WEEK 8

Blockchains & Distributed Ledgers



Christina Ovezik

CC BY

Slide credits: CO, Aggelos Kiayias, Dimitris Karakostas

Lecture Outline

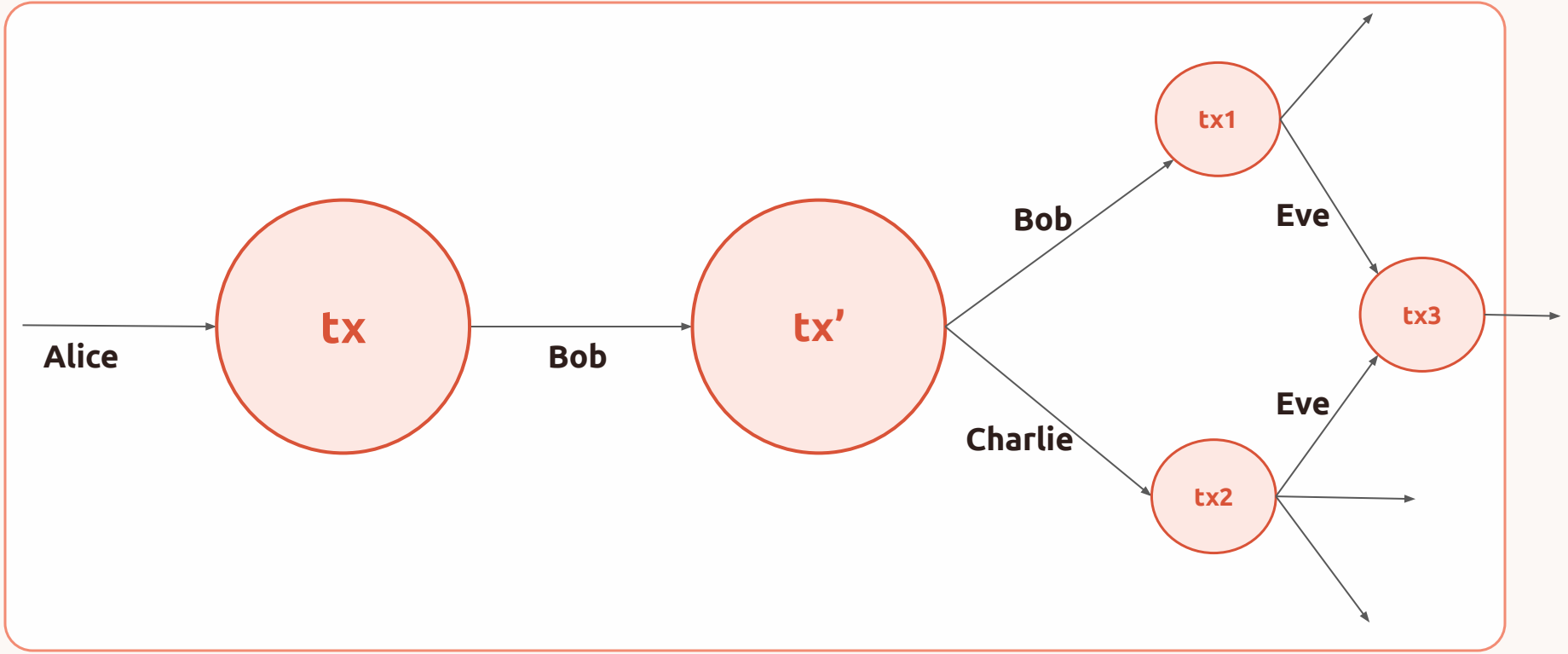
- Introduction to smart contracts
- Bitcoin scripting
- Ethereum architecture and EVM
- Solidity programming basics
- Common security vulnerabilities and attacks
- Good design patterns and gas fairness

What is a smart contract?

- Computer Program
- Contract code is executed independently by all full nodes in the network.
- The outcome of a smart contract is the same for everyone.
- Once deployed, the code of a smart contract cannot be modified.

Bitcoin

Bitcoin Transactions



Bitcoin Programs

Transaction: A transfer of value in the Bitcoin network.

INPUT

A transaction output from which it spends bitcoins:

- **Previous tx address**
- **Index**
- **ScriptSig**

OUTPUT

Instructions for spending the sent bitcoins:

- **Value:** Amount of bitcoins to send
- **ScriptPubKey:** Instructions on how to spend the sent bitcoins

To Validate a Transaction:

1. Concatenate ScriptSig (current tx) with ScriptPubKey (referenced tx)

2. Execute & verify if combined script executes successfully

Bitcoin Script

Execution Architecture
Stack-based Execution

Script Notation
Data enclosed in angle brackets: **<sig>**, **<pubKey>**

Opcodes: Commands or Functions

Arithmetic
OP_ABS, OP_ADD

Stack Operations
OP_DROP, OP_SWAP

Flow Control
OP_IF, OP_ELSE

Bitwise Logic
OP_EQUAL, OP_EQUALVERIFY

Cryptographic Hashing
OP_SHA1, OP_SHA256

Signature Verification
OP_CHECKSIG, OP_CHECKMULTISIG

Locktime Constraints
OP_CHECKLOCKTIMEVERIFY
OP_CHECKSEQUENCEVERIFY

Bitcoin Unspent Transaction Output (UTxO)

UTxO Definition: An unspent output of a transaction that can be spent in a future transaction.

PREVIOUS BLOCKS



Chain of prior transaction outputs

BLOCK N

Output:

```
OP_DUP
OP_HASH160
<pubKeyHash1>
OP_EQUALVERIFY
OP_CHECKSIG
```

ScriptPubKey

LOCKING SCRIPT

ScriptPubKey Role:

Defines the spending conditions required to unlock this UTxO in a future transaction.

P2PKH Pattern:

Standard Pay-to-Public-Key-Hash script requiring a signature and public key matching **`pubKeyHash1`**.

Bitcoin Script Example Step by Step

Stack	Script	Description
Empty	`<sig1> <pubKey1> OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG`	Add constant values from left to right to the stack until we reach an opcode.
`<sig1> <pubKey1>`	`OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG`	Duplicate top stack item
`<sig1> <pubKey1> <pubKey1>`	`OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG`	Hash at the top of the stack
`<sig1> <pubKey1> <pub1Hash>`	`<pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG`	Push the hashvalue to the stack
`<sig1><pubKey1> <pub1Hash><pubKeyHash1>`	`OP_EQUALVERIFY OP_CHECKSIG`	Check if top two items are equal
`<sig1> <pubKey1>`	`OP_CHECKSIG`	Verify the signature.
Empty	`TRUE`	If stack empty return True, else return False.

Bitcoin’s scripting language limitations

Lack of Turing-completeness: No loops

Lack of state: Cannot keep internal state.

Value-blindness: Cannot control the amount being sent – balance is pushed forward.

Blockchain-blindness: Cannot access block header values such as nonce, timestamp and previous hash block.

Extending Bitcoin functionality

Building a protocol on top of Bitcoin:

Pros:

- Take advantage of the underlying network and mining power.
- Lower development cost

Cons:

- Little flexibility – have to work within Bitcoin's limitations at the Bitcoin API.

Build an independent network:

Pros:

- Easy to add and extend new opcodes.
- Flexibility.

Cons:

- Need to attract miners / maintainers to sustain the network.
- Higher development cost.

Ethereum

Same principles as Bitcoin

- **A peer-to-peer network:** connects the participants
- **Sybil resistance:** Proof-of-Stake (former Proof-of-Work)
- **A digital currency:** ether
- **A global ledger:** the blockchain
  - Addresses: key pair
  - Wallets
  - Transactions: digital signatures
  - Blocks

Ethereum: A universal Replicated State Machine

- **Transaction-based deterministic state machine**
  - Global state (singleton)
  - A virtual machine that applies changes to global state
- **A global decentralized computing infrastructure**
- **Anyone can create their own state transition functions**
- **Stack-based bytecode language**
- **Turing-completeness**
- **Smart contracts**
- **Decentralized applications (Dapps)**

Ethereum accounts



Global State

The global state of Ethereum is composed of **accounts**.



Interactions

Accounts **interact** with each other through **transactions** (or messages).



Address & State

Each account maintains a **state** and a **20-byte address** (160-bit identifier).

Ethereum account



address

The unique 20-byte public identifier that represents the account on the network.



balance

The current amount of Ether owned by this account, denominated in Wei.

Note: There are no UTXoS in Ethereum!



nonce

A counter representing the number of transactions sent from this account.

UTxO vs Accounts



UTxO pros:

• Harder to link transactions → Potentially higher degree of privacy

• Better for parallelism: valid transactions have the same effect on the blockchain state in any order



Account pros:

• Space saving

• Conceptual simplicity

Two types of accounts



Personal accounts

(what we've seen)



Contract accounts

Ethereum Contract Account

Unique to Contract Accounts (Empty in Personal Accounts)



address

H(addr + nonce of creator)



code

EVM Bytecode to be executed



storage

Persistent state data of the contract



balance

ETH balance (in Wei)



nonce

transactions sent

Personal vs contract accounts

	Personal account	Contract account
address	H(pub_key)	H(addr + nonce of creator)
code	∅	Code to be executed
storage	∅	Data of the contract
balance	ETH balance (in Wei)	ETH balance (in Wei)
nonce	# transactions sent	# transactions sent

Ethereum transaction

from	signature	to	amount



from

The sender address initiating the transaction



signature

Digital signature created by the sender's private key



to

Recipient address



amount

Amount being transferred (in Wei)

Note: Every transaction has a "nonce" that counts the transactions sent by the address.
This ensures "freshness" and protects against replay attacks

A transaction about a contract

from	signature	to	amount	data ★
from Sender address	**signature** Digital signature	**to** Target address	**amount** Value transferred	**data** **Contract payload**

Role of the data field in Ethereum transactions:

Transaction about personal accounts:

- **Field is unused**

Standard transfer of ETH requires no data payload.

Transaction about contracts:

- **Will contain data about the contract**

Encodes function call signatures and input parameters.

Transaction for contract creation

from	signature	to ★	amount	data ★
from Sender address	**signature** Digital signature	**to** **Empty recipient**	**amount** Optional contract funding	**data** **Contract code**

Requirements for Contract Creation in Ethereum:

to Field: Empty Recipient

- **Must be left intentionally empty (null address)**

Signals to the network to create a new contract account.

data Field: Contract Payload

- **Smart contract code + initial arguments**

Contains compiled EVM bytecode and constructor parameters.

Transaction for contract interaction

from	signature	to ★	amount ★	data ★
from Sender address	**signature** Digital signature	**to** Contract address	**amount** Contract funding	**data** Method call + args

Key Fields for Contract Interaction in Ethereum:

to Field

- **Contract Address**

Identifies target smart contract account on the network.

amount Field

- **Transferred ETH**

Amount transferred directly to contract's account.

data Field

- **Method Call + Arguments**

Specifies which function to invoke and with what arguments.

Contract method call

When contract account is activated:



Code Execution

Contract **code** runs



State Access

It can read/write to **internal storage**



External Interactions

It can **send other transactions** or **call other contracts**



Passive Entities

Contracts can’t initiate new transactions on their own

Can only send transactions in response to other transactions received

Messages

Like **transactions** except they are **produced by a contract**

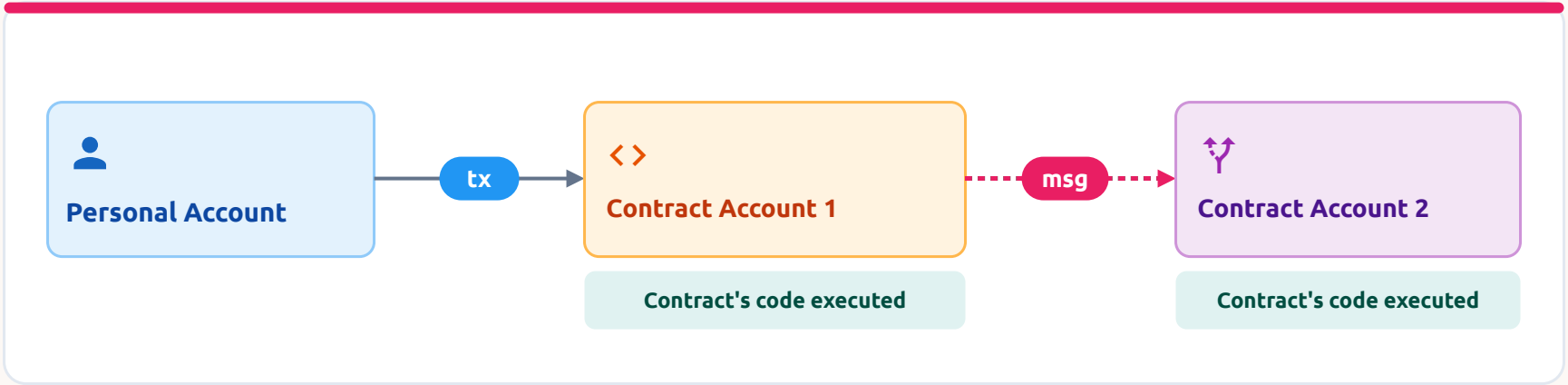
Virtual objects

Exist **only** in the **Ethereum execution environment – not in the network**

A message leads to the recipient account running its code

Contracts can have **relationships** with **other contracts**

Transactions & Messages



Types of transactions

	send	create	call
from	sender	creator	caller
signature	sig	sig	sig
to	receiver	∅	contract
amount	ETH	ETH	ETH
data	∅	code	f, args

Ethereum Virtual Machine

- Series of **bytecode** instructions (EVM code)
- Each **bytecode** represents an **operation** (opcode)
- A quasi **Turing complete** machine
- **Stack-based** architecture (1024-depth)
- **32-byte** words (256-bit words)
- **Crypto** primitives

EVM: contract execution



Three types of storage:

- **Stack**
- **Memory** (expandable byte array)
- **Storage** (key/value store)



Access:

- **value**
- **sender**
- **data**
- **gas** limit
- **block header** data (depth, timestamp, miner, block id, ...)

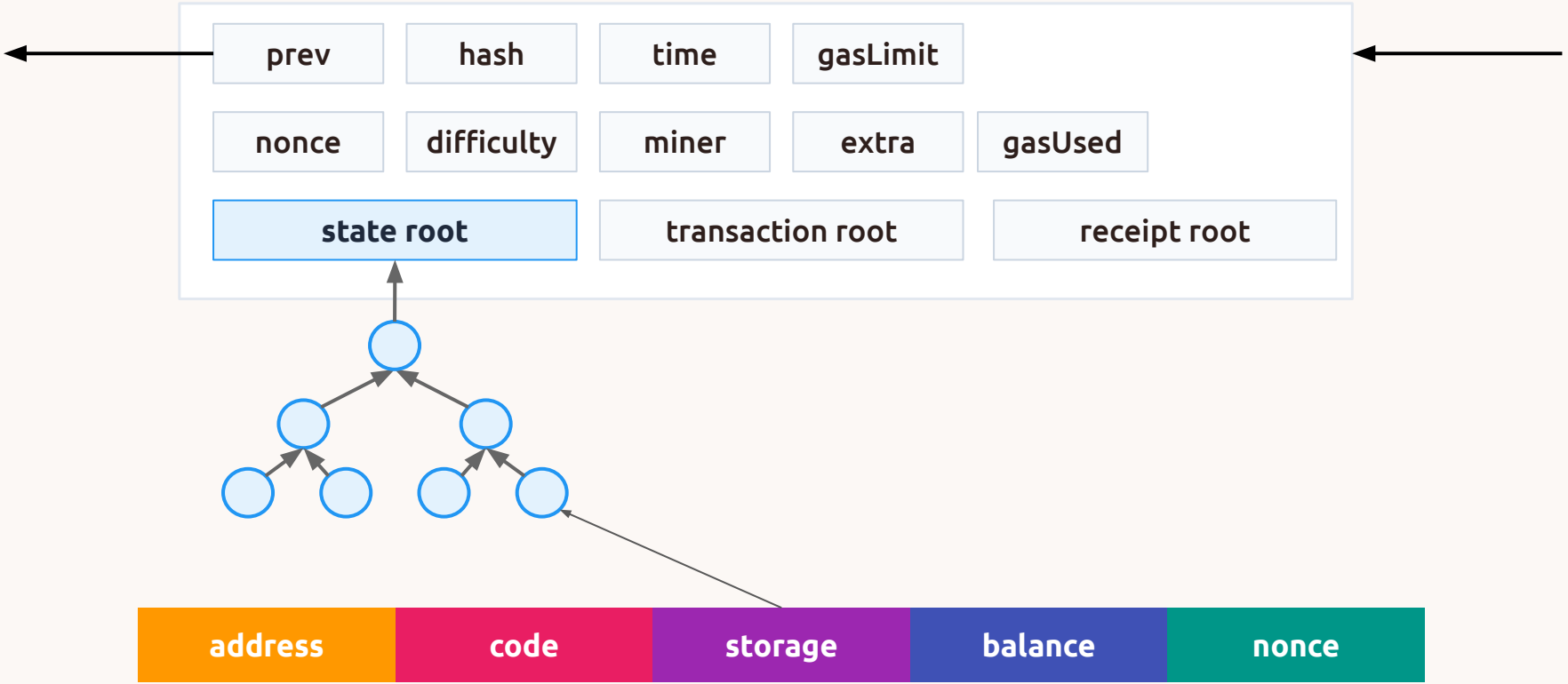


All memory is **zero-initialized**

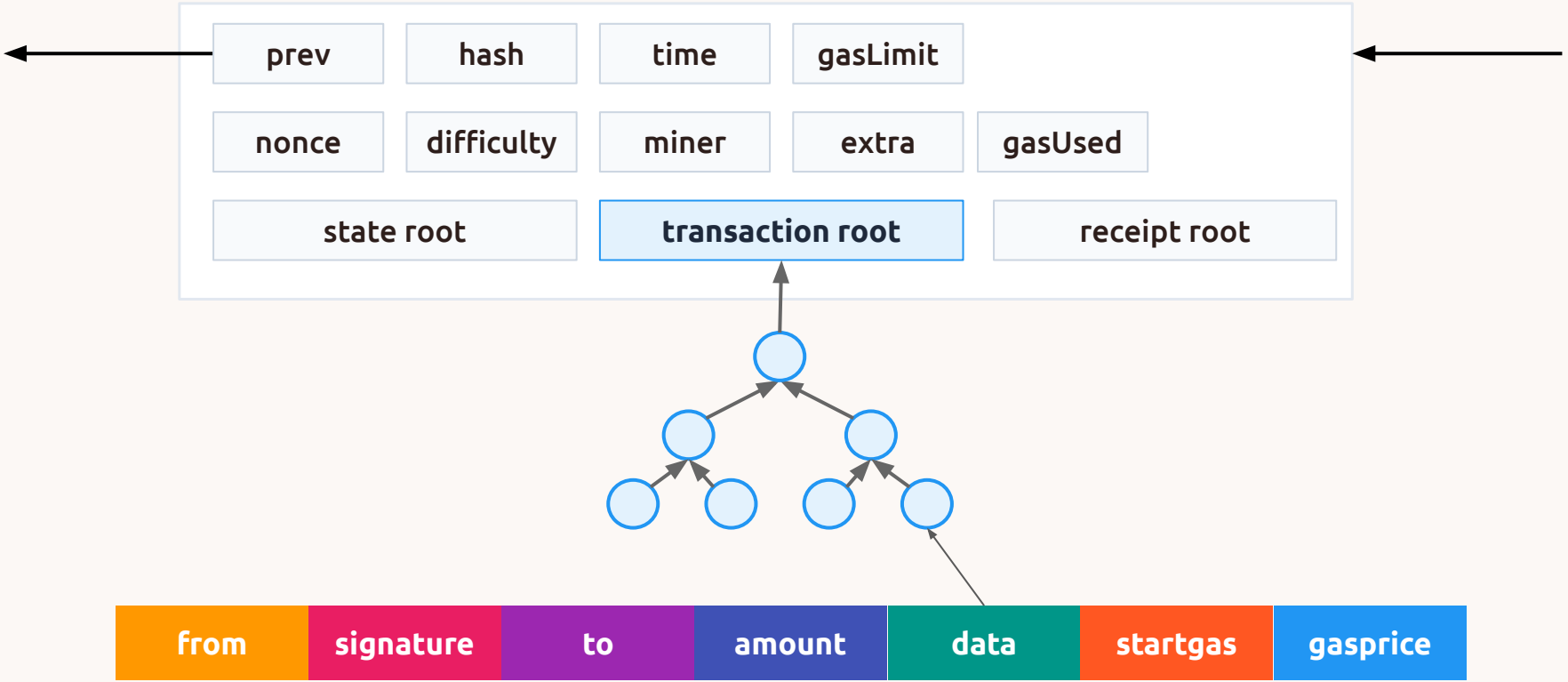


All nodes run EVM and execute smart contract code

Ethereum block



Ethereum block



Ethereum Block Production

- **Blocks** contain: **transaction** list and most **recent state**
- Block **time**: 12 **seconds**
- *(Since 2022)* Proof-of-stake (Gasper)
    - Previously **Proof-of-work**: Ethash (originally designed to be **memory-hard**)
- Block **rewards and fees**:
    - Previously: 2 ETH + tx fees (paid to miner)
    - Now:
        - Base block reward (new ETH) paid to block proposer (validator)
            - amount depends on total amount of ETH staked
        - Base transaction fee **burned,** reducing the total supply of ETH
        - Optional transaction priority fees (**tips**) paid to block proposer
        - [more here]

Transaction fees: the phone booth model



Gas: a necessary evil

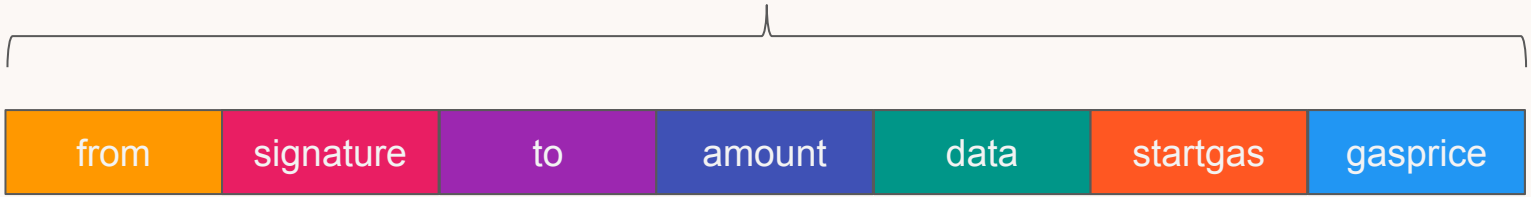
- Every node on the network:
  - evaluates all **transactions**
  - stores all **state**
- The *halting problem*:
  - Miners cannot determine if a program can/will finish

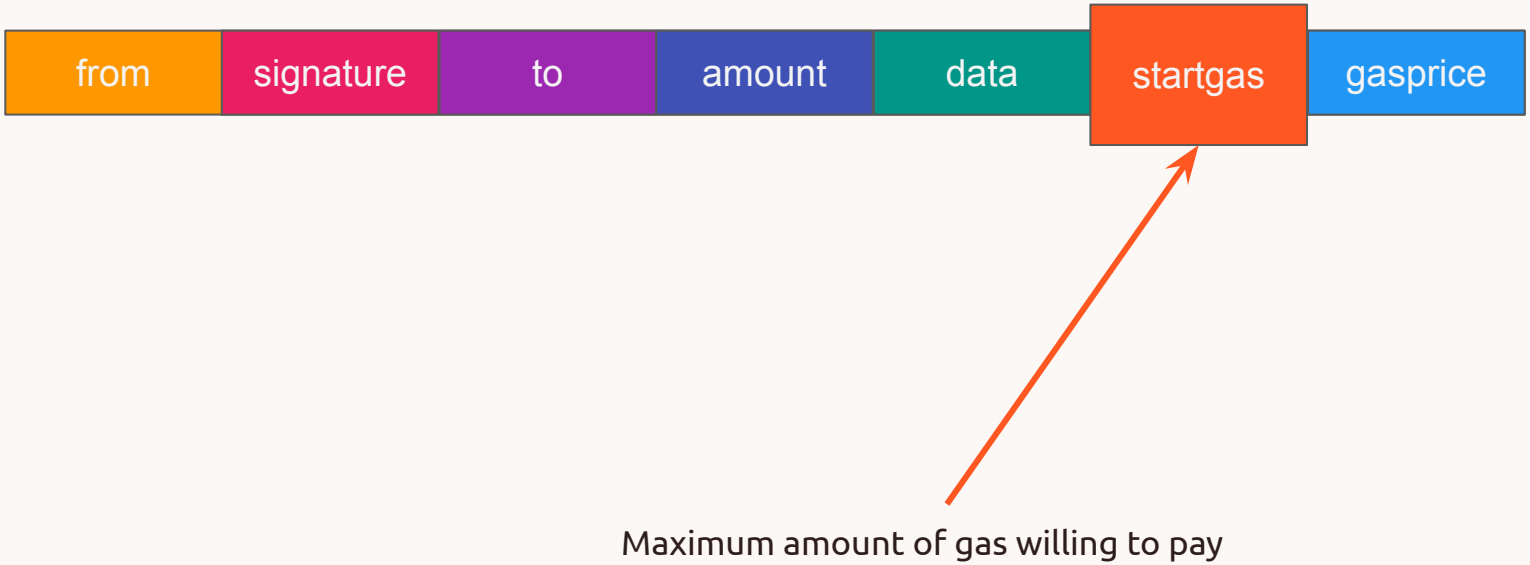
Solution

- Every **computation step** has a **fee**
- Fee is **paid** in **gas**
- **Gas** is the **unit** used to **measure computations**



Ethereum transaction

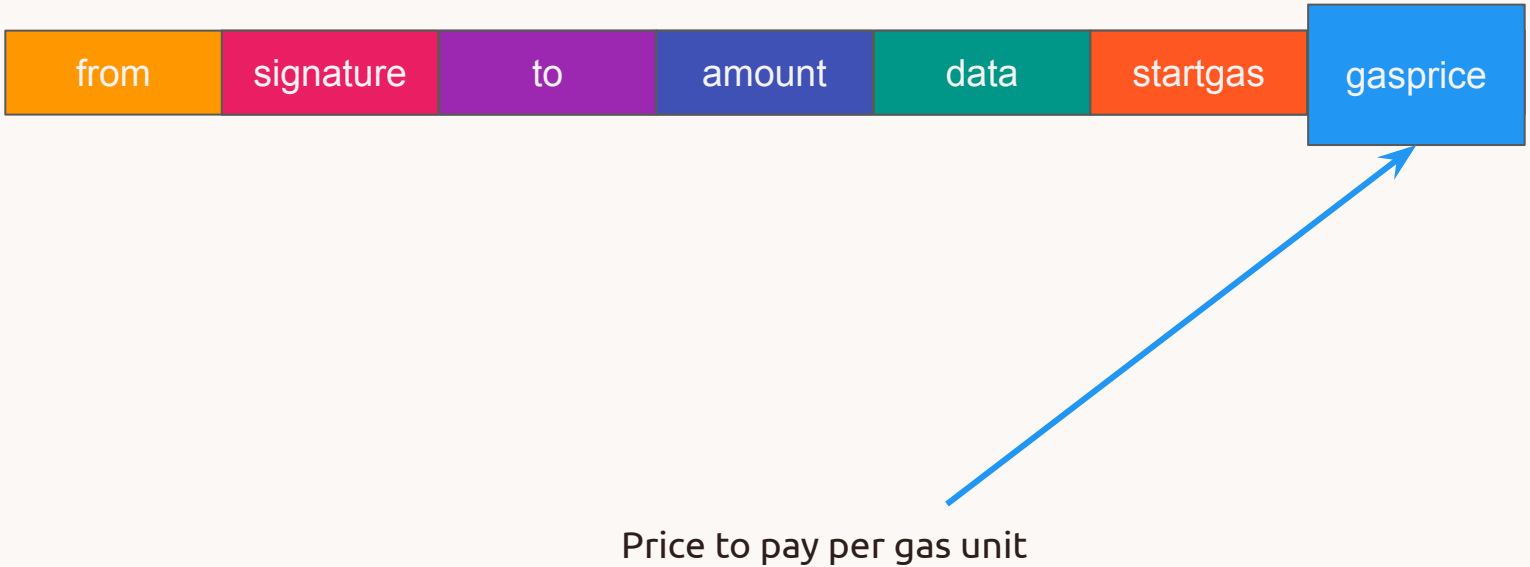



from
signature
to
amount
data
startgas
gasprice
Maximum amount of gas willing to pay

Gas Limit

- Equals to startgas
- All **unused gas** is **refunded** at the end of a transaction
- **Out of gas** transactions are **not refundable**
- Blocks also have a **gas limit**




from
signature
to
amount
data
startgas
gasprice
Price to pay per gas unit

Gas Price

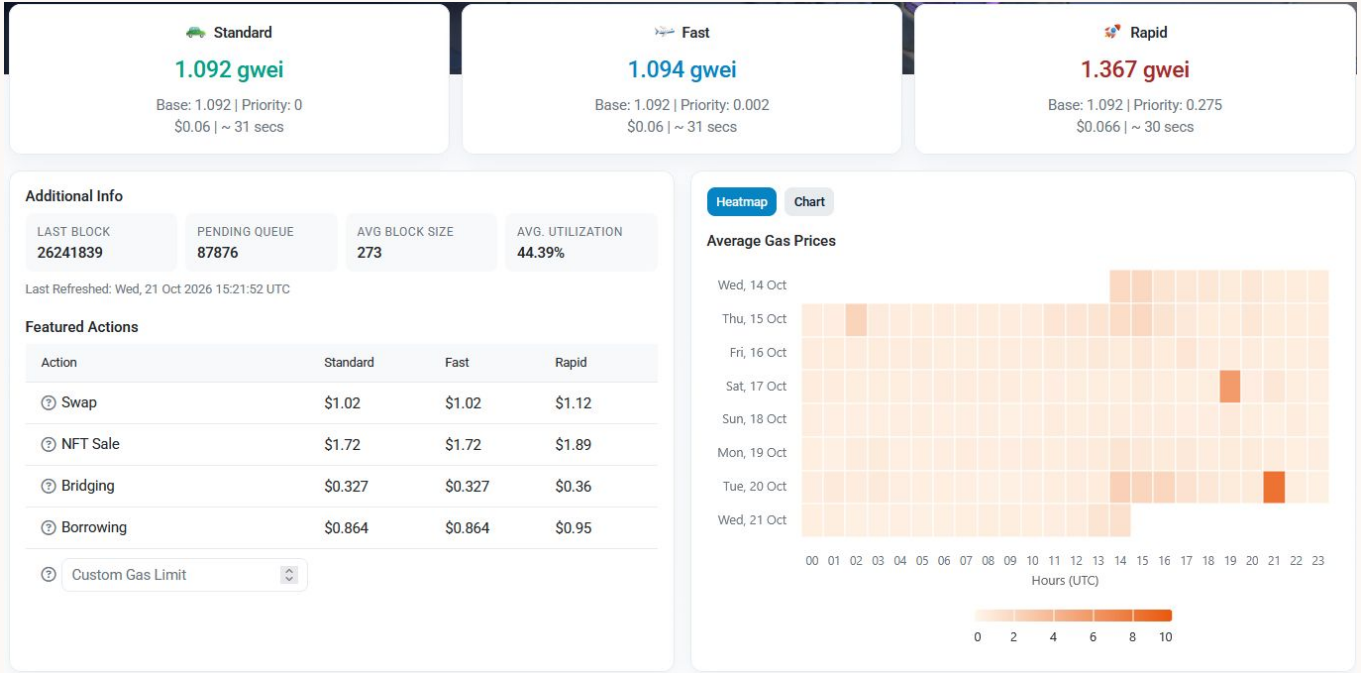
- Measured in **gwei** (10^9 Wei)
- Determines how **quickly** a transaction will be **included in a block**
    - Higher gas price makes transactions more appealing to miners



Transaction Fees (example)



Gas price (Oct 2026)



https://etherscan.io/gastracker

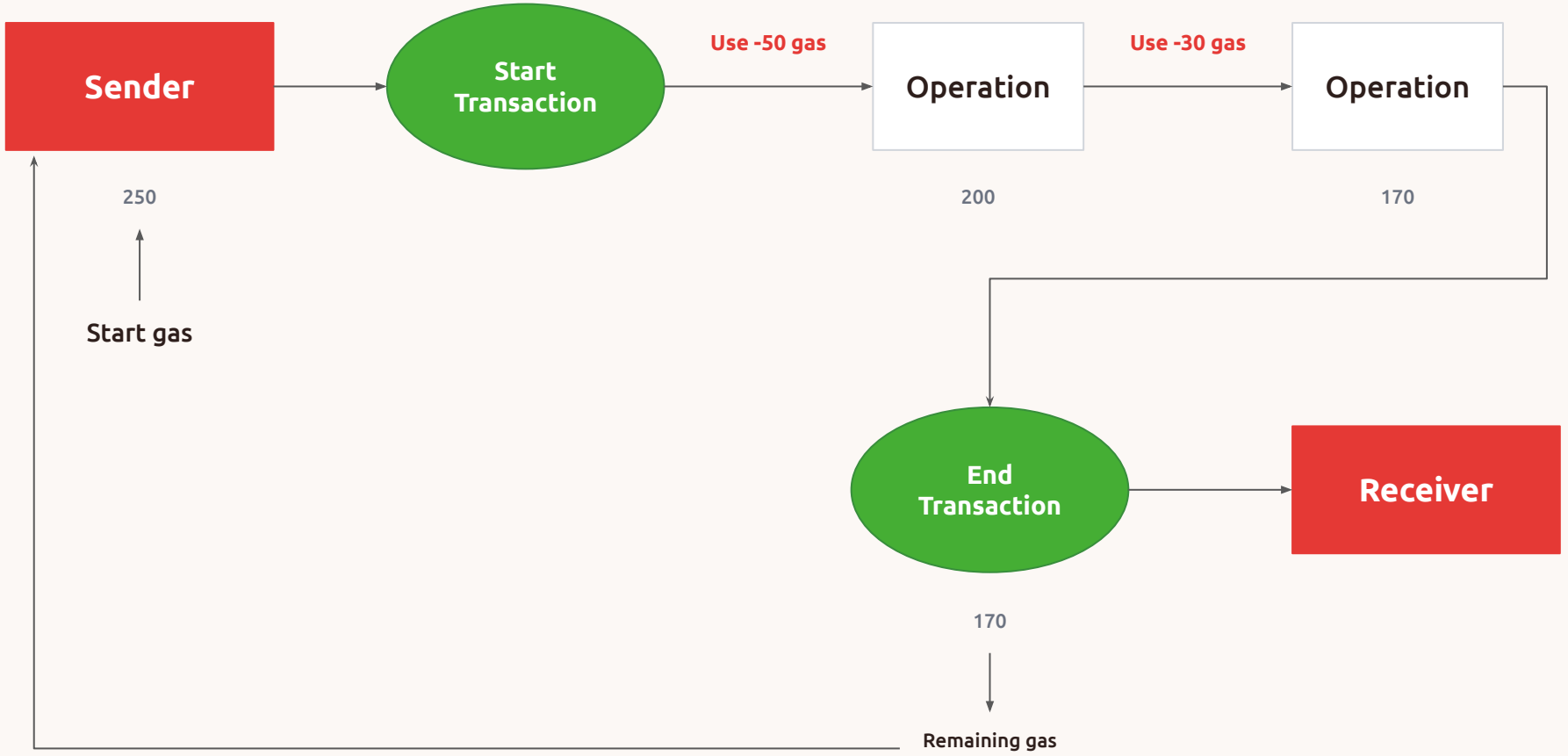
Storage in Ethereum

ETH Price: $2,502 (21 October 2026) - Gas Price: 1.092 Gwei

Data Size	1KB	1MB	10MB
Gas Required	677,000	~693,000,000	~7,000,000,000
Cost in ETH	~0.00074 ETH	~0.757 ETH	~7.64 ETH
Cost (USD)	**$1.85**	**$1,893**	**$19,125**

Computation steps

- If **gas_limit** * **gas_price** > **balance** then **halt**
- **Deduct** gas_limit * gas_price from **balance**
- Set gas = gas_limit
- **Run code** deducting from gas
- After termination **return remaining gas** to **balance**


Sender
Start Transaction
Use -50 gas
Operation
Use -30 gas
Operation
250
200
170
Start gas
End Transaction
Receiver
170
Remaining gas

Out of gas exceptions

- State **reverts** to **previous state**
- gas_limit * gas_price is **still deducted** from **balance**



Introduction to Solidity

Solidity

- A **high level programming** language for **writing** smart **contracts** on **Ethereum**
- **Compile** code for the **Ethereum Virtual Machine**
- **Syntax** similar to **JavaScript**
- **Remix IDE is the most widely used editor for Solidity**

Documentation: docs.soliditylang.org

Solidity

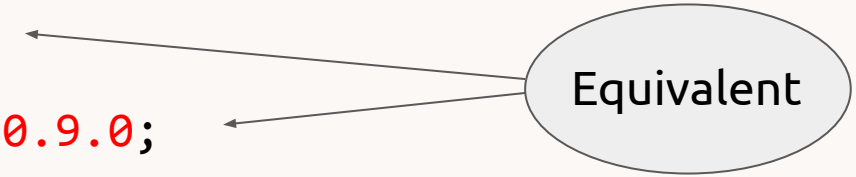
- **Contracts** look like **classes / objects**
- **Statically**-typed language (variable types must be set explicitly)
- Most of the control structures from **JavaScript** are available in **Solidity** (conditions, loops, exception handling, etc.)

HelloWorld contract

```
pragma solidity >=0.8.0 <0.9.0;
contract HelloWorld {
    function print () public pure returns (string memory) {
return 'Hello World!';
    }
}
```

Pragmas

```
pragma solidity 0.8.0;

pragma solidity ^0.8.1;

pragma solidity >=0.8.1 < 0.9.0;
```



The pragma keyword specifies which compiler version to use. Follows the same syntax used by npm.

Contract

```
contract <ContractName> { ... }
```

Constructors

```
contract HelloWorld1 {
    constructor () { … }
}


contract HelloWorld2 {
    constructor (uint x, string y) { … }
}
```

Solidity: Variables

- State variables:
  - Contract variables
  - **Permanently stored** in contract **storage**
  - **Must declare** at compilation time
- Local variables
  - Within a **function**: **cannot** be **accessed** outside
  - **Complex** types: at **storage** by default
  - **Value** types: in the **stack**
  - Function **arguments**

Types

- The **type** of each variable **needs to be specified** (Solidity is a statically typed language)
- **Two** categories:
  - **Value** types
  - **Reference** types
- “**undefined**” or “**null**” values **do not exist** in Solidity
- **Variables** without a value **always** have a **default value** (zero-state) dependent on their type.
- Solidity follows the scoping rules of C99 (variables are visible until the end of the smallest {}-block)

Value types

Types: booleans

```
contract Booleans {
    bool p = true;
    bool q = false;
}
```

Operators: `!`, `&&`, `||`, `!=`, `==`

Types: integers

```
contract Integers {
    uint256 x = 5;
    int8 y = -5;
}
```

- Two types:
  - `int` (signed)
  - `uint` (unsigned)
- Keywords: `uint8` / `int8` to `uint256` / `int256` in step of 8.
- `uint` / `int` are alias for `uint256` / `int256`.
- Operators as usual:
  - Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>`
  - Arithmetic operators: `+`, `-`, `*`, `/`, `%`, `**`
  - Bitwise operators: `&`, `|`, `^`
  - Shift operators: `>>`, `<<`
- Range: $2^b - 1$ where $b \in \{ 8, 16, 24, 32, \ldots, 256 \}$
- Division always results in an integer and round towards zero (5 / 2 = 2).
- No floats!

Types: address

```
contract Address {
    address owner;
    address payable anotherAddress;
}
```

Address type holds an Ethereum address (20 byte value).
The “payable” keyword enables to send Ether to the address (you cannot send to plain addresses).

Types: fixed-size byte arrays

```
contract ByteArrays {
    bytes32 y = 0xa5b9...;
    // y.length == 32
}
```

- `bytes1`, `bytes2`, `bytes3`, …, `bytes32`
- `byte` is alias for `byte1`
- `length`: fixed length of the byte array. You cannot change the length of a fixed byte array.

Types: Enum

```
contract Purchase {
    enum State { Created, Locked, Inactive }
}
```

Example Enum

```
pragma solidity ^0.8.1;

contract Enum {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    function getChoice() public view returns (ActionChoices) {
        return choice;
    }
}
```

Reference types

Types: arrays, static and dynamic

```
contract Arrays {
      uint256[2] x;
      uint8[] y;
      bytes z;
      string name;
      // 2D: dynamic rows, 2 columns!
      uint [2][] flags;

      function create () public {
            uint[] memory a = new uint[](7);
            flags.push([0, 1]);
      }
}
```

- The **notation** of declaring **2D** arrays is **reversed** when compared to **other languages**!
  - **Declaration**: `uint[columns][rows] z;`
  - **Access**: `z[row][column]`
- `bytes` and `string` are **special** arrays.
- `bytes` is similar to `byte`[] but is **cheaper** (gas).
- `string` is **UTF-8-encoded**.
- Members:
  - push: push an element at the end of array.
  - length: return or set the size of array.
- `string` does **not** have **length** member.
- **Allocate** memory **arrays** by using the **keyword** `new`. The size of memory arrays has to be known at compilation (in this case 7). You **cannot** resize a memory array.

Types: Mappings

```
contract Mappings {

    mapping(address => uint256) balances;

}
```



Types: Struct

```
contract Vote {

    struct Voter {

        bool voted;

        address voter;

        uint vote;

    }

}
```

- A struct **cannot** contain a struct of its **own type** (the size of the struct has to be **finite**).
- A struct **can** contain **mappings**.

Example Structs

```
pragma solidity ^0.4.24;

contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

```
contract CrowdFunding {
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping(uint => Funder) funders;
    }
}
```

Visibility

Visibility

- **public**: Public functions can be called from other contracts, internally, and from personal accounts. For public state variables a getter function is automatically created.
- **external**: External functions cannot be called internally. Variables cannot be declared as external.
- **internal**: Internal functions and variables can be called only internally. Contracts that inherit another contract can access the parent’s internal variables and functions.
- **private**: Private functions and variables can be called only by the contract in which they are defined and not by a derived contract.

Solidity: Functions

- Can return multiple values
- Access
  - **Public**: Accessed by **anyone**
  - **Private**: Accessed **only** from the **contract**
  - **Internal**: Accessed **only internally**
  - **External**: Accessed **only externally**
- Declarations
  - **View**: They promise **not** to **modify** the **state**
  - **Pure**: They promise **not** to **read** from or **modify** the **state**.
  - **Payable**: Must be used to **accept Ether**

Remember that on-chain data is public regardless of access declaration!

Solidity: Inheritance

- Multiple inheritance
- **One contract** is created on the **blockchain** for all derived contracts: codes concatenate
- The general **inheritance** system is very **similar** to **Python's**

Solidity: Inheritance

- Use ***is*** keyword to **extend** a contract
- **Derived** contracts: **access** all non-private members, internal functions and state variables
- **Abstract** contracts can be used as **interfaces**
- **Functions** can be **overridden**
- **Interfaces**: functions are not implemented

Solidity: Inheritance

```
pragma solidity ^0.8.1;

interface Regulator {
   function checkValue(uint amount)  external returns (bool);
   function loan() external returns (bool);
}
contract LocalBank is Bank(10) {
string private name;
uint private age;

function setName(string newName) public {
name = newName;
}
function getName() public view returns (string) {
return name;
}
function setAge(uint newAge) public {
age = newAge;
}
function getAge() public view returns (uint) {
return age;
}
}
```

```
contract Bank is Regulator {
uint private value;
constructor (uint amount) {
value = amount;
}
function deposit(uint amount) public {
value += amount;
}
function withdraw(uint amount) public {
if (checkValue(amount)) {
value -= amount;
}
}
function balance() public view returns (uint) {
return value;
}
function checkValue(uint amount) public view returns (bool) {
return value >= amount;
}
function loan() public view returns (bool) {
return value > 0;
}
}
```

Solidity: Inheritance

```
pragma solidity ^0.8.1;

contract Jedi {
function computeForce() internal pure returns (uint){
return 50;
}

function getExtraForce() private pure returns (uint) {
return 100;
}
}


contract Ewok {
Jedi j = new Jedi();
uint force = j.computeForce(); // error private method
}
```

```
pragma solidity ^0.8.1;

contract Human is Jedi {
uint age = 70;
string name = “Luke”;
string lastName = “Skywalker”;
bool isMaster = false;
uint force = 0;

function setMaster(bool _master) external {
isMaster = _master;
force = computeForce(); // internal call
force = force + getExtraForce(); // error private method
}

function getJedi() public view returns (uint, string, string, bool){
return (age, name, lastName, isMaster) // multi-values
}

}
```

Data location

Data location: areas

- Every reference type (array, struct, mapping) has a data location.
- Two main data locations: **storage** and **memory**.
- **Calldata**: special location for function’s arguments.
- As of Solidity version **0.5.0** you must **always declare** the data **location** of reference types inside functions’ body, arguments and returned values.

Data location: areas

- Storage:
  - Persistent
  - All state variables are saved to storage
- Memory:
  - Non-persistent
  - Can be used for function variables or arguments
- Calldata:
  - Non-modifiable (read-only)
  - Function arguments
  - Cheaper than memory
  - Used for dynamic params of an *external* function

Data location: assignment copy/reference rules

- Assignment of the form “variable1 = variable2”
- Assignment by copy
  - storage <-> memory
  - all other assignments to storage (e.g., to state variables)
- Assignment by reference
  - memory <-> memory
  - storage -> local storage variable

https://docs.soliditylang.org/en/v0.8.21/types.html#data-location-and-assignment-behavior

Events,
Modifiers,
and Global variables

Solidity: events

- EVM logging mechanism
- Arguments are stored in the transaction log
- An alternative to store data cheaply
- Client software can create “listeners” to events (eg. in Python/JS)

Solidity: events

Contract - Solidity

```
pragma solidity ^0.8.1;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable
{
        emit Deposit(msg.sender, _id,
msg.value);
    }
}
```

Client - Javascript

```
var abi = /* abi as generated by compiler */;
var web3 = /* http/ws connection */;
var contractObject = web3.eth.contract(abi);
var contractInstance = contractObject.at(
    "0x1234...ab67" /* address */
);

var event = contractInstance.Deposit();

// watch for changes
event.watch(function(error, result){
    if (!error)
        console.log(result);
    ....
    /* use result to access event data */
});
```

Solidity: Modifiers

Declare modifier

```
pragma solidity ^0.8.1;

contract owned {
    address owner;

    constructor() { owner = msg.sender; }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}
```

Apply modifier

```
contract mortal is owned {
    function withdraw() public onlyOwner {
        uint256 balance = address(this).balance;
        (bool success, ) = payable(msg.sender)
            .call{value: balance}("");
    }
}
```

Solidity: units and globally available variables

- Ether Units
  - A literal number can take a suffix of wei, finney, szabo or ether (2 ether == 2000 finney evaluates to true)
- Time Units
  - Suffixes like seconds, minutes, hours, days, and weeks (1 hours == 60 minutes)

Solidity: units and globally available variables

- Block and Transaction Properties
  - block.blockhash
  - Block.coinbase
  - block.timestamp
  - msg.data
  - msg.value
  - msg.sender
  - tx.origin

Solidity: units and globally available variables

- Error Handling
  - via error objects (see https://docs.soliditylang.org/en/v0.8.21/control-structures.html)
  - assert
  - require
  - revert
- Mathematical and Cryptographic Functions
  - addmod, mulmod
  - Keccak256 (SHA-3), sha256, ripemd160

Solidity: units and globally available variables

- Address Related
  - <address>.balance
  - <address>.transfer
  - <address>.send
  - <address>.call, <address>.delegatecall
- Contract Related
  - this

Sending Ether and Contract interactions

Fallback functions

```
contract Fallback {

    receive() external {

        …

    }

    fallback() external {

        …

    }

}
```

- No arguments (`msg.*` is accessible, contains all data about incoming transaction, incl. sender and value).
- No returned values.
- Mandatory visibility: external.
- `Receive` is executed if no data (transaction field) is supplied. It `is implicitly payable`.
- `Fallback` is executed if the function that a user tries to call does not exist. May or may not be `payable`.
- In the absence of a fallback function a contract cannot receive Ether and an exception is thrown.
- Should be simple - without consuming too much gas.

Send ether

Function	Gas forwarded	Error handling	Notes
`transfer`	2300	throws error on failure	• **Safe** against re-entrancy* • **Fails** if recipient contract's fallback function consumes >2300 gas
`send`	2300	`false` on failure	• **Safe** against re-entrancy* • **Fails** if recipient contract's fallback function consumes >2300 gas
`call`	all remaining gas	`false` on failure	• **Not safe** against re-entrancy*

(* described in later slides)

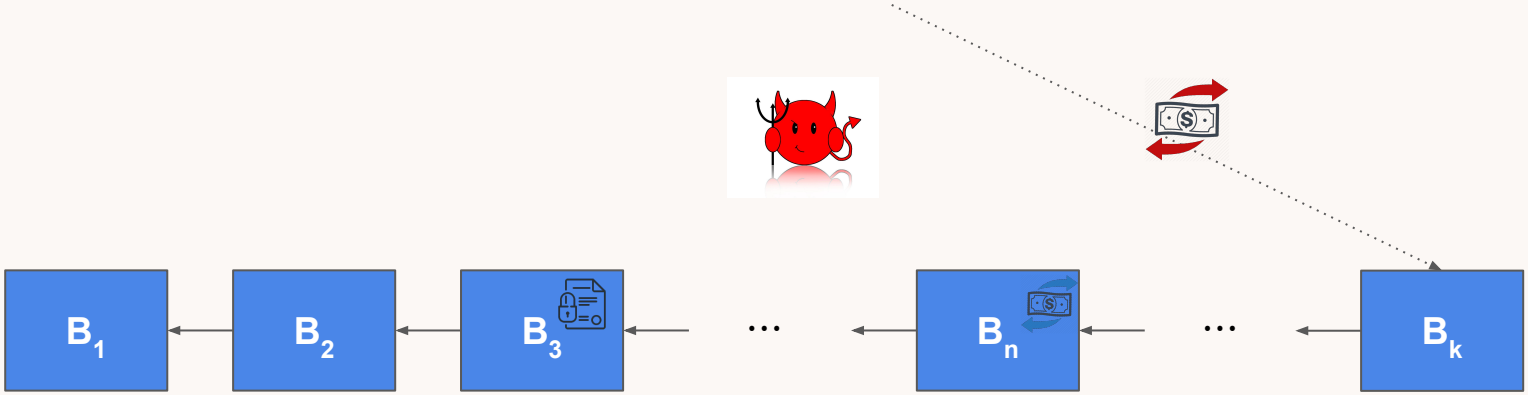
Interacting with other contracts

```
contract Planet {
     string private name;
     constructor (string memory _name){ name = _name; }
     function getName() public returns(string memory) { return name; }
}

contract Universe {
     address[] planets;
     event NewPlanet(address planet, string name);

     function createNewPlanet(string memory name) public {
          Planet p = new Planet(name);
          planets.push(address(p));
          emit NewPlanet(address(p), p.getName());
     }
}
```

Smart Contract Attacks and Mitigations

Smart Contract Attack - Overview

- An adversary exploits a hazard in the contract, by sending a transaction that somehow breaks its functionality
- Known attacks include: Denial-of-Service, Griefing, Re-entrancy, Front-running, Randomness Manipulation



Attacks: Denial-of-Service & Griefing

DoS: Unbounded operation

```
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.send(investors[i].dividendAmount));
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Unbounded operation

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.send(investors[i].dividendAmount));
}
```

- Operation requires more gas as array becomes larger
- After some point, it might be impossible (beyond gas limits) to execute it

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Griefing

```
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.send(investors[i].dividendAmount));
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Griefing

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.send(investors[i].dividendAmount));
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

Error handling

- If a send/transfer **call fails**, the contract might get **stuck**
- It is **possible to force** a call to fail (e.g., by getting the victim contract to send to another contract that fails)
- **Errors** need to be **handled**, instead of simply reverting

Design pattern: pull over push

```
function bid() payable {
      require(msg.value >= highestBid);

      if (highestBidder != address(0)) {
            highestBidder.transfer(highestBid);
      }

      highestBidder = msg.sender;
      highestBid = msg.value;
}
```

https://consensys.github.io/smart-contract-best-practices

Design pattern: pull over push

```
// BAD DESIGN (PUSH)

function bid() payable {
      require(msg.value >= highestBid);

      if (highestBidder != address(0)) {
            highestBidder.transfer(highestBid);
      }

      highestBidder = msg.sender;
      highestBid = msg.value;
}
```

https://consensys.github.io/smart-contract-best-practices

Design pattern: pull over push

```
// BAD DESIGN (PUSH)

function bid() payable {
      require(msg.value >= highestBid);

      if (highestBidder != address(0)) {
            highestBidder.transfer(highestBid);
      }

      highestBidder = msg.sender;
      highestBid = msg.value;
}
```

```
// GOOD DESIGN (PULL)

function bid() payable external {
      require(msg.value >= highestBid);

      if (highestBidder != address(0)) {
            refunds[highestBidder] += highestBid;
      }

      highestBidder = msg.sender;
      highestBid = msg.value;
}

function withdrawRefund() external {
      uint refund = refunds[msg.sender];
      refunds[msg.sender] = 0;
      msg.sender.transfer(refund);
}
```

https://consensys.github.io/smart-contract-best-practices

Design pattern: pull over push

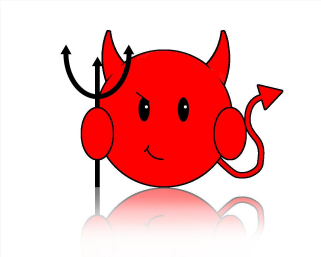
- **Do not transfer** ETH to users (push); let them **withdraw** (pull) their funds.
- **Isolates** each **external call** into its own transaction.
- **Avoids** multiple `send()` calls in a single transaction.
- **Reduces** problems with **gas limits**.
- Possibly increases **gas fairness** (each user pays the gas for receiving their own funds).
- **Tradeoff** between **security** and **user experience**.

Attack: Reentrancy

Reentrancy



Contract A

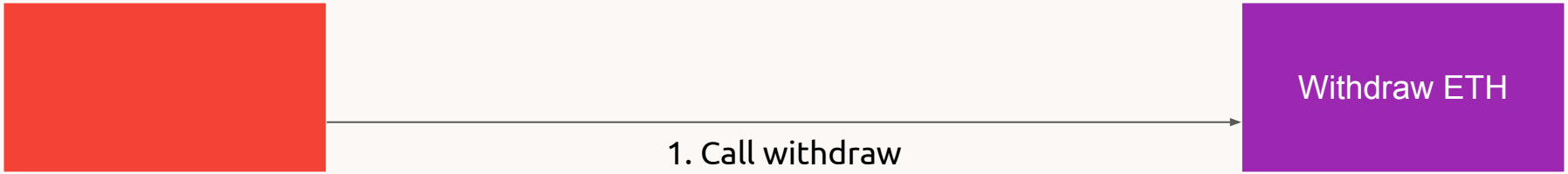




Contract B



Reentrancy



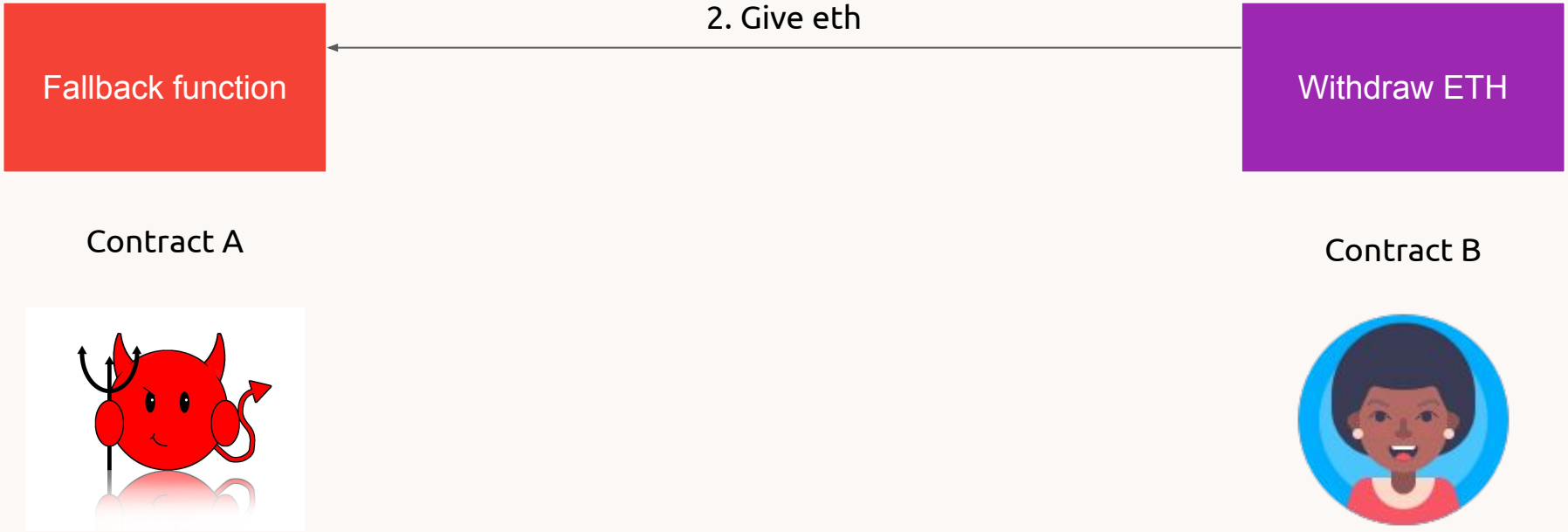
Contract A



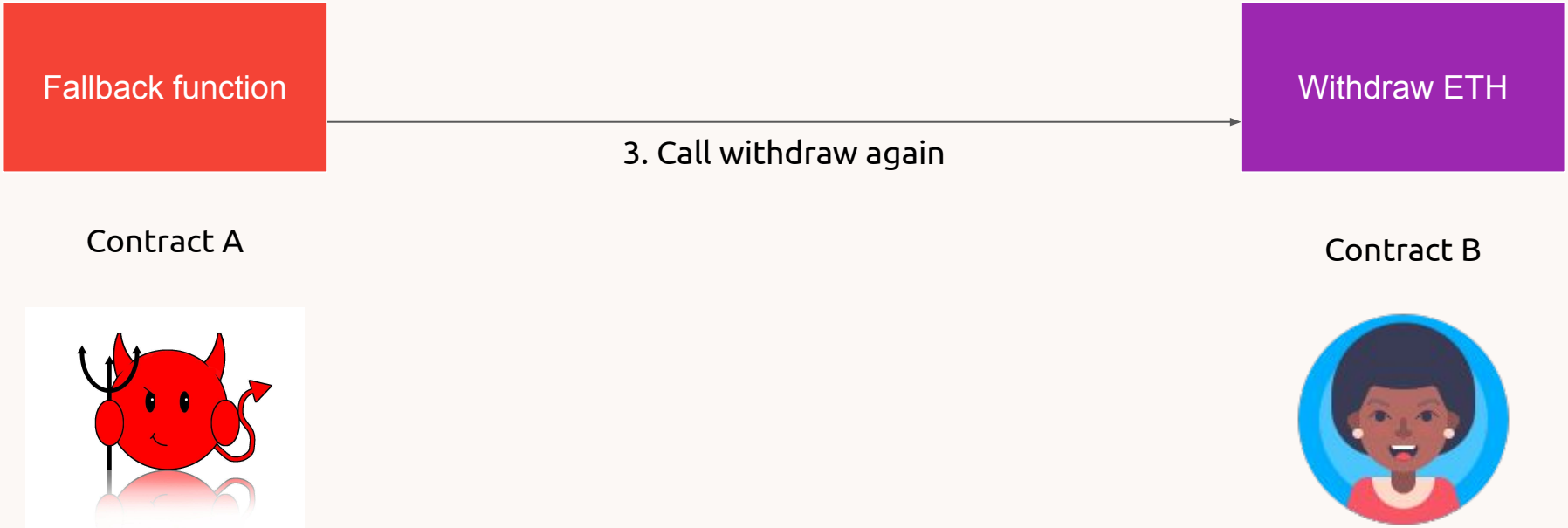
Contract B



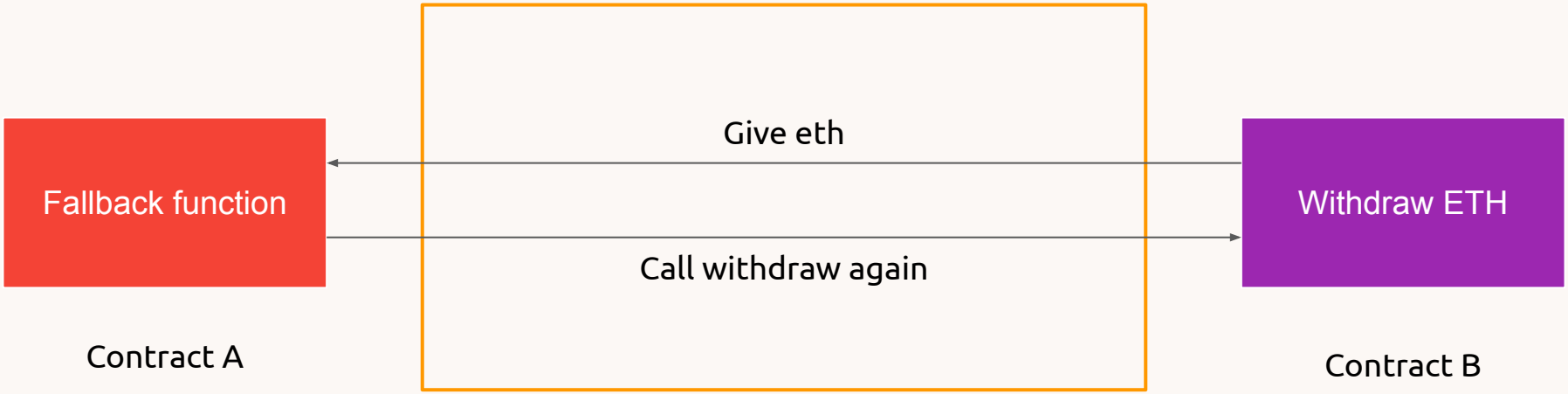
Reentrancy


Fallback function
2. Give eth
Withdraw ETH
Contract A
Contract B

Reentrancy


Fallback function
Withdraw ETH
3. Call withdraw again
Contract A
Contract B

Reentrancy


Fallback function
Give eth
Withdraw ETH
Call withdraw again
Contract A
Contract B
Loop of function calls





Reentrancy example

```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

      require(msg.sender.call.value(amountToWithdraw)());

      userBalances[msg.sender] = 0;

}
```



Reentrancy example

```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

      require(msg.sender.call.value(amountToWithdraw)());

      userBalances[msg.sender] = 0;

}
```



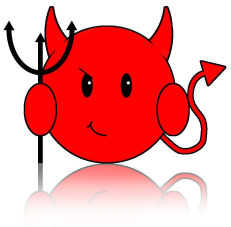
Reentrancy example

Begin attack by sending eth to attacker contract

```
// INSECURE
mapping (address => uint) private userBalances;
function withdrawBalance() public {
      uint amountToWithdraw = userBalances[msg.sender];
      require(msg.sender.call.value(amountToWithdraw)());
      userBalances[msg.sender] = 0;
}
```

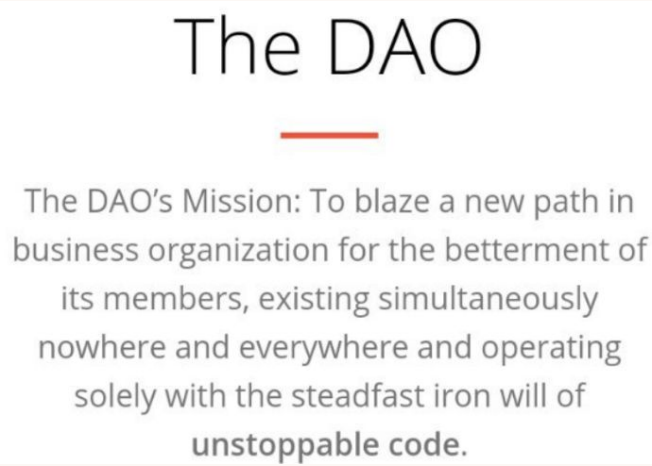
```
function receive() payable {
  if (victimContract.balance >= msg.value) {
        victim.withdrawBalance();
    }
}
```





Re-entrancy in the wild: The DAO

- The DAO (distributed autonomous organization*)
  - Designed by slock.it in 2016
  - Purpose: Create a population of stakeholders
  - Stake (in the form of DAO tokens) enables them to participate in decision making
  - Decision-making to choose which proposals to fund



*According to the SEC, neither "distributed" nor "autonomous": https://www.sec.gov/news/press-release/2017-131

THE DAO IS AUTONOMOUS.

1071.36 M
DAO TOKENS CREATED

10.73 M
TOTAL ETH

116.81 M
USD EQUIVALENT



1.10
CURRENT RATE
ETH / 100 DAO TOKENS

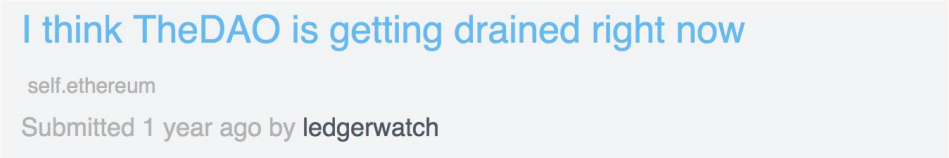
15 hours
NEXT PRICE PHASE

11 days
LEFT
ENDS 28 MAY 09:00 GMT

~150 million USD in ~ 1 month

The DAO Attack (2016)

- 12 June: The reentrancy bug is identified (but stakeholders are “reassured”)
- 17 June: Attacker exploits it draining ~$50Million at the time of the attack
  - Active deliberation about changing the system to nullify the attack
- 15 July: Ethereum Classic manifesto
- 19 July: “Hard Fork” neutralizes attacker’s smart contract



Reentrancy: solutions

```
// SECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

      userBalances[msg.sender] = 0;

      msg.transfer(amountToWithdraw);

}
```

- Finish all internal work (state changes) and then call external functions
- Checks-Effects-Interactions Pattern
- Mutexes
- Pull over push pattern

Design pattern: Checks-Effects-Interactions

1. Perform **checks** e.g., on inputs, sender, value, arguments etc
2. Enforce **effects** and update the **state** accordingly
3. **Interact** with other accounts via external calls or send/transfer

Solidity specific hazards

Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance` (when contract relies on it)

```
contract Vulnerable {
    function receive() external {
        revert();
    }

    function fallback() external {
        revert();
    }

    function somethingGood() {
        require(this.balance == 0);
        // something good
    }
}
```

https://github.com/demining/Solidity-Forcibly-Send-Ether-Vulnerability

Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance` (when contract relies on it)
- How can you **send ether** to a contract **without** firing contact's **fallback** function ?

Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance` `(when contract relies on it)`
- How can you **send ether** to a contract **without** firing contact's **fallback** function ?
  - Contract's address = hash(sender address, nonce): anyone can **calculate** a contract's address **before** it is **created** and send ether to it
  - Set contract's address as **recipient of block rewards**

Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance` (when contract relies on it)
- How can you **send ether** to a contract **without** firing contact's **fallback** function ?
  - Contract's address = hash(sender address, nonce): anyone can **calculate** a contract's address **before** it is **created** and send ether to it
  - Set contract's address as **recipient of block rewards**
- Lesson: **Avoid** strict equality checks with the contract's balance

Delegate call

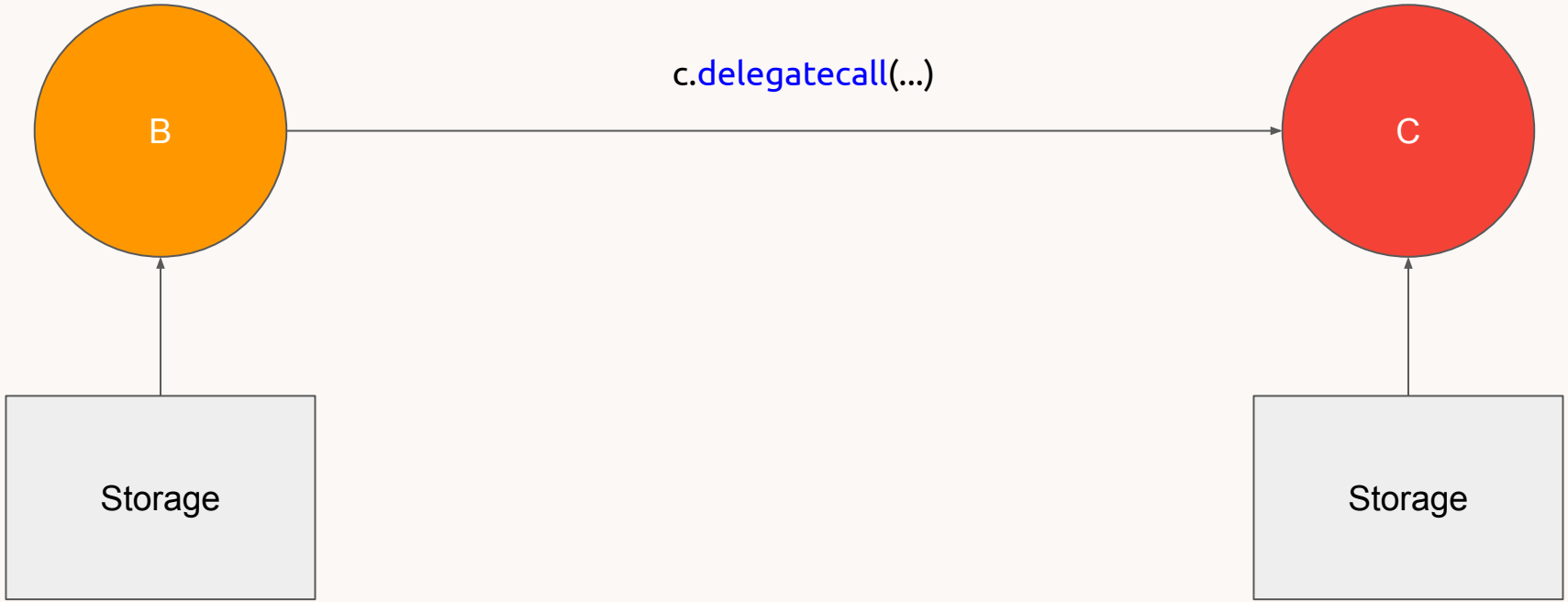
- Special function that forwards calls from one contract to another
- msg.sender and msg.value keep their original values
- Storage, current address and balance still refer to the calling contract, only code is taken from the called address
- Main uses: libraries and contract upgrades

- But…

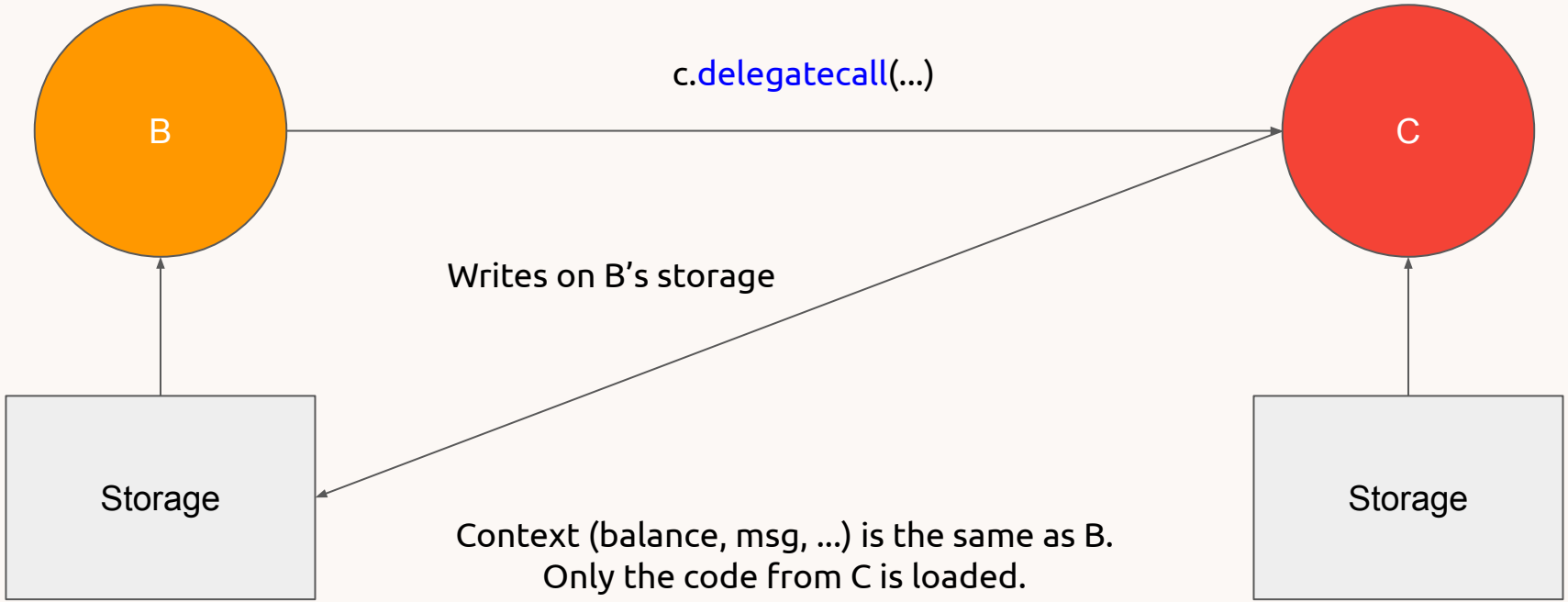
Delegate call



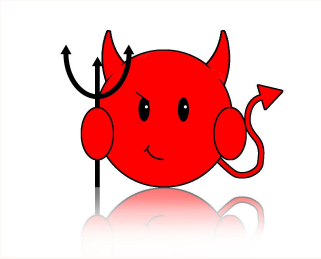
Delegate call



Delegate call



Delegate call

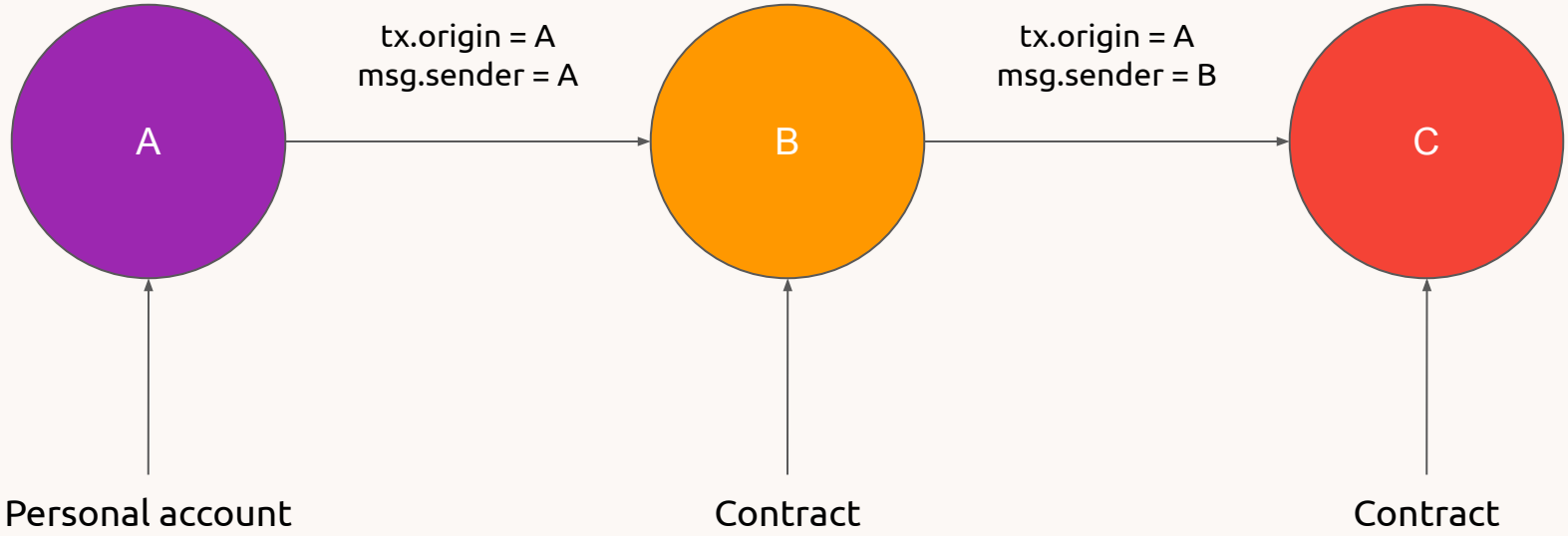


```
// INSECURE
address public owner;

Library library =

function() public {
      require(library.delegatecall(msg.data));
}
```



```
  address public owner;

  constructor (address _owner) {
    owner = _owner;
  }

  function pwn() public {
    owner = msg.sender;
  }
}
```

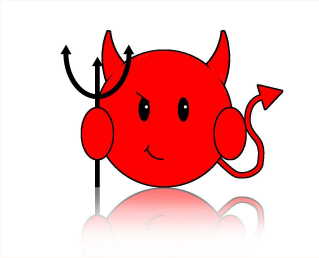
Use of tx.origin



Use of tx.origin

```
// INSECURE
contract Bank {

    address owner;

    constructor() {
        owner = msg.sender;
    }

    function sendTo(address payable receiver, uint amount)
public {
        require(tx.origin == owner);
        receiver.call.value(amount)();
    }

}
```

Use of tx.origin



```
// INSECURE
contract Bank {

    address owner;

    constructor() {
        owner = msg.sender;
    }

    function sendTo(address payable receiver, uint amount)
public {
        require(tx.origin == owner);
        receiver.call.value(amount)();
    }

}
```

```
function receive() external payable {

    victim.sendTo(attacker,msg.sender.balance);

}
```

Keep fallback function simple

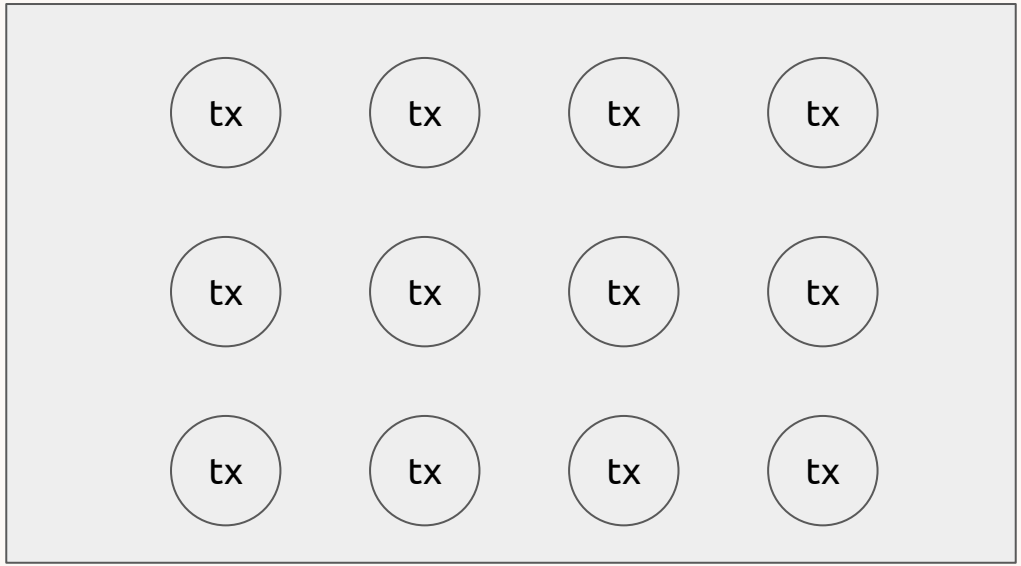
```
// BAD

function receive() payable {
      balances[msg.sender] += msg.value;
}
```

```
// GOOD

function deposit() payable external {
      balances[msg.sender] += msg.value;
}

function receive() payable {
      require(msg.data.length == 0);
      emit LogDepositReceived(msg.sender);
}
```

https://consensys.github.io/smart-contract-best-practices

Attack:
Front-running

Front-Running



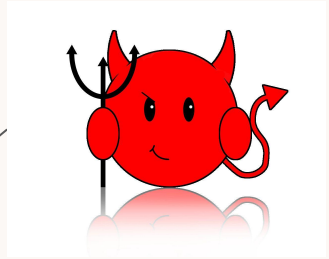
Miner: sortByGasPrice(txs, ‘desc’)

Front-Running: user



50 GWei

tx

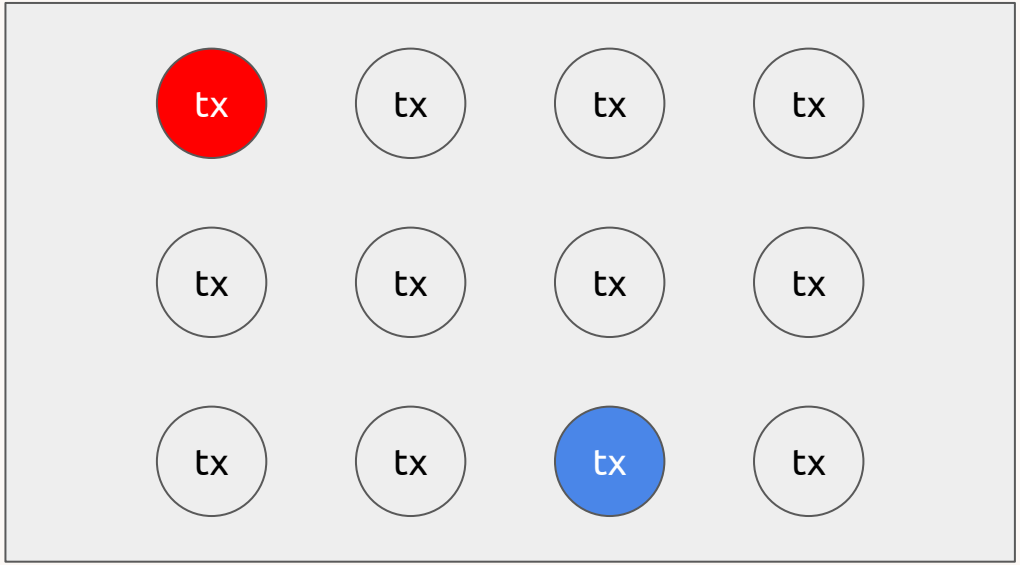


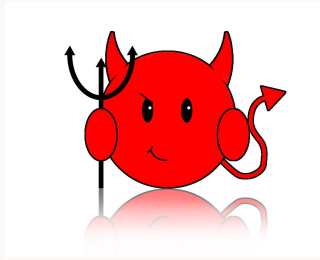
2 GWei

tx



Front-Running: user

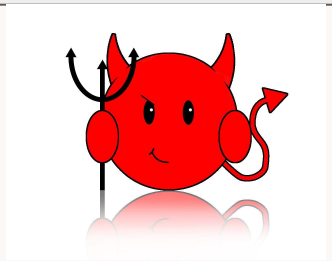






Front-Running: miner





1 GWei

tx

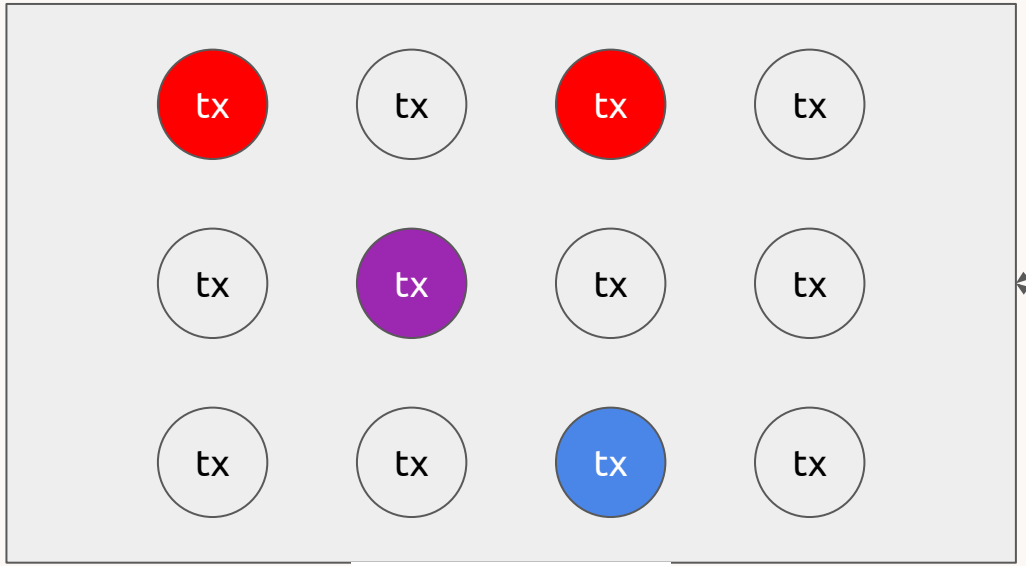


2 GWei

tx



Front-Running: miner





1 GWei

tx



2 GWei

tx



Front-Running: example

```
// INSECURE

function registerName(bytes32 name) public {

     names[name] = msg.sender;

}
```

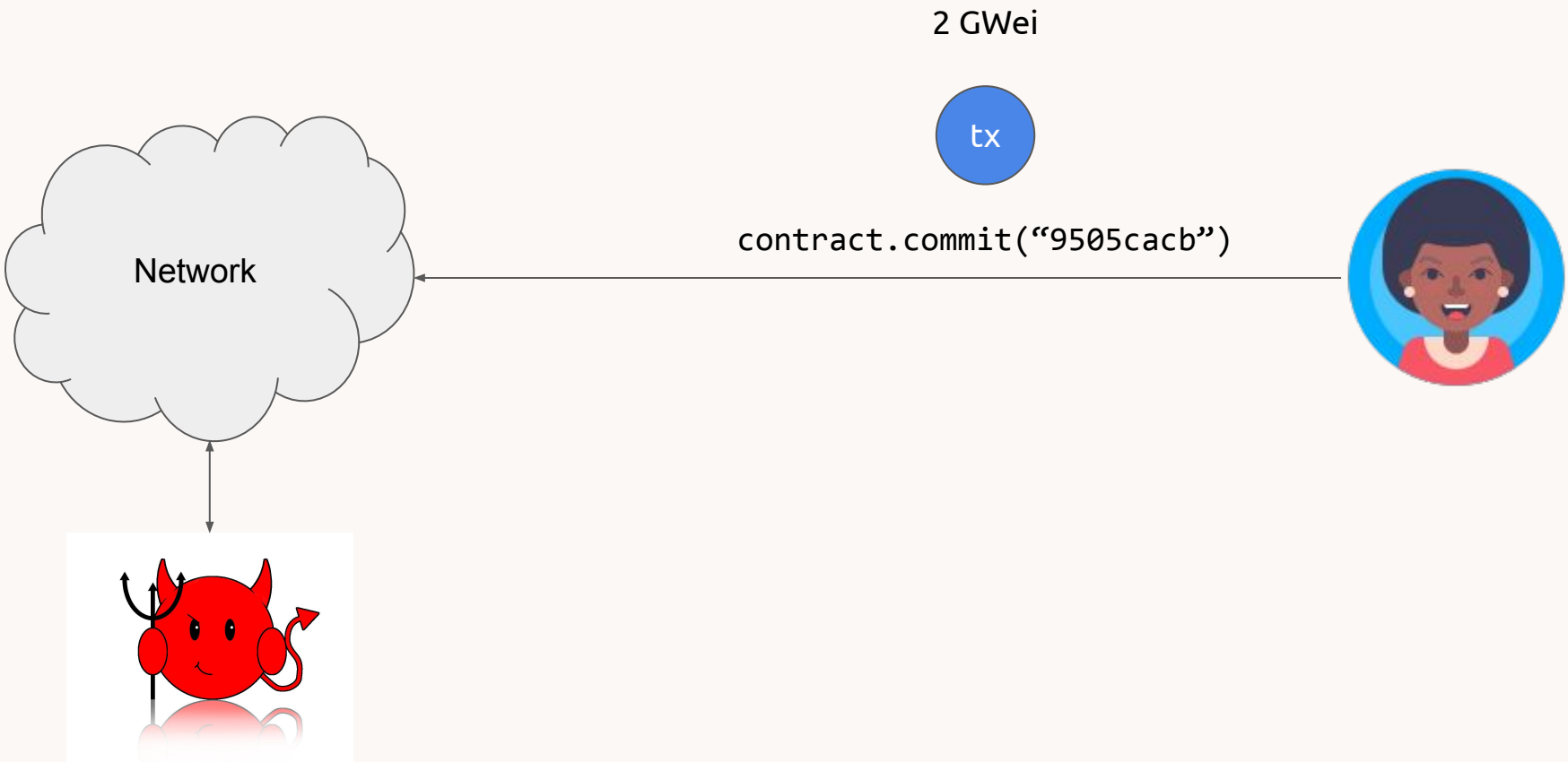
Front-Running solution: commitment scheme

- Employ a cryptographic **commitment scheme**
- Implementation
  - commit: c = hash(<value, nonce>) *(Note: nonce space should be large!)*
  - reveal: v = <value', nonce'>
  - verify: c == hash(v)
- Properties
  - **Binding**: a commitment can be opened only to its committed value
  - **Hiding**: a commitment reveals no information about its committed value

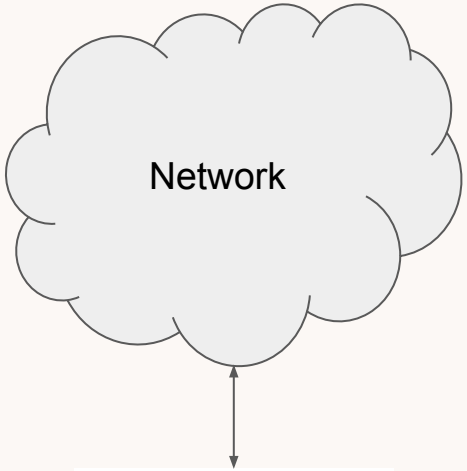
Front-Running solution example

```
// INSECURE

function registerName(bytes32 name) public {

     names[name] = msg.sender;

}


// MORE SECURE, BUT…

function registerName(bytes32 name, bytes32 nonce) public {

     require(commitments[makeCommitment(name, nonce)] == msg.sender, “Not found!”);

     names[name] = msg.sender;

}
```

Front-Running: example



Front-Running: example





2 GWei

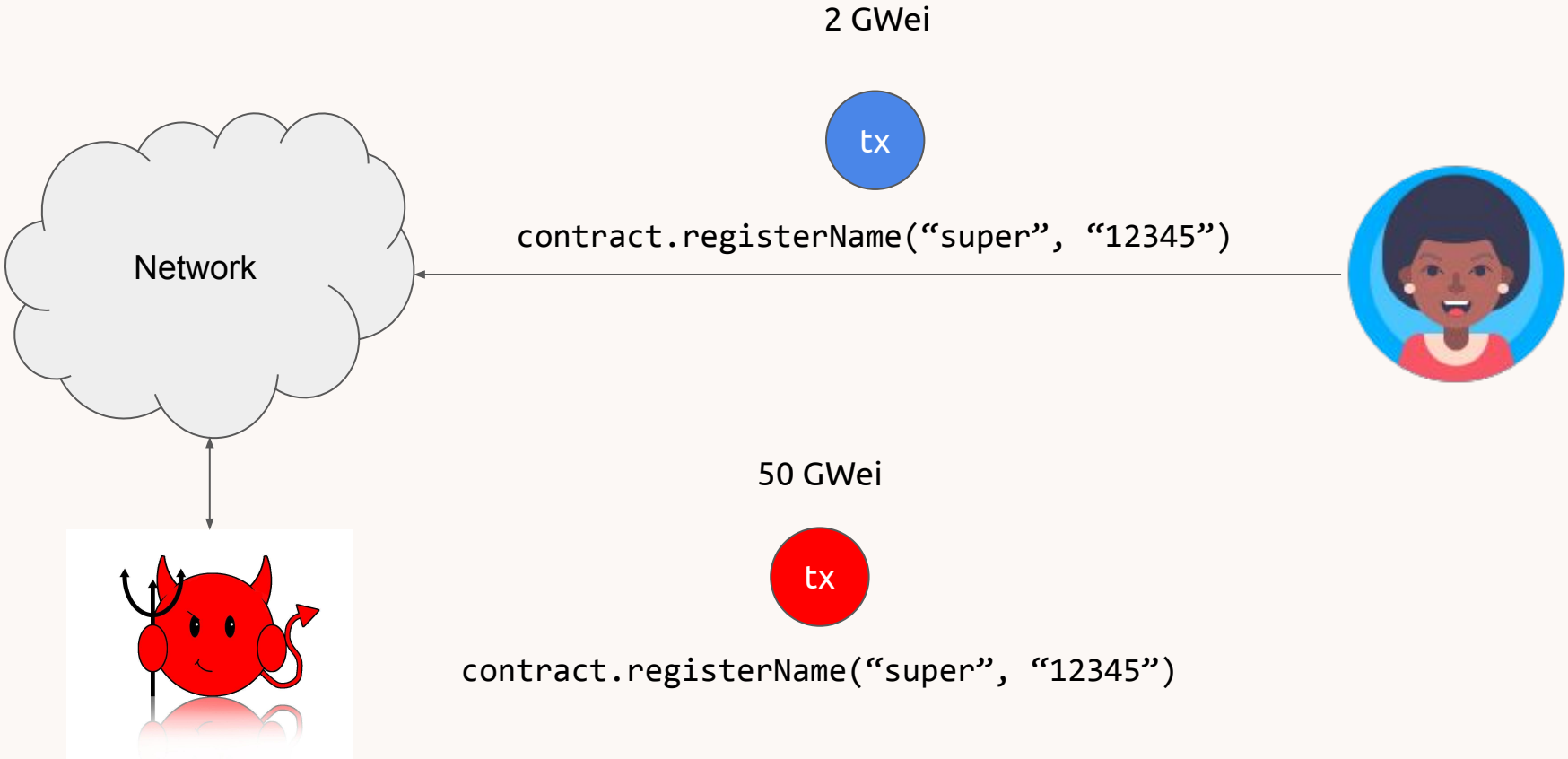
tx

```
contract.commit(“9505cacb”)
```

Front-Running: example



Front-Running: example



Front-Running solution

- Employ a cryptographic **commitment scheme**
- **Keep track** of committed values
  - Prevent a user from posting a commitment already posted by another user
- Possible **DoS** and **forced gas cost**
  - Attacker can front-run a user’s commit operation and post the commitment as their own
  - User is forced to spend extra gas for new tx that posts new commitment
  - Attacker can continue front-running until they run out of money (to pay gas)

Generating Randomness

Randomness: sources (?)

- block.number
- block.timestamp
- block.hash
- block.difficulty
- block.coinbase
- block.gasLimit
- now
- msg.sender



Randomness: sources (?)

- block.number
- block.time
- block.has
- block.diff

- lock.coinbase
- k.gasLimit
- n
- ender



They can be manipulated by a malicious miner.
They are shared within the same block to all users.

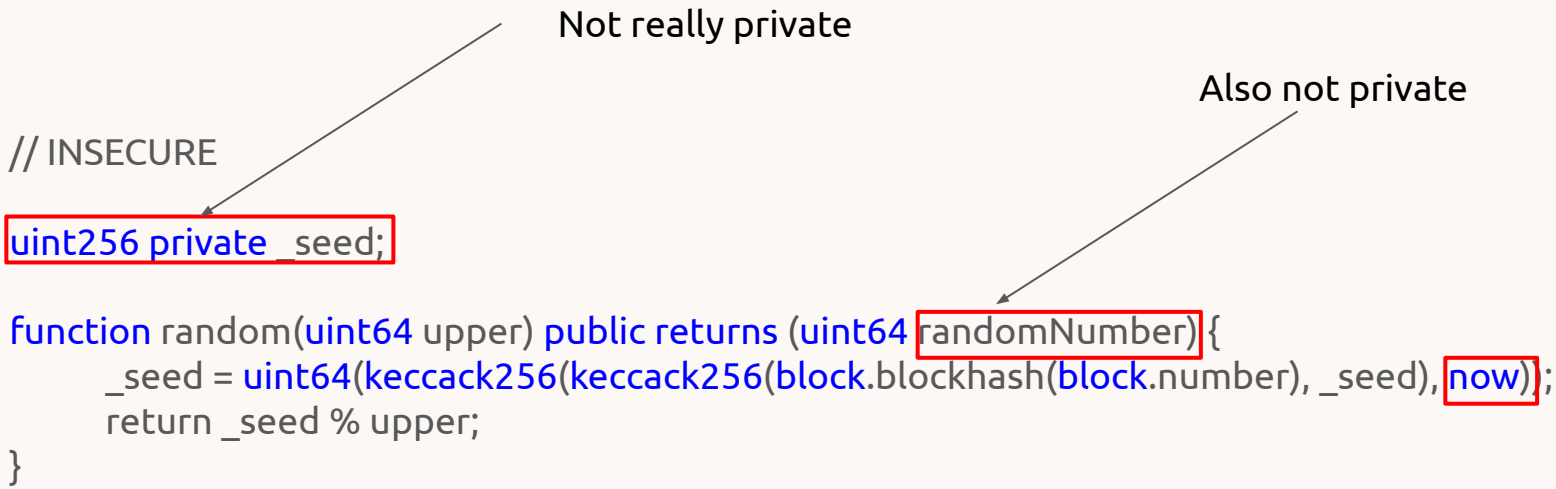
Randomness

```
// INSECURE
bool won = (block.number % 2) == 0;
```

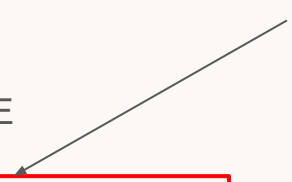
```
// INSECURE
uint random = uint(keccak256(block.timestamp)) % 2;
```

```
// INSECURE
address seed1 = contestants[uint(block.coinbase) % totalTickets].addr;
address seed2 = contestants[uint(msg.sender) % totalTickets].addr;
uint seed3 = block.difficulty;
bytes32 randHash = keccak256(seed1, seed2, seed3);
uint winningNumber = uint(randHash) % totalTickets;
address winningAddress = contestants[winningNumber].addr;
```

Randomness: blockhash



Randomness: blockhash




```
// INSECURE

uint256 constant private FACTOR =
115792089237316195423570985008687907853269984665640564039457584007913129639 9;

function rand(uint max) pure private returns (uint256 result) {
        uint256 factor = FACTOR * 100 / max;
        uint256 lastBlockNumber = block.number - 1;
        uint256 hashVal = uint256(block.blockhash(lastBlockNumber));
        return uint256((uint256(hashVal) / factor)) % max;
}
```

Randomness: intra-transaction information leak

```
if (replicatedVictimConditionOutcome() == favorable)
    victim.tryMyLuck();
```

https://media.dedaub.com/bad-randomness-is-even-dicier-than-you-think-7fa2c6e0c2cd

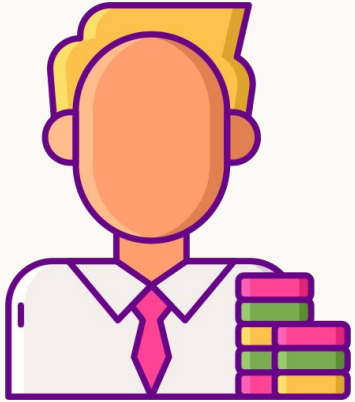
Sources of randomness

- **Block information** can be **manipulated by miner**
- Block information **shared** by all users in the same block
- In Ethereum, **all data** posted on the chain are **visible**
- “private” vars are only private w.r.t. object-oriented programming **visibility**
- If same-block txs share randomness source, attacker can **check** whether conditions are favorable **before** acting

What about future blocks ?



1. Player makes a bet and the casino stores the block.number of the transaction

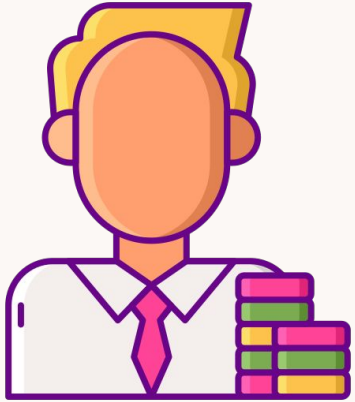


Casino

Player



2. A few blocks later, player requests from the casino to announce the winning number

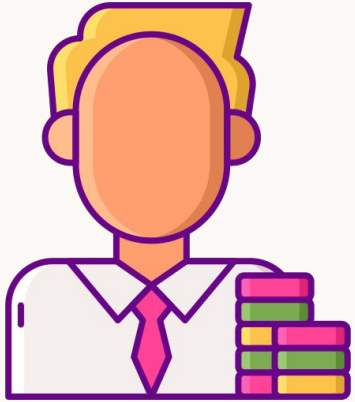


Casino

Player



3. Casino uses, as a source of randomness, the hash of a block produced after the bet is placed



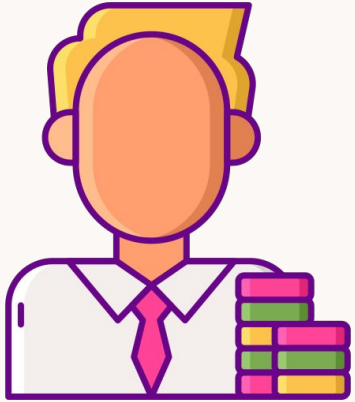
Casino

Player

Validate block.number age!

3. Casino uses, as a source of randomness, the hash of a block produced <u>after</u> the bet is placed





Casino

Player

Is the hash of a future block a good source of randomness (against a malicious miner)?

- A contract can access the hashes of only the **last 256** blocks; blockhash older than that defaults to 0
- Always **validate** block's age
- With some probability (how high?), a malicious **miner** will **create the specific future block**
- In PoS, the **proposer** of a future block might be **known beforehand**
- A **miner** can keep newly-mined **blocks hidden**, until they mine a favorable one

Randomness: towards safer random number gen

- Commitment schemes
  - Prover commits to a message *m* by publishing *h = H(m)* (H is a hash function)
  - After some time, prover reveals message *m*
  - Verifier wants to be sure that the originally committed message is the revealed one
    - Verifier checks that: *h == H(m)*
  - Binding property:
    - Collision resistance: it should be infeasible to find m' s.t. H(m) == H(m')
  - Hiding property:
    - Honest prover wants no information about *m* to be retrievable from *H(m)*
    - H needs to behave as a random oracle
    - *m* should be unpredictable; if domain is small, use salt

Randomness: towards safer random number gen

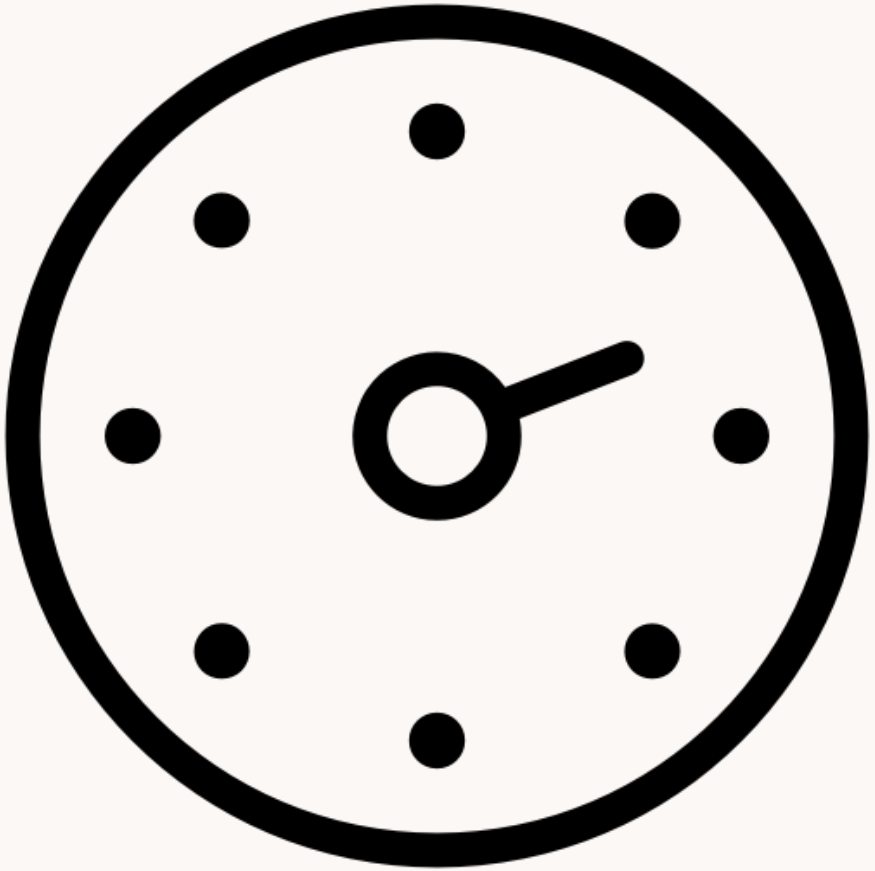
- 2-party coin flipping via commitments
- Example:
  - Casino and player each commit to a random value
  - Casino and player reveal their values
  - Casino XORs the random values to produce a seed
    - the seed can also be combined with the hash of a future block
  - If *either* casino *or* player honest, then the seed is random (why?)
  - What happens in the case of aborts?

On-chain data is public

- Applications (games, auctions, etc) required **data** to be **private** up until some point in time
- Every data that is published on-chain is **visible** by everyone
- Best strategy: **commitment schemes**
- Watch out for front-running!

Overflow/Underflow

$2^{256} - 1$ 0



2^{254}

2^{255}

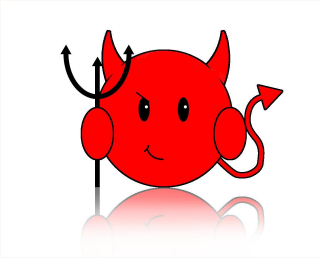
Integer Overflow and Underflow

```
// INSECURE
function withdraw(uint256 _value) {
     require(balanceOf[msg.sender] >= _value);
     msg.sender.call.value(_value)();
     balanceOf[msg.sender] -= _value;
}
```

Integer Overflow and Underflow

```
// INSECURE
function withdraw(uint256 _value) {
     require(balanceOf[msg.sender] >= _value);
     msg.sender.call.value(_value)();
     balanceOf[msg.sender] -= _value;
}
```

Integer Overflow and Underflow



```
// INSECURE
function withdraw(uint256 _value) {
      require(balanceOf[msg.sender] >= _value);
      msg.sender.call.value(_value)();
      balanceOf[msg.sender] -= _value;
}
function donate(uint256 _value) public payable {
      require(msg.value == value);
      balanceOf[msg.sender] += _value;
}
```

```
function attack() {
      performAttack = true;
      victim.donate(1);
      victim.withdraw(1);
}
function() {
      if (performAttack) {
            performAttack = false;
            victim.withdraw(1);
      }
}
```

Integer Overflow and Underflow: solutions

Solidity 0.8+ protects natively against over/underflows.

For older versions, use OpenZeppelin's SafeMath library.

```
// OpenZeppelin: SafeMath.sol

function add(uint256 a, uint256 b) internal pure returns
(uint256) {
      uint256 c = a + b;
      require(c >= a, "SafeMath: addition overflow");

      return c;
}

function sub(uint256 a, uint256 b) internal pure returns
(uint256) {
      require(b <= a, "SafeMath: subtraction overflow");
      uint256 c = a - b;

      return c;
}
```

(Gas) Fairness

Gas Fairness

Crowdfunding Contract #1

R sets a threshold

Contract collects contributions

When balance exceeds threshold, it sends funds to R and returns any surplus to contributors.

VS.

Crowdfunding Contract #2

R sets a threshold

Contract collects contributions

When balance exceeds threshold, it allows R to withdraw the threshold and return any surplus to contributors

VS.

Crowdfunding Contract #3

R sets a threshold

Contract collects contributions

When balance exceeds threshold, it allows R and contributors to withdraw the threshold and surplus respectively

Funding paid by last contributor

R pays for funding

R and contributors pay for funding

A (horribly insecure) ✊ ✋ ✌ contract

```
3  pragma solidity >=0.7.0 <0.9.0;
4
5  contract RockPaperScissors { // Winner gets 1 ETH
6      struct round {
7          address payable player;
8          bytes32 commitment;
9          uint256 hand;
10     }
11     round[] private rounds;
12
13     function commit(uint256 hand) payable public {
14         require((hand == 1 || hand == 2 || hand == 3) && (rounds.length < 2));
15         rounds.push(round(payable(msg.sender), sha256(abi.encode(hand)), 0));
16     }
17
18     function open(uint256 hand) public {
19         require(rounds.length == 2);
20         for (uint256 i = 0; i < 2; i++) {
21             if (rounds[i].commitment == sha256(abi.encode(hand))) {
22                 rounds[i].hand = hand;
23             }
24             if (rounds[(i + 1) % 2].hand == 0) {
25                 return;
26             }
27         }
28         if ((rounds[0].hand == 1 && rounds[1].hand == 2) ||
29             (rounds[0].hand == 2 && rounds[1].hand == 3) ||
30             (rounds[0].hand == 3 && rounds[1].hand == 1)) {
31             rounds[0].player.transfer(1 ether);
32         }
33         else if (rounds[0].hand != rounds[1].hand) {
34             rounds[1].player.transfer(1 ether);
35         }
36         selfdestruct(payable(msg.sender));
37     }
38 }
```