

# Lesson 3: Cryptographic Hash Functions

## Module A: Blockchain Foundations

BSc Blockchain & Cryptocurrency

University Course

2025

# Learning Objectives

By the end of this lesson, you will be able to:

1. Define cryptographic hash functions and their key properties
2. Explain the avalanche effect with concrete examples
3. Understand SHA-256 algorithm and its role in Bitcoin
4. Analyze collision resistance and the birthday paradox
5. Construct and verify Merkle trees step-by-step
6. Identify real-world applications of hash functions beyond blockchain

**Prerequisites:** L02 - DLT Concepts (Merkle trees introduction)

# What is a Hash Function?

## Hash Function Definition

A **hash function** is a mathematical function that takes an input (message) of arbitrary length and produces a fixed-size output (hash/digest).

### Mathematical Notation:

$$H : \{0, 1\}^* \rightarrow \{0, 1\}^n$$

Where  $\{0, 1\}^*$  is any binary string, and  $\{0, 1\}^n$  is a fixed  $n$ -bit output.

### Example (SHA-256):

- Input: "Hello World" (any length)
- Output: a591a6d40bf420404... (always 256 bits = 64 hex characters)

*Hash functions are "digital fingerprints" - unique identifiers for data*

## Non-Cryptographic Hashes

- Fast computation
- Designed for hash tables, checksums
- NOT resistant to adversarial attacks
- Collision attacks are easy

### Examples:

- CRC32 (cyclic redundancy check)
- MD5 (broken, not cryptographic anymore)
- MurmurHash (fast, non-crypto)

*Use Case:* Hash tables in programming languages

**Key Difference:** Cryptographic hashes must withstand deliberate attacks

## Cryptographic Hashes

- Slower, but secure
- Collision resistant
- Preimage resistant
- Unpredictable (pseudorandom)

### Examples:

- SHA-256 (Bitcoin, SSL/TLS)
- SHA-3 (Keccak, used in Ethereum)
- BLAKE2 (fast, modern)

*Use Case:* Digital signatures, blockchain, passwords

## Deterministic Property

The same input always produces the same output. No randomness involved.

### Example:

- $H(\text{"blockchain"}) = \text{ef7797e13d3a75526946a3bcf00daec9fc9c9c4d51ddc7cc5df888f74dd434d1}$
- Computing this hash 1,000 times yields the same result every time

### Why This Matters:

- Enables verification: Anyone can recompute the hash
- Makes auditing possible: Deterministic proofs
- Foundation for digital signatures

### Contrast with Random Functions:

- Random:  $f(x)$  might return different values each time
- Hash:  $H(x)$  is a pure function (functional programming)

## Property 2: Fixed-Length Output

### Fixed-Length Property

Regardless of input size, the hash output is always the same fixed length.

#### Examples (SHA-256):

- $H(\text{"a"}) = 256$  bits (64 hex characters)
- $H(\text{entire Bitcoin whitepaper}) = 256$  bits (same length)
- $H(1 \text{ GB video file}) = 256$  bits (still 64 hex chars)

#### Common Hash Sizes:

Algorithm	Output Size (bits)	Hex Characters
MD5 (broken)	128	32
SHA-1 (deprecated)	160	40
SHA-256 (Bitcoin)	256	64
SHA-512	512	128

**Implication:** Infinite inputs map to finite outputs  $\Rightarrow$  collisions must exist (pigeonhole principle)

## Property 3: Avalanche Effect (Sensitivity)

### Avalanche Effect

A tiny change in input (even 1 bit) causes a massive, unpredictable change in the output. Approximately 50% of output bits flip.

#### Example (SHA-256):

- Input: "blockchain"
  - Hash: ef7797e13d3a75526946a3bcf00daec9fc9c9c4d51ddc7cc5df888f74dd434d1
- Input: "Blockchain" (capital B)
  - Hash: 625da44e4eaf58d61cf048d168aa6f5e492dea166d8bb54ec06c30de07db57e1

#### Analysis:

- Only 1 bit changed in input (ASCII: b = 01100010, B = 01000010)
- Output is completely different (no pattern recognizable)
- Approximately 128 out of 256 bits flipped (50%)

*Visualizing this effect will be the focus of Lab 4*

## Property 4: Preimage Resistance (One-Way)

### Preimage Resistance

Given a hash output  $h$ , it is computationally infeasible to find any input  $m$  such that  $H(m) = h$ .

**Analogy:** Easy to scramble an egg, impossible to unscramble it

#### Mathematical Formulation:

- Computing  $h = H(m)$  is fast ( $\approx$  microseconds)
- Finding  $m$  given  $h$  requires trying all  $2^{256}$  possibilities (for SHA-256)
- At 1 trillion hashes/second, this would take  $10^{58}$  years

#### Practical Implications:

- Password storage: Store  $H(\text{password})$ , not password itself
- Blockchain integrity: Cannot reverse-engineer block data from hash
- Commitment schemes: Hash your choice before revealing

*Exception: Rainbow tables for weak passwords (mitigated by salting)*



### Second Preimage Resistance

Given input  $m_1$  and its hash  $h = H(m_1)$ , it is computationally infeasible to find a different input  $m_2 \neq m_1$  such that  $H(m_2) = h$ .

#### Scenario:

- You sign a contract: "Pay Alice \$1,000"
- Hash:  $H(\text{contract}) = h$
- Attacker tries to find alternate message: "Pay Bob \$1,000,000" with same hash  $h$
- Second preimage resistance prevents this attack

#### Difference from Preimage Resistance:

- **Preimage:** Given  $h$ , find any  $m$  where  $H(m) = h$
- **Second Preimage:** Given  $m_1$  and  $h = H(m_1)$ , find different  $m_2$  where  $H(m_2) = h$

*Second preimage attacks are easier than preimage attacks, but still infeasible for strong hashes*

### Collision Resistance

It is computationally infeasible to find any two different inputs  $m_1 \neq m_2$  such that  $H(m_1) = H(m_2)$ .

#### Theoretical Guarantee:

- Pigeonhole principle: Collisions MUST exist (infinite inputs, finite outputs)
- But finding them should be practically impossible

#### Birthday Paradox Attack:

- For  $n$ -bit hash, finding collision requires  $\approx 2^{n/2}$  attempts (not  $2^n$ )
- SHA-256:  $2^{128}$  operations needed ( $\approx 10^{38}$  hashes)
- Still infeasible with current technology

#### Broken Examples:

- MD5 collisions found in 2004 (now insecure)
- SHA-1 collisions demonstrated in 2017 (deprecated for security)

# The Birthday Paradox

## Classic Probability Problem:

*How many people needed in a room for 50% probability that two share a birthday?*

Answer: Only 23 people (counterintuitive!)

## Why It Matters for Hashing:

- With 365 possible birthdays (like hash outputs)
- Finding a collision takes  $\sqrt{365} \approx 23$  samples
- Generalized: For  $N$  possible outputs, collision in  $\approx \sqrt{N}$  attempts

## Application to SHA-256:

- Total possible hashes:  $N = 2^{256}$
- Collision search:  $\sqrt{2^{256}} = 2^{128}$  attempts
- At 1 trillion hashes/second:  $10^{19}$  years

*This is why we need large hash outputs: Collision resistance scales as  $2^{n/2}$*

## SHA-256

**Secure Hash Algorithm 256-bit** is a cryptographic hash function designed by the NSA, published by NIST in 2001 as part of the SHA-2 family.

### Specifications:

- Output: 256 bits (32 bytes, 64 hex characters)
- Block size: 512 bits (processes data in 512-bit chunks)
- Internal state: Eight 32-bit words (256 bits total)
- Rounds: 64 compression iterations per block

### Uses in Bitcoin:

- Block hashing: Double SHA-256 ( $\text{SHA256}(\text{SHA256}(\text{header}))$ )
- Transaction IDs: SHA-256 hash of transaction data
- Address generation: SHA-256 + RIPEMD-160
- Merkle tree construction

## Step-by-Step:

1. **Padding:** Append bits to make length  $\equiv 448 \pmod{512}$ 
  - Add single 1 bit, then zeros, then 64-bit length field
2. **Parsing:** Break padded message into 512-bit blocks
3. **Initialize:** Set eight 32-bit hash values (constants from square roots of first 8 primes)
4. **Compression:** For each block:
  - Expand 512 bits to 2,048 bits (message schedule)
  - 64 rounds of bitwise operations (AND, XOR, rotate, add)
  - Update internal state
5. **Output:** Concatenate final 8 words into 256-bit hash

*Details involve modular arithmetic and bitwise logic (beyond scope, but implementations widely available)*

**Input:** “abc” (3 bytes)

**Step 1 - Padding:**

- Binary: 01100001 01100010 01100011 (24 bits)
- Add 1 bit: 01100001 01100010 01100011 1...
- Pad with zeros until length  $\equiv 448 \pmod{512}$
- Append 64-bit length: ...0000011000 (24 in binary)

**Step 2 - Initialize Hash Values (first 8 primes):**

- $H_0 = 6a09e667$ ,  $H_1 = bb67ae85$ , ...,  $H_7 = 5be0cd19$

**Step 3 - Compression (64 rounds):**

- Complex bitwise operations (Ch, Maj,  $\Sigma_0$ ,  $\Sigma_1$ , etc.)

**Final Output:**

$$H(\text{“abc”}) = \text{ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad}$$

## Why Double Hashing?

Bitcoin uses  $\text{SHA256}(\text{SHA256}(x))$  instead of single SHA-256 to guard against length-extension attacks (theoretical vulnerability in Merkle-Damgård construction).

### Process:

1. Compute  $h_1 = \text{SHA256}(\text{data})$
2. Compute  $h_2 = \text{SHA256}(h_1)$
3. Use  $h_2$  as final hash

### Example - Bitcoin Block Hash:

- Input: 80-byte block header
- First hash:  $h_1 = \text{SHA256}(\text{header})$
- Second hash:  $h_2 = \text{SHA256}(h_1)$
- Result:  $h_2$  must be below difficulty target to be valid

*Performance: Modern hardware computes millions of double-SHA256 per second*

**Goal:** Efficiently verify transaction inclusion without downloading all data

**Construction Algorithm:**

1. Start with  $n$  transactions:  $T_{x_1}, T_{x_2}, \dots, T_{x_n}$
2. Hash each transaction:  $H_1 = H(T_{x_1}), H_2 = H(T_{x_2}), \dots, H_n = H(T_{x_n})$
3. Pair and hash:  $H_{12} = H(H_1 || H_2), H_{34} = H(H_3 || H_4), \dots$
4. If odd number, duplicate last hash:  $H_{nn} = H(H_n || H_n)$
5. Repeat until single root hash (Merkle Root)

**Example (4 transactions):**

- Level 0:  $T_{x_1}, T_{x_2}, T_{x_3}, T_{x_4}$
- Level 1:  $H_1, H_2, H_3, H_4$
- Level 2:  $H_{12} = H(H_1 || H_2), H_{34} = H(H_3 || H_4)$
- Level 3 (Root):  $R = H(H_{12} || H_{34})$



**Scenario:** Light client wants to verify  $T_{x_3}$  is in block (4 transactions total)

**Verifier Has:**

- Block header with Merkle Root  $R$
- Transaction  $T_{x_3}$

**Prover Sends (Merkle Proof):**

- $H_4$  (sibling of  $H_3$ )
- $H_{12}$  (sibling of  $H_{34}$ )

**Verification Steps:**

1. Compute  $H_3 = H(T_{x_3})$
2. Compute  $H_{34} = H(H_3 || H_4)$  (using provided  $H_4$ )
3. Compute  $R' = H(H_{12} || H_{34})$  (using provided  $H_{12}$ )
4. If  $R' = R$ , then  $T_{x_3}$  is proven to be in the block

**Efficiency:** Only 2 hashes sent instead of 3 other transactions

## Space Complexity:

- For  $n$  transactions, tree has  $\approx 2n$  nodes
- Merkle proof requires  $\log_2(n)$  hashes

## Proof Size Comparison:

Transactions in Block	Full Data	Merkle Proof
10	$\approx 2.5$ KB	4 hashes (128 bytes)
100	$\approx 25$ KB	7 hashes (224 bytes)
1,000	$\approx 250$ KB	10 hashes (320 bytes)
10,000	$\approx 2.5$ MB	14 hashes (448 bytes)

## Real-World Impact:

- Bitcoin block:  $\approx 2,000$  transactions  $\Rightarrow$  11-hash proof ( $\approx 352$  bytes)
- Full block:  $\approx 1$  MB
- Savings:  $\frac{352}{1,000,000} = 0.035\%$  of data needed

## Binary Merkle Tree

- Each node has 2 children
- Used in Bitcoin
- Proof size:  $O(\log_2 n)$
- Simple to implement

## Merkle Patricia Trie (Ethereum)

- Combines Merkle tree + Patricia trie
- Supports key-value storage
- Enables state root (all accounts)
- More complex, but more powerful

## Verkle Trees (Future Ethereum)

- Uses polynomial commitments
- Constant-size proofs (regardless of tree size)
- Enables stateless clients
- Based on vector commitments

## Sparse Merkle Trees

- Fixed depth (e.g., 256 levels)
- Most branches empty (pruned)
- Supports non-membership proofs
- Used in some zero-knowledge systems

**Problem:** Storing passwords in plaintext is insecure (database breach exposes all passwords)

**Solution:** Store  $H(\text{password})$  instead

1. User creates account with password "mypassword123"
2. System computes  $h = H(\text{"mypassword123"})$
3. Database stores only  $h$ , not the password
4. Login: User enters password, system hashes it, compares with stored  $h$
5. Attacker with database cannot reverse  $h$  to get password (preimage resistance)

**Enhancements:**

- **Salting:** Add random data before hashing to prevent rainbow tables
  - Store  $h = H(\text{password}||\text{salt})$  and salt
- **Key Stretching:** Use slow hash functions (bcrypt, Argon2) to resist brute-force

**Use Case:** Ensure downloaded file hasn't been tampered with

**Process:**

1. Software developer publishes file hash on official website
  - Example: Ubuntu ISO hash on ubuntu.com
2. User downloads file from mirror site
3. User computes hash of downloaded file locally
4. User compares computed hash with official hash
5. If hashes match, file is authentic and unmodified

**Real-World Example:**

- Download Ubuntu 24.04 LTS ISO (4 GB)
- Official SHA-256: c2e6f4dc37ac944e2f8b21de00e9610c79c61d11...
- Compute: `sha256sum ubuntu-24.04-desktop-amd64.iso`
- Match confirms integrity

*Critical for security: Prevents man-in-the-middle attacks during downloads*

## How Git Uses Hash Functions:

- Every commit has a SHA-1 hash (Git transitioning to SHA-256)
- Hash is computed from:
  - Commit message
  - Author/commmitter metadata
  - Tree object (directory structure)
  - Parent commit hash(es)
- Forms a Merkle tree of commits (immutable history)

## Example:

- Commit: a3f5c79... points to parent b2e4d31...
- Changing history requires recomputing all subsequent hashes
- Makes history tampering detectable

## Similarity to Blockchain:

- Git is a content-addressed storage system
- Hashes link commits, just like blocks in blockchain
- Both rely on collision resistance for integrity

## Digital Signatures (Overview - more in L05):

- Hash the message:  $h = H(m)$
- Sign the hash with private key:  $\sigma = \text{Sign}(h, sk)$
- Verify signature:  $\text{Verify}(\sigma, h, pk)$
- Signing hash (32 bytes) is faster than signing entire message (MB+)

## SSL/TLS (HTTPS):

- Certificate authorities (CAs) sign website certificates
- CA computes  $h = H(\text{certificate})$
- CA signs  $h$  with private key
- Your browser verifies signature using CA's public key
- Ensures you're connected to legitimate website, not impostor

## Hash Functions Used:

- Modern TLS 1.3: SHA-256, SHA-384
- Legacy systems: SHA-1 (deprecated due to collision attacks)

## Mining Process:

1. Collect transactions into candidate block
2. Construct block header (80 bytes)
3. Compute:  $h = \text{SHA256}(\text{SHA256}(\text{header}))$
4. Check if  $h < \text{target}$  (difficulty requirement)
5. If no: increment nonce (4-byte counter), repeat step 3
6. If yes: broadcast valid block, receive reward

## Example Difficulty (Block 800,000):

- Target: 00000000000000000000... (19 leading zeros)
- Probability of success per hash:  $\frac{1}{2^{76}} \approx 10^{-23}$
- Network hash rate:  $\approx 400$  EH/s (exahashes/second)
- Average time to find block: 10 minutes (by design)

## Why Hashes Enable PoW:

- Unpredictable output (no shortcut, must try all nonces)
- Fast verification (anyone can check  $h < \text{target}$  instantly)



## What You Should Remember:

1. **Hash Functions:** Transform arbitrary input to fixed-size output (digital fingerprints)
2. **Core Properties:** Deterministic, fixed-length, avalanche effect, preimage resistance, collision resistance
3. **SHA-256:** 256-bit output, used in Bitcoin (double hashing), computationally infeasible to break
4. **Collision Resistance:** Birthday paradox reduces attack from  $2^n$  to  $2^{n/2}$  operations
5. **Merkle Trees:** Binary hash trees enable  $O(\log n)$  proofs for transaction inclusion
6. **Applications:** Passwords, file integrity, Git, digital signatures, Proof-of-Work

## Critical Insight

Cryptographic hash functions are the foundation of blockchain security. Without collision resistance and preimage resistance, the entire system collapses.

## Consider and discuss:

1. **Quantum Computing Threat:** Will quantum computers break SHA-256?
  - Research: Grover's algorithm reduces preimage attack to  $2^{128}$  operations
2. **Hash Function Lifespan:** When should we migrate to SHA-3 or BLAKE3?
  - Consider: Coordination challenge in decentralized networks
3. **Environmental Impact:** Can we reduce PoW energy consumption without sacrificing security?
  - Explore: Alternative consensus mechanisms (PoS, PoSpace)

## Standards & Specs

- NIST FIPS 180-4 (SHA-2 specification)
- RFC 6234 (SHA and HMAC-SHA)
- Keccak Team (SHA-3 documentation)

## Academic Papers

- Merkle (1988): *Digital Signature Based on Hash Functions*
- Wang et al. (2005): *Finding Collisions in SHA-1*

## Tools

- `sha256sum` (Linux command-line)
- Online SHA-256 calculator
- Python `hashlib` library

## Learning Resources

- Computerphile: "Hashing Algorithms"
- Khan Academy: Cryptography course
- 3Blue1Brown: "But what is a hash function?"

### L04: Lab - Hash Experiments

Hands-on exercises:

- Generate SHA-256 hashes using Python `hashlib`
- Visualize the avalanche effect (1-bit input change)
- Build a Merkle tree from scratch
- Verify Merkle proofs manually
- Experiment with collision search (limited scale)
- Analyze hash distribution properties

**Required Setup:** Python 3.8+, Jupyter Notebook or Python IDE

**Deliverables:** Lab report with code snippets and visualizations

Thank you

Questions?

See you in Lab 4: Hash Experiments