Lecture Notes: Lesson 3

Ethereum and Smart Contracts

Cryptocurrency Course

Contents

1 Introduction to Ethereum

1.1 Historical Background

Ethereum was proposed by Vitalik Buterin in late 2013 and launched in July 2015. It extends Bitcoin's blockchain concept to support arbitrary computation through smart contracts.

Ethereum's Vision

While Bitcoin is **digital gold** (store of value), Ethereum is a **world computer** enabling:

- Decentralized applications (DApps)
- Programmable money and assets
- Trustless agreements (smart contracts)
- Decentralized finance (DeFi)
- Non-fungible tokens (NFTs)

1.2 Key Differences from Bitcoin

Feature	Bitcoin	Ethereum
Purpose	Digital currency	World computer
Scripting	Limited (Bitcoin Script)	Turing-complete (EVM)
Block Time	10 minutes	12 seconds
Consensus (2024)	Proof-of-Work	Proof-of-Stake
Supply Cap	21 million BTC	No fixed cap
Account Model	UTXO	Account-based

2 Ethereum Virtual Machine (EVM)

2.1 Architecture Overview

The EVM is a stack-based virtual machine that executes smart contract bytecode. It provides a sandboxed execution environment where code runs deterministically across all nodes.

2.1.1 Key Characteristics

- **Turing-Complete**: Can express any computation (with gas limits)
- **Deterministic**: Same input always produces same output
- **Isolated**: Contracts run in sandbox, can't access file system/network
- **Persistent**: State stored on blockchain between executions

2.2 EVM Components

2.2.1 Stack

- 1024 element maximum depth
- Each element is 256 bits (32 bytes)
- All operations work on stack items

- LIFO (Last In, First Out) structure

Example Operations:

```
PUSH1 0x05    // Push 5 onto stack
PUSH1 0x03    // Push 3 onto stack
ADD           // Pop two items, push sum (8)
PUSH1 0x02    // Push 2 onto stack
MUL           // Pop two items, push product (16)
```

2.2.2 Memory

- Volatile (cleared between transactions)
- Byte-addressable array
- Expands dynamically (costs gas)
- Used for temporary data during execution

2.2.3 Storage

- Persistent key-value store
- Each contract has own storage
- 256-bit keys and values
- Expensive to use (high gas cost)
- Survives between transactions

Storage Cost

Storage is the most expensive operation:

- **SSTORE** (write): 20,000 gas (new) or 5,000 gas (update)
- **SLOAD** (read): 200 gas
- **Memory**: 3 gas per word

Optimization strategy: Minimize storage writes, use events for logging.

2.3 Account Model

Ethereum uses an account-based model (unlike Bitcoin's UTXO model).

2.3.1 Externally Owned Accounts (EOA)

Controlled by private keys:

- Address derived from public key (last 20 bytes of Keccak-256 hash)
- Balance in wei (1 ETH = 10^{18} wei)
- Nonce (transaction counter)
- No code

2.3.2 Contract Accounts

Controlled by code:

- Address derived from creator address + nonce
- Balance in wei
- Nonce (creation counter)
- Code hash (immutable bytecode)
- Storage root (Merkle tree of state)

2.4 State Transition Function

Ethereum's state machine transitions are deterministic:

$$\sigma_{t+1} = \Upsilon(\sigma_t, T)$$

where:

- σ_t is the world state at block t
- T is a transaction
- Υ is the state transition function
- σ_{t+1} is the resulting new state

3 Gas Mechanism

3.1 Purpose of Gas

Gas serves three critical functions:

1. **Metering**: Limits computation to prevent infinite loops
2. **Pricing**: Aligns costs with resource usage (computation, storage)
3. **Fee Market**: Prioritizes transactions during congestion

Gas Economics

$$\text{Transaction Cost} = \text{Gas Used} \times \text{Gas Price}$$
$$\text{Max Fee} = \text{Gas Limit} \times \text{Gas Price}$$

Refund: If transaction uses less gas than limit, difference is refunded:

$$\text{Refund} = (\text{Gas Limit} - \text{Gas Used}) \times \text{Gas Price}$$

3.2 Gas Costs by Operation

Operation	Gas Cost	Category
Addition (ADD)	3	Arithmetic
Multiplication (MUL)	5	Arithmetic
Division (DIV)	5	Arithmetic
Storage Write (SSTORE)	20,000 / 5,000	Storage
Storage Read (SLOAD)	200	Storage
Keccak-256 Hash	30 + 6/word	Crypto
Call to Contract	700	Call
Create Contract	32,000	Create
Transaction Base	21,000	Base

3.3 EIP-1559: Gas Fee Reform

Introduced in August 2021 (London hard fork):

New Fee Structure:

$$\text{Total Fee} = \text{Gas Used} \times (\text{Base Fee} + \text{Priority Fee})$$

- **Base Fee**: Algorithmically determined, burned (not given to miners)
- **Priority Fee** (Tip): User sets, goes to validators
- **Max Fee**: User's maximum willingness to pay

Base Fee Adjustment:

$$\text{New Base Fee} = \text{Old Base Fee} \times \left(1 + \frac{8(\text{Gas Used} - \text{Target})}{(\text{Target} \times 8)}\right)$$

Increases by up to 12.5% if block is full, decreases if empty.

Gas Calculation Example

User submits transaction with:

- Gas Limit: 100,000
- Max Fee: 200 gwei
- Priority Fee: 2 gwei

If base fee is 150 gwei and transaction uses 80,000 gas:

$$\begin{aligned}
\text{Actual Fee per Gas} &= 150 + 2 = 152 \text{ gwei} \\
\text{Total Cost} &= 80,000 \times 152 = 12,160,000 \text{ gwei} = 0.01216 \text{ ETH} \\
\text{Burned} &= 80,000 \times 150 = 12,000,000 \text{ gwei} \\
\text{To Validator} &= 80,000 \times 2 = 160,000 \text{ gwei}
\end{aligned}$$

4 Solidity Programming Language

4.1 Overview

Solidity is a high-level, statically-typed language for writing smart contracts. It compiles to EVM bytecode.

Characteristics:

- Syntax similar to JavaScript/C++
- Strongly typed
- Supports inheritance, libraries, user-defined types
- Version specified with pragma directive

4.2 Basic Contract Structure

Minimal Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    // State variable stored on blockchain
    uint256 public storedData;

    // Constructor called once at deployment
    constructor(uint256 initialValue) {
        storedData = initialValue;
    }

    // Function to update storage
    function set(uint256 newValue) public {
        storedData = newValue;
    }

    // View function (read-only, no gas when called externally)
    function get() public view returns (uint256) {
        return storedData;
    }
}
```

4.3 Data Types

4.3.1 Value Types

- **Boolean**: `bool` (true/false)
- **Integers**:
  - Unsigned: `uint8` to `uint256` (steps of 8)
  - Signed: `int8` to `int256`
  - Default `uint` = `uint256`
- **Address**: `address` (20 bytes), `address payable`
- **Bytes**: `bytes1` to `bytes32`, `bytes` (dynamic)
- **String**: `string` (dynamic UTF-8)
- **Enums**: Custom enumerated types

4.3.2 Reference Types

- **Arrays**: `uint[]` (dynamic), `uint[10]` (fixed)
- **Structs**: Custom composite types
- **Mappings**: `mapping(address => uint)`

Data Location (must specify for reference types):

- `storage`: Persistent, expensive
- `memory`: Temporary, cheaper
- `calldata`: Read-only, function parameters

Advanced Types Example

```
pragma solidity ^0.8.0;

contract AdvancedTypes {
    // Struct definition
    struct Person {
        string name;
        uint age;
        address wallet;
    }

    // Mapping from address to Person
    mapping(address => Person) public people;

    // Dynamic array
    address[] public allAddresses;

    // Enum
    enum Status { Pending, Active, Closed }
    Status public currentStatus;

    function addPerson(string memory _name, uint _age) public {
        people[msg.sender] = Person(_name, _age, msg.sender);
        allAddresses.push(msg.sender);
        currentStatus = Status.Active;
    }

    function getPerson(address _addr) public view
        returns (string memory, uint, address) {
        Person memory p = people[_addr];
        return (p.name, p.age, p.wallet);
    }
}
```

4.4 Function Modifiers

4.4.1 Visibility

- `public`: Callable internally and externally
- `external`: Only callable externally (more gas efficient)

- `internal`: Only within contract and derived contracts
- `private`: Only within current contract

4.4.2 State Mutability

- `view`: Reads state, doesn't modify (no gas when called externally)
- `pure`: Doesn't read or modify state (no gas when called externally)
- `payable`: Can receive Ether
- (none): Can modify state (costs gas)

4.5 Special Variables and Functions

Variable	Description
`msg.sender`	Address of immediate caller
`msg.value`	Amount of wei sent with transaction
`msg.data`	Complete calldata
`block.timestamp`	Current block timestamp (Unix epoch)
`block.number`	Current block number
`tx.origin`	Original transaction sender (EOA)
`address(this)`	Current contract address
`address.balance`	Balance of address in wei

4.6 Events

Events provide logging facility for DApps to listen to state changes.

Event Usage

```
pragma solidity ^0.8.0;

contract EventExample {
    // Event declaration (indexed parameters can be filtered)
    event Transfer(
        address indexed from,
        address indexed to,
        uint256 value
    );

    mapping(address => uint256) public balances;

    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient␣
            balance");

        balances[msg.sender] -= amount;
        balances[to] += amount;

        // Emit event
        emit Transfer(msg.sender, to, amount);
    }
}
```

Event Storage: Events are stored in transaction logs (not in contract storage), making them much cheaper for recording information.

5 Smart Contract Lifecycle

5.1 Development Workflow

1. **Write**: Develop Solidity code
2. **Compile**: Convert to bytecode using `solc` compiler
3. **Test**: Unit tests on local blockchain (Hardhat, Ganache)
4. **Deploy**: Send creation transaction to network
5. **Verify**: Publish source code on Etherscan
6. **Interact**: Call functions via transactions
7. **Upgrade** (if designed): Use proxy patterns for upgradeability

5.2 Contract Deployment

Deployment creates a special transaction with:

- `to` address: empty (null)
- `data`: Contract bytecode + constructor parameters
- `gas`: Sufficient for deployment

Contract Address Calculation:

$$\text{Address} = \text{Keccak256}(\text{RLP}(\text{sender\_address}, \text{nonce}))[:20]$$

Deployment with Hardhat

```
// scripts/deploy.js
const hre = require("hardhat");

async function main() {
  // Get contract factory
  const SimpleStorage = await hre.ethers.getContractFactory(
    "SimpleStorage"
  );

  // Deploy with constructor argument
  const contract = await SimpleStorage.deploy(42);

  await contract.deployed();

  console.log("Contract deployed to:", contract.address);
  console.log("Initial value:", await contract.get());
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

5.3 Contract Interaction

5.3.1 Reading State (Calls)

- Execute locally on node
- Don't create transactions
- Free (no gas cost)
- Return values immediately
- Used for `view` and `pure` functions

5.3.2 Modifying State (Transactions)

- Broadcast to network
- Require gas payment
- Mined into block (confirmation delay)
- Generate transaction receipt
- Return transaction hash, not function return value

6 Common Design Patterns

6.1 Access Control

Ownable Pattern

```
pragma solidity ^0.8.0;

contract Ownable {
    address public owner;

    event OwnershipTransferred(
        address indexed previousOwner,
        address indexed newOwner
    );

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Invalid address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}

contract MyContract is Ownable {
    uint256 public value;

    // Only owner can call this
    function setValue(uint256 _value) public onlyOwner {
        value = _value;
    }
}
```

6.2 Checks-Effects-Interactions

Best practice to prevent reentrancy attacks:

1. **Checks**: Validate conditions
2. **Effects**: Update state
3. **Interactions**: Call external contracts

Reentrancy Vulnerability

Vulnerable Code:

```
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);
    // BAD: External call before state update
    msg.sender.call{value: amount}("");
    balances[msg.sender] -= amount;
}
```

Secure Code:

```
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);
    // GOOD: Update state first
    balances[msg.sender] -= amount;
    msg.sender.call{value: amount}("");
}
```

Famous example: The DAO hack (2016) exploited reentrancy, leading to $60M loss and Ethereum/Ethereum Classic fork.

7 Study Questions

7.1 Conceptual Questions

1. Explain the difference between `memory` and `storage` in Solidity.
2. Why is storage more expensive than memory or calldata?
3. What happens if a transaction runs out of gas during execution?
4. How does EIP-1559 improve on the legacy gas pricing mechanism?
5. Describe the purpose of events and why they're cheaper than storage.

7.2 Design Problems

1. Design a voting contract where each address can vote once on proposals.
2. Implement a simple escrow contract that holds funds until both parties agree.
3. Create a contract that tracks employee work hours and calculates payment.
4. Build a crowdfunding contract with refund mechanism if goal isn't met.

7.3 Security Analysis

1. Identify the vulnerability in a given contract code snippet.
2. Explain how the checks-effects-interactions pattern prevents reentrancy.
3. Why should you avoid using `tx.origin` for authentication?
4. What are the risks of using `block.timestamp` for critical logic?

8 Further Reading

- Ethereum Yellowpaper (Formal specification)
- Solidity Documentation: `https://docs.soliditylang.org`
- OpenZeppelin Contracts: Secure, audited implementations
- Consensys Smart Contract Best Practices
- Trail of Bits: Building Secure Contracts
- Ethereum Improvement Proposals (EIPs)