

Lesson 3: Cryptographic Hash Functions

Module A: Blockchain Foundations

BSc Blockchain & Cryptocurrency

University Course

2025

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. Describe the 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)

The Problem: How can we prove data integrity without revealing the data?

Part 1/2: Hash Functions (Fundamentals)

The Challenge

How can we compress any data (a single byte or an entire database) to a fixed-size “fingerprint” that uniquely identifies it and proves it hasn’t been tampered with?

Why It Matters

- Without hash functions, we cannot detect tampering in blockchain blocks
- Every Bitcoin block must prove integrity of thousands of transactions

Today’s lesson: How cryptographic hash functions provide the foundation for blockchain security

What We Need

- Collision resistance (no two inputs produce same output)
- Preimage resistance (cannot reverse the hash)
- Second preimage resistance (cannot forge alternate data)

The Cryptoeconomics Question

How do we create trust in data without a central authority?

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.

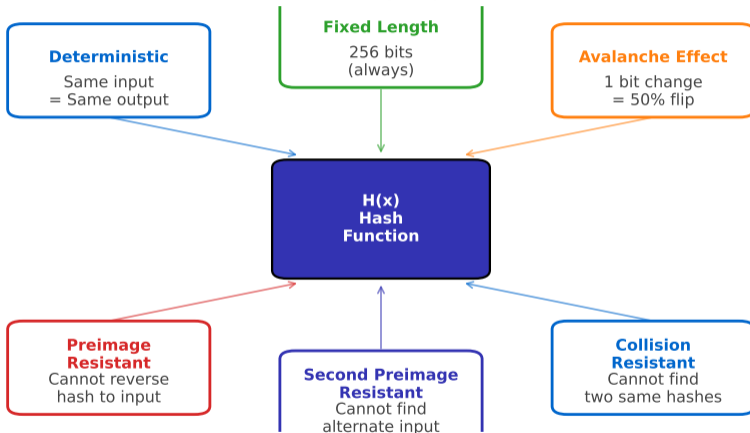
Example (SHA-256):

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

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

Six Essential Properties of Cryptographic Hashes

Six Essential Properties of Cryptographic Hash Functions



Key Insight: All six properties are required for cryptographic security.

Non-Cryptographic vs. Cryptographic Hashes

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)

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)

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)
- $H(\text{1 GB video file}) = 256$ 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)

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):

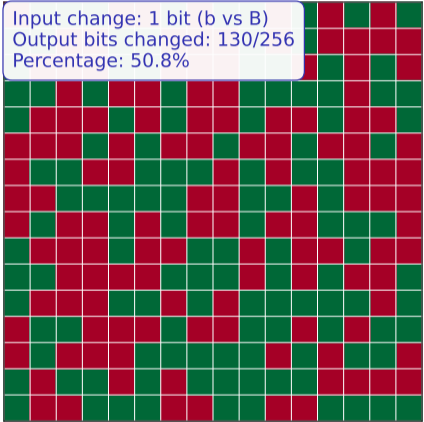
- Input: "blockchain"
 - Hash: ef7797e13d3a75526946a3bcf00daec9fc9c9c4d51ddc7cc5df888f74dd434d1
- Input: "Blockchain" (capital B)
 - Hash: 625da44e4eaf58d61cf048d168aa6f5e492dea166d8bb54ec06c30de07db57e1

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%)

Avalanche Effect: Visual Demonstration

Hash 1: ef7797e13d3a7552...
Hash 2: 6250a44e4ea75806...
Avalanche Effect: SHA-256("blockchain") vs SHA-256("Blockchain")



Green = Same bit | Red = Different bit

Observation: 1-bit input change causes approximately 50% of output bits to flip.

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

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 $H(m_2) = h$

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

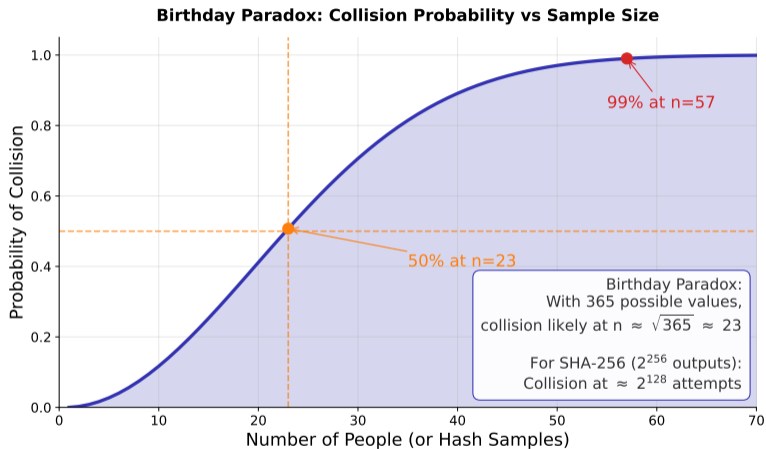
Birthday Paradox Attack:

- 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



Implication: Collision resistance scales as $2^{n/2}$, not 2^n . This is why we need large hash outputs.

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.

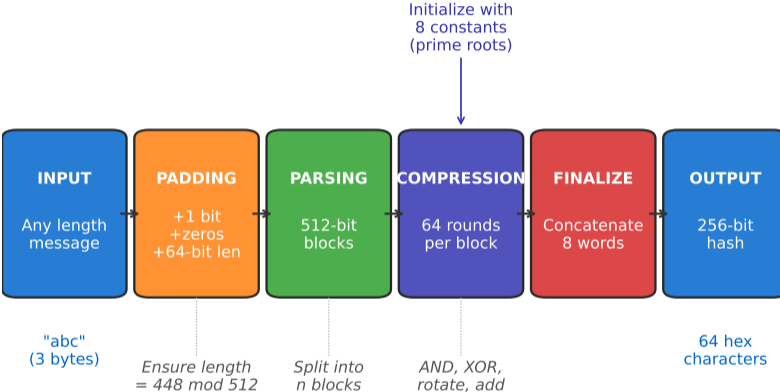
Specifications:

- 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 (SHA256(SHA256(header)))
- Transaction IDs: SHA-256 hash of transaction data
- Address generation: SHA-256 + RIPEMD-160
- Merkle tree construction

SHA-256 Algorithm: High-Level Process Flow



Key Steps: Padding ensures proper length; compression uses 64 rounds of bitwise operations.

SHA-256 Example Calculation

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, Σ_0 , Σ_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-Damgard construction).

Process:

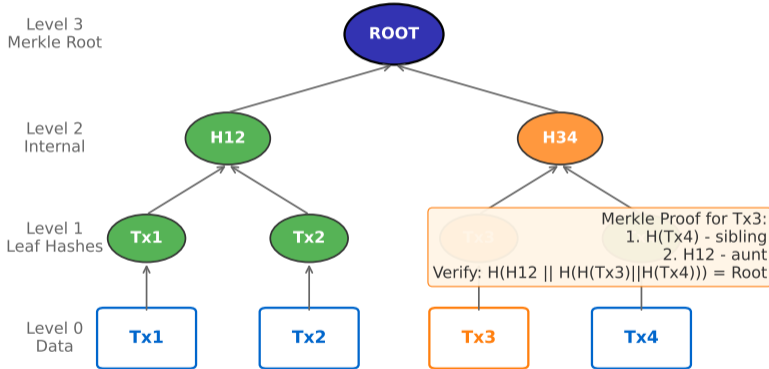
1. Compute $h_1 = \text{SHA256}(\text{data})$
2. Compute $h_2 = \text{SHA256}(h_1)$
3. Use h_2 as final hash

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

Merkle Tree: Hierarchical Hash Structure



Key Insight: Merkle proofs enable $O(\log n)$ verification of transaction inclusion.

Merkle Tree Construction Algorithm

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	≈ 2.5 KB	4 hashes (128 bytes)
100	≈ 25 KB	7 hashes (224 bytes)
1,000	≈ 250 KB	10 hashes (320 bytes)
10,000	≈ 2.5 MB	14 hashes (448 bytes)

Real-World Impact:

- Bitcoin block: $\approx 2,000$ transactions \Rightarrow 11-hash proof (≈ 352 bytes)
- Full block: ≈ 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

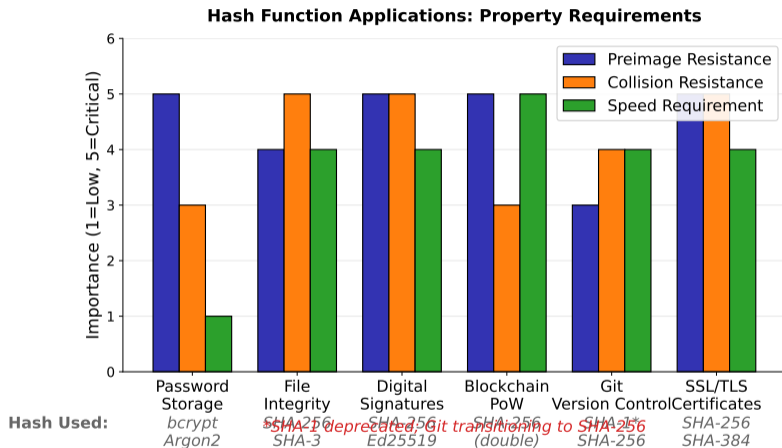
Verkle Trees (Future Ethereum)

- 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

Hash Function Applications Overview



Note: Different applications prioritize different hash properties.

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

How Git Uses Hash Functions:

- Every commit has a SHA-1 hash (Git transitioning to SHA-256)
- Hash is computed from:
 - Commit message
 - Author/committer 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

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: ≈ 500 EH/s (Block 800,000, July 2023; now $\sim 1,000$ EH/s, surpassing 1 ZH/s)

Why Hashes Enable PoW:

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

The Original Problem

How can we prove data integrity without revealing the data?

How Hash Functions Solve It

- SHA-256 provides 256-bit security level (computationally infeasible to break)
- Merkle trees enable efficient verification of large datasets ($O(\log n)$ proofs)
- Collision resistance ensures unique fingerprints for different data

Hash functions solve data integrity verification but face future quantum threats

Remaining Limitations

- Quantum computers threaten current hash functions (Grover's algorithm reduces security)
- MD5 and SHA-1 already broken (collision attacks found)

Open Questions

- When should blockchain migrate to post-quantum hash functions?
- Can we develop hash functions resistant to quantum attacks?
- Risk: Coordination challenge in decentralized networks for hash function upgrades

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 (1987): *A Digital Signature Based on a Conventional Encryption Function*
- 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

Q1. What is the output size of SHA-256?

- A) 128 bits B) 256 bits C) 512 bits D) 1024 bits

Answer: B – SHA-256 produces a fixed 256-bit (32-byte) output, regardless of input size.

Q2. Which property ensures the same input always produces the same hash?

- A) Avalanche effect B) Collision resistance C) Deterministic D) Preimage resistance

Answer: C – Deterministic means no randomness; same input = same output always.

Q3. What percentage of output bits typically flip when 1 input bit changes?

- A) 10% B) 25% C) 50% D) 100%

Answer: C – The avalanche effect causes approximately 50% of output bits to flip.

Q4. Which hash algorithm is broken and no longer considered secure?

- A) SHA-256 B) SHA-3 C) MD5 D) BLAKE2

Answer: C – MD5 collisions were found in 2004; it's not cryptographically secure.

Q5. What does preimage resistance prevent?

- A) Finding input from hash B) Finding two inputs with same hash C) Fast computation D) Fixed output length

Answer: A – Preimage resistance means you cannot reverse a hash to find the original input.

Q6. How many rounds of compression does SHA-256 perform per 512-bit block?

- A) 16 B) 32 C) 64 D) 128

Answer: C – SHA-256 uses 64 rounds of bitwise operations per block.

Q7. Why does Bitcoin use double SHA-256 instead of single hashing?

- A) Faster computation B) Smaller output C) Guard against length-extension attacks D) Easier implementation

Answer: C – Double hashing protects against theoretical length-extension vulnerabilities.

Q8. What is the birthday paradox complexity for finding hash collisions?

- A) 2^n B) $2^{n/2}$ C) 2^{2n} D) $\log n$

Answer: B – Birthday attack reduces collision search from 2^n to $2^{n/2}$ operations.

Q9. For SHA-256, approximately how many operations are needed for birthday attack?

- A) 2^{64} B) 2^{128} C) 2^{256} D) 2^{512}

Answer: B – $2^{256/2} = 2^{128}$ operations, still computationally infeasible.

Q10. What distinguishes second preimage resistance from preimage resistance?

- A) Faster computation B) You're given the original input C) Shorter output D) No difference

Answer: B – Second preimage: find different m_2 for given m_1 where $H(m_1) = H(m_2)$.

Q11. How many hashes are needed for a Merkle proof with 1,000 transactions?

- A) 5 B) 10 C) 100 D) 500

Answer: B – $\log_2(1000) \approx 10$ hashes needed for proof path to root.

Q12. What happens if a Merkle tree has an odd number of leaf nodes?

- A) Error occurs B) Last hash is duplicated C) Tree is invalid D) Extra node is discarded

Answer: B – The last hash is duplicated to create a pair for hashing.

Q13. Which Merkle tree variant does Ethereum use?

- A) Binary Merkle Tree B) Merkle Patricia Trie C) Verkle Tree D) Sparse Merkle Tree

Answer: B – Ethereum uses Merkle Patricia Trie for state storage with key-value support.

Q14. What is the space complexity of a Merkle proof for n transactions?

- A) $O(n)$ B) $O(n^2)$ C) $O(\log n)$ D) $O(1)$

Answer: C – Proof size grows logarithmically with number of transactions.

Q15. In password storage, what is the purpose of salting?

- A) Make hash faster B) Prevent rainbow table attacks C) Reduce output size D) Enable reversibility

Answer: B – Salt (random data) prevents precomputed hash tables (rainbow tables).

Q16. Which hash function is Git transitioning to from SHA-1?

- A) MD5 B) SHA-256 C) SHA-3 D) BLAKE2

Answer: B – Git is migrating to SHA-256 due to SHA-1 collision vulnerabilities.

Q17. What makes collision resistance critical for digital signatures?

- A) Faster signing B) Prevents forged documents with same hash C) Smaller signatures D) Easier verification

Answer: B – Without collision resistance, attacker could create alternate message with identical hash.

Q18. In Bitcoin mining, what must the hash be less than?

- A) Merkle root B) Previous block hash C) Difficulty target D) Nonce value

Answer: C – Hash must be below difficulty target (e.g., certain number of leading zeros).

Q19. What is the block size that SHA-256 processes data in?

- A) 128 bits B) 256 bits C) 512 bits D) 1024 bits

Answer: C – SHA-256 processes input in 512-bit (64-byte) blocks.

Q20. Which quantum algorithm threatens preimage resistance of hash functions?

- A) Shor's algorithm B) Grover's algorithm C) Deutsch-Jozsa D) Simon's algorithm

Answer: B – Grover's algorithm reduces preimage search from 2^{256} to 2^{128} operations.