Passwords, Logins, and CHF

Ashkan Jangodaz
British Columbia Institute of Technology

The Real Problem

- Most security breaches don't involve sophisticated hacking.
- Over 80% of breaches involve stolen or weak credentials. [1]
  - If someone has your password, they can simply *be* you.
- Authentication answers one question:
  - *Who are you?*
- The problem:
  - Secrets can be copied, shared, stolen, or guessed.

Factors of Authentication

Factor	Description	Examples
Something you know	Secret in your memory	Password, PIN
Something you have	Physical object you possess	Phone, hardware token
Something you are	Biometric characteristic	Fingerprint, face

How Should Systems Store Passwords?

- **Wrong:** Store passwords in plain text
  - Database stolen → all passwords immediately exposed
- **Wrong:** Encrypt passwords
  - Encryption is reversible
  - If attacker gets the key, all passwords exposed
- **Correct:** Store password *hashes*
  - One-way transformation
  - Cannot be reversed
  - System never needs to store actual password

Password Verification

- **When you set a password:**
  - **You type:** MySecretPassword
  - **System computes:** hash("MySecretPassword") → a7f3b2c9...
  - **System stores:** a7f3b2c9...
  - Original password discarded
- **When you log in:**
  - You type: MySecretPassword
  - System computes hash of what you typed
  - Compares to stored hash
  - Match → access granted

What Linux Actually Stores

File	Purpose	Readable By
`/etc/passwd`	Identity information (username, UID, home, shell)	Everyone
`/etc/shadow`	Password verification data (hashes)	Root only

Anatomy of /etc/shadow

- alice:yj9T$7R3xK9pL$WvM8nQ2kF5hJ1xY4bN6cR9dG3mP7sT0wA2zX8vB4:19750:0:99999:7:30:20089:

Field	Value	Meaning
Username	`alice`	Account name
Password hash	`$y$j9T$7R3xK9pL$WvM8nQ2k...`	yescrypt hash with salt
Last changed	`19750`	Days since Jan 1, 1970
Min age	`0`	Can change password anytime
Max age	`99999`	Password never expires (~274 years)
Warn period	`7`	Warn user 7 days before expiry
Inactive	`30`	Disable account 30 days after expiry
Expire date	`20089`	Account expires on Jan 1, 2025

The Hash Format

- idsalt$hash

Component	Purpose
`$id$`	Identifies the hashing algorithm
`salt`	Random value unique to this password
`hash`	The actual hash output

- **Common identifiers:**
  - \$1\$ → MD5crypt (legacy, weak)
  - \$5\$ → SHA-256crypt
  - \$6\$ → SHA-512crypt
  - \$y\$ → yescrypt (modern Linux default)
  - \$2a\$, \$2y\$ → bcrypt

What is a Salt?

- **Problem without salt:**
  - Two users with password "password123" get identical hashes
  - Attacker can precompute hashes for common passwords (rainbow tables)
  - Attacker can see which users have the same password
- **With salt:**
  - Each password gets a unique random salt
  - Same password + different salts → different hashes
  - Precomputation becomes useless

Important About Salts

- Salts are stored alongside the hash
  - Visible in /etc/shadow
  - This is intentional, system needs salt to verify passwords
- Salts prevent pre-computation attacks (rainbow tables)
- Salts do NOT prevent brute-force attacks
- Attacker with hash file has all the salts too
- **Salts ensure each password must be cracked independently.**

Dictionary Attacks

- Attackers don't start with "aaaaaa" and work up, they start with lists of known passwords:
  - Passwords from previous breaches
  - Common words and phrases
  - Keyboard patterns (qwerty, 123456)
  - Common substitutions (p@ssw0rd, passw0rd)
- If your password is in a wordlist, length doesn't matter.

Lab

- Separation between /etc/passwd and /etc/shadow
- Stored value is verification material, not the password
- If dictionary cracking succeeds quickly, weakness is predictability
- Dictionary attacks exploit human choice
- Brute force exploits small search space
- Strong rules don't help if password is predictable

Lab

- Create a test account with a password you choose
- Examine how Linux stores identity vs. password data
- Extract a hash and understand its structure
- Attempt dictionary cracking on your own password
- Compare dictionary attack to brute force
- Clean up completely when finished

Hash Functions

- A **hash function H** accepts a variable-length **block of data M** as input and produces a **fixed-size hash value (hash code) h**
  - h = 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)

Hash Functions

- A change to any bit or bits in **M** results, with high probability, in a change to the hash value **h**
  - Hash functions are often used to determine whether or not data has changed
- Cryptographic hash functions are **keyless** cryptographic algorithms
  - There are however some “keyed” hash functions (e.g. MAC)

Hash Functions

- Typically, the input is **padded** out to an integer multiple of some fixed length (e.g., 1024 bits)
  - Padding includes the value of the length of the original message in bits.
  - The **length field** is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value

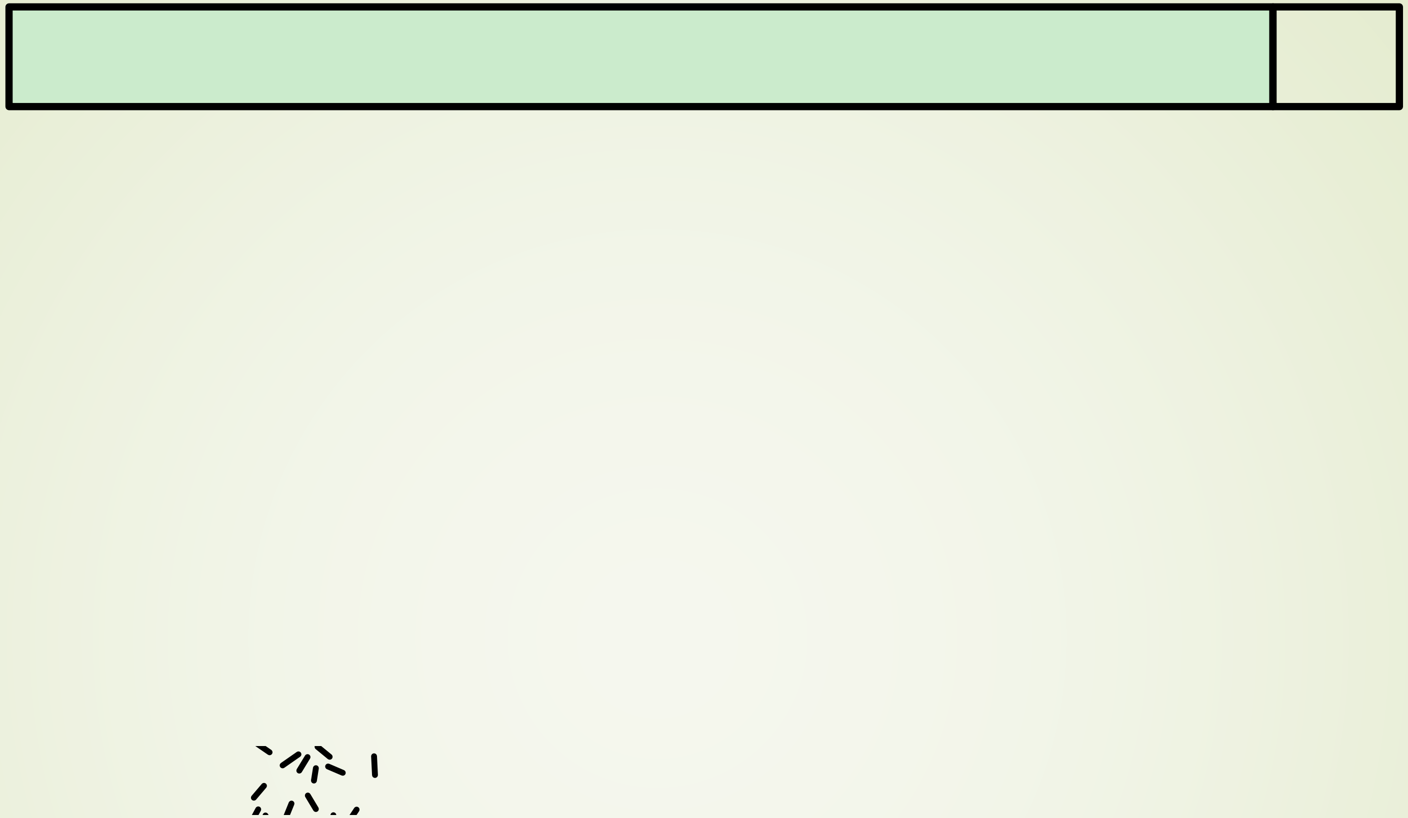


Figure 11.1 Cryptographic Hash Function; h = H(*M*)

Hash Functions Applications

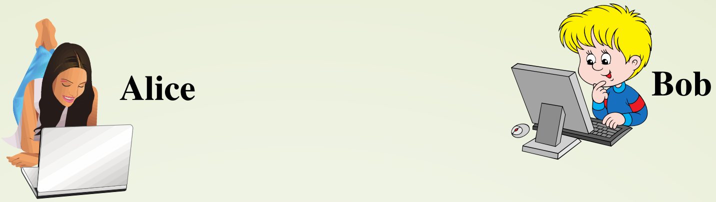
- **Message Authentication**
  - A mechanism or service used to verify the **integrity** of a message
  - Message authentication assures that data received are exactly as sent (i.e., there is no modification, insertion, deletion, or replay)
  - The hash value is often referred to as a **message digest**
  - Is achieved using a Message Authentication code (MAC)

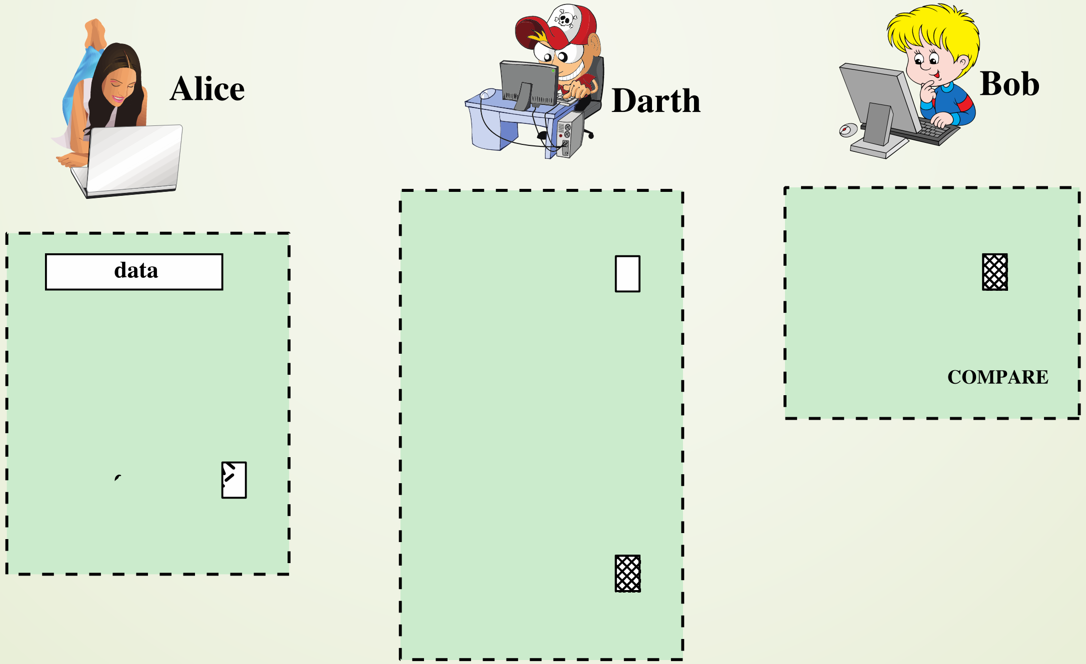
Message Authentication Code (MAC)

- 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





(b) Man-in-the-middle attack

Figure 11.2 Attack Against Hash Function

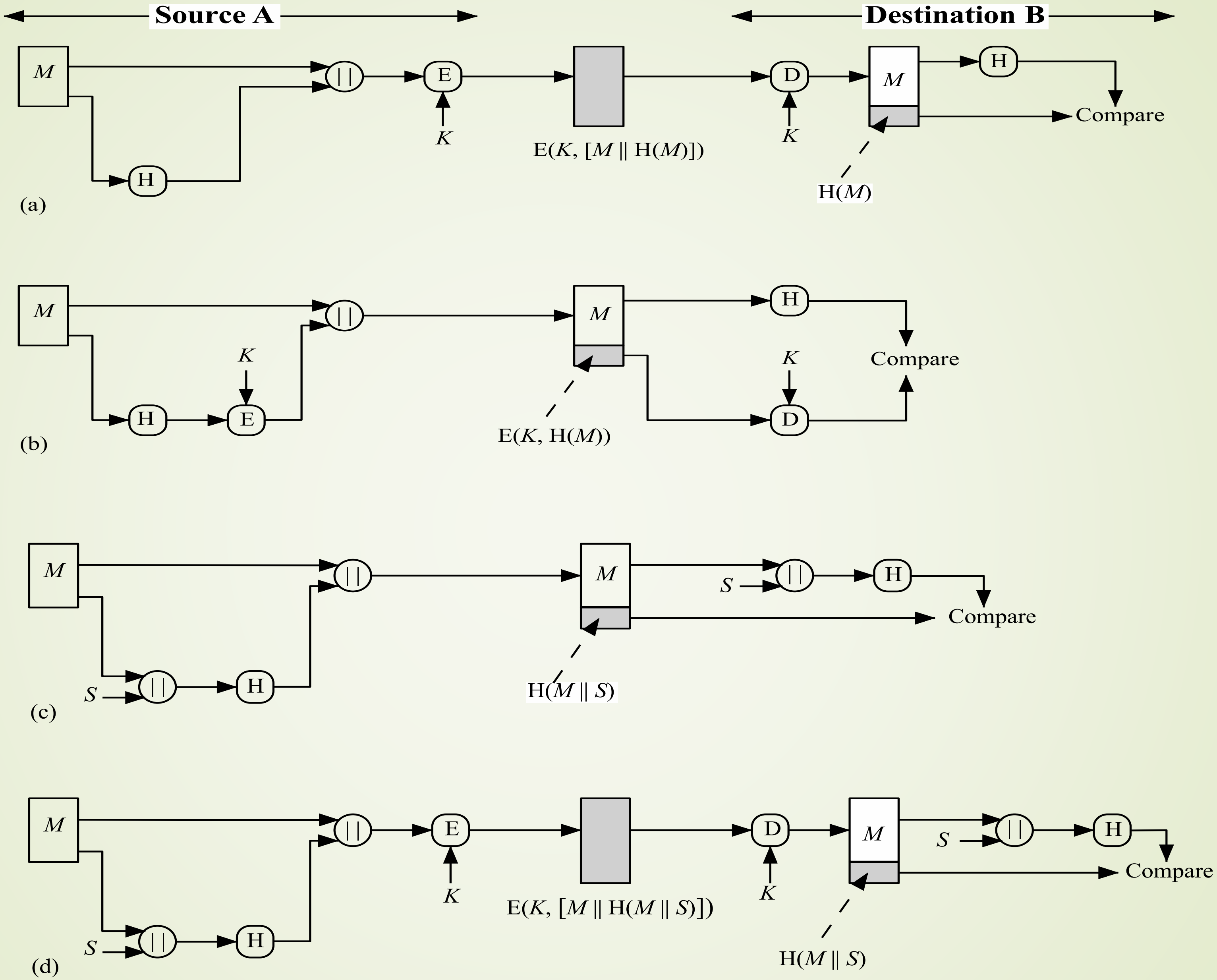


Figure 11.3 Simplified Example of the use of a Hash Function for Message Authentication

Hash Functions Applications

- **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

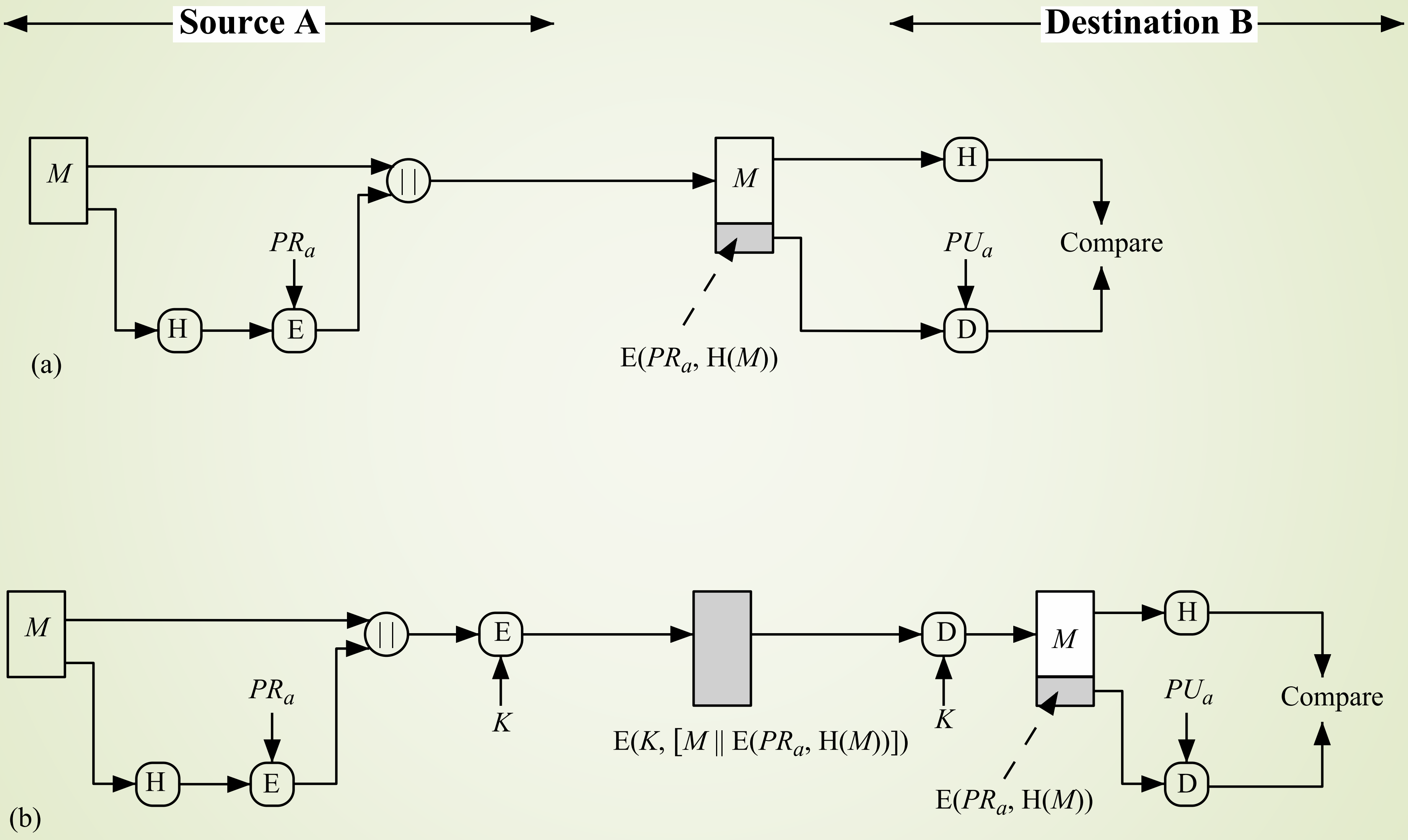


Figure 11.4 Simplified Examples of Digital Signatures

Other Hash Functions Applications

- 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)**

Other Hash Functions Applications

- 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 (message, file, etc.) 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

1. Bit-by-bit exclusive-OR (XOR) of every block
   - $C_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im}$
   - Produces a simple parity for each bit position and is known as a longitudinal redundancy check (p. 13 to 15)
   - Reasonably **effective for random data** as a data integrity check

Two Simple Hash Functions

- The probability that a data error will result in an unchanged hash value is $\mathbf{2^{-n}}$
  - With more predictably formatted data, the function is less effective
  - For example, in most normal text files, the high-order bit of each octet is always zero
    - if a **128-bit** hash value is used, instead of an effectiveness of $\mathbf{2^{-128}}$, the hash function on this type of data has an effectiveness of $\mathbf{2^{-112}}$
    - ASCII for 'H': 01001000, for 'i': 01101001, and for '!': 00100001

Two Simple Hash Functions

2. A simple way to improve matters is to perform a one-bit circular shift, or rotation, on the hash value after each block is processed.
   - Initially set the n-bit hash value to zero.
   - Process each successive n-bit block of data as follows:
     1. Rotate the current hash value to the left by one bit.
     2. XOR the block into the hash value.

- This has the effect of “randomizing” the input more completely and overcoming any regularities that appear in the input

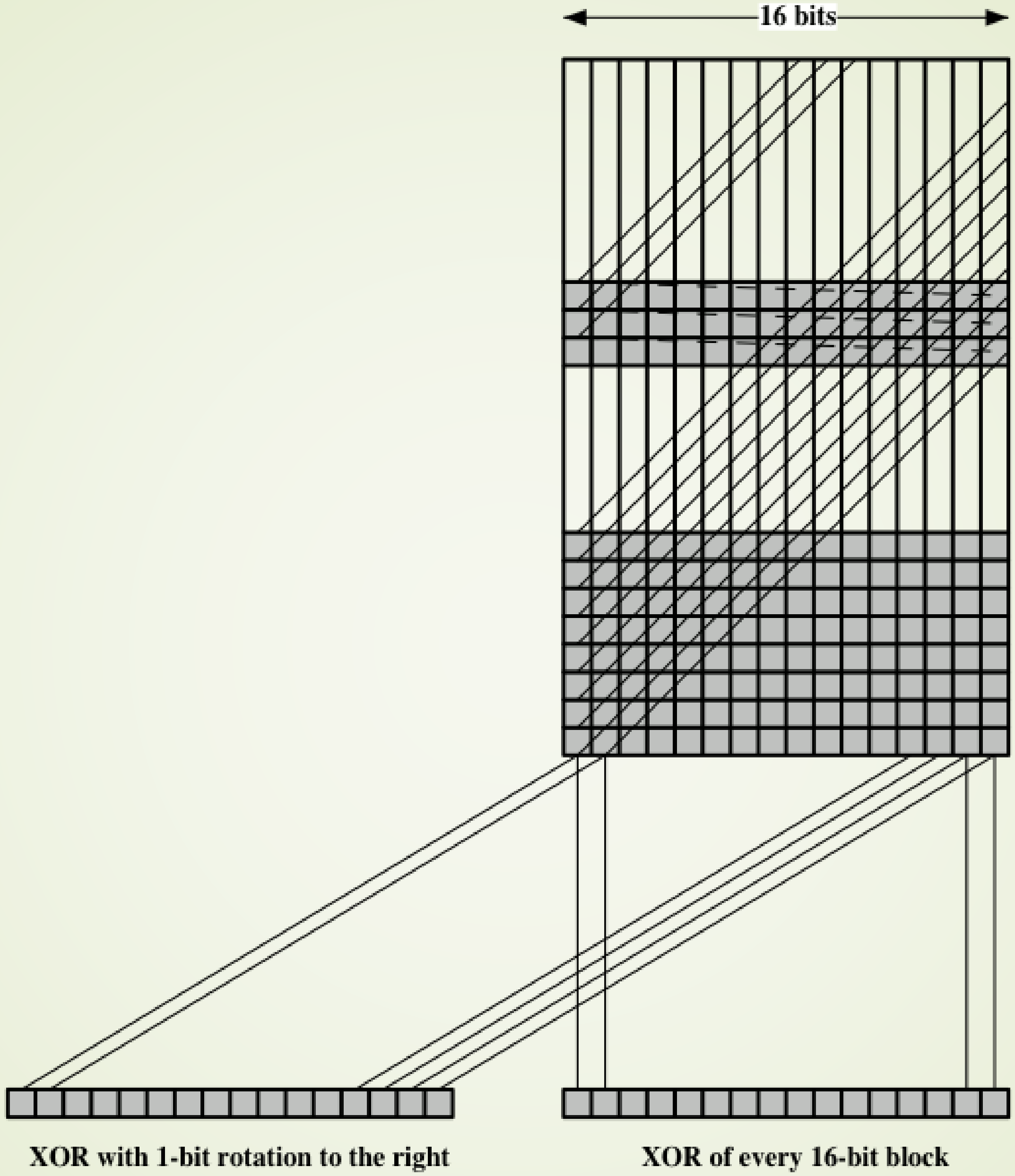


Figure 11.5 Two Simple Hash Functions (16-bit hash values)

Two Simple Hash Functions

- Although the second procedure provides a good measure of data integrity, it is virtually useless for data security when an encrypted hash code is used with a plaintext message
  - **Figures 11.3 (b) and 11.4 (a)**
- Given a message, it is an easy matter to produce a new message that yields that hash code
- A simple XOR or rotated XOR (RXOR) is insufficient if only the hash code is encrypted
  - Even if the entire message is encrypted, they are not totally sufficient (p. 345 and 346 of the textbook)

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 ≠ y** and **H(x) = H(y)**
  - Because we are using hash functions for data integrity, collisions are clearly undesirable

Number of Potential Collisions

- Suppose the length of the hash code is **n** bits
- The function **H** takes as input messages or data blocks of length **b** bits
  - **b > n**
  - The total number of possible messages is $\mathbf{2^b}$ and the total number of possible hash values is $\mathbf{2^n}$
  - On average, each hash value corresponds to $\mathbf{2^{b-n}}$ preimages
  - If **H** tends to uniformly distribute hash values, each hash value will have close to $\mathbf{2^{b-n}}$ preimages
- Now imagine **b** to be variable size!

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

Table 11.1 Requirements for a Cryptographic Hash Function H

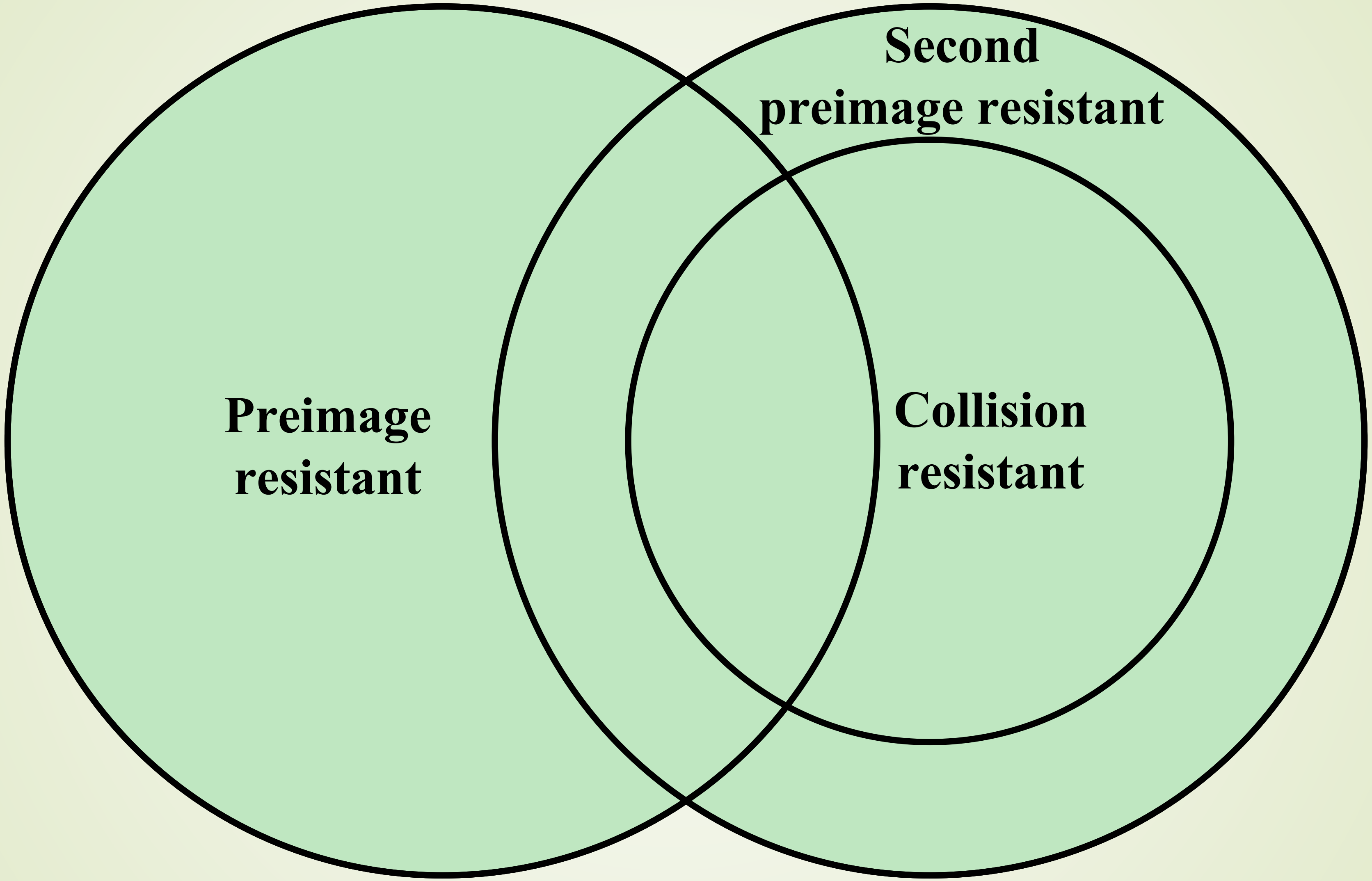


Figure 11.6 Relationship Among Hash Function Properties

	Preimage Resistant	Second Preimage Resistant	Collision Resistant
Hash + digital signature	yes	yes	yes*
Intrusion detection and virus detection		yes	
Hash + symmetric encryption			
One-way password file	yes		
MAC	yes	yes	yes*

* Resistance required if attacker is able to mount a chosen message attack

Table 11.2 Hash Function Resistance Properties Required for Various Data Integrity Applications

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

Preimage and Second Preimage Attacks

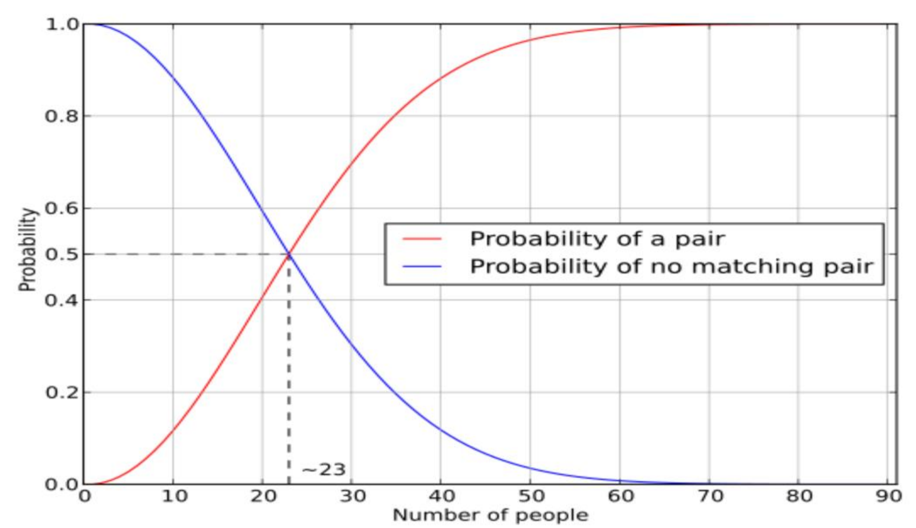
- Adversary wishes to find a value **y** such that **H(y)** is equal to a given hash value **h**
- The brute-force method is to pick values of **y** at random and try each value until a collision occurs
  - For an **m-bit** hash value, the level of effort is proportional to $\mathbf{2^m}$
  - The adversary would have to try, on average, $\mathbf{2^{m-1}}$ values of **y** to find one that generates a given hash value **h** Appendix E [Equation (E.1)]

Collision Resistant Attacks

- Adversary wishes to find two messages or data blocks, **x** and **y**, that yield the same hash function: **H(x) = H(y)**
  - The effort required is explained by a mathematical result referred to as the birthday paradox
- Require considerably less effort than a preimage or second preimage attack
  - for an **m-bi**t hash value, **if we pick data blocks at random,** we can expect to find two data blocks with the same hash value within $\sqrt{2^m} = 2^{m/2}$ Appendix E

Birthday Paradox

- The probability that, in a set of n randomly chosen people, at least two will share a birthday
  - Only 23 people are needed for that probability to exceed 50% (!)
  - If you are interested, read the calculations in Wikipedia
  - Also read this: Birthday Attack



Merkle – Damgard Structure

- Is the structure of most hash functions in use today, including SHA
- The hash function takes an input message and partitions it into **L fixed-sized blocks of b** bits each
  - If necessary, the final block is padded to b bits
  - The final block also includes the value of the total length of the input to the hash function
- The hash algorithm involves repeated use of a compression function, **f**
  - Takes two inputs (an **n-bit** input from the previous step, called the chaining variable and a **b-bit** block) and produces an **n-bit** output

Merkle – Damgard Structure

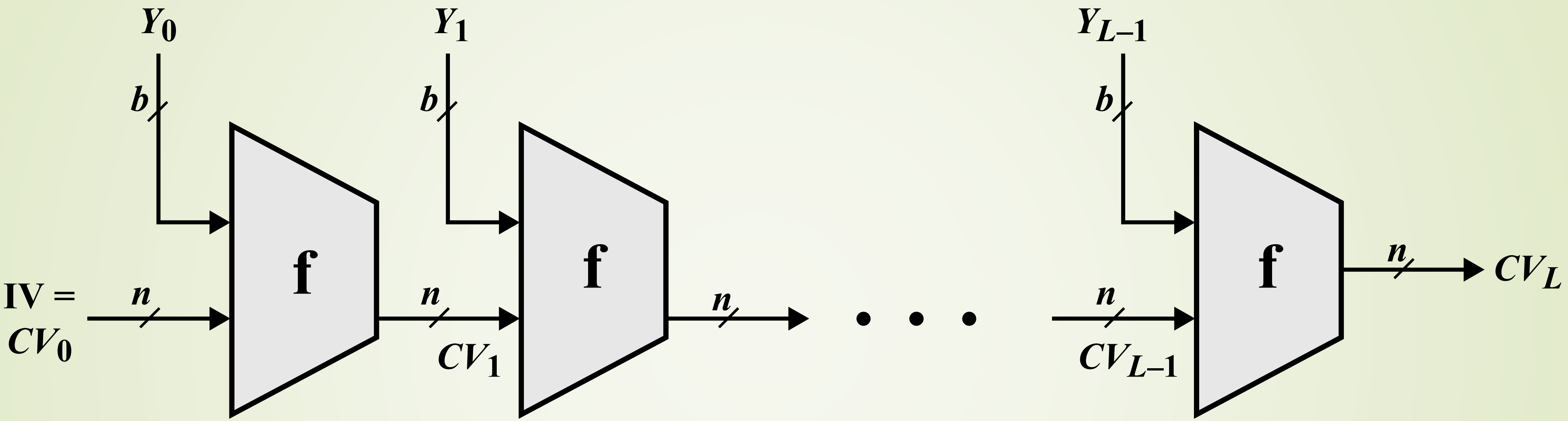
- The final value of the chaining variable is the hash value:

$$\mathbf{CV_0 = IV = \text{initial n-bit value}}$$

$$\mathbf{CV_i = f(CV_{i-1}, Y_{i-1})\ 1 \le i \le L}$$

$$\mathbf{H(M) = CV_L}$$

- The input to the hash function is a message **M** consisting of the blocks $\mathbf{Y_0, Y_1, \dots , Y_{L-1}}$



IV	=	Initial value	L	=	number of input blocks
CV_i	=	chaining variable	n	=	length of hash code
Y_i	=	ith input block	b	=	length of input block
f	=	compression algorithm			

Figure 11.8 General Structure of Secure Hash Code – Merkle-Damgard

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 – **SHA-0**
- Was revised in 1995 as **SHA-1**
- Based on the hash function **MD4** and its design closely models **MD4**
- **SHA-1** 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**

Secure Hash Algorithm (SHA)

- The most widely used hash function has been the Secure Hash Algorithm (SHA)
- A collision was found in **SHA-1** in 2017
- Microsoft, Google, Apple, and Mozilla have all announced that their respective browsers have stopped accepting SHA-1 SSL certificates in 2017

Algorithm	Message Size	Block Size	Word Size	Message Digest Size
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

* All sizes are measured in bits.

Table 11.3 Comparison of SHA Parameters

A SHA-256 Example

- Ashkan:
  - 03cf25a5df5694becfd17663f774dd28e51d36c4ed4110373ac21a64508dcc2a
- ashkan:
  - e43adf78c5327baeb591e846de459826765963f82e41804f3cbfeefcf706c0c1
- This Lecture PDF File:
  - Try yourself!

Optional (FYI) Slides

For a Complete SHA-512 Implementation refer to: [p. 353 to 361] of the Textbook

SHA-512

- The algorithm takes as input a **message with a maximum length** of less than $\mathbf{2^{128}}$ bits and produces as **output a 512-bit message digest**

1. The input is processed in **1024-bit** blocks
   - The message is padded so that its length is congruent to 896 modulo 1024 [$\text{length} \equiv 896 (\text{mod } 1024)$]
   - Padding is always added
   - Number of padding bits is in the range of 1 to 1024
   - Padding consists of a single 1 bit followed by the necessary number of 0 bits

SHA-512

2. A block of 128 bits is appended to the message
   - An unsigned 128-bit integer (most significant byte first)
   - contains the length of the original message in bits (before the padding)
3. A 512-bit buffer is used to hold intermediate and final results of the hash function
   - The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h)

a = 6A09E667F3BCC908	e = 510E527FADE682D1
b = BB67AE8584CAA73B	f = 9B05688C2B3E6C1F
c = 3C6EF372FE94F82B	g = 1F83D9ABFB41BD6B
d = A54FF53A5F1D36F1	h = 5BE0CD19137E2179

SHA-512

4. The heart of the algorithm is a module that consists of **80 rounds**
   - Process message in 1024-bit (128-byte) blocks
   - Each round takes as input the 512-bit buffer value (abcdefgh)
   - Each round t makes use of a 64-bit value W_t, derived from the current 1024-bit block being processed (M_i)

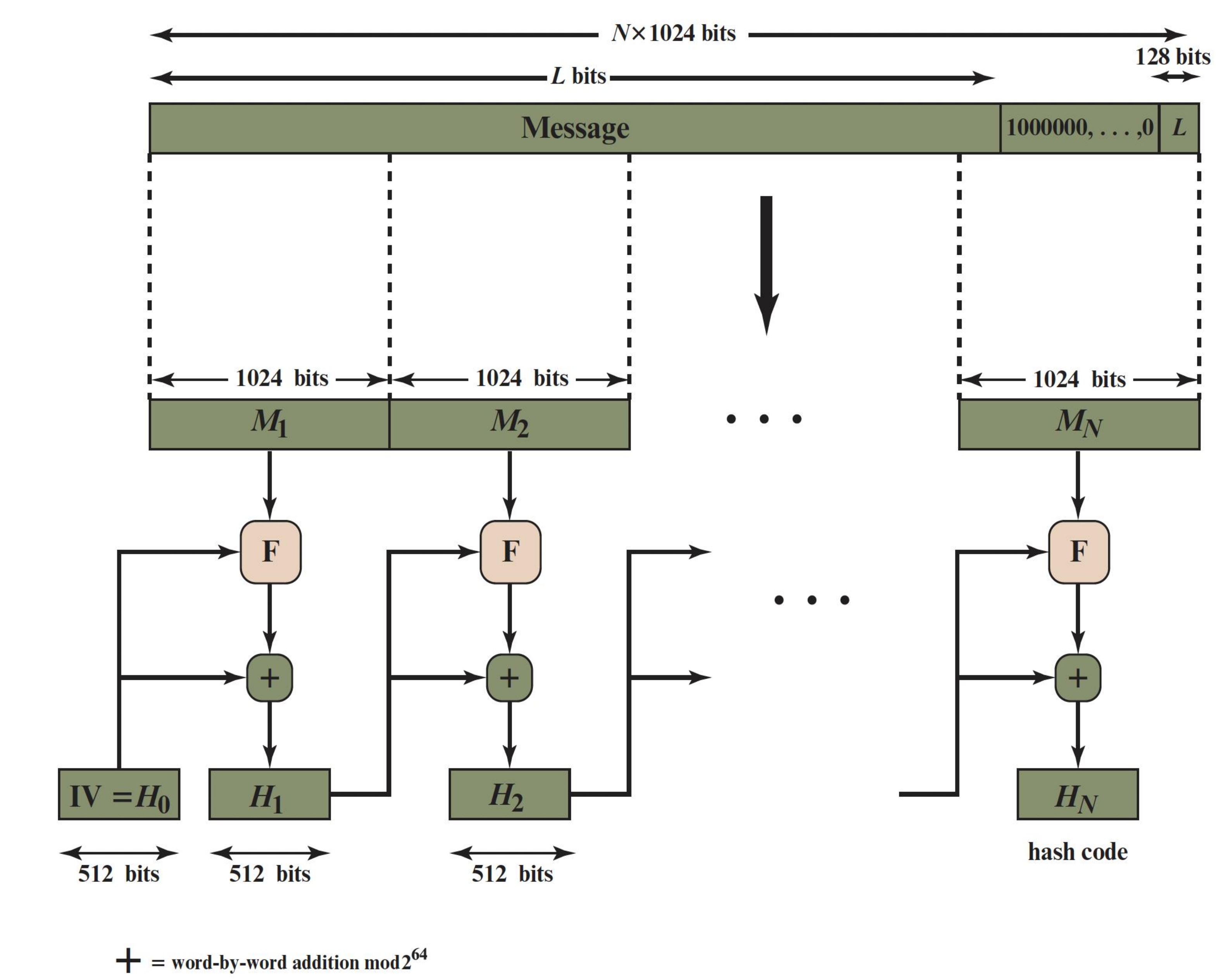


Figure 11.9 Message Digest Generation Using SHA-512

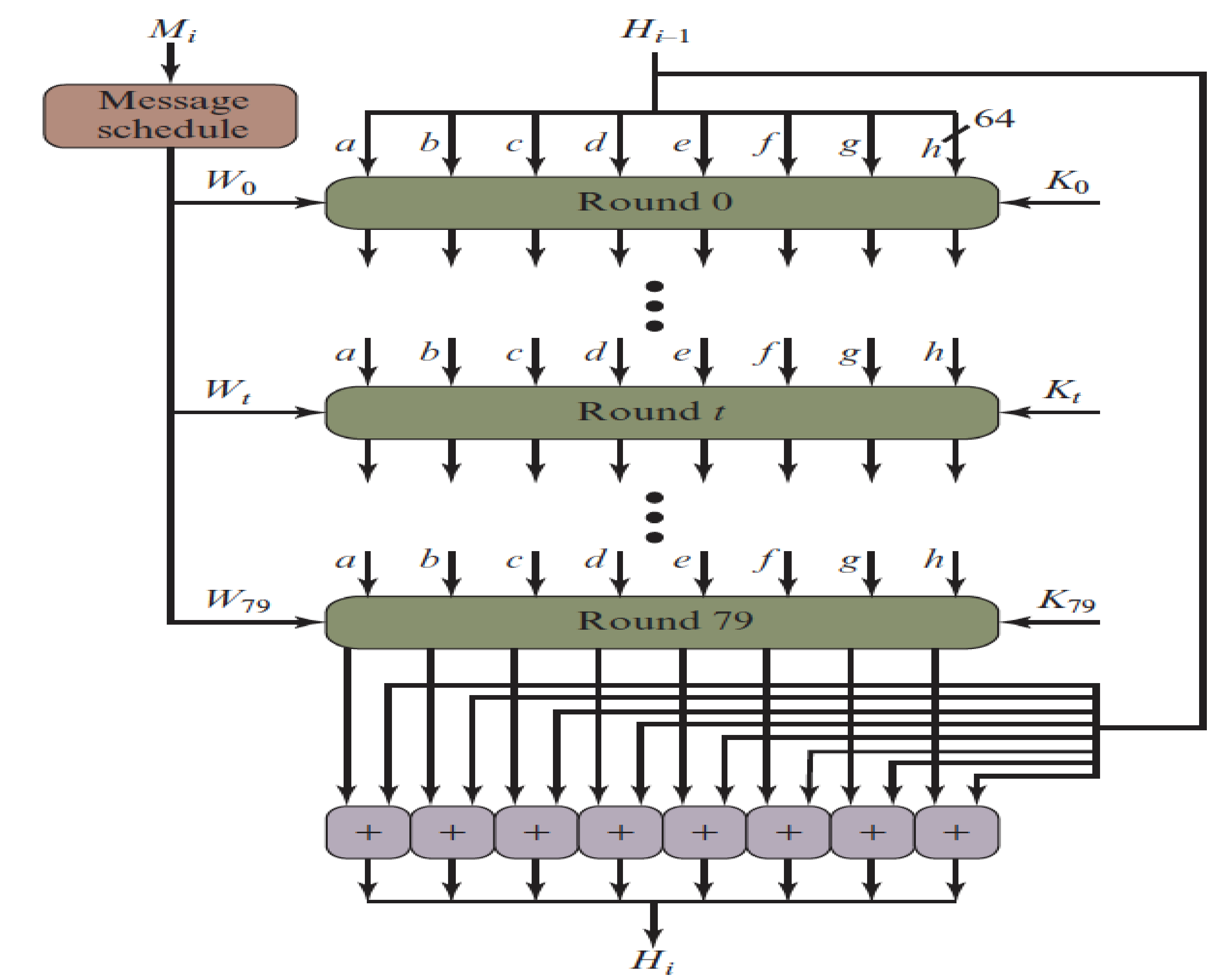


Figure 11.10 SHA-512 Processing of a Single 1024-Bit Block

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

Table 11.4 SHA-512 Constants

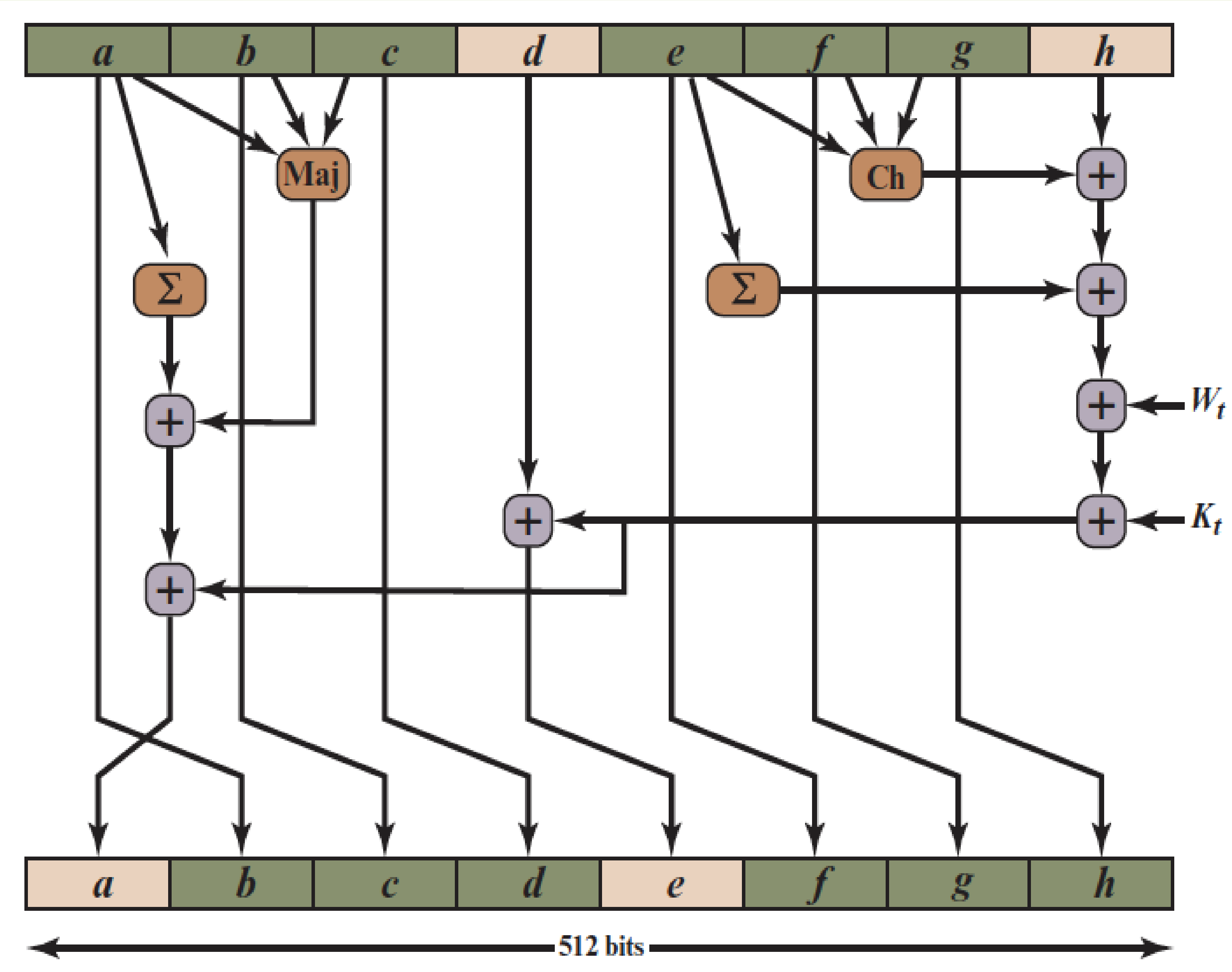


Figure 11.11 Elementary SHA-512 Operation (single round)

Real Life Example of SHA

- SHA-256 plays a fundamental role in blockchain technology
  - It generates the hash values of blocks in a blockchain, ensuring the immutability and integrity of the entire chain
- SHA-256 is the hash function and mining algorithm of the **Bitcoin** protocol
- Also used for transaction verification
- Bitcoin uses double SHA-256, meaning that it applies the hash functions twice
- https://github.com/bitcoin/bitcoin

Summary

- Applications of cryptographic hash functions
- Hash function used for Message Authentication needs to be secured
- Differences among preimage resistant, second preimage resistant, and collision resistant properties
- The basic structure of cryptographic hash functions

References

1. William Stallings: Cryptography and Network Security. Eight Edition 2020. ISBN-13: 9780135763971 Pearson
2. Longitudinal Redundancy Check (LRC)/2-D Parity Check - Geeksforgeeks
3. The City University of New York
4. Birthday Paradox - University of Regina
5. The Birthday Paradox - Harvard CS125
6. The Birthday Paradox and Collisions in a Hash table – Medium - Arish Sateesan
7. Merkle-Damg˚ard Revisited: how to Construct a Hash Function