L02: Cryptographic Foundations

Extended Slides – BSc Blockchain Course

Digital Finance

2026

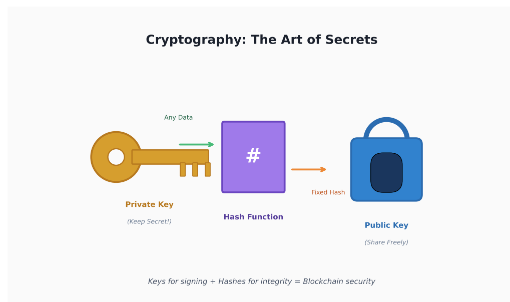
Learning Objectives

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

1. Explain the properties of cryptographic hash functions
2. Describe how digital signatures provide authentication
3. Understand elliptic curve cryptography basics (ECDSA)
4. Explain how Merkle trees enable efficient verification
5. Derive a blockchain address from a public key

Prerequisites: Basic understanding of binary numbers and modular arithmetic.

The Foundation of Trust



Purpose: Cryptography makes blockchain possible. Without hashing, digital signatures, and public-key cryptography, there would be no secure, trustless systems.

Every transaction you make on a blockchain relies on these mathematical guarantees.

Why Cryptography Matters

Cryptography Enables:

- **Integrity:** Detect any tampering with data
- **Authentication:** Prove identity without trusted third party
- **Non-repudiation:** Cannot deny having signed
- **Efficiency:** Verify large datasets quickly

Without Cryptography:

- Anyone could spend others' coins
- Transaction history could be altered
- No way to prove ownership

Blockchain replaces institutional trust with mathematical trust.

What is a Hash Function?

Definition: A function that maps arbitrary-length input to fixed-length output.

Mathematical Notation:

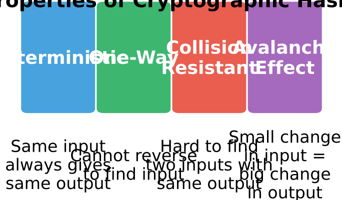
$$H : \{0,1\}^* \rightarrow \{0,1\}^n$$

Example (SHA-256):

- Input: “Hello” (5 bytes)
- Output: 185f8db32271fe25f561a... (32 bytes / 256 bits)
- Input: 1 GB file
- Output: Still 32 bytes / 256 bits

Fixed output size regardless of input size is crucial for efficiency.

Four Key Properties of Cryptographic Hash Functions




```
Example: SHA-256("Hello") = 185f8db32271fe25f561a...
```

All four properties are required for cryptographic security.

Property 1: Deterministic

Same Input = Same Output (Always)

Example:

```
import hashlib
# Run 1
hashlib.sha256(b"Hello").hexdigest()
# '185f8db32271fe25f561a6fc938b2e26...'

# Run 2 (same result)
hashlib.sha256(b"Hello").hexdigest()
# '185f8db32271fe25f561a6fc938b2e26...'
```

Why It Matters:

- Verification is reproducible
- Anyone can independently verify
- No randomness in output

If non-deterministic, verification would be impossible.

Property 2: One-Way (Preimage Resistance)

Given $H(x)$, computationally infeasible to find x

Example:

- Given: 185f8db32271fe25f561a...
- Find: "Hello" (the input)
- Only option: Try all possible inputs (brute force)

Security Level:

- SHA-256: Need 2^{256} attempts (impossible)
- Age of universe: $\sim 4 \times 10^{17}$ seconds
- $2^{256} \approx 10^{77}$ – comparable to atoms in observable universe ($\sim 10^{80}$)

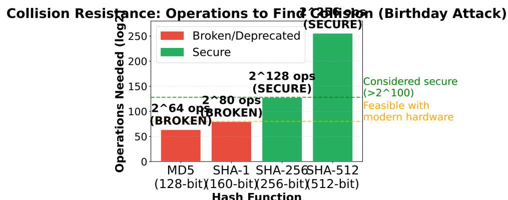
This property ensures passwords cannot be recovered from hashes.

Property 3: Collision Resistance

Hard to find $x \neq y$ **such that** $H(x) = H(y)$

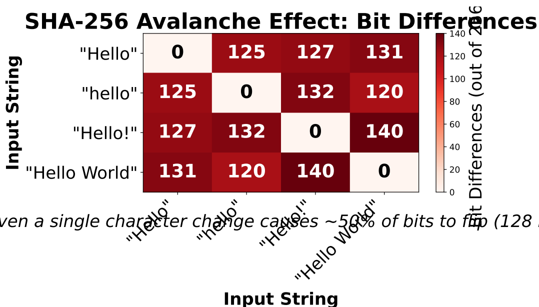
Birthday Paradox:

- 23 people in room: 50% chance two share birthday
- For n-bit hash: collision expected after $\sim 2^{n/2}$ attempts
- SHA-256: 2^{128} attempts (still infeasible)



MD5 and SHA-1 have been broken – collisions found in practice.

Property 4: Avalanche Effect



Even a single character change causes ~50% of bits to flip (128 bits)

Small input change causes unpredictable, large output change.

Hash Functions in Blockchain

Where Hashes Are Used:

1. **Block linking:** Previous block hash in header
2. **Merkle root:** Summarize all transactions
3. **Addresses:** Derived from public keys
4. **Mining:** Proof-of-work puzzle
5. **Script hashes:** P2SH, P2WPKH addresses

Hash Functions Used:

- **Bitcoin:** SHA-256 (double hashed), RIPEMD-160
- **Ethereum:** Keccak-256 (SHA-3 variant)

Hashing is the most frequently executed operation in blockchain.

Public Key Cryptography

Asymmetric Cryptography: Two Keys

- **Private Key:** Secret, used to sign
- **Public Key:** Shared, used to verify

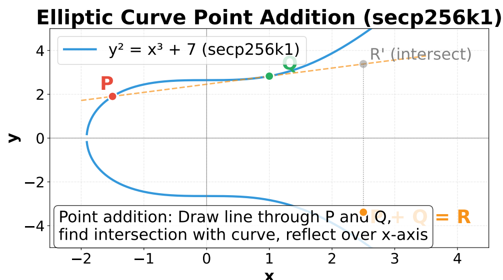
Key Properties:

- Easy: Private key $\rightarrow$ Public key
- Hard: Public key $\rightarrow$ Private key
- Mathematical relationship, not random

Historical Context:

- 1976: Diffie-Hellman key exchange
- 1977: RSA algorithm
- 1985: Elliptic curve cryptography (ECC)

ECC provides same security as RSA with smaller keys.



secp256k1: $y^2 = x^3 + 7$ *over finite field* $\mathbb{F}_p$.

Elliptic Curve Point Addition

Adding Two Points P and Q:

1. Draw line through P and Q
2. Find third intersection point R’
3. Reflect R’ over x-axis to get R = P + Q

Point Doubling (P + P):

- Use tangent line at P
- Same reflection process

Scalar Multiplication:

$$k \cdot G = G + G + ... + G \text{ (k times)}$$

- G is the generator point
- k is the private key
- $k \cdot G$ is the public key

Scalar multiplication is easy; finding k from $k \cdot G$ is the discrete log problem.

secp256k1 Curve Parameters

Bitcoin and Ethereum use secp256k1:

Curve equation: $y^2 = x^3 + 7 \pmod{p}$

Parameters:

- $p = 2^{256} - 2^{32} - 977$ (prime field)
- $n \approx 1.16 \times 10^{77}$ (group order)
- $G =$ generator point (specified coordinates)

Why secp256k1?

- Efficient computation (special prime)
- Not designed by government (vs NIST curves)
- Well-analyzed, no known weaknesses

secp256k1 chosen over NIST curves, possibly due to NSA influence concerns on NIST parameters.

Private key -> Public key (easy) | Public key -> Private key (impossible)

Key Generation Process

256 bits
of entropy

Keep secret
Never share

k * G
(secp256k1)

512 bits
(x,y) coords

Hash +
checksum

PRIVATE (keep secret) | **PUBLIC (share freely)**

Entropy quality is critical – weak randomness leads to stolen funds.

Private Key Security

Private Key = 256 Random Bits

- Any number from 1 to $n - 1$ (curve order)
- Must be truly random (not pseudorandom)
- Must never be reused or shared

Common Attack Vectors:

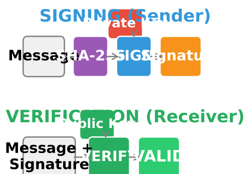
- Weak random number generators
- Predictable seed values
- Reusing k value in ECDSA (PlayStation 3 hack)
- Side-channel attacks (timing, power analysis)

Best Practices:

- Use hardware random number generator
- Use established libraries (never roll your own)
- Store in hardware wallet for large amounts

Millions of dollars have been lost to weak key generation.

Digital Signature Flow



Private key signs, public key verifies. Only the owner can sign, anyone can verify.

Signature proves ownership; verification is non-interactive.

ECDSA Signing Algorithm

Inputs: Message hash z, private key d

Signing Steps:

1. Choose random k (ephemeral key)
2. Calculate point $R = k \cdot G$
3. Set $r = R_x \pmod{n}$
4. Calculate $s = k^{-1}(z + r \cdot d) \pmod{n}$
5. Signature is (r, s)

Critical: k must be random and never reused!

- Same k twice reveals private key
- PlayStation 3 was hacked this way

Modern implementations use deterministic k (RFC 6979).

ECDSA Verification Algorithm

Inputs: Message hash z, signature (r, s), public key Q

Verification Steps:

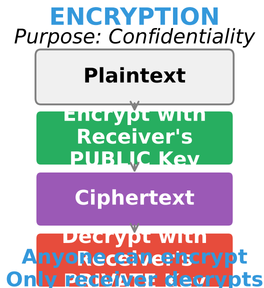
1. Check r, s are in valid range $[1, n-1]$
2. Calculate $w = s^{-1} \pmod{n}$
3. Calculate $u_1 = z \cdot w \pmod{n}$
4. Calculate $u_2 = r \cdot w \pmod{n}$
5. Calculate point $R' = u_1 \cdot G + u_2 \cdot Q$
6. Signature is valid if $R'_x \equiv r \pmod{n}$

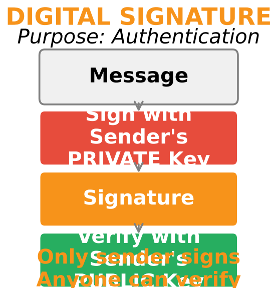
Properties:

- Anyone with public key can verify
- No secret information needed
- Computationally efficient

Verification is slightly faster than signing.

Encryption vs Digital Signatures





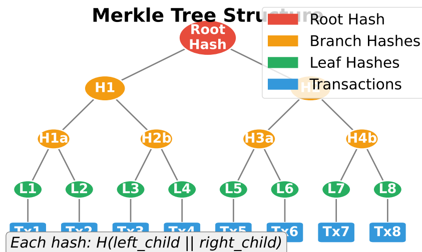
Blockchain uses signing (authentication), not encryption (confidentiality).

What is a Merkle Tree?

- Binary tree of hashes
- Leaves are hashes of data blocks
- Parent = hash of concatenated children
- Root summarizes all data

Invented by Ralph Merkle (1979)

- Originally for digital signatures
- Now used in Git, ZFS, torrents, blockchains



Every block header contains the Merkle root of all transactions.

Merkle Tree Construction

Building from Transactions:

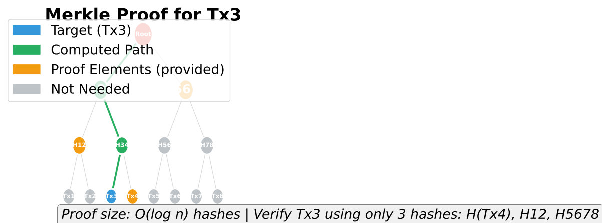
1. Hash each transaction: $L_i = H(Tx_i)$
2. Pair adjacent leaves: $H_{12} = H(L_1 || L_2)$
3. Continue pairing until root
4. If odd number, duplicate last node

Properties:

- Height $= \lceil \log_2(n) \rceil$ for n transactions
- Root changes if any transaction changes
- Efficient updates (only recalculate affected path)

Bitcoin blocks typically have 1000-3000 transactions.

Merkle Proofs



SPV (Simplified Payment Verification) relies on Merkle proofs.

Merkle Proof Efficiency

Proof Size Comparison:

Transactions	Full Block	Merkle Proof
1,000	250 KB	320 bytes (10 hashes)
10,000	2.5 MB	416 bytes (13 hashes)
100,000	25 MB	544 bytes (17 hashes)

Complexity:

- Proof size: $O(\log n)$
- Verification time: $O(\log n)$
- Construction time: $O(n)$

Light clients download only block headers + Merkle proofs.

Address Derivation

Key Differences:

Bitcoin: Double hash + checksum | Ethereum: Single hash, no checksum (EIP-55 for case)

Address is not the public key – it is a hash of the public key.

Bitcoin Address Types

P2PKH (Pay-to-Public-Key-Hash)

- Starts with "1"
- Example: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
- Uses RIPEMD-160(SHA-256(pubkey))

P2SH (Pay-to-Script-Hash)

- Starts with "3"
- Used for multisig, complex scripts

Bech32 (SegWit)

- Starts with "bc1"
- Better error detection
- Lower transaction fees

Bech32 is recommended for new wallets.

Ethereum Addresses

Simpler than Bitcoin:

- Take last 20 bytes of Keccak-256(public key)
- Prefix with "0x"
- Example: 0xAb5801a7D398351b8bE11C439e05C5B3259aeC9B

EIP-55 Checksum:

- Mixed case encoding provides error detection
- Uppercase if corresponding hex digit of hash ≥ 8
- No checksum in address itself (unlike Bitcoin)

Contract Addresses:

- Derived from deployer address + nonce
- Or CREATE2: hash(0xff, deployer, salt, bytecodeHash)

Vitalik's address shown above – easily verifiable on Etherscan.

Security Levels

Bits of Security:

Algorithm	Bits	Status
MD5	64 (broken)	Do not use
SHA-1	80 (broken)	Deprecated
SHA-256	128	Secure
secp256k1	128	Secure
AES-256	256	Secure

What 128-bit security means:

- $2^{128} \approx 3.4 \times 10^{38}$ operations to break
- If 1 billion computers try 1 billion keys/second
- Still takes 10^{13} years (longer than universe age)

Quantum computers may reduce security – see post-quantum cryptography.

Common Cryptographic Mistakes

Implementation Errors:

- Using Math.random() for key generation
- Reusing ECDSA nonce (k value)
- Not validating signature before processing
- Trusting user-provided public keys

Real-World Incidents:

- **Android Bitcoin wallet (2013):** Weak PRNG
- **PlayStation 3 (2010):** Constant ECDSA nonce
- **Blockchain.info (2014):** RNG failure

Rule: Never implement cryptography yourself!

Use audited libraries: libsecp256k1, OpenSSL, NaCl.

Key Takeaways

Remember These Points

1. Hash functions: deterministic, one-way, collision-resistant, avalanche
2. ECDSA: private key signs, public key verifies
3. secp256k1: $y^2 = x^3 + 7$ – used by Bitcoin and Ethereum
4. Merkle trees: $O(\log n)$ verification of transaction inclusion
5. Addresses: hashed public keys with checksums

Next Lesson: Bitcoin Deep Dive – UTXO model, transaction structure, and mining.

Cryptography is the foundation; next we see how Bitcoin uses these primitives.