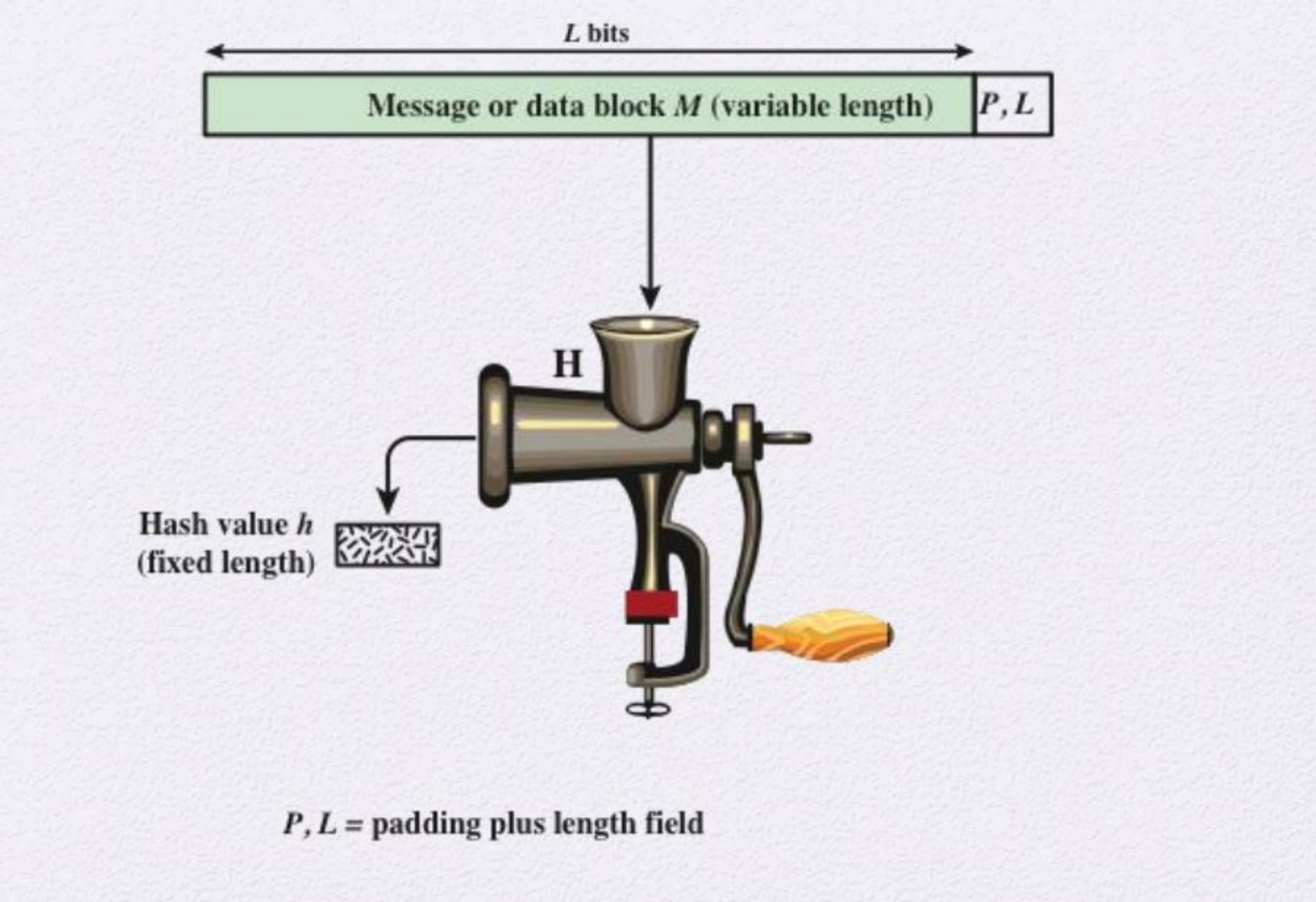
# CRYPTOGRAPHIC HASH FUNCTIONS

# HASH FUNCTIONS

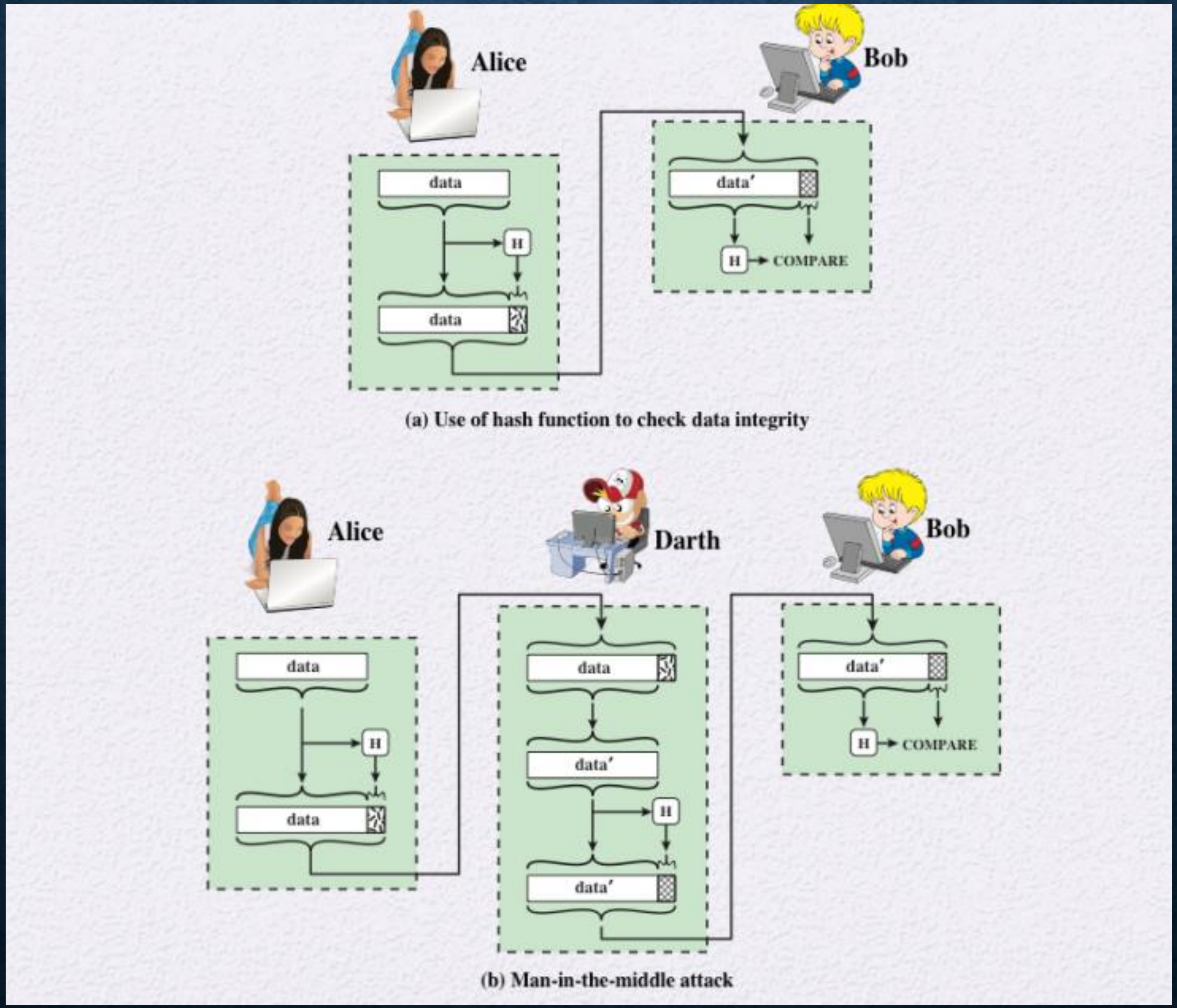
- A hash function H accepts a variable-length block of data *M* as input and produces a fixed-size hash value
  - $h = \mathrm{H}(M)$
  - Principal object is data integrity
- Cryptographic hash function
  - An algorithm for which it is computationally infeasible to find either:

    (a) a data object that maps to a pre-specified hash result (the *one-way property*)

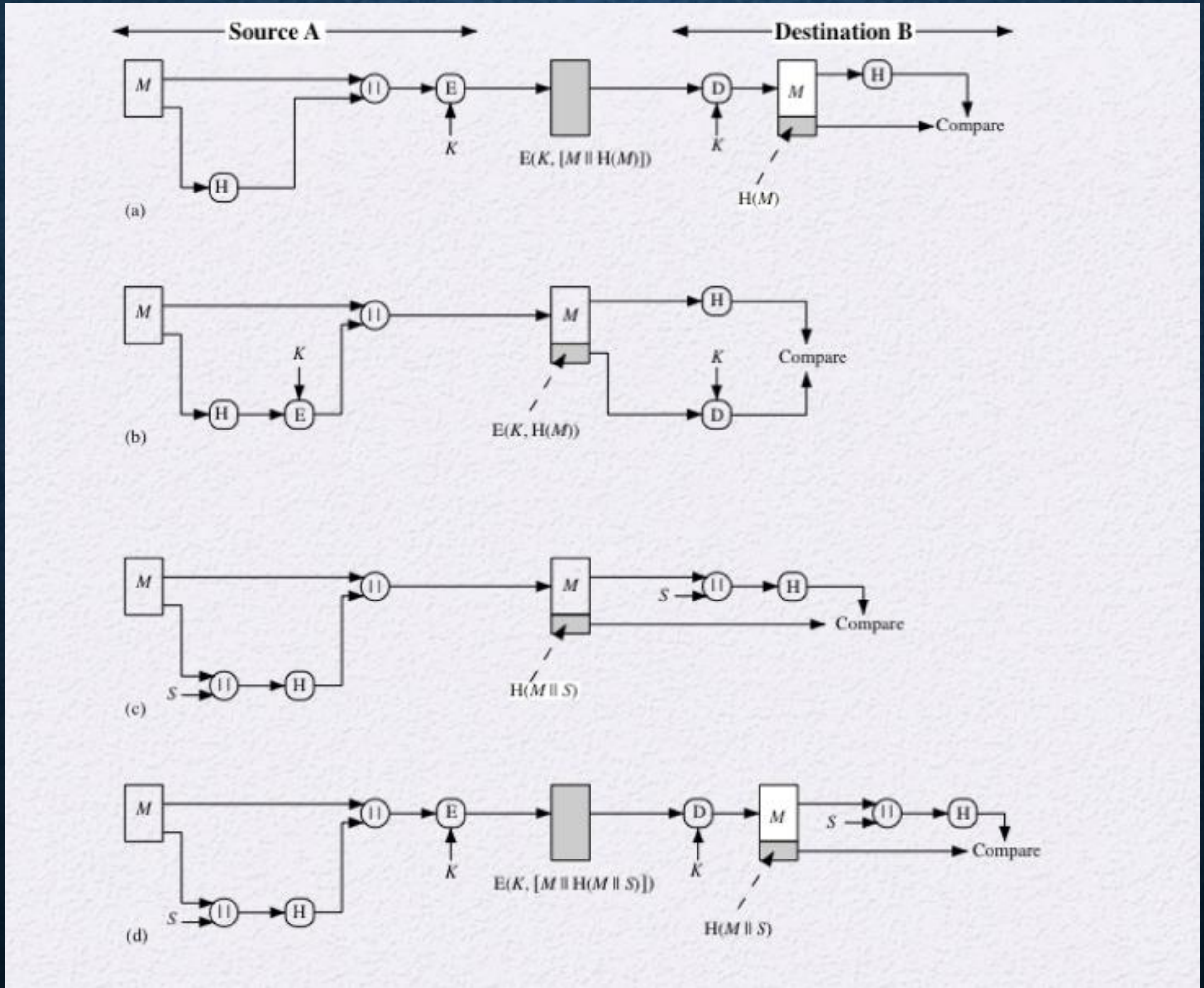
    (b) two data objects that map to the same hash result (the *collision-free property*)


L bits
Message or data block M (variable length)
P, L
H
Hash value h
(fixed length)
P, L = padding plus length field

# ATTACKS AGAINST HASH FUNCTIONS



# USE OF HASH FUNCTIONS



# MESSAGE AUTHENTICATION CODE (MAC)

- Also known as a *keyed hash function*
- Typically used between two parties that share a secret key to authenticate information exchanged between those parties

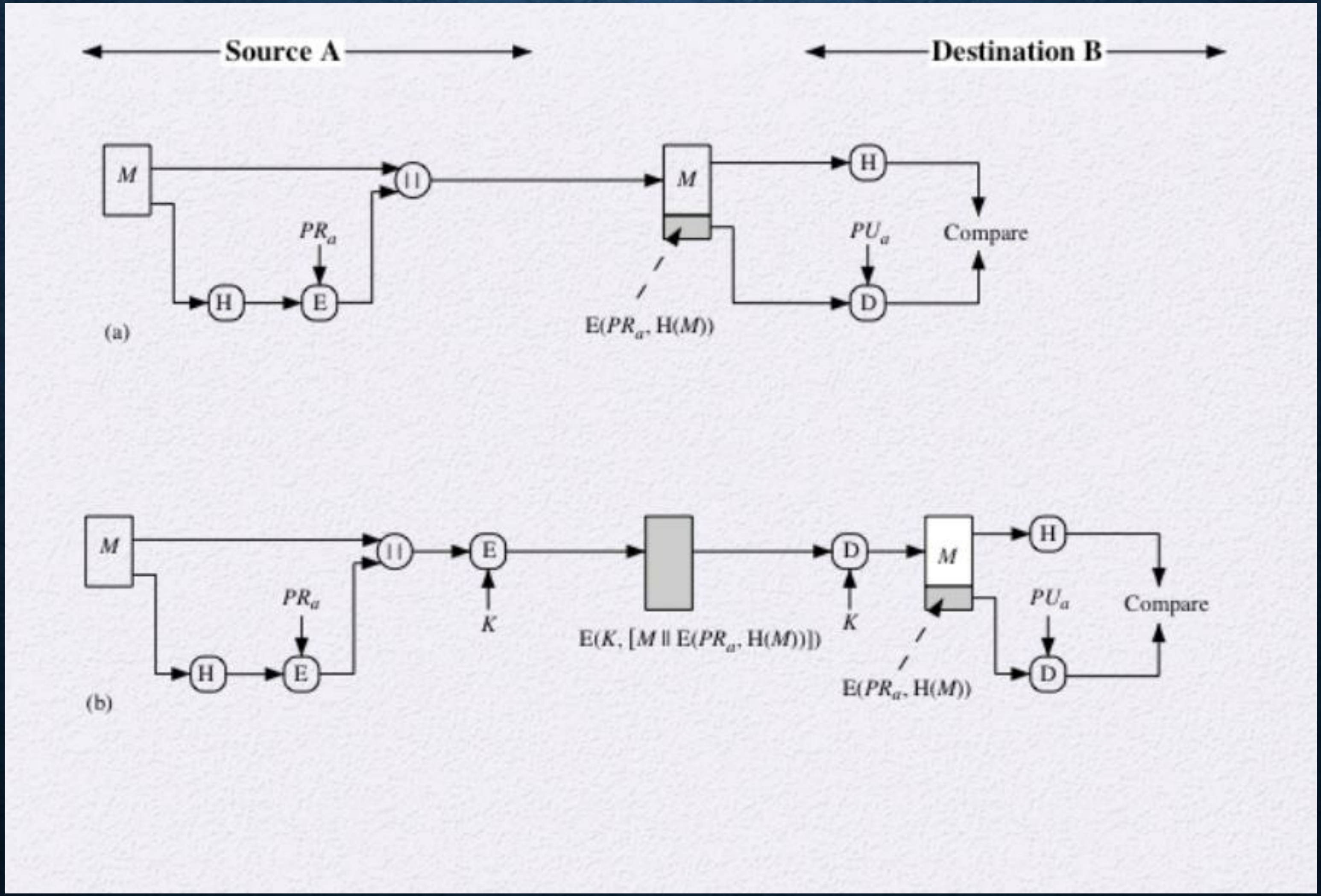
**Takes as input a secret key and a data block and produces a hash value (MAC) which is associated with the protected message**

- If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the associated MAC value
- An attacker who alters the message will be unable to alter the associated MAC value without knowledge of the secret key

# DIGITAL SIGNATURE

- Operation is similar to that of the MAC
- The hash value of a message is encrypted with a user's private key
- Anyone who knows the user's public key can verify the integrity of the message
- An attacker who wishes to alter the message would need to know the user's private key
- Implications of digital signatures go beyond just message authentication

# DIGITAL SIGNATURES



# OTHER HASH FUNCTION USES

**Commonly used to create a one-way password file**

- When a user enters a password, the hash of that password is compared to the stored hash value for verification
- This approach to password protection is used by most operating systems

**Can be used for intrusion and virus detection**

- Store H(F) for each file on a system and secure the hash values
- One can later determine if a file has been modified by recomputing H(F)
- An intruder would need to change F without changing H(F)

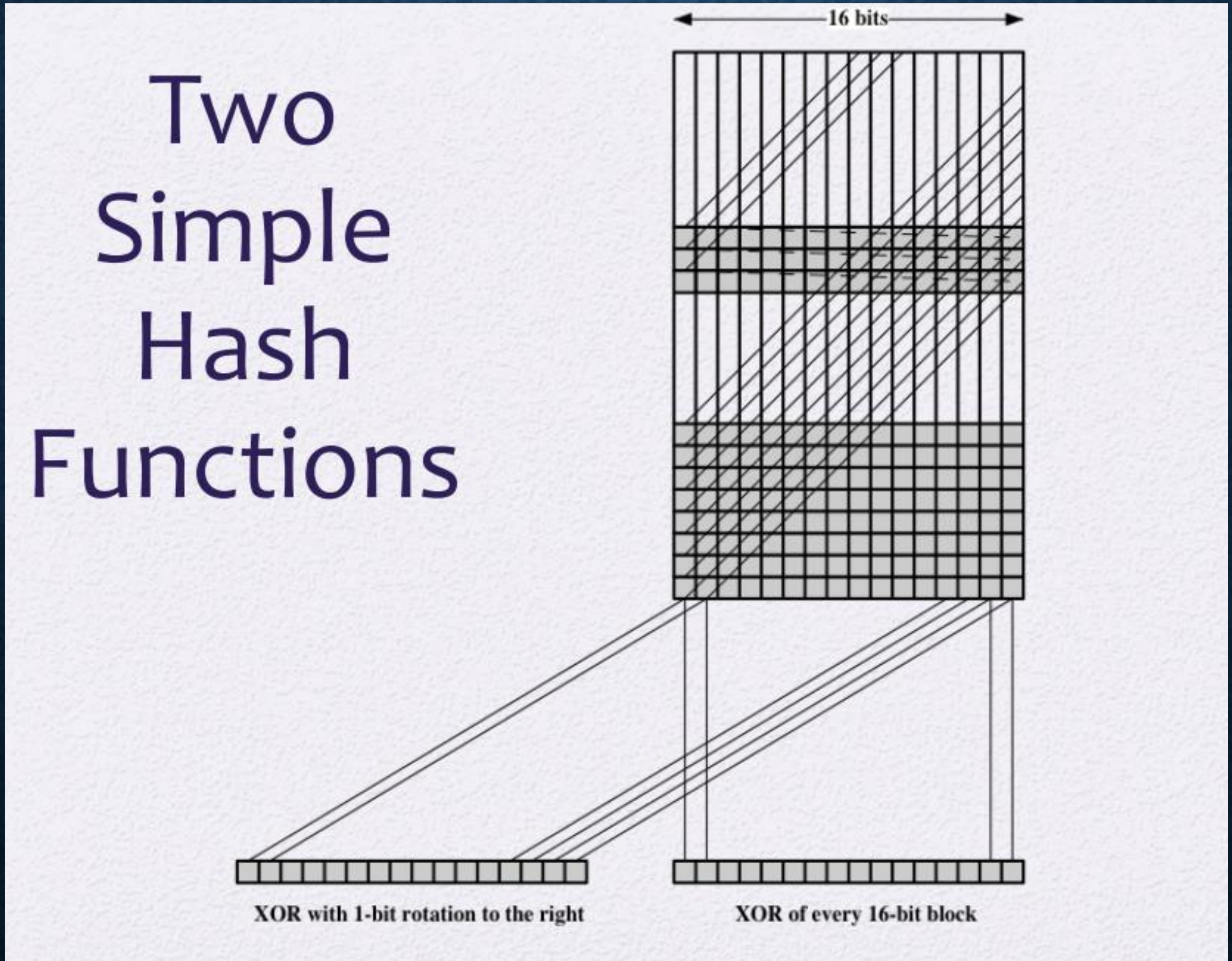
**Can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG)**

- A common application for a hash-based PRF is for the generation of symmetric keys

# TWO SIMPLE HASH FUNCTIONS

- Consider two simple insecure hash functions that operate using the following general principles:
  - The input is viewed as a sequence of *n*-bit blocks
  - The input is processed one block at a time in an iterative fashion to produce an *n*-bit hash function
- Bit-by-bit exclusive-OR (XOR) of every block
  - $C_i = b_{i1} \text{ xor } b_{i2} \text{ xor} \ldots \text{xor } b_{im}$
  - Produces a *simple parity* for each bit position and is known as a longitudinal redundancy check
  - Reasonably effective for random data as a data integrity check
- Perform a one-bit *circular shift* on the hash value after each block is processed
  - Has the effect of randomizing the input more completely and overcoming any regularities that appear in the input

# TWO SIMPLE HASH FUNCTIONS



# REQUIREMENTS AND SECURITY

## Preimage

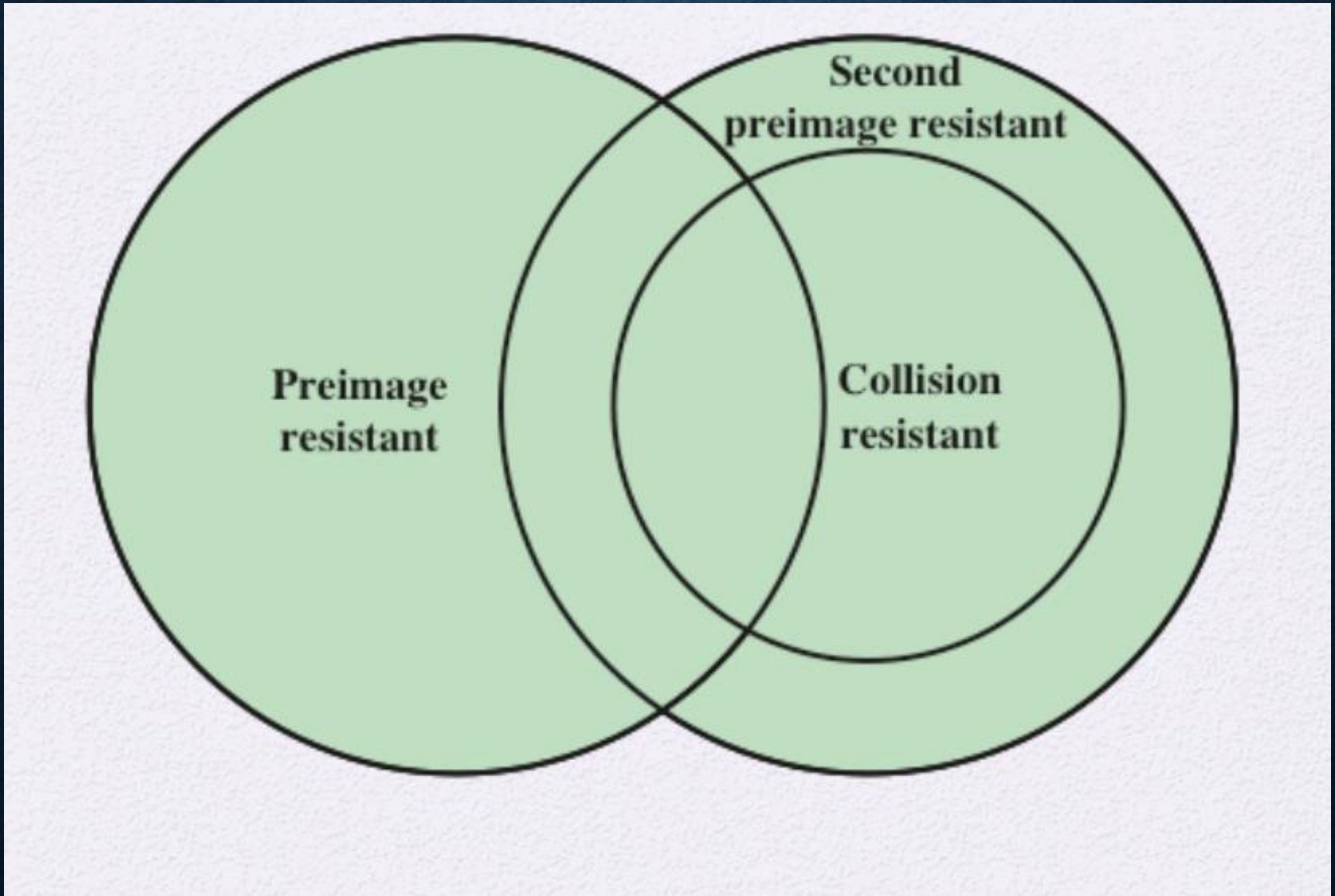
- $x$ is the preimage of $h$ for a hash value $h = H(x)$
- Is a data block whose hash function, using the function H, is $h$
- Because H is a many-to-one mapping, for any given hash value $h$, there will in general be multiple preimages

## Collision

- Occurs if we have $x \neq y$ and $H(x) = H(y)$
- Because we are using hash functions for data integrity, collisions are clearly undesirable

# QUIREMENTS FOR A CRYPTOGRAPHIC HASH FUNCTION H

| Requirement | Description |
|---|---|
| Variable input size | H can be applied to a block of data of any size. |
| Fixed output size | H produces a fixed-length output. |
| Efficiency | $H(x)$ is relatively easy to compute for any given $x$, making both hardware and software implementations practical. |
| Preimage resistant (one-way property) | For any given hash value $h$, it is computationally infeasible to find $y$ such that $H(y) = h$. |
| Second preimage resistant (weak collision resistant) | For any given block $x$, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$. |
| Collision resistant (strong collision resistant) | It is computationally infeasible to find any pair $(x, y)$ such that $H(x) = H(y)$. |
| Pseudorandomness | Output of H meets standard tests for pseudorandomness |


Second
preimage resistant
Preimage
resistant
Collision
resistant

# ATTACKS ON HASH FUNCTIONS

## Brute-Force Attacks

- Does not depend on the specific algorithm, only depends on bit length
- In the case of a hash function, attack depends only on the bit length of the hash value
- Method is to pick values at random and try each one until a collision occurs

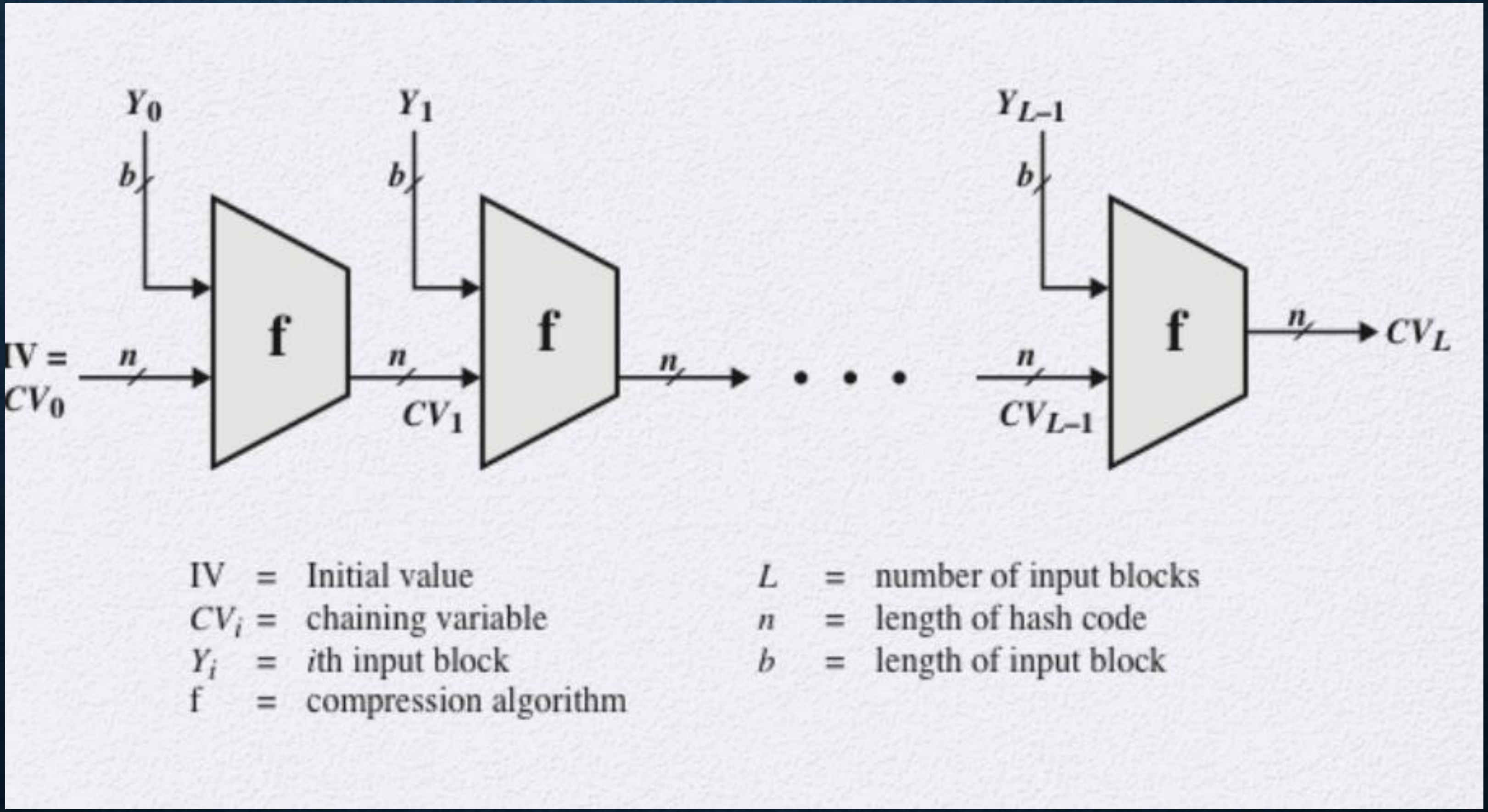
## Cryptanalysis

- An attack based on weaknesses in a particular cryptographic algorithm
- Seek to exploit some property of the algorithm to perform some attack other than an exhaustive search

# BIRTHDAY ATTACKS

- For a collision resistant attack, an adversary wishes to find two messages or data blocks that yield the same hash function
  - The effort required is explained by a mathematical result referred to as the *birthday paradox*
- How the birthday attack works:
  - The source (A) is prepared to sign a legitimate message *x* by appending the appropriate *m*-bit hash code and encrypting that hash code with A's private key
  - Opponent generates $2^{m/2}$ variations *x'* of *x*, all with essentially the same meaning, and stores the messages and their hash values
  - Opponent generates a fraudulent message *y* for which A's signature is desired
  - Two sets of messages are compared to find a pair with the same hash
  - The opponent offers the valid variation to A for signature which can then be attached to the fraudulent variation for transmission to the intended recipient
    - Because the two variations have the same hash code, they will produce the same signature and the opponent is assured of success even though the encryption key is not known

# GENERAL STRUCTURE OF SECURE HASH CODES



# HASH FUNCTIONS BASED ON CIPHER BLOCK CHAINING

- Can use block ciphers as hash functions
  - Using $H_0$ initial value
  - Compute: $H_i = E(M_i \; H_{i-1})$
  - Use final block $H_n$ as the hash value
  - Similar to CBC but without a key
- Resulting hash is too small (64-bit)
  - Both due to direct birthday attack
  - And “meet-in-the-middle” attack
- Other variants also susceptible to attack

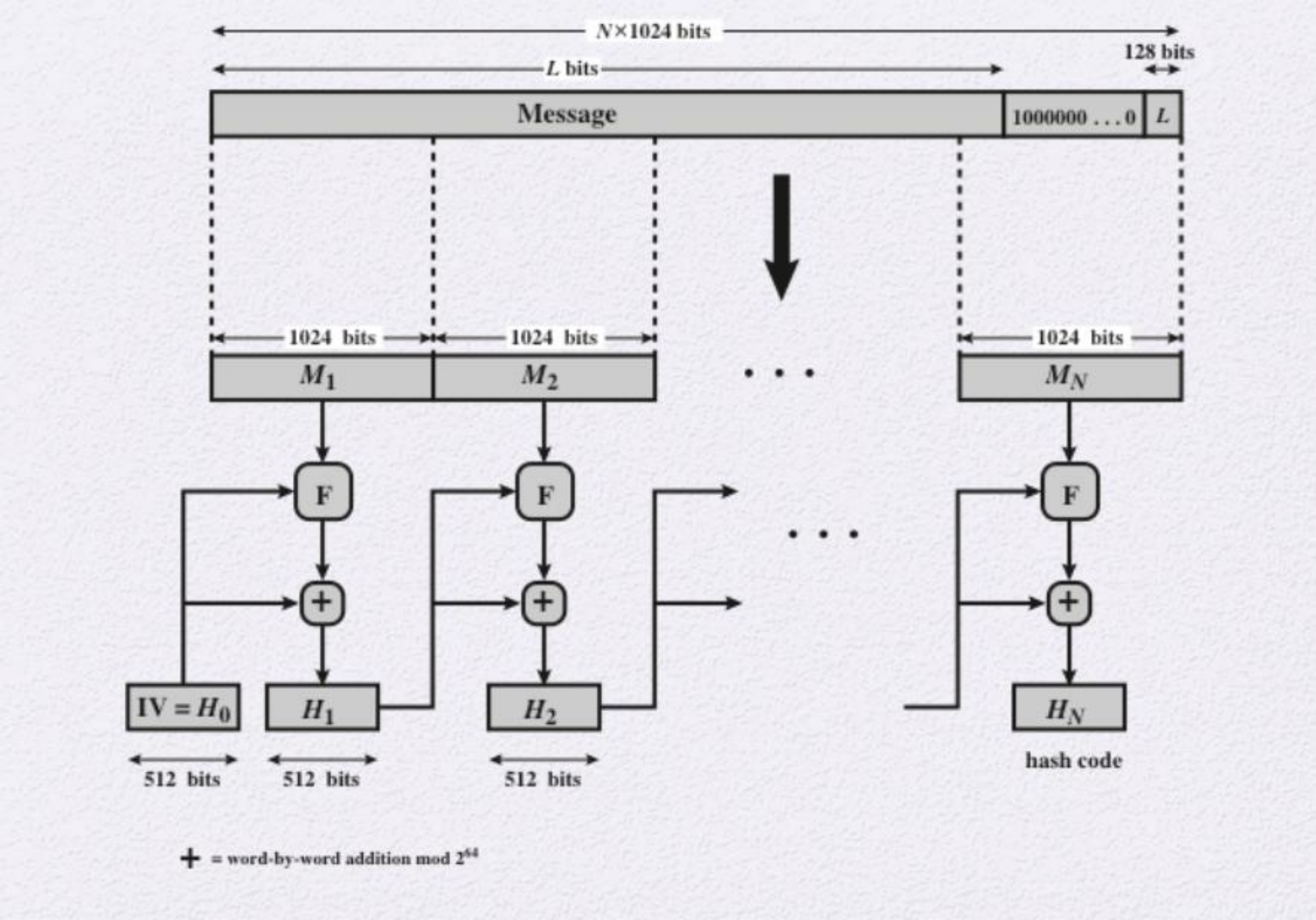
# SECURE HASH ALGORITHM (SHA)

- SHA was originally designed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993
- Was revised in 1995 as SHA-1
- Based on the hash function MD4 and its design closely models MD4
- Produces 160-bit hash values
- In 2002 NIST produced a revised version of the standard that defined three new versions of SHA with hash value lengths of 256, 384, and 512
    - Collectively known as SHA-2

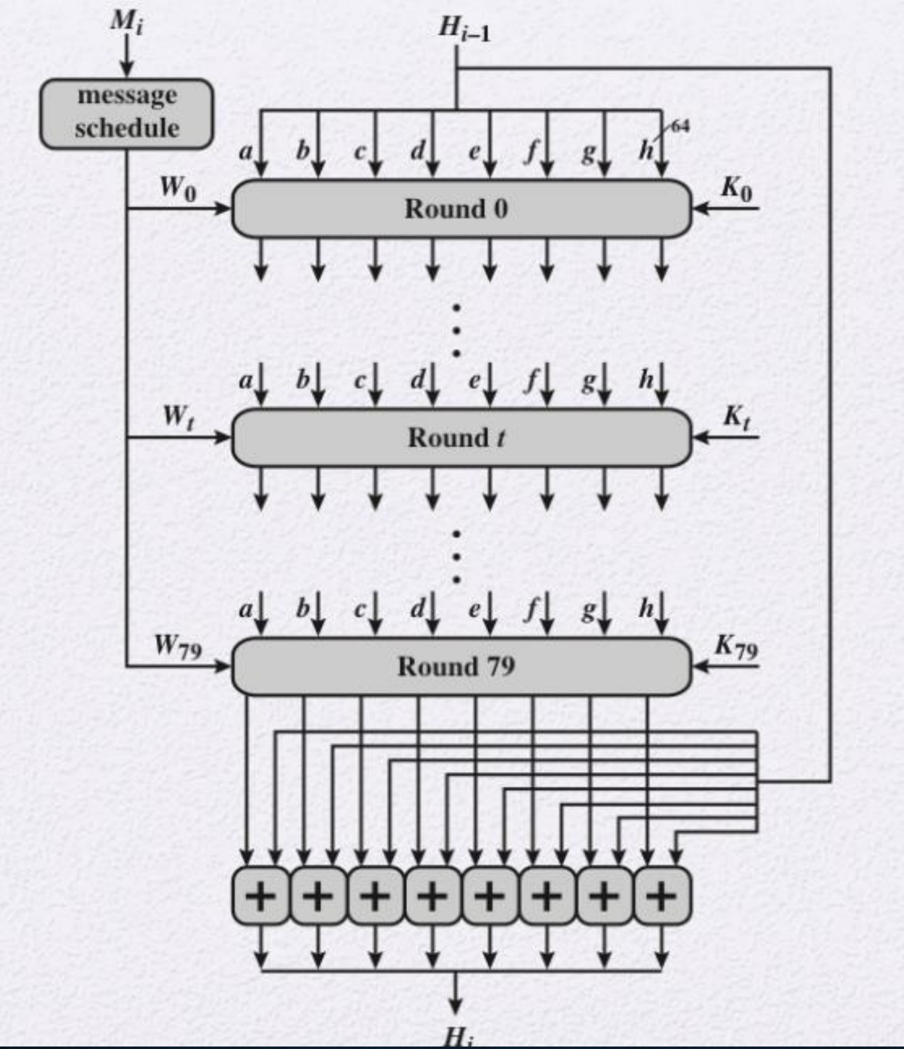
# COMPARISON OF SHA PARAMETERS

| | SHA-1 | SHA-224 | SHA-256 | SHA-384 | SHA-512 |
|---|---|---|---|---|---|
| Message Digest Size | 160 | 224 | 256 | 384 | 512 |
| Message Size | $< 2^{64}$ | $< 2^{64}$ | $< 2^{64}$ | $< 2^{128}$ | $< 2^{128}$ |
| Block Size | 512 | 512 | 512 | 1024 | 1024 |
| Word Size | 32 | 32 | 32 | 64 | 64 |
| Number of Steps | 80 | 64 | 64 | 80 | 80 |

# SHA 512 MESSAGE DIGEST GENERATION



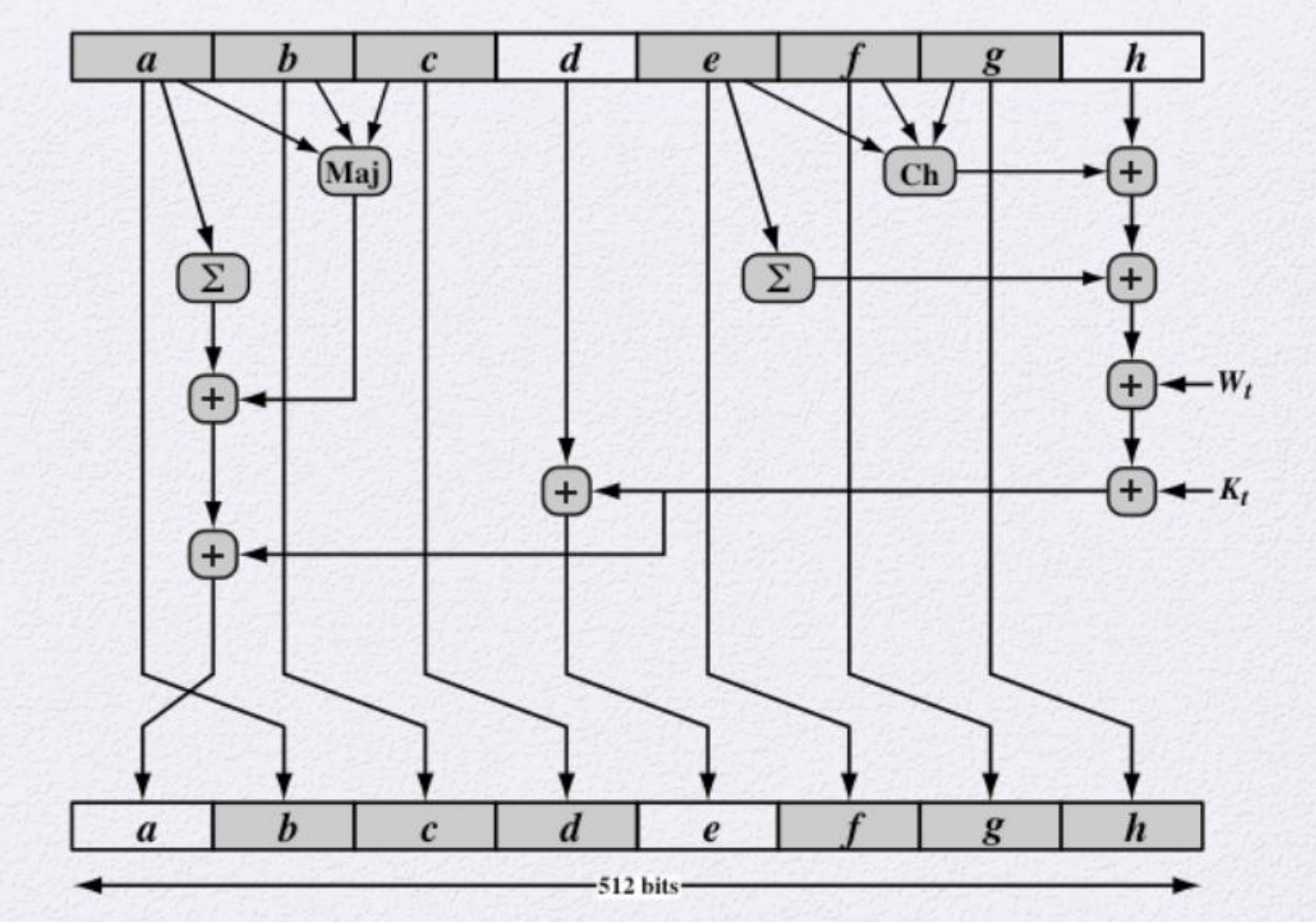
# PROCESSING A SINGLE 1024 BIT BLOCK



# SHA 512 CONSTANTS

| | | | |
|---|---|---|---|
| 428a2f98d728ae22 | 7137449123ef65cd | b5c0fbcfec4d3b2f | e9b5dba58189dbbc |
| 3956c25bf348b538 | 59f111f1b605d019 | 923f82a4af194f9b | ab1c5ed5da6d8118 |
| d807aa98a3030242 | 12835b0145706fbe | 243185be4ee4b28c | 550c7dc3d5ffb4e2 |
| 72be5d74f27b896f | 80deb1fe3b1696b1 | 9bdc06a725c71235 | c19bf174cf692694 |
| e49b69c19ef14ad2 | efbe4786384f25e3 | 0fc19dc68b8cd5b5 | 240ca1cc77ac9c65 |
| 2de92c6f592b0275 | 4a7484aa6ea6e483 | 5cb0a9dcbd41fbd4 | 76f988da831153b5 |
| 983e5152ee66dfab | a831c66d2db43210 | b00327c898fb213f | bf597fc7beef0ee4 |
| c6e00bf33da88fc2 | d5a79147930aa725 | 06ca6351e003826f | 142929670a0e6e70 |
| 27b70a8546d22ffc | 2e1b21385c26c926 | 4d2c6dfc5ac42aed | 53380d139d95b3df |
| 650a73548baf63de | 766a0abb3c77b2a8 | 81c2c92e47edaee6 | 92722c851482353b |
| a2bfe8a14cf10364 | a81a664bbc423001 | c24b8b70d0f89791 | c76c51a30654be30 |
| d192e819d6ef5218 | d69906245565a910 | f40e35855771202a | 106aa07032bbd1b8 |
| 19a4c116b8d2d0c8 | 1e376c085141ab53 | 2748774cdf8eeb99 | 34b0bcb5e19b48a8 |
| 391c0cb3c5c95a63 | 4ed8aa4ae3418acb | 5b9cca4f7763e373 | 682e6ff3d6b2b8a3 |
| 748f82ee5defb2fc | 78a5636f43172f60 | 84c87814a1f0ab72 | 8cc702081a6439ec |
| 90befffa23631e28 | a4506cebde82bde9 | bef9a3f7b2c67915 | c67178f2e372532b |
| ca273eceea26619c | d186b8c721c0c207 | eada7dd6cde0eb1e | f57d4f7fee6ed178 |
| 06f067aa72176fba | 0a637dc5a2c898a6 | 113f9804bef90dae | 1b710b35131c471b |
| 28db77f523047d84 | 32caab7b40c72493 | 3c9ebe0a15c9bebc | 431d67c49c100d4c |
| 4cc5d4becb3e42b6 | 597f299cfc657e2a | 5fcb6fab3ad6faec | 6c44198c4a475817 |

# SINGLE ROUND SHA-512 ELEMENTARY OPERATION



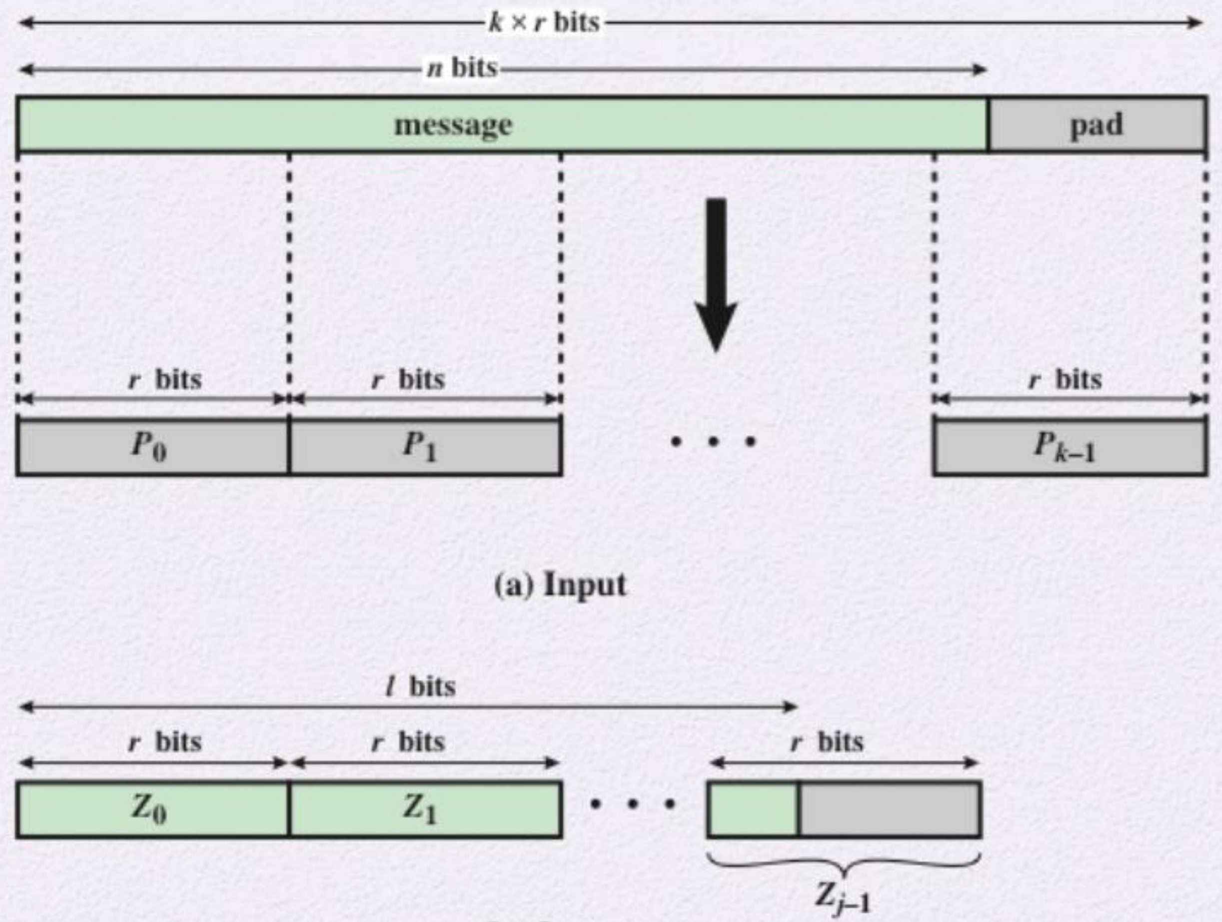
# SHA-3

- SHA-1 has not yet been "broken”
- No one has demonstrated a technique for producing collisions in a practical amount of time
- Considered to be insecure and has been phased out for SHA-2
- SHA-2 shares the same structure and mathematical operations as its predecessors so this is a cause for concern
- Because it will take years to find a suitable replacement for SHA-2 should it become vulnerable, NIST decided to begin the process of developing a new hash standard
- NIST announced in 2007 a competition for the SHA-3 next generation NIST hash function • Winning design was announced by NIST in October 2012
- SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications

# THE SPONGE CONSTRUCTION

- Underlying structure of SHA-3 is a scheme referred to by its designers as a *sponge construction*
- Takes an input message and partitions it into fixed-size blocks
- Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block
- The sponge function is defined by three parameters:
  - f = the internal function used to process each input block
  - r = the size in bits of the input blocks, called the *bitrate*
  - pad = the padding algorithm

# THE SPONGE FUNCTION – INPUT AND OUTPUT



(a) Input

(b) Output

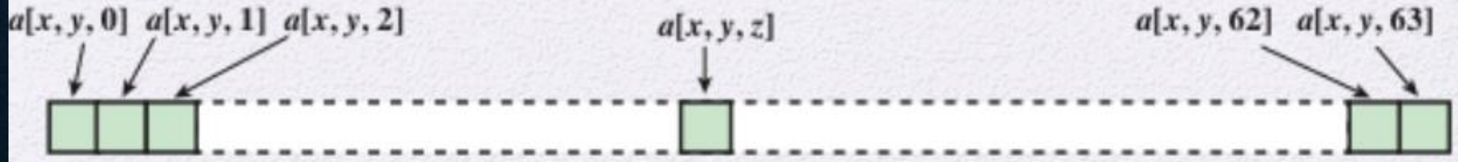
# SHA-3 PARAMETERS

| | | | | |
|---|---|---|---|---|
| **Message Digest Size** | 224 | 256 | 384 | 512 |
| **Message Size** | no maximum | no maximum | no maximum | no maximum |
| **Block Size (bitrate *r*)** | 1152 | 1088 | 832 | 576 |
| **Word Size** | 64 | 64 | 64 | 64 |
| **Number of Rounds** | 24 | 24 | 24 | 24 |
| **Capacity *c*** | 448 | 512 | 768 | 1024 |
| **Collision resistance** | $2^{112}$ | $2^{128}$ | $2^{192}$ | $2^{256}$ |
| **Second preimage resistance** | $2^{224}$ | $2^{256}$ | $2^{384}$ | $2^{512}$ |

# SHA-3 STATE MATRIX

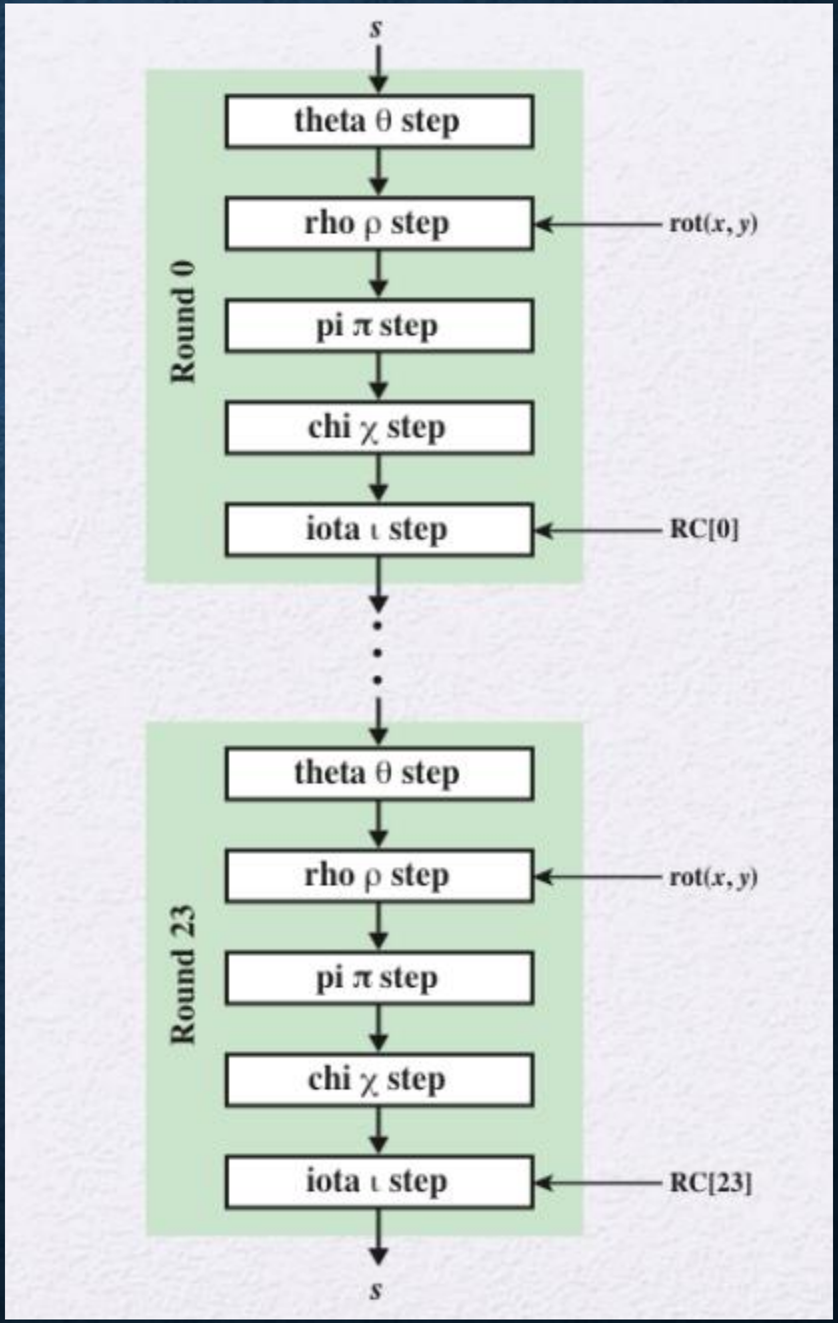
| | x = 0 | x = 1 | x = 2 | x = 3 | x = 4 |
|---|---|---|---|---|---|
| y = 4 | L[0, 4] | L[1, 4] | L[2, 4] | L[3, 4] | L[4, 4] |
| y = 3 | L[0, 3] | L[1, 3] | L[2, 3] | L[3, 3] | L[4, 3] |
| y = 2 | L[0, 2] | L[1, 2] | L[2, 2] | L[3, 2] | L[4, 2] |
| y = 1 | L[0, 1] | L[1, 1] | L[2, 1] | L[4, 1] | L[4, 1] |
| y = 0 | L[0, 0] | L[1, 0] | L[2, 0] | L[3, 0] | L[4, 0] |

(a) State variable as 5 × 5 matrix A of 64-bit words




(b) Bit labeling of 64-bit words

# SHA-3 ITERATION FUNCTION



# THETA AND CHI STEPS

| | x = 0 | x = 1 | x = 2 | x = 3 | x = 4 |
|---|---|---|---|---|---|
| y = 4 | L[0, 4] | L[1, 4] | L[2, 4] | L[3, 4] | L[4, 4] |
| y = 3 | L[0, 3] | L[1, 3] | L[2, 3] | L[3, 3] | L[4, 3] |
| y = 2 | L[0, 2] | L[1, 2] | L[2, 2] | L[3, 2] | L[4, 2] |
| y = 1 | L[0, 1] | L[1, 1] | L[2, 1] | L[4, 1] | L[4, 1] |
| y = 0 | L[0, 0] | L[1, 0] | L[2, 0] | L[3, 0] | L[4, 0] |

$L[2, 3] \leftarrow C[1] \oplus L[2, 3] \oplus ROT(C[3], 1)$

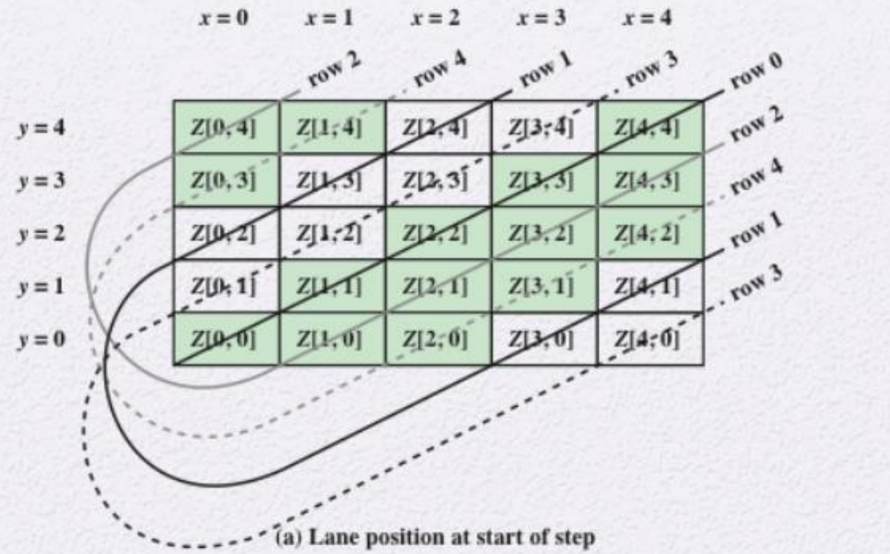
(a) θ step function

| | x = 0 | x = 1 | x = 2 | x = 3 | x = 4 |
|---|---|---|---|---|---|
| y = 4 | L[0, 4] | L[1, 4] | L[2, 4] | L[3, 4] | L[4, 4] |
| y = 3 | L[0, 3] | L[1, 3] | L[2, 3] | L[3, 3] | L[4, 3] |
| y = 2 | L[0, 2] | L[1, 2] | L[2, 2] | L[3, 2] | L[4, 2] |
| y = 1 | L[0, 1] | L[1, 1] | L[2, 1] | L[4, 1] | L[4, 1] |
| y = 0 | L[0, 0] | L[1, 0] | L[2, 0] | L[3, 0] | L[4, 0] |

$L[2, 3] \leftarrow L[2, 3] \oplus \overline{L[3, 3]} \text{ AND } L[4, 3]$

(b) χ step function

# PI STEP FUNCTION




(a) Lane position at start of step

| | x = 0 | x = 1 | x = 2 | x = 3 | x = 4 |
|---|---|---|---|---|---|
| y = 4 | Z[2, 0] | Z[3, 1] | Z[4, 2] | Z[0, 3] | Z[1, 4] |
| y = 3 | Z[4, 0] | Z[0, 1] | Z[1, 2] | Z[2, 3] | Z[3, 4] |
| y = 2 | Z[1, 0] | Z[2, 1] | Z[3, 2] | Z[4, 3] | Z[0, 4] |
| y = 1 | Z[3, 0] | Z[4, 1] | Z[0, 2] | Z[1, 3] | Z[2, 4] |
| y = 0 | Z[0, 0] | Z[1, 1] | Z[2, 2] | Z[3, 3] | Z[4, 4] |

(b) Lane position after permutation

# SUMMARY

- Applications of cryptographic hash functions
  - Message authentication
  - Digital signatures
  - Other applications
- Requirements and security
  - Security requirements for cryptographic hash functions
  - Brute-force attacks
  - Cryptanalysis
- Hash functions based on cipher block chaining
- Secure hash algorithm (SHA)
  - SHA-512 logic
  - SHA-512 round function
- SHA-3
  - The sponge construction
  - The SHA-3 Iteration Function *f*