CS405 – Computer Security

Lab05 – Hash Functions

Content

Content
Introduction
Secure Hash Functions
Building Hash Functions
The SHA Family of Hash Functions
The BLAKE2 Hash Function
Attacks
Code Demos

Introduction

- Hash functions are found everywhere:
  - Digital signatures,
  - public-key encryption,
  - integrity verification,
  - message authentication,
  - password protection,
  - key agreement protocols
  - Identifying identical and modified files
  - Git systems to identify files in repositories
  - Intrusion Detection Systems
  - Forensics analysis to prove that digital artifacts have not been modified
  - Blockchains

Introduction

- Hash functions take a long input and produce a short output.
  - The output is called a *hash value* or *digest*

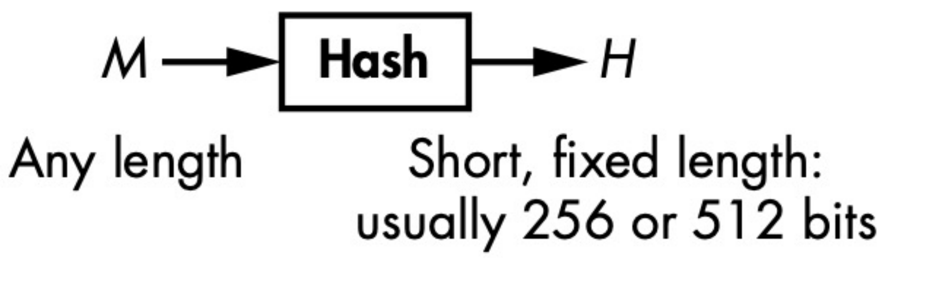
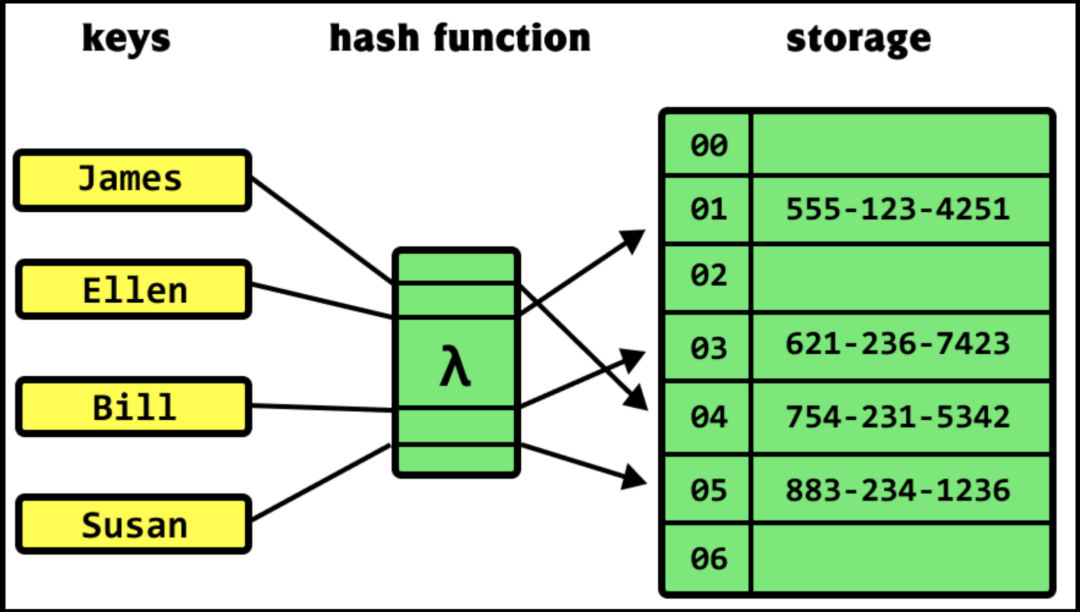
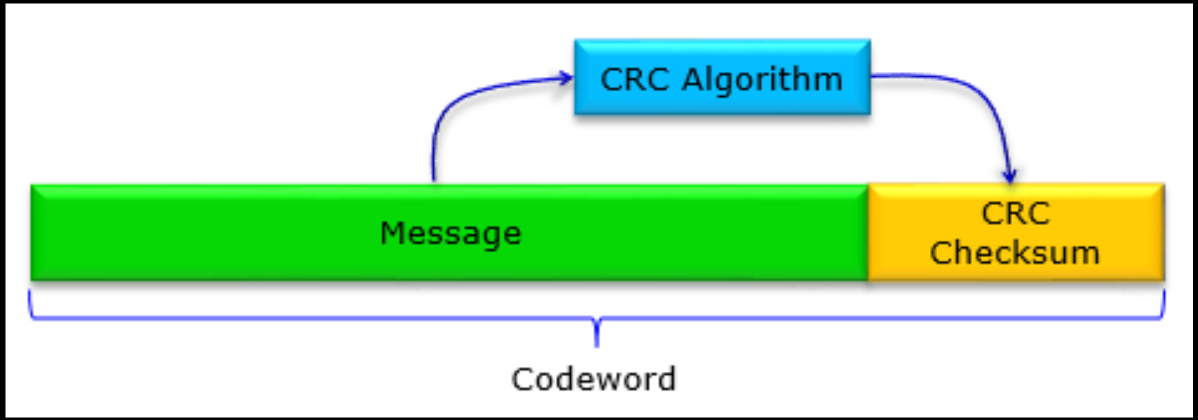


Figure 6-1: A hash function's input and output

Introduction

- Non-crypto hash functions:
  - Used in data structures. E.g., hash tables
  - Used in detecting accidental errors. E.g., cyclic redundancy check (CRC)
  - Not secure.





Content

Content
Introduction
Secure Hash Functions
Building Hash Functions
The SHA Family of Hash Functions
The BLAKE2 Hash Function
Attacks
Code Demos

Secure Hash Functions

- Hash functions protect data integrity.
  - Ensures that the data has not been modified.
  - The data can be clear or encrypted.

- Secure hash function = unique hash value for each input.
  - A hash value is like the fingerprint.

- Example:
  - $hash(0101010) = XYZ$
  - $hash(0101011) = ABC$

Secure Hash Functions

- In digital signatures, applications signs the hash of a message.

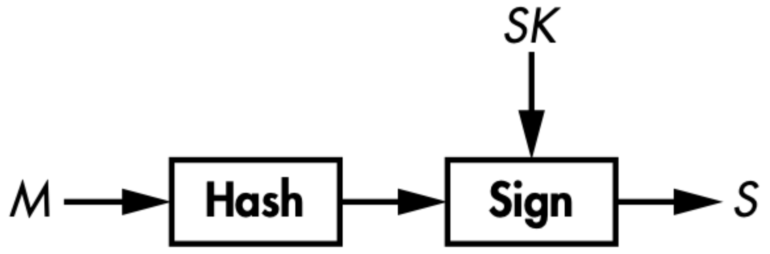


Figure 6-2: A hash function in a digital signature scheme. The hash acts as a proxy for the message.

- Signing a message's hash is as secure as signing the message itself.
- Signing a short hash is faster than signing a large message.

Secure Hash Functions

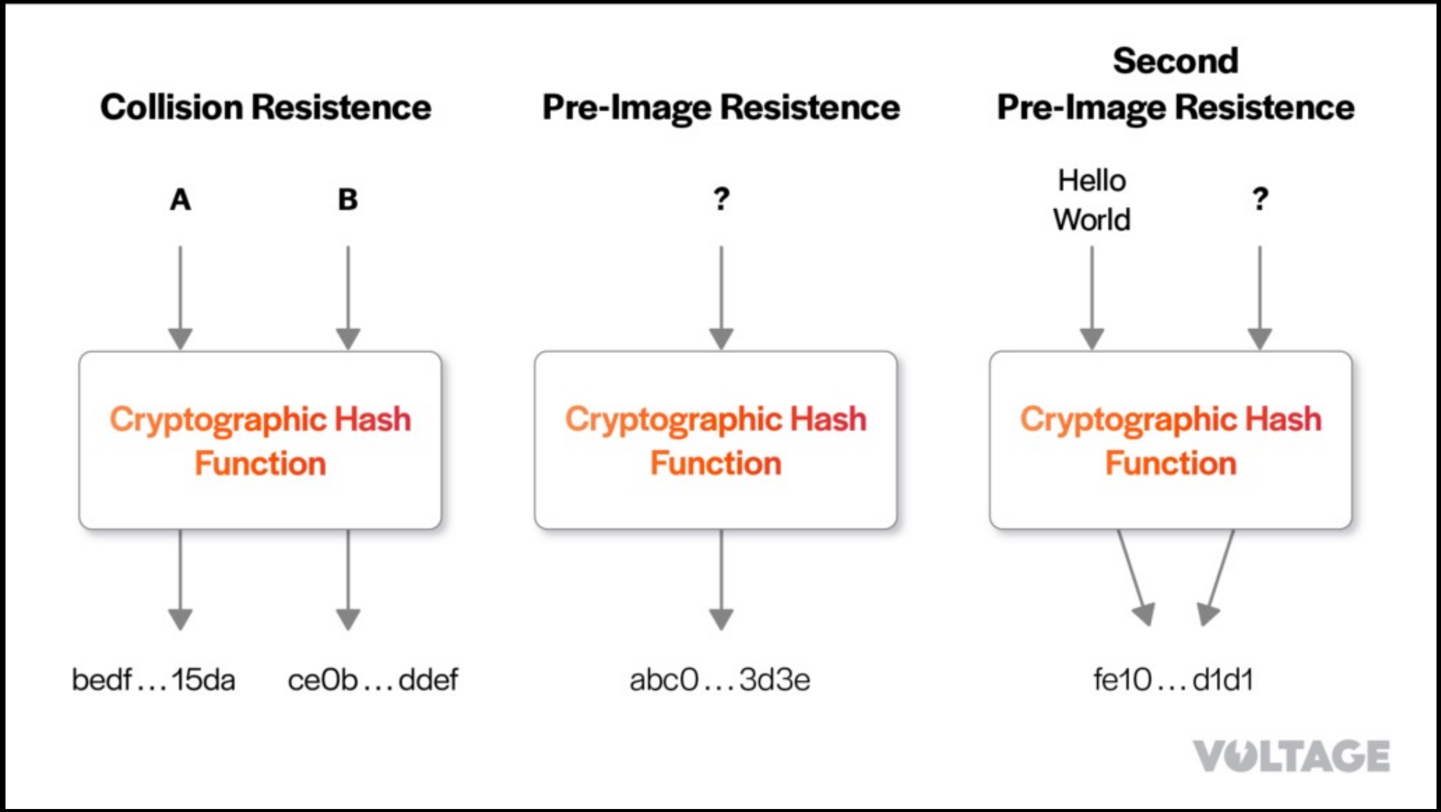
- The output of hash function should be unpredictable.

```
SHA-256("a") = 87428fc522803d31065e7bce3cf03fe475096631e5e07bbd7a0fde60c4cf25c7
SHA-256("b") = a63d8014dba891345b30174df2b2a57efbb65b4f9f09b98f245d1b3192277ece
SHA-256("c") = edeaaff3f1774ad2888673770c6d64097e391bc362d7d6fb34982ddf0efd18cb
```

- If you know the hash of “a”, “b”, and “c”, you cannot predict the hash of ”d”.
- Secure hash functions are PRFs.

Secure Hash Functions

- Notions to define secure hash functions:
  - Hash functions are one-way functions.



Secure Hash Functions

Pre-image resistance

- It means that a hash function cannot be inverted.

- Given unlimited computation power, you can’t find the pre-image of a hash.
    - There are infinite number of pre-images.

Secure Hash Functions

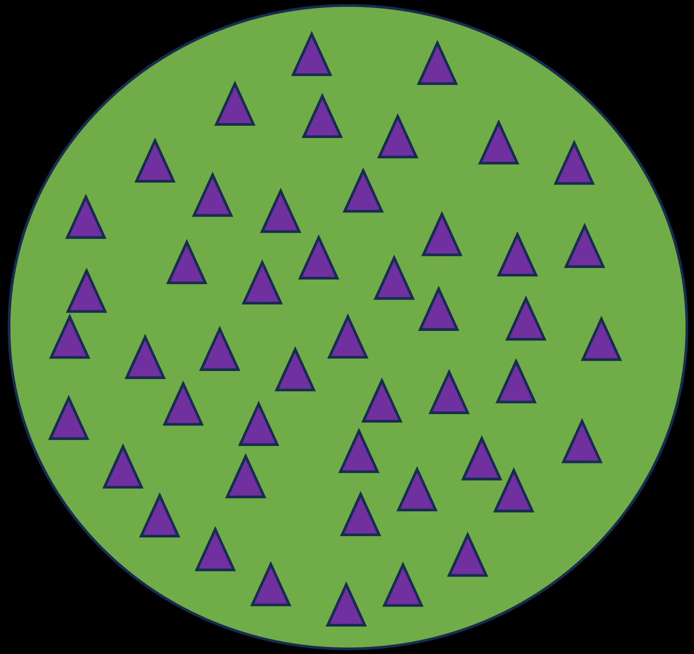
Pre-image resistance

- Given a hash function that produces 256-bit hash.
  - Then, the set of all possible hashes include 2^{256} hashes.

Secure Hash Functions

Pre-image resistance

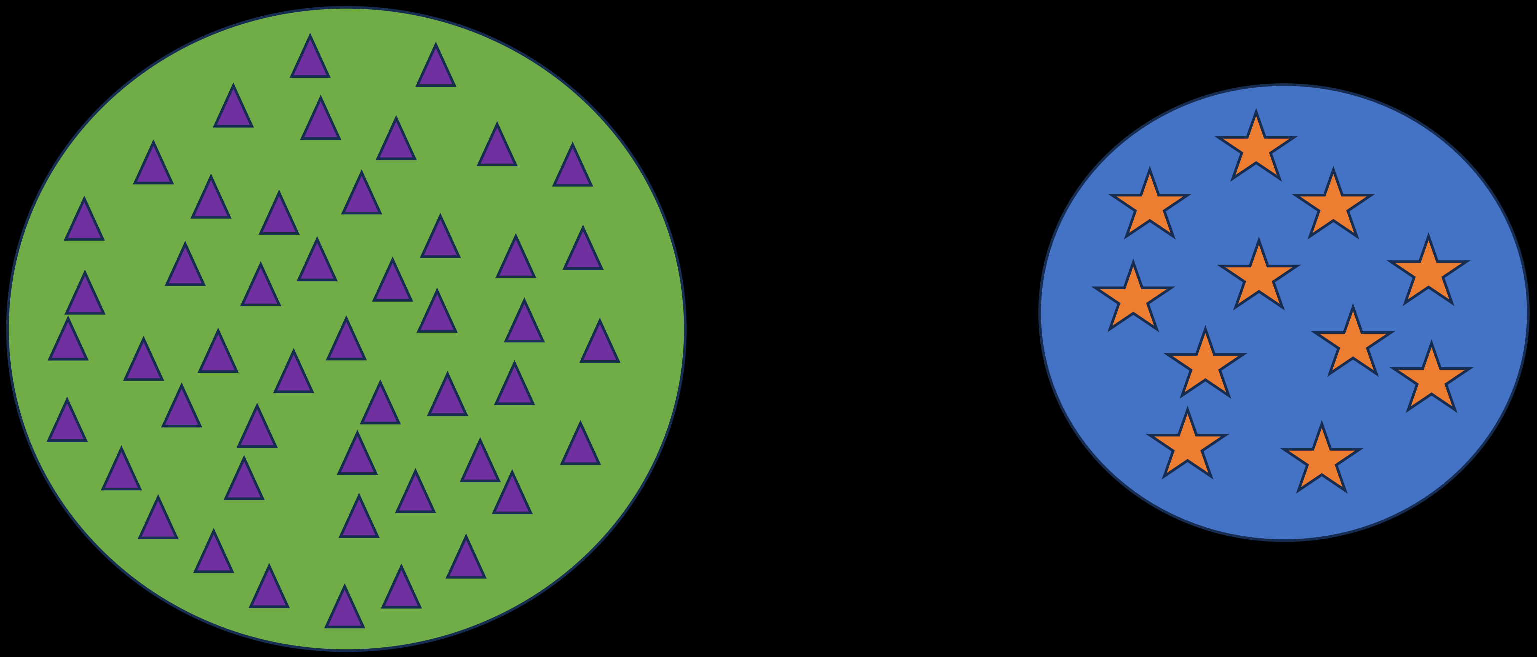
- Given a set of messages, where each is 1024-bit.
  - Then, the set of all possible messages include 2^{1024} messages.



Secure Hash Functions

Pre-image resistance

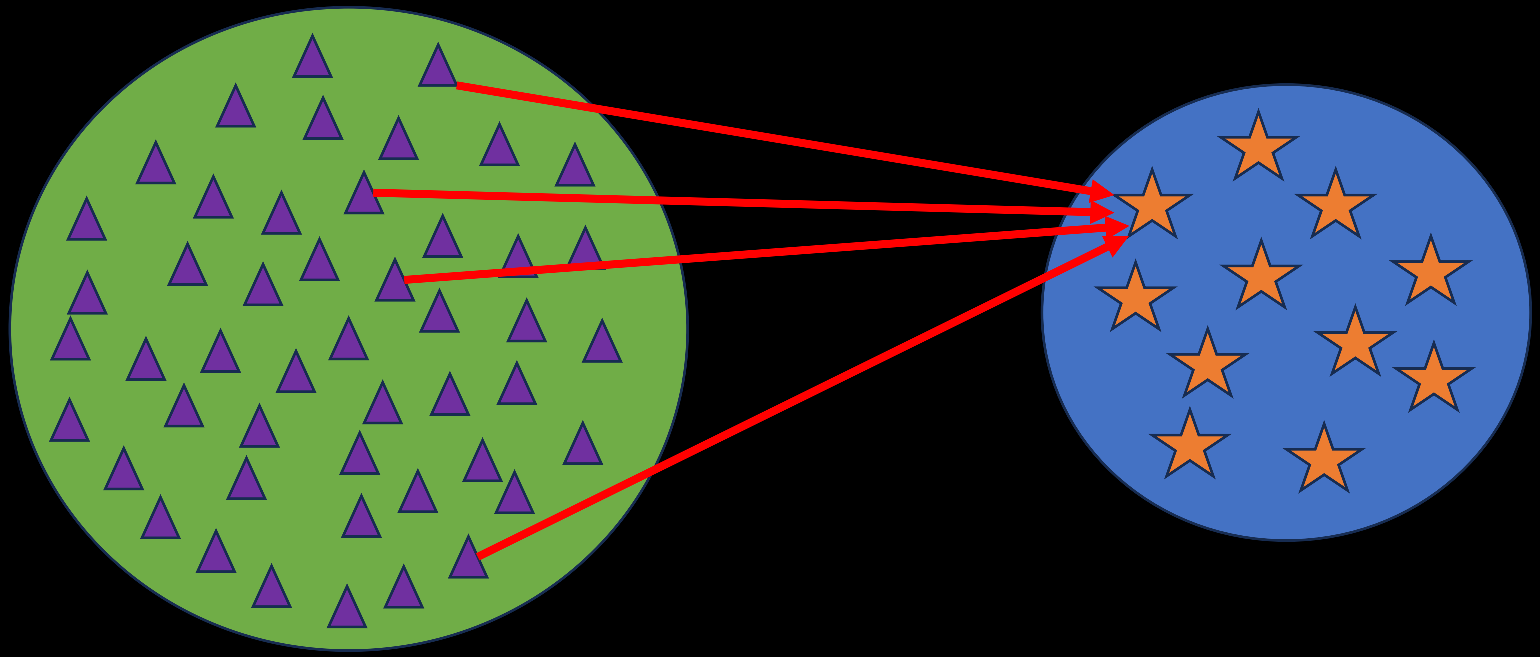
- The message space (2^{1024}) $>$ hash space (2^{256}).



Secure Hash Functions

Pre-image resistance

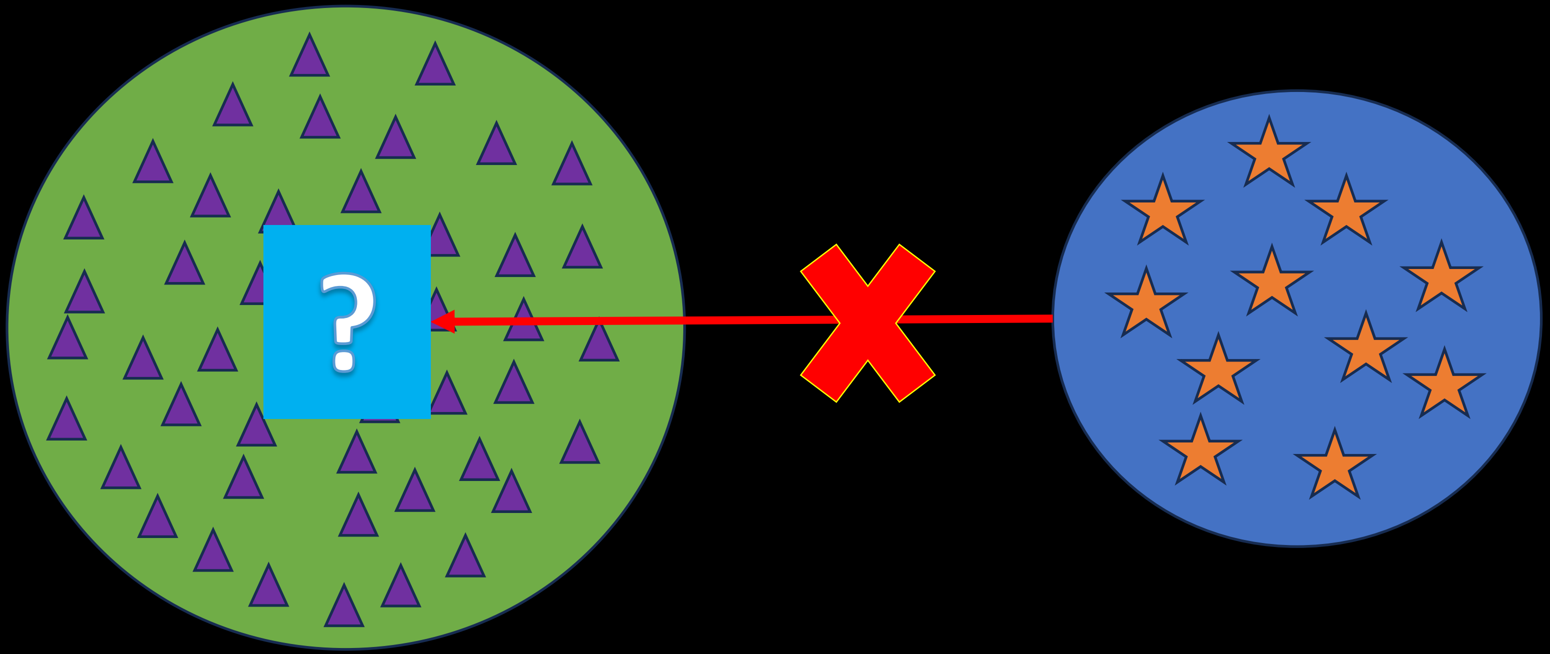
- So, each possible hash can have $2^{1024}/2^{256} = 2^{768}$ pre-images.
    - So, which of the 2^{768} was the actual message I hashed?



Secure Hash Functions

Pre-image resistance

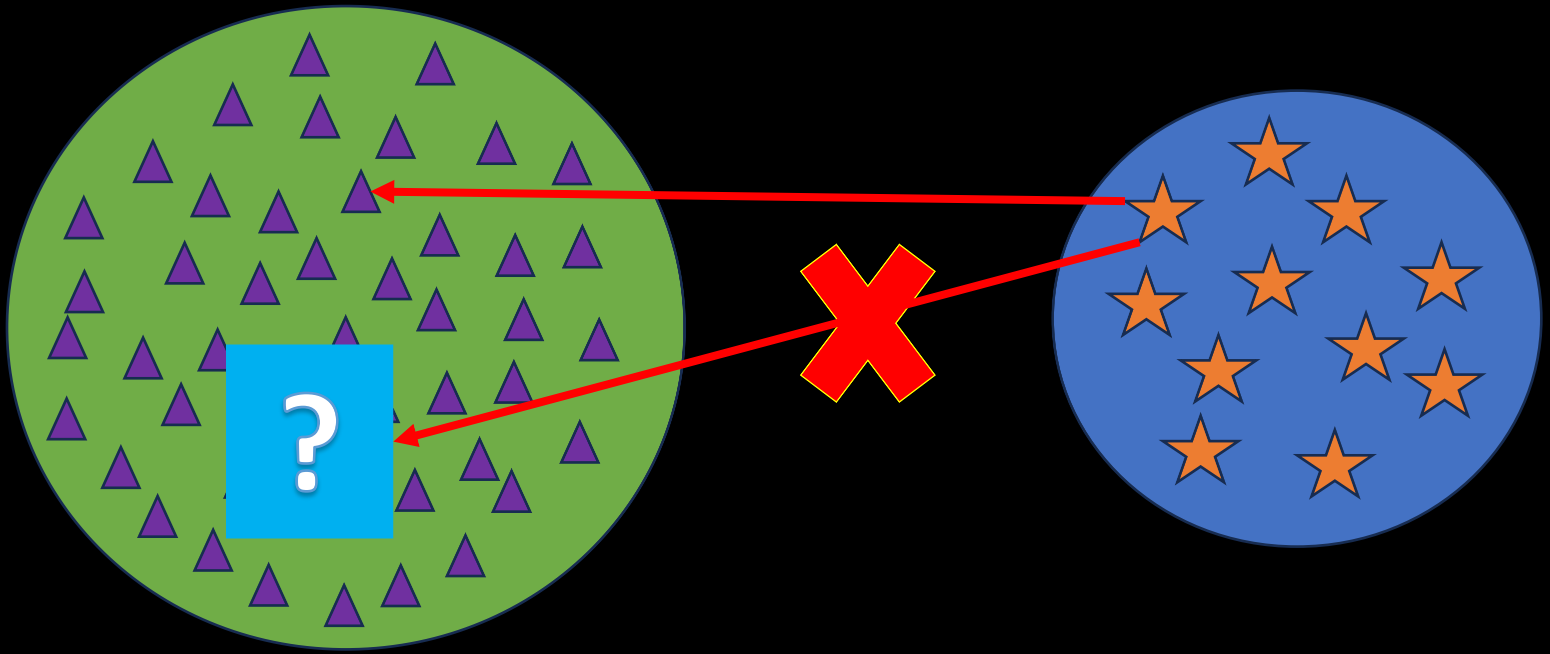
- Pre-images resistance: practically impossible to find a message that hashes to a given value.



Secure Hash Functions

Second pre-image resistance

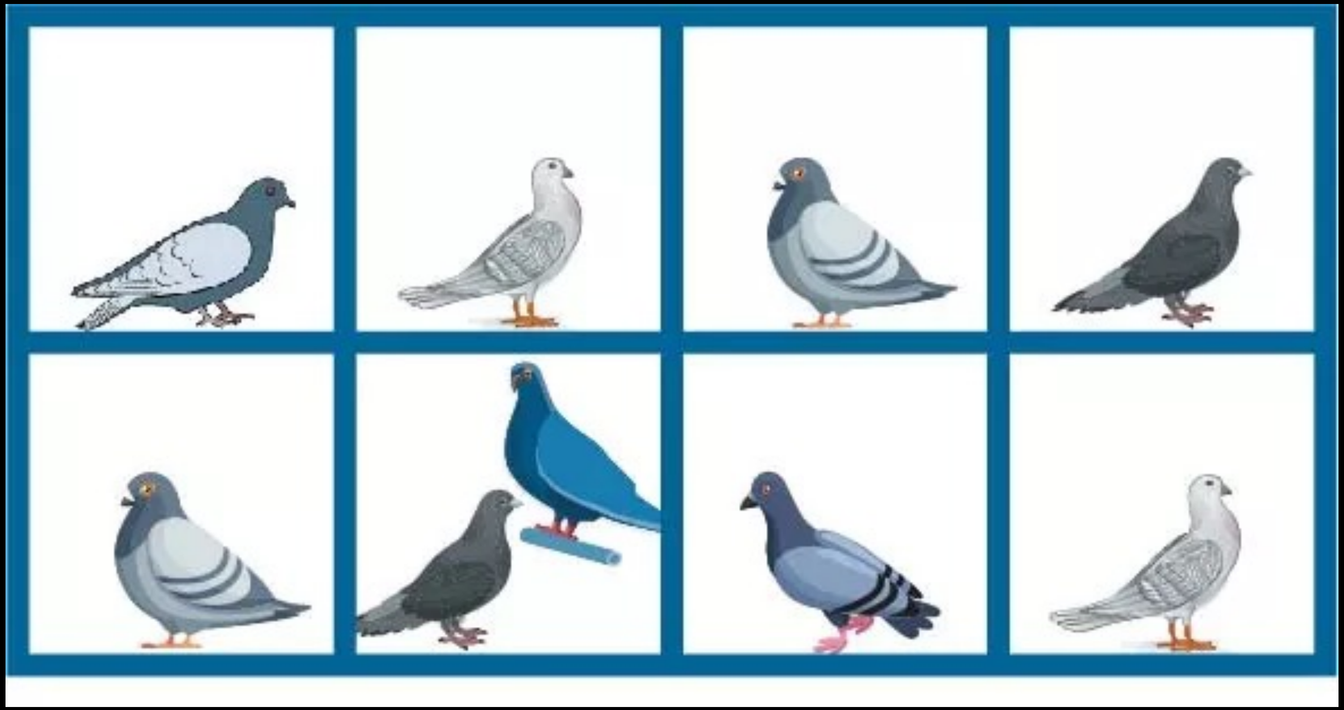
- Given a message, M_1, it's practically impossible to find another message, M_2, that hashes to the same value that M_1 does.



Secure Hash Functions

Collision resistance

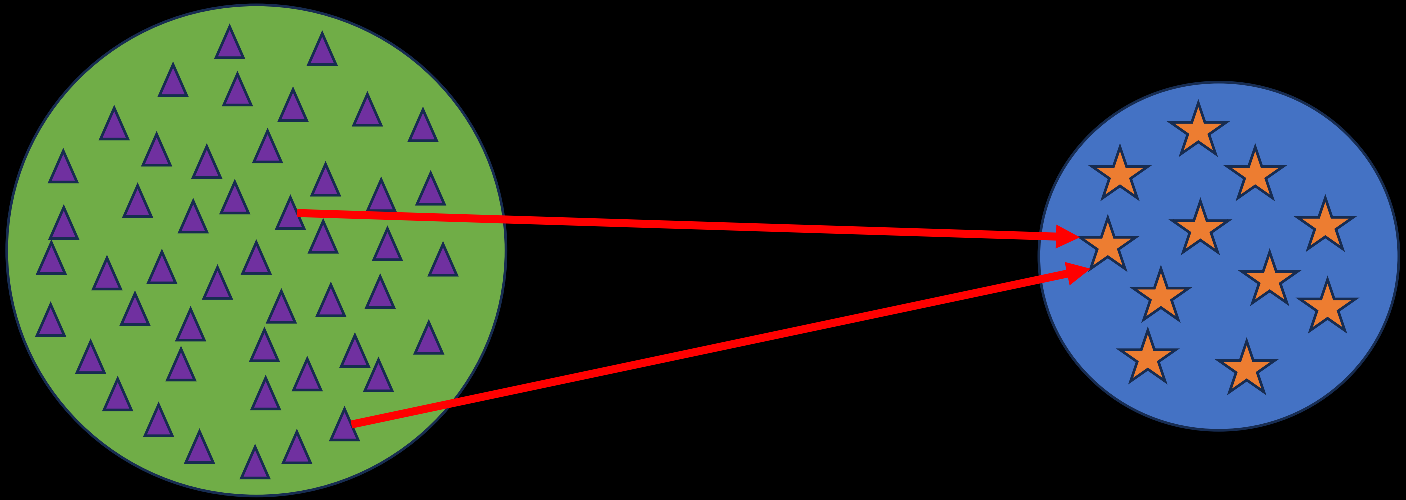
- Collisions will inevitably exist due to the *pigeonhole principle*



Secure Hash Functions

Collision resistance

- Practically impossible to find distinct messages that hash to the same value.
- Find second preimages = find collisions.
- Any collision-resistant hash is also second preimage resistant.

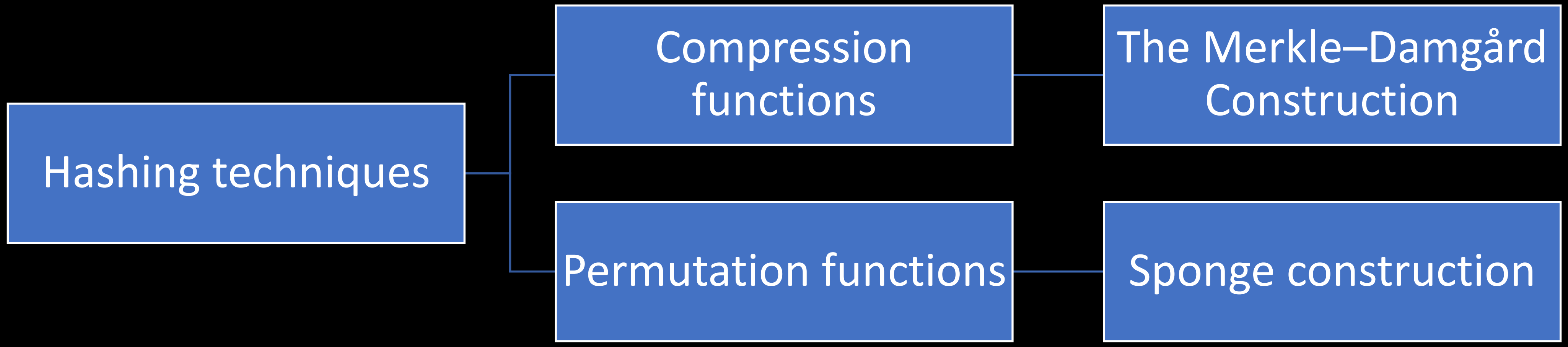


Content

Content
Introduction
Secure Hash Functions
Building Hash Functions
The SHA Family of Hash Functions
The BLAKE2 Hash Function
Attacks
Code Demos

Building Hash Functions

- Iterative hashing: split the message into chunks and process each chunk consecutively.



Building Hash Functions

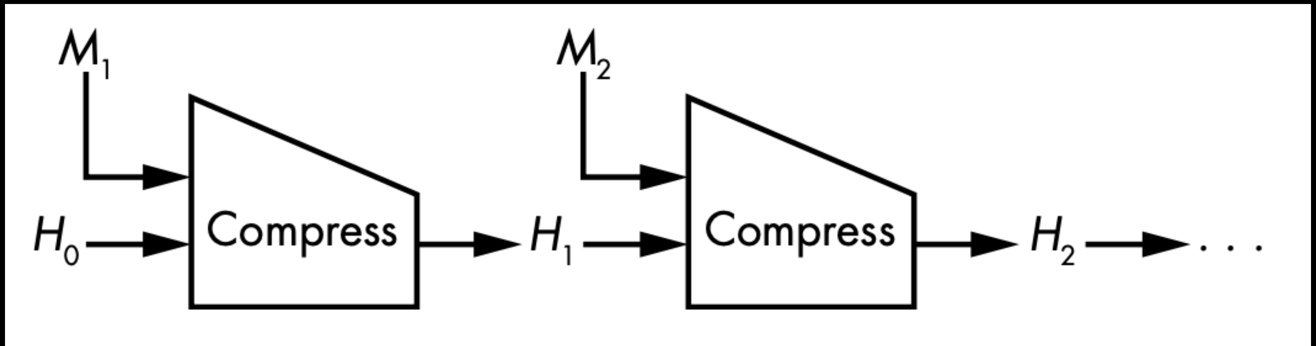
Compression-Based Hash Functions: The Merkle–Damgård Construction

- It uses compression functions to transform the input into smaller output.
- Examples:
  - MD4, MD5
  - SHA-1 and SHA-2 family
  - RIPEMD
  - Whirlpool
- The M–D construction:
  - Not perfect,
  - Simple and secure enough.

Building Hash Functions

Compression-Based Hash Functions: The Merkle–Damgård Construction

- Splits the message into blocks of identical size and mixes these blocks with an internal state using a compression function.
  - Blocks can be 512-bit (as in SHA-256) or 1024-bit (as in SHA-512)



  - H_0 is an initial value (IV)
  - H_1, H_2, … are called the chaining values
  - The last H value is message's hash.

Building Hash Functions

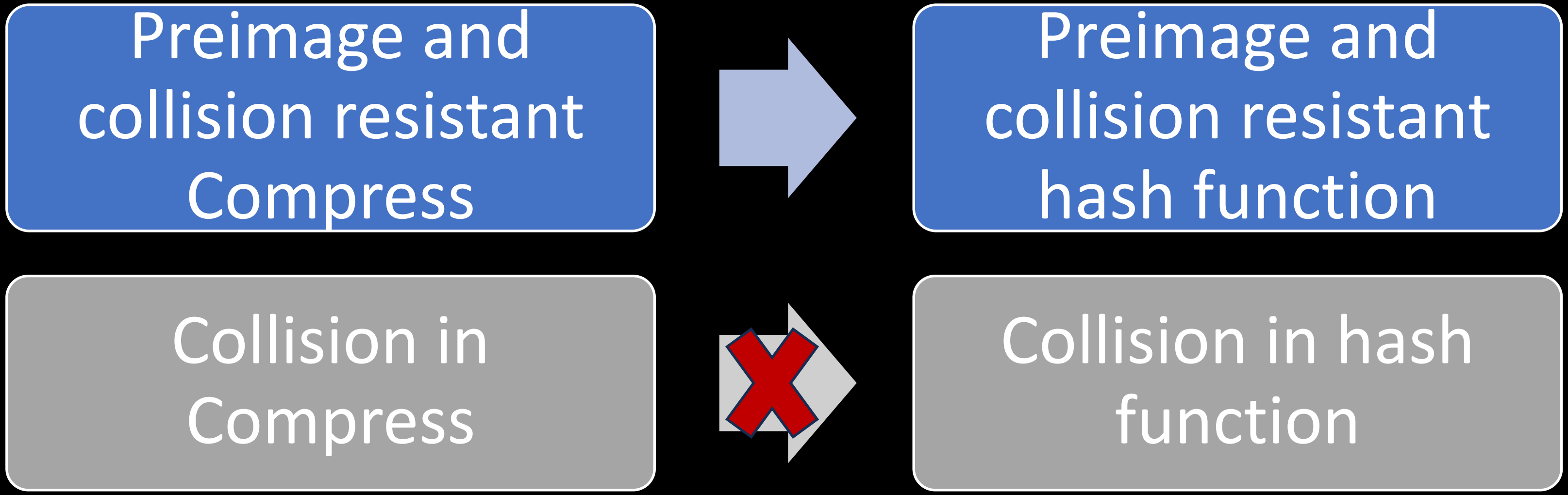
Compression-Based Hash Functions: The Merkle–Damgård Construction

- What if the message is not aligned to the block size?
- Pad it:
  - Take the remaining bits.
  - Append 1 bit.
  - Append 0 bits.
  - Append the length of the original message.

- Example, given the 8-bit message “10101010”. Pad into 512-bit block:
  101010101(0000.....000)1000

Building Hash Functions

Compression-Based Hash Functions: The Merkle–Damgård Construction



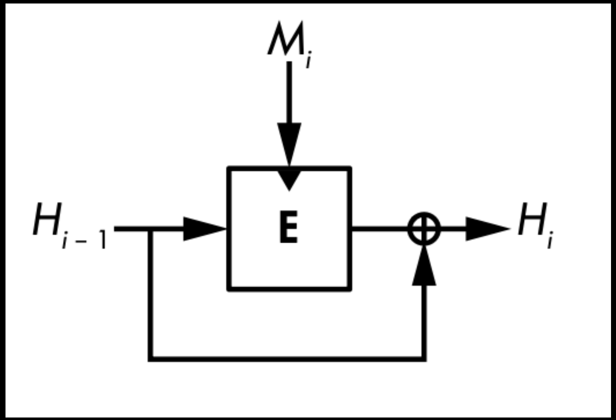
A collision in Compress does not necessarily give a collision on M-D hash.

If $Compress(X, M_1) = Compress(Y, M_2)$, for chaining values X and Y, won't result in a collision for the hash because the M-D construction is an iterative chain of hashes.

Building Hash Functions

Compression-Based Hash Functions: The Merkle–Damgård Construction

- The compression function can use the Davies–Meyer construction.
- In the DM construction, the compression function is based on a block cipher.



- The DM compression function uses a block cipher, E, to compute the new chaining value as $H_i = E(M_i, H_{i-1}) \oplus H_{i-1}$.
    - M_i acts as the key, H_{i-1} acts as the plaintext.

Building Hash Functions
Compression-Based Hash Functions: The Merkle–Damgård Construction

- Secure block cipher → collision and preimage resistant hash function.
- Examples: SHA256, BLAKE2
- Other constructions besides the Davies–Meyer.
  - Not popular, more complex.

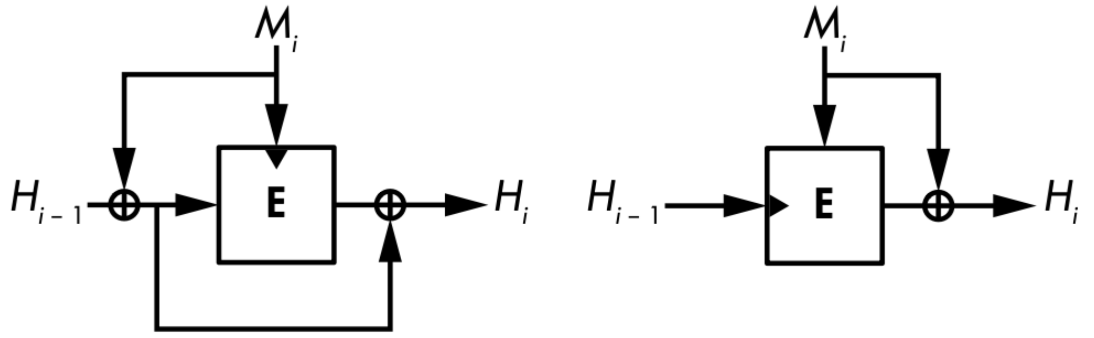
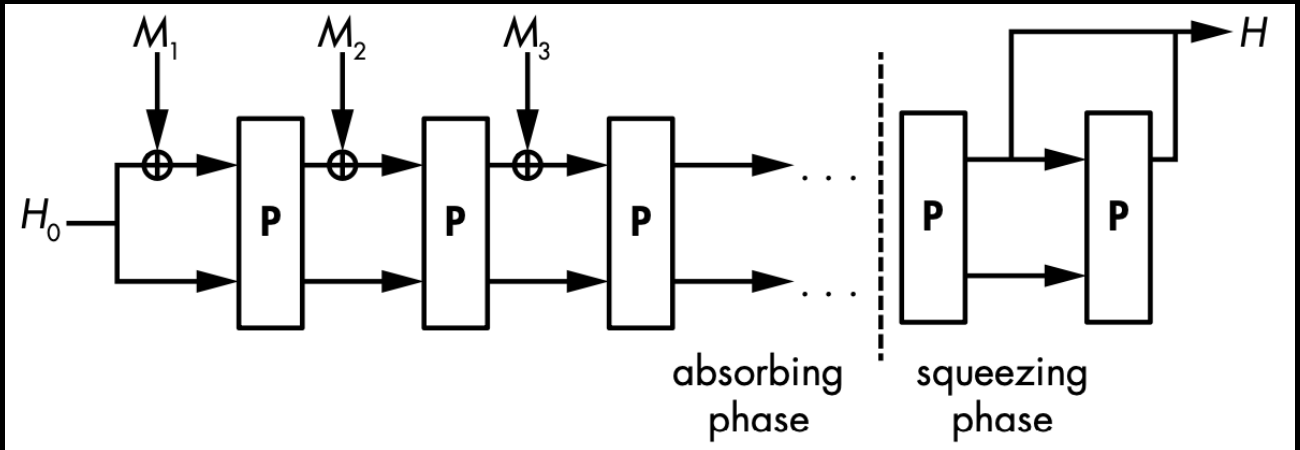


Figure 6-6: Other secure block cipher–based compression function constructions

Building Hash Functions

Permutation-Based Hash Functions: Sponge Functions

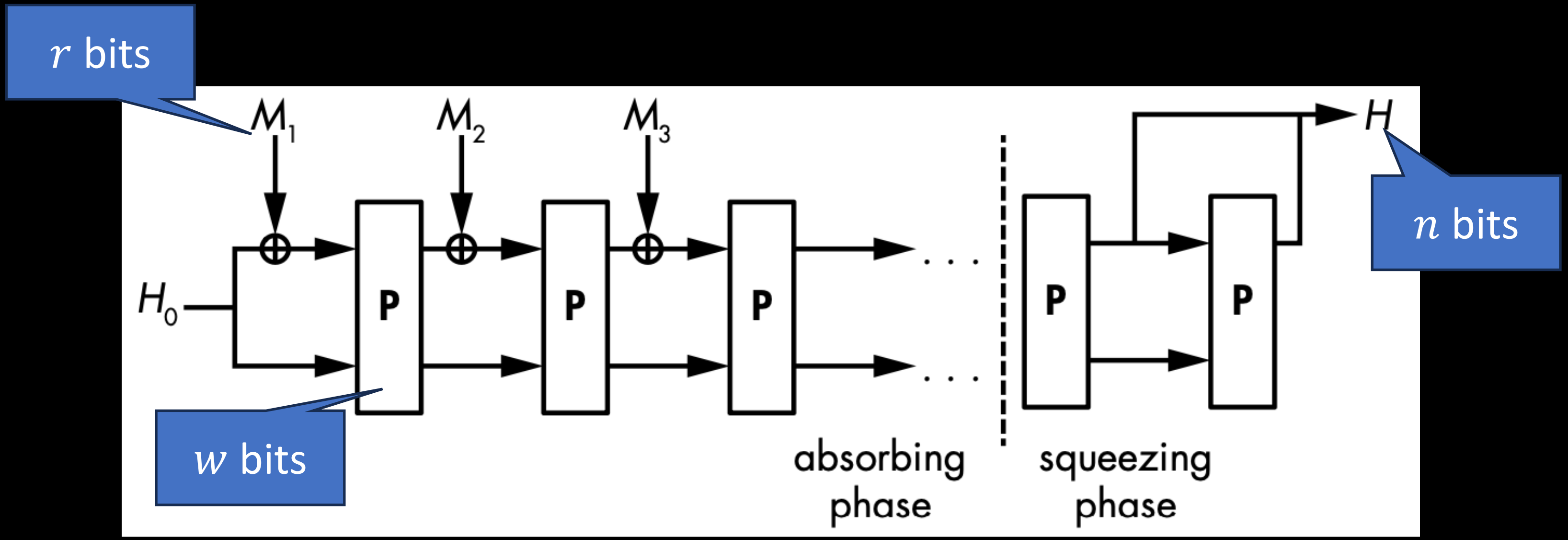
- Sponge functions use a single permutation instead of a compression function and a block cipher.
  - The sponge function just do an XOR.
  - The permutation should be random.
  - Padding: add ”1” bit followed by enough ”0”s.



Building Hash Functions

Permutation-Based Hash Functions: Sponge Functions

- Block size = r bits, internal state's size = w bits, hash value's size = n bits.
- Security level = $\min(c/2, n/2)$
- Sponge capacity $c = w - r$



Building Hash Functions

Permutation-Based Hash Functions: Sponge Functions

- Example: to reach 256-bit security with 64-bit message blocks, the internal state's size should be

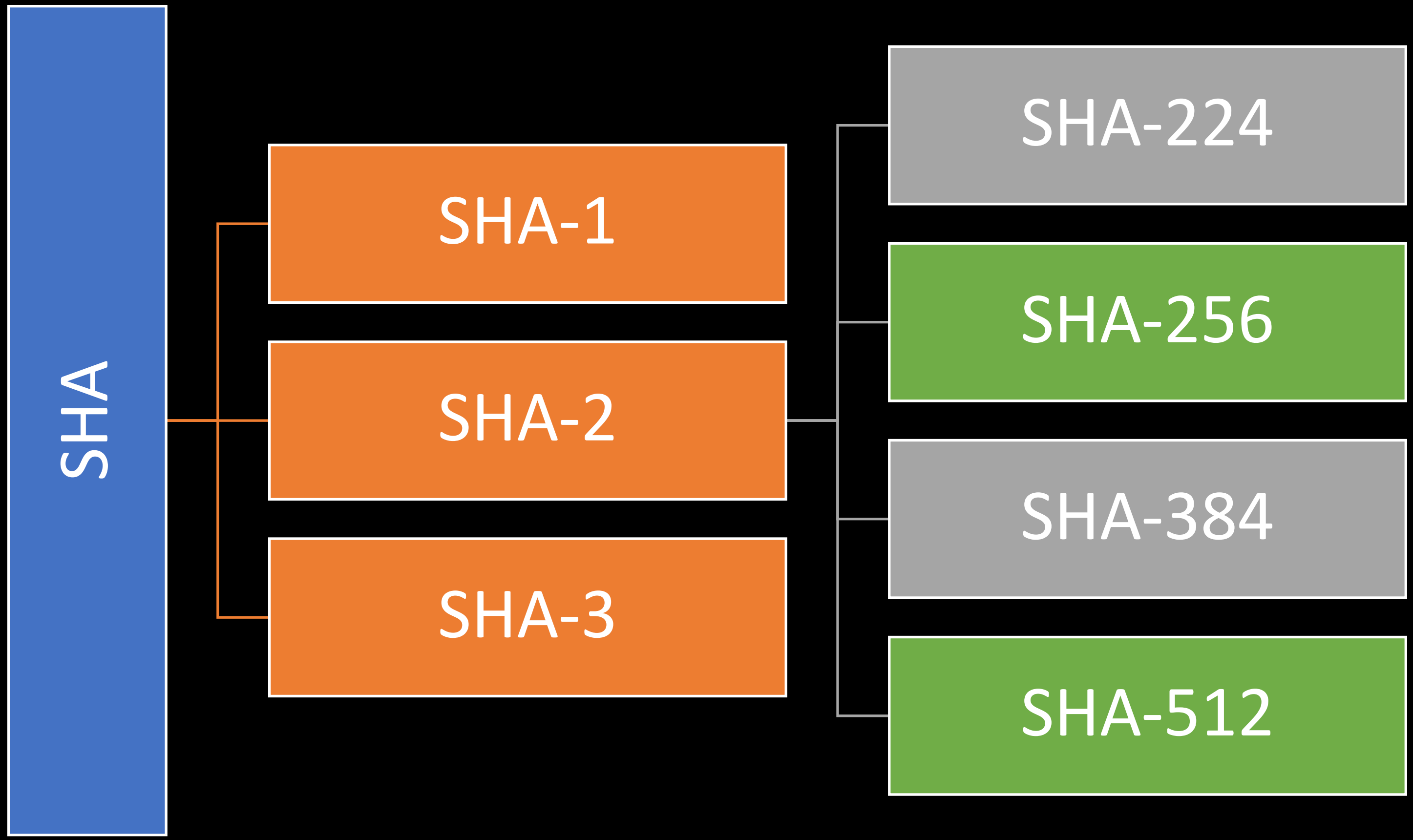
$c/2 = 256 \rightarrow (w - r)/2 = 256 \rightarrow w = 2\times 256 + 64 = 576$ bits

Content

Content
Introduction
Secure Hash Functions
Building Hash Functions
The SHA Family of Hash Functions
The BLAKE2 Hash Function
Attacks
Code Demos

The SHA Family of Hash Functions

- SHA = Secure Hashing Algorithm



The SHA Family of Hash Functions
SHA-1

- Replaces SHA-0:
  - Find collisions in 2^{33} operations (less than an hour).
- Based on MD-DM construction.
  - Uses a special block cipher called SHACAL.
- Applies the transformation: $H = E(M, H) + H$
  - $E(M, H)$ and H are viewed as arrays of 32-bit integers.
  - H is a 160-bit constant
- Block size 512 bits, used as a key in the block cipher.
- The output hash is 160-bit, viewed as an array of five 32-bit words.

The SHA Family of Hash Functions
SHA-1

- SHA-1 compression function

```
SHA1-compress(H, M) {
    (a0, b0, c0, d0, e0) = H   // parsing H as five 32-bit big endian words
    (a, b, c, d, e) = SHA1-blockcipher(a0, b0, c0, d0, e0, M)
    return (a + a0, b + b0, c + c0, d + d0, e + e0)
}
```

The SHA Family of Hash Functions
SHA-1

- SHA-1 Block cipher:

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

The SHA Family of Hash Functions
SHA-1

- SHA-1 Block cipher:

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

Takes a 512-bit message block, outputs an array of eighty 32-bit words

The SHA Family of Hash Functions
SHA-1

- SHA-1 Block cipher:

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

Iterate 80 times to process the expanded message block

The SHA Family of Hash Functions
SHA-1

- SHA-1 Block cipher:

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

K is a list of pre-defined values

The SHA Family of Hash Functions
SHA-1

- SHA-1 Block cipher:

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

Return the new transformed block.

The SHA Family of Hash Functions
SHA-1

- SHA-1 expansion function:

```
expand(M) {
    // the 512-bit M is seen as an array of sixteen 32-bit words
    W = empty array of eighty 32-bit words
    for i = 0 to 79 {
        if i < 16 then W[i] = M[i]
        else
            W[i] = (W[i - 3] ⊕ W[i - 8] ⊕ W[i - 14] ⊕ W[i - 16]) <<< 1
    }
    return W
}
```

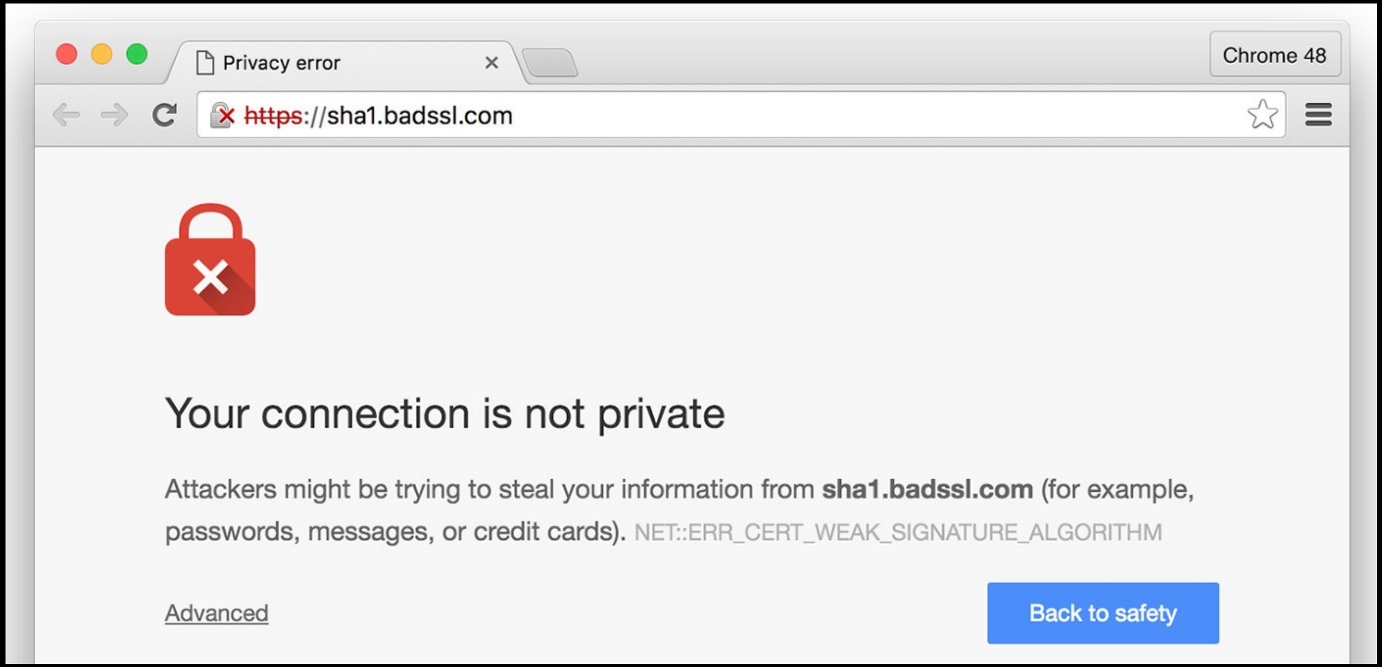
The SHA Family of Hash Functions
SHA-1

- SHA-1's f function is a sequence of basic bitwise logical operations that depends on the round number.

```
f(i, b, c, d) {
    if i < 20 then return ((b & c) ⊕ (~b & d))
    if i < 40 then return (b ⊕ c ⊕ d)
    if i < 60 then return ((b & c) ⊕ (b & d) ⊕ (c & d))
    if i < 80 then return (b ⊕ c ⊕ d)
}
```

The SHA Family of Hash Functions
SHA-1

- The hash value of SHA-1 is 160 bits → gives 160/2 = 80 bits of security.
- Weaknesses lead to finding collisions in 2^{63} operations instead of 2^{80}.
- Browsers mark it as insecure.



The SHA Family of Hash Functions SHA-2

- SHA-2:
  - Replaces SHA-1
  - Block size 512 bits
  - Chaining values are 256 bits, viewed as eight 32-bit words
- It makes 64 rounds.
  - The number of rounds less than of SHA-1.
- More complex rounds and compression.
- Expands the 16-word message block to a 64-word message block

The SHA Family of Hash Functions SHA-2

- SHA-2 expand function:

```
expand256(M) {
    // the 512-bit M is seen as an array of sixteen 32-bit words
    W = empty array of sixty-four 32-bit words
    for i = 0 to 63 {
        if i < 16 then W[i] = M[i]
        else {
            // the ">>" shifts instead of a ">>>" rotates and is not a typo
            s0 = (W[i - 15] >>> 7) ⊕ (W[i - 15] >>> 18) ⊕ (W[i - 15] >> 3)
            s1 = (W[i - 2] >>> 17) ⊕ (W[i - 2] >>> 19) ⊕ (W[i - 2] >> 10)
            W[i] = W[i - 16] + s0 + W[i - 7] + s1
        }
    }
    return W
}
```

Listing 6-8: SHA-256's expand256() function

The SHA Family of Hash Functions
SHA-2

- SHA-512 is similar to SHA-256 except:
  - It works with 64-bit words instead of 32-bit words
  - It uses 512-bit chaining values (eight 64-bit words)
  - Uses 1024-bit message blocks (sixteen 64-bit words)
  - Makes 80 rounds instead of 64

The SHA Family of Hash Functions
SHA-3

- In 2007, NIST Hash Function Competition.
  - Need to have a hash standard other than SHA-1 (broken) and SHA-2 (not yet broken)
- Requirements:
  - At least as secure and as fast as SHA-2
  - Do not be similar to SHA-1 and SHA-2

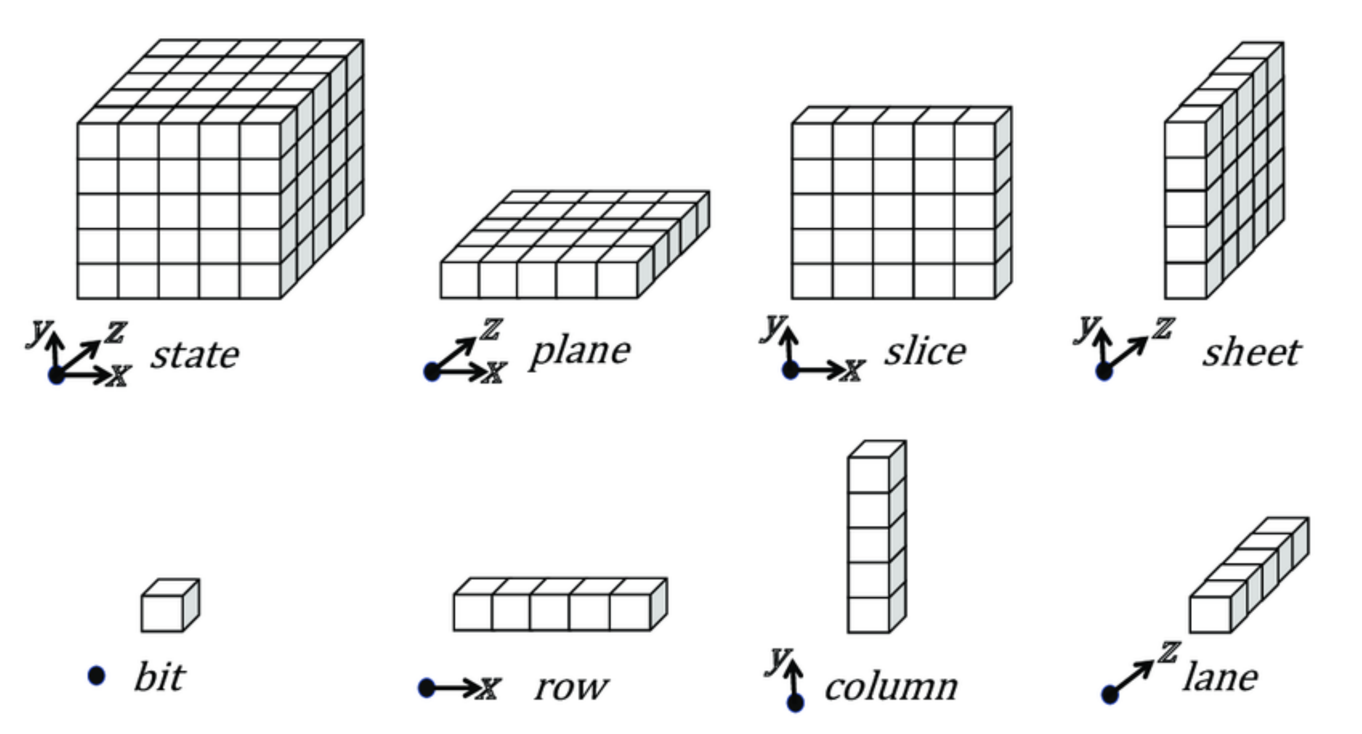
- In 2009, out of 64 submissions, NIST announced five finalists:
  - Blake, Grostl, JH, Keccak, Skein

The SHA Family of Hash Functions
SHA-3

BLAKE	Grostl	JH	Keccak	Skein
• MD construction • Its block cipher is based on a stream cipher, ChaCha • Performs additions, XORs, and rotations	• Enhanced MD construction • Its block cipher uses two permutations, based on AES	• Tweaked sponge function • Message blocks are injected before and after the permutation	• Sponge construction • Performs only bitwise operations	• Different from MD constructions • Compression is based on a new block cipher • Performs additions, XORs, and rotations

The SHA Family of Hash Functions
SHA-3

- Keccak (SHA-3):
  - Sponge function.
  - Its permutation functions operate on 1600-bit state.
  - Block size: 1152, 1088, 832, or 576 bits.
  - Hash length: 224, 256, 384, or 512 bits.



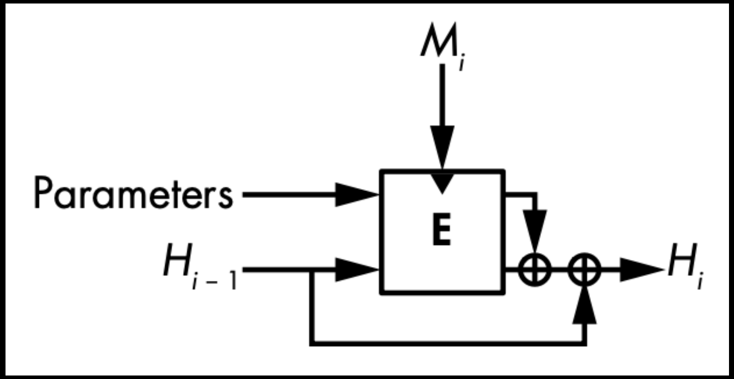
Content

The BLAKE2 Hash Function

- BLAKE2 is the fastest non-NIST standard hash algorithm.
  - Released after SHA-3 competition.
- It has two versions:
  - **BLAKE2b**, optimized for 64-bit platforms, produces digests ranging from 1 to 64 bytes.
  - **BLAKE2s**, optimized for 8- to 32-bit platforms, produces digests ranging from 1 to 32 bytes.
- Integrated into OpenSSL and Sodium libraires

The BLAKE2 Hash Function

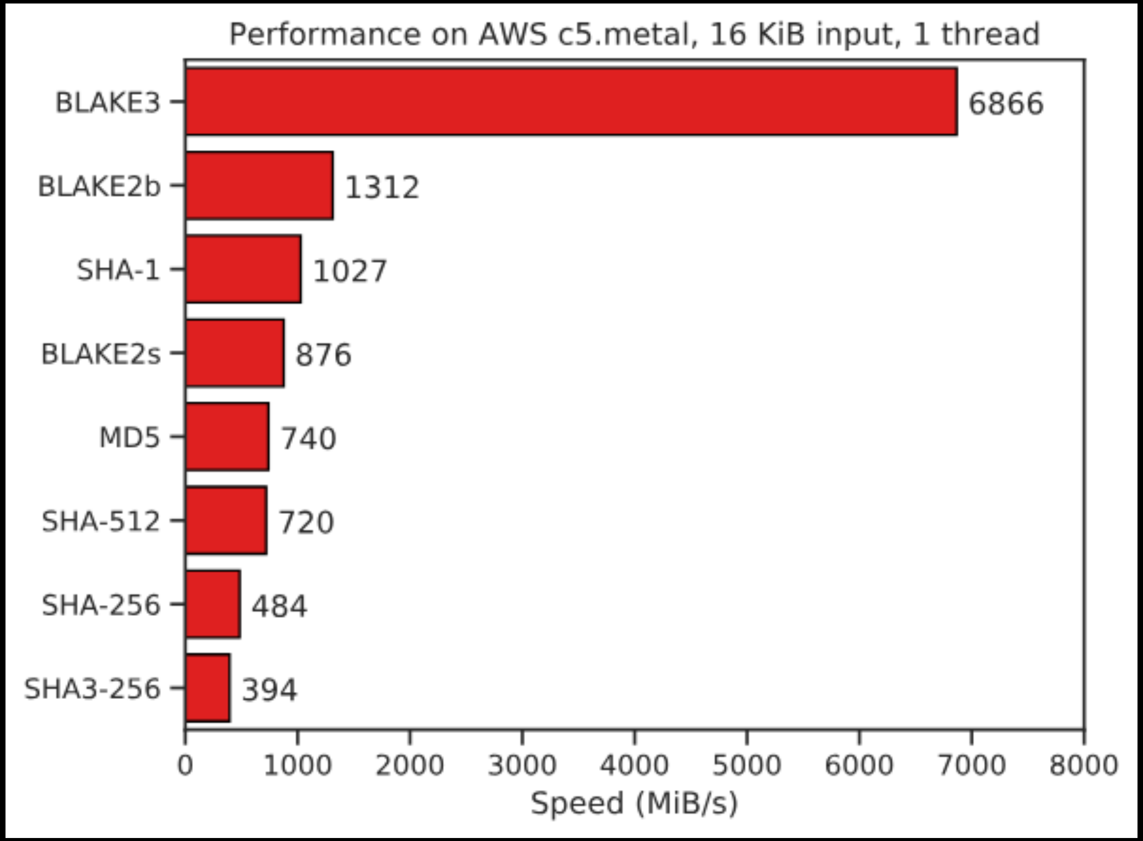
- BLAKE2 compression function is based on the Davies–Meyer construction:



  - The compression function is based on the stream cipher ChaCha.
  - Parameters:
    - 1) counter: ensures that the output of each compression function is unique.
    - 2) flag: indicates whether to process the last message block or not.

The BLAKE2 Hash Function

- BLAKE3:
  - Much faster than MD5, SHA-1, SHA-2, SHA-3, and BLAKE2.
  - Secure against length extension attack.
  - Highly parallelizable.
  - One algorithm with no variants.



https://github.com/BLAKE3-team/BLAKE3

Content

Content
Introduction
Secure Hash Functions
Building Hash Functions
The SHA Family of Hash Functions
The BLAKE2 Hash Function
Attacks
Code Demos

Attacks
Birthday Attack

- Birthday attack allows us to find collisions in $2^{n/2}$ operations.
    - n is the bit length of the hash value.

- Naïve algorithm:
    1. Compute $2^{\frac{n}{2}}$ hashes of $2^{\frac{n}{2}}$ random msgs and store all the msg/hash pairs in a list.
    2. Sort the list by the hash value to move any identical hash values next to each other.
    3. Search the sorted list to find two consecutive entries with the same hash value.

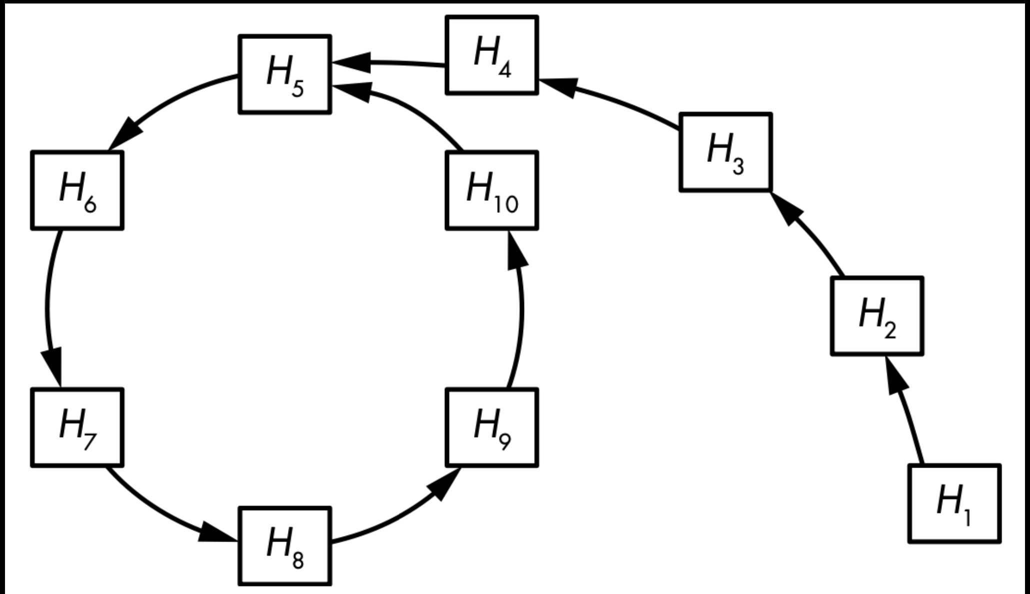
Attacks
Birthday Attack

- The naïve algorithm is inefficient, because it requires $2^{\frac{n}{2}}$ memory space.

- The Rho Method: low memory collision search
  1. Pick random value (H_1) and define $H_1 = H'_1$
  2. Compute $H_2 = Hash(H_1)$ and $H'_2 = Hash\big(Hash(H'_1)\big)$
  3. Repeat the process and compute $H_{i+1} = Hash(H_i)$ and $H'_{i+1} = Hash\big(Hash(H'_i)\big)$
  4. Stop when reaching i such that $H_{i+1} = H'_{i+1}$

Attacks
Birthday Attack

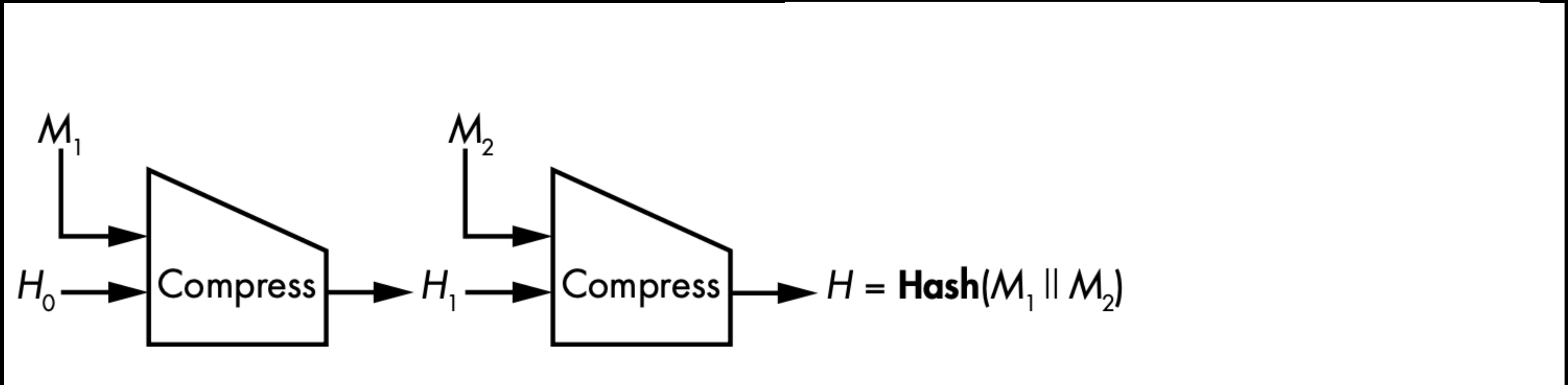
- The idea is that to find a collision, you need to enter the cycle of the hash.



- The cycle starts at H_5 where $Hash(H_4) = Hash(H_{10}) = H_5$

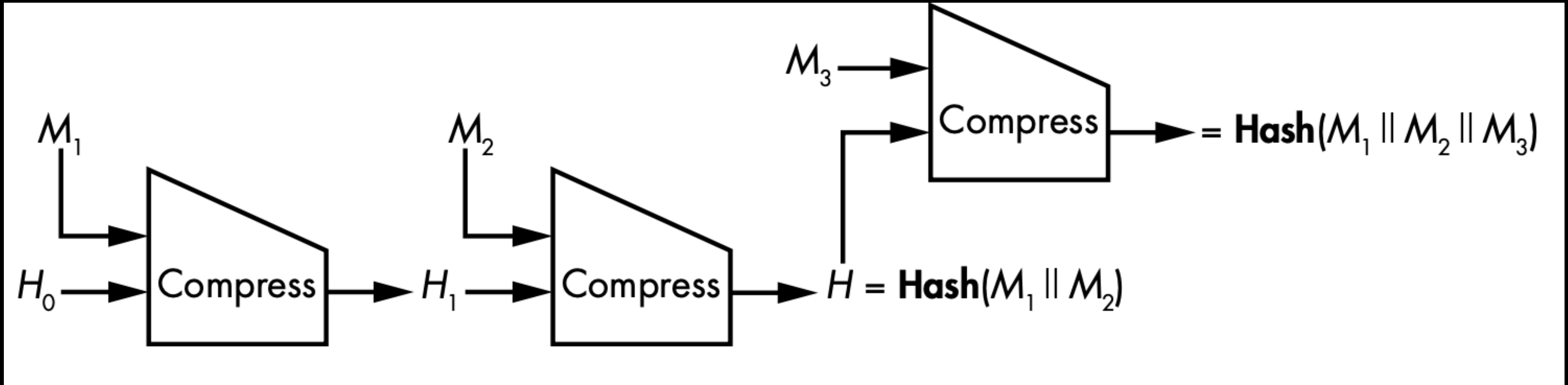
Attacks
The Length-Extension Attack

- The main threat to the Merkle–Damgård construction.
- Given an *unknown* message M that is composed of two blocks M_1 and M_2 .
- In a MD hash construction, the hash is computed as



Attacks
The Length-Extension Attack

- An attacker can append new data to the hash to calculate a valid hash.
  - Thus, violating the integrity of the message.
- Mitigation: make the last compression function call different from all others.
  - That’s what BLAKE2 does



Attacks
Fooling Proof-of-Storage Protocols

- Proofs of storage (PoS) are cryptographic protocols that allow a client to efficiently verify the integrity of remotely stored data.

The premise: if the server does not maintain a valid Data, it wouldn't be able to fool the client.

Data

Data

Random C

$$H_1 = Compress(H_0, D)$$
$$X = Compress(H_1, C)$$

Send X

$$H_1 = Compress(H_0, D)$$
$$Y = Compress(H_1, C) == X?$$

Attacks
Fooling Proof-of-Storage Protocols

- Proofs of storage (PoS) are cryptographic protocols that allow a client to efficiently verify the integrity of remotely stored data.

The server deletes Data

Data

$H_1 = Compress(H_0, D)$

Random C

$X = Compress(H_1, C)$

Send X

$H_1 = Compress(H_0, D)$
$Y = Compress(H_1, C) == X?$

Attacks
Fooling Proof-of-Storage Protocols

- This trick will work for SHA-1, SHA-2, as well as SHA-3 and BLAKE2.
- Mitigation: compute $Hash(C||D)$ instead of $Hash(D||C)$

Content

Content
Introduction
Secure Hash Functions
Building Hash Functions
The SHA Family of Hash Functions
The BLAKE2 Hash Function
Attacks
Code Demos

Code Demos

- $hashlib$ is a python module that implements hash functions:
  - md5, sha1, sha224, sha256, sha384, sha512, sha3_224, sha3_256, sha3_384, sha3_512
  - Blake2b, Blake2s

- Hashes.py, Find_Preimage.py, Find_Collisions.py, Naïve_Birthday_Attack.py

TASK

- What is salting in hashing algorithms?
- Implement the Pollard Rho method to find a collision in the SHA256.
  - Assume the starting value is $0x000 \ldots 000$
  - Use only 12 nibbles.
  - Measure the execution time.
- Challenge: you are given the hash value of a **common** password below. Identify the type of the hash and the password.

“9716177b3bac86c47e2c69c25a1aa04c2252cf76570932a3b2500c8f1ae72017ee39d5ff30a2cb9e2da76af07f400c58a0533aa194c4093dbc6aabffed211195”

  - HINT: You do not need to use advanced brute forcing tools, search for online tools.