# Cryptography for Blockchain

## Lesson 2 of 4: Cryptocurrency & Blockchain Development

Prof. Dr. Joerg Osterrieder

University Lecture Series

January 25, 2026

# Lesson Outline

**Duration:** 45 minutes — **Prerequisites:** Lesson 1 (Blockchain Fundamentals)

# What is a Hash Function?

**Definition**

A hash function is a mathematical algorithm that:

- Takes input of **any size**
- Produces output of **fixed size**
- Acts as a "digital fingerprint"

**Analogy**

Like a fingerprint scanner:

- Every person $\rightarrow$ unique fingerprint
- Every file $\rightarrow$ unique hash
- Cannot reconstruct person from fingerprint

**Mathematical Notation**

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

Where:

- $H$ = hash function
- $\{0,1\}^*$ = arbitrary-length binary input
- $\{0,1\}^n$ = fixed-length $n$-bit output

**Example (SHA-256)**

- Input: "Hello" (5 bytes)
- Output: 256 bits (32 bytes)
- Always 64 hex characters

**Hash functions are fundamental building blocks of blockchain security**

# Essential Properties of Cryptographic Hash Functions

**1. Deterministic**
Same input $\rightarrow$ same output

- Reproducible results
- No randomness involved
- Essential for verification

$$H(x) = H(x) \text{ always}$$

**2. One-Way (Pre-image Resistant)**
Cannot reverse the hash

- Given $h = H(x)$
- Infeasible to find $x$
- Protects original data

$$H(x) \rightarrow h\checkmark$$

$$h \rightarrow x\times$$

**3. Collision-Resistant**
Hard to find two inputs with same hash

- Given $x$, hard to find $y$
- Where $H(x) = H(y)$
- Ensures uniqueness

$$P(H(x) = H(y)) \approx 0$$

| Property | Security Guarantee | Blockchain Use |
|---|---|---|
| Deterministic | Verification | Block validation |
| One-way | Data protection | Transaction privacy |
| Collision-resistant | Integrity | Unique block IDs |

**Breaking any of these properties would compromise blockchain security**

# SHA-256: The Blockchain Standard

### SHA-256 Specifications

- **Name:** Secure Hash Algorithm 256-bit
- **Family:** SHA-2 (designed by NSA)
- **Output:** 256 bits = 32 bytes = 64 hex chars
- **Block size:** 512 bits
- **Rounds:** 64 compression rounds

### Security Level

- $2^{256}$ possible outputs
- More than atoms in universe ($\approx 10^{80}$)
- Brute force: infeasible
- No known practical attacks

### Example Hash

Input: `"Bitcoin"`

SHA-256 Output:

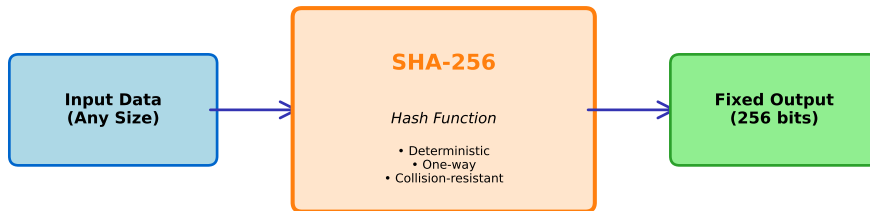
```
b4056df6691f8dc7...
...2e6d7c26c374ef0
```

### Why Bitcoin Uses SHA-256

- Block header hashing
- Merkle tree construction
- Proof-of-Work puzzles
- Address generation (with RIPEMD-160)

**Fun Fact:** Bitcoin mining is essentially a race to find an input that produces a hash starting with many zeros.

## Hash Function Process




Example:
Input: "Hello, World!"
Output: dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f

*Key Properties:*
*1. Same input always produces same output*
*2. Cannot reverse engineer input from output*
*3. Tiny input change drastically changes output*

# The Avalanche Effect

**Definition**
A small change in input causes a dramatic change in output.

**Example**
Input 1: `"Hello, Blockchain!"`
Input 2: `"Hello, Blockchain."`
(Only difference: `!` vs `.`)

**Result**

- Completely different hashes
- ≈50% of bits change
- No pattern or predictability

SHA-256 Avalanche Effect
*Tiny input change → Drastically different output*

Input 1: "Bitcoin"
Hash 1: b4056df6691f8dc72e56302ddad345d6...

Input 2: "bitcoin"
Hash 2: 6b88c087247aa2f07ee1c5956b8e1a9f...

← Only 1 char different!

Binary Bit Comparison (first 64 bits shown)
Hash 1:
Hash 2:

Bit Differences: 138 out of 256 bits (53.9%)
Red boxes show differing bits

*The Avalanche Effect ensures that even the smallest change in input produces a completely different, unpredictable hash output.*

## Why This Matters for Blockchain

+ Tampering is immediately detectable
+ Cannot predict hash without computing it
+ Mining difficulty can be precisely controlled

**The avalanche effect ensures that even tiny modifications are immediately apparent**

# Code Demo: Hashing in Python

### Basic SHA-256 Hashing

```python
import hashlib

def sha256_hash(data: str) -> str:
    """Compute SHA-256 hash."""
    return hashlib.sha256(
        data.encode()
    ).hexdigest()

# Example usage
text = "Hello, Blockchain!"
hash_value = sha256_hash(text)
print(f"Input: {text}")
print(f"Hash:  {hash_value}")
```

### Demonstrating Avalanche Effect

```python
text1 = "Hello, Blockchain!"
text2 = "Hello, Blockchain."

hash1 = sha256_hash(text1)
hash2 = sha256_hash(text2)

# Count differing bits
diff = bin(
    int(hash1, 16) ^ int(hash2, 16)
).count('1')

print(f"Bits changed: {diff}/256")
# Output: ~128 bits (50%)
```

**Try it:** Run `python code/hashing_demo.py` to see these examples in action

# Why Blockchain Needs Hashing

**Block Identification**

Each block has a unique hash:

- Block hash = fingerprint
- Links to previous block
- Creates immutable chain

**Data Integrity**

- Hash of transactions (Merkle root)
- Any change → different hash
- Instant tamper detection

**Proof-of-Work Mining**

- Find nonce where:
- $H(\text{block}) < \text{target}$
- Requires massive computation
- Provides security

**Address Generation**

- Public key → Hash → Address
- Shorter, more manageable
- Additional privacy layer

| Use Case | Hash Function | Purpose |
|---|---|---|
| Bitcoin blocks | SHA-256 (double) | Block ID & PoW |
| Merkle trees | SHA-256 | Transaction verification |
| Bitcoin addresses | SHA-256 + RIPEMD-160 | Address derivation |
| Ethereum | Keccak-256 | All hashing operations |

**Without hash functions, blockchain integrity would be impossible to maintain**

# Symmetric vs. Asymmetric Encryption

**Symmetric Encryption**
Same key for encrypt & decrypt

- \+ Fast and efficient
- \+ Simple implementation
- \- Key distribution problem
- \- Need secure channel to share key

**Examples:** AES, DES, ChaCha20

$$E_k(m) = c \quad D_k(c) = m$$

**Asymmetric Encryption**
Different keys: public & private

- \+ No key distribution problem
- \+ Enables digital signatures
- \- Slower than symmetric
- \- Computationally intensive

**Examples:** RSA, ECDSA, Ed25519

$$E_{pub}(m) = c \quad D_{priv}(c) = m$$

**Blockchain uses asymmetric cryptography** because users need to prove ownership without revealing their private keys to anyone.

**Public key cryptography solved the key distribution problem that plagued symmetric encryption**

# Understanding Key Pairs

**Private Key**

- Secret number (256 bits for ECDSA)
- Generated randomly
- **Never share!**
- Used to sign transactions
- Proves ownership of funds

**Think of it as:**

- Password to your bank vault
- Master key to everything
- Irreplaceable if lost

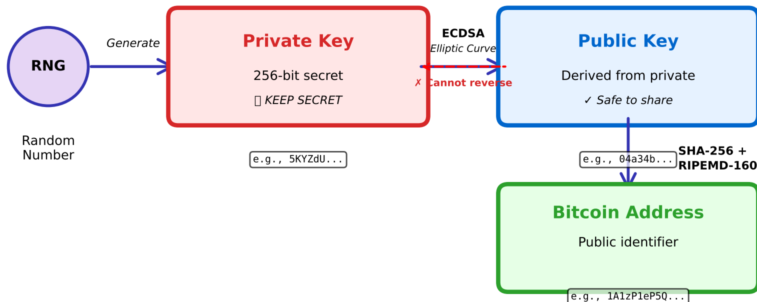
**Public Key**

- Derived from private key
- Can be shared freely
- Used to verify signatures
- Basis for wallet address
- Cannot derive private key from it

**Think of it as:**

- Your email address
- Bank account number
- Safe to share publicly

| Aspect | Private Key | Public Key |
|---|---|---|
| Visibility | Secret | Public |
| Purpose | Sign | Verify |
| Derivation | Random | From private key |
| Loss impact | Funds lost forever | None (regenerate) |




Security Properties:
- Private key must remain secret
- Public key can be freely shared
- Cannot derive private from public
- One private key → One public key

Use Cases:
- Private key: Sign transactions
- Public key: Verify signatures
- Address: Receive Bitcoin

# How Keys Work in Blockchain Transactions

**Sending Cryptocurrency**

1. Create transaction message
2. Sign with **private key**
3. Broadcast to network
4. Miners verify with **public key**
5. Transaction added to block

**Why This Works**

- Only private key holder can sign
- Anyone can verify with public key
- Proves ownership without revealing key

**Receiving Cryptocurrency**

1. Share your **address** (derived from public key)
2. Sender creates transaction to your address
3. Transaction references your public key
4. Only you can spend (with private key)

**Security Model**

- No central authority needed
- Math guarantees authenticity
- Cryptographic proof of ownership

**Key Insight:** The blockchain never stores private keys. It only records transactions signed by private keys and verifiable by public keys.

# Elliptic Curve Cryptography (ECC)

**What is ECC?**
Mathematical system based on elliptic curves:

$$y^2 = x^3 + ax + b$$

**Why Blockchain Uses ECC**

- Smaller keys than RSA
- Same security level
- Faster operations
- Less storage needed

**Security Comparison**

| RSA | ECC |
|---|---|
| 3072 bits | 256 bits |
| 7680 bits | 384 bits |

**SECP256k1 Curve**
Used by Bitcoin & Ethereum:

- $a = 0,\ b = 7$
- $y^2 = x^3 + 7$
- 256-bit keys
- Not NSA-designed (transparency)

**Key Generation (Simplified)**

1. Choose random private key $d$
2. Compute public key $Q = d \times G$
3. $G =$ generator point on curve
4. Reverse computation infeasible

**ECC provides equivalent security to RSA with much smaller key sizes, making it ideal for blockchain**

# Code Demo: Key Generation in Python

## Generate ECDSA Key Pair

```python
from ecdsa import SigningKey, SECP256k1

def create_keypair():
    """Generate new key pair."""
    # Private key (random)
    private_key = SigningKey.generate(
        curve=SECP256k1
    )

    # Public key (derived)
    public_key = private_key.get_verifying_key()

    return private_key, public_key

# Generate keys
priv, pub = create_keypair()
print(f"Private: {priv.to_string().hex()}")
print(f"Public:  {pub.to_string().hex()}")
```

## Key Properties

```python
# Private key: 32 bytes (256 bits)
private_hex = priv.to_string().hex()
print(f"Length: {len(private_hex)} hex chars")
# Output: 64 hex chars

# Public key: 64 bytes (512 bits)
# (uncompressed: x + y coordinates)
public_hex = pub.to_string().hex()
print(f"Length: {len(public_hex)} hex chars")
# Output: 128 hex chars

# Compressed public key: 33 bytes
# (x coordinate + parity flag)
```

## Required Library

```
pip install ecdsa
```

# Security Considerations

**Random Number Generation**

- ! Private key must be truly random
- ! Poor RNG = compromised security
- ✓ Use cryptographic RNG
- ✓ Hardware entropy sources

**Key Storage Risks**

- Digital theft (malware, hacking)
- Physical theft (device stolen)
- Loss (hardware failure, forgotten)
- Social engineering attacks

**Best Practices**

- ✓ Use hardware wallets for large amounts
- ✓ Backup seed phrases securely
- ✓ Use strong encryption
- ✓ Enable 2FA where possible
- ✓ Verify addresses before sending

**Quantum Computing Threat**

- Shor's algorithm threatens ECC
- Post-quantum cryptography emerging
- Bitcoin/Ethereum monitoring this

**Remember:** "Not your keys, not your coins." Self-custody requires understanding and responsibility.

# What is a Digital Signature?

**Definition**

A mathematical scheme that:

- Proves message authenticity
- Verifies sender identity
- Ensures message integrity
- Provides non-repudiation

**Real-World Analogy**

Like a handwritten signature but:

- Cannot be forged (mathematically)
- Tied to specific message
- Can be verified by anyone
- Different for each message

**Components**

1. **Key Generation:** Create key pair
2. **Signing:** $\sigma = Sign(sk, m)$
3. **Verification:** $Verify(pk, m, \sigma)$

**Properties**

- Only private key holder can sign
- Anyone with public key can verify
- Signature is unique to message
- Cannot extract private key from signature

| Property | Physical Signature | Digital Signature |
|---|---|---|
| Forgery resistance | Low | Very high |
| Message binding | No | Yes |
| Verification | Subjective | Mathematical |

# Why Blockchain Transactions Need Signatures

## Authorization Without Trust

In traditional finance:

- Bank verifies your identity
- PIN/password authenticates
- Centralized trust model

In blockchain:

- No central authority
- Signature proves ownership
- Math replaces trust

## What Gets Signed

- Sender address
- Recipient address
- Amount to transfer
- Transaction metadata

## Protection Provided

- ✓ **Authentication:** Proves sender identity
- ✓ **Integrity:** Detects tampering
- ✓ **Non-repudiation:** Cannot deny sending
- ✓ **Authorization:** Only owner can spend

## Without Signatures

- × Anyone could spend your coins
- × No proof of transaction origin
- × Easy to forge transactions
- × Blockchain would be useless

**Digital signatures are the foundation of ownership in decentralized systems**

# ECDSA: Elliptic Curve Digital Signature Algorithm

**Signing Process**

Given message $m$ and private key $d$:

1. Compute $e = Hash(m)$
2. Select random $k$
3. Calculate point $(x_1, y_1) = k \times G$
4. Compute $r = x_1 \mod n$
5. Compute $s = k^{-1}(e + d \cdot r) \mod n$
6. Signature $= (r, s)$

**Output**

- Two 256-bit numbers: $r$ and $s$
- Total signature: 64 bytes
- Unique for each message

**Verification Process**

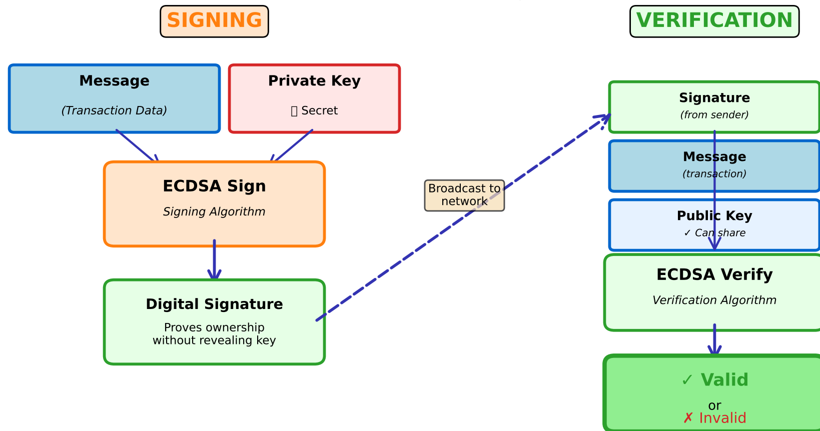
Given message $m$, signature $(r, s)$, public key $Q$:

1. Compute $e = Hash(m)$
2. Compute $u_1 = e \cdot s^{-1} \mod n$
3. Compute $u_2 = r \cdot s^{-1} \mod n$
4. Calculate $(x_1, y_1) = u_1 \times G + u_2 \times Q$
5. Valid if $r = x_1 \mod n$

**Security Basis**

- Discrete logarithm problem
- Cannot find $d$ from $Q = d \times G$

**Note:** The random value $k$ must be truly random and unique for each signature. Reusing $k$ can leak the private key!


Digital Signature Flow
How Bitcoin transactions are signed and verified
SIGNING
VERIFICATION
Message
(Transaction Data)
Private Key
Secret
ECDSA Sign
Signing Algorithm
Digital Signature
Proves ownership
without revealing key
Broadcast to
network
Signature
(from sender)
Message
(transaction)
Public Key
✓ Can share
ECDSA Verify
Verification Algorithm
✓ Valid
or
✗ Invalid
Key Properties:
• Only private key holder can create valid signature

# Code Demo: Signing a Message

## Create Signature

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

def sign_message(private_key, message):
    """Sign a message with ECDSA."""
    # Hash the message
    msg_hash = hashlib.sha256(
        message.encode()
    ).digest()

    # Sign the hash
    signature = private_key.sign_digest(
        msg_hash
    )

    return signature
```

## Usage Example

```python
# Generate key pair
private_key = SigningKey.generate(
    curve=SECP256k1
)

# Message to sign
message = "Transfer 10 BTC to Alice"

# Create signature
signature = sign_message(
    private_key,
    message
)

print(f"Message: {message}")
print(f"Signature: {signature.hex()}")
# Signature: 64 bytes (r + s)
```

**Important:** The signature is unique to this specific message. Changing even one character would require a new signature.

# Code Demo: Verifying a Signature

## Verification Function

```python
from ecdsa import BadSignatureError

def verify_signature(public_key,
                     message,
                     signature):
    """Verify ECDSA signature."""
    try:
        msg_hash = hashlib.sha256(
            message.encode()
        ).digest()

        public_key.verify_digest(
            signature,
            msg_hash
        )
        return True
    except BadSignatureError:
        return False
```

## Testing Verification

```python
# Get public key from private key
public_key = private_key.get_verifying_key()

# Verify original message
is_valid = verify_signature(
    public_key, message, signature
)
print(f"Original: {is_valid}")  # True

# Try tampered message
tampered = "Transfer 100 BTC to Alice"
is_valid = verify_signature(
    public_key, tampered, signature
)
print(f"Tampered: {is_valid}")  # False

# Try wrong public key
wrong_pk = SigningKey.generate(
    curve=SECP256k1
).get_verifying_key()
is_valid = verify_signature(
    wrong_pk, message, signature
)
print(f"Wrong key: {is_valid}")  # False
```

# Non-Repudiation: The Legal Power of Signatures

**What is Non-Repudiation?**
The inability to deny having performed an action.

**In Blockchain Context**

- Cannot deny signing a transaction
- Signature proves you authorized it
- Recorded permanently on chain
- Cryptographic evidence

**Why It Matters**

- Legal accountability
- Dispute resolution
- Audit trails
- Trust without intermediaries

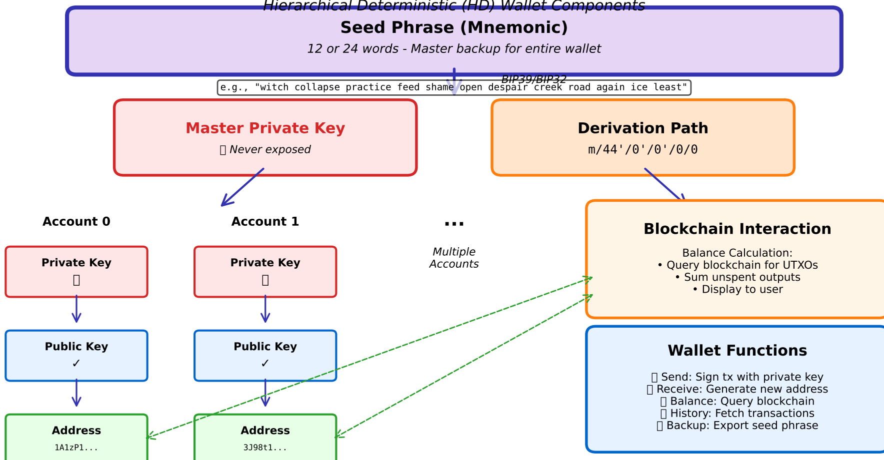
**How It Works**

1. Only you have your private key
2. Only private key can create valid signature
3. Signature is publicly verifiable
4. Therefore: you must have signed it

**Implications**

- ! Key theft = transaction theft
- ! No "undo" or "cancel"
- ! No "I forgot my password" recovery
- ✓ Full control of your assets
- ✓ No middleman needed

**Legal Note:** In many jurisdictions, digital signatures have the same legal standing as handwritten signatures for contracts and agreements.


Bitcoin Wallet Architecture
Hierarchical Deterministic (HD) Wallet Components
Seed Phrase (Mnemonic)
12 or 24 words - Master backup for entire wallet
BIP39/BIP32
e.g., "witch collapse practice feed shame open despair creek road again ice least"
Master Private Key
Never exposed
Derivation Path
m/44'/0'/0'/0/0
Account 0
Account 1
...
Multiple
Accounts
Private Key
Public Key
✓
Address
1A1zP1...
3J98t1...
Blockchain Interaction
Balance Calculation:
• Query blockchain for UTXOs
• Sum unspent outputs
• Display to user
Wallet Functions
Send: Sign tx with private key
Receive: Generate new address
Balance: Query blockchain
History: Fetch transactions
Backup: Export seed phrase
⚠ Security: Never share seed phrase or private keys!
Whoever has the seed can control ALL funds in the wallet.

# From Seed Phrase to Wallet Address

### The Derivation Chain

1. **Entropy** (128-256 bits)
   Random data from secure source
2. **Mnemonic Seed** (12-24 words)
   Human-readable backup (BIP-39)
3. **Seed** (512 bits)
   PBKDF2 stretch of mnemonic
4. **Master Key**
   Root of HD wallet tree (BIP-32)
5. **Child Keys**
   Derived for each address
6. **Public Key**
   From private key (ECC)
7. **Address**
   Hash of public key

### Example Seed Phrase

`abandon abandon abandon abandon abandon abandon abandon abandon abandon abandon abandon about`

### Why Seed Phrases?

- ✓ Human-readable backup
- ✓ Generate unlimited addresses
- ✓ Restore entire wallet
- ✓ Standardized (works across wallets)

### Security Warning

- ! Anyone with seed phrase owns your funds
- ! Store offline, never digitally
- ! Use metal backup for fire/water protection

# Code Demo: Generate Wallet Address

### Key Generation

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

def generate_keypair():
    """Generate ECDSA key pair."""
    private = SigningKey.generate(
        curve=SECP256k1
    )
    public = private.get_verifying_key()
    return (
        private.to_string().hex(),
        public.to_string().hex()
    )

priv_key, pub_key = generate_keypair()
print(f"Private Key: {priv_key}")
print(f"Public Key:  {pub_key}")
```

### Address Derivation (Ethereum-style)

```python
def derive_address(public_key_hex):
    """Derive address from public key."""
    # Convert to bytes
    pub_bytes = bytes.fromhex(
        public_key_hex
    )

    # Hash with Keccak-256 (simplified)
    # Using SHA-256 for demo
    hash_digest = hashlib.sha256(
        pub_bytes
    ).digest()

    # Take last 20 bytes, add 0x
    address = '0x' + hash_digest[-20:].hex()
    return address

address = derive_address(pub_key)
print(f"Address: {address}")
# Example: 0x7a250d5630b4cf539739df...
```

**Try it:** Run `python code/wallet_demo.py` for the complete demonstration

# Hot Wallets vs. Cold Wallets

**Hot Wallets**
Connected to the internet

**Types:**

- Mobile apps (Trust Wallet, Metamask Mobile)
- Browser extensions (Metamask, Phantom)
- Exchange wallets (Coinbase, Binance)
- Desktop wallets (Exodus, Electrum)

**Characteristics:**

+ Convenient for daily use
+ Easy transactions
- Vulnerable to hacks
- Malware risk

**Cold Wallets**
Offline storage

**Types:**

- Hardware wallets (Ledger, Trezor)
- Paper wallets (printed keys)
- Air-gapped computers
- Steel/metal backups

**Characteristics:**

+ Maximum security
+ Immune to online attacks
- Less convenient
- Physical security needed

| Use Case | Recommended | Amount |
|---|---|---|
| Daily spending | Hot wallet | Small amounts |
| Short-term trading | Hot wallet | Medium amounts |
| Long-term holding | Cold wallet | Large amounts |
| Inheritance planning | Cold + documentation | Any amount |

# Wallet Security Best Practices

**Seed Phrase Protection**

- ✓ Write on paper immediately
- ✓ Store in multiple locations
- ✓ Consider metal backup
- × Never store digitally
- × Never screenshot
- × Never email/message

**Transaction Safety**

- ✓ Always verify addresses
- ✓ Send test transactions first
- ✓ Double-check amounts
- ✓ Understand gas fees

**Device Security**

- ✓ Keep software updated
- ✓ Use antivirus protection
- ✓ Enable device encryption
- ✓ Use strong passwords
- ✓ Enable 2FA everywhere

**Avoid Common Scams**

- ! Phishing sites (check URLs!)
- ! Fake customer support
- ! "Free" crypto giveaways
- ! Clipboard hijacking malware

**Golden Rule:** If someone asks for your seed phrase or private key, it's a scam. No legitimate service will ever request this.

# Key Takeaways

**Hash Functions**

- Deterministic, one-way, collision-resistant
- SHA-256: blockchain standard
- Avalanche effect ensures security
- Used for blocks, Merkle trees, PoW

**Public Key Cryptography**

- Private key: secret, used to sign
- Public key: shareable, used to verify
- ECC (SECP256k1) used in blockchain
- Enables trustless ownership

**Digital Signatures**

- ECDSA: sign transactions
- Proves ownership without revealing key
- Non-repudiation: cannot deny signing
- Foundation of blockchain security

**Wallets**

- Key management, not coin storage
- Seed phrase → unlimited addresses
- Hot wallets: convenient, less secure
- Cold wallets: secure, less convenient

**Core Concept:** Cryptography replaces trust. Math, not humans, guarantees security and ownership in blockchain systems.

# Next Lesson: Ethereum & Smart Contracts

**What We'll Cover**

- Ethereum architecture
- Smart contract fundamentals
- Solidity programming basics
- Gas and transaction costs
- EVM (Ethereum Virtual Machine)
- Deploying your first contract

**Prerequisites**

- ✓ Lesson 1: Blockchain Fundamentals
- ✓ Lesson 2: Cryptography (today)
- Basic programming knowledge

**Preparation**

Install the following:

- Node.js (v18+)
- Hardhat development environment
- Metamask browser extension
- VS Code with Solidity plugin

**Reading Material**

- Ethereum Whitepaper (selected sections)
- Solidity documentation (basics)
- Understanding gas (Ethereum.org)

**Lesson 3:** Introduction to Ethereum & Smart Contracts

# Questions?

Discussion Topics:

- How would quantum computing affect blockchain cryptography?
- What are the trade-offs between security and usability in wallets?
- How do hardware wallets provide additional security?

Contact: joerg.osterrieder@usi.ch