Cryptography in Blockchain

MAT364 - Cryptography Course

Instructor: Adil Akhmetov
University: SDU
Week 12

Press Space for next page →

Week 12 Focus

Motivation

- Blockchains rely on cryptographic primitives for security
- Digital signatures prove ownership and authorize transactions
- Hash functions ensure data integrity and consensus
- Understanding crypto is essential for blockchain development

Agenda

- Blockchain cryptographic primitives overview
- Bitcoin transaction signatures and address generation
- Ethereum transaction signatures and EIP-1559
- Merkle trees and Merkle proofs
- Hash functions in consensus (Proof of Work)
- Smart contract cryptographic security
- Lab: Build a simple blockchain with crypto

Learning Outcomes

1. Implement Bitcoin/Ethereum transaction signing with ECDSA
2. Build Merkle trees for data verification
3. Generate blockchain addresses from public keys
4. Understand cryptographic security in smart contracts

Blockchain Cryptography Overview

Cryptographic Primitives in Blockchain

Core Components

- **Hash Functions** (SHA-256, Keccak-256)
  - Block hashing, Merkle trees, proof-of-work
- **Digital Signatures** (ECDSA, EdDSA)
  - Transaction authorization, ownership proof
- **Public Key Cryptography**
  - Address generation, key derivation

Security Properties

- **Immutability** - Hash chains prevent tampering
- **Authentication** - Signatures prove ownership
- **Integrity** - Merkle trees verify data
- **Consensus** - Cryptographic puzzles (PoW)

Blockchain Structure

```
Block:
  - Previous Block Hash (SHA-256)
  - Merkle Root (SHA-256 of transactions)
  - Timestamp
  - Nonce (for PoW)
  - Transactions (signed with ECDSA)
```

Transaction Flow

1. User creates transaction
2. Sign with private key (ECDSA)
3. Broadcast to network
4. Miners verify signature
5. Include in block with Merkle tree
6. Block hash links to previous block

Key insight: Blockchains are essentially cryptographic data structures secured by hash functions and digital signatures.

Bitcoin Cryptography

Bitcoin Transaction Signatures

Transaction Structure

- **Inputs:** Previous transaction outputs (UTXOs)
- **Outputs:** Recipient addresses and amounts
- **ScriptSig:** Signature + public key
- **ScriptPubKey:** Spending conditions

Signing Process

1. Create transaction hash (double SHA-256)
2. Sign hash with private key (ECDSA secp256k1)
3. Include signature in ScriptSig
4. Miners verify signature during validation

ECDSA Parameters

- **Curve:** secp256k1
- **Hash:** SHA-256 (double hashed)
- **Signature format:** DER-encoded (r, s)

Python Implementation

```
import hashlib
from cryptography.hazmat.primitives.asymmetric import e
from cryptography.hazmat.primitives import hashes, seri
from ecdsa import SigningKey, SECP256k1

class BitcoinTransaction:
    def __init__(self, private_key):
        self.private_key = private_key
        self.public_key = private_key.public_key()

    def create_transaction_hash(self, inputs, outputs,
        """Create transaction hash for signing"""
        # Simplified - real Bitcoin uses more complex s
        tx_data = f"{inputs}{outputs}{locktime}".encode
        # Double SHA-256 (Bitcoin standard)
```

Bitcoin Address Generation

Address Generation Steps

1. **Generate ECDSA key pair** (secp256k1)
2. **Compress public key** (33 bytes: 0x02/0x03 + x-coordinate)
3. **Hash public key** with SHA-256
4. **Hash again** with RIPEMD-160 → 20 bytes
5. **Add version byte** (0x00 for mainnet)
6. **Double SHA-256** for checksum (first 4 bytes)
7. **Base58 encode** final address

Address Types

- **P2PKH** (Pay-to-Public-Key-Hash): Legacy, starts with '1'
- **P2SH** (Pay-to-Script-Hash): Multisig, starts with '3'
- **Bech32** (SegWit): Native SegWit, starts with 'bc1'

Implementation

```
import hashlib
import base58
from cryptography.hazmat.primitives.asymmetric import e
from cryptography.hazmat.primitives import serializatio

def generate_bitcoin_address(public_key, network='mainn
    """Generate Bitcoin address from public key"""

    # Compress public key
    public_bytes = public_key.public_bytes(
        encoding=serialization.Encoding.X962,
        format=serialization.PublicFormat.CompressedPoi
    )

    # SHA-256
```

Bitcoin Script and Verification

Script Language

- **Stack-based** programming language
- **ScriptSig:** Unlocks previous output
- **ScriptPubKey:** Locks current output
- **Execution:** ScriptSig + ScriptPubKey must evaluate to true

Common Script Types

- **P2PKH:** `OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG`
- **P2SH:** `OP_HASH160 <scriptHash> OP_EQUAL`
- **Multisig:** `OP_2 <pubkey1> <pubkey2> <pubkey3> OP_3 OP_CHECKMULTISIG`

Simplified Script Execution

```
class BitcoinScript:
    def __init__(self):
        self.stack = []

    def op_dup(self):
        """Duplicate top stack item"""
        if len(self.stack) < 1:
            return False
        self.stack.append(self.stack[-1])
        return True

    def op_hash160(self):
        """Hash top stack item with RIPEMD-160(SHA-256(
        if len(self.stack) < 1:
            return False
```

Ethereum Cryptography

Ethereum Transaction Signatures

Transaction Structure

- **nonce:** Transaction sequence number
- **gasPrice/gasFeeCap:** Fee parameters
- **gasLimit:** Maximum gas to use
- **to:** Recipient address (or contract)
- **value:** ETH amount (wei)
- **data:** Contract call data
- **v, r, s:** ECDSA signature components

EIP-1559 (London Fork)

- **Base fee:** Network-determined fee
- **Priority fee:** User-determined tip
- **Max fee:** Maximum user willing to pay
- **Chain ID:** Prevents replay attacks

Python Implementation

```
import rlp
from eth_account import Account
from eth_account.messages import encode_defunct
from web3 import Web3
import hashlib

class EthereumTransaction:
    def __init__(self, private_key):
        self.private_key = private_key
        self.account = Account.from_key(private_key.pri
            encoding=serialization.Encoding.DER,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryp
        ))
```

Ethereum Address Generation

Address Generation

1. **Generate ECDSA key pair** (secp256k1)
2. **Serialize public key** (uncompressed, 64 bytes)
3. **Keccak-256 hash** of public key
4. **Take last 20 bytes** as address
5. **Add '0x' prefix** for hex representation

Key Differences from Bitcoin

- **No Base58 encoding** - uses hex
- **Keccak-256** instead of SHA-256 + RIPEMD-160
- **No version byte** or checksum
- **Shorter addresses** (20 bytes vs 25 bytes)

Implementation

```
from Crypto.Hash import keccak

def generate_ethereum_address(public_key):
    """Generate Ethereum address from public key"""

    # Get uncompressed public key (64 bytes, no 0×04 pr
    public_bytes = public_key.public_bytes(
        encoding=serialization.Encoding.X962,
        format=serialization.PublicFormat.UncompressedP
    )

    # Remove 0×04 prefix (first byte)
    public_key_bytes = public_bytes[1:]

    # Keccak-256 hash
```

Merkle Trees

Merkle Trees in Blockchain

What is a Merkle Tree?

- **Binary tree** of hashes
- **Leaf nodes:** Hash of data (transactions)
- **Internal nodes:** Hash of child nodes
- **Root:** Single hash representing all data

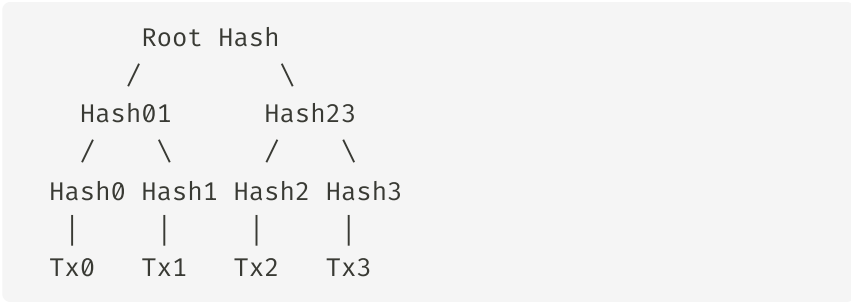
Properties

- **Efficient verification:** O(log n) proof size
- **Tamper detection:** Any change affects root
- **Batch verification:** Verify multiple items at once
- **SPV (Simplified Payment Verification):** Light clients

Use Cases

- **Blockchain:** Transaction verification
- **Git:** Commit verification

Merkle Tree Structure



Merkle Proof

To prove Tx2 is in the tree:

- Provide: Hash3, Hash01, Root
- Verify: Hash(Tx2) → Hash2, Hash(Hash2 || Hash3) → Hash23, Hash(Hash01 || Hash23) → Root

Merkle Tree Implementation

Building the Tree

```python
import hashlib
from typing import List, Optional

class MerkleTree:
    def __init__(self, data: List[bytes]):
        self.data = data
        self.leaves = [self.hash(item) for item in data
        self.root = self.build_tree(self.leaves)

    def hash(self, data: bytes) -> bytes:
        """Double SHA-256 (Bitcoin style)"""
        return hashlib.sha256(
            hashlib.sha256(data).digest()
        ).digest()
```

Merkle Proof Generation

```python
    def generate_proof(self, index: int) -> List[bytes]
        """Generate Merkle proof for leaf at index"""
        if index >= len(self.leaves):
            return []

        proof = []
        current_level = self.leaves
        current_index = index

        while len(current_level) > 1:
            # Find sibling
            if current_index % 2 == 0:
                sibling_index = current_index + 1
            else:
                sibling_index = current_index - 1
```

Merkle Tree Usage Example

Example: Transaction Verification

```
# Create transactions
transactions = [
    b"Alice → Bob: 1 BTC",
    b"Bob → Charlie: 0.5 BTC",
    b"Charlie → Alice: 0.3 BTC",
    b"Dave → Eve: 2 BTC"
]

# Build Merkle tree
tree = MerkleTree(transactions)
print(f"Merkle Root: {tree.root.hex()}")

# Generate proof for transaction at index 1
proof = tree.generate_proof(1)
print(f"Proof length: {len(proof)} nodes")
```

Block Header Structure

```
class BlockHeader:
    def __init__(self, prev_hash, merkle_root, timestar
        self.prev_hash = prev_hash
        self.merkle_root = merkle_root
        self.timestamp = timestamp
        self.nonce = nonce
        self.difficulty = difficulty

    def hash(self):
        """Hash block header"""
        data = (
            self.prev_hash +
            self.merkle_root +
            self.timestamp.to_bytes(8, 'big') +
            self.nonce.to_bytes(4, 'big') +
```

SPV (Simplified Payment Verification)

- **Light clients** don't download full blockchain
- **Only need:** Block headers + Merkle proofs
- **Verify:** Transaction is in block without full data

Benefits

- **Reduced storage:** Only headers, not full blocks
- **Fast verification:** Merkle proofs are small
- **Security:** Can't fake Merkle proof without breaking hash

Hash Functions in Consensus

Proof of Work (PoW)

How PoW Works

1. **Mine block:** Find nonce such that hash(block + nonce) < target
2. **Difficulty:** Target adjusts to maintain block time
3. **Validation:** Anyone can verify hash < target
4. **Security:** Requires computational work to find valid nonce

Hash Function Requirements

- **Preimage resistance:** Can't find input from hash
- **Avalanche effect:** Small change → completely different hash
- **Deterministic:** Same input → same output
- **Fast computation:** Millions of hashes per second

Simplified PoW Implementation

```
import hashlib
import time

class ProofOfWork:
    def __init__(self, difficulty=4):
        self.difficulty = difficulty
        self.target = 2 ** (256 - difficulty * 4)  # Si

    def mine_block(self, block_data: bytes) -> tuple:
        """Mine block by finding valid nonce"""
        nonce = 0
        start_time = time.time()

        while True:
            # Create block with nonce
```

Bitcoin PoW

- **Algorithm:** Double SHA-256
- **Target:** Adjusts every 2016 blocks

Smart Contract Security

Cryptographic Security in Smart Contracts

Common Vulnerabilities

- **Reentrancy:** External calls before state updates
- **Integer overflow:** Arithmetic operations
- **Access control:** Missing authorization checks
- **Randomness:** Predictable random numbers
- **Signature replay:** Reusing signatures across chains

Cryptographic Considerations

- **Signature verification:** Always verify signatures
- **Hash functions:** Use Keccak-256 for Ethereum
- **Randomness:** Use block hashes + external randomness
- **Key management:** Never store private keys in contracts

Secure Signature Verification

```
// Solidity example
pragma solidity ^0.8.0;

contract SecureSignature {
    using ECDSA for bytes32;

    function verifySignature(
        bytes32 messageHash,
        bytes memory signature,
        address signer
    ) public pure returns (bool) {
        // Recover signer from signature
        bytes32 ethSignedMessageHash = messageHash.toEt
        address recovered = ethSignedMessageHash.recove
```

Commit-Reveal Scheme

Problem: On-Chain Randomness

- **Block hashes** are predictable by miners
- **Block timestamps** can be manipulated
- **Need:** Unpredictable randomness for games, lotteries

Solution: Commit-Reveal

1. **Commit phase:** Users submit hash(secret + value)
2. **Reveal phase:** Users reveal secret + value
3. **Verification:** Check hash(secret + value) matches commit
4. **Randomness:** Combine all revealed values

Use Cases

- **Voting:** Hide votes until reveal
- **Lotteries:** Fair random selection
- **Auctions:** Sealed bid auctions

Implementation

```
contract CommitReveal {
    struct Commit {
        bytes32 commitment;
        bool revealed;
        uint256 value;
    }

    mapping(address ⇒ Commit) public commits;
    uint256 public revealDeadline;
    uint256 public randomSeed;

    function commit(bytes32 commitment) public {
        require(block.timestamp < revealDeadline, "Comm
        commits[msg.sender] = Commit(commitment, false
    }
```

Python Commit Generation

```
import hashlib
import secrets
```

Lab: Simple Blockchain

🎯 Student Lab Assignment

Scenario

Build a simplified blockchain that demonstrates core cryptographic concepts.

Requirements

1. **Block Structure:**
   - Previous block hash
   - Merkle root of transactions
   - Timestamp
   - Nonce (for PoW)
   - Block hash
2. **Transaction Structure:**
   - Sender address
   - Recipient address
   - Amount

✅ Solution Outline

Block Class

```
class Block:
    def __init__(self, prev_hash, transactions, difficu
        self.prev_hash = prev_hash
        self.transactions = transactions
        self.timestamp = int(time.time())
        self.merkle_root = self.calculate_merkle_root()
        self.nonce = 0
        self.hash = None
        self.mine(difficulty)

    def calculate_merkle_root(self):
        tree = MerkleTree([tx.hash() for tx in self.tra
        return tree.root

    def mine(self, difficulty):
```

Transaction Class

```
class Transaction:
    def __init__(self, sender, recipient, amount, priva
        self.sender = sender
        self.recipient = recipient
        self.amount = amount
        self.signature = self.sign(private_key)

    def sign(self, private_key):
        tx_data = f"{self.sender}{self.recipient}{self.
        return private_key.sign(tx_data, ec.ECDSA(hashe

    def verify(self, public_key):
        tx_data = f"{self.sender}{self.recipient}{self.
        try:
            public_key.verify(
```

Best Practices & Pitfalls

Security Best Practices

- **Use standard curves:** secp256k1 for Bitcoin/Ethereum
- **Verify all signatures:** Never trust, always verify
- **Protect private keys:** Use hardware wallets, never store in code
- **Use proper randomness:** `secrets` module, not `random`
- **Validate all inputs:** Check addresses, amounts, nonces

Implementation Guidelines

- **Libraries:** Use `cryptography`, `eth-account`, `web3.py`
- **Testing:** Test with testnets before mainnet
- **Key management:** Use proper key derivation (BIP32/BIP44)
- **Error handling:** Always handle signature verification failures
- **Documentation:** Document cryptographic assumptions

Common Mistakes

- ❌ Reusing nonces in ECDSA signatures
- ❌ Storing private keys in smart contracts
- ❌ Using predictable randomness
- ❌ Not verifying Merkle proofs
- ❌ Ignoring signature malleability

Resources

- **Bitcoin:** Bitcoin Developer Guide
- **Ethereum:** Ethereum Yellow Paper
- **Cryptography:** Real-World Cryptography

Summary

- Blockchains rely on **hash functions** (SHA-256, Keccak-256) for integrity and consensus
- **Digital signatures** (ECDSA) prove ownership and authorize transactions
- **Merkle trees** enable efficient verification with O(log n) proofs
- **Address generation** differs between Bitcoin (Base58) and Ethereum (hex)
- **Proof of Work** uses cryptographic puzzles to secure the network
- **Smart contracts** require careful cryptographic design to prevent vulnerabilities

Next Week: Quantum cryptography and post-quantum cryptography.

Assignment: Complete the blockchain lab and submit code with test cases.

Questions?

Thanks for exploring blockchain cryptography! ⛓️🔐