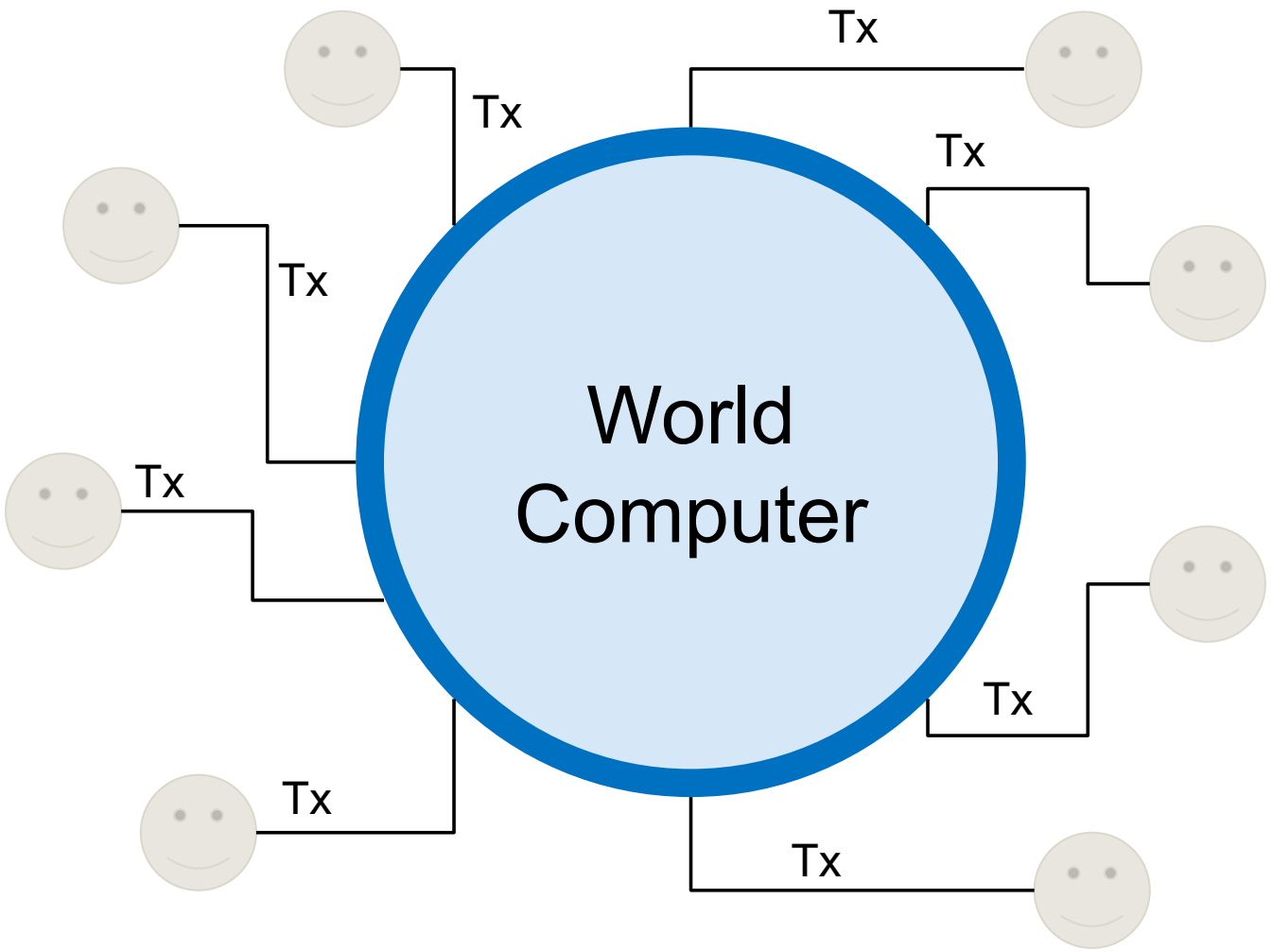
Recap Ethereum

Blockchain-Based Systems Engineering

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

World Computer

State Machine



Tx = Transaction

Properties

- **All participants** are **using** the **same computer**
- Users issue **transactions to call programs** on the computer
- **Everyone shares** the same **resources** and **storage**
- The **computer has no explicit**, single **owner**
- **Using** the computer's **resources costs money**

© sebis

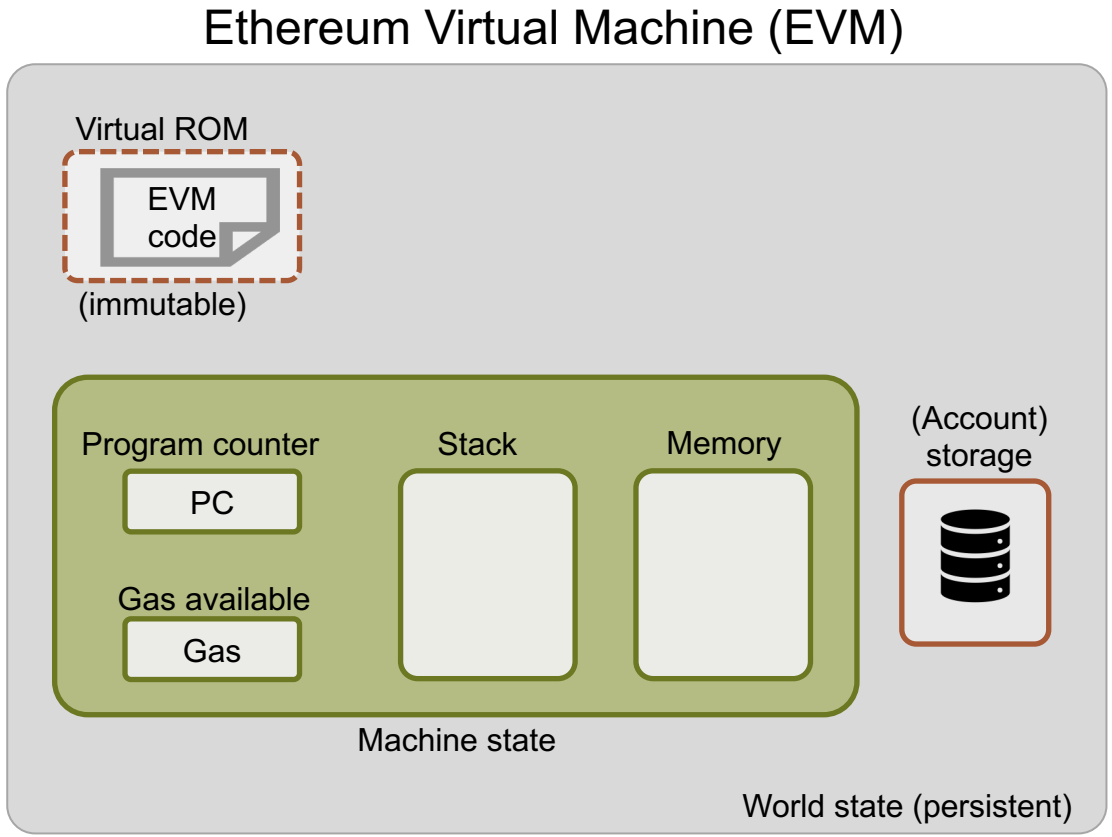
Brief Insight into the Ethereum Virtual Machine (EVM)

What is a virtual machine in a blockchain?

- Simply, virtual machines (VM) are mechanisms for creating instances of software that imitate real machines.

What is the Ethereum Virtual Machine (EVM)?

- EVM was the first virtual machine to be placed on a blockchain network, allowing programmers to conduct calculations on the blockchain for the first time. It's a stack-based architecture designed by Vitalik Buterin, which allows developers to create decentralized apps (dApps) on Ethereum. All Ethereum accounts and smart contracts are stored on this virtual machine.



© sebis

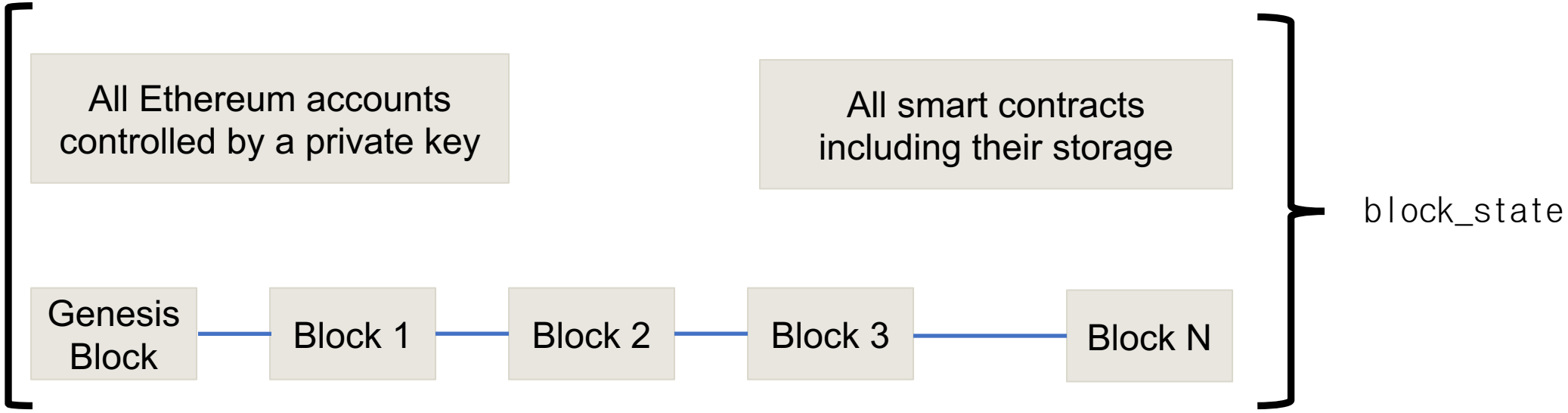
Ethereum Virtual Machine (EVM)

State Machine

The **EVM** specifies an **execution model for state changes** of the blockchain.

Formally, the **EVM** can be specified by the **following tuple**:
(block_state, transaction, message, code, memory, stack, pc, gas)

The *block_state* represents the **global state** of the whole blockchain including **all accounts, contracts and storage**



© sebis

Ethereum Account Types

Compared to Bitcoin, **Ethereum** uses an **account-based ledger**. Each **distinct address** represents a separate, **unique account**.

Ethereum supports two types of accounts:

1. **Accounts that are controlled by private keys and owned externally**
   - This account type is called “Externally Owned Account” (EOA).
   - Accounts that are controlled by a private key do not have any code stored on the blockchain. This type can be seen as the **default wallet of a user**[1]. It can sign transactions, issue smart contract functions calls and send Ether from one account to another.
   - The **origin of any transaction** is **always** an account **controlled by a private key.**

2. **Smart contract[2] accounts which are controlled by their code**
   - Smart contracts are treated as **account** entities with their **own**, unique **address**.
   - Contracts[3] **can send messages** to other accounts, both externally controlled and smart contracts.
   - They **can’t issue a transaction themselves.**
   - They **have** a persistent **internal storage** for reading and writing.

[1]In this context, a user is an human being. Thus, EOAs are controlled by humans.
[2]Deeper overview on smart contracts can be found in the following slides.
[3]”Contract” is used as a short form of “smart contract”.

© sebis

Account Properties

On an abstract level, an Ethereum account is a 4-tuple containing the following data:
(nonce, balance, contract_code, storage)

nonce
An increasing number that is attached to any transaction to prevent replay attacks.

balance
The current account balance of the account in Ether.

contract_code
The bytecode representation of the account. If no contract code is present, then the account is externally controlled.

storage
The data storage used by the account and empty by default. Only contract accounts can have their own storage.

© sebis

Ethereum Virtual Machine (EVM)

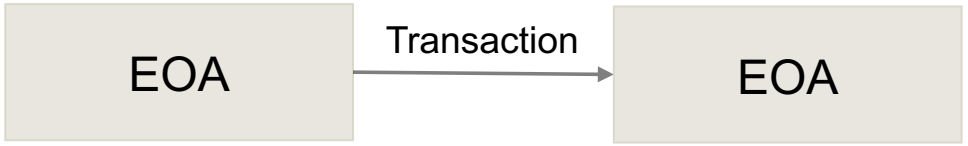
Transaction

A **transaction** is a **signed data package** that is **always sent by a EOA** and contains the following data:

- The recipient of the transaction
- A signature identifying the sender
- The amount of ether to transfer from the sender to the recipient
- An optional data field – data field is used for function call arguments (e.g. function inputs)
- A *GASLIMIT*[1] value, representing the maximum amount of gas you are willing to consume on a transaction
- A *GASPRICE* value, representing the fee the sender pays per computational step
- There are two types of transactions: From EOA to EOA and from EOA to smart contract

Type 1: EOA to Smart Contract

Type 2: EOA to EOA



[1]Same terminology is also used for defining the maximum amount of gas a block uses.

© sebis

Ethereum Virtual Machine (EVM)

Message

A message is very similar to a transaction. Messages are only sent by contracts and exist only virtually, i.e. they are not mined into a block like transactions.

A message contains:

- The sender of the message (implicit)
- The recipient of the message
- The amount of ether to transfer alongside the message
- An optional data field
- A *GASLIMIT* value

Whenever a **contract calls** a method on **another contract**, a virtual **message** is sent.
Whenever an **EOA calls** a method on a contract, a **transaction** is sent.

© sebis

Ethereum Virtual Machine (EVM)

Code & Memory

code

The code basically represents a smart contract as bytecode. For the EVM, a smart contract is a sequence of opcodes similar to assembly code.

Example:

PUSH1 0x60
PUSH1 0x40
MSTORE
PUSH1 0x04
CALLDATASIZE
LT
PUSH2 0x00b6
JUMPI
PUSH4 0xffffffff

memory

An infinitely expandable byte array that is non-persistent and used as temporal storage during execution.

© sebis

Ethereum Virtual Machine (EVM)

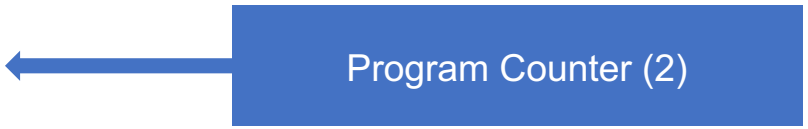
Stack & Program Counter

stack
The stack is also used as a fast, non-persistent buffer to which 32 byte values can be pushed and popped during execution.

pc
PC stands for “program counter”. The program counter is always initialized with 0 and points to the position of the current opcode instruction.

Simple Opcode Execution Example:

```
0       PUSH1 0x60
1       PUSH1 0x40
2       MSTORE
3       PUSH1 0x04
4       CALLDATASIZE
5       LT
6       PUSH2 0x00b6
7       JUMPI
8       PUSH4 0xffffffff
```

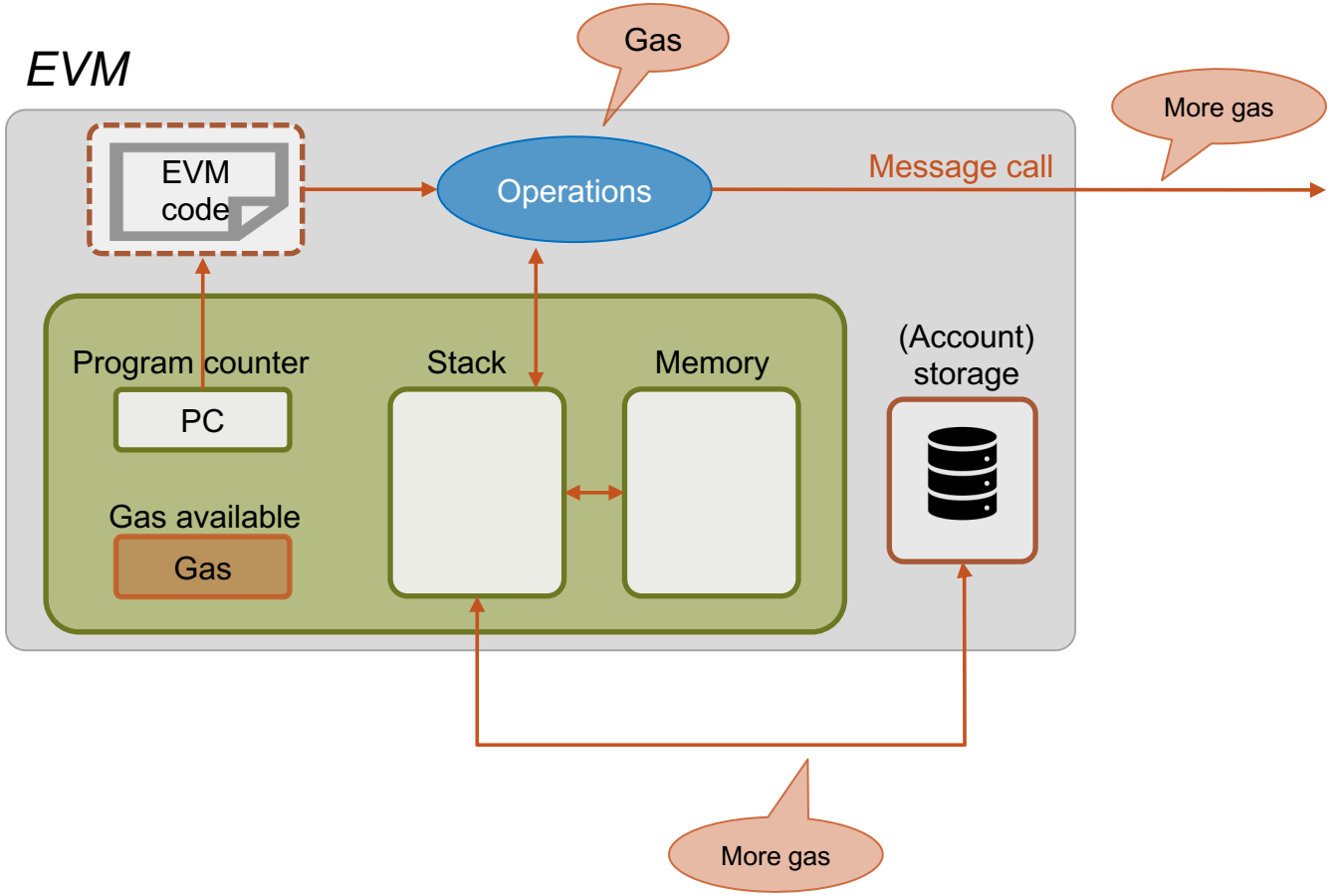


© sebis

Ethereum Virtual Machine (EVM)

Gas

- **Every executed opcode instruction** uses a miner's computational resources and therefore **costs a certain fee (called gas)**.
- Each opcode uses a certain amount of gas which may depend on the arguments of the operation, e.g., number of bytes to be allocated.
- The opcode for **selfdestruct (address)** uses **negative gas** because it frees up space from the blockchain.



© sebis

Brief Insight into Smart Contracts

- A **smart contract** is a **set of functions** that can be called by other users or contracts.
- They can be used to **execute functions, send ether or store data**.
- Each smart contract is an account holding object, i.e. **has its own address**.
- Smart contracts have some peculiarities compared to traditional software.

Security

The **development process** of smart contracts **requires special attention on security**.
Once **deployed**, a **contract** is **publicly accessible** by anyone on the network with the following information:

- Address of the smart contract
- OPCODE
- Number of public functions and their hash signature

- Furthermore, the whole transaction history is accessible (function calls + actual arguments).
- **Smart contracts** – once **deployed** – **cannot** be **changed** or **patched** anymore.

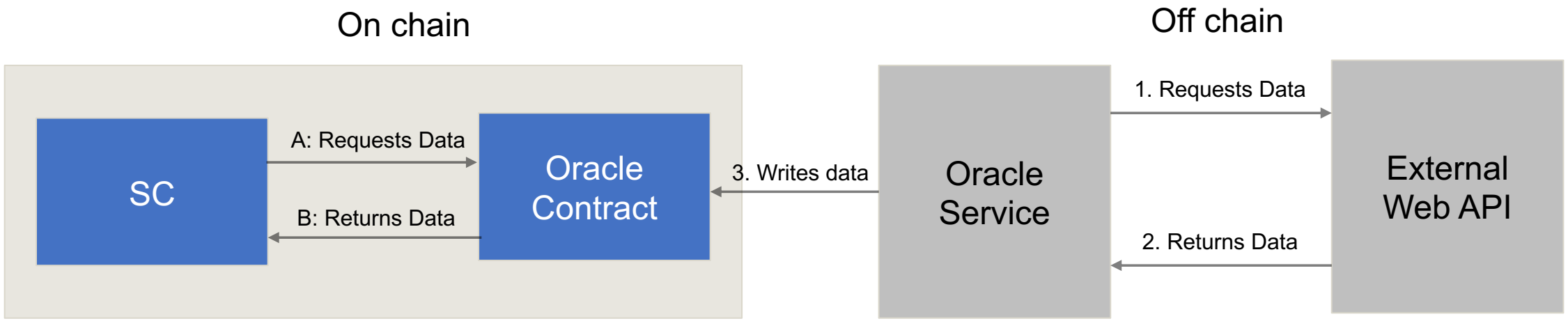
➔ All contracts deployed on the Ethereum blockchain are publicly accessible and can't be patched.

© sebis

Brief Insight into Smart Contracts as a Closed System

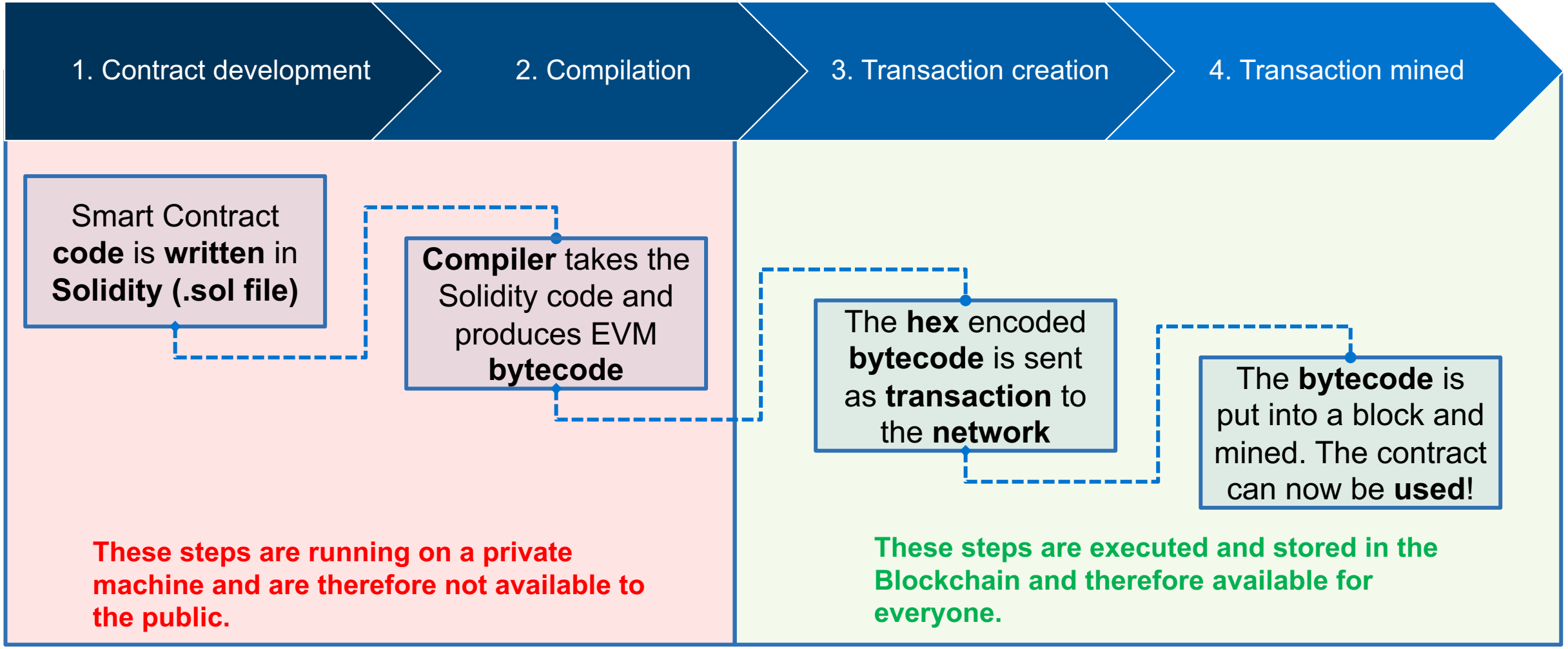
Smart contracts can't access any **data** from **outside** the **blockchain** on their own. There are no HTTP or similar network methods implemented to call external services. This is on purpose to **prevent non-deterministic behavior** once a function is called (there are also no functions to generate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

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 verify 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.



© sebis

From Solidity Source Code to a Deployed Smart Contract



© sebis

Anatomy of a Solidity Smart Contract File

File: **BBSE.sol**

```
contract BBSE {

    struct Tutor {
        string firstName;
        string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
        require(msg.sender == professor);
        _;
    }

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

    function getProfessor() view returns (address) {
        return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) onlyProfessor {
        Tutor tutor = tutors[tutorAddress];
        tutor.firstName = firstName;
        tutor.lastName = lastName;
    }

}
```

State variables

- State variables are permanently stored in the contract's storage.
- Changing the state requires a transactions and therefore costs ether.
- Reading the state of a contract is free and does not require a transaction.

© sebis

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```
contract BBSE {

        struct Tutor {
                string firstName;
                string lastName;
        }
        mapping (address => Tutor) tutors;
        address professor;

        modifier onlyProfessor {
                require(msg.sender == professor);
                _;
        }

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

        function getProfessor() view returns (address) {
                return professor;
        }

        // This function adds a new tutor
        function addTutor(address tutorAddress,
        string firstName, string lastName) onlyProfessor {
                Tutor tutor = tutors[tutorAddress];
                tutor.firstName = firstName;
                tutor.lastName = lastName;
        }

}
```

Function modifiers

- Function modifiers are a convenient way to reuse pieces of code.
- Changes the behavior of a function.
- Can execute code either before and/or after the actual function execution.
- The low dash _ indicates where the actual function code is injected.
- Often used for authentication.

© sebis

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```
contract BBSE {

        struct Tutor {
                string firstName;
                string lastName;
        }
        mapping (address => Tutor) tutors;
        address professor;

        modifier onlyProfessor {
                require(msg.sender == professor);
                _;
        }

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

        function getProfessor() view returns (address) {
                return professor;
        }

        // This function adds a new tutor
        function addTutor(address tutorAddress,
        string firstName, string lastName) onlyProfessor {
                Tutor tutor = tutors[tutorAddress];
                tutor.firstName = firstName;
                tutor.lastName = lastName;
        }

}
```

Constructor

- The constructor function is executed once when the contract is created through a transaction.
- The function cannot be called after the creation of the contract.
- Usually used to initialize the state of a contract.
- Execution costs gas and more complex constructors lead to higher deployment costs.

© sebis

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```
contract BBSE {

    struct Tutor {
        string firstName;
        string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
        require(msg.sender == professor);
        _;
    }

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

    function getProfessor() view returns (address) {
        return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) onlyProfessor {
        Tutor tutor = tutors[tutorAddress];
        tutor.firstName = firstName;
        tutor.lastName = lastName;
    }

}
```

Functions

- Functions are used to change the state of a contract.
- Can also be used to read the state of the contract.
- Consist of a name, a signature, a visibility, a type, a list of modifiers, and a return type.

Formal definition:

function (<parameter types>)
{internal|external|public|private}
[pure|constant|view|payable]
[(modifiers)]
[returns (<return types>)]

© sebis

Language Features Overview

Solidity is **inspired by JavaScript** and comes with a very similar syntax. Furthermore, it implements the standard set of features for high-level (object-oriented) programming languages. Compared to the dynamically-typed JavaScript, Solidity uses static types.

Built-in data types

int, uint, bool, array, struct, enum, mapping

Built-in first level objects

block, msg, tx, address

Built-in functions

Error handling: assert(), require(), revert()
Math & Crypto: addmod(), mulmod(), sha3(), keccak256(), sha256(), ripemd160(), ecrecover()
Information: gasleft(), blockhash()
Contract related: selfdestruct()

A set of literals

Solidity comes with some Ethereum specific literals (like eth for units, e.g., int a = 5 eth)

Flow control

if, else, do, while, break, continue, for, return, ? … : … (ternary operator)

© sebis

Function and Variable Visibility

In Solidity, functions can be declared with four different visibility types.

External
External methods can be called by other contracts and via transactions issued by a certain wallet. Methods declared as external **are always publicly visible** and **can't be called directly by the contract itself**.

Public
Public can **be called internally** by the contract itself but also **externally** by other contracts and via transactions. **State variable** which are defined as public will **by default have getter** method created automatically by the compiler.

Internal
Internal methods can only be accessed by the contract itself or by any contract derived from it. They are not callable from other contracts nor via transactions.

Private
Private methods can **only** be called **internally** by the contract who owns the method. **Derived contracts cannot access** a private method of their parent contract.

© sebis

Special Function Types (cont.)

Payable function
By default, it is not possible to send ether to a function because the function will by default revert the transaction. The behavior is intentional, it should prevent Ether that is accidentally sent from being lost. However, sometimes it is necessary to pay a contract, e.g. in case of an ICO. Therefore, Solidity implements so-called *payable* functions.

Example

```
function buyInICO() public payable { /* … */ }
```

- The keyword payable is also required for declaring constructors and addresses that can receive Ether (e.g., constructor payable { /* … */ }, function withdraw (address payable _to) public { /* … */ }).

- While implicit conversions are allowed from address payable to address, a casting function called payable (<address>) must be used for conversions from address to address payable.

```
address public customer;

function transfer (uint amount) public {
    payable(customer). transfer(amount);
}
```


© sebis