Cryptography

Cryptographic Data Integrity Algorithms

M S Vilkhu

23 Nov 2024(C1/C3/C5)

Topic to cover

- Data Integrity: Applications of Cryptographic Hash Functions, Message Authentication Codes, Digital Signatures, Birthday Paradox
- Digital Signatures, Key Management and Distribution, PKI

Topics

CRYPTOGRAPHIC HASH FUNCTIONS

11.1 Applications of Cryptographic Hash Functions

Message Authentication
Digital Signatures
Other Applications

11.2 Two Simple Hash Functions

11.3 Requirements and Security

Security Requirements for Cryptographic Hash Functions
Brute-Force Attacks
Cryptanalysis

11.4 Hash Functions Based on Cipher Block Chaining

11.5 Secure Hash Algorithm (SHA)

SHA-512 Logic
SHA-512 Round Function
Example

11.6 SHA-3

The Sponge Construction
The SHA-3 Iteration Function *f*

Learning Objectives

After studying this chapter, you should be able to:

◆ Summarize the applications of cryptographic hash functions.

◆ Explain why a hash function used for message authentication needs to be secured.

◆ Understand the differences among preimage resistant, second preimage resistant, and collision resistant properties.

◆ Present an overview of the basic structure of cryptographic hash functions.

◆ Describe how cipher block chaining can be used to construct a hash function.

◆ Understand the operation of SHA-512.

◆ Understand the birthday paradox and present an overview of the birthday attack.

Introduction

- **A hash function** H accepts a **variable-length block** of data M as input and produces a **fixed-size hash value h = H(M).**
- **A "good" hash function** has the property that the **results of applying the function** to a **large set of inputs** will produce **outputs that are evenly distributed and apparently random.**
- the **principal object** of a **hash function is data integrity**. A change to any bit or bits in M results, with high probability, in a change to the hash value.

Introduction

- The **kind of hash function** needed **for security applications** is referred to as a **cryptographic hash function**.
- A cryptographic hash function is an algorithm for which it is **computationally <u>infeasible</u>** (**because no attack is significantly more efficient than brute force**) to find either

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

  (b) **two data objects that map** to the **same hash** result (the **collision-free property**).
- hash functions are often used to determine **whether or not data has changed**.

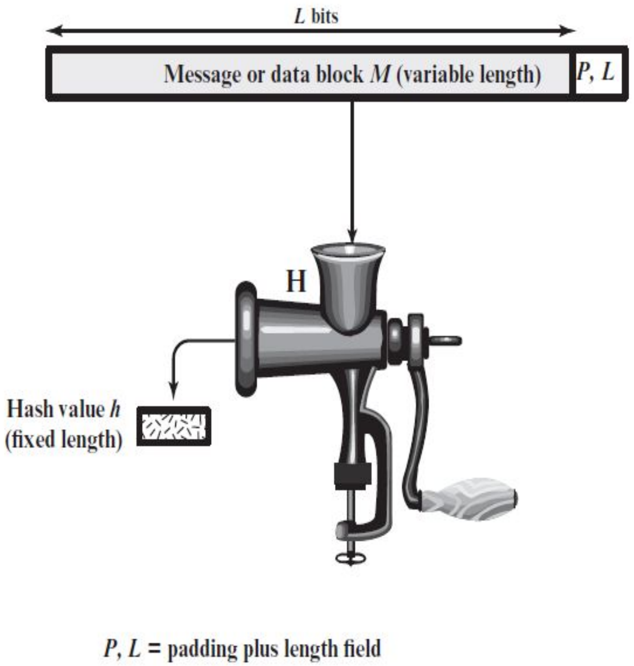


Figure 11.1 Cryptographic Hash Function; $h = H(M)$

Broad Topic

- Discuss wide variety of **applications for Cryptographic hash functions**.
- Next, we look at the **security requirements for such Functions**.
- Then we look at the **use of cipher block chaining to implement a cryptographic hash function.**
- Finally look at most important and widely used family of cryptographic hash functions, the **Secure Hash Algorithm (SHA) family**.
- **MD5**, a well-known cryptographic hash function with similarities to SHA-I.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- Widely used and versatile – Cryptographic Hash function
- **Applications**
  - **Message authentications**
  - **Digital Signature**
  - **Other applications**

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

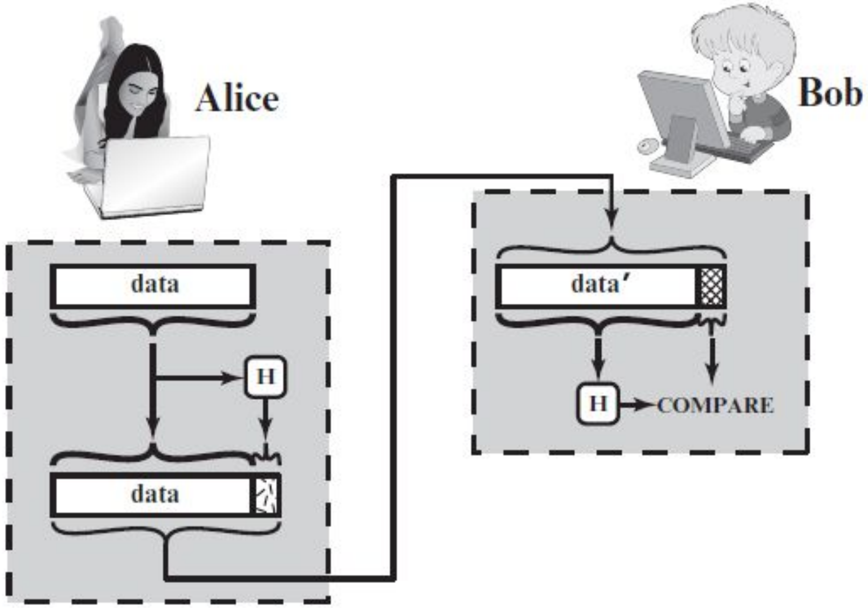
- **Message authentications**
- Message authentication is a mechanism or service used to **verify the integrity of a message.**
- Message authentication assures that **data received are exactly as sent** (i.e., there is no **modification, insertion, deletion, or replay**).
- In many cases, there is a **requirement** that the authentication mechanism assures that purported **identity of the sender is valid**.
- When a hash function is used to **provide message authentication**, the hash function value is often referred to as a **message digest.**
- The **essence of the use of a hash function** for message **integrity** is as follows. …

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Message authentications**
- The **essence of the use of a hash function** for **message integrity** is as follows.
- The sender computes a hash value as a function of the bits in the message and transmits both the hash value and the message.
- The receiver performs the same hash calculation on the message bits and compares this value with the incoming hash value.
- If mismatch – then message is modified.
- The hash value must be transmitted in **a secure fashion**.
- It is likely someone intercepts the message and the hash, modifies the message and creates new hash sends with the message – this will fool the receiver.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Message authentications**



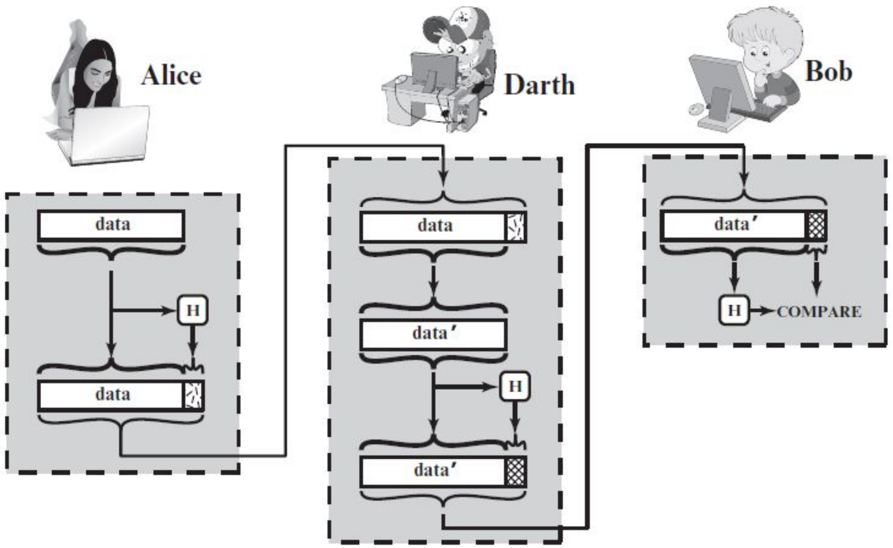
for message integrity is as follows

nction of the bits in the message and transmits

ılation

ecure f

Attack against Hash Function



(b) Man-in-the-middle attack

- It is likely someone intercepts the message and

creates new hash sends with the message – this will fool the receiver.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Message authentications**
- variety of ways **hash code can be used** to provide **message authentication**, as follows.

1. The **message plus concatenated hash code** is **encrypted** **using symmetric encryption**. Because only A and B share the secret key, the message must have come from A and has not been altered.
    - The **hash code** provides the **structure or redundancy required to achieve authentication.**
    - Because encryption is applied to the entire message plus hash code, **confidentiality** is also provided.
2. Only the **hash code is encrypted,** using **symmetric encryption**. This reduces the processing burden for those applications that **do not require confidentiality**.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Message authentications**
- variety of ways **hash code can be used** to provide **message authentication**, as follows.

3. It is possible to **use a hash function** but **no encryption** for **message authentication**. The technique **assumes that the two communicating parties share a common secret value S**. A computes the **hash value** over the **concatenation of M and S** and appends the resulting **hash value to M**. Because B possesses S, it can recompute the hash value to verify. Because the **secret value itself is not sent**, an opponent cannot **modify an intercepted message** and cannot **generate a false message**.

4. Confidentiality can be added to the approach of method (3) by **encrypting the entire message plus the hash code**.

When confidentiality is not required then encryption can be avoided because processing overheads and the cost of the hardware per node adds up.

Authentical is achieved by **Message authentication code** also known as **keyed hash function.**

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

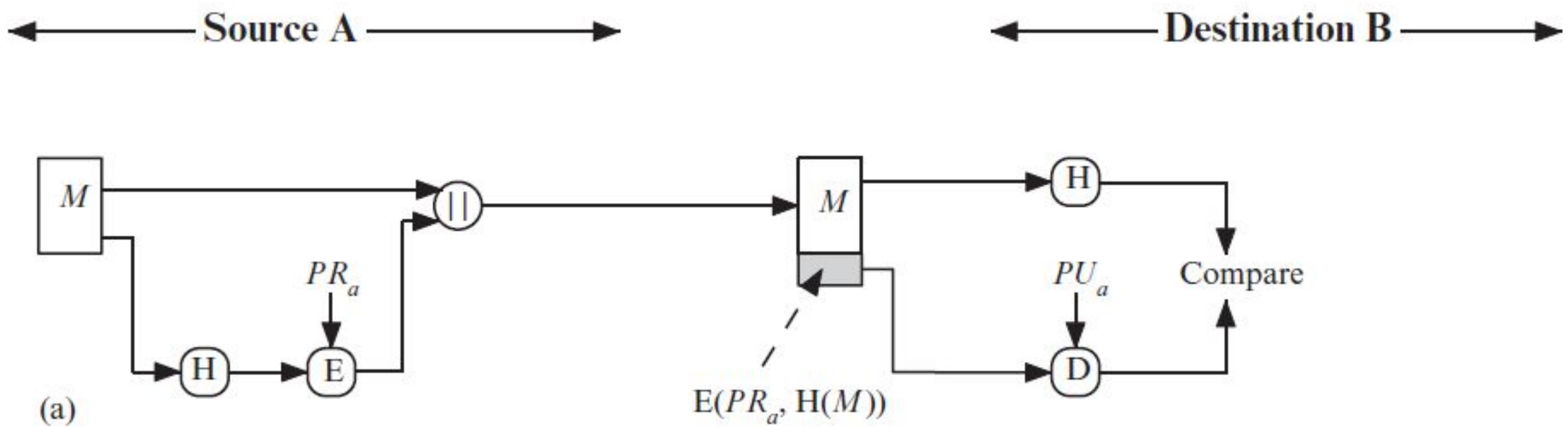
- **Message authentications**
- Authentical is achieved by **Message authentication code** also known as **keyed hash function.**
- Typically, **MACs are used between two parties** that **share a secret key** to **authenticate information exchanged** between those parties.
- A MAC function **takes as input a secret key** and a **data block** and **produces a hash value**, referred to as the 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**. Note that the **verifying party** also knows who the **sending party** is because **no one else knows the secret key**.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Digital Signatures**
- Another important application, which is similar to the message authentication application, is the **digital signature**.
- The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, 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** that is associated with the digital signature.
- In this case, an **attacker who wishes** to alter the message would need to know the user's private key.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Digital Signatures**
- Simplified version how **Hash is used for digital signature**.

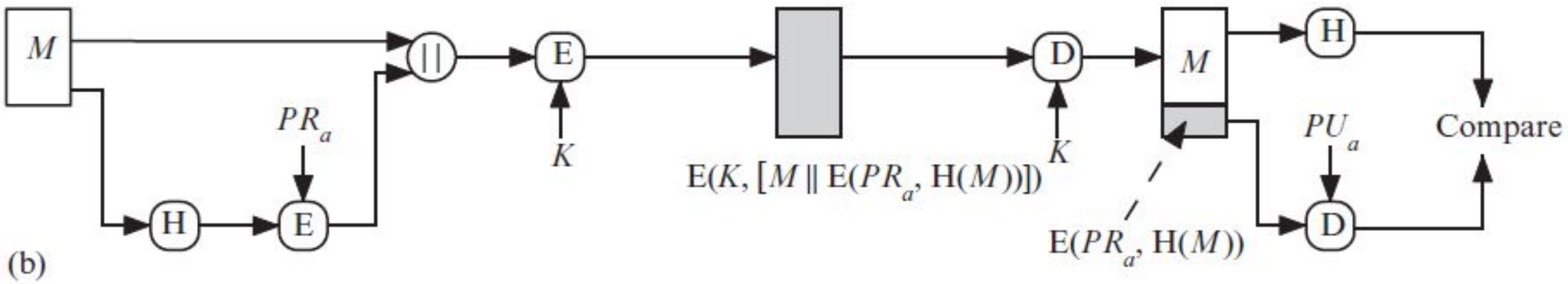


APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Digital Signatures**
- Simplified version how **Hash is used for digital signature**.

1. **Authentication**. The **hash code** is encrypted, using **public-key encryption** with the sender's private key. As with Figure, this provides **authentication**. It also provides a **digital signature**, because only the **sender could have produced the encrypted hash code**. In fact, this is the essence of the digital signature technique.

2. **Confidentiality**. If **confidentiality as well as a digital signature** is desired, then the message plus the private-key-encrypted hash code can be **encrypted** using a **symmetric secret key**. This is a common technique.



APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Other Applications**
  - **One way password file**
  - **Intrusion detection**
  - **Virus detection**
  - **a pseudorandom number generator**

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Other Applications**
- **one-way password file**
- Hash functions are commonly used to create a **one-way password file**.
- **hash of a password** is stored by an operating system **rather than the password** itself.
- Thus, the actual password is **not retrievable** by a hacker who gains access to the 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**.

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

- **Other Applications**
- **intrusion detection** and **virus detection**.
- Hash functions can be used for
- **Store H(F) for each file** on a system and secure the hash values (e.g., on a CD-R that is kept secure). 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).
- **pseudorandom function**
- A cryptographic hash function 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

Two Simple Hash Functions

- To get **some feel for the security considerations** involved in **cryptographic hash functions - two simple, insecure hash functions**
- **All hash functions 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**.
- One of the **simplest hash functions** is the **bit-by-bit exclusive-OR (XOR)** of every block. This can be expressed as

$$C_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im}$$

where

C_i = ith bit of the hash code, 1 <= I <= n

m = number of n-bit blocks in the input

b_{ij} = ith bit in jth block

$\oplus$ = XOR operation

Two Simple Hash Functions

- This **operation produces** a **simple parity bit** for each bit position and is known as a **longitudinal redundancy check**.
- It is **reasonably effective** for **random data as a data integrity check**. Each n-bit hash value is equally likely.
- Thus, the **probability** that a **data error** will result in an unchanged hash value is 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. So if a 128-bit hash value is used, instead of an effectiveness of 2^{-128} the hash function on this type of data has an effectiveness of 2^{-112}

Two Simple Hash Functions

1. **XOR-Based Hash Function**
2. **Modular Sum Hash Function**

Two Simple Hash Functions

1. XOR-Based Hash Function

A basic hash function using the XOR operation to combine all bytes of data into a single hash value.

Algorithm:

1. Initialize a hash variable to 0 (e.g., hash = 0).
2. For each byte in the input data:
    - Perform an XOR operation between the **hash variable and the byte**.
    - Update the hash value with the result.
3. Return the final hash value.

Two Simple Hash Functions

Example Walkthrough for "hello":

1. ASCII values: 'h' = 104, 'e' = 101, 'l' = 108, 'l' = 108, 'o' = 111.
2. Perform XOR step by step:
    - Initial hash: 0
    - After 'h': 0 XOR 104 = 104
    - After 'e': 104 XOR 101 = 13
    - After 'l': 13 XOR 108 = 97
    - After next 'l': 97 XOR 108 = 13
    - After 'o': 13 XOR 111 = 98

Result: Hash value = **980**

Limitations:

- Very simple and **not secure** for cryptographic purposes.
- Useful for **lightweight error detection**, **not security**.

Two Simple Hash Functions

- An XOR-based hash function is primarily used in lightweight or non-secure applications where speed and simplicity are more important than cryptographic security. Here are some typical use cases:

1. **Error Detection**

- **Scenario**: A system transmits data across a network and wants to ensure that **no corruption** occurs during transmission.
- **Usage**:
- Compute the XOR-based hash of the data before sending it.
- At the receiving end, compute the hash again and compare it with the transmitted hash.
- If the hashes don't match, the data has been corrupted.

Two Simple Hash Functions

- An XOR-based hash function is primarily used in lightweight or non-secure applications where speed and simplicity are more important than cryptographic security. Here are some typical use cases:

2. **Checksums for Small Data**

- **Scenario**: Ensuring **file integrity for lightweight applications**, such as verifying simple log files or configuration files.

Usage:

- Store the XOR-based hash **alongside the file**.
- When accessing the file, **recompute the hash and compare** it with the stored value.
- A mismatch indicates potential tampering or corruption.

basic integrity checks or **error detection** in non-critical applications, particularly in environments where simplicity and performance are prioritized over security.

Two Simple Hash Functions

2. Modular Sum Hash Function

- A simple hash function that sums the numeric values of all bytes in the input and takes the modulus with a fixed number (e.g., 256).
- **Algorithm:**

1. Initialize a hash variable to 0.
2. For each byte in the input data:
    1. Add the byte value to the hash variable.
    2. Take the modulus (e.g., hash = hash % 256) to limit the hash size.
3. Return the final hash value.

Two Simple Hash Functions

Example Walkthrough for "hello":

1. ASCII values: 'h' = 104, 'e' = 101, 'l' = 108, 'l' = 108, 'o' = 111.
2. Perform modular addition:
    - Initial hash: 0
    - After 'h': (0 + 104) % 256 = 104
    - After 'e': (104 + 101) % 256 = 205
    - After 'l': (205 + 108) % 256 = 57
    - After next 'l': (57 + 108) % 256 = 165
    - After 'o': (165 + 111) % 256 = 20

Result: Hash value = **20**

Two Simple Hash Functions

- **Characteristics:**
- **Simple to Compute:** Easy to implement with low computational overhead.
- **Fixed Range Output:** Modulus ensures the hash value remains within a specified range, such as 0–255.
- **Non-Cryptographic:** Not suitable for security purposes due to high susceptibility to collisions.

- **Use Cases:**

1. **Checksum in Data Transmission:** Validate that data has not been corrupted during transfer.
2. **File Validation:** Lightweight integrity checks for small files.
3. **Indexing for Small Tables:** Use the hash value as an index in lookup tables.

For sensitive or large-scale applications, use cryptographic hash functions like **SHA-256** instead

Requirements and Security

- Before proceeding, we need to define two terms.
- For a **hash value h = H(x),** we say **that x is the preimage** of h. That is, x is a data block whose hash value, 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**.
- A **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**.

Security Requirements of Cryptographic Hash Functions

Table 11.1 Requirements 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) with $x \neq y$, such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

Security Requirements of Cryptographic Hash Functions

- The **first three properties** are requirements for the practical application of a hash function.
- The **fourth property,** preimage resistant, is the **one-way property**: it is easy to generate a code given a message, but virtually impossible to generate a message given a code. This property is important if the **authentication technique** involves the use of a secret value.
- Preimage resistance, also known as the **one-way property**, is a fundamental requirement of secure cryptographic hash functions.
- It ensures that, given a hash output h(x), it is **computationally infeasible to reverse-engineer** the original input x.
- Essentially, the function works in **one direction (input → hash),** but reversing it **(hash → input)** is **extremely difficult**.
- If **h("password123")=a1b2c3d4**, preimage resistance ensures that, even if an attacker obtains a1b2c3d4 they **cannot determine** that the original input was "password123."

Security Requirements of Cryptographic Hash Functions

- The **fifth property**, **second preimage resistant**, guarantees that it is infeasible to find an **alternative message with the same hash value as a given message**. This **prevents forgery** when an encrypted hash code is used
- **Second preimage resistance** is a fundamental property of cryptographic hash functions. It ensures that, given a specific input x1 and its hash h(x1), it is computationally infeasible to find another distinct input x2 $(x2 \neq x1)$ such that $h(x2)=h(x1)$.
- If this property were not true, an attacker would be capable of the following sequence:
  - **First**, observe or intercept a message plus its encrypted hash code;
  - **second**, generate an unencrypted hash code from the message;
  - **third**, generate an alternate message with the same hash code.

Security Requirements of Cryptographic Hash Functions

- A hash function that satisfies the **first five properties** in Table is **referred to as a weak hash function**.
- If the **sixth property**, collision resistant, is also **satisfied**, then it is referred to as a **strong hash function**.
- A strong hash function **protects against an attack** in which one party **generates a message for another party to sign.**
- **Collision resistance** is a critical property of cryptographic hash functions. It ensures that it is computationally infeasible to find two distinct **inputs $x1$ and $x2$** such that their hash outputs are the same, i.e., $\boldsymbol{h(x1)=h(x2)}$
- For example, suppose **Bob writes an IOU message**, sends it to Alice, and **she signs it**. Bob finds **two messages with the same hash**, one of which requires Alice to pay a small amount and one that requires a large payment. Alice signs the first message, and Bob is then able to claim that the second message is authentic.

Security Requirements of Cryptographic Hash Functions

- Figure shows the relationships among the **three resistant properties**.
- A function that is **collision resistant** is also **second preimage resistant**, but the **reverse is not necessarily true.**
- A function can be **collision resistant but not preimage resistant and vice versa**.
- A function can be preimage resistant but not second preimage resistant and vice versa.

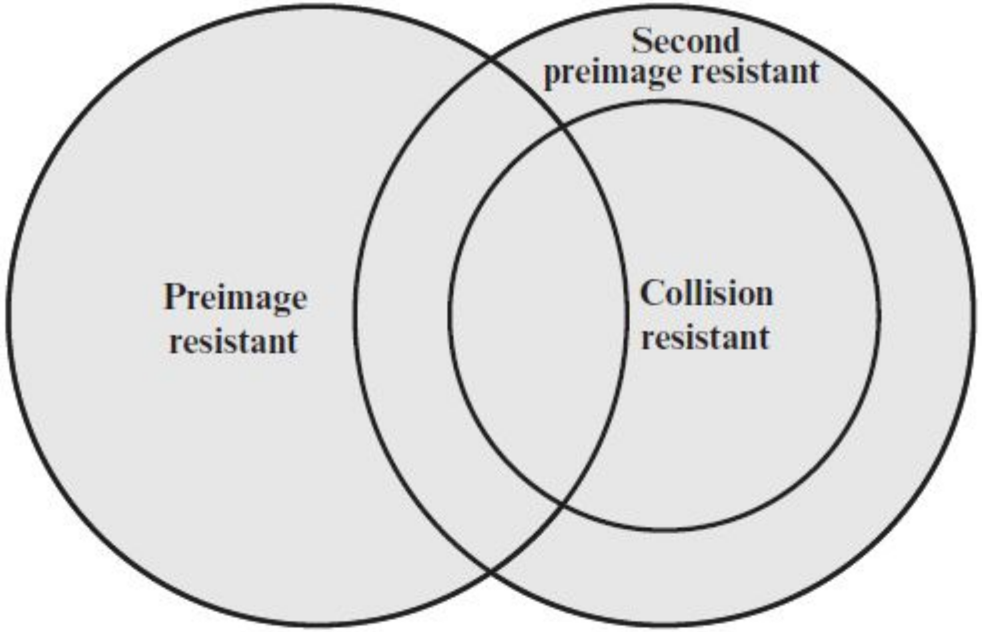


Figure 11.6 Relationship Among Hash Function Proper

Security Requirements of Cryptographic Hash Functions

- The **final requirement** in Table **pseudorandomness**, has **not traditionally been listed** as a requirement of cryptographic hash functions but is **more or less implied.**
- cryptographic hash functions are commonly used for **key derivation** and **pseudorandom number** generation, and that in **message integrity applications**, the three resistant properties depend on the **output of the hash function** appearing to be random. Thus, it makes sense to verify that in fact a given hash function **produces pseudorandom output**.

Cryptography

Cryptographic Data Integrity Algorithms

M S Vilkhu

- 23 Nov 2024(C1/C3/C5)

25 Nov 2024(C1/C3/C5)

Topics

CRYPTOGRAPHIC HASH FUNCTIONS

11.1 Applications of Cryptographic Hash Functions Message
- Authentication
- Digital Signatures
- Other Applications

11.2 Two Simple Hash Functions

11.3 Requirements and Security
- Security Requirements for Cryptographic Hash Functions
- Brute-Force Attacks
- Cryptanalysis

11.4 Hash Functions Based on Cipher Block Chaining

11.5 Secure Hash Algorithm (SHA)
- SHA-512 Logic SHA-512
- Round Function
- Example

11.6 SHA-3
- The Sponge Construction
- The SHA-3 Iteration Function f

Cryptographic Hash Functions

11.1 **Applications of Cryptographic Hash Functions**
- Message Authentication
- Digital Signatures
- Other Applications

11.2 **Two Simple Hash Functions**

11.3 **Requirements and Security**
- Security Requirements for Cryptographic Hash Functions
- Brute-Force Attacks
- Cryptanalysis

11.4 **Hash Functions Based on Cipher Block Chaining**

11.5 **Secure Hash Algorithm (SHA)**
- SHA-512 Logic
- SHA-512 Round Function
- Example

11.6 **SHA-3**
- The Sponge Construction
- The SHA-3 Iteration Function *f*

Security Requirements of Cryptographic Hash Functions

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

	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

Brute Force Attack

- As with encryption algorithms, there are **two categories** of attacks on hash functions:
  - **brute-force attacks** and
  - **cryptanalysis**.
- A **brute-force attack** does **not depend on the specific algorithm** but **depends only on bit length.**
- A **cryptanalysis**, in contrast, is an attack based on **weaknesses in a particular cryptographic algorithm**.

- **PREIMAGE AND SECOND PREIMAGE ATTACKS**

Brute Force Attack

- **PREIMAGE AND SECOND PREIMAGE ATTACKS**
- For a preimage or second preimage attack, an 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 2^m.
- Specifically, 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.

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- For a **collision resistant attack**, an adversary wishes to **find two messages or data blocks**, **x and y, that yield the same hash function: H(x) = H(y).**
- This turns out to **require considerably less effort** than a preimage or second preimage attack.
- The **effort required is explained** by a mathematical result referred to as the **birthday paradox**.

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- In essence, if we choose random variables from a uniform distribution in the range 0 through N-1, then the probability that a repeated element is encountered exceeds 0.5 after $\sqrt{\mathbf{N}}$ choices have been made.
- Thus, for an m-bit hash value, if we pick data blocks at random, we can expect to find two data blocks with the same hash value within $\sqrt{\mathbf{2m}} = \mathbf{2^{m/2}}$ attempts.
- Explanation in more simple terms…

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- **Birthday Paradox** with example
- The **birthday paradox** refers to the **surprising probability phenomenon** where, in a group of people, the **chances of two people** sharing the **same birthday** are higher than most people intuitively expect.
- **How It Works (Intuitively):**

1. **Misleading Intuition:**
   1. Most people assume the **probability of shared birthdays** requires a **large group**, like 365 people, because there are 365 days in a year.
   2. However, the paradox reveals that in a group as small as **23 people**, the probability of at **least two people** sharing a birthday is more than **50%**.
2. **Key Idea:**
   1. Instead of calculating the probability of a **specific person** sharing a birthday with someone else, the paradox considers **all possible pairs of people** in the group. As the **group grows**, the number of **pairs increases rapidly**.

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- **Birthday Paradox with example**
- The **birthday paradox** refers to the surprising probability phenomenon where, in a group of people,

3. Example:

1. In a group of **23 people**, there are $\binom{23}{2} = 253$ **pairs**
2. With this many comparisons, the chance of **at least one pair sharing a birthday is significant.**

$$\binom{23}{2} = 253$$

$$\binom{n}{r} = \frac{n!}{r! \cdot (n-r)!}$$

$$\binom{23}{2} = \frac{231!}{2!\ (23-2)!} = 253 \text{ pairs}$$

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- **Birthday Paradox with example**
- **Connection to Cryptography:**
- The birthday paradox is often used to explain **collision probability** in hash functions:
- In cryptographic terms, a "**collision**" occurs when **two different inputs** produce the **same hash value.**
- For a hash function with n-bit outputs, it **only takes about $2^{n/2}$** inputs (not 2^n to find a collision. This is called the **birthday attack.**

- **Why It's Counterintuitive:**
- The paradox challenges intuition because **humans** typically think **linearly**, not **combinatorially**.
- The **rapid increase in pairwise comparisons** as **group size grows** is the key to the phenomenon.

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- **Birthday Paradox with example**
- **<u>Thinking linearly</u>**, not **<u>combinatorially</u>**.
- **Linear Thinking:**
  - Focuses on direct, sequential, or proportional relationships.
  - You consider one step or one variable at a time.
  - Often involves predictable, straightforward cause-and-effect patterns.
- **For example:**
- If a task takes 1 minute and you have 5 tasks, the total time will be 5×1=55 minutes.
- **Combinatorial Thinking:**
  - Involves considering all possible combinations or permutations of elements in a system.
  - Explores exponential growth or complexity due to interconnections or interactions.
- **For example:**
- If you have 5 items and want to consider all pairwise comparisons, you deal with $\binom{5}{2} = 10$ combinations.

Brute Force Attack

- **COLLISION RESISTANT ATTACKS**
- If collision **resistance is required** (and this is desirable for a general-purpose secure hash code), then the value **$2^{m/2}$ determines the strength** of the hash code **against brute-force attacks**.
- Van Oorschot and Wiener [VAN094] presented a design for a **$10 million collision search machine for MD5**, which has a **128-bit hash length**, that could find a collision in **24 days.** Thus, a 128-bit code may be **viewed as inadequate.**
- The **next step up**, if a hash code is treated as a sequence of 32 bits, is a **160-bit hash length**. With a hash length of 160 bits, the same search machine would require over **four thousand years** to find a collision. With today's technology, the time would be **much shorter**, so that **160 bits now appears suspect.**

Cryptanalysis

- As with encryption algorithms, cryptanalytic attacks on hash functions seek to **exploit some property of the algorithm** to perform some attack **other than an exhaustive search.**
- The **way to measure the resistance** of a hash algorithm to cryptanalysis is to compare its **strength to the effort required for a brute-force attack**.
- That is, an **ideal hash algorithm** will require a **cryptanalytic effort greater than or equal to the brute-force effort.**

Cryptanalysis

- Cryptanalysis attacks on hash functions aim to **exploit weaknesses** in the hash algorithm to compromise its security properties.
- The goal of these attacks is to **break the fundamental characteristics** of a secure hash function:
- **preimage resistance**,
- **second preimage resistance**, and
- **collision resistance**.

Cryptography

Cryptographic Data Integrity Algorithms

M S Vilkhu

25 Nov 2024(C1/C3

- 23 Nov 2024(C1/C3/C5)

-29 Nov 2024(C5)

Topics

CRYPTOGRAPHIC HASH FUNCTIONS

11.1 Applications of Cryptographic Hash Functions Message
- Authentication
- Digital Signatures
- Other Applications

11.2 Two Simple Hash Functions

11.3 Requirements and Security
- Security Requirements for Cryptographic Hash Functions
- Brute-Force Attacks
- Cryptanalysis

11.4 Hash Functions Based on Cipher Block Chaining

11.5 Secure Hash Algorithm (SHA)
- SHA-512 Logic SHA-512
- Round Function
- Example

11.6 SHA-3
- The Sponge Construction
- The SHA-3 Iteration Function f

Cryptographic Hash Functions

Secure Hash Algorithm (SHA)

Secure Hash Algorithm (SHA)

- most widely used hash function has been the **Secure Hash Algorithm (SHA).**
- Indeed, **because** virtually **every other widely used hash function** had been found to have **substantial cryptanalytic weaknesses**,
- **SHA-O** SHA was developed by the **National Institute of Standards and Technology (NIST)** and published as a ***federal information processing standard*** **(FIPS 180) in 1993.**
- When **weaknesses were discovered in SHA, i.e. known as SHA-O**, a revised version was issued as **FIPS 180-1 in 1995 and is referred to as SHA-I**.
- The actual standards document is **entitled** "**Secure Hash Standard**."
- SHA is **based** on the hash function **MD4**, and its **design** closely **models** MD4.

Secure Hash Algorithm (SHA)

- **SHA-I** produces a **hash value of 160 bits.**
- **SHA-2.** In 2002, NIST produced a **revised version** of the standard, **FIPS 180-2**, that defined **three new versions of SHA**, with **hash value lengths of 256, 384, and 512 bits**, known as **SHA-256**, **SHA-384**, **and SHA-512**, respectively. **Collectively**, these hash algorithms are known as **SHA-2**.
- These new versions have the **same underlying structure** and use the **same types of modular arithmetic** and **logical binary operations as SHA-I**.
- A **revised document** was issued as **FIP PUB 180-3 in 2008**, which added a 224-bit version
- **SHA-I and SHA-2** are also specified in RFC 6234, which essentially duplicates the material in FIPS 180-3

Secure Hash Algorithm (SHA)

- In **2015**, NIST issued FIPS **180-4**, which added two additional algorithms: **SHA-512/224 and SHA-512/256**.

Table 11.3 Comparison of SHA Parameters

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

Note: All sizes are measured in bits.

Secure Hash Algorithm (SHA)

- **In 2005**, NIST announced the intention to **phase out approval of SHA-I** and move to a reliance on **SHA-2 by 2010**.
- Shortly thereafter, a research team **described an attack in which two separate messages** could be found that **deliver the same SHA-I hash** using **2^{69} operations**, far fewer than the **2^{80}** operations **previously thought needed to find a collision** with an SHA-I hash [WANG05]. This result should hasten the transition to SHA-2.

SHA-512 Logic

SHA-512 Logic

- The algorithm **takes as input a message** with a **maximum length** of less than 2^{128} bits and produces as **output a 512-bit** message digest.
- The **input** is processed in **1024-bit blocks**.
- one of the largest hash functions in the SHA-2 family.
- This algorithm is frequently used for **email address hashing, password hashing, and digital record verification**. SHA-512 is also used in **blockchain technology**, with the BitShares network becoming the most known example.

SHA-512 Logic

- **basic properties**
- **Deterministic** – The same input will always get the same result.
- **Fast to compute** – The hash for any given data can be calculated very quickly.
- **Irreversible** – You cannot determine the original input from its hash.
- **Collision-resistant** – It is computationally challenging to discover two distinct inputs that generate the same hash.
- **Avalanche effect** – A small change in input (even flipping a single bit) results in a significantly different hash.

SHA-512 Logic

- **How SHA-512 Works?**
- Without going too far into the mathematical concepts, SHA-512 operates as follows –

1. **Initialization** – It starts with **eight hash values** calculated from the **square roots of the initial eight prime numbers**.
2. **Pre-processing** – The input message is **padded** so that it is a **multiple of the Block size**. The original message's 128-bit length (before padding) is added to the very end of the padded message.
3. **Parsing** – The message is then **separated into 1024-bit parts**.
4. **Main Loop** – The **main loop** analyses each 1024-bit block in **80 rounds**, **manipulating** the data via **logical operations**, **bitwise shifts**, and **modular arithmetic**.
5. **Output** – After **all of the blocks have been processed**, the **resulting 512-bit message digest** is output as the hash.

SHA-512 Logic

2. Pre-processing – The input message is **padded** so that it is a **multiple of the Block size**. The **original message's 128-bit length (before padding)** is **added to the very end of the padded message.**

- Meaning
- **Context:**
- When hashing a message, most cryptographic hash functions require the input to be of a specific size (usually a multiple of a block size, such as 512 bits). If the message isn't the correct size, it needs to be **padded** to fit.

SHA-512 Logic

2. Pre-processing – The input message is **padded** so that it is a **multiple of the Block size**. The **original message's 128-bit length (before padding)** is **added to the very end of the padded message.**

- **Meaning**

Explanation:

1. **Original Message Length:**

   The **original length** of the message (in bits) is determined. For instance, if the original message is **128 bits long**, that number is **noted**.

2. **Padding:**

   To make the message **fit the required block size**, the message is **padded** with a **specific pattern** (e.g., **a single 1 bit** followed by **enough 0 bits**).

3. **Appending the Original Length:**

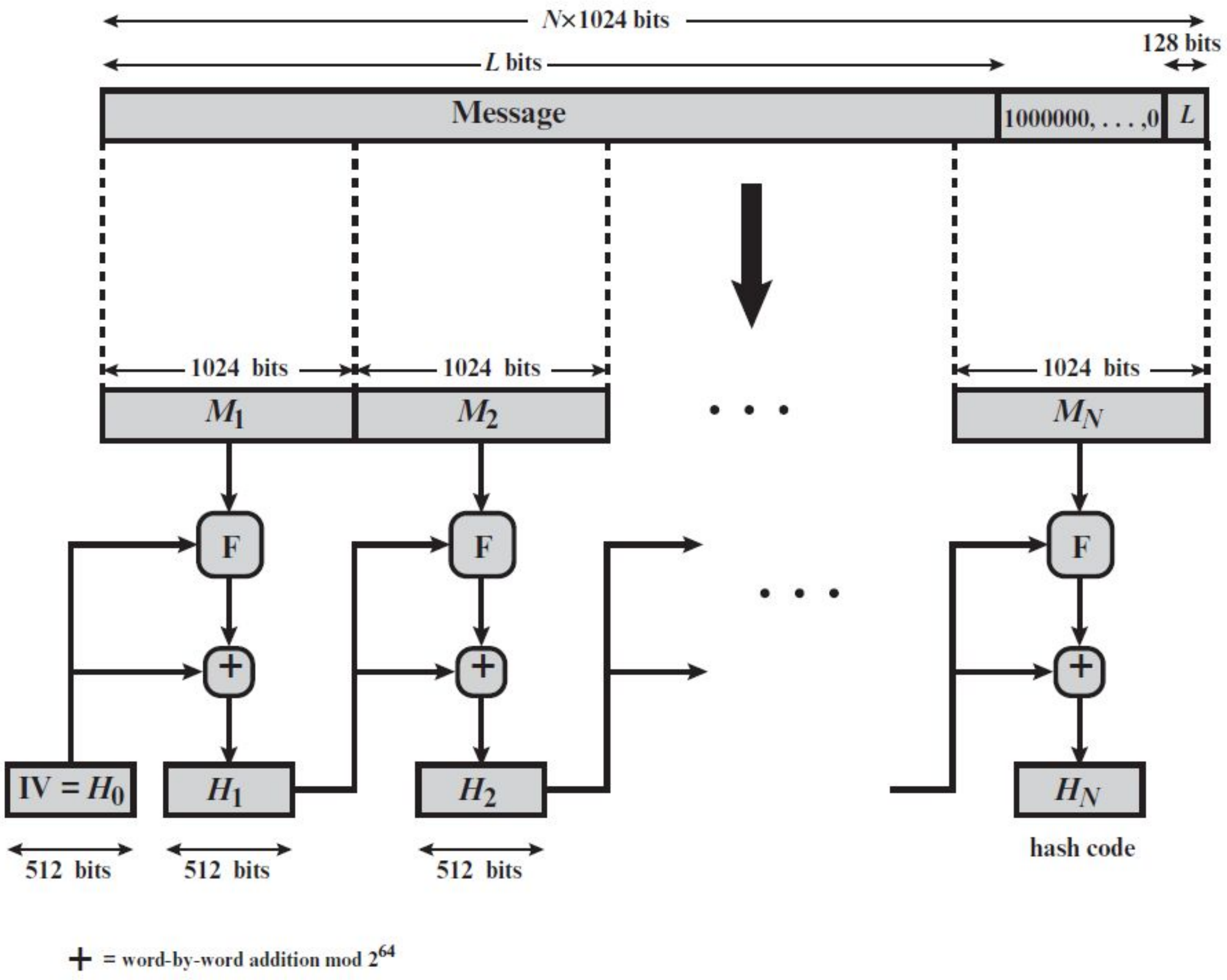
   After padding, the **original length of the message** (128 bits in this case) is **encoded as a fixed-size binary number** and **added to the end of the padded message**. This ensures the **hash function** has information about the **original message length**

SHA-512 Logic

- **Algorithm**
- The SHA-512 algorithm consists of the following steps –

1. **Message Padding** – First, your message is **padded** to ensure that it is the **correct size** for the algorithm. This ensures that it **can be broken down into blocks** and processed.
2. **Initial hash values** – The algorithm starts with **eight initial hash values**. These set values serve as the **basis for the hashing procedure**.
3. **Message processing** – The padded message is **divided into blocks**. Each block progresses over a **series of stages known as rounds**. In **each round**, the block is **mixed** and **adjusted** using specific techniques.
4. **Final hash value** – **After all blocks** have been **examined**, the hash **value is computed**. This hash value serves as a **unique fingerprint** for the original message.
5. **Output** – The SHA-512 algorithm generates the final hash result, which is generally a **string of hexadecimal integers**. This is the value returned after hashing your original message.

SHA-512 Logic



SHA-512 Logic

- **Applications**
- **SHA-512** and its **siblings from the SHA-2 family** are commonly used in a number of security applications and protocols, including –
- **Digital signatures** are used to validate the **integrity** **of a message or document**.
- **Certificate creation** is a process used by Certificate Authorities (CAs) to assure the security of digital certificates.
- **Password hashing** involves storing passwords in databases as hashes rather than plain text.
- **Blockchain and cryptocurrencies**: Used to ensure data integrity and security.

Thank You