_{L-1}

SHAKE

- Traditional hash functions (SHA-256, SHA3-256, …) have **fixed length outputs**

- SHAKE: Extendable-Output Functions (XOFs)
  - It allows variable-length outputs (depending on the application's needs)

SHAKE

- Traditional hash functions (SHA-256, SHA3-256, …) have **fixed length outputs**

- SHAKE: Extendable-Output Functions (XOFs)
  - It allows **variable-length outputs** (depending on the application's needs)
  - SHAKE128, SHAKE256
    - The number at the end is **the target security level**
    - Informally, 128 bit security $\approx O(2^{128})$ operations to break
  - Applications: Generating long pseudorandom-looking bit strings, …

SHAKE

- Traditional hash functions (SHA-256, SHA3-256, …) have **fixed length outputs**

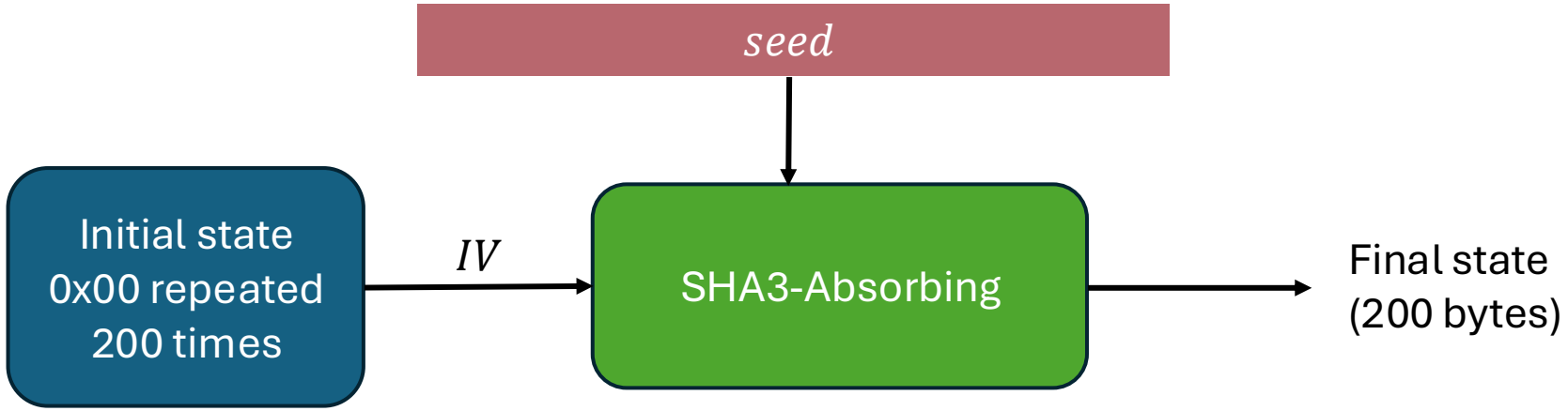
- SHAKE: Extendable-Output Functions (XOFs)
  - It allows **variable-length outputs** (depending on the application's needs)
  - SHAKE128, SHAKE256
    - The number at the end is **the target security level**
    - Informally, 128 bit security $\approx O(2^{128})$ operations to break
  - Applications: Generating long pseudorandom-looking bit strings, …

- Example: Given a 128-bit seed, generate a 256-bit pseudorandom string S

SHAKE256(seed) = S

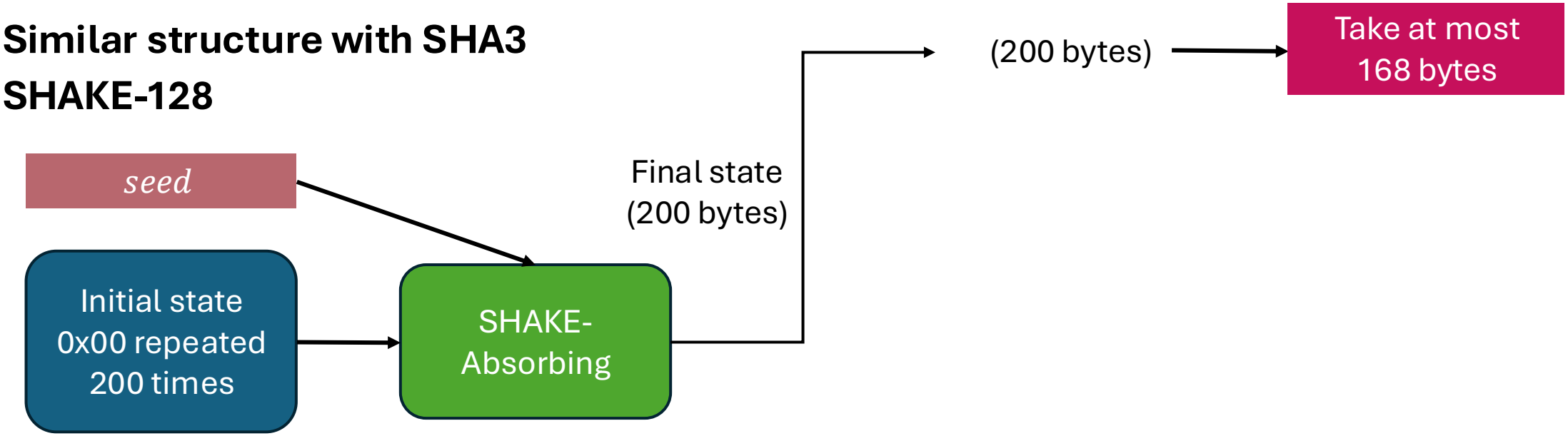
SHAKE

- **Similar structure with SHA3**



SHAKE

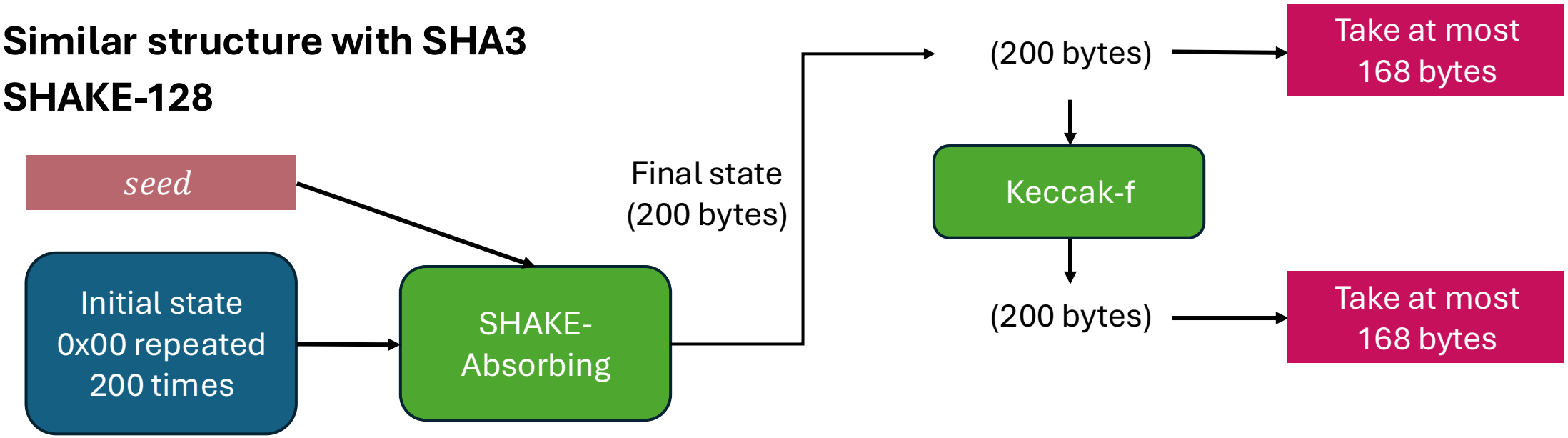
- **Similar structure with SHA3**
- **SHAKE-128**



Squeezing

SHAKE

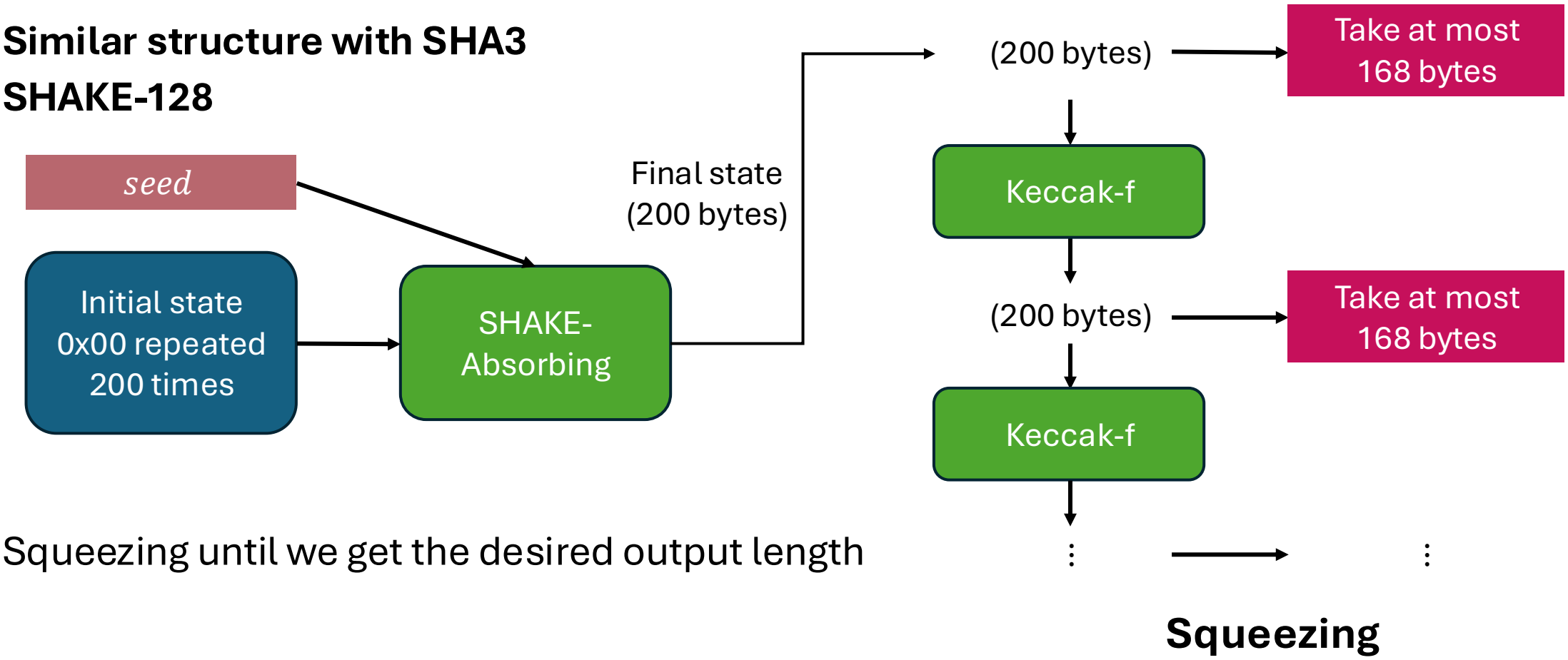
- **Similar structure with SHA3**
- **SHAKE-128**



Squeezing

SHAKE

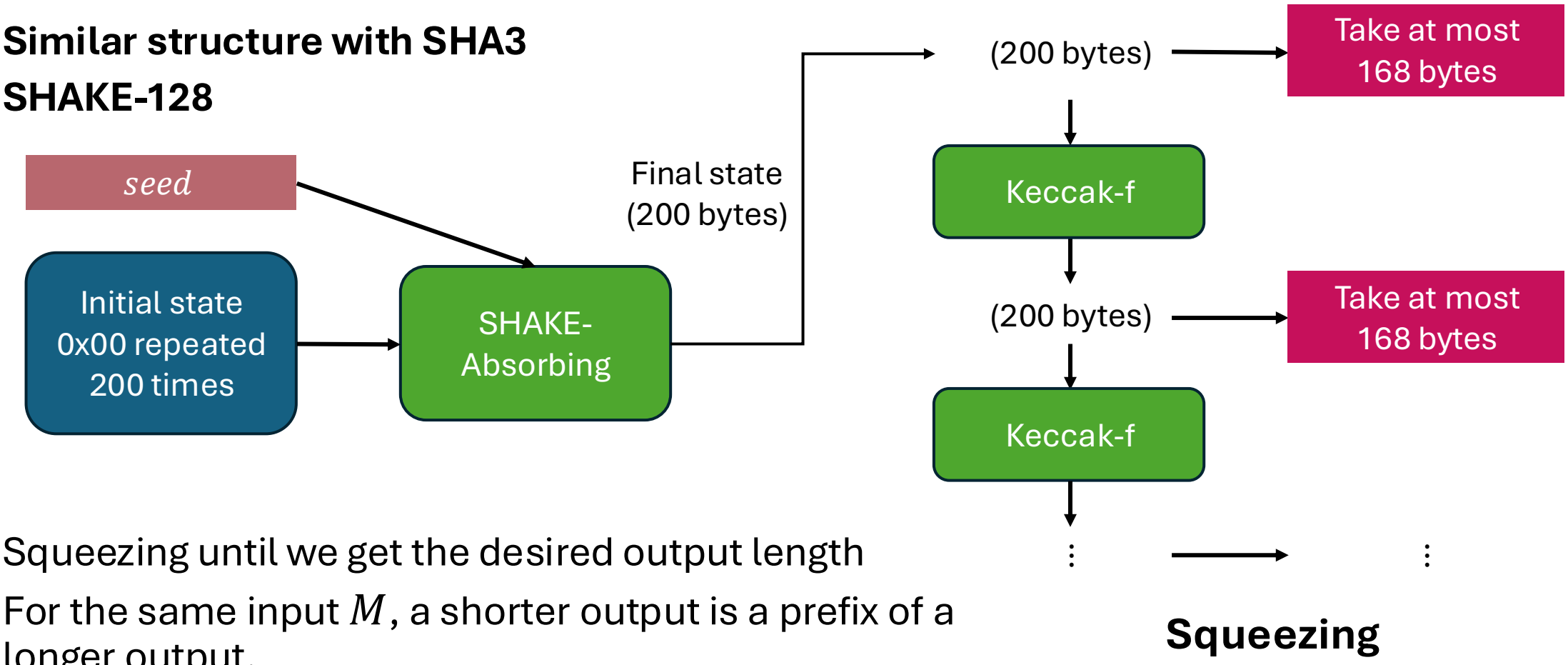
- **Similar structure with SHA3**
- **SHAKE-128**



- Squeezing until we get the desired output length

SHAKE

- **Similar structure with SHA3**
- **SHAKE-128**



- Squeezing until we get the desired output length
- For the same input M, a shorter output is a prefix of a longer output.

API for Hash Functions

- Common APIs for hash functions

```rust
use sha2::{Digest, Sha256};

let mut hasher = Sha256::new();

hasher.update(b"Hello, ");
hasher.update(b"world!");

let hash = hasher.finalize(); // 32-byte digest
```

- update(): Feed message data incrementally (and update the internal state)
- finalize(): Process the final padding and produce the digest

- The incremental hash API reflects the internal structure of SHA-2 and SHA-3
  - Namely, process message blocks as data arrives, then finalize the computation to produce the output.

API for Hash Functions

- Common APIs for SHAKE

```rust
use shake::{Shake128, digest::{Update, ExtendableOutput, XofReader}};

let mut hasher = Shake128::default();

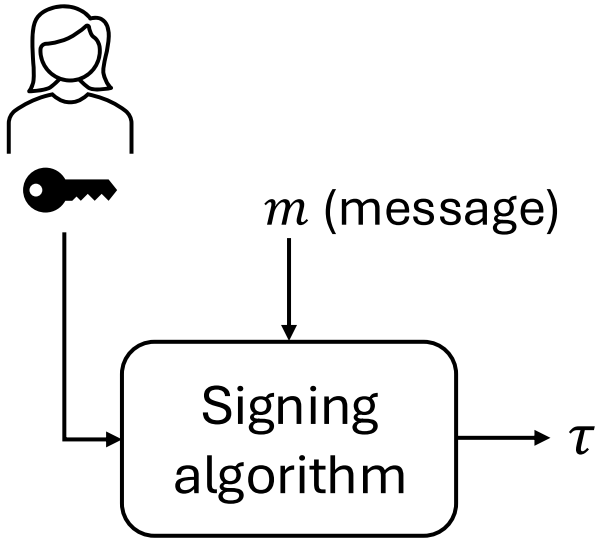
hasher.update(b"Hello, ");
hasher.update(b"world!");

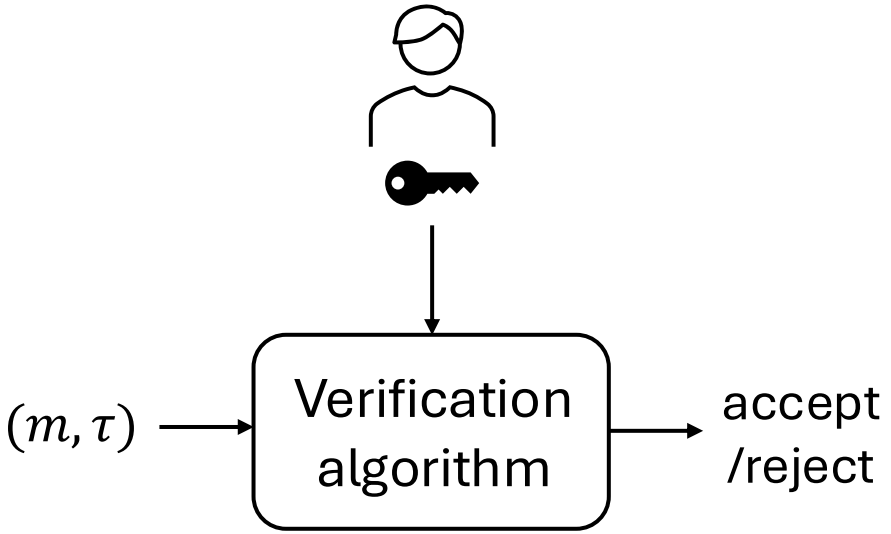
let mut reader = hasher.finalize_xof();

let mut output = [0u8; 100];
reader.read(&mut output); // First 100 bytes
```

Symmetric Primitives - MAC

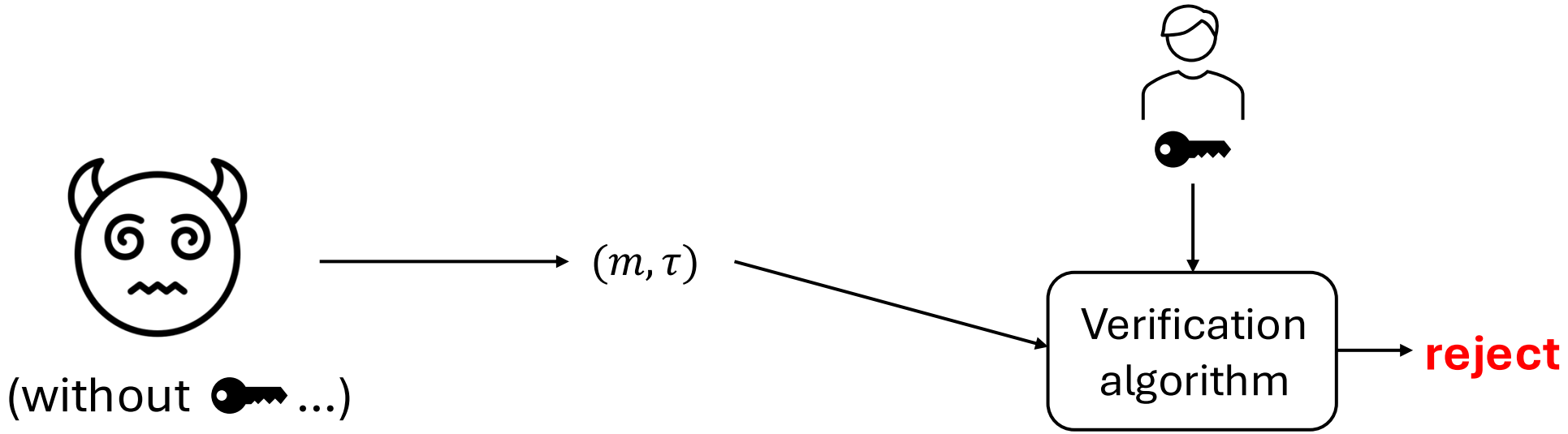
- Message Authentication Code (MAC)
  - **Integrity** (…cannot forge a valid MAC tag without knowing the secret key…)
  - **HMAC** (Hash-based message authentication code)





Symmetric Primitives - MAC

- Message Authentication Code (MAC)
  - **Integrity** (…cannot forge a valid MAC tag without knowing the secret key…)
  - **HMAC** (Hash-based message authentication code)



HMAC

- Let H be a hash function with an output length of L bytes
- Let B be the block size of H
- $opad$ and $ipad$ are two fixed byte strings
  - RFC 2104: $ipad = 0\text{x}36$ repeated B times, $opad = 0\text{x}5\text{c}$ repeated B times **(so they are B-byte strings)**

$$\text{HMAC}(K, m) = H\left((K' \oplus opad) \parallel H\left((K' \oplus ipad) \parallel m\right)\right)$$

$$K' = \begin{cases} K \parallel 0\text{x}00^{B-|K|_{\text{byte}}} & \text{If } |K|_{\text{byte}} \leq B\text{, then pad it with zeros to } B \text{ bytes} \\ H(K) \parallel 0\text{x}00^{B-L} & \text{Otherwise } (|K|_{\text{byte}} > B)\text{, hash the key to } L \text{ bytes, and pad it (with zeros) to } B \text{ bytes again} \end{cases}$$

HMAC-SHA256

- Let H = SHA256, with an output length of **32 bytes**
- The block size of SHA256 is **64 bytes**
- $opad$ and $ipad$ are two fixed 64-byte strings
  - RFC 2104: $ipad = 0\text{x}36$ repeated 64 times, $opad = 0\text{x}5\text{c}$ repeated 64 times

$$\text{HMAC–SHA256}(K, m) = H\left((K' \oplus opad) \parallel H\left((K' \oplus ipad) \parallel m\right)\right)$$

$$K' = \begin{cases} K \parallel 0\text{x}00^{64-|K|_{\text{byte}}} & \text{If } |K|_{\text{byte}} \leq 64\text{, then pad it with zeros to 64 bytes} \\ H(K) \parallel 0\text{x}00^{32} & \text{Otherwise } (|K|_{\text{byte}} > 64)\text{, hash the key to 32 bytes, and pad it (with zeros) to 64 bytes again} \end{cases}$$

HMAC

- You need the secret key to verify a MAC code

- **HMAC is *canonical*: Verification = Recompute-and-compare**
  - Generate HMAC code t of a message m using K : $t = \mathrm{HMAC}(K, m)$
  - Verify HMAC: Given K and (m, t), check $t =_? \mathrm{HMAC}(K, m)$
  - Cryptographic libraries typically provide a dedicated verification API to ensure secure tag comparison **(so, use the API!).**
    - For example: mac.verify_slice(tag).is_ok()

HMAC

- You need the secret key to verify a MAC code

- **HMAC is *canonical*: Verification = Recompute-and-compare**
  - Generate HMAC code t of a message m using K : $t = \mathrm{HMAC}(K, m)$
  - Verify HMAC: Given K and (m, t), check $t =_? \mathrm{HMAC}(K, m)$
  - Cryptographic libraries typically provide a dedicated verification API to ensure secure tag comparison **(so, use the API!).**
    - For example: mac.verify_slice(tag).is_ok()

- Some security properties:
  - Modify m or t ⇒ Rejected by the verification
  - Modify K ⇒ Subtler, but for fix-length K, also cause rejection

Symmetric Primitives - KDF

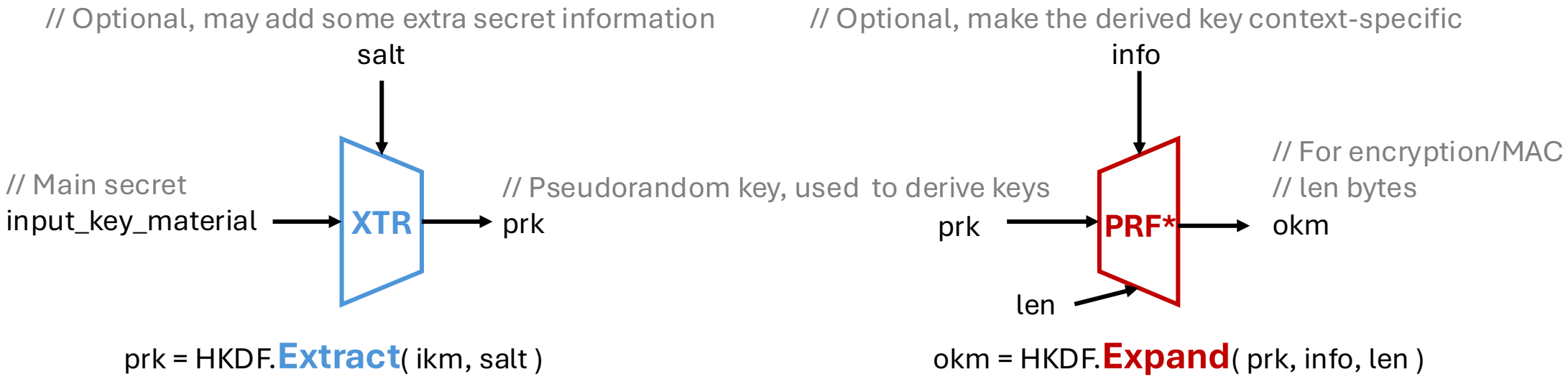
- Key Derivation Function (KDF)

  KDF(“…shared secret with randomness…”) = a symmetric key

  - Used to derive a key for symmetric key encryption
  - HKDF (based on HMAC)
  - Derive keys of arbitrary lengths

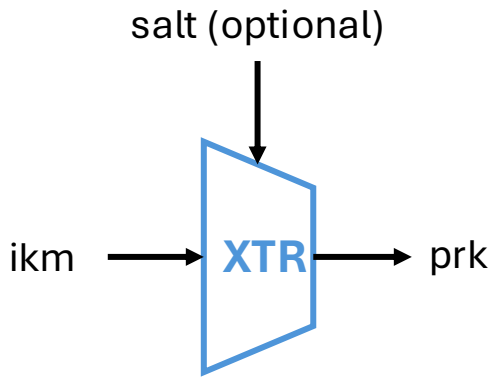
HKDF

- HKDF: HMAC-based Key Derivation Function
  - Two interfaces: **Extract (XTR)** and **Expand (PRF*)**



HKDF.Extract

- HKDF.Extract
  - ikm (input key material)
    - The **source** of secret
    - **Must remain secret**
  - salt
    - Optional, preferably random or pseudorandom
    - **Could be public**
    - Defaults to a string of zeros (with the hash output length) if omitted
  - prk (pseudorandom key)
    - A fixed-length pseudorandom key for HKDF.Expand



HKDF.Extract

- HKDF.**Extract** formula: prk = HKDF.**Extract**(ikm, salt) is actually

$$prk = HMAC(K = salt, m = ikm)$$

- So HKDF.Extract just uses…
  - the salt as the HMAC key
  - the ikm as the HMAC message
  - the HMAC tag as the output prk

HKDF.Extract

- (Somehow misleading) intuition of "Extract"...

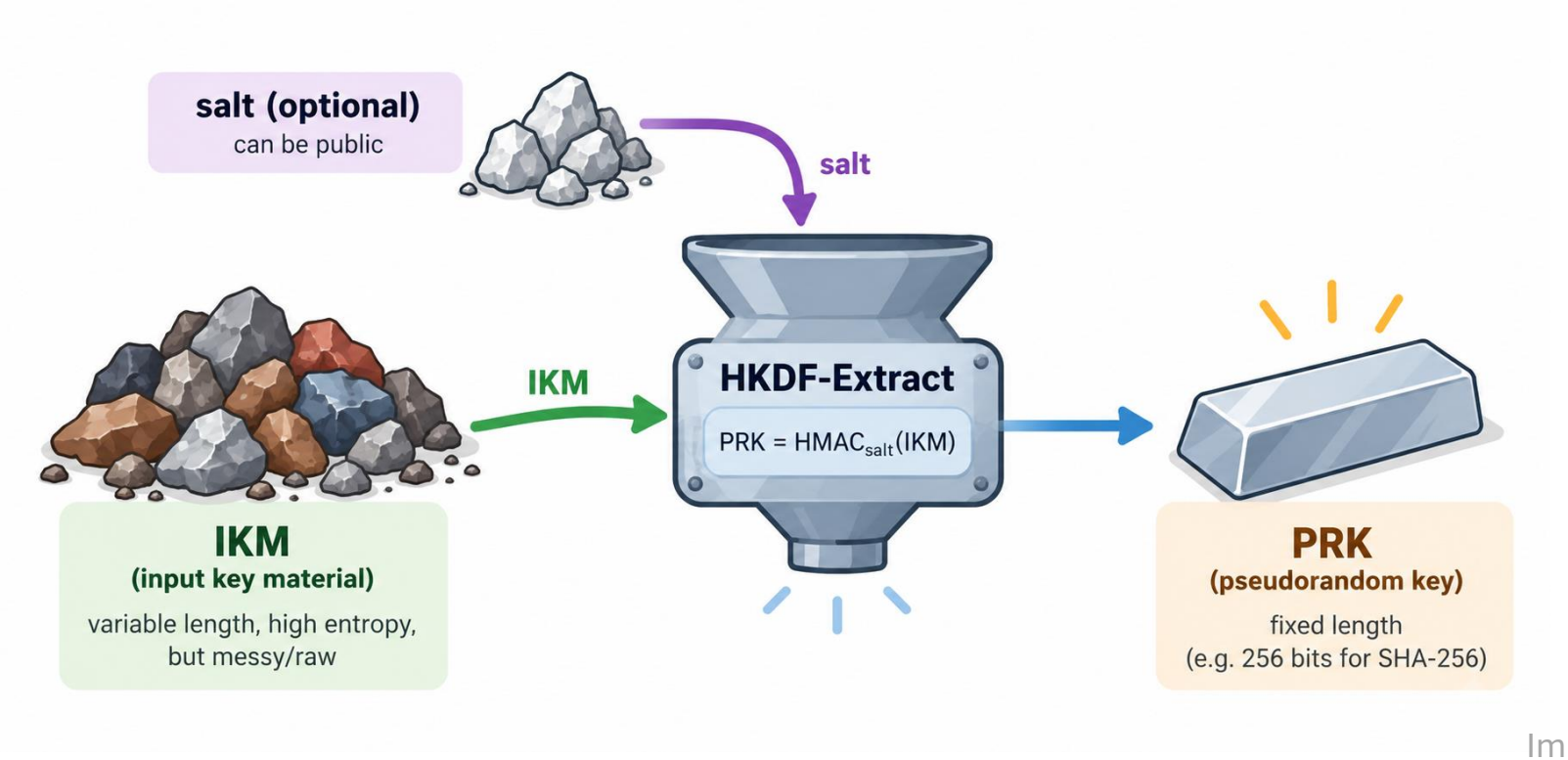


Image from ChatGPT-6 Sol

HKDF.Extract

- HKDF.**Extract** formula: prk = HKDF.**Extract**(ikm, salt) is actually

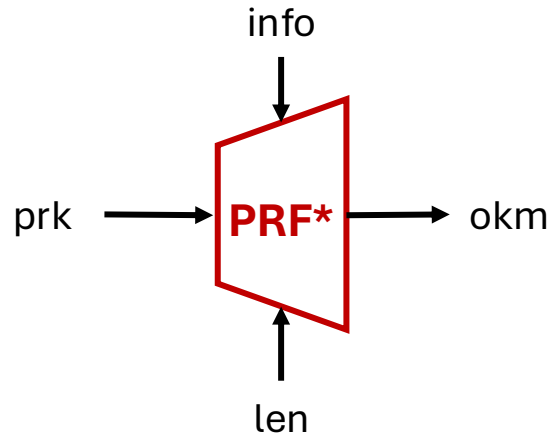
$$prk = HMAC(K = salt, m = ikm)$$

- So HKDF.Extract just uses…
  - the salt as the HMAC key
  - the ikm as the HMAC message
  - the HMAC tag as the output prk

- **Why not use ikm as the HMAC key?** Please see after-class exercises.

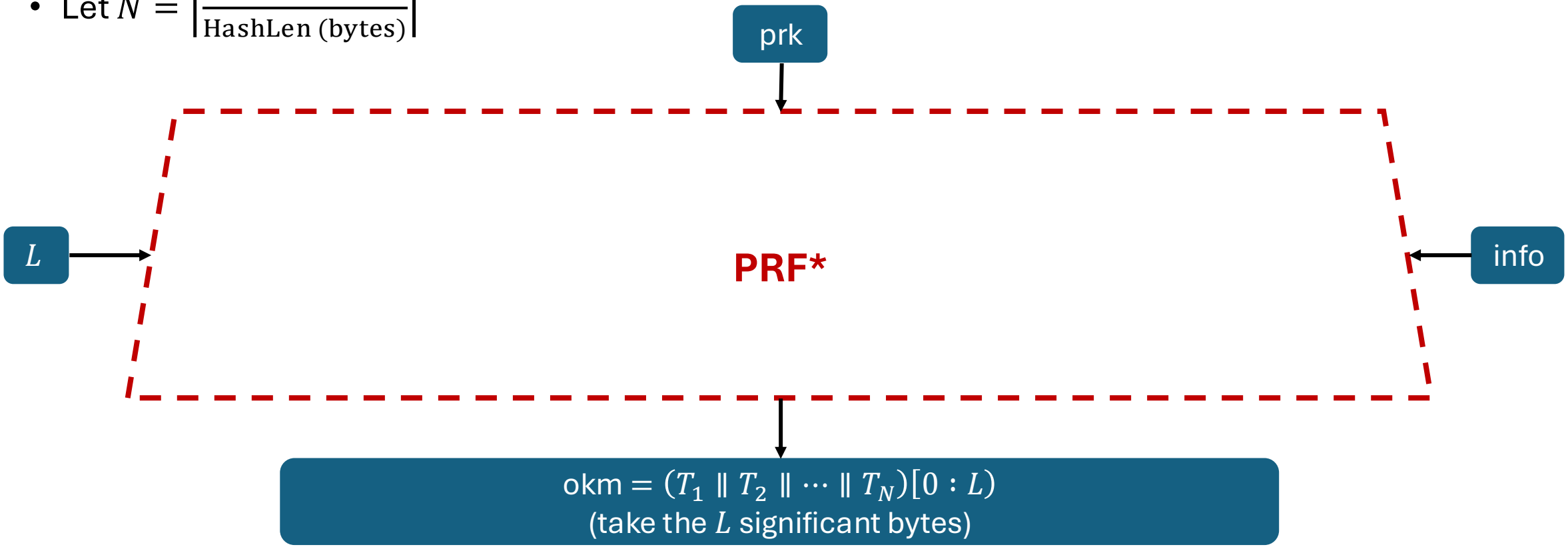
HKDF.Expand

- HKDF.**Expand:** Essentially a variable-output-length (denoted by *) pseudorandom function (PRF)
  - prk
    - The fixed-length pseudorandom key from HKDF.**Extract**.
    - **Must remain secret**
  - info
    - Optional, application-specific context and label string
    - Enables **domain separation** between different derived keys.
  - len
    - The desired length of the output key material
  - okm (output keying material):
    - Key material for cryptographic purposes.



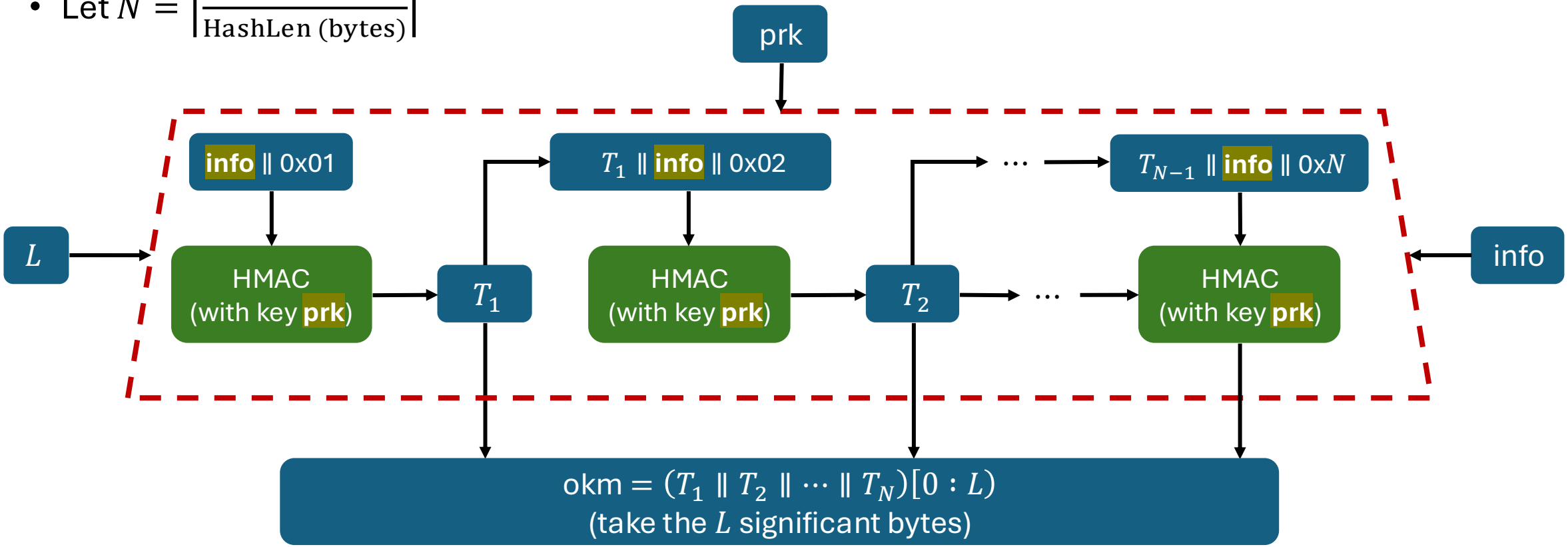
HKDF.Expand

- Let L = len (number of bytes)
- Let $N = \left\lceil \frac{L}{\text{HashLen (bytes)}} \right\rceil$



HKDF.Expand

- Let L = len (number of bytes)
- Let $N = \left\lceil \frac{L}{\text{HashLen (bytes)}} \right\rceil$



HKDF.Expand

- Let L = len (number of bytes)
- Let $N = \left\lceil \frac{L}{\text{HashLen (bytes)}} \right\rceil$

- $T_1 = \text{HMAC}(K = \text{prk}, m = \text{info} \parallel 0\text{x}01)$
- $T_2 = \text{HMAC}(K = \text{prk}, m = T_1 \parallel \text{info} \parallel 0\text{x}02)$
- …
- $T_N = \text{HMAC}(K = \text{prk}, m = T_{N-1} \parallel \text{info} \parallel 0\text{x}N)$

- okm = $(T_1 \parallel T_2 \parallel \cdots \parallel T_N)[0 : L)$ // Take the first L bytes

Using HKDF

- Suppose that we have some secret g^{xy}.
- prk = HKDF.Extract(salt = "0^{256}", ikm = bytes of g^{xy}) // Extract prk from g^{xy}, using zero salt
- K_1 = HKDF.Expand(prk = prk, info = "My 1st key", len = 32) // Expand a 256-bit key
- K_2 = HKDF.Expand(prk = prk, info = "My 2nd key", len = 64) // Expand a 512-bit key
- K_3 = HKDF.Expand(prk = prk, info = "My 3rd key", len = 32) // Expand a 256-bit key
- $K_4 \parallel K_5$ = HKDF.Expand(prk = prk, info = "My 4th and 5th keys", len = 64) // Expand two 256-bit keys

Using HKDF

- Suppose that we have some secret g^{xy}.
- prk = HKDF.Extract(salt = "0^{256}", ikm = bytes of g^{xy}) // Extract prk from g^{xy}, using zero salt
- $K_4 \parallel K_5$ = HKDF.Expand(prk = prk, info = "My 4th and 5th keys", len = 64) // Expand two 256-bit keys

- A nice property of HKDF: Knowing K_4 does not reveal K_5!
  - The two derived keys are computationally indistinguishable from two independent random keys.

Using HKDF

- **// Wrong use!**
- // K_1 and K_2 share the same prefix
- K_1 = HKDF.Expand(prk = prk, info = “LABEL_123”, len = 32)
- K_2 = HKDF.Expand(prk = prk, info = “LABEL_123”, len = 64)

- **Exercise: Explain why**

- Lesson: Use different **info** values when deriving keys for different purposes!
  - The output length does **not** provide domain separation

Next Topics

- SKE, AE, AEAD
- Padding
- Key strength
- Summary of symmetric-key primitives
- Introduction to public-key primitives

- **Reminder:**
  - The first two hours (with a 10-min break) on November 5 will be devoted to lectures

Code Review

- https://github.com/RunzhiZeng/Cryptography-Engineering-WiSe26-27/tree/main/L1

Exercise

- Practice your Rust skill

1. **Read the example code of XOF (SHAKE-128, SHAKE-256)**
2. **HMAC-SHA256:** Implement HMAC-SHA256 based on SHA256
3. **HKDF-SHA256:** Implement HKDF-SHA256 based on SHA256

Exercise

- Deepen your understanding of cryptography

1. HMAC is more than a MAC (code exercise):

- Generate 1,000 different IKMs, each consisting of **16 random bytes followed by 16 zero bytes.**
- For each IKM, compute HMAC-SHA256(salt, IKM) using a fixed zero salt.
- Compare the input and output bit distributions:
  - Which input bit positions are always zero?
  - Do any output bit positions appear fixed?
  - Is the proportion of 0s and 1s in the outputs approximately equal?

2. According to the structure of HKDF-Expand, explain why two calls to HKDF-Expand with the same (prk, info) input but different length inputs will produce two keys with same prefix.

Further Reading

- Rust book: https://doc.rust-lang.org/book/title-page.html
- Some common programming concepts in Rust (variables, functions, etc)
  - https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html
- Error handling in Rust (e.g. Result<T, E>, “?”, …)
  - https://doc.rust-lang.org/book/ch09-00-error-handling.html