Lecture Notes: Lesson 2

Cryptography and Security

Cryptocurrency Course

Contents

1 Introduction to Cryptography in Blockchain

Cryptography is the cornerstone of blockchain security. It provides:

- **Data Integrity**: Ensuring data hasn't been tampered with
- **Authentication**: Proving identity without revealing secrets
- **Non-repudiation**: Preventing denial of actions taken
- **Confidentiality**: Protecting sensitive information (in some blockchains)

Core Cryptographic Primitives

Blockchain systems rely on two fundamental cryptographic tools:

1. **Hash Functions**: For data integrity and chain linkage (SHA-256)
2. **Digital Signatures**: For authentication and authorization (ECDSA)

2 SHA-256: Secure Hash Algorithm

2.1 Algorithm Structure

SHA-256 belongs to the SHA-2 family designed by the NSA and published by NIST in 2001. It produces a 256-bit (32-byte) hash from arbitrary-length input.

2.1.1 Internal Architecture

The algorithm operates on 512-bit message blocks through:

1. **Message Padding**:
   - Append bit '1' to message
   - Append '0' bits until length $\equiv 448 \pmod{512}$
   - Append original message length as 64-bit integer
2. **Initialize Hash Values**: Eight 32-bit registers (H0-H7)

$$\begin{aligned} H_0 &= \text{0x6a09e667} \quad H_1 = \text{0xbb67ae85} \\ H_2 &= \text{0x3c6ef372} \quad H_3 = \text{0xa54ff53a} \\ H_4 &= \text{0x510e527f} \quad H_5 = \text{0x9b05688c} \\ H_6 &= \text{0x1f83d9ab} \quad H_7 = \text{0x5be0cd19} \end{aligned}$$

   These are the fractional parts of square roots of first 8 primes.
3. **Message Schedule**: Expand 512-bit block to 64 words ($W_0 \dots W_{63}$)
4. **Compression Function**: 64 rounds of operations using:
   - Logical functions: Ch, Maj, Σ_0, Σ_1, σ_0, σ_1
   - Constants K_t (fractional parts of cube roots of first 64 primes)
   - Modular addition and bitwise operations
5. **Finalization**: Add compressed values to current hash values

2.2 Key Operations

2.2.1 Logical Functions

$$
\begin{aligned}
\mathrm{Ch}(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
\mathrm{Maj}(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x) \\
\Sigma_1(x) &= \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x) \\
\sigma_0(x) &= \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x) \\
\sigma_1(x) &= \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x)
\end{aligned}
$$

where ROTR is rotate right and SHR is shift right.

2.3 Avalanche Effect Demonstration

The avalanche effect ensures that small input changes cascade through the hash, producing drastically different outputs.

Avalanche Effect in SHA-256

```python
import hashlib

def demonstrate_avalanche():
    """Show how 1-bit change affects entire hash"""
    messages = [
        "Hello World",
        "Hello World!",  # Added 1 character
        "hello World",   # Changed 1 bit (H->h)
    ]

    for msg in messages:
        hash_val = hashlib.sha256(msg.encode()).hexdigest()
        print(f"Input:  '{msg}'")
        print(f"SHA256: {hash_val}")
        print()

demonstrate_avalanche()
```

Output:

```
Input:  'Hello World'
SHA256: a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e

Input:  'Hello World!'
SHA256: 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069

Input:  'hello World'
SHA256: 5eb63bbbe01eeed093cb22bb8f5acdc3299ae8ad6e45a5b9b8f0a35a33c15f95
```

Analysis: Changing a single bit flips approximately 50% of the output bits, demonstrating excellent avalanche properties.

2.4 Security Strength

Computational Security

SHA-256's 256-bit output provides:

- **Pre-image Resistance**: 2^{256} operations ($\approx 10^{77}$)
- **Collision Resistance**: 2^{128} operations (Birthday paradox)
- **Current Status**: No practical attacks known

For comparison: There are only $\approx 10^{80}$ atoms in the observable universe.

3 Public Key Cryptography

3.1 Asymmetric Encryption Fundamentals

Unlike symmetric encryption (same key for encryption/decryption), public key cryptography uses a key pair:

- **Private Key** (sk): Secret, used for signing
- **Public Key** (pk): Shared publicly, used for verification

Mathematical Property:

$$\text{Verify}_{pk}(\text{Sign}_{sk}(m)) = \text{True}$$

but knowing pk doesn't reveal sk (computational one-way function).

3.2 Elliptic Curve Cryptography (ECC)

Bitcoin and Ethereum use ECC, specifically the `secp256k1` curve, for digital signatures.

3.2.1 Elliptic Curve Definition

An elliptic curve over finite field $\mathbb{F}_p$ is defined by:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

For `secp256k1`:

- $a = 0$, $b = 7$
- $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- Generator point G
- Order n (prime number of points)

Equation simplifies to:

$$y^2 \equiv x^3 + 7 \pmod{p}$$

3.2.2 Point Addition

The curve has a group structure with point addition operation:

Geometric Interpretation: To add points P and Q:

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

Algebraic Formula: For $P = (x_1, y_1)$ and $Q = (x_2, y_2)$:

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P = Q \text{ (point doubling)} \end{cases}$$

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

3.2.3 Scalar Multiplication

Key operation: $Q = k \cdot G$ (add G to itself k times)

Efficient Computation: Use double-and-add algorithm in $O(\log k)$ time.

Elliptic Curve Discrete Logarithm Problem (ECDLP)

Given $Q = k \cdot G$, it's computationally infeasible to find k (the private key).
This is the foundation of ECC security. Best known attacks require $O(\sqrt{n})$ time, where $n \approx 2^{256}$ for secp256k1.

4 Digital Signatures with ECDSA

4.1 Elliptic Curve Digital Signature Algorithm

ECDSA provides authentication and non-repudiation for blockchain transactions.

4.1.1 Key Generation

1. Choose random private key: $sk \in [1, n-1]$
2. Compute public key: $pk = sk \cdot G$

Key Generation Example

```python
from ecdsa import SigningKey, SECP256k1
import hashlib

# Generate private key
private_key = SigningKey.generate(curve=SECP256k1)

# Derive public key
public_key = private_key.get_verifying_key()

# Export as hex
sk_hex = private_key.to_string().hex()
pk_hex = public_key.to_string().hex()

print(f"Private Key: {sk_hex}")
print(f"Public Key:  {pk_hex}")

# Generate Bitcoin-style address
pk_hash = hashlib.sha256(public_key.to_string()).digest()
address_hash = hashlib.new('ripemd160', pk_hash).hexdigest()
print(f"Address (simplified): {address_hash}")
```

4.1.2 Signature Generation

To sign message m with private key sk:

1. Hash the message: $e = H(m)$ (where H is SHA-256)
2. Choose random nonce: $k \in [1, n-1]$
3. Compute point: $P = k \cdot G = (x, y)$
4. Calculate $r = x \bmod n$
5. Calculate $s = k^{-1}(e + r \cdot sk) \bmod n$
6. Signature is pair: (r, s)

Nonce Security Critical

The nonce k must be:

- **Random**: Predictable k leaks private key
- **Unique**: Reusing k for different messages leaks private key
- **Secret**: Revealing k leaks private key

Historical disaster: PlayStation 3 used constant k, leading to complete system compromise in 2010.

4.1.3 Signature Verification

Given message m, signature (r, s), and public key pk:

1. Hash the message: $e = H(m)$

2. Calculate: $w = s^{-1} \bmod n$
3. Calculate: $u_1 = e \cdot w \bmod n$
4. Calculate: $u_2 = r \cdot w \bmod n$
5. Calculate point: $P = u_1 \cdot G + u_2 \cdot pk = (x, y)$
6. Verify: $r \equiv x \pmod{n}$

If $r \equiv x \pmod{n}$, signature is valid.

4.2 Mathematical Proof of Correctness

Why ECDSA Works

During signing:

$$s = k^{-1}(e + r \cdot sk) \implies k = s^{-1}(e + r \cdot sk)$$

During verification:

$$\begin{aligned} P &= u_1 \cdot G + u_2 \cdot pk \\ &= (e \cdot s^{-1}) \cdot G + (r \cdot s^{-1}) \cdot (sk \cdot G) \\ &= s^{-1}(e + r \cdot sk) \cdot G \\ &= k \cdot G \quad \text{(same point computed during signing)} \end{aligned}$$

Therefore, x-coordinate matches r, confirming authenticity.

5 Wallet Security

5.1 Wallet Types

5.1.1 Hot Wallets

Connected to the internet:

- **Web Wallets**: Browser-based (e.g., MetaMask)
- **Mobile Wallets**: Smartphone apps
- **Desktop Wallets**: Computer software

Pros: Convenient for frequent transactions
Cons: Vulnerable to malware, phishing, server breaches

5.1.2 Cold Wallets

Offline storage:

- **Hardware Wallets**: Dedicated devices (Ledger, Trezor)
- **Paper Wallets**: Printed private keys
- **Steel Wallets**: Engraved metal plates

Pros: Maximum security for long-term storage
Cons: Less convenient for frequent use

5.2 Hierarchical Deterministic Wallets (HD Wallets)

HD wallets (BIP-32 standard) generate multiple addresses from a single seed.

5.2.1 Derivation Path

```
Master Seed (128-256 bits)
    |
    v
Master Private Key (m)
    |
    +--> Child Key (m/0)
    |       |
    |       +--> Grandchild (m/0/0)
    |
    +--> Child Key (m/1)
            |
            +--> Grandchild (m/1/0)
```

Standard Paths:

- Bitcoin: `m/44'/0'/0'/0/i`
- Ethereum: `m/44'/60'/0'/0/i`

5.2.2 Seed Phrase (Mnemonic)

BIP-39 encodes seed as 12-24 words from a 2048-word dictionary.

Mnemonic Example

```
witch collapse practice feed shame open despair
creek road again ice least
```

This 12-word phrase represents 128 bits of entropy and can recover the entire wallet.

Mnemonic Security

Best Practices:

- Never store digitally (no photos, no cloud)
- Write on paper, store in secure location
- Consider metal backup for fire/water resistance
- Never share with anyone
- Beware of "recovery" scams asking for your phrase

5.3 Address Generation

5.3.1 Bitcoin Address

1. Start with public key (33 or 65 bytes)
2. Apply SHA-256: $h_1 = \text{SHA256}(pk)$

3. Apply RIPEMD-160: $h_2 = \text{RIPEMD160}(h_1)$
4. Add version byte (0x00 for mainnet)
5. Compute checksum: $c = \text{SHA256}(\text{SHA256}(\text{version}||h_2))[:4]$
6. Encode in Base58: $\text{Base58}(\text{version}||h_2||c)$

Result: Address like `1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa`

5.3.2 Ethereum Address

1. Take public key (64 bytes, uncompressed, without prefix)
2. Apply Keccak-256 hash
3. Take last 20 bytes
4. Add `0x` prefix

Result: Address like `0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb`

5.4 Multi-Signature Wallets

Require m out of n signatures to authorize transactions.

Example: 2-of-3 multisig

- Three key holders (e.g., you, partner, lawyer)
- Any two can authorize transaction
- Single key compromise doesn't lose funds

Use Cases:

- Corporate funds (require multiple executives)
- Escrow services
- Personal backup (key redundancy)

6 Common Security Threats

6.1 Private Key Compromise

Attack Vectors:

- Malware/keyloggers
- Phishing websites
- Clipboard hijacking
- Weak random number generators
- Physical theft

Mitigation:

- Use hardware wallets for significant holdings
- Verify addresses character-by-character
- Use operating system with verified integrity
- Never enter private keys on websites

6.2 Transaction Malleability

Before SegWit, Bitcoin transaction IDs could be modified without invalidating signatures.

Problem: Attackers could change transaction ID, causing wallet confusion.

Solution: Segregated Witness (SegWit) separates signature data, fixing transaction ID calculation.

6.3 Replay Attacks

After a blockchain fork, transactions can be valid on both chains.

Example: Bitcoin/Bitcoin Cash split (2017)

- Sending BTC on Bitcoin chain also sent BCH on Bitcoin Cash chain
- Users lost funds on one chain

Mitigation: Replay protection mechanisms, split coins intentionally

7 Study Questions

7.1 Conceptual Questions

1. Explain why reusing the same nonce k in ECDSA is catastrophic.
2. How does the avalanche effect contribute to blockchain security?
3. What is the difference between pre-image resistance and collision resistance?
4. Why can't someone derive a private key from a public key in ECC?
5. Describe the trade-offs between hot and cold wallet storage.

7.2 Mathematical Problems

1. If a hash function has 160-bit output, estimate the number of hashes needed to find a collision (use birthday paradox).
2. Given ECDSA signature (r, s) and two messages m_1, m_2 signed with the same nonce k, derive a formula to recover the private key sk.
3. Calculate the probability that a random private key matches a specific public key for secp256k1 (order $n \approx 2^{256}$).
4. How many bits of security does SHA-256 provide against collision attacks? Against pre-image attacks?

7.3 Practical Exercises

1. Implement ECDSA signature generation and verification using a cryptography library.
2. Create a program to demonstrate the avalanche effect by measuring Hamming distance between hashes of similar inputs.
3. Write code to generate Bitcoin addresses from public keys (including Base58Check encoding).
4. Build a simple HD wallet that derives multiple addresses from a single seed.

8 Further Reading

- NIST FIPS 180-4: Secure Hash Standard (SHA-256 specification)
- NIST FIPS 186-4: Digital Signature Standard (ECDSA specification)
- Koblitz, N. (1987). *Elliptic curve cryptosystems*. Mathematics of Computation.
- BIP-32: Hierarchical Deterministic Wallets
- BIP-39: Mnemonic code for generating deterministic keys
- Antonopoulos, A. M. (2017). *Mastering Bitcoin*, Chapter 4: Keys, Addresses.