Ethereum Design Patterns

Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2022). “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
   - Tight Variable Packing

2. Solidity Design Patterns
   - Oracles
     - Synchronous Oracle
     - Asynchronous Oracle
   - Randomness
     - By Oracle
     - By Block Hash
     - By Commit-Reveal Scheme
   - Evolution Support
     - Data Segregation
     - Proxy

3. Token Standards
   - ERC20
   - ERC721

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.
- OpenZeppelin is a library that automates operations and delivers reusable, secure, tested and community-audited code. Most of the critical building blocks that are needed for a contract are already pre-programmed in it, so users should utilize the existing library instead of writing their own code. In this section, we will go through the most prevalent idioms of Solidity smart contract programming through the OpenZeppelin library.
- *SafeMath*[1] is library that validates if an arithmetic operation would cause an integer overflow/underflow.
  - Until Solidity *version 0.8*, it had to be manually included and utilized by the smart contract developer
  - Since *version 0.8*, it is implemented on the language level

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

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

[1] https://docs.openzeppelin.com/contracts/2.x/api/math
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

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*

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);
        }
}
```

Secure Ether Transfer Idiom

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 idiom mitigates the risk associated with Ether transfers.

Secure Ether Transfer Idiom (cont.)

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.

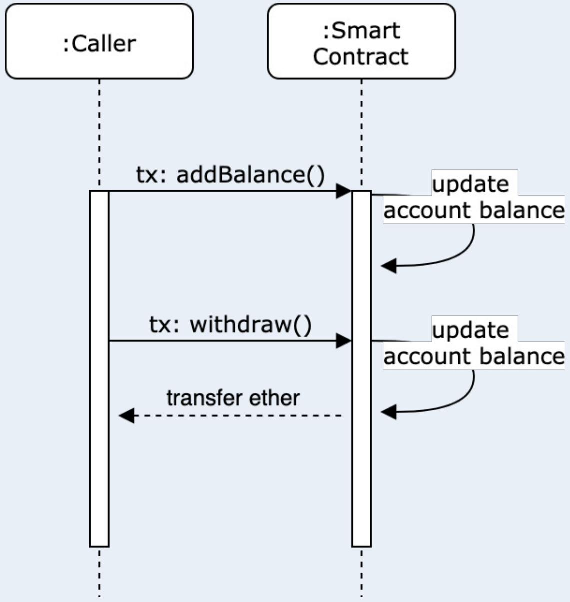
```
contract BadAuction {
    address highestBidder;
    uint highestBid;
    function bid() public payable {
        require(msg.value >= highestBid);
        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }
            highestBidder = msg.sender;
            highestBid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

Tight Variable Packing

Description

Gas is used to deploy a contract and utilize its capabilities. The purpose of this idiom is to reduce the quantity of gas utilized. This idiom is simple to use and has no effect on the contract logic.

Participants

In this idiom, the sole player is the contract that implements it. The modifications only affect how data is kept, no other entities dealing with the contract are affected.

Applicability

- Using the lowest data type feasible while yet ensuring accurate code execution.
- Group all data types that should belong together into a single 32-byte slot and declare them one by one in your code. It's crucial to group data types together since the EVM keeps variables in the order they are provided.

https://fravoll.github.io/solidity-patterns/tight_variable_packing.html, accessed on 24.03.2022

Tight Variable Packing (cont.)

Example:
In this example, there are two structs where one makes use of the tight variable packing idiom and the other does not. The one that utilizes packing can pack **a** and **c** into the same slot since they do not exceed 32 bytes when combined. Thus, 2 storage slots in total are sufficient to store the struct. Since **b** interrupts the order in the other struct, it requires 3 storage slots.

```
contract StructPacking{
        // 2 storage slots
        struct CheapStruct {
                uint128 a; // 16 bytes
                uint128 c; // 16 bytes
                uint256 b; // 32 bytes
        }
}
```

```
contract StructWithoutPacking{
        // 3 storage slots
        struct ExpensiveStruct {
                uint128 a; // 16 bytes
                uint256 b; // 32 bytes
                uint128 c; // 16 bytes
        }
}
```

Outline

1. Solidity Idioms
   - Access Restriction
   - Secure Ether Transfer
   - Tight Variable Packing

2. Solidity Design Patterns
   - Oracles
     - Synchronous Oracle
     - Asynchronous Oracle
   - Randomness
     - By Oracle
     - By Block Hash
     - By Commit-Reveal Scheme
   - Evolution Support
     - Data Segregation
     - Proxy

3. Token Standards
   - ERC20
   - ERC721

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

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

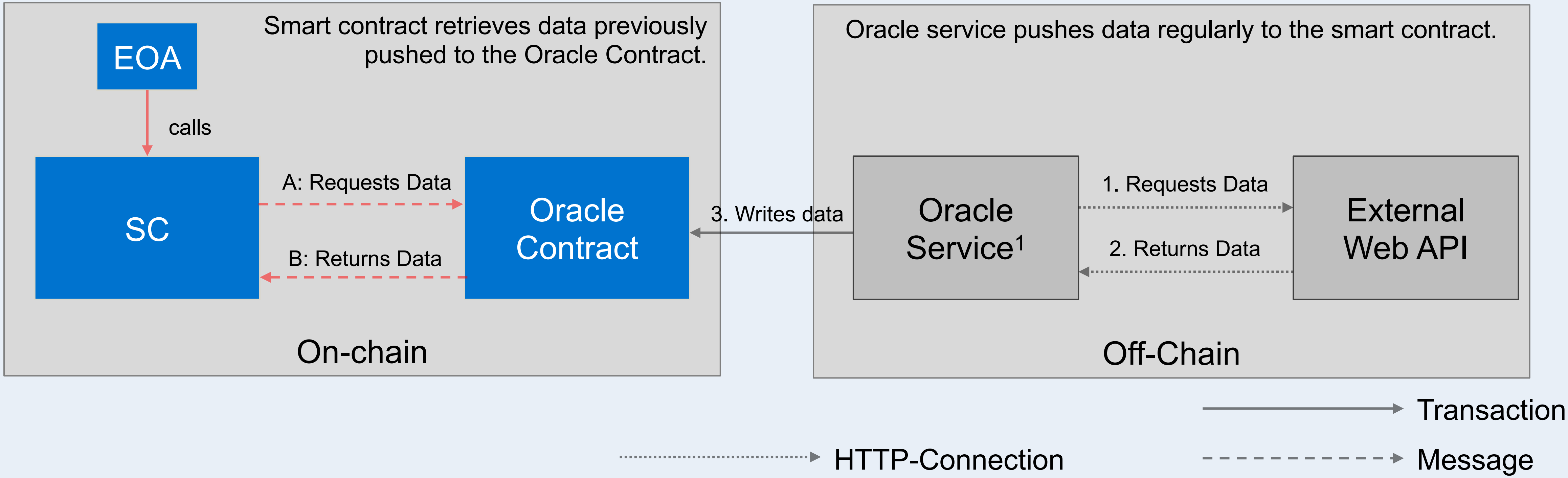
Any scenario in which a smart contract **relies on external data for computation** of future states.

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.*

Synchronous Oracle Contract Example

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.

08 Ethereum Design Patterns - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2022). “Blockchain-based Systems Engineering”. Lecture Slides. TU Munich.

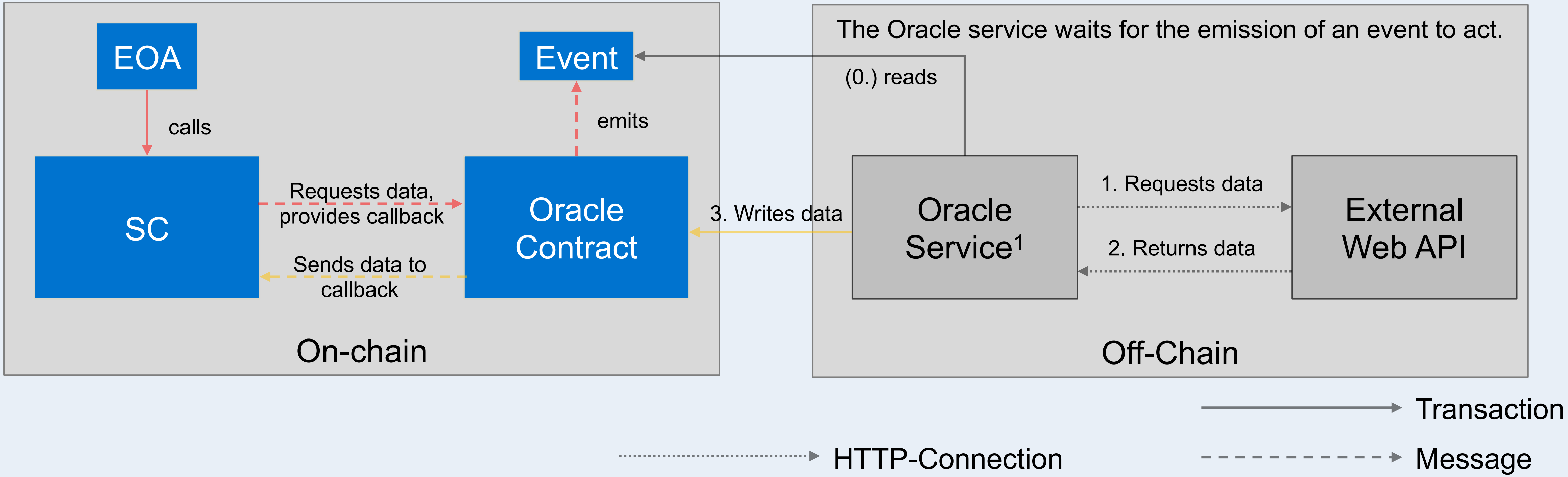
Asynchronous Oracle

Inversion of Control

Solution

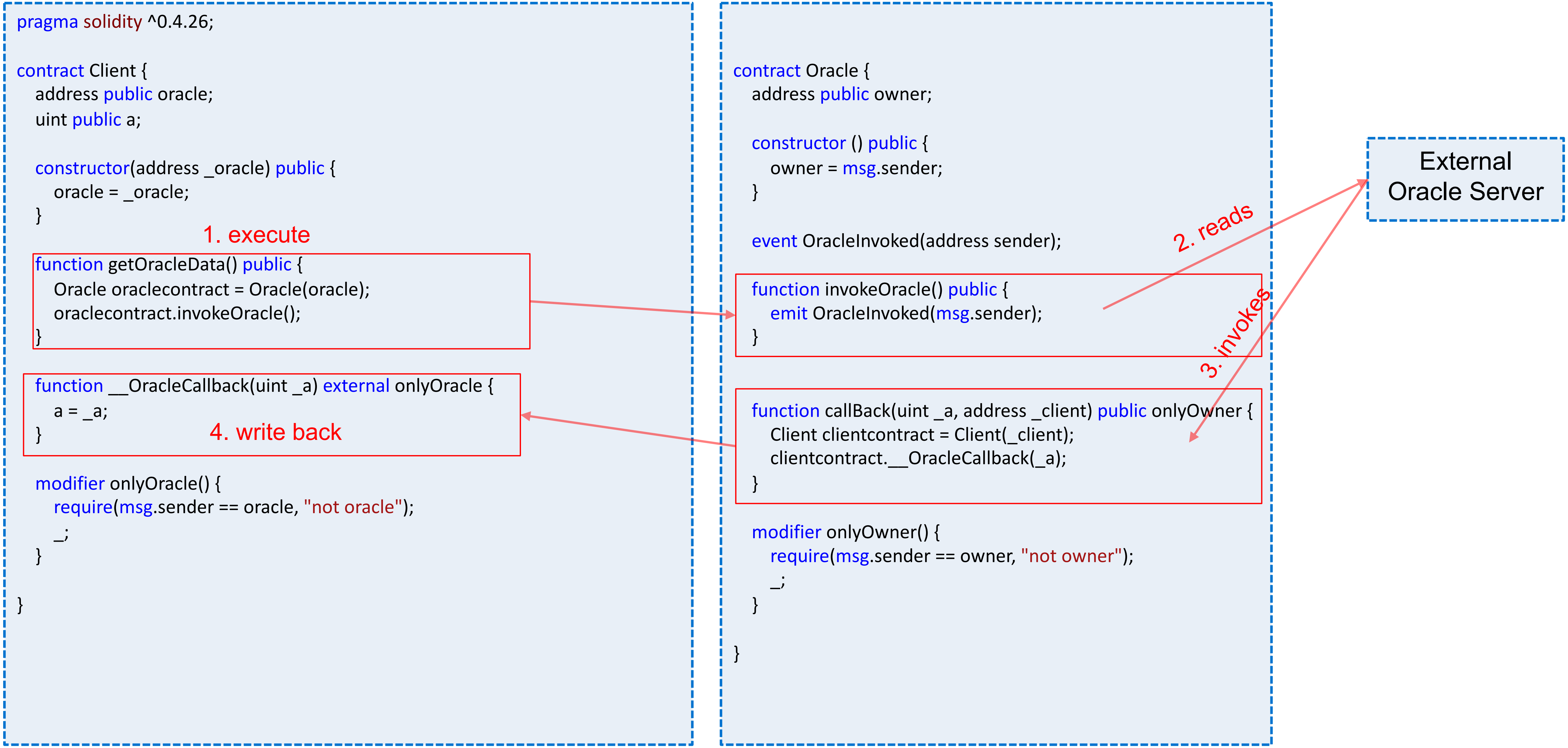
In contrast to the synchronous Oracle (in which data is periodically pushed to the Oracle contract and retrieved by the smart contract), the asynchronous Oracle acts as a proxy for the Oracle service to send fresh data to the original smart contract.

Asynchronous Oracle



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

Asynchronous Oracle Contract Example



```
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");
        _;
    }

}
```

External Oracle Server

1. execute

2. reads

3. invokes

4. write back

Limit Gas Usage in Asynchronous Oracle Contract

- 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");
        _;
    }

}
```

Oracle Pattern

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)

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.

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

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

Randomness Pattern

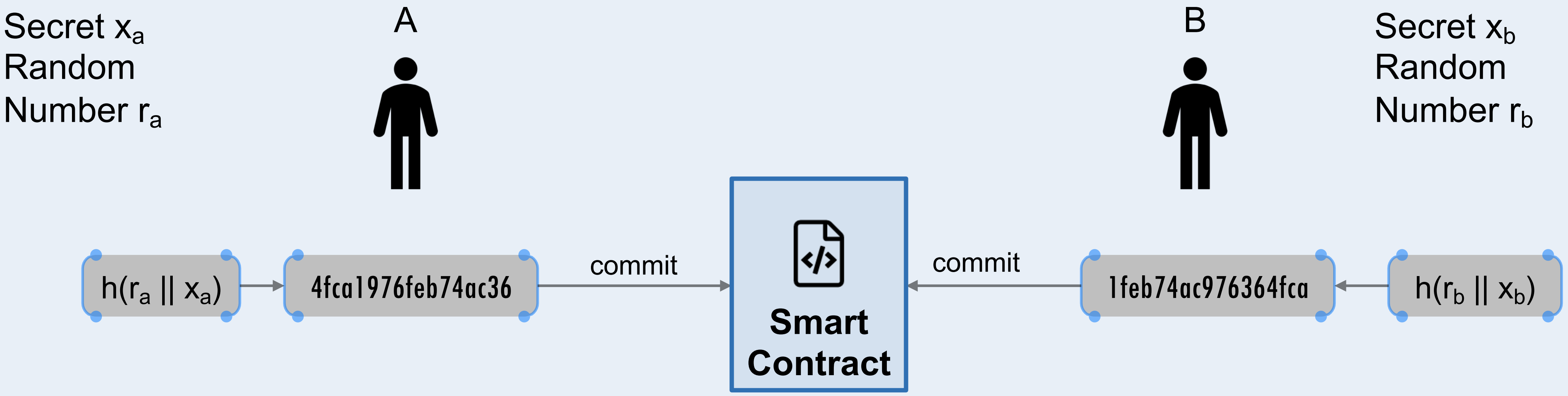
Solution with Commit and Reveal Scheme

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.

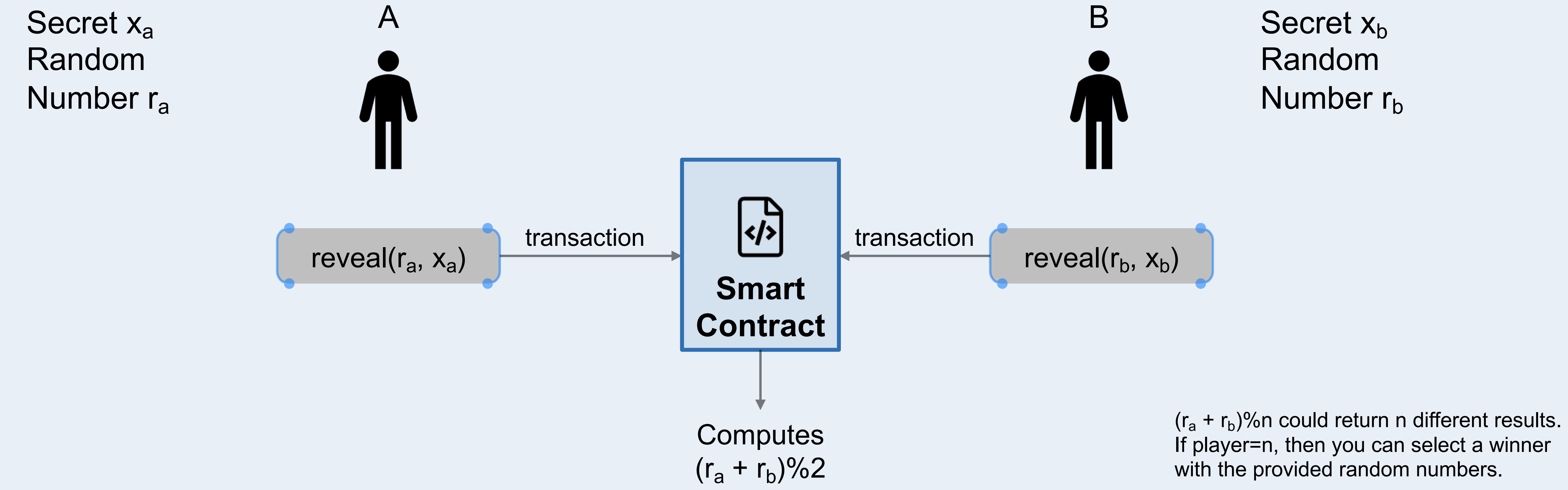
Phase 1: Commit



Randomness Pattern

Solution with Commit and Reveal Scheme (cont.)

Phase 2: Reveal



Secret x_a Random Number r_a
A
B
Secret x_b Random Number r_b
reveal(r_a, x_a)
transaction
Smart Contract
transaction
reveal(r_b, x_b)
Computes $(r_a + r_b)\%2$

$(r_a + r_b)\%n$ 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

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

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.

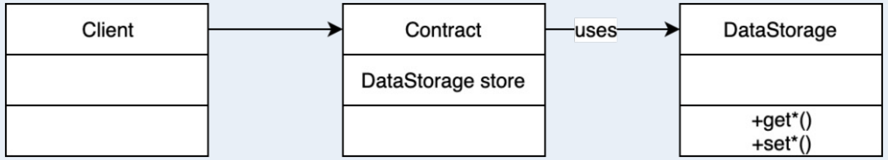
Applicability

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

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;
        }
}
```

08 Ethereum Design Patterns - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2022). “Blockchain-based Systems Engineering”. Lecture Slides. TU Munich.

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

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

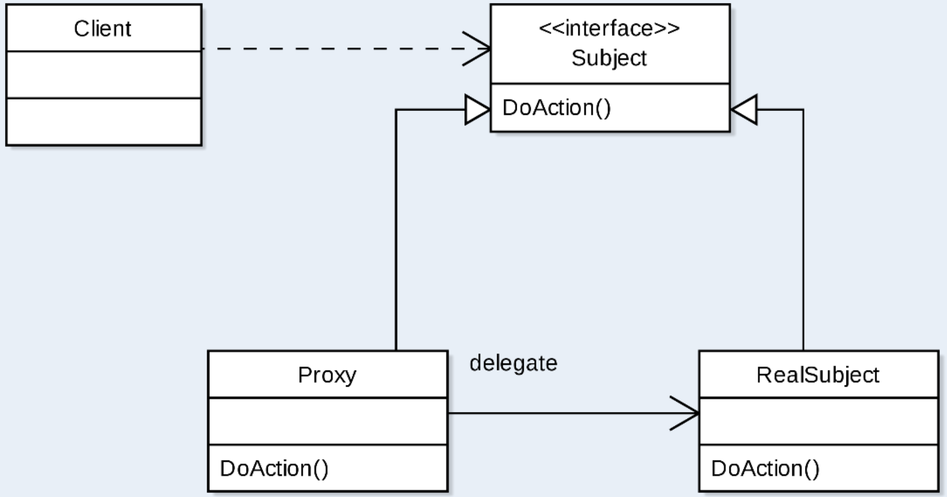
- Scenarios where a vulnerability is found in a smart contract,
- whenever the business logic needs to be updated in a way that the address of the contract does not change.

08 Ethereum Design Patterns - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2022). “Blockchain-based Systems Engineering”. Lecture Slides. TU Munich.

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;
        }
}
```

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

Outline

1. Solidity Idioms
   - Access Restriction
   - Secure Ether Transfer
   - Tight Variable Packing

2. Solidity Design Patterns
   - Oracles
     - Synchronous Oracle
     - Asynchronous Oracle
   - Randomness
     - By Oracle
     - By Block Hash
     - By Commit-Reveal Scheme
   - Evolution Support
     - Data Segregation
     - Proxy

3. Token Standards
   - ERC20
   - ERC721

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[1]: Each token is uniquely identifiable by an ID.

3. ERC-1155 Multi-token standard: Allows for each token ID to represent a new configurable token type, which may have its own metadata, supply and other attributes.

[1] This technology is being used in the contemporary NFT culture.

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

```
interface IERC20 {
        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);
}
```

ERC20 Reference Implementation (by OpenZeppelin)

```
contract ERC20 is Context, IERC20, IERC20Metadata {

        mapping (address => uint256) private _balances;
        mapping (address => mapping (address => uint256)) private _allowances;
        uint256 private _totalSupply;

        string private _name;
        string private _symbol;

        constructor (string memory name_, string memory symbol_) {
                _name = name_;
                _symbol = symbol_;
        }

        function name() public view virtual override returns (string memory){
                return _name;
        }

        function symbol() public view virtual override returns (string memory){
                return _symbol;
        }

        function decimals() public view virtual override returns (uint8){
                return 18;
        }
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

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

function balanceOf(address account) public view virtual override 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 to, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, amount);
        return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

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

/**
* @dev See `IERC20.approve`.
*
* Requirements:
*
* - `spender` cannot be the zero address.
*/
function approve(address spender, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, amount);
        return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @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:
* - `from` and `to` cannot be the zero address.
* - `from` must have a balance of at least `amount`.
* - the caller must have allowance for `from``'s tokens of at least
* `amount`.
*/
function transferFrom(address from, address to, uint256 amount) public virtual override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev Atomically increases the allowance granted to spender by the caller
*
* Emits a `Approval` event.
*
* Requirements:
*
* - `spender` cannot be the zero address.
*/
function  increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool){
        address owner = _msgSender();
        _approve(owner, spender, allowance (owner, spender) + addedValue);
        return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev Atomically decreases the allowance granted to spender by the caller
*
* Emits a `Approval` event.
*
* Requirements:
*
* - `spender` cannot be the zero address.
* - `spender` must have allowance for the caller of at least.
* - `subtractedValue`.

function decreaseAllowance(address spender , uint256 subtractedValue) public virtual returns (bool){
        address owner = _msgSender();
        uint256 currentAllowance = allowance (owner, spender);
        require(currentAllowance >= subtractedValue, „ERC20: decreased allowance below zero“);
        unchecked {
                _approve (owner, spender, currentAllowance – subtractedValue);
        }

        return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev Moves tokens `amount` from `sender` to `recipient`.
*
* Emits a `Transfer` event.
*
* Requirements:
*
* - `from` cannot be the zero address.
* - `to` cannot be the zero address.
* - `from` must have a balance of at least `amount`.
*/
function _transfer(address from, address to, uint256 amount) internal virtual {
        require (from != address(0), "ERC20: transfer from the zero address");
        require (to != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer (from, to, amount);

        uint256 fromBalance = _balances[from];
        require (fromBalance >= amount, „ERC20: transfer amount exceeds balace“);
        unchecked{
                _balances[from] = fromBalance – amount ;
        }
        _balances[to] += amount;
        emit Transfer (from, to, amount);
        _afterTokenTransfer (from, to, amount);

}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev creates `amount` tokens and assigns them to  `account`
*
* Emits a `Transfer` event with `from` set to the zero adress .
*
* Requirements:
*
* - `account` cannot be the zero address.
*/
function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer (adress(0), account, amount);

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer( address(0), account, amount);

        _afterTokenTransfer(address(0), account, amount);
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev Destroys `amount` tokens from `account` reducing the total supply.
*
* Emits a `Transfer` event.
*
* Requirements:
*
* - `account` cannot be the zero address.
* - `account` must have at least `amount` tokens .
*/
function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account];
        require (accountBalance >= amount, „ERC20: burn amount exceeds balace“);
        unchecked{
                _balances[account] = accountBalance – amount ;
        }
        _totalSupply -= amount;
        emit Transfer(account, address(0), amount);
        _afterTokenTransfer(account, address(0), amount);
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @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 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

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

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev updates `owner` s allowance for `spender` based on spent `amount`
*
* Might emit an `approval` event.
*
*/
function _spendAllowance(address owner, address spender, uint256 amount) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type (uint256.max){
        require(currentAllowance >= amount, „ERC20: insufficient allowance“);
        unchecked{
                _approve (owner, spender, currentAllowance – amount);
        }
    }
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

ERC20 Reference Implementation (by OpenZeppelin)

```
/**
* @dev Hook that is called before any transfer of tokens.
*
* Conditions:
*
* - when `from` and `to` are both non-zero, `amount` of `from` s tokens will
* be transfered to ` to`
* - when `from` is zero, `amount` tokens will be minted for ` to`.
* - when `to` is zero, `amount` of `from` tokens will be burned
* - `from` and `to` are never both zero.
*/
function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual {}

/**
* @dev Hook that is called after any transfer of tokens.
*
* Conditions:
*
* - when `from` and `to` are both non-zero, `amount` of `from` s tokens will
* be transfered to ` to`
* - when `from` is zero, `amount` tokens have been minted for ` to`.
* - when `to` is zero, `amount` of `from` tokens have been burned
* - `from` and `to` are never both zero.
*/
function _afterTokenTransfer(address from, address to, uint256 amount) internal virtual {}

}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

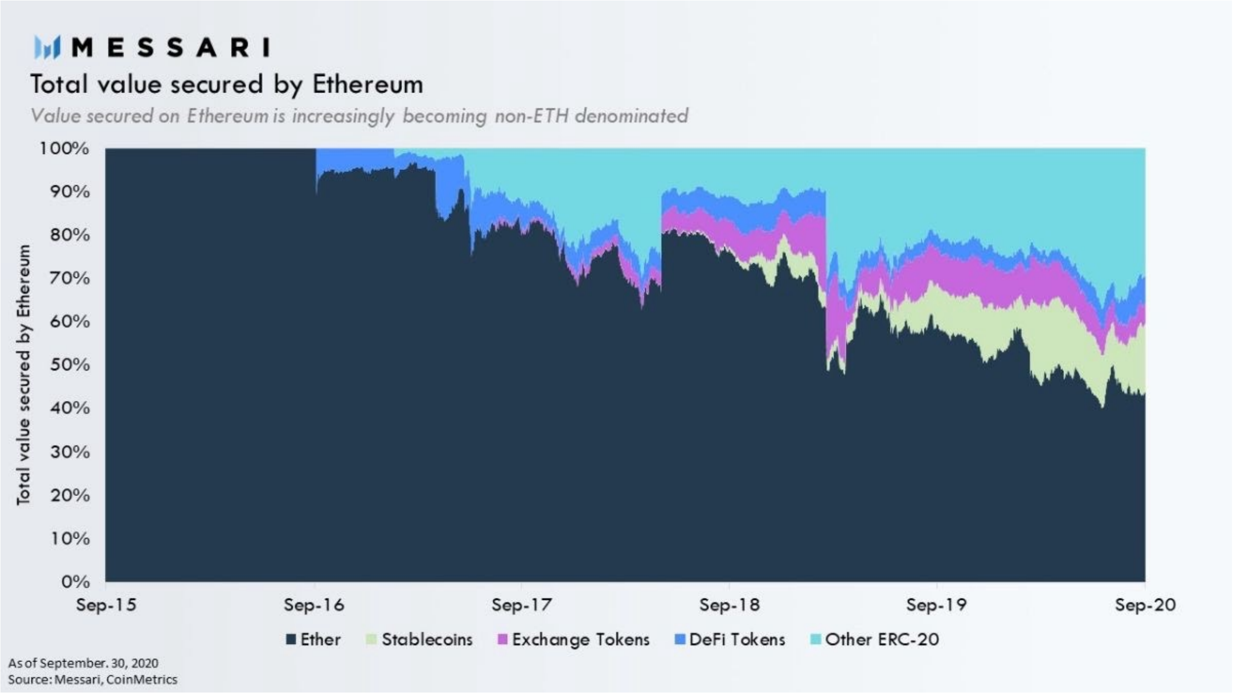
Ethereum Tokens (cont.)

A brief explanation of the tokens that are mentioned in the graph:

- **Stablecoins** are digital currencies whose value is linked to real-world assets such as the US dollar.

- An **exchange token** is a digital asset that is unique to a cryptocurrency exchange. Most exchange tokens are aimed to boost an exchange's liquidity, incentivize trading activity, or simplify an exchange's community governance process in general.

- **DeFi tokens** are a collection of coins that are native to automated, decentralized networks that use smart contracts to function. These provide customers access to a suite of blockchain-based financial applications and services, as well as $75 billion worth of cryptocurrency (as of 20th March 2022).



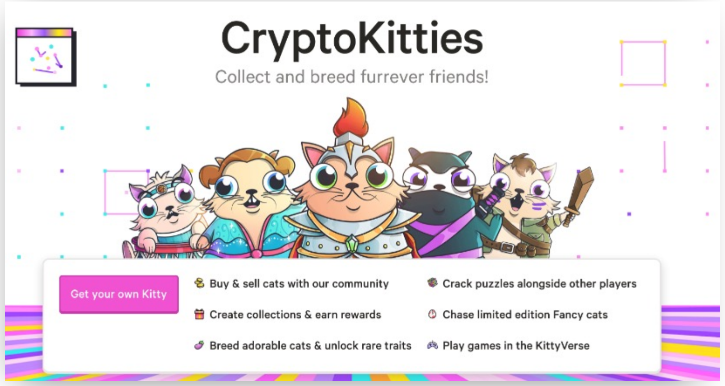
As of September. 30, 2020
Source: Messari, CoinMetrics

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

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

Decentraland

ER721 Token Standard

Interface

```
pragma solidity ^0.4.20;

/// @title ERC-721 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);
}
```