Ethereum – Design Patterns

Gallersdörfer, U., Holl, P., & Matthes, F. (2019). "Blockchain-based Systems Engineering“. Lecture Slides. TU Munich.

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

1. Solidity idioms
   - Access restriction
   - Secure Ether transfer
   - Safe arithmetic

2. Solidity design patterns
   - Oracles
     - Synchronous Oracle
     - Asynchronous Oracle
   - Randomness
     - By Oracle
     - By block hash
     - By commit-reveal scheme
   - Evolution support
     - Data segregation
     - Proxy

2. Token standards
   - ERC20
   - ERC721

CC BY-SA 4.0

Solidity idioms

- **Programming idioms** are language-specific patterns for recurring programming problems.
- Idioms are on a lower abstraction level than **design patterns** which are template solutions for recurring software engineering problems.

Example in Java:

```
// Java idiom for generating random number.
Random rand = new Random();
int diceRoll() {
  return rand.nextInt(6) + 1;
}
```

- The current Solidity code base uses many programming idioms (e.g., the *SafeMath* library is used in more than 60% of all verified Smart Contracts from Etherscan (25% of all Smart Contracts)).

A Solidity idiom is a practice-proven code **pattern** for a **recurring coding problem.**

https://github.com/ethereum/wiki/wiki/Useful-%C3%90app-Patterns
Alexander Hefele, "A Conceptual Model for Ethereum Blockchain Analytics", Master's Thesis, Technical University Munich, 2019.

Access restriction idiom

Problem description

- Each contract deployed on the main network is publicly accessible.
- Since all external and public functions can be called by anyone, third parties might execute a function on a contract they should not be allowed to.
- Misconfigured access restrictions led to the largest Ethereum thefts so far[1].

Participants

- An entity that calls a publicly accessible function in a smart contract.
- A smart contract which is called by a transaction or a message.

Applicability

- To protect contract functions from unauthorized calls.
- Examples for such functions: `selfdestruct()`, `mint()`

[1] *https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, accessed on 02.06.2019*

CC BY-SA 4.0

Access restriction idiom (cont.)

Solution

Restrict access of other accounts to execute functions and to modify the state of a contract. In Solidity, access restriction can be achieved by implementing proper **function modifiers** that check if the caller is allowed to execute the actual function logic. To make the contract code more maintainable, the authorization logic is usually put in a separate contract.

Ownable Contract

```
contract Ownable {
        address public owner;

        constructor() public {
                owner = msg.sender;
        }

        /**
        * @dev Throws if called by any account other than the owner.
        */
        modifier onlyOwner() {
                require(msg.sender == owner);
                _;
        }
/…
```

Contract which implements Ownable

```
/…
        /**
        * @dev Allows the current owner to transfer control
        * @param newOwner The address to transfer ownership to.
        */
        function transferOwnership(address newOwner) public onlyOwner {
                        require(newOwner != address(0));
                        owner = newOwner;
        }

        /**
        * @dev Allows the current owner to relinquish control */
        function renounceOwnership() public onlyOwner {
                        owner = address(0);
        }
}
```

CC BY-SA 4.0

Secure Ether transfer idiom

Problem Description

Sending Ether to another account in Ethereum requires the sending account to issue a transaction or message to the receiver. However, if the receiver is a contract account, it is possible to let the transaction intentionally fail. For instance, when the fallback function of the receiving contract throws an exception.

This behavior is usually not intended when sending Ether as it can result in disabling the sending contract.

Participants

An entity that wants to receive Ether by actively issuing a withdraw transaction.
A smart contract that keeps track of all account balances.

Applicability

Scenarios where Ether needs to be transferred by a Smart Contract. The pattern mitigates the risk associated with Ether transfers.

CC BY-SA 4.0

Secure Ether transfer idiom (cont.)

Problem example:

- The following example illustrates a vulnerable contract for an auction.
- On a new highest bid, the previous leader gets her money back.
- A malicious attacker could exploit the property of the transfer function to freeze the contract and remain the highest bidder.

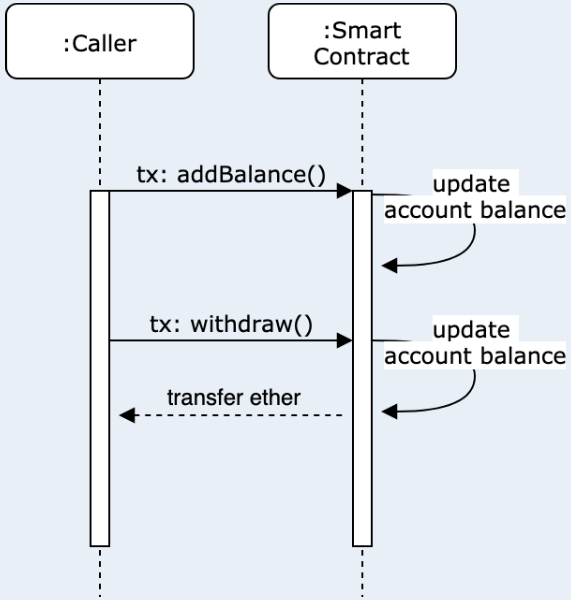
```
contract BadAuction {
    address highestBidder;
    uint highestBid;
    function bid() public payable {
        require(msg.value >= highestBid);
        if (highestBidder != 0) {
                highestBidder.transfer(highestBid);
        }
                highestBidder = msg.sender;
                highestBid = msg.value;
        }
}
```

Secure Ether transfer idiom (cont.)

Solution: Pull over Push

To prevent malicious contracts from halting other contracts through Ether transfers, functions which send Ether should be isolated. A common pattern is to have a separate, isolated function which needs to be actively called by the sender (**pull**).

- Keep track of individual account balances
- Implement an isolated withdraw() function



```
contract PullOverPush {
        mapping(address => uint) accountBalances;
        // ...
        function addBalance() public payable {
                accountBalances[msg.sender] += msg.value;
        }

        function withdraw() public {
                uint amount = accountBalances[msg.sender];
                require(amount != 0);
                require(address(this).balance >= amount);
                accountBalances[msg.sender] = 0;
                msg.sender.transfer(amount);
        }
}
```

CC BY-SA 4.0

Safe arithmetic

Problem Description

An overflow describes the behavior of a fixed size integer value that gets incremented over its maximum value via an arithmetic operation. Vice versa, an underflow decrements a fixed size integer under its minimum value.

Solidity leaves it to the developer to handle integer over- and underflows. This behavior requires special attention when writing Smart Contracts since it can introduce severe vulnerabilities.

Participants

An entity that triggers an arithmetic operation on a function execution.
A smart contract that uses arithmetic operations.

Applicability

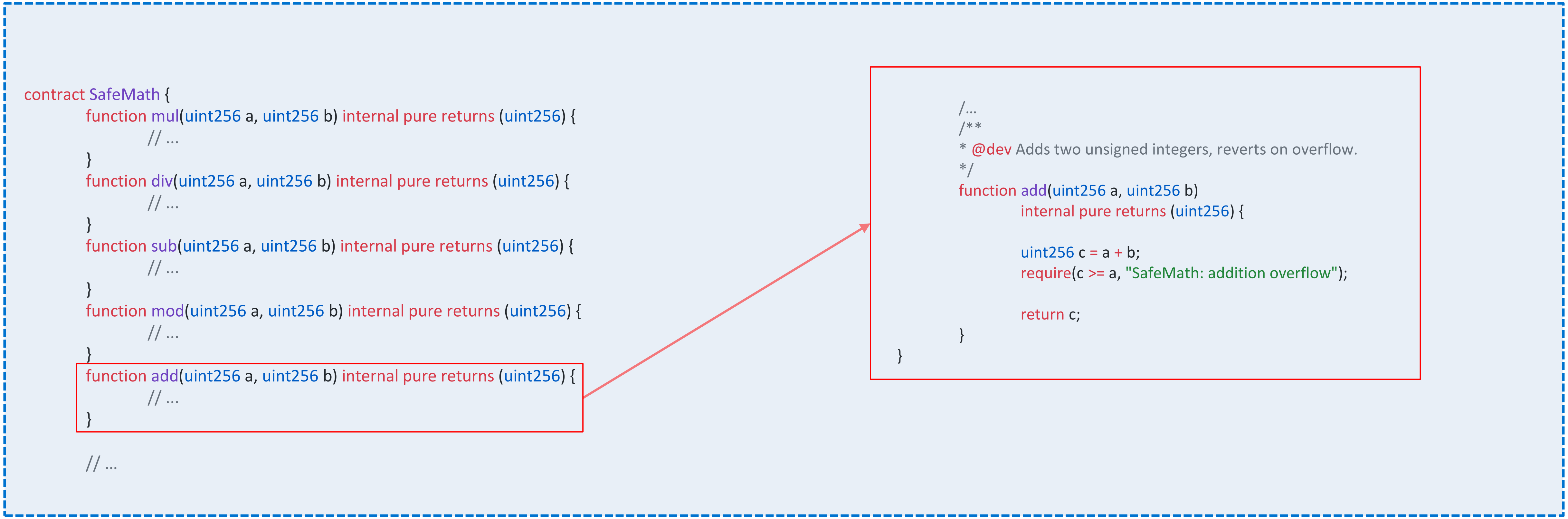
Every function that implements arithmetic operations.

CC BY-SA 4.0

Safe arithmetic (cont.)

Solution

- Implement a wrapper function for every basic (+,-,*,%,/) arithmetic operation that checks if the result produces an over- or underflow. Such tested wrapper functions are available through the most commonly used Solidity library *SafeMath.*
- Your contract inherits *SafeMath*.
- In case of an over- or underflow, the current transaction fails with an error message.



```
contract SafeMath {
        function mul(uint256 a, uint256 b) internal pure returns (uint256) {
                // ...
        }
        function div(uint256 a, uint256 b) internal pure returns (uint256) {
                // ...
        }
        function sub(uint256 a, uint256 b) internal pure returns (uint256) {
                // ...
        }
        function mod(uint256 a, uint256 b) internal pure returns (uint256) {
                // ...
        }
        function add(uint256 a, uint256 b) internal pure returns (uint256) {
                // ...
        }

        // ...
```

```
                /...
                /**
                * @dev Adds two unsigned integers, reverts on overflow.
                */
                function add(uint256 a, uint256 b)
                        internal pure returns (uint256) {

                        uint256 c = a + b;
                        require(c >= a, "SafeMath: addition overflow");

                        return c;
                }
        }
```

CC BY-SA 4.0

Outline

1. Solidity idioms
   - Access restriction
   - Secure Ether transfer
   - Safe arithmetic

2. Solidity design patterns
   - Oracles
     - Synchronous Oracle
     - Asynchronous Oracle
   - Randomness
     - By Oracle
     - By block hash
     - By commit-reveal scheme
   - Evolution support
     - Data segregation
     - Proxy

2. Token standards
   - ERC20
   - ERC721

CC BY-SA 4.0

Smart Contract design patterns

Designing decentralized applications on the basis of a blockchain infrastructure is a rather new area in software engineering. Similar to traditional software engineering, there are recurring problems that are shared across a large set of smart contracts.

Design patterns are **template solutions** for **recurring design problems**.

Design patterns are specifically important for smart contract development:

- Determinism by design makes use of external data and random numbers challenging.
- The code of a deployed contract is immutable ➔ Contracts cannot be updated.
- In most cases, financial value is at risk.
- Transaction finality ➔ Stolen money is gone forever.
- Availability of source and bytecode makes it easier for attackers to find potential vulnerabilities.

https://github.com/ethereum/wiki/wiki/Useful-%C3%90app-Patterns

CC BY-SA 4.0

Oracle Pattern

Problem Description

Smart contracts can’t access any **data** from **outside** the **blockchain** on their own. There are no HTTP or similar network methods implemented to call or access external services. This is on purpose to **prevent non-deterministic behavior** once a function is called (there are also no functions to generate random values).

Participants

- A Smart Contract which **requires data** from external sources
- An external party that is willing to **provide data** from external sources via a **separate Smart Contract**

Applicability

Any scenario in which a Smart Contract **relies on external data for computation** of future states.

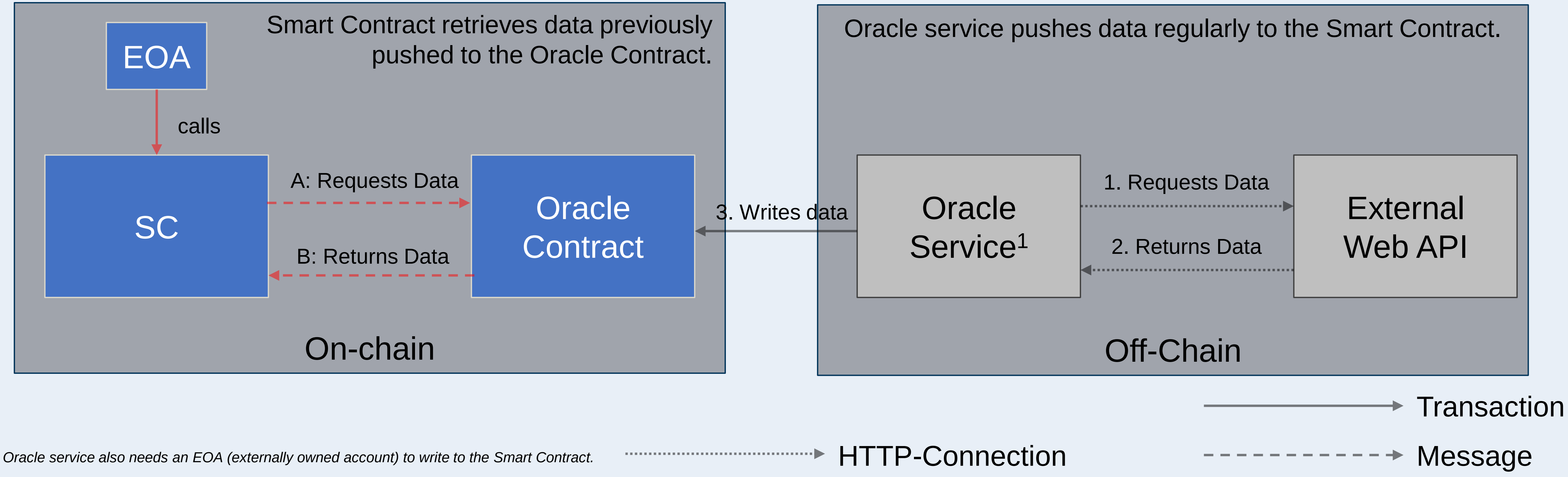
CC BY-SA 4.0

Oracle Pattern: Synchronous Oracle

Solution

Currently, the **only way** to write smart contracts **using external data** (e.g. weather data, traffic data etc.) is to **use oracles**. Oracles are basically third-party services that collect data from web services and write the data via a special Smart Contract to the blockchain. Other smart contracts can now call the Oracle contract to get the data. We differentiate between **synchronous** and **asynchronous** Oracles.

Synchronous Oracle



[1] *The Oracle service also needs an EOA (externally owned account) to write to the Smart Contract.*

Simple example: Synchronous Oracle contract

A simple Oracle contract for the current market prices of Ethereum and Bitcoin.

```
contract SimpleOracle {
    address public controller;
    enum Coins {Ethereum, Bitcoin}
    Coins public coins;
    mapping (uint => uint) public prices;

    constructor() public {
        controller = msg.sender;
    }

    modifier isController {
        require(msg.sender == controller);
        _;
    }

    function update(uint coin, uint price) public isController {
        if(coin != uint(coins.Ethereum) && coin != uint(coins.Bitcoin)) {
            revert();
        }
        prices[coin] = price;
    }

    function getPrice(uint coin) public view returns(uint price) {
        return prices[coin];
    }
}
```

The controller of the Oracle contract

The storage that represents the market data

Controller is set by contract deployment

An update function that lets the controller of the contract set the current market prices

A publicly accessible view function that returns the current price of a particular coin.

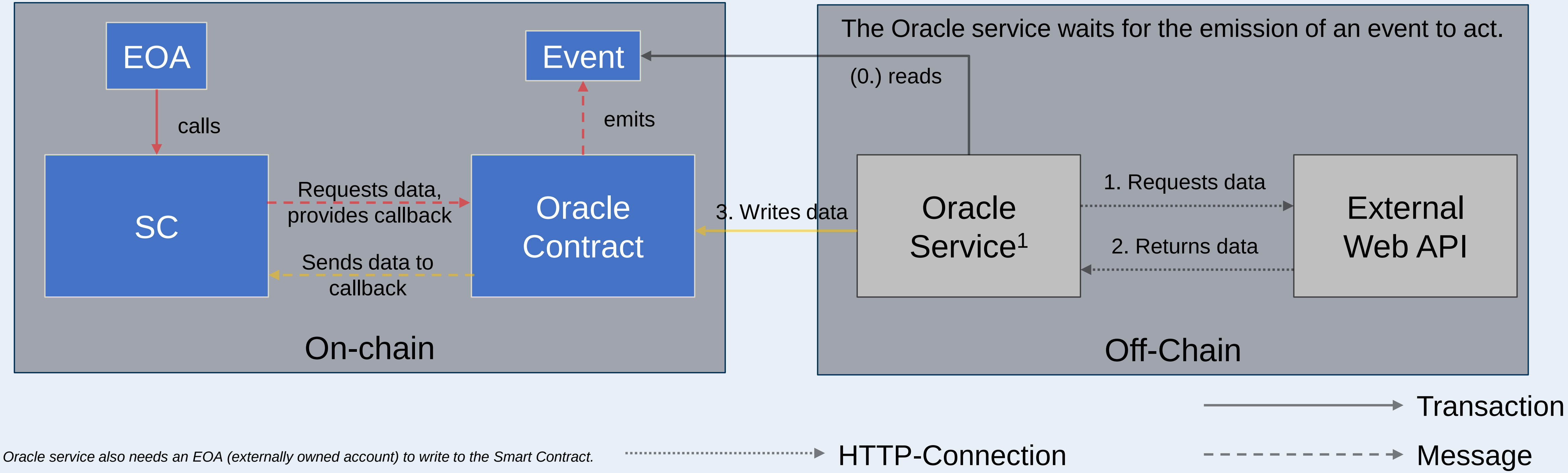
CC BY-SA 4.0

Oracle Pattern: Asynchronous Oracle / Inversion of Control

Solution

In contrast to the synchronous Oracle (in which data is periodically pushed to the Oracle contract and retrieved by the Smart Contract), the asynchronous Oracle acts as a proxy for the Oracle service to send fresh data to the original Smart Contract.

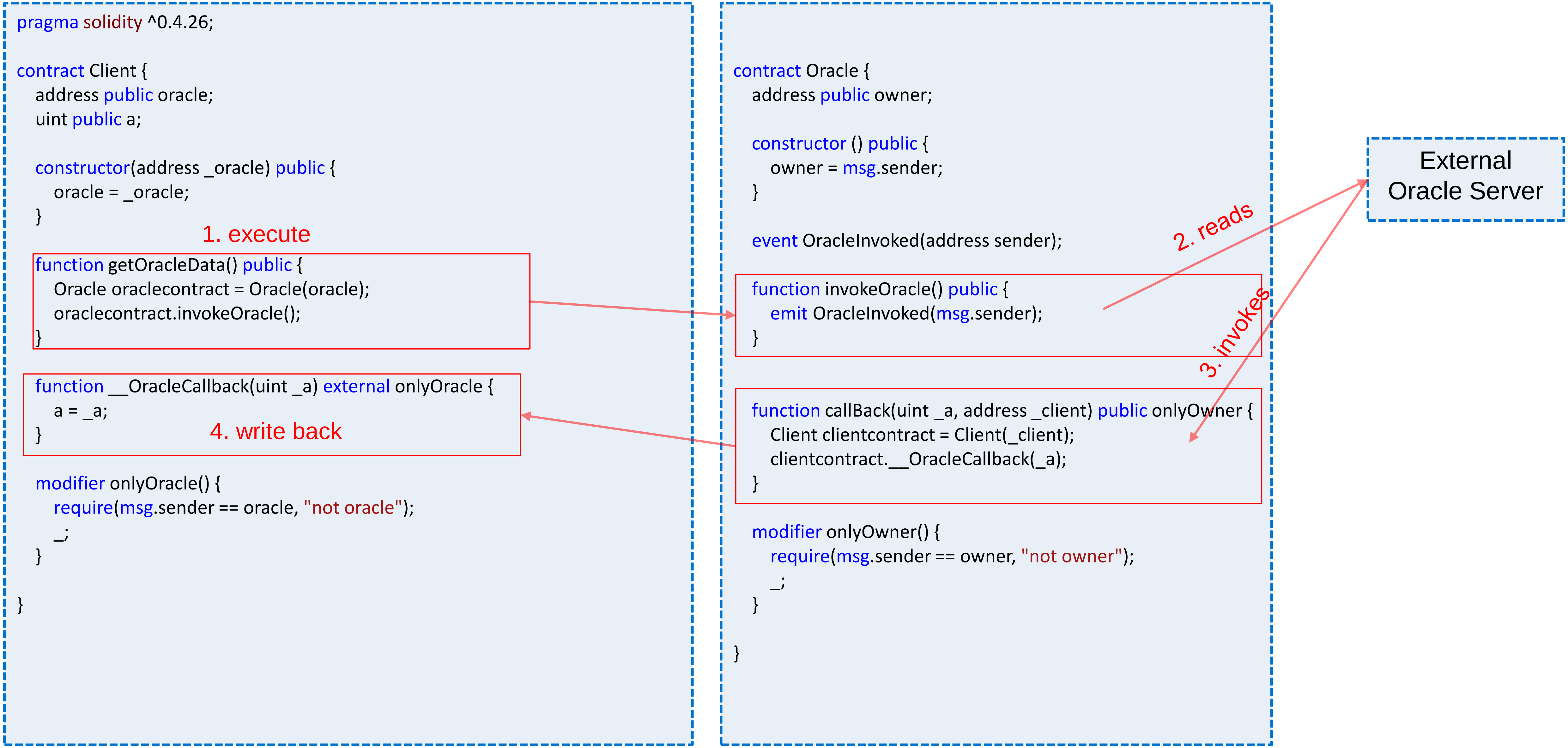
Asynchronous Oracle



[1]The Oracle service also needs an EOA (externally owned account) to write to the Smart Contract.

CC BY-SA 4.0

Simple example: Asynchronous Oracle contract



```
pragma solidity ^0.4.26;

contract Client {
   address public oracle;
   uint public a;

   constructor(address _oracle) public {
      oracle = _oracle;
   }

   function getOracleData() public {
      Oracle oraclecontract = Oracle(oracle);
      oraclecontract.invokeOracle();
   }

   function __OracleCallback(uint _a) external onlyOracle {
      a = _a;
   }

   modifier onlyOracle() {
      require(msg.sender == oracle, "not oracle");
      _;
   }

}
```

```
contract Oracle {
   address public owner;

   constructor () public {
      owner = msg.sender;
   }

   event OracleInvoked(address sender);

   function invokeOracle() public {
      emit OracleInvoked(msg.sender);
   }

   function callBack(uint _a, address _client) public onlyOwner {
      Client clientcontract = Client(_client);
      clientcontract.__OracleCallback(_a);
   }

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

}
```

CC BY-SA 4.0

Asynchronous Oracle contract: Limit gas usage

- In case of the asynchronous oracle, the oracle server sends a **transaction** to the oracle contract which **invokes** a **message** to the client contract.
- In our case, **all gas** from the original transaction **is forwarded** to the message.
- The Client Smart Contract could consume this gas for malicious purposes.
- Therefore, it is advisable to **limit** the **forwarded gas**. The limitation is introduced with an additional parameter. The parameter _gas defines the to-be forwarded amount of gas.

  clientcontract.__OracleCallback(**gas _gas**)(_a);

- **Be careful**: Low gas can make transactions fail!

```
contract Oracle {
    address public owner;

    constructor () public {
        owner = msg.sender;
    }

    event OracleInvoked(address sender);

    function invokeOracle() public {
        emit OracleInvoked(msg.sender);
    }


    function callBack(uint _a, address _client, uint _gas) public onlyOwner {
        Client clientcontract = Client(_client);
        clientcontract.__OracleCallback(gas _gas)(_a);
    }

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

}
```

CC BY-SA 4.0

Oracle Pattern (cont.)

Advantages

- Enables data retrieval from external sources
- Either easy to use (synchronous Oracle) or live data (asynchronous Oracle)
- Can be used for varying purposes (randomness, stock data, weather data, …)

Disadvantages

- Costly in terms of gas consumption
- Dependence on a third party
  - In terms of data manipulation (Oracle owner can manipulate data)
  - In terms of availability (Oracle service could be offline → Smart Contract not functioning)

CC BY-SA 4.0

Randomness Pattern

Problem Description

Solidity **does not provide any functions** that **generate random numbers**. This is due to the forced **deterministic** behavior of the EVM. Having random numbers would make it impossible for other nodes to validate the correct output of a function.

Participants

- A Smart Contract which **requires random numbers**
- (Potential participants which provide the random number)

Applicability

Any scenario in which a Smart Contract **relies on random numbers for computation** of future states.

CC BY-SA 4.0

Randomness Pattern: Solution with Oracles

Solution

Oracles are usually used to access data from outside of the Blockchain. Additionally, Oracles can provide random numbers, either from local sources or from services (e.g., WolframAlpha).

Limitations

- Costly in terms of gas consumption
- Dependence on a third party in terms of data manipulation (Oracle owner can manipulate data)
- Potential predictability

CC BY-SA 4.0

Randomness Pattern: Solution with block hash

Solution

A commonly used approach to create pseudo-random numbers is to use the hash of the currently mined block. The hash of the block is not known upfront and therefore more or less random. In theory, a miner could affect and manipulate the block hash by deciding which transactions are included in the block and which not.

Example Code

```
contract SimpleRandom {
  function randomNumber() public view returns (uint) {
    return uint(blockhash(block.number - 1));
    // -1 for the latest block where the transaction with the function call is included
  }
}
```

Limitations

- Miner could try to delay transactions that do not lead to the desired result
- Only relevant for small amounts of money (< mining reward)
- Potential predictability by miner

CC BY-SA 4.0

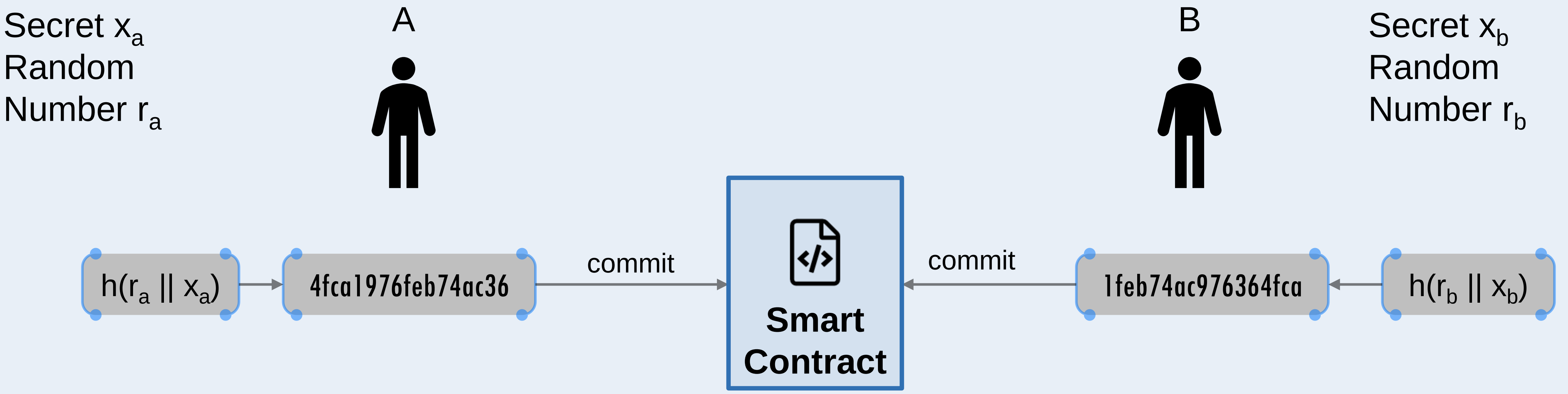
Randomness Pattern: Solution with commit and reveal scheme

Solution

Another potential approach is the commit and reveal scheme, which utilizes the hiding property of cryptographic hash functions. It consists of two phases:

1. Users (>1) hash their random number and publish it to the Smart Contract.
2. Users reveal their random number and the Smart Contract calculates a random number from these commits.

Phase 1: Commit

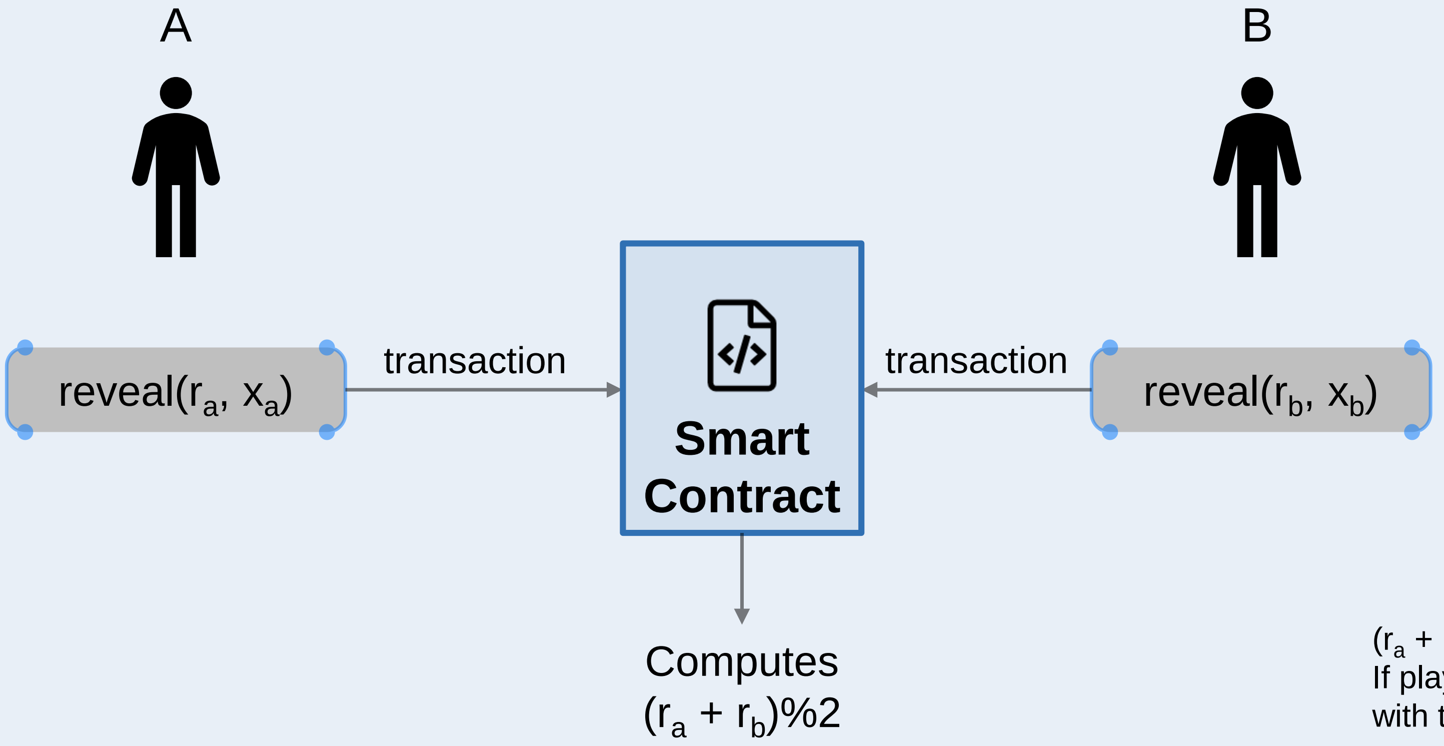


CC BY-SA 4.0

Randomness Pattern: Solution with commit and reveal scheme (cont.)

Phase 2: Reveal

Secret x_a
Random
Number r_a



A

B

reveal(r_a, x_a)

transaction

Smart Contract

transaction

reveal(r_b, x_b)

Computes
$(r_a + r_b)\%2$

Secret x_b
Random
Number r_b

$(r_a + r_b)\%n$ could return n different results. If player=n, then you can select a winner with the provided random numbers.

Limitations

- Requires user interaction
- Users could withhold their reveals
  - Possible solution: If information is not revealed, the stakes are distributed among other players

CC BY-SA 4.0

Evolution support patterns

The code of deployed contracts is immutable by design which makes evolving them difficult.

However, sometimes it is necessary to replace a contract with a newer version, e.g., to fix a bug or add new functionality.

Replacing a contract with another version is challenging:

- Each contract deployed on the blockchain gets a new address
- Entities referencing a previous version of a contract need to be notified about the update
- Each contract has its own state
- State migration is not straightforward since the more recent contract must inherit not only the data but all permissions, as well.
- Potential state inconsistencies when the previous contract version is still used by some entities

CC BY-SA 4.0

Data segregation

Problem Description

- Each smart contract maintains its own state.
- Most smart contracts do **not separate state data** from **business logic**.
- **Close coupling** becomes a **problem** whenever the contract needs to be **replaced by a newer version**.
- A contract update requires a migration process for its state.

Participants

- An entity that wants to replace a smart contract with a more recent version.
- A smart contract that implements some business logic.

Applicability

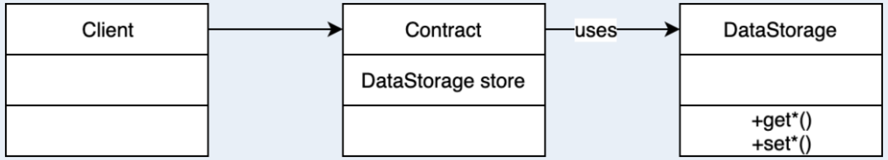
- Scenarios where a vulnerability is found in a smart contract.
- Whenever business logic needs to be updated.

CC BY-SA 4.0

Data segregation (cont.)

Solution

- **Decoupling** the **business logic** from the **state data** by using two separate contracts.
- One contract implements all the functions for the business logic.
- Another one acts as a **raw data storage contract**.
- The storage contract must implement an interface that provides all the necessary getter and setter functions.
- Whenever the business logic needs to be updated, a new contract could access the data through the storage contract.



```
contract DataStorage {
        // Whatever types the contract needs to store...
        mapping(bytes32 => uint) uintStore;

        function getIntValue(bytes32 key) public view returns (uint) {
                return uintStore[key];
        }

        function setIntValue(bytes32 key, uint value) public {
                uintStore[key] = value;
        }
}
```

CC BY-SA 4.0

Data segregation (cont.)

Advantages

- Separation of concerns – Low coupling
- Updating the business logic of a contract is straight forward
- Migration to a new business logic does not require separate migration of the state

Disadvantages

- The user has to trust the maintainer of the contract
- One has to maintain multiple contracts at once
- Introduces overhead to ensure access rights and permissions
- In case of an update, applications need to be notified about the contract address of the new version
- Requires proper lifecycle management

CC BY-SA 4.0

Proxy

Problem Description

- **Replacing a smart contract** with a newer version comes with the implication that the newly deployed contract has a **different account address** as the older one.
- This behavior leads to the problem that **all entities** which are using the older version of the contract **need to be notified** about the **new contract address**.
- All contracts that reference an old version need to update their state.

Participants

- An entity that wants to replace a smart contract with a more recent version.
- A smart contract that implements some business logic.

Applicability

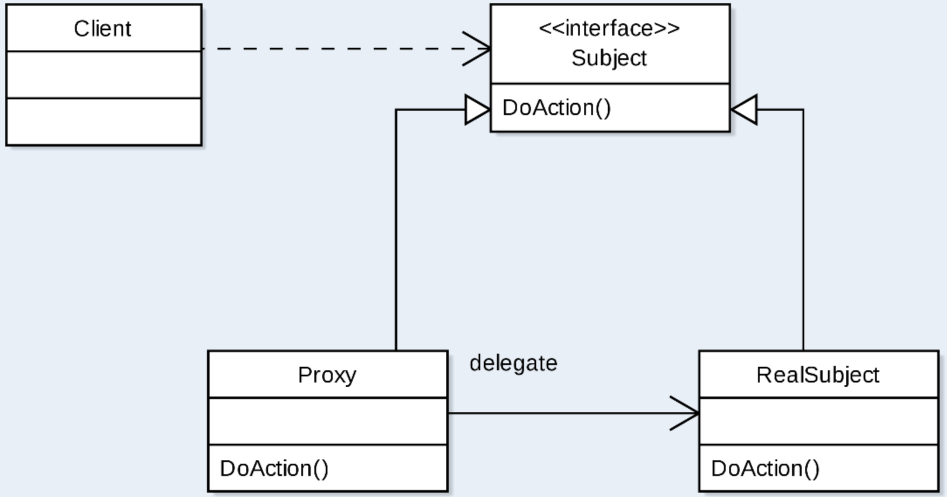
- Scenarios where a vulnerability is found in a smart contract,
- whenever the business logic needs to be updated in a way that the address of the contract does not change.

CC BY-SA 4.0

Proxy (cont.)

Solution

- Separate the business logic and the data from the actual, application-facing contract.
- Implement a proxy contract which forwards all calls to the contract with the business logic.



```
import "../../authorization/Ownership.sol";
import "./Proxy.sol";
contract Proxy is Owned {
        address logicAddress;

        function doSomething() public onlyOwner {
                Logic l = Logic(logicAddress);
                l.doSomething();
        }

        function updateLogicAddress(address _address) public onlyOwner {
                logicAddress = _address;
        }
}
```

CC BY-SA 4.0

Proxy (cont.)

Advantages

- Separation of concerns – Low coupling
- Updating the business logic of a contract is straightforward
- Deployments of new contract versions do not require an address change
- Bugs in contracts can be fixed

Disadvantages

- One has to trust that the maintainer of the proxy to not replace the business logic with malicious code
- The interface is still immutable, it is not possible to add new callable functions (at least without using a hack through the fallback function and delegatecall()-method)
- Proxies make it harder to read and verify the logic of a smart contract
- Introduces overhead to ensure access rights and permissions
- One has to maintain multiple contracts at once

CC BY-SA 4.0

Outline

1. Solidity idioms
   - Access restriction
   - Secure Ether transfer
   - Safe arithmetic

2. Solidity design patterns
   - Oracles
     - Synchronous Oracle
     - Asynchronous Oracle
   - Randomness
     - By Oracle
     - By block hash
     - By commit-reveal scheme
   - Evolution support
     - Data segregation
     - Proxy

2. Token standards
   - ERC20
   - ERC721

CC BY-SA 4.0

Ethereum tokens

Tokens are smart contracts that implement a standardized interface and are currently the main use case in the Ethereum ecosystem.

Depending on the actual use case, different token standards exist. The most common use cases and standards are:

- ICOs and crowd funding (mostly ERC 20)
- Gaming (mostly ERC 20 and ERC 721)

Token classes:

1. Fungible tokens: All tokens are indistinguishable and have exactly the same value.

2. Non-fungible tokens: Each token is uniquely identifiable by an ID.

CC BY-SA 4.0

ERC20 – Token standard

The **ERC20** standard was first proposed by Fabian Vogelsteller and Vitalik Buterin in November 2015. The specification **defines an interface** that a contract must implement to be ERC20 compliant. It does **not specify** the **actual implementation** of the functions.

It is the most commonly used token standard in the Ethereum network with 190,000 contracts deployed on the main network. Each **ERC20 contract** defines a class of **fungible tokens**.

Interface

```
contract ERC20Interface {
        uint256 public totalSupply;
        function totalSupply() public view returns (uint);
        function balanceOf(address tokenOwner) public view returns (uint balance);
        function allowance(address tokenOwner, address spender) public view returns (uint remaining);
        function transfer(address to, uint tokens) public returns (bool success);
        function approve(address spender, uint tokens) public returns (bool success);
        function transferFrom(address from, address to, uint tokens) public returns (bool success);

        event Transfer(address indexed from, address indexed to, uint tokens);
        event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```

CC BY-SA 4.0

ERC20 – Reference implementation

```
contract ERC20 is IERC20 {
    using SafeMath for uint256;
    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;
    uint256 private _totalSupply;

    function totalSupply() public view returns (uint256) {
    return _totalSupply;
    }

    function balanceOf(address account) public view returns (uint256) {
    return _balances[account];
    }

    /**
    * @dev See `IERC20.transfer`.
    *
    * Requirements:
    *
    * - `recipient` cannot be the zero address.
    * - the caller must have a balance of at least `amount`.
    */
    function transfer(address recipient, uint256 amount) public returns (bool) {
        _transfer(msg.sender, recipient, amount);
        return true;
    }
```

CC BY-SA 4.0

ERC20 – Reference implementation (cont.)

```
/**
* @dev See `IERC20.allowance`.
*/
function allowance(address owner, address spender) public view returns (uint256) {
                return _allowances[owner][spender];
}

/**
* @dev See `IERC20.approve`.
*
* Requirements:
*
* - `spender` cannot be the zero address.
*/
function approve(address spender, uint256 value) public returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
}
```

CC BY-SA 4.0

ERC20 – Reference implementation (cont.)

```
/**
* @dev See `IERC20.transferFrom`.
*
* Emits an `Approval` event indicating the updated allowance. This is not
* required by the EIP. See the note at the beginning of `ERC20`;
*
* Requirements:
* - `sender` and `recipient` cannot be the zero address.
* - `sender` must have a balance of at least `value`.
* - the caller must have allowance for `sender`'s tokens of at least
* `amount`.
*/
function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, msg.sender, _allowances[sender][msg.sender].sub(amount));
        return true;
}
```

CC BY-SA 4.0

ERC20 – Reference implementation (cont.)

```
/**
* @dev Moves tokens `amount` from `sender` to `recipient`.
*
* This is internal function is equivalent to `transfer`, and can be used to
* e.g. implement automatic token fees, slashing mechanisms, etc.
*
* Emits a `Transfer` event.
*
* Requirements:
*
* - `sender` cannot be the zero address.
* - `recipient` cannot be the zero address.
* - `sender` must have a balance of at least `amount`.
*/
function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
}
```

CC BY-SA 4.0

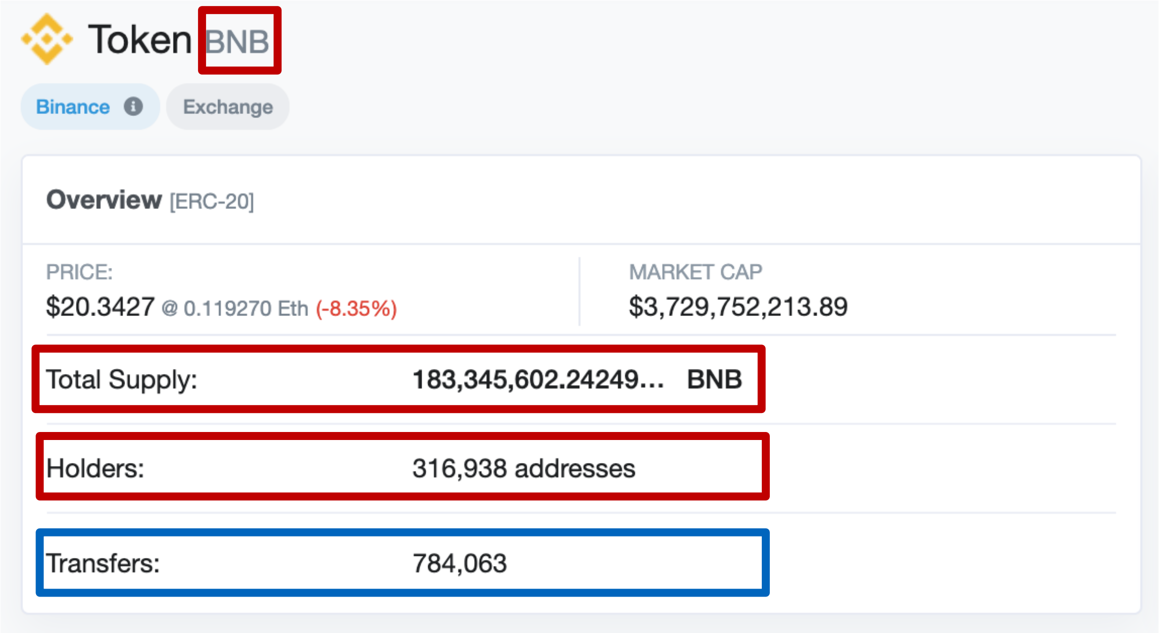
ERC20 – Reference implementation (cont.)

```
/**
* @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
*
* This is internal function is equivalent to `approve`, and can be used to
* e.g. set automatic allowances for certain subsystems, etc.
*
* Emits an `Approval` event.
*
* Requirements:
*
* - `owner` cannot be the zero address.
* - `spender` cannot be the zero address.
*/
function _approve(address owner, address spender, uint256 value) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = value;
        emit Approval(owner, spender, value);
}

}
```

CC BY-SA 4.0

Ethereum tokens (cont.)

Interaction



Token BNB

Binance · Exchange

Overview [ERC-20]

PRICE:
$20.3427 @ 0.119270 Eth (-8.35%)

MARKET CAP
$3,729,752,213.89

Total Supply: 183,345,602.24249... BNB

Holders: 316,938 addresses

Transfers: 784,063

Services and applications **interact** with token contracts **through** the **interface** defined by the standard.

The screenshot on the left shows an ERC20 token how it is represented on etherscan.io.

🟥 Directly readable from the contracts’ state

🟦 Implicit by the transaction log

Statistics

- All tokens have a combined market cap of 12.4 billion $ (Ethereum without tokens has 18 billion $).
- More than 10 million token transactions so far.
- More than 190,000 token contracts have been deployed on the main network.

CC BY-SA 4.0

ERC721 – Token standard

ERC20 tokens are not suitable to represent ownership of individual and unique assets like a house or a unique artwork. Therefore, the ERC721 token standard was created in January 2018 as a standard interface for non-fungible tokens. Each ERC721 token is distinguishable by a unique ID.

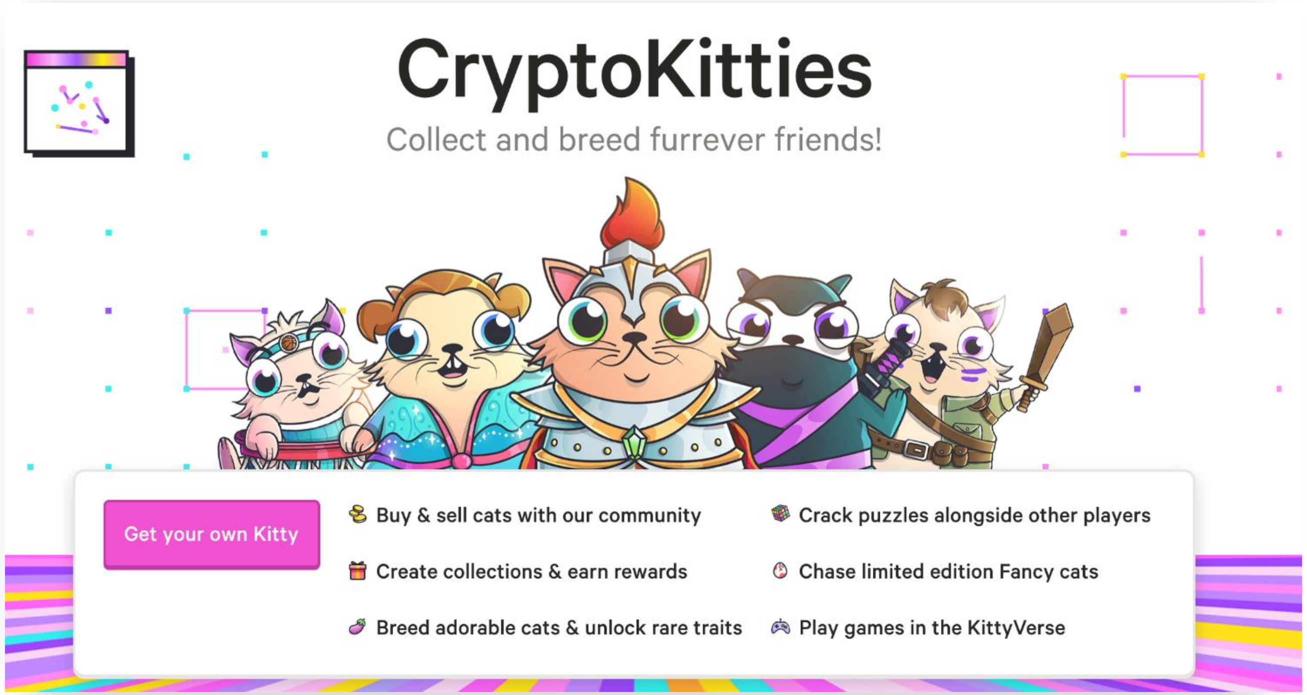
Use cases:

- Digital goods
  - In-Game items
  - Collectables
  - Music
  - etc.

- Physical property
  - Real estate
  - etc.

- Negative value assets
  - Loans
  - etc.

CryptoKitties, the most used ERC 721 contract



CryptoKitties

Collect and breed furrever friends!

Buy & sell cats with our community

Create collections & earn rewards

Breed adorable cats & unlock rare traits

Crack puzzles alongside other players

Chase limited edition Fancy cats

Play games in the KittyVerse

CC BY-SA 4.0

ER721 – Token standard (cont.)

Interface

```
pragma solidity ^0.4.20;

/// @title ERC721 Non-Fungible Token Standard
/// @dev See https://eips.ethereum.org/EIPS/eip-721

interface ERC721 {
        function balanceOf(address _owner) external view returns (uint256);
        function ownerOf(uint256 _tokenId) external view returns (address);
        function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data) external payable;
        function safeTransferFrom(address _from, address _to, uint256 _tokenId) external payable;
        function transferFrom(address _from, address _to, uint256 _tokenId) external payable;
        function approve(address _approved, uint256 _tokenId) external payable;
        function setApprovalForAll(address _operator, bool _approved) external;
        function getApproved(uint256 _tokenId) external view returns (address);
        function isApprovedForAll(address _owner, address _operator) external view returns (bool);

        event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);
        event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId);
        event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);
}
```

CC BY-SA 4.0