universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„A Microservice-based Execution Logic For Blockchain Systems“

verfasst von / submitted by

Florian Jaklitsch, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2020 / Vienna, 2020

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Abstract

A blockchain is a decentralized system running a transactional distributed database. It consists of a group of nodes where each node holds a replication of the database. These nodes are maintaining and agreeing on a set of global states stored in the database. Changes and updates on the states are made by transactions, which are all logged in blocks. Every block is linked with the previous block by including its cryptographic hash so that all transactions on the database states are immutably logged in the blockchain.

Smart contracts allow us to write and execute programming code on blockchain systems which makes it possible to create replicated and secure applications. With regard to data processing capabilities, however, the potentialities of smart contracts are often very limited. The constraints are due to hardware limitations on the one hand and factors specific to blockchain design on the other hand.

In this thesis, a new approach is introduced by redesigning the execution process and implementing the execution environment as a microservice running in a high-performance cloud infrastructure.

The first part surveys the background and state of the art in blockchain technology. The second part presents the design and implementation of a blockchain with a microservice-based execution environment built on Ethereum as an established blockchain system. Finally, an experimental analysis is performed and the possible impact of the presented approach with a focus on Internet-Of-Things devices is discussed.

Zusammenfassung

Die Blockchain ist ein dezentralisiertes System mit einer verteilten Datenbank. Es bildet ein Netzwerk bestehend aus unterschiedlichen Geräten, die alle ein Abbild der gemeinsamen Datenbank besitzen. Diese Geräte sind sich untereinander über den globalen Status der Datenbank einig. Änderungen am Status der Datenbank werden durch Transaktionen ausgeführt. Diese Transaktionen werden in Blöcken festgeschrieben, über die anschließend ein kryptographischer Sicherungswert errechnet wird. Jeder Block beinhaltet den Sicherungswert des vorhergehenden Blocks. Somit sind alle Blöcke kryptographisch verkettet und alle Transaktionen unveränderbar gespeichert.
Smart Contracts ermöglichen das Ausführen von Programmen auf der Blockchain. Damit ist es möglich, verteilte und gleichzeitig sichere Anwendungen zu entwickeln. Allerdings sind die Datenverarbeitungskapazitäten von Smart Contracts oftmals sehr beschränkt. Die Beschränkungen sind einerseits begründet durch Hardwaregrenzen und andererseits durch designspezifische Faktoren der Blockchain.
In dieser Arbeit wird ein neuer Ansatz einer cloudbasierten Ausführungseinheit für Blockchain Systeme präsentiert. Damit werden Smart Contracts nicht mehr lokal sondern in hoch performanten Cloud Umgebungen ausgeführt.
Der erste Teil der Arbeit umfasst eine umfangreiche Einführung in den technischen Hintergrund und den aktuellen Stand der Blockchain Technologie. Der zweite Teil präsentiert das Design und die Implementierung einer Blockchain mit einer Ausführungseinheit basierend auf Microservices in der Cloud. Abschließend werden die Ergebnisse experimenteller Analysen sowie die mögliche Bedeutung des präsentierten Ansatzes für zukünftige Entwicklungen und besonders für den Internet-Of-Things Bereich diskutiert.

Inhaltsverzeichnis

1 Introduction

1.1 Motivation

The blockchain, a young and novel technology, is believed to have a great impact on existing, upcoming and future technologies. It revolutionizes the understanding of secure and reliable machine-to-machine communication by no longer requiring a trusted third party. Although the technology is quite new, it has already gained a great deal of attention mainly because of the success of crypto-currencies. These were the first groundbreaking application for the blockchain technology that enables a decentralized, peer-to-peer based, financial system without a bank as central trusted party. Possibilities and fields of application of this technology, however, are more widespread.
We are talking about a technology that is essentially a distributed ledger operated by a group of nodes. These nodes are agreeing on a set of central states. Changes on the states are tracked in the distributed ledger. The state values are stored in a database replicated among all the nodes and are modified by applications called 'smart contracts'. Such applications are running on top of a blockchain platform and are coded in common programming languages. Thus, a blockchain system can be compared with a common database system where the distributed ledger is the immutable protocol of all operations on the data storage. Smart contracts are performing the data operations and consensus protocols are validating and enforcing agreement on the operations in the network. With the capability of executing smart contracts, current blockchain systems are able to run stand-alone programs on a distributed database. Thus, it is possible to build decentralized and replicated applications, which combine the main characteristics of a blockchain system: *Decentralization, Immutability, Transparency, Privacy* and *Traceability* [4].
There are actually numerous projects in different scientific areas trying to integrate blockchain technology with its benefits. On the one hand, there are projects that create or migrate systems and procedures onto the blockchain with the help of smart contracts. The authors in [16] have introduced a blockchain based solution for detecting tampered images by tracking ownership and copyrights of the authors and descriptive information of images. Such projects are challenging the data processing capabilities of a blockchain system. On the other hand, there are projects integrating the blockchain

technology into established systems. Especially in the area of Internet-Of-Things (IoT), there is much research on integrating blockchain technology into IoT devices and networks. An example for entirely integrating a blockchain on IoT devices is [26] where a solution for electric vehicle battery refueling is presented. Other projects are using the blockchain technology to tackle current IoT challenges like scalability as well as data privacy and security [18].

Although the blockchain technology has many benefits and plays an important role in new technologies, we are also facing new limitations, in particular when looking at blockchain systems as data processing platforms. First of all, some blockchains are using their own smart contract programming languages. Ethereum [7] as one of the most famous blockchain systems uses Solidity, a language that is still quite new and does not provide the range of capabilities that an established programming language can offer. This prevents wide-spread adoption and limits the programming possibilities [14]. Furthermore, every smart contract is replicated and executed on all nodes of the blockchain. In combination with computational heavy consensus mechanisms, this results in an enormous loss of speed in execution and transaction throughput. In the case of Ethereum, there is another limiting factor called 'gas', which is a parameter that determines the maximum amount of resources that can be used during an execution. Apart from this, we also have to consider that the participating nodes are equipped with different hardware resources. Executing calculation tasks with higher computational effort causes more load on weaker clients. Especially IoT devices are suffering from limited resources, including computation and storage capabilities. Thus, even standard operations are stressing factors. The main loading factors on an IoT device without including a blockchain software currently are the algorithms of the operations, the use of network protocols, the amount of transmitted data, received and processed information and data acquisition from sensors or other sources [17, p.1]. Looking at the current developments in the area of IoT technologies and projects integrating IoT and blockchain systems, new strategies are necessary to guarantee blockchain benefits on low hardware devices. A promising strategy would be combining IoT with cloud computing, which empowers the IoT systems with extra computing and storage capabilities [4, p.3].

In this thesis, a new approach is presented for a blockchain based on Eth-

ereum. It is more powerful regarding the execution times and data processing capabilities and that makes the blockchain compatible for IoT devices. This approach introduces a microservice-based blockchain system that outsources the Ethereum Virtual Machine (EVM) into the cloud. By moving the computational heavy parts off the blockchain we take the load off the nodes. This makes the blockchain technology more attractive for other technological areas like IoT, sensor networks and mobile devices.

1.2 Research Questions

1. *How to keep the security and integrity when decoupling the execution environment?*

   The workload for execution is separated from the transaction itself and passed to the cloud for processing. Afterwards, the results must be returned to the correct node and associated with their transaction again.

2. *How to manage the data access inside the microservice-based EVM without a local database?*

   When detaching the virtual machine from the Ethereum client and deploying it into the cloud, there is no replicated database available.

1.3 Research Approach

The context comprises the design and implementation of a distributed smart contract environment for blockchain systems according to the concepts of a service-oriented architecture. The idea is to provide an improved environment for executing programs and algorithms on the Ethereum blockchain without execution time and resource limitations. Further redesigning the execution logic should bring the blockchain technology closer to devices with reduced computational resources like IoT. This can be achieved by rebuilding the execution logic as stand-alone microservices running in the cloud or on high-performance environments and by redesigning the data flow and data management.

1.4 Outline

The following section provides an introduction into the state of the art of a blockchain system and its functionalities. Section 3 addresses the related work in this area. In section 4, the design and in-depth functionalities of the approach are presented. Then in section 5 the prototypical implementation of the approach is described in detail. Section 6 is about the experimental analysis and its implications. In section 7, advantages and drawbacks of the approach as well as its possible impact on future technologies are discussed. In the end, the thesis concludes with a forecast to future research.

2 Background Knowledge and State-Of-The-Art in blockchain technology

2.1 Basic Terms

- **Blockchain:** A growing list of blocks where all the blocks are cryptographically linked together.
- **Blockchain System:** The complete environment around a blockchain including physical devices and software components.
- **Block:** One element of a blockchain.
- **Smart Contract:** A small program executed on the blockchain.
- **Consensus:** An algorithm for finding agreement among a group of nodes.
- **State:** The state a of the data stored on the blockchain.
- **Trie:** Denotes the Merkle Tree, which is used as storage model.
- **geth:** The GoLang-based implementation of the Ethereum blockchain.
- **CEVM:** Cloud-based Ethereum Virtual Machine.
- **Low power device:** A device equipped with weak hardware components.

2.2 Distributed Systems

Distributed systems in general are systems without a centralized party. Centralized systems are currently the basis of the internet. It consists of a client that makes a request for some data and a server that responds to the request. If two clients want to communicate with each other, they need a server as a third party to relay the messages. Such centralized systems are usually operated by companies that manage all the data [19, p.9].
Distributed systems overcome this centralized concept by offering peer-to-peer (P2P) communication and cooperative group-based functionalities. Every node in such a system can be client and server. In case of a blockchain system, which is also a distributed system, the clients do not even have to know each other in order to find trust and agreement among each other in the network.

2.3 Functionality of Blockchains

A blockchain is a distributed system consisting of a group of nodes that maintain a set of shared, global states. The states are modified by transactions which are carried out by the nodes. All nodes are agreeing on every single transaction and on the order in which they are executed [6, p.2]. All historical states and transactions are stored in a list, a so called 'distributed ledger'. This list is divided into blocks that are cryptographically linked together, which are eponymous for the term as 'blockchain'. Agreement on the ledger among all the nodes is reached by consensus mechanisms.
Every node has one or more cryptographic identities (accounts) that are triggering and paying for transactions. Transactions can be simple state modifications like moving money from one account to another or smart contract calls performing advanced state changes.
A blockchain network consists of various types of nodes with different areas of responsibility. The two types that are part of every blockchain system are 'full nodes' and 'mining nodes'. Full nodes are the backbone of a blockchain holding an entire copy of the ledger and distributing it to other nodes. Additionally, they are validating all transactions and blocks. Mining nodes are also full nodes but with the privilege to append transactions to a block and publish new blocks to the blockchain. They are allowed to participate in the consensus and so mining nodes have the chance to influence the state of the blockchain [29, p.6].

Ethereum is one of the first platforms that has introduced smart contracts to the blockchain. Considered as a whole, it can be viewed as a transaction-based state machine [32, p.2]. In the beginning of an Ethereum blockchain, a genesis state is initialized as well as the initial accounts that are allowed to conduct transactions on the state. Afterwards, transactions are incrementally transforming the state. A state can include everything that is currently processible by a computer such as account balances, reputations, trust arrangements or data of the physical world [32, p.2]. A transaction is the transformation of a state into a new final state. All transactions of a certain time period are registered in blocks. The content of a block includes all transactions and the cryptographic hash of the previous block. When the new block is agreed on by all mining nodes in the network, the state transformations are globally accepted. Additionally, by including the cryptographic hashes, all blocks are chained together. Hence, the mining nodes

are not only agreeing on the newest block, but also on the transactions of all previous blocks. With this agreement, a new final state of the blockchain is reached.
The state transformations are performed by small programs called smart contracts. These programs are executed by accounts of a node when conducting a transaction. Therefore, a valid final state is one that comes through the agreement on the results of smart contracts.

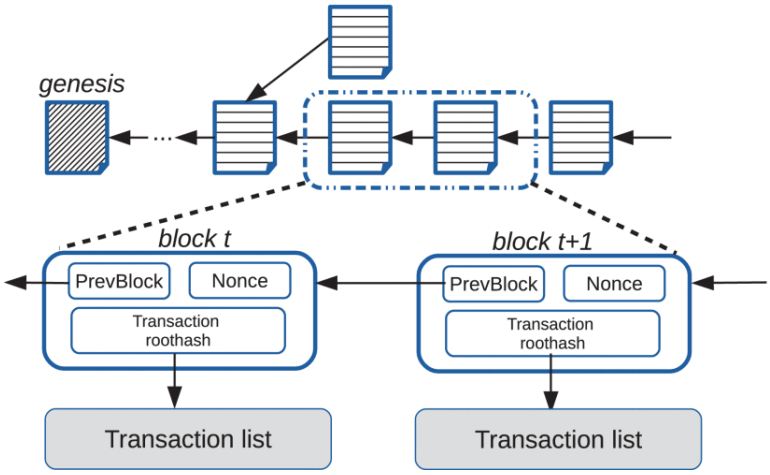


Figure 1: Blockchain extract. Transactions are stored in blocks and blocks are linked together. Source: [6, p.2]

Figure 1 presents a detailed overview of the blockchain structure. In the upper area of the figure, the linked blocks starting from the genesis are shown. The area below gives an insight into a block. First of all, each block includes the cryptographic hash of the previous block that forms the chain. Beside this, the block contains a 'nonce' which is a necessary input for the calculation of cryptographic hashes. In Ethereum, the nonce is a scalar equal to the number of transactions or the number of created contracts by the sender account [32, p.3]. Finally, the block contains the list of all transactions and an overall roothash, which is the signature of all transactions including the nonce.

2.4 Classification of Blockchains

Blockchains can be categorized into public, private and consortium blockchains. Public blockchains are platforms like Bitcoin or the public Ethereum

chain where anyone can participate by reading and writing data without identification.
In private blockchains, read and write access is restricted mainly by a single authority.
A consortium blockchain has open read access but only a pre-selected set of nodes can influence and control the consensus process [35, p.11]. In private and consortium blockchains, the connected nodes are identified. Thus, private networks are faster since they provide a level of trust which allows the use of simpler consensus protocols. Public blockchains lack this level of trust, which leads to the need for stronger consensus protocols. A more detailed discussion of consensus mechanisms can be found in section 2.62.

2.5 Blockchain and the CAP Theorem

The CAP Theorem defines specific transactional properties for any distributed system: *Consistency*, *Availability* and *Partition tolerance*. A distributed system always consists of a set of connected nodes that are cooperatively processing tasks. The CAP theorem states that it is impossible for any distributed system to fulfill all three properties at the same time. Such a network can guarantee at most two of the three properties [35, p.10]:

1. *Consistency*: Each node can always read the most recent values.
2. *Availability*: Any request at any point of time for some data always returns a result.
3. *Partition tolerance*: Enables a normal operability of the network even if parts of the network are down or failing.

Blockchain systems comply with the criteria of the CAP theorem. In this context *Consistency* means that all nodes are maintaining the latest replication of the distributed ledger. *Availability* guarantees that any call or transaction at any time is processed by the network and recorded in the distributed ledger. *Partition tolerance* in the context of blockchains states that even if parts of the network are inactive, the network of remaining nodes is still able to process transactions and enforce consensus.
Therefore, blockchain systems theoretically fulfill all three properties and contradict the CAP theorem at the same time. Yet, it still does not violate it. The blockchain's consistency is only an eventual consistency which is not

achieved simultaneously as availability and partition tolerance but after a period of time when reaching consensus on multiple subsequent blocks [35, p.11]. Consensus on multiple subsequent blocks is necessary to guarantee the continuity of the chain since forks can appear. Depending on whether forks are resolved sooner or later, (*eventual*) consistency is reached or no consistency is guaranteed [5, p.5].

2.6 Blockchain key concepts

A blockchain system is a distributed ledger based on a consensus mechanism that provides a verifiably append-only data structure. It is a dezentralized architecture that does not rely on a centralized authority [34, p.3]. Working with a blockchain comprises five layers where data are processed: *Application Layer*, *Contract Layer*, *Consensus Layer*, *Network Layer* and the *Data Layer*.

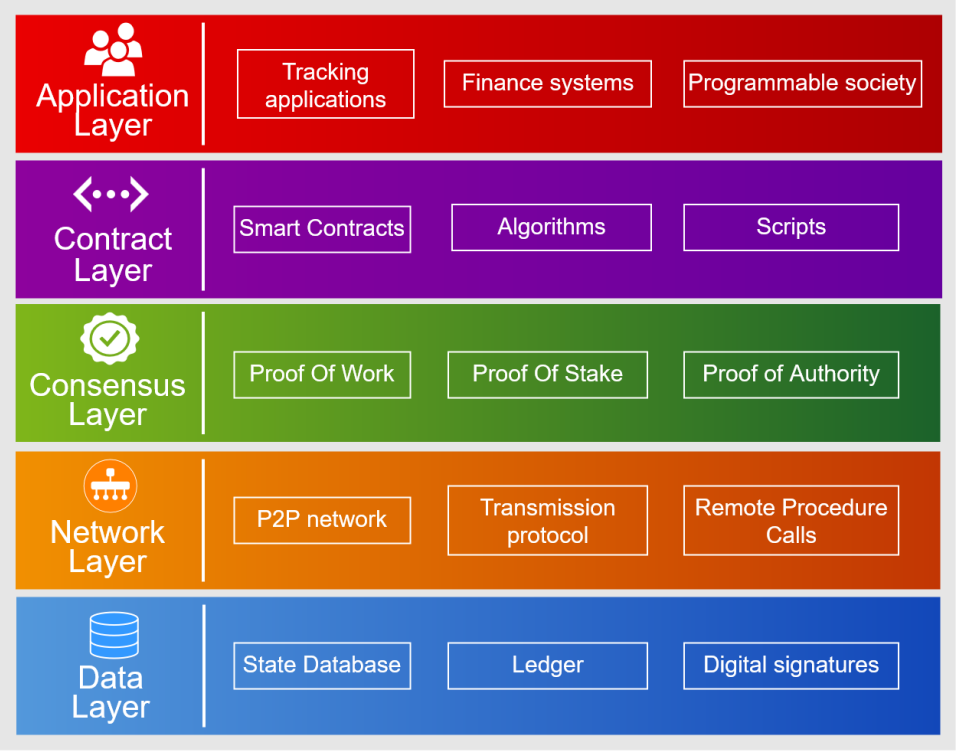


Figure 2: Blockchain layers

The overall fragmentation of a blockchain system is presented in *Figure 2*. First of all, it must be emphasized that the 'Application Layer' is not part of the blockchain system itself. It is the layer of applications that is interacting or integrating the blockchain as a service, a financial ledger or as a secure peer-to-peer network. These applications are conducting the Smart Contracts, running on the blockchain. Smart contracts are located at the *Contract Layer* and are performing the read and write operations as well as the data processing tasks on data stored in the blockchain database. Since a blockchain is a distributed network consisting of multiple nodes where each node maintains a replication of the database, agreement among all the nodes on their state of the database is required. Enforcing consensus in the network is the task of the *Consensus Layer*. The type of consensus mechanism depends on the network configuration. The *Network Layer* comprises the communication protocols. It enables the inter-network, peer-to-peer as well as the applications to network communication and data exchange. The *Data Layer* includes data management and storage utilities. It provides the state database, the ledger and all necessary signatures and keys.
All these layers are empowering the key functionalities that make a blockchain system: *Distributed Leger*, *Consensus*, *Cryptography* and *Smart Contracts*.

2.6.1 Distributed Ledger

A distributed ledger is the core component of a blockchain system. In most platforms it is implemented as an append only data structure that keeps track of all current and historical state modifications. All participating nodes in the network are agreeing on the distributed ledger and performing the state modifications on their local data storage. There is no unique architecture as all platforms are coming up with their own realisations of the distributed ledger.

2.6.2 Consensus

Finding consensus is a basic challenge in distributed systems where nodes are cooperatively working together. Since the ledger of a blockchain is replicated among all nodes, updates on the ledger must be agreed by all parties without a central entity. Numerous consensus mechanisms are existing starting from mainly computation based to purely communications based protocols. The

type of consensus protocol used for a blockchain, depends on the control of network participation which can be classified as 'permissioned' and 'permissionless' [33, p.8].
Permissionless blockchains are mainly public blockchains where anyone can participate without authorization. This means that every node is able to send, receive and validate transactions as well as to participate in the consensus [33, p.9]. The nodes in a public network cannot trust each other. This is why protocols are necessary to bring consensus to a group of untrustworthy nodes. Such networks like Bitcoin[3] or the public Ethereum network [7] use consensus based on 'Proof-of-Work' (PoW).
Permissioned blockchains are mainly private or consortium blockchains. These networks consist of a group of authorized and identified nodes. By relying on identity, less expensive consensus like 'Byzantine Fault Tolerance' (BFT) or 'Proof-Of-Authority' (PoA) can be used.
By now the most widely known consensus is Proof-Of-Work. It is a simple protocol where the mining nodes are solving computationally heavy problems. The one who solves the problem fist can contribute a new block to the chain. Since numerous computers worldwide are trying to create a new block this protocol causes large amounts of energy consumption.
In general there is a deep interest in finding efficient consensus mechanisms. An energy efficient alternative to PoW is the 'Proof-Of-Stake' (PoS) approach. PoS introduces a stake that refers to the network tokens held by a participant that can be invested in the consensus process [33, p.16]. Thus the chance for a PoS miner to propose a block is proportional to their stake value [33, p.16]. Another consensus protocol especially for permissioned blockchains is Proof-Of-Authority (PoA). This consensus protocol consists of some pre-defined nodes which are considered trusted authorities and which are allowed to propose the next blocks. Each authority is given a time slice, via round-robin scheduling, where it can append new blocks to the blockchain. This simple protocol avoids single point of failure while ensuring balanced workloads among the authorities [6, p.9].

2.6.3 Cryptography

Blockchain systems are dependent on cryptographic techniques for their security model to guarantee the integrity of the ledgers as well as the identities of the user and the transaction signatures. Integrity here refers to the abil-

ity to detect tampering of the blockchain data. The most commonly used cryptographic method for integrity is hashing.

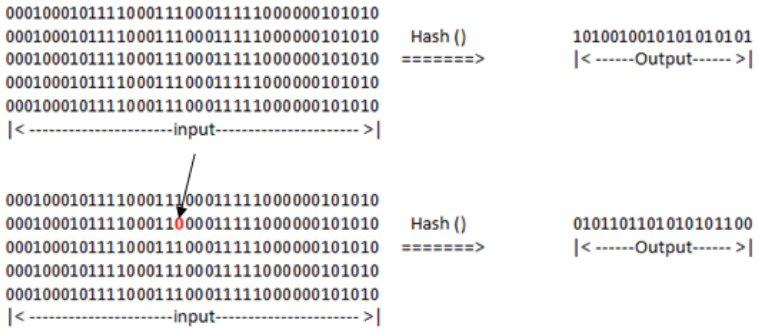


Figure 3: Uniqueness of a hash. Source: [19, p.16]

A hashing function is a one-way function that calculates a unique result of some input data, also called 'fingerprint'. Depending on the strength of the hashing function, there exists no second different input that generates the same result. A small change in the input leads to a completely new and different output.
Some blockchain systems use multiple levels of integrity protection, which in the case of Ethereum are two. The first level protects the global states by using a hash tree ('Merkle tree').

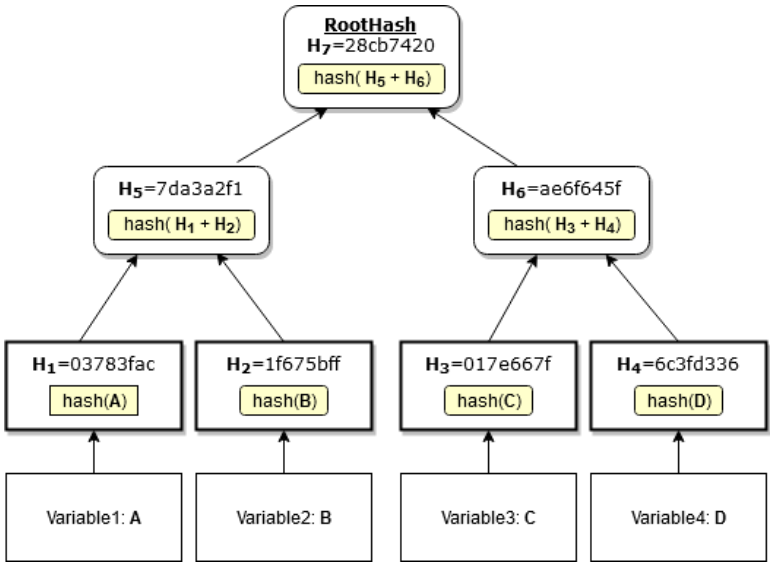


Figure 4: Merkle Hashtree.

Leaves of the tree contain the states and internal nodes comprise the hashes of the children up to the root. Any state change in the leaves results in a new root hash.
The second level is established by using cryptographic hash pointers which are linking all the blocks through the chain. Once a block is appended to the blockchain, it is immutable. This is achieved by linking hashes of blocks. The content of block $n + 1$ contains the hash of block n. Consequently, any modification in block n invalidates all the subsequent blocks. Combining Merkle trees and hash pointers creates a secure data model for tracking historical changes to global states.
User and transaction identities are guaranteed by public key cryptography. Identities are derived from public key certificates and the private key is used to sign the transactions, which in the following can be verified by others with the public key [6].

2.6.4 Smart Contract

A smart contract is a program that contains a collection of instructions and data that have been saved at a certain address in the distributed database. By exposing public functions and application binary interfaces (APIs), a smart contract offers a way to interact with users and to offer contract agreement [20, p.2]. Based on this architecture, a smart contract allows users to achieve agreements among parties in a blockchain network.
This means that a smart contract is a distributed application that is executed on the blockchain. It refers to the computation executed when a transaction or a call is performed. It can be regarded as a stored procedure that is called by an interaction. The inputs, outputs and state modifications by the smart contract execution are agreed by all participating nodes in the network. Smart contracts can be characterized by their language and runtime environments. On the one hand, there are systems like Bitcoin that provide fewer than 200 opcodes to create stack-based scripts and they are executing their scripts in the same runtime environment as the rest of the blockchain system. On the other hand, there are platforms like Ethereum which is even Turing complete and able to run arbitrary program codes in its own virtual machines [6, p.5].

2.7 Microservices

In the course of time, software development has moved from a monolithic centralized architecture to a more decoupled service-based architecture. In a monolithic architecture the logic of the different layers are bundled together in a package or archive [25, p.5]. The service-oriented architecture decouples the particular components where each component provides their services to other components. In the microservice architecture the components are autonomous. In this case all the components are providing an Application Programming Interface (API) where different clients and components can access the services. Thus, the components are entirely independent from each other. Each component can be developed, tested and deployed independently without disrupting the whole application [25, p.7]. Running a microservice as a container makes the service hardware independent, scalable and deployable in the cloud.

3 Related Work

Blockchain, a new technology that could have a major impact on future technologies, is the research basis for many projects. Approaches and ideas regarding the data processing capabilities, the impact on Iot as well as the usability and drawbacks are presented and discussed by scientists around the world.

A good comparison of established blockchain systems regarding their data processing capabilities can be found in [6]. The authors also elaborated a benchmark for measuring blockchain performances. A comprehensive overview of the various types of smart contracts for blockchains, their architecture and applications is discussed in [28].

A blockchain project that realized an external execution logic is Hyperledger fabric [2]. It uses Docker containers as execution environments and is the first blockchain technology that enables the use of standard programming languages. EOS [15] as another blockchain system follows a different approach of a smart contract execution logic by implementing WebAssembly [31] as execution engine. This has the benefit of execution speed and the support for common programming languages.

A comprehensive article presenting a survey on blockchain and IoT integration is [21]. The authors discuss ideas on how blockchain capabilities can help addressing the drawbacks of IoT. IoT and blockchains are two promising technologies in the research area of 'smart cities'. In [17] the authors propose a sensor network based on blockchain technology with 'smart cars' as participating nodes. They made an analysis of conditions under which a blockchain can be used in distributed sensor networks and IoT. Further they developed several design models of blockchain networks for devices with limited resources.

Another blockchain based vehicular network with IoT devices for smart cities is presented in [24]. They try to build an intelligent, secure, distributed and autonomous transport system with better utilization of infrastructure and resources. [4] surveys how blockchain technology can be integrated with IoT. The authors also discuss the challenges and how a blockchain can empower Internet of Things.

An interesting approach of combining blockchain technology and microservices is discussed in [34]. This paper introduces a blockchain-enabled decentralized microservices architecture for smart public safety. The security services are

containerized microservices, build by smart contracts within a permissioned blockchain network.
Regarding the performance of blockchains, [22] developed a methodology for evaluating a blockchain platform based on execution time, average latency and average throughput.

4 Design of the microservice based EVM blockchain

This section focuses on the design concerns of the the Ethereum Virtual Machine (EVM) running as a microservice. To give a better understanding of the necessary re-engineering steps from the original Ethereum implementation, the section is splitted into two parts. The first part introduces the core functionalities in detail, whereas the second part presents the design and the necessary changes at the Ethereum system for the microservice based EVM. The design of the Ethereum blockchain presented in this thesis detaches the virtual machine from the rest of the system and moves it into the cloud. This requires some redesigning of established procedures and modifications of the virtual machine.

4.1 Ethereum Protocol

4.1.1 Design of Ethereum node

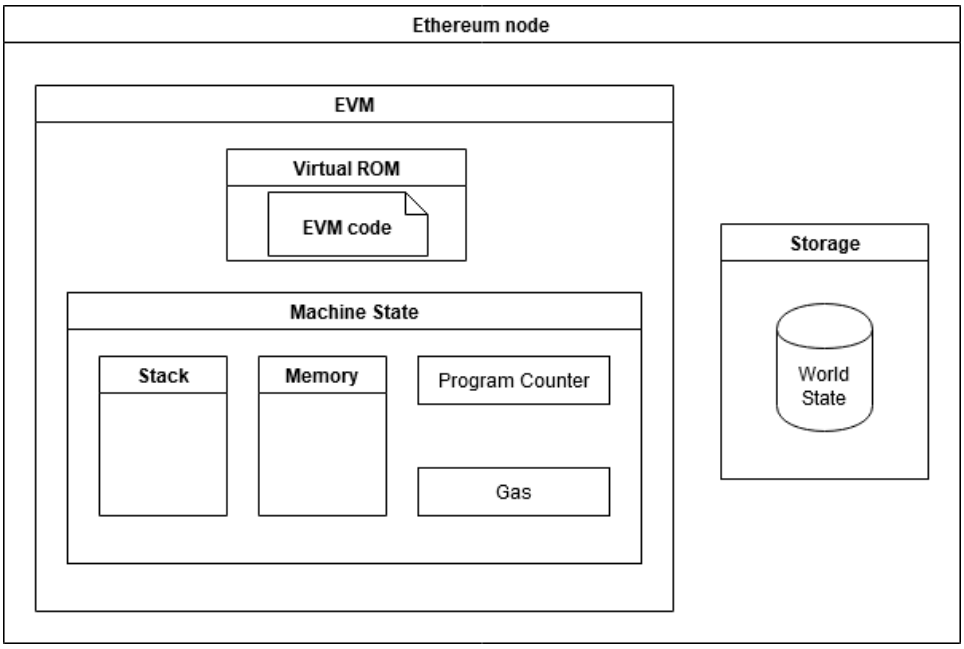


Figure 5: Ethereum node architecture

Ethereum is a blockchain with *Turing* complete state machine model support which enables the execution of programming code. The execution engine (EVM) is a stack-based architecture with a word-addressed byte array as memory model. Instead of following the standard of Von Neumann architecture where program code is stored in a generally-accessible memory, the engine uses a virtual ROM [32, p.10]. This is visualized in *Figure 5*. A program counter sets a pointer to the current instruction on the stack. The memory is used to cache relevant values and results of the stack. The gas bounds the whole execution and avoids programs from running out of control. EVM and storage including the state database are components of every Ethereum node.

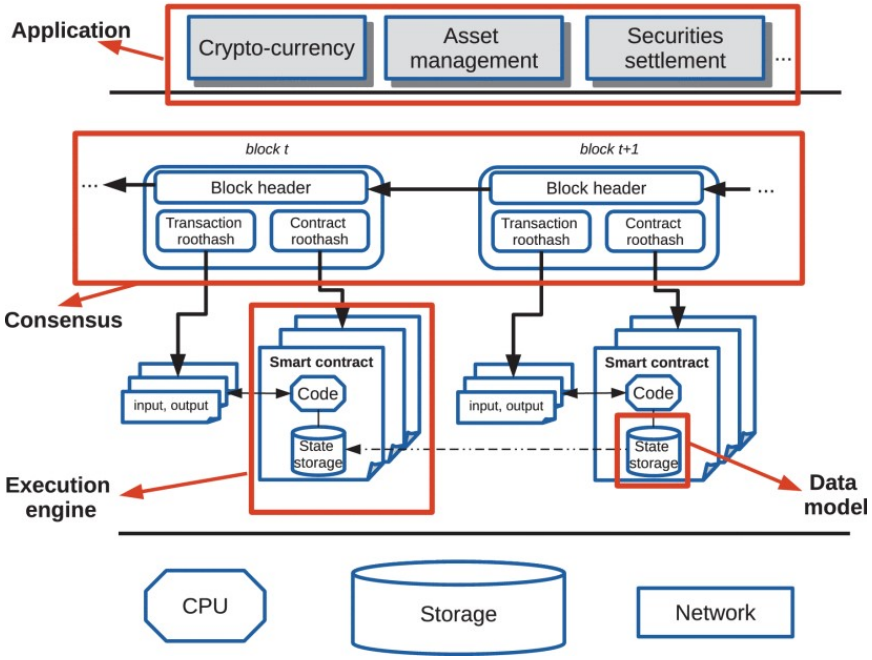


Figure 6: Blockchain software stack. Source: [6, p.11]

Figure 6 shows the overall software stack of a blockchain system with subdivisions into the corresponding layers. This figure accurately describes the software stack of Ethereum. The top layer represents the applications which are interacting with the blockchain. Such applications are crypto-currencies, asset management or external applications interacting with the blockchain. Right below there are the blocks chained together. Block $t+1$ is cryptographically dependent on the previous block t by including the hash

of its neighbor t in the block header. Beside this a block stores two root hashes: In the first place the new roothash of the 'Transaction Trie' which is the immutable cryptographic signature of transactions in this block. It is the result of all hashes of transactions, interacting with Smart Contracts on the blockchain. Further details on the Ethereum 'Trie' storage model are presented in section 4.1.3. In the second place a block contains the contract roothash which is the cryptographic signature of the current contract state. Finally consensus is achieved on every newly appended block. So the whole network is agreeing on the content and signatures of every single block in the chain. The content of a block is the result of smart contracts. A smart contract is a program running on the blockchain. It is executed by the execution engine called Ethereum Virtual Machine (EVM). It has its own state storage for variables and data. Transactions are updating the state storage by sending and reading input and output data. State storage modifications are performed by all transactions of one block which leads to a new smart contract roothash after the transactions are executed. This new contract roothash is again stored in the block.

4.1.2 The Stack-based Execution

In Ethereum the execution is performed by a virtual state machine, known as the Ethereum Virtual Machine (EVM). EVM uses a stack architecture with a memory model consisting of a simple word-addressed byte array. The last item pushed on the stack is the first item taken from the stack.

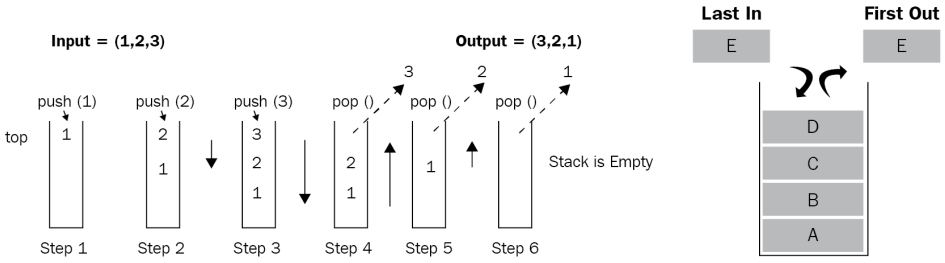


Figure 7: Stack machine functionality. The last item pushed on the stack is the first pulled off the stack. Source: [19, p.46]

The stack size is limited to 1024 items and a maximum item size of 256-

bit [32, p.10]. Smart Contracts are deployed and executed in EVM bytecode format which is an assembler like bytecode language consisting of instructions. These instructions are processed on the stack. The set of instructions comprises on the one side standard instructions for stack operations, arithmetics, jumps and local memory access [12, p.4]. On the other side the set is enriched with some additional opcodes for hashing, creating transactions, accessing and modifying the storage of the executing account and accessing the environment in which the contract was called in [12, p.4].
Each instruction consumes a certain amount of gas. The gas limits are specified by the sender of the transaction. So exceeding the gas limit during an execution would result in an exception that reverts the current transaction.

4.1.3 The storage layer

The overall organisation of data in Ethereum is based on Merkle trees (*Figure 4*). There are different trees for the various types of data. Ethereum makes use of a data structure called 'Merkle Patricia tree' (or trie). The trie uses a database backend that maintains a mapping of bytearrays to bytearrays [32, p.3]. There are three types of tries: *State Trie*, *Storage Trie* and a *Transaction Trie*.

State Trie: It is the only global state of Ethereum that is constantly updated. The trie contains a key value pair for every account that exists on the network. Every account is a 4 item array of [nonce,balance,storageRoot,codeHash] where *storageRoot* is the root of another patricia trie.
The purpose of the storage trie is the cryptographic dependence of the root node on all internal data. As described in the Ethereum paper [32] on page 3, this means that it is the hash and secure identity of the entire system state at a given point in time. Furthermore it allows any previous state to be recalled by simply altering the root hash accordingly. Since all such root hashes are stored in the blockchain, it is possible to revert to old states.

Storage Trie: Each account in the state trie has its own storage trie. The hash of the storage tree is the *storageRoot* position in the account array of the state trie. All the contract data are stored in the storage trie.

Transaction Trie: Ethereum creates a separate trie for every block. It stores the transactions and the positions of the transactions contained in a block. After the block is mined, it is final and cannot be updated. Thus, the transaction trie never updates and thereby the position of a transaction can never change.

Storage is maintained as part of the system state which means that contract variables are stored in a merkle tree. Every smart contract running in the EVM has its own permanent storage. This storage can be seen as a large array with a maximum size of 2^{256} initialized with zeros.
For further details or possible updates the reader is referred to the latest version of the Ethereum yellow paper [32].

4.1.4 From programming code to bytecode

```
1   contract Storage {
2      uint256 val1 = 12;
3      uint256 val2 = 2;
4      uint256 res;
5
6       function calculate() public view returns (){
7           res = val1 * val2;
8       }
9   }
```

Listing 1: Smart contract with a function multiplying two values from storage.

Programs for Ethereum are mainly written in Solidity programming language. The compiler in the following transforms the human readable Solidity program into a machine readable bytecode. A bytecode is the low-level version of a smart contract code. It can be broken down into a number of instructions which are describing the actions performed by the EVM on the stack. The code snippet in *Listing 1* presents a smart contract storing the result of a multiplication of two persistent variables **val1** and **val2** from the storage in variable **res**. Compiling this code snippet produces the bytecode shown in *Listing 2*.

```
0x6080604052600c600055600260015560006002553480156 01e
57600080fd5b50606d80602c6000396000f3fe60806040526004
3610601c5760003560e01c80633662412114602157 5b600080fd
5b60276029565b00 5b6001546000540260028190555056 fea265
627a7a723158200b42b099bcd13a033aa33f999bb15a4628f9e9
e56d2b28836682fd18c54bd75a64736f6c634300050c0032
```

Listing 2: Bytecode version of codesnippet *Listing 1*

The bytecode contains the sequence `5b6001546000540260028190555056`. It is the logic of the calculate() function of the code snippet in *Listing 1*. This bytesequence can be broken down into the following instructions:

5B	JUMPDEST
60	PUSH1 0x01
54	SLOAD
60	PUSH1 0x00
54	SLOAD
02	MUL
60	PUSH1 0x02
81	DUP2
90	SWAP1
55	SSTORE
50	POP
56	*JUMP

The `JUMPDEST` instruction defines the destination of the function logic in the bytecode. The `PUSH1` instructions are pushing values on the stack which are used in the following by the `SLOAD` instructions to load the values from the storage locations $0x00$ and $0x01$. This values are also pushed on the stack and multiplied by the `MUL` instruction. Then the third storage location $0x02$ is pushed which is used by the `SSTORE` to write the result to location $0x02$ in the database.

A more detailed demonstration of the stack-based OPCode execution is presented in section 4.1.6.
The EVM has an integrated instruction table where the OPCodes and their attributes are defined. Each OPCode has an instruction identifier code, an attribute δ that defines the number of items to be removed from the stack and an attribute α that defines the number of items to be placed on the stack. The Mnemonic representation of all OPCodes are referenced in [32].
The bytecode interpretation works the following: The snippet **60**01**5460**00**5402** starts with 60 which is the identifier of PUSH1. PUSH1 has the attributes $\delta = 0$ and $\alpha = 1$ meaning that it pushes the subsequent item of 1 byte onto the stack, which is 01. The next value is 54, the identifier for SLOAD with attributes $\delta = 1$ and $\alpha = 1$. SLOAD takes one item from the stack, which

is in this example the previously pushed 01, uses this value to look up the storage position and pushes back the stored item onto the stack. Then 60 (PUSH1) and 54 (SLOAD) is performed again for a second item at storage position 00. Afterwards 02 MUL is executed with $\delta = 2$ and $\alpha = 1$. It takes the two storage items from the stack, multiplies them and pushes back the result.

4.1.5 Database access

As already described in section 4.1.2, Ethereum uses a word-addressable word array as storage model [32]. Storage denotes the replicated database of the Ethereum network. It is the area where the smart contracts are storing their persistent data. Every smart contract has its own storage accessible via a unique address. It can neither read nor write to any other storage apart from its own. The length of the storage for each smart contract is 2^{256} slots. Each storage slot can store 32 bytes (256 bits). Every defined variable in a contract is mapped to a slot in storage. The storage layout is according to the ascending positions. So the first defined variable gets storage position 0x00, the second variable 0x01. Every slot is initialized with zero. Operations on the storage are only performed by the EVM which offers two instructions for interaction:

- `SLOAD`: Read data from the storage.
- `SSTORE`: Write data to the storage.

The array based storage model requires different strategies to represent fixed length and arbitrary length datatypes.

Fixed length datatypes

These are datatypes where it is known at the time of initialization how much information the values can store. In the Ethereum system such datatypes are among others:

- `uint8, uint32, uint64, uint256, byte`
- `[10]uint8, [32]byte, bytes32`
- Structs that are containing the above types

Each variable has its own slot in the storage of the smart contract where it can store values with a size up to 32 bytes. The values of statically sized variables are simply laid out in storage slots starting from index 0. This is how the values are represented in the storage:

Fixed length storage				
Address	0x4337df0AB94590787840F12DDd0E3541647228AD			
Slot	0x00	0x01	0x02	0x03
Values	10	999	0x3E7	0xC0FEFE

Arbitrary length types

Aside from values of fixed length, there are data types that can expand dynamically as more data is appended. Such dynamic types are for example:

- `mapping(bytes32 => uint256)`
- `[]uint256, []byte, etc.`
- `strings, bytes`

For dynamic datatypes it is not known how many bytes and storage slots are needed. Ethereum uses different strategies according to the specific datatype.

In case of `arrays` of arbitrary length the slot `(p)` reserved for the respective variable, stores only the length of the array. The data of the array or its individual index positions are located at the slots that are resulting from hashing `p` with the keccack256 function `+i` where `i` is the incremented index position.

Array storage				
Address	0x4337df0AB94590787840F12DDd0E3541647228AD			
Slot	(p)	keccak256(p)	keccak256(p)+1	keccak256(p)+2
Values	array.length=3	uint256: 12	uint256: 2	uint256: 93

This example demonstrates how the uint256 array [12,2,93] is laid out in storage.

`String` and `byte` datatypes are basically behaving like arrays with a slightly different strategy. If the data to be stored is smaller or equal 31

bytes, then they are using the rest of the same variable slot where also the length is stored. In this case the data is stored in the 31 higher-order bytes (left aligned). The last lower-order byte stores the `length of the data * 2`.
If the data is longer than 32 bytes, the variable slot `(p)` keeps only the `data length * 2 + 1` and the rest is stored according to arrays by hashing with `keccak256( p )`. Unlike arrays, the `length` is not the number of used storage slots but the encoded length of data. To get the actual data length and number of used storage slots:

$$length = (encodedLength - 1)/2$$

$$slots = ceil(length/32)^1$$

Mappings are always assigning a value to a corresponding key `(k)`. The map variable slot `(p)` itself stores no information. The values are located at the output of hashing key combined with the map slot `keccak256(k.p)`.

Mapping storage				
Address	0x4337df0AB94590787840F12DDd0E3541647228AD			
Slot	(p)	$keccak256(k_1, p)$	$keccak256(k_2, p)$	$keccak256(k_3, p)$
Values		uint256: 12	uint256: 2	uint256: 93

This example demonstrates how the mapping ["k1": 12, "k2": 2, "k3": 93] is laid out in storage.

4.1.6 EVM

The EVM is a stack-based architecture with a 256-bit stack item size and a maximum stack size of 1024 [32, p.10]. It is the runtime environment of the Ethereum blockchain. This machine executes program code written in arbitrary programming languages like Solidity. The code is processed by the EVM as bytecode which requires a translation from human readable program code to machine readable bytecode. The translation from high-level program code to low-level bytecode is made by an external compiler. The compiler is not included in the EVM or blockchain system.

[1] ceil: means rounding up; returning the next highest integer.

As mentioned at the beginning, the EVM is a stack machine following the concept of `Last In First Out (LIFO)` [19, p.46]. All operations are performed by pushing and pulling data on a stack. The last item put on the stack is the first taken from the stack (LIFO). The machine is performing operations by taking off the values at the top of the stack and pushing back the results onto the stack. The following snippet gives a demonstration of how the core of the machine works. It calculates $12 + 2 * 93 - (10 + 8)$ where 10 and 8 are stored in the storage:

```
1    contract Calculate {
2        uint256 v1 = 10;
3        uint256 v2 = 8;
4
5        function getResult() public view returns
          ↪  (uint256){
6            return 12 + 2 * 93 - (v1 + v2);
7        }
8    }
```

Listing 3: Smart contract with a function calculating the result of multiple values.

The theoretical stack-based execution of the `getResult()` function:

	OPCODE	Stack [bottom-top]	Description
0	PUSH1 0x0c	[12]	push 12 on stack
1	PUSH1 0x02	[12, 2]	push 2 on stack
2	PUSH1 0x5D	[12, 2, 93]	push 93 on stack
3	MUL	[12, 186]	multiply 2 with 93
4	ADD	[198]	add 12 and 186
5	PUSH1 0x00	[198, 0]	push 0 on stack
6	SLOAD	[198, 10]	read storage at pos 0
7	PUSH1 0x01	[198, 10, 1]	push 1 on stack
8	SLOAD	[198, 10, 8]	read storage at pos 1
9	ADD	[198, 18]	add 10 and 8
10	SUB	[180]	subtract 18 from 198
11	POP	[ ]	pop 180 from stack
12	STOP		halt execution

The PUSH1 opcodes are pushing the values in hexadecimal representation on to the stack. In line 3 the MUL instruction multiplies the top values 2 and 93 by taking the values from the stack and pushing the result 186 back. The ADD instruction in line 4 works identical to MUL except that it sums up the top values 12 and 186. In line 5 the value 0 is pushed on the stack which is the first position of the contract storage. The value stored at position 0, here this value is 10, is read by the SLOAD instruction in line 6. The same process is carried out for the second storage variable. In Line 9 both storage variables are added and subsequently subtracted from 198. POP takes the final results 180 from the stack and STOP halts the execution.

4.1.7 Gas parameter

In the Ethereum network the gas parameter is used to control the transactions and executions. The purpose of the gas is on the one hand to prevent smart contract programs from executing infinitely. This can be caused for example by programming errors like infinite loops. As a consequence it limits the total amount of computations that can be done by the EVM. On the other hand the gas makes sure that all operations on the blockchain

performed by client applications are fairly charged. The latter is mainly necessary on public blockchain networks because it is the basis of their financial system that keeps them alive. The sender must pay a fee for every interaction with smart contracts that are performing modifications on the blockchain. This fee is payed to the miners of the network which are sequencing transactions, creating new blocks and maintaining the unique state of the network. The gas price is globally set within the network. Each transaction specifies the amount of gas it is going to use. If a transaction uses more than the specified amount of gas, the execution is aborted and all state changes are reverted. The fee of a transaction can either be set manually or be estimated by execution simulations. In private networks the gas could have other purposes like powering certain services.
To sum up, gas bounds the execution of smart contracts and depending on the configuration, it allows the network to charge transactions proportional to the computational cost they cause [13, p.3].

4.1.8 Calls and Transactions

Calls and transactions are the two main methods for interacting with smart contracts. Calls are executing a function of a smart contract. The returned values of the function call can be the result of calculations or storage accesses (for example see *Listing 3*). Calls are processing values passed as inputs or stored in the database but they are not performing modifications on the blockchain state. No write operations are made by calls and so they are not consuming any gas.
Transactions on the contrary are functions which are processing and writing data to the storage. Thus, transactions are modifying the blockchain state and are requiring consensus of the network. Such a state change consumes gas. Calls as well as transactions are performed per RPC (Remote Procedure Call) calls to the Ethereum node by passing a package containing information about the sender, the target contract and the function to execute. *Listing 4* presents a function call on a smart contract. The function called is defined as `getMapAt(uint256)`. It returns the value stored at a certain map index position that is passed as input value. The example call is `getMapAt(62)`. The package passed via RPC to the Ethereum node looks like the following:

```
{
    "From": "0xF1d37500d01d148Ecdb98778F83c1eF72F79D096",
    "To" : "0x23c29845BaFb6cD48E40d9Fa800D912EA8C0117c",
    "Gas" : "0x4630c0",
    "GasPrice": 20000000000,
    "Value": 0,
    "Data": "0xa222f34e000000000000000[...]00000000003e"
}
```

Listing 4: Content of a package for a smart contract call

The *From* field contains the public key of the sender. *To* specifies the smart contract storage address. *Gas* and *Gasprice* are maximum consumable and maximum used gas determined by the compiler. The *Data* field contains the signature of the called function including the passed input parameters. The first part of the *data* field is the hexadecimal representation of the 4byte truncated keccak256 hashed function `getMapAt(uint256)`:

```
0xa222f34e = keccak256(getMapAt(uint256))
```

Listing 5: Function signature with keccak256 hashing.

The passed value, in this example 62, is located at the end of the datafield. Since the whole datafield uses hexadecimal representation, the input value 62 becomes 3e.
The necessary *Gas* must be determined prior sending the transaction. If the afford of the transaction is known before, the *Gas* can be set manually which allows immediate execution.

4.2 The microservice based Ethereum blockchain

4.2.1 Overview

The previous part presented a detailed introduction into the Ethereum system. This section describes the microservice based design of the Ethereum blockchain. As shown in *Figure 9*, the implementation consists of two parts: The *Cloud Layer* and the *Ethereum network*. The *Cloud Layer* comprises all the detached functionalities of the blockchain. This includes the outsourced execution engine of Ethereum which is now called Cloud-based Ethereum

Virtual Machine (CEVM) and the *PreExec Microservice* for resolving database accesses and managing concurrency.
The Ethereum client, dedicated to the *Ethereum network*, has been redesigned in order to interact with the loose coupled microservices. As initial software the go implementation of Ethereum geth [9] has been used. The EVM has been detached from the Ethereum client and reimplemented as a stand-alone system. Considering the characteristics of service oriented architectures providing fine granularity, loose coupling and continuous delivery, the EVM has been reconstructed as a microservice. It is equipped with a REST API that runs within a container which can be deployed into the cloud.
At the Ethereum client, a new layer for data access and communication with the microservices has been implemented. To guarantee data consistency, another microservice which is responsible for storage resolving and storage access has been implemented.
Thus a new network originates that consists of Ethereum nodes running the distributed database and performing their smart contract executions in the cloud. Since the system is designed to empower the execution engine of Ethereum, it mainly targets private blockchains.
The approach describes a fully working Ethereum network that does not violate any of the blockchain or Ethereum principles. Further the system is designed to be fully compatible with the original *geth* implementation.

4.2.2 System Architecture

The System architecture presents the design of the microservice-based standalone Ethereum virtual machine and the redesign of the geth implementation for microservice compatibility. For this purpose, on the geth side, the EVM has been detached and replaced by a new layer that manages necessary preprocessing, data resolving and the interaction between the *Ethereum network* and the *Cloud Layer*.

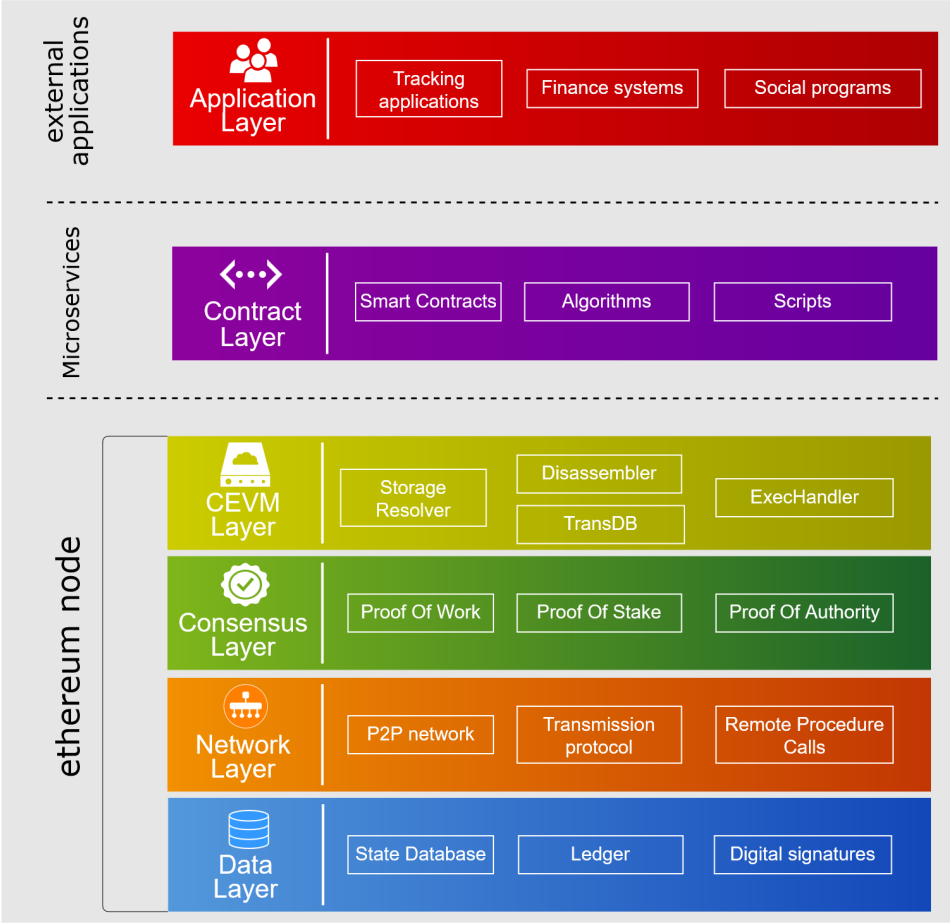


Figure 8: Layers of the redesigned Blockchain node.

Figure 8 presents the layers of the redesigned Ethereum version. Next to the standard blockchain layers *Data Layer*, *Network Layer*, *Consensus Layer*, *Contract Layer* and *Application Layer* there is also the introduced *CEVM Layer* depicted.

The *Application Layer* represents the external applications which interact with the blockchain. The *Data Layer* includes the database and the merkle trees. The whole chain structure, data, timestamps and signatures are managed by the data layer. In the *Network Layer* the whole communication between all nodes within the network is performed. The *Consensus Layer* is responsible for achieving agreement on the state of the *Data Layer* among

all nodes in the network. Before the smart contract execution can take place, the data must be prepared in the *CEVM Layer*. At this level the bytecode is disassembled and the storage accesses are simulated. Beside this the storage values are resolved. Finally the bytecode with the resolved storage variables is passed to the microservice-based *Contract Layer* for execution.
In a more systematic way, the network can be partitioned into two basic layers: the *Ethereum network* and the *Cloud layer*.

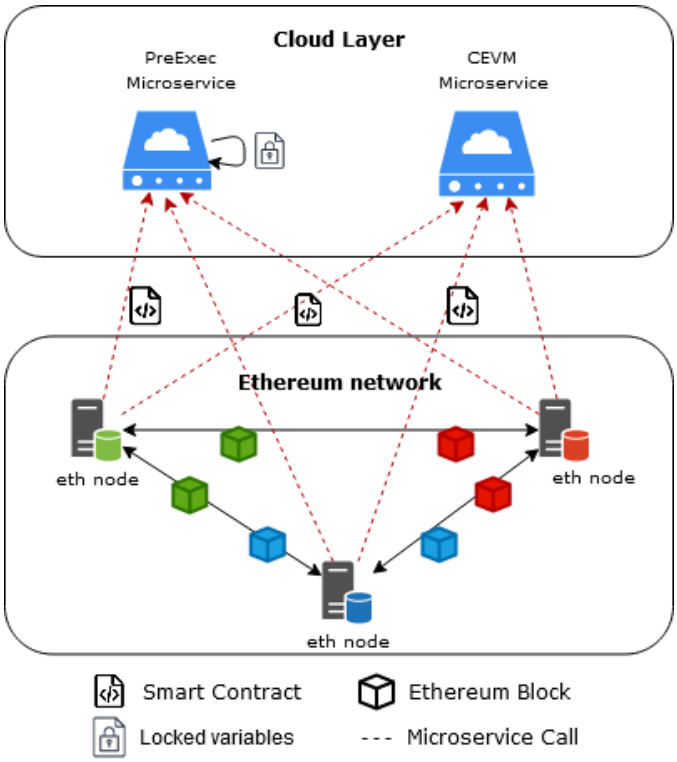


Figure 9: CEVM overview. Ethereum network with cloud-based microservices.

Figure 9 gives an overview of the system's functionality. There is a set of distributed Ethereum nodes (eth nodes), communicating and exchanging blocks with transactions. Each of the eth nodes maintains a replication of the distributed database and communicates with the microservices when a call or transaction is made. Once a smart contract is invoked the node starts the communication with the cloud. At first it passes the smart contract to the *PreExec Microservice*. The *PreExec Microservice* resolves all read and write storage accesses of the called smart contract function and locks

corresponding storage locations. This guarantees data consistency during the initial execution phase. Details on data consistency are discussed in section 4.2.5.
Afterwards the node passes the smart contract to the CEVM for execution. The CEVM executes the actual byte code and returns the results if necessary.

Figure 10 shows a detailed overview of the systems activities partitioned into 3 layers. The first layer gives an insight into the Ethereum (geth) node. The starting point is triggered with a new call or transaction on the geth node. Then a new transport database 'TransDB' object is created which transfers the relevant 'stateDB' values to the microservice. Beside this a 'LockProposal' object is also created responsible for write permissions at the CEVM microservice. After the initial operations the bytecode of the called smart contract is processed. With this action the CEVM layer in the geth node is entered. At first the bytecode is disassembled and the OPCodes according to the called function are processed. The main purpose of the OPCode processing is to resolve database read operations. The resolved values are stored in the TransDB. Afterwards a TransportPackage is created that contains all Ethereum Virtual Machine relevant configuration data as well as the TransDB and the LockProposal. Finally the TransportPackage is passed to the CEVM microservice. In the third layer of the diagram the CEVM microservice stays idle as long as no request is processed. As soon as a new TransportPackage arrives, the configuration values, TransDB and LockProposal are extracted and used to setup a new EVM instance. Then the EVM is executed. It processes the bytecode of the smart contract and performs appearing read or write database operations on the TransDB. Because of the fact that all geth nodes are executing on CEVM, concurrency must be considered. Therefore, if a write operation (SSTORE) is performed, the lock state of the variable must be checked before. The lock state is stored in the LockProposal. If there is already a LockProposal set and the value to be modified is locked, a lock error will be created and the whole execution aborted. If the state is unlocked then a lock for this variable will be set and the values are written to the TransDB. Regardless of whether the execution is finished or aborted, at the end a response package with either the abortion error or the results as well as the modified TransDB is created and returned to the calling geth node. Layer 2 unpacks the CEVM response and returns it to Layer 1. The returned values are interpreted. If it contains a locked

error, the initial database reading operations are performed again.

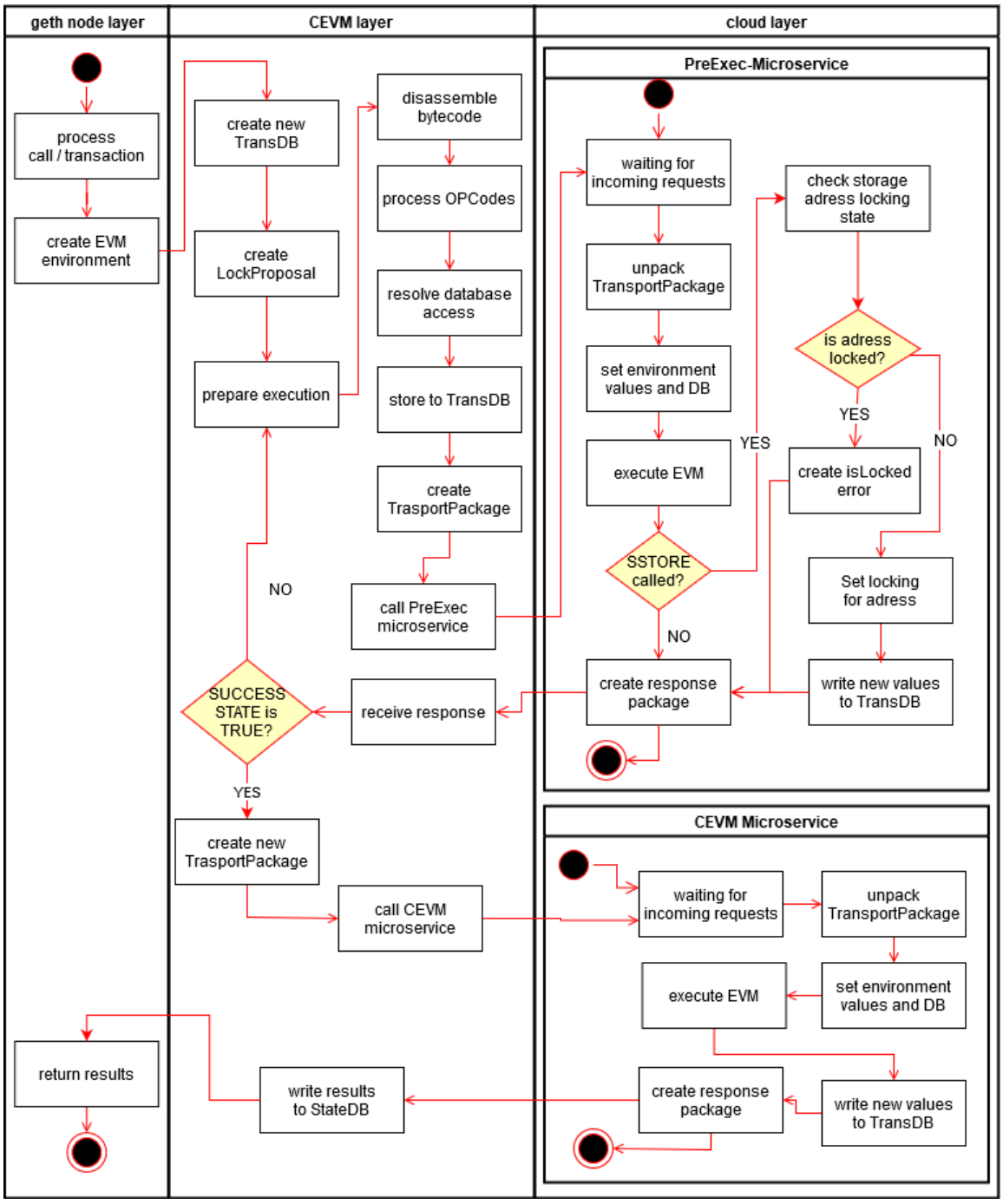


Figure 10: CEVM overview. Activities and decisions.

If the response contains execution results and no errors, these results are finally mapped to the StateDB and returned to the caller.
A component based view of the system is provided in *Figure 11* divided into the microservices and the blockchain node. The microservices are both providing an HTTP interface. The interface on the PreExec microservice as well as the CEVM microservice is only called by the geth node. The router in each microservice maps the calls to the relevant 'MSController' functions which are in first place unpacking and encoding the data and then passing them to the EVM. Inside the geth node the ExecHandler is the component which controls the dataflow and interacts with the microservices.

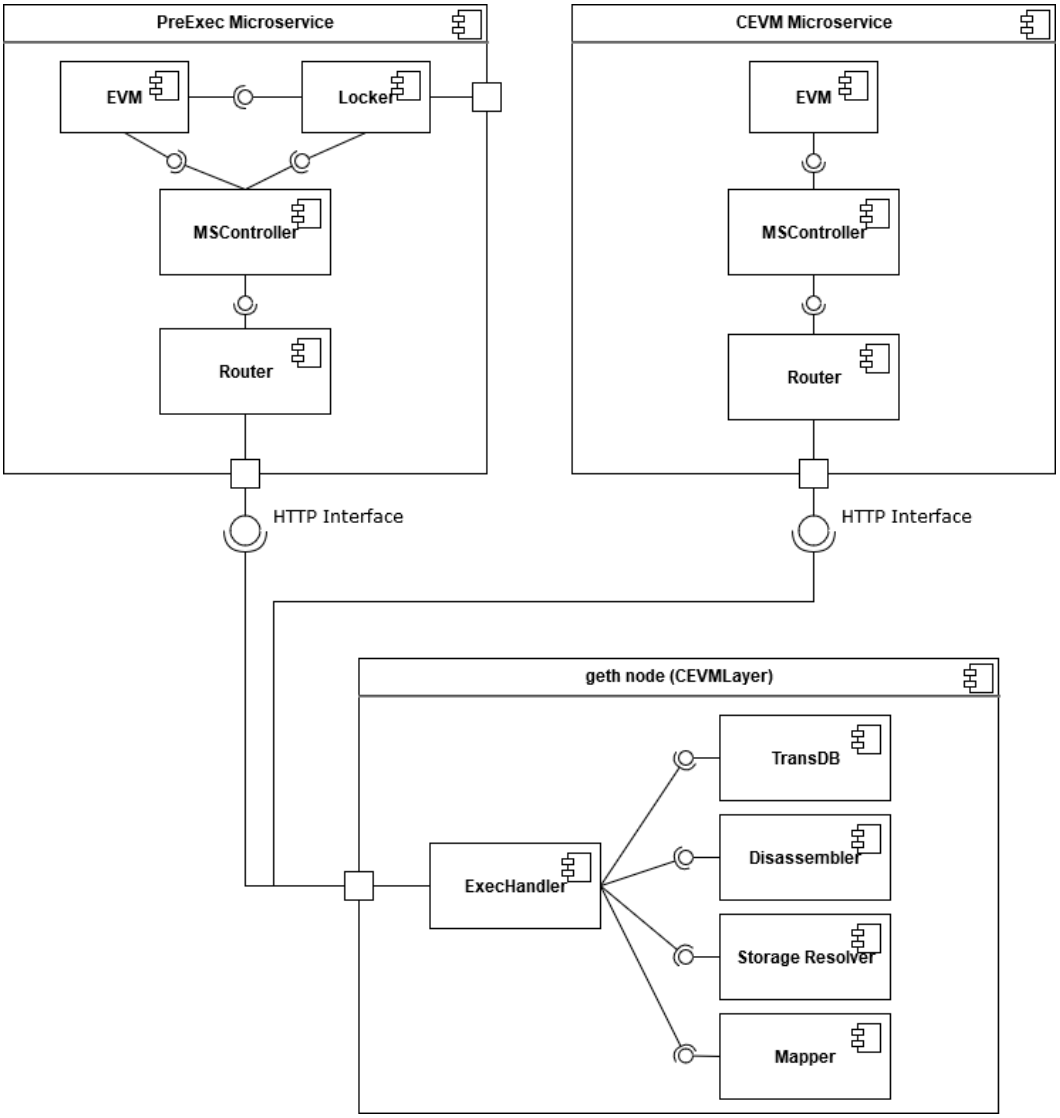


Figure 11: CEVM component diagram.

4.2.3 Dataflow redesign

The integration of a microservice based execution engine requires some changes in the dataflow of the blockchain system. Dataflow in this case means the way the data is passed through the Ethereum node and the cloud. The whole data flow starts when a smart contract is invoked by an interaction (call or transaction). In this case the Ethereum node receives a package see *Listing 4*.

The input field contains the hashed hexadecimal encoded function that is called. Within this function it is also possible to pass input values. In the following the bytecode is disassembled and the function related storage opcodes are pre-resolved. The pre-resolving step is especially necessary for the array datatypes. As described in section 4.1.5, array datatypes are storing their length in the variable storage slot and are incrementally hashing the array index positions up to the length. Thus, without the length value it is not possible afterwards for the *PreExec Microservice* to perform a stack-based resolving of the array.
In original Ethereum nodes, the EVM is locally integrated. The usual dataflow of Ethereum looks like this:

1. Transaction call to a smart contract with data passing.
2. The Ethereum node accesses the database storage of the smart contract
3. The Ethereum node passes the input data, the smart contract bytecode and the storage address to the EVM
4. The EVM executes the bytecode and performs database reads and temporary writes.
5. In case of a simple read call, the result is returned to the caller. In case of a transaction with write operations, a consensus in the network is needed.
6. The Ethereum node puts the transaction into the transaction pool (TxPool) which will be distributed to all consensus nodes to achieve agreement in the network.
7. Finally point 3 and 4 are executed again by all participating nodes in the network. This time the database writes are persistent.

The reimplemented version performs remote calls to the microservice EVM. This implies some fundamental changes in the dataflow. The microservice only comprises the EVM but no database. So database reads and writes must be performed before the microservice is called.
Therefore, another layer has been implemented in the Ethereum node that performs data management and communication with the CEVM. Now the dataflow looks the following:

1. Transaction call to a smart contract with data passing.
2. The Ethereum node analyses the bytecode and pre-resolves necessary database locations for the *PreExec Microservice*. It stores the values in a temporary transport database.
3. The Etherem node passes the transport database to the *PreExec Microservice*.
4. The *PreExec Microservice* pre-executes the code and collects all required storage addresses. It returns the transport database with the addresses.
5. The Ethereum node fills all storage addresses of the transport database with values of the main state database.
6. The Ethereum node passes the inputdata, the smart contract bytecode and the filled transport database to the CEVM.
7. The CEVM executes the bytecode and performs database reads and writes on the transport database.
8. In case of a simple read call the result is returned to the Ethereum node and subsequently back to the caller. In case of a transaction the transport database and the result is returned to the Ethereum node.
9. The Ethereum node submits the transaction to the transaction pool which will be distributed to all mining nodes in the network.
10. Finally the points 2-7 are executed again by all participating nodes in the network. Every node is now performing exactly the same execution process. Afterwards the values from the transport database are persistently written to the state database by every full node.

4.2.4 Consensus protocol: Proof-of-Authority

The consensus protocol applied on the CEVM blockchain is Proof-of-Authority (PoA). It is a faster and more resource-saving consensus mechanism whose prominence is due to performance increases with respect to typical BFT algorithms [5, p.3]. As shown in the table of blockchain technology employed in the industry [1, p.10], the PoA consensus mechanism is already widely

used. The algorithm is based on a set of N trusted nodes called authorities. Only a majority $N/2 + 1$ of them is assumed as honest. The authorities are relying on a mining rotation schema. This schema determines the next leader responsible for creating the next block based on the block number and the total number of authorities in the network. Beside the selected leader, other authorities are still allowed to propose a block every $N/2 + 1$ blocks. Thus, at any point in time at most $N - (N/2 + 1)$ authorities are allowed to propose a block and maliciously acting authorities can be voted out [5, p.5]. This makes the PoA algorithm a very secure and efficient consensus mechanism which is further compatible for devices with reduced computation capabilities.

4.2.5 Concurrency and data consistency

In a distributed system, concurrent data access endangers the data consistency. As already discussed in section 2.5, a distributed system can guarantee at most two of the three properties *Consistency*, *Availability* and *Partition tolerance*. In a blockchain system *Consistency* is the property that is only eventually guaranteed. Data consistency in Ethereum is guaranteed by the decentralized consensus protocol. It controls the admission of new blocks, the secure verification of the blockchain and the consistency of data content of transaction records [35, p.3].
External Applications are interacting with the nodes by sending transactions or calls to be performed on the blockchain. On each mining node these transactions are ordered, primarily sorted by gas and when consensus is reached, executed successively by the EVM on all full nodes in the network. However the execution circle in the microservice-based design might cause problems regarding the data consistency for external applications. These consequences are discussed in section 7.2.3. In the main geth implementation, the EVM has direct access to the local replication of the database. In the redesigned version, the EVM is outsourced as microservice without immediate database access. Thus, the storage access must be resolved before the execution can take place. This creates a time period where consistency could be violated as demonstrated in the following models.

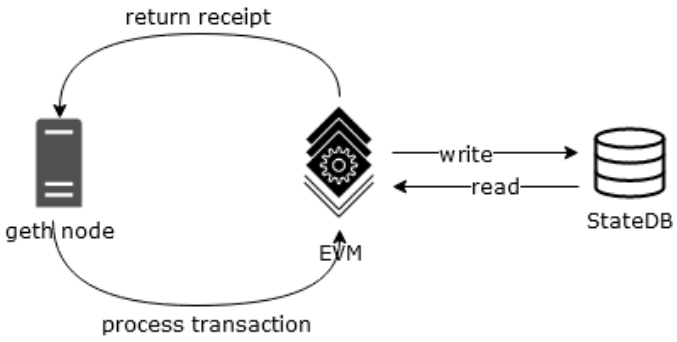


Figure 12: Execution circle

Figure 12 shows the overall execution circle. When an interaction with a smart contract is conducted, the geth node calls the *EVM* which executes the program. During the execution the EVM reads and writes data to the *StateDB*. Finally the *EVM* returns the receipt and results of the execution. When comparing the steps necessary for execution in detail, the extra time period which the microservice-based execution needs, becomes clear.

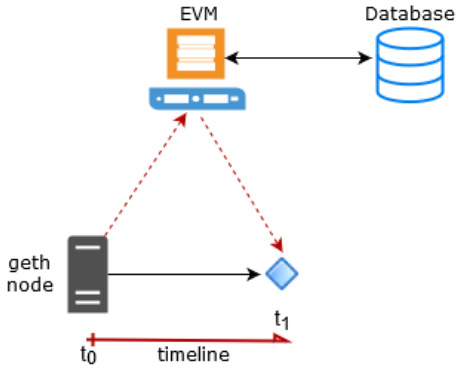


Figure 13: Execution dataflow of the original geth implementation

Figure 13 shows the execution dataflow of the original Ethereum implementation. The timeline marks each step a geth node runs through. One step stands for the complete execution of one task. Here the geth node only passes the bytecode to the EVM for execution. The EVM has immediate access to the database and can resolve storage accesses itself. So the time needed for data access is negligible.

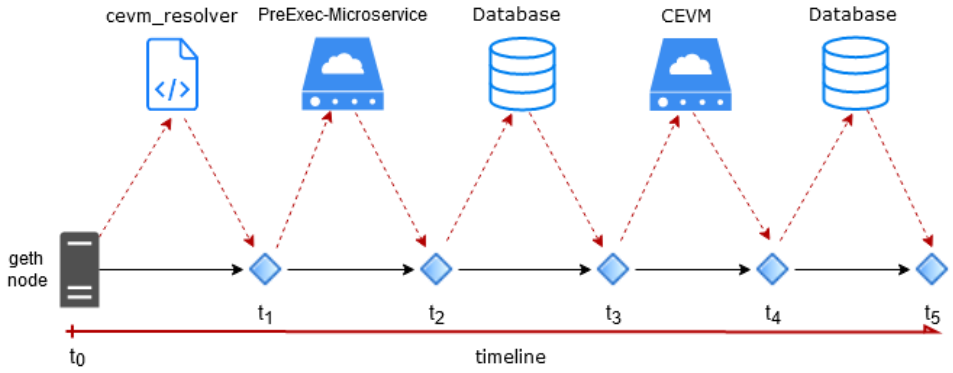


Figure 14: Execution dataflow of the CEVMGeth implementation

The same procedure of the microservice-based geth implementation is shown in *Figure 14*. At first the geth node pre-resolves the storage locations by the cevm_resolver which takes one timestep. The real execution time of here is negligible. Then at t_1 the *PreExec-Microservice* runs the stack-based storage access resolving. The *PreExec-Microservice* is an externally located microservice. So the 'Round-Trip-Time' (RTT) inclusive processing time must be considered. Processing time can be neglegated again but the RTT is a few milliseconds. The database access at t_2 is performed locally by the geth node. At t_3 the code execution is conducted. Like it was the case at t_1 with the *PreExec-Microservice*, the RTT for communication and execution must be considered. Finally the geth node writes the state changes by the *CEVM* to the database at t_4. It takes at least 4 timesteps in the pre-execution phase until the execution itself takes place and another one until the pending storage updates are tracked by the geth node.
This sums up to a few milliseconds. Compared to one timestep with negligible time in the original implementation there is an enormous gap until the database is updated. This reveals a period of time where data consistency is not guaranteed for tasks of external applications like the gas estimation. Details regarding execution periods are discussed in section 6.2.

5 Implementation

This section presents a prototypical implementation based on the geth [9] implementation.

Specific terms:

- `StateDB:` The programming name of the main Ethereum state database.
- `TransDB:` Stands for transportation database, the mobile version of the StateDB.
- `SResolver:` The programming name of the *Storage Resolver*.
- `TransportPackage:` The package containing all the data that are exchanged between Ethereum node and microservices.
- `ExecHandler:` The programming name of the *Execution Handler*.

5.1 Necessary components

As discussed in section 4.2.2, another layer for data preparation and communication with the EVM microservice is necessary. This layer includes the following set of components:

- **Disassembler**: The disassembler decodes the bytecode of the smart contract to OPCodes.
- **Resolver:** The Resolver uses the disassembled bytecode for pre-resolving storage accesses of variable length values. Since it tries to extract the storage locations from the bytecode it is not a stack based execution. The yellow paper [32] specifications describe how storage is used by stack operations. It performs only a superficial storage resolving of addresses that could be simple variables or the length of a byte array.
- **Mapper:** The task of the mapper is copying the required storage locations from the StateDB to the TransDB and back.
- **Transportation Database:** It is the mobile version of the stateDB. It is sent by the *Execution Handler* to the microservices where it is used as temporary stateDB. On the geth side the TransDB is filled

and read by the *Mapper* while on the microservice side it is directly accessed by the EVM.

- **Execution Handler:** Manages the communication between the geth client and the microservices. It creates a *TrasportPackage* with all the data needed by the EVM for execution and sends it to the microservice API. It also decodes the responses from the microservices.
- **PreExec-Microservice:** This microservice pre-executes the called smart contract function and tracks the storage read and write locations.

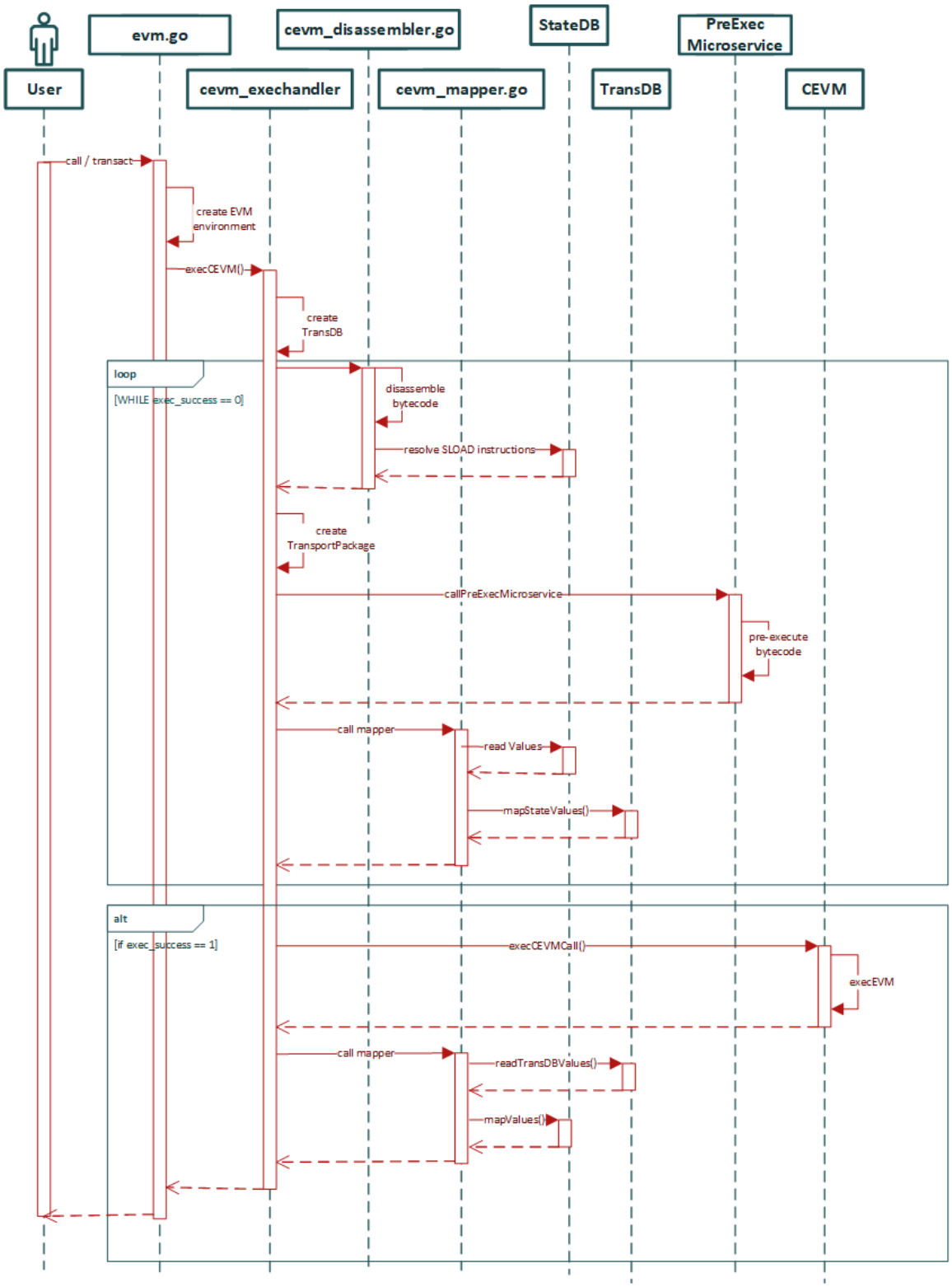


Figure 15: CEVM sequence diagram.

5.2 Reimplementation of the Ethereum node

The starting point of the CEVM implementation is the `core/vm` package of the *go-ethereum (geth)* [9] project. Only minor changes are made on the blockchain code itself. Most of the code is added. The re-implementation mainly affects the gate where the EVM is called. Therefore the `Call()` and `Create()` functions of the `evm.go` file in the `core/vm` package have been changed to pass the data to the *ExecHandler* of the CEVM layer instead of the local EVM as *Figures 15 and 16* show. The *ExecHandler* coordinates the whole dataflow. It triggers the disassembling of the code, executes the storage location resolving, runs the data mapping between TransDB and StateDB, creates the transport packages and finally interacts with the microservices.

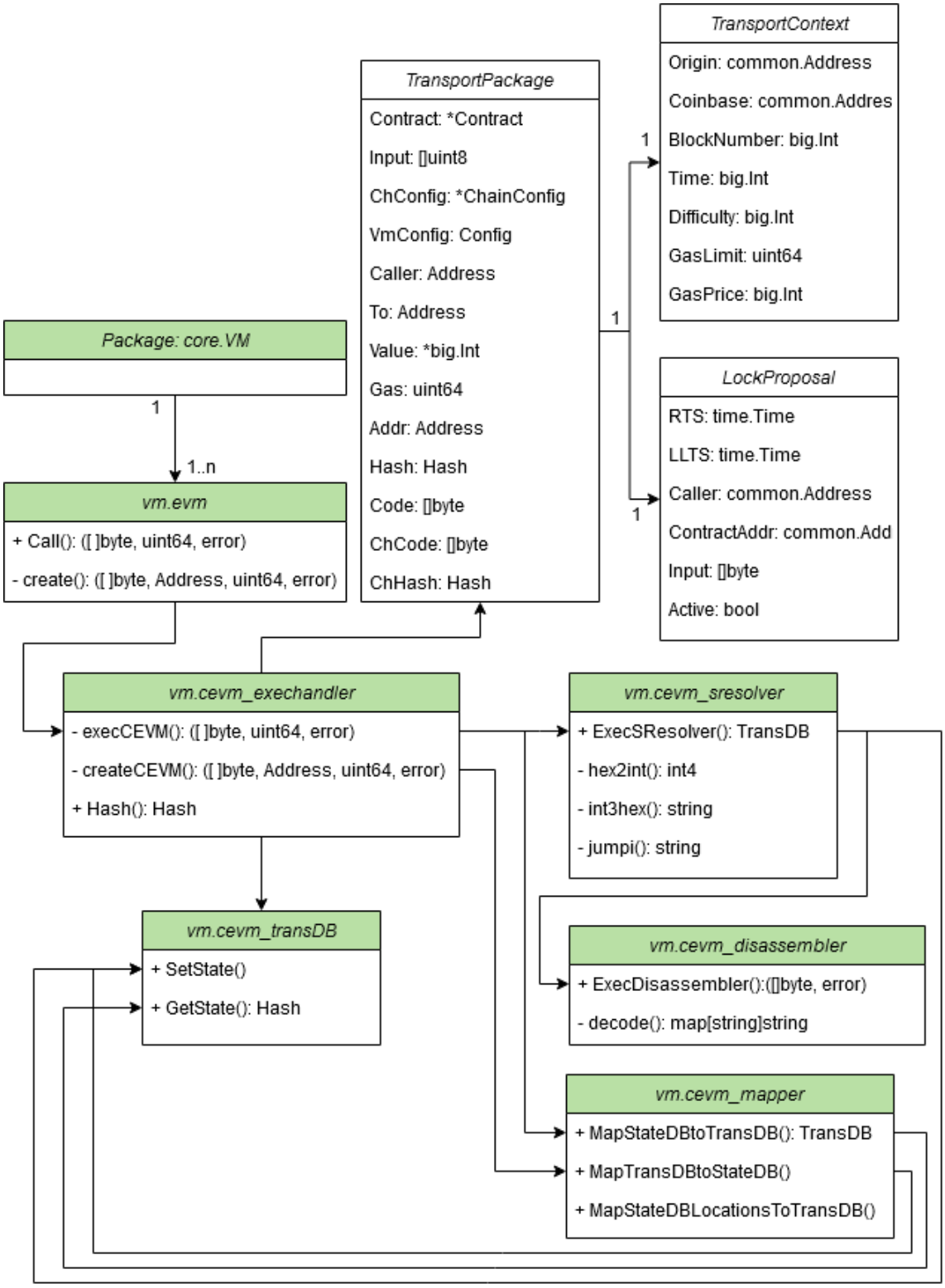


Figure 16: Relations of software components

Figure 16 presents the relations of the components and their function calls. If the Ethereum node receives a new transaction, the corresponding contract bytecode is loaded from the storage. First of all the bytecode as well as all execution relevant information, like contract data or EVM context configurations, are passed to the *ExecHandler*. The *ExecHandler* is the main component in the geth node that collects all relevant data and interacts with the microservices. It triggers all the steps presented in *Figure 15*. Its behaviour is controlled by a permanent state S and aborts the execution if the state deviates. The state S is determined by the microservice response code R.

$$S = \begin{cases} SUCCESS, & \text{if } R = 1 \\ HALTING, & \text{if } R = 0 \\ ABORTING, & \text{if } R > 1 \end{cases} \tag{1}$$

As long as the response code is $R = 1$ the *ExecHandler* continues. When $R = 0$, the *ExecHandler* stops and retries to execute on the microservice again. A deviation occurs if one of the microservice calls is not reaching a successful result state R.

First of all it passes the contract bytecode to the *Disassembler* which decodes the bytecode and returns a list of OPCodes. The decoded program coupled with the input data is then passed to the *Storage Resolver*. At this point also a new instance of the *TransDB* is created. The *Storage Resolver* jumps through the OPCodes and tracks `SLOAD` instructions. It takes the function signature from the input data as a starting point in the program code because only the instructions of the called function are of relevance. It collects all relevant `PUSH` instructions and tries to resolve all possible storage locations when `SLOAD` appears. The resolved locations are then mapped to the *TransDB*. Even though the complete storage resolving process is performed by the *PreExec microservice* afterwards, the task of the *Storage Resolver* is still necessary to provide the length of possible arbitrary length data types to the *PreExec microservice*.

In the next step the original contract bytecode coupled with the input data and the *TransDB* is passed to the *PreExec microservice*. The *PreExec microservice* is the first of the two microservices. Its tasks are on the one hand the tracking of all storage addresses accessed inside the function by simulating the code and on the other hand the pre-executions for gas estimations as

discussed in section 7.2.3. It is a slightly modified version of the EVM that tracks the storage addresses in the *TransDB*. So, it also performs a stack-based execution and is therefore able to count the gas usage of pre-executed smart contracts. Moreover the stack-based execution even allows the resolving of complex storage accesses. As discussed in section 4.1.5, the storage addresses of data types of arbitrary length require a `keccak256` calculation. In case of an array this calculation is dependent on its length value. If the length is not known, the resolving of the indexes of the array would not be possible. A practical example would be a loop based array access. Thus, the *PreExec microservice* is indispensable.
Apart from that the *PreExec microservice* also performs the value locking. For this case it maintains a global list with locks that are set when a `SSTORE` instruction is performed. If the address is already locked, the execution is aborted. As soon as the simulated execution is finished, the *PreExec microservice* returns the *TransDB* to the *ExecHandler*. The received response is a byte array of form:

Response byte array			
Bytes	0 - 3	4 - 8	9 - len(TransDB)
Content	Error state	len(TransDB)	TransDB content

The first 4 bits are containing the state. If the pre-execution succeeded, the state is 1, otherwise 0 or > 0.

State codes	
1	success
0	Locked value error
2	Out of gas error
$>$**2**	Other error

Bytes $4 - 8$ are containing the length of the data transferred and bytes $9 - \textit{end}$ are containing the actual data.
The *ExecHandler* runs the *PreExec microservice* execution in a loop based on the state code. If the state is ≥ 1 the loop finishes and the *ExecHandler* continues or exits at this point. If the loop is equal 0, this means that one

storage value is currently locked and the loop continues with another try until the state is anything else than 0.
In the next step the *TransDB*, which at this point only contains empty storage addresses, is passed to the *Mapper*. Here the values according to the addresses in the *TransDB* are copied from the StateDB.
Finally the updated *TransDB* as well as the original bytecode again coupled with the input data are passed to the CEVM microservice for execution. The CEVM now executes the code with all correct database values. `SSTORE` instructions are written to the *TransDB* instance that is returned to the *ExecHandler* as soon as the execution is finished. Then the *TransDB* is again processed by the *Mapper* which updates the StateDB.

5.3 CEVM Microservice

5.3.1 Overview

CEVM is the original EVM that has been removed from the Etherem node and re-implemented as a microservice. The conversion includes the `core.vm` package of the geth implementation that has been installed in a containerized microservice environment. This way the Ethereum virutal machine is able to run standalone in the cloud.
The implementation is based on Gin [11], a HTTP web framework written in go. Gin provides the web interface for communicating with the system and routing the requests. Behind it, a controller is located that processes the incoming and outgoing data and passes the requests to the according EVM functions. Every request to the microservice creates a new instance of the EVM and so allows parallel EVM execution. The whole microservice is executed in a docker container. This enables a scalable deployment in any cloud environment.

5.3.2 Detailed functionality

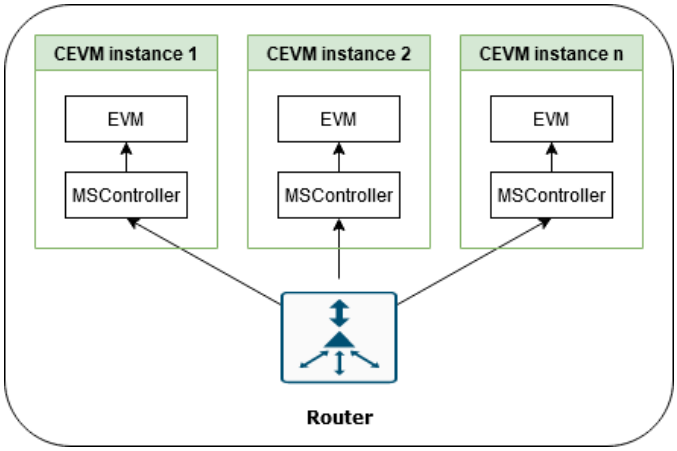


Figure 17: CEVM Router and multiple EVM instances.

The *Gin* web framework provides the API that sends and receives the data from the network. It also provides the router that passes the incoming requests to the according controller functions. With each request a new instance is started.

```
msc := controllers.MSController{}
routes := m.router.Group("/ms")
{
    routes.POST("/call", msc.CevmCall)
    routes.POST("/create", msc.CevmCreate)
}
```

Listing 6: The CEVM microservice routes.

Listing 6 shows the routes the microservice is processing. Here `/call` is used for executing a called function of a smart contract. `/create` executes the relevant code for creating a smart contract. The *MSController* accomplishes multiple tasks. It extracts the package received from the geth node, takes the relevant data for initializing a new EVM and runs it, passes the data for execution to the EVM, packs the execution results and finally returns the response to the geth node.
The received package is a *TransportPackage* object. It contains the context for running the EVM, the bytecode, its function calls and the data of

the *StateDB*. The context information includes the configuration data of the EVM. The data for execution are the sending accounts address, the contract address, the bytecode for execution and the input data. The *StateDB* content in the CEVM is represented by the *TransDB*. For this purpose the EVM has a modified database package where the *TransDB* replaces the *StateDB*. The database logic itself has been retained according to the original *StateDB* implementation. All read and write operations in the EVM are now performed on the *TransDB*. At the point of execution the *TransDB* is already filled with the required state data and during execution it is also updated by the EVM. Afterwards it is returned in the response to the geth node.

5.4 PreExec-Microservice

This is the first of the two microservices and it pre-executes the bytecodes. Its structure is basically the same as the CEVM microservice. It is also a containerized microservice based on Gin [11] web framework and a controller but with a slightly modified version of the EVM.

5.4.1 Detailed functionality and locking

Since the overall implementation is the same as the CEVM microservice, this section mainly concentrates on the differences of the EVM. One task of the *PreExec microservice* is simulating all the storage accesses a smart contract execution performs by running a stack-based execution. Thus, at this point it is also possible to lock all storage locations that will be modified by the CEVM microservice in order to save possible concurrent pre-executions from data inconsistency. This is just an optional functionality to avoid wrong gas estimations of external libraries because of inconsistent data. For further information on this problem, please refer to section 7.2.3.
For the locking functionality the *PreExec microservice* maintains a static *Locker* object that acts as a global black board listing all current blockchain variable lockings. Beside this, its EVM is modified in order to check and set the lockings. The EVM has access to the *Locker* object at instructions-set level and in the instructions-set the `SSTORE` OPCode is extended with locker functionality. When the `SSTORE` instruction is executed, it checks the black board if the respective storage address is already locked and depending on the locking state, it adds the storage location to the black board or aborts the execution at this point if it is already set.

```
1   func opSstore(pc *uint64, interpreter *EVMInterpreter,
    ↪  contract *Contract, memory *Memory, stack *Stack)
    ↪  ([]byte, error) {
2       loc := common.BigToHash(stack.pop())
3       val := stack.pop()
4
5       lockState :=
        ↪  interpreter.locker.GetLock(contract.Address(), loc,
        ↪  contract.LockProposal)
6       if lockState == 0 {
7           interpreter.locker.SetLockTo(contract.Address(),
            ↪  loc, contract.LockProposal)
8       }
9       if lockState == 1 {
10          return nil, fmt.Errorf("Location locked! Aborting!",
            ↪  contract.caller.Address(), " ", loc.String())
11      }
12      if lockState == 2 {
13          fmt.Println("Already locked by same block execution.
            ↪  Continuing...")
14      }
15
16      interpreter.evm.StateDB.SetState(contract.Address(),
        ↪  loc, common.BigToHash(val))
17
18      interpreter.intPool.put(val)
19      return nil, nil
20  }
```

Listing 7: Modified SSTORE instruction

Listing 7 shows the code of the modified `SSTORE` instruction. At line 5 it checks if the storage position of the current contract is listed on the black board. The returned value is the `lockState` which can either be unlocked $lockState = 0$ or locked $lockState > 0$ whereby there are two locked states. Line 7 sets the lock on the black board for the storage location `loc` in the contract of address `contract.Address()`. In line 9, if the `lockState == 1` the EVM aborts the execution. It means that the storage location of the smart contract is already locked by another different execution. `lockState == 2` in line 12 means that storage location is locked by another Ethereum node that runs the same execution. This is not a conflicting state. After consensus is reached on a new block, all full nodes in the network are executing all instructions. The execution ignores the locking and simply continues. In line 16 if the execution reaches this point, the storage location `loc` is ini-

tialized in the *TransDB*. Since the *PreExec microservice* only simulates the smart contract execution without any state values, `val` is 0 at this point.
The EVM at the *PreExec microservice* uses the *TransDB* instead of the *StateDB*. In order to avoid conflicts, the original database logic and naming convention has been retained as shown in line 16 of *Listing 6*. The database instance is still called `StateDB` here although its basis has been replaced by the `TransDB`.

5.4.2 Network reconfiguration

To achieve better results and for an optimal usage of the CEVM concept, the blockchain network requires some basic reconfiguration. As discussed in section 4.2.4, the PoA consensus algorithm qualifies best for usage even on low powered devices. Further, to guarantee a dynamic behaviour of the blockchain regarding the transaction execution, the block generation period is set to 5 seconds. Additionally adjustments in the source code regarding the maximum allowed size of input data and the maximum execution time have been carried out. This allows the processing of large amounts of data.

5.5 Acquainted problems

The execution process causes a certain level of overhead that directly influences the execution time. This overhead has multiple reasons. First of all the pre-resolving of the storage locations runs with a level of uncertainty. Since there is no stack based execution of the bytecode instructions, it is not possible to exactly detect the storage locations of arbitrary length datatypes which are needed for the pre-execution. These storage locations are guessed whereby much more locations are resolved than needed.
Furthermore, the necessary network communications with the microservices are causing a variable latency. Depending on the location of the microservice infrastructure this latency can be higher or lower.
In general all executions on the network are loaded with this overhead.

6 Experimental Analysis

In the following, the performance and behaviour of the CEVM blockchain implementation is tested and analyzed.

6.1 Environmental Setup

For the experiments a test network consisting of 4 devices has been set up:

Server	Intel(R) Xeon(R) CPU E3-1245 V2, 8 cores	32GB RAM
BCSystem 1	Intel(R) Xeon(R) CPU W3520, 8 cores	16GB RAM
BCSystem 2	Intel(R) Core(TM) i7-3770 CPU, 8 cores	8GB RAM
Raspberry B+(R1)	CPU: ARM 700Mhz CPU, 1 core	512MB RAM

S1 is the server, simulating the cloud, which is hosting the *PreExec-Microservice* and the *CEVM-Microservice*. BCSystem 1 and BCSystem 2 are ordinary blockchain nodes. R1 is a Raspberry Pi computer representing an IoT device. Except from the Server, all devices are equipped with the *CEVMGeth* software and the original geth for comparison. The devices are forming a blockchain network of 3 different nodes.
Two private test networks have been configured, both running Proof-Of-Authority as consensus mechanism. The first network is based on *CEVM-Geth*, the second one on v1.9 of the go-ethereum implementation [9] which was the most recent version at this time. Both networks have been operated in succession.
Different test scenarios based on benchmarks for testing the data processing and computation capabilities of the different devices have been developed. The used benchmarks are part of the Blockbench framework introduced in [6].
The first scenario focuses on the computational performance by executing a sorting algorithm. The second scenario measures the data management performance by executing heavy read and write operations on the database. The third test scenario measures the transaction throughput. All tests have been executed on both networks.
For controlling the experiments, a python program called CEVMCommander that interacts with a blockchain node through the Web3 [30] library has been created. This script creates random input values and calls the smart con-

tracts on the blockchain.
The tests were performed with different amounts of input values doubling up to 100000. In order to be able to execute the benchmarks with such a great number of data, some adoptions in the *geth* source code were necessary: The execution limit of 5 seconds has been increased as well as the maximum input size of the Ethereum http API from 5.2MB to 10.4MB.

6.2 Experiments

The *CEVMGeth* performance has been tested in different scenarios with the objective to find its strengths and weaknesses compared to the standard implementation. The execution times, the behaviour of the overall system as well as the behaviour of the different components and their influence on the efficiency have been analyzed. In particular a focus on the latency and communication overhead, the execution itself and data management operations has been set.
The microservice based execution engine approach comes with an unavoidable drawback: overhead T_O. This overhead consists of latency T_L, transmission delay T_N and extra data management effort T_M. Since the system uses two microservices, it is sending and receiving data 2 times which is $4 * T_L$. Latency is a constant that does not change with the calculation task or the amount of data.
The *CEVMGeth* network is able to optimize the execution by skipping the *PreExec-Microservice* step if the executed smart contract code does not require any read or write storage access. With this optimization the latency overhead can be reduced to $2 * T_L$.

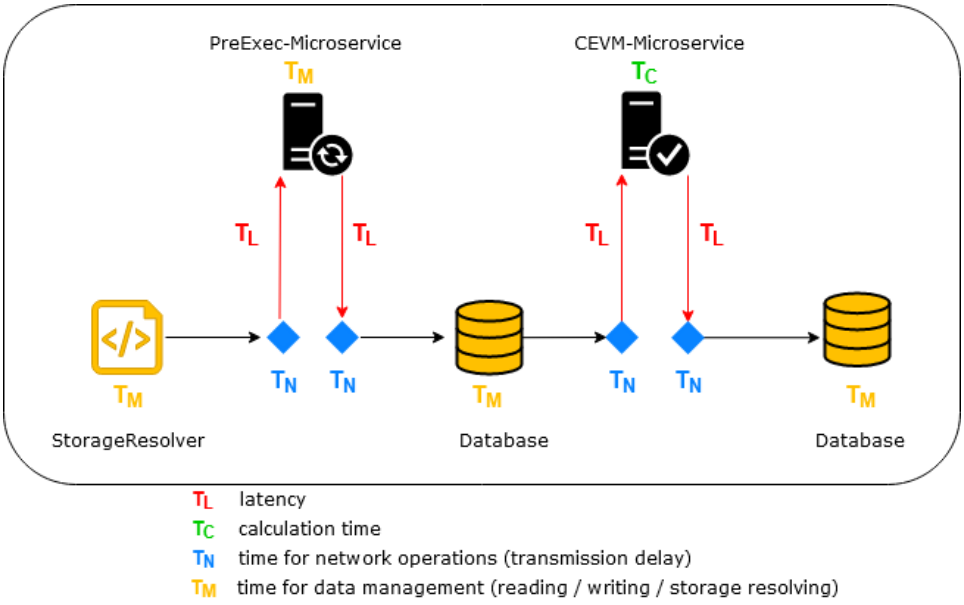


Figure 18: Overhead times and appearance

Figure 18 shows the appearance of the different delays. T_L appears only when communicating with the microservices. T_N stands for the additional network based delays: *processing delay* which is in this case the time it takes the device to encode the data for the network transmission; *transmission delay* which is the time it takes to put the bits on the network. This overhead appears on the node side as well as on the microservice side. T_M includes everything that is needed for accessing data inclusive the storage resolving.

T_N in general is not so important on systems with a powerful hardware but it becomes relevant when running the tests on the Raspberry Pi with low hardware resources. For such a device the network operations are more stressful than for a system with a powerful CPU and thus directly influences the total execution performance. Data management time T_M is heavily dependent on the amount of processed data. It is performed before and after the pre- and main execution. It comprises the time of the storage resolving process, reading and writing data to the database and the time it takes for the *PreExec-Microservice* to pre-execute the code without counting the

latency. The total possible overhead T_O is defined as

$$T_O = 4 * T_L + n * (T_N + T_M) \tag{2}$$

The calculation time T_C is the main execution time of the *CEVM-Microservice*. It only contains the pure execution time without latencies.
T_S is the time needed for database reading and writing. Therefor the overall execution time based on a smart contract with storage access is:

$$T_{\text{CEVMGeth}} = 4 * T_L + W * (T_N + T_M) + W * (T_C + T_S) \tag{3}$$

$$T_{\text{geth}} = W * (T_C + T_S) \tag{4}$$

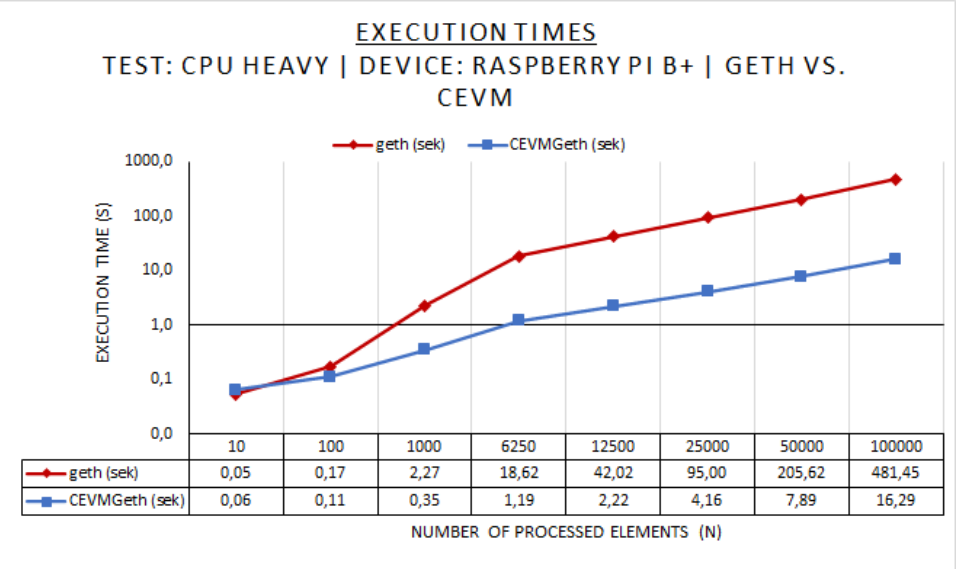
According to the *CEVMGeth* formula, there is still overhead to deal with which is independent of the environment performance. The system needs a certain workload to become efficient. This efficiency is defined by the efficiency threshold E. It defines the work W needed in the *CEVMGeth* system to be as efficient as the original blockchain implementation: $T_{\text{CEVMGeth}} = T_{\text{geth}}$.

The following experiments will, among others, determine the impact which the overheads are having on the different types of devices.

6.2.1 CPUHeavy executon on Raspberry Pi

Execution times				
n	ping (ms)	CEVM (ms)	CEVMGeth (s)	geth (s)
10	25,1	0,368	0,06	0,05
100	24,9	6,68	0,11	0,17
1000	24,8	48,05	0,35	2,27
6250	25,0	351,9	1,19	18,62
12500	25,0	731,09	2,22	42,02
25000	24,8	1543	4,16	95,00
50000	24,9	3259	7,89	205,62
100000	25,1	6795	16,29	481,45

In this test scenario the Raspberry Pi is executing a cpu heavy benchmark once on the original *geth* implementation and once on the *CEVMGeth* implementation. The CPU heavy benchmark is part of the Blockbench framework introduced in [6]. This benchmark is a smart contract with an implementation of the quicksort algorithm that sorts input values in ascending order. Beside this the benchmark only works with input data and performs no database accesses. This allows the *CEVMGeth* to run in optimized mode which spares the pre-execution on the *PreExec-Microservice*. The execution times have been measured based on different amounts of input values. Starting with low values from $n = 10$ up to $n = 6250$ and then doubled stepwise up to $n = 100000$.



	10	100	1000	6250	12500	25000	50000	100000
geth (sek)	0,05	0,17	2,27	18,62	42,02	95,00	205,62	481,45
CEVMGeth (sek)	0,06	0,11	0,35	1,19	2,22	4,16	7,89	16,29

Figure 19: CPUHeavy execution. Geth vs. CEVMGeth on Raspberry Pi B+

Figure 19 presents the results of CPUHeavy execution on the Raspberry Pi. Considering the overall result, it shows that the *CEVMGeth* is mostly faster than the *geth*. Only when sorting 10 values, it takes 0,05s on *geth* and 0,06s on *CEVMGeth*. The reason for this is the overhead arising during the execution dataflow mainly caused by the microservice communication. The *CEVMGeth* based test network basically has a latency of $\sim 12,5ms$ oneway to the microservice which results in a total latency overhead of $T_L \sim 25ms$. When executing the benchmark with a workload of 100 values,

the threshold by which the *CEVMGeth* overhead slows down the implementation has already been exceeded. For sorting 100 values the *CEVMGeth* execution takes 0,11s which is already faster than the *geth* execution with 0,17s. With a growing number of input values the *geth* execution time grows strongly by seconds while the *CEVMGeth* time grows slowly by milliseconds. Running the algorithm with 6250 values already takes 18,62s on *geth* and 1,19s on *CEVMGeth*. Finally for sorting 100000 values, the Raspberry Pi needs 481,45s on *geth* and 16,29s on *CEVMGeth* which is about 30 times the time the execution takes on *geth* than on *CEVMGeth*. As the execution times table shows, the total execution on *CEVMGeth* at a workload of $n = 10$ is slower than the execution on *geth*. But *CEVMGeth* has already caught up with a workload of $n = 100$ values.

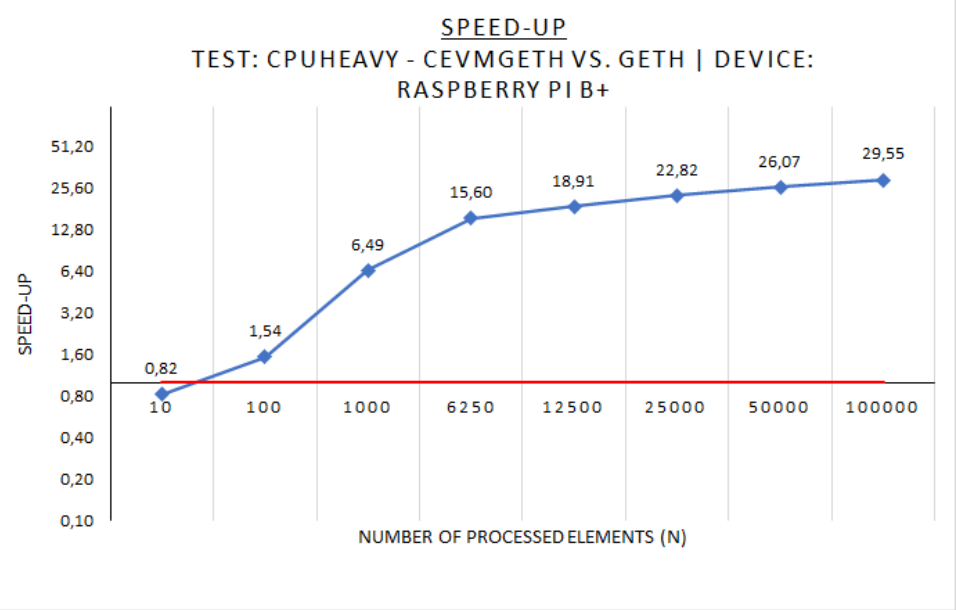


Figure 20: CPUHeavy speedup of the CEVMGeth vs. Geth on Raspberry Pi B+

This clearly reflects the speed-up curve in *Figure 20*. The red line represents the efficiency threshold E at which both networks are equally fast. With 10 input values, the *CEVMGeth* has a speed-up of 0,82. The threshold is reached at an input work of $n \sim 36$. With a workload of $n = 1000$ the speedup is already 6,49. From $n = 6250$ on the workload doubles and the speedup grows from now on by an average of 3,49. The speed-up grows

equally which means that the overhead as well as the calculation times are growing equally. Finally the sorting of 100000 values is about 30 times the execution time on *CEVMGeth*.

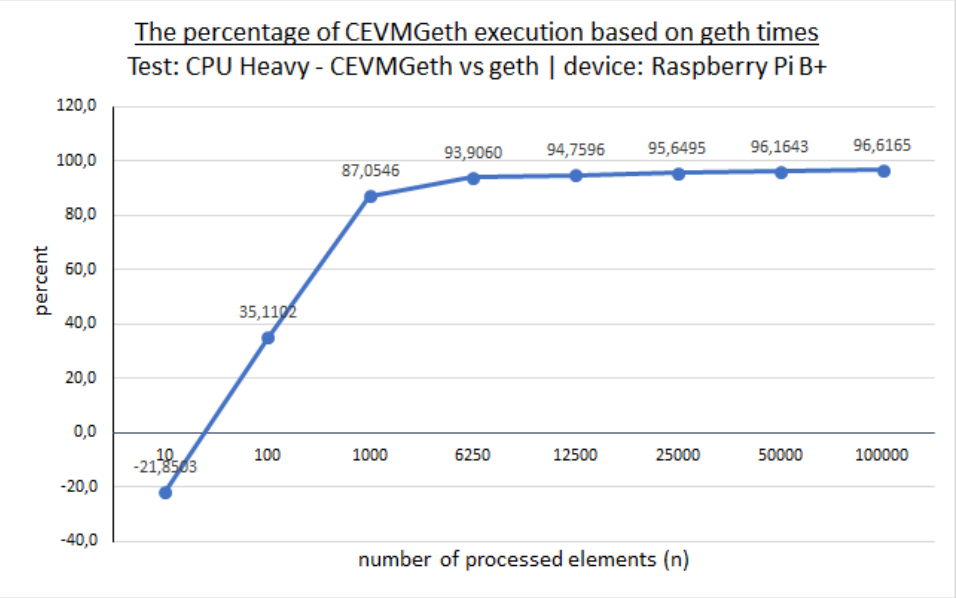
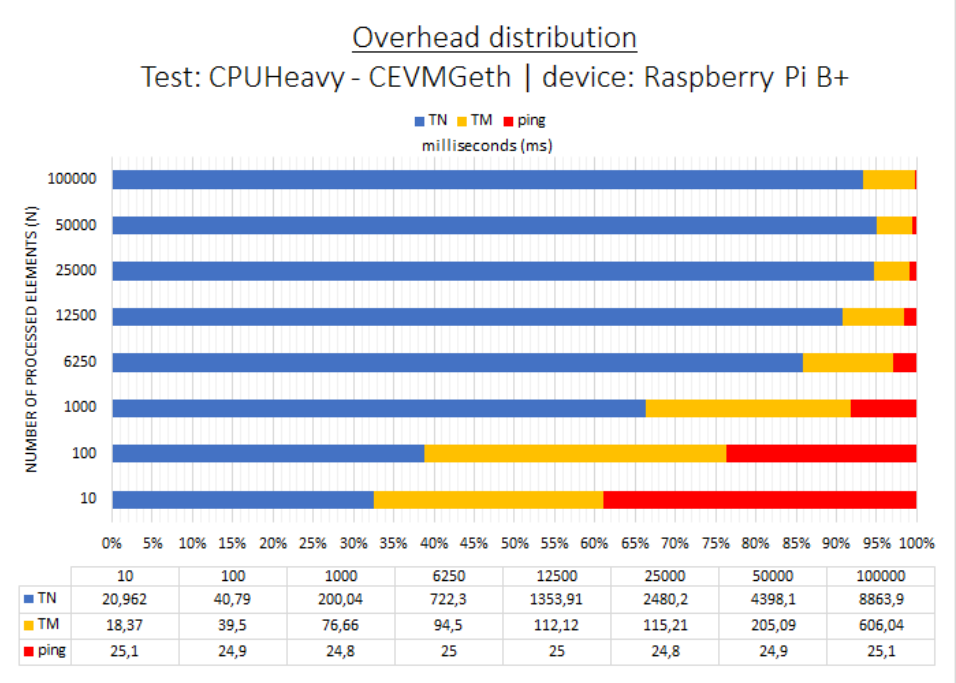


Figure 21: CPUHeavy percentage of CEVMGeth execution based on Geth times

Figure 21 presents the percentage of the *CEVMGeth* execution times based on the *geth* times. The x-axis with $0,0\%$ corresponds to the standard *geth* execution. The positive area shows how many percent the execution time on *CEVMGeth* is faster whereas the negative area shows how many percent it is slower with respect to the workload n. This analysis gives information about possible disproportional growing loads in the *CEVMGeth*. With $n = 10$, *CEVMGeth* takes $\sim 21,9\%$ more time for execution than *geth*. At $n = 100$ the running time is already $\sim 35,1\%$ faster. Thus, with a workload of $n = 100$ the efficiency threshold where both systems are equally fast has already been exceeded. *CEVMGeth* still accelerates up to $n = 1000$ where its execution time is 87,1% faster than *geth* times. Then the curve flattens out and with $n = 6250$ it only grows in the single-digit range.
The reason for the behavior of this curve is the overhead that grows according to the input work. With $n = 10$ and $n = 100$ there are less values in relation to the other workloads which means less overhead caused by sending

and receiving data and moreover a fast calculation in the cloud. Then the work size increases and with it the network based overhead T_N and the data management effort T_M. The curve approximates to $\sim 97\%$ which means that the workload dependent overheads have reached their maximum extra load. This further means that the execution of *CEVMGeth* will be at most $\sim 97\%$ faster than the *geth* execution. Moreover, since the curve does not decrease at a certain workload, the system has no disproportional growing overheads which means that there are no components that are causing unproportional load with a growing number of workload. A more detailed representation regarding the development of the different overhead types during the execution is presented in the following.



	10	100	1000	6250	12500	25000	50000	100000
TN	20,962	40,79	200,04	722,3	1353,91	2480,2	4398,1	8863,9
TM	18,37	39,5	76,66	94,5	112,12	115,21	205,09	606,04
ping	25,1	24,9	24,8	25	25	24,8	24,9	25,1

Figure 22: CPUHeavy overhead distribution

Figure 22 represents the overhead distribution according to the processed values. The blue bar shows the network based delay T_N. The yellow bar presents the data management effort T_M and the red bar is the latency T_L. During the execution with workload $n = 10$, T_L is the dominating fraction. There is less data management effort and less data up- and download. With the growing workload, the network based and the data management over-

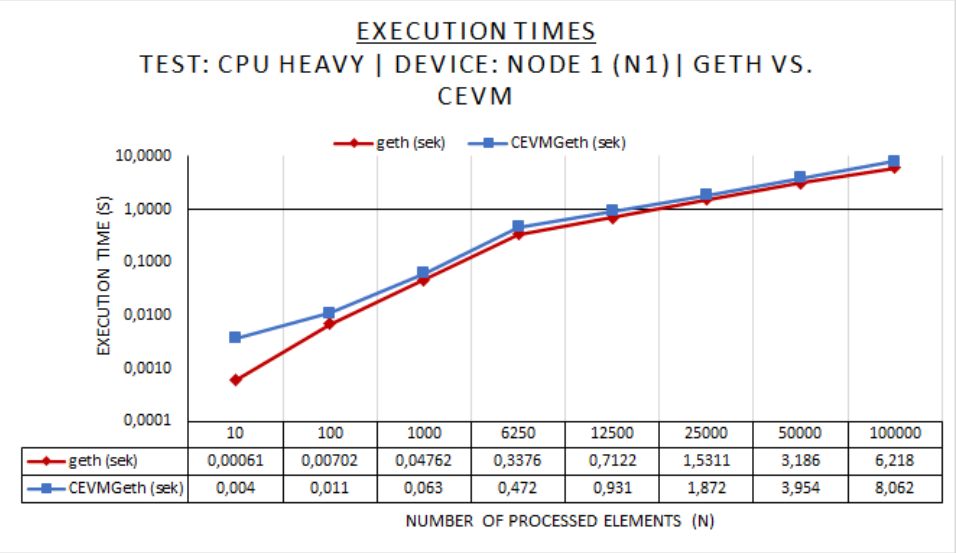
heads are growing. The latency stays nearly constant across all executions. From $n = 6250$ on, the percentage of T_N remains almost the same by taking up between $\sim 85\%$ and $\sim 95\%$ across the remaining workloads. With the execution of $n = 100000$, latency is the smallest delaying factor of the total overhead whereas T_N takes $\sim 93\%$.

6.2.2 CPUHeavy execution on server:

In the second CPUHeavy scenario, the benchmark has been executed again on *CEVMGeth* and *geth* running on an average powered server system: BC-System 1. The purpose of this test is to get the overall performance of the *CEVMGeth* without reduced hardware capacities.

Execution times				
n	ping (ms)	CEVM (ms)	CEVMGeth (s)	geth (s)
10	1,01	0,523	0,004	0,00061
100	0,92	6,59	0,011	0,00702
1000	1,12	46,7	0,063	0,04762
6250	1,21	348,77	0,472	0,3376
12500	1,04	731,16	0,931	0,7122
25000	0,89	1543	1,872	1,5311
50000	0,96	3271	3,954	3,186
100000	1,0	6781	8,062	6,218

The timetable contains the results of *CEVMGeth* and the original *geth* implementation. In this scenario also the pure CEVM microservice execution, the ping, the total *CEVMGeth* execution and the total *geth* execution has been measured. The total workloads are again $n = 10$ up to $n = 100000$ input values. The round-trip-time between BCSystem 1 and Server averages $1,0ms$. Furthermore *CEVMGeth* is again able to execute the benchmark in optimized mode since no storage locations for read or write operations are accessed in this benchmark.



	10	100	1000	6250	12500	25000	50000	100000
geth (sek)	0,00061	0,00702	0,04762	0,3376	0,7122	1,5311	3,186	6,218
CEVMGeth (sek)	0,004	0,011	0,063	0,472	0,931	1,872	3,954	8,062

Figure 23: CPUHeavy execution on BCSystem 1.

Figure 23 presents the execution times of BCSystem 1. Within all tests with input workload from $n = 10$ to $n = 100000$, the original *geth* implementation is throughout faster in sorting the values than the *CEVMGeth* implementation.
The reasons for this are the additional time needed for disassembling the bytecode, processing its opcodes and communicating with the CEVM microservice. The greatest percentage of the difference in the execution times is mainly caused by the communication overhead T_N between *CEVMGeth* and the microservice. The overheads altogether are having the greatest impact at a workload of $n = 10$. This is illustrated in *Figure 23* which clearly presents that the execution difference is the highest with the lowest values. Thus, with a workload of $n = 10$ the communication and especially the latency T_L are having the greatest braking effect. But T_L is a static overhead that does not change. Compared with the *geth* times where the execution runs locally on the same system, the *CEVMGeth* takes $\sim 516\%$ more time in running the benchmark as *Figure 24* shows. With a growing workload the calculation times as well as the other overheads T_N and T_M are growing and suppressing the static overhead T_L.

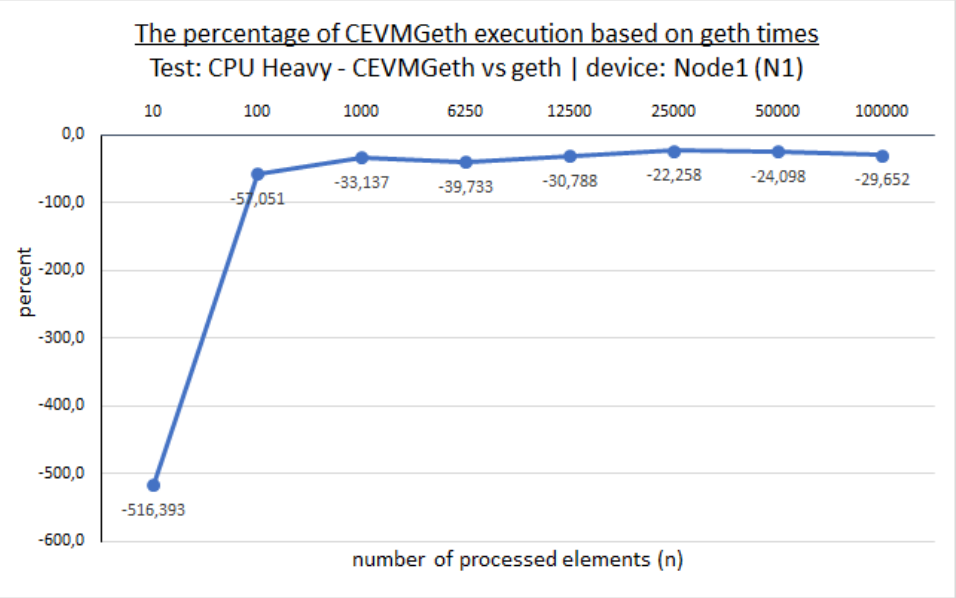


Figure 24: CPUHeavy percentage of CEVMGeth execution based on Geth times (BCSystem 1)

Figure 24 presents the percentage of *CEVMGeth* execution time based on *geth* times on device BCSystem 1. The x-axis represents the *geth* execution level. The negative area represents how much *CEVMGeth* is slower and the positive area how much *CEVMGeth* would be faster than *geth*.
Here *CEVMGeth* is throughout slower in executing the benchmark whereas the worst value is reached at a workload of $n = 10$ where the runtime is $\sim 516\%$ more than the runtime of *geth*. The best value is reached with a workload of $n = 25000$ where *CEVMGeth* takes $\sim 22,3\%$ more time. From $n = 1000$ on, the percentage values are approximately in the same range.
Another thing that can be observed is the behaviour of the CEVM microservice.
Comparing *CEVMGeth* column and the *geth* column of the time table, it shows that the total geth time is even lower than the CEVMgeth execution. The reason behind this delay is again the network caused overhead on the microservice side. The received *TransportPackage* needs to be decoded and the results need to be encoded and returned.

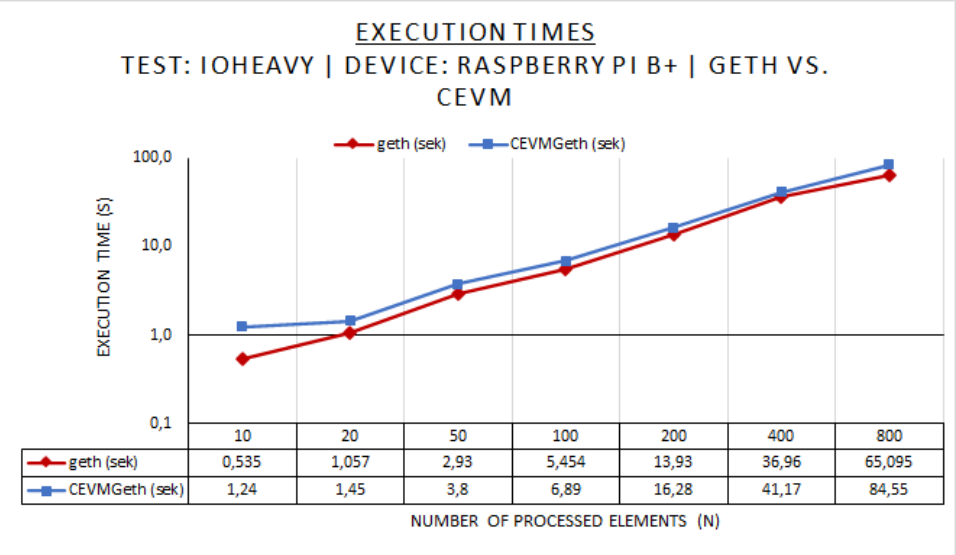
The overall execution of the *CEVMGeth* approach on a usual computer system is slower by design than the original geth implementation. *CEVM-*

Geth has to communicate with the cloud for execution whereas *geth* can execute the code locally. Except from $n = 10$ where the *geth* execution time is in the range of nanoseconds, all executions up to $n = 12500$ are times of the milliseconds range. The times from $n = 25000$ to $n = 100000$ are in the seconds range.
Thus, since the network environment causes at least a latency in the area of milliseconds, the difference of times is the highest with $n = 10$ which makes $\sim 516\%$. The time difference with $n = 100$ makes $\sim 57\%$ and from $n = 1000$ on all executions up to $n = 100000$ are having a percentual difference on an average of 30%.
The results show that even with a growing workload, the application scales evenly. Different from the test scenario with a low powered IoT device in *section 6.2.1*, the network based overheads in this test scenario are not growing to the same extent. This implicates that with enough CPU calculation capabilities, the network overheads are less significant.

6.2.3 IOHeavy execution

In this test scenario an IO heavy benchmark has been executed on the Raspberry Pi running the original geth implementation and the CEVM implementation. As well as the CPUHeavy benchmark, the IOHeavy benchmark is also a smart contract part of the Blockbench framework [6]. But for this application it has been slightly modified according to the test requirements. This benchmark performs heavy read and write operations on the database by filling up a mapping key-value datatype with bytes. Our modified version of the contract stores n times 32-bytes arrays of length 100. This makes $n*(100*32)$ bytes where n is the number of mappings. Each mapping stores $100 * 32 = 3.2kB$. The testcases have been performed with workloads from $n = 10$ up to $n = 800$ which amounts to a total key-value storage volume of $2.56MB$.
The original benchmark test uses events as a programming feature of smart contracts for determining the processing time. Since the implementation of the CEVM approach also intervened fundamental functionalities of the blockchain system, the overall functionality and performance of the system running real-world scenarios and not just test cases is interesting. Therefore, an external test program has been created that interacts with the blockchain client per RPC and triggers the execution. So, the test measures the per-

formance of the whole system and not just of the EVM. Furthermore it is possible to check if the writing operations are performed successfully at the correct locations.



	10	20	50	100	200	400	800
geth (sek)	0,535	1,057	2,93	5,454	13,93	36,96	65,095
CEVMGeth (sek)	1,24	1,45	3,8	6,89	16,28	41,17	84,55

Figure 25: IOHeavy execution on Raspberry Pi B+

Figure 25 shows the execution curves of the IOHeavy benchmark executed on the CEVM implementation and the original geth implementation. Both curves are moving almost equally. The percentual differences are throughout between 11% and 37% except from $n = 10$. Just as in the previous experiments, the greatest difference has been measured with the lowest workload. The same behaviour can be observed in this test scenario. With workload $n = 10$ the execution times are having a difference of $\sim 57\%$. Here again the part of the network based overhead has an enormous weight on the total execution. From $n = 100$ on the execution proceeds almost equally. With $n = 800$ the difference between the execution times is about 30% as presented in *Figure 26*.
In this test the data to be stored is generated during execution and directly written to the storage. This means for the CEVM system that the values are generated at the CEVM microservice, stored at the temporary *TransDB* and then transported back to the CEVMGeth node where they are finally mapped to the real database. The mapping process of the CEVMGeth node

is the same process that happens in the original geth implementation already during the EVM execution and consequently causes no extra delay. The main delaying factors in this experiment are only the network based overheads T_N.

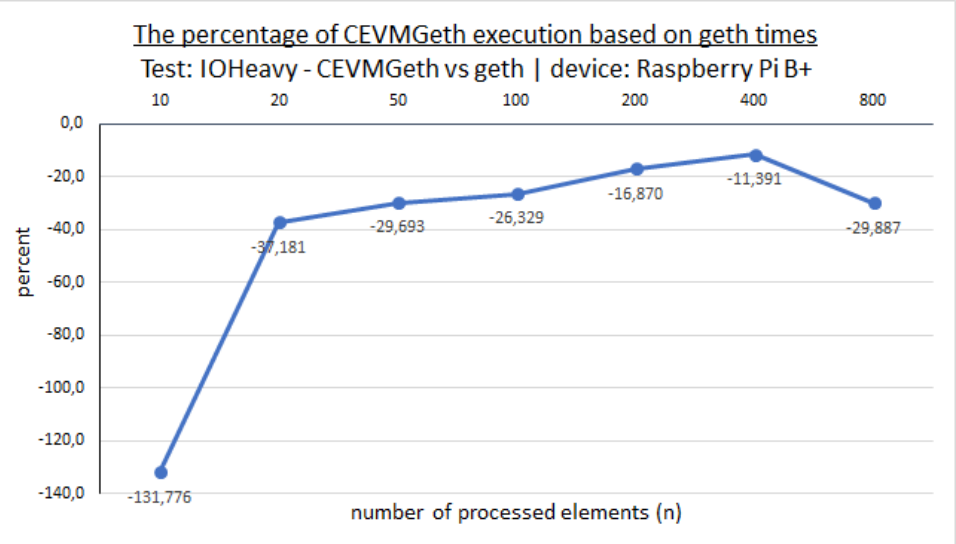


Figure 26: IOHeavy percentage of CEVMGeth execution based on Geth times (Raspberry Pi)

6.2.4 Transactions per second

In this scenario the amount of transactions (Tx) are measured that can be processed by the network per second. Again the CEVM network has been compared with the geth network and the IoT device with a server system. To unload the IoT device from the expense of the signing task caused by the Proof-of-Authority consensus mechanism, the block generation period has been set to 5 seconds.

For this exercise a python script has been created that starts up as many concurrent threads as transactions to be triggered. Thus, all transactions are passed to the blockchain node almost concurrently and then executed consecutively. Afterwards the python script counts the blocknumbers of all transaction receipts. This way the number of transactions per block are counted and results in the successfully processed transactions during the 5 seconds block generation time. Finally it has been calculated down to one second.

The smart contract function used here does a simple process-and-store task. It takes the input value from the transaction, multiplies it with another already stored value and finally pushes it into an array in the storage.

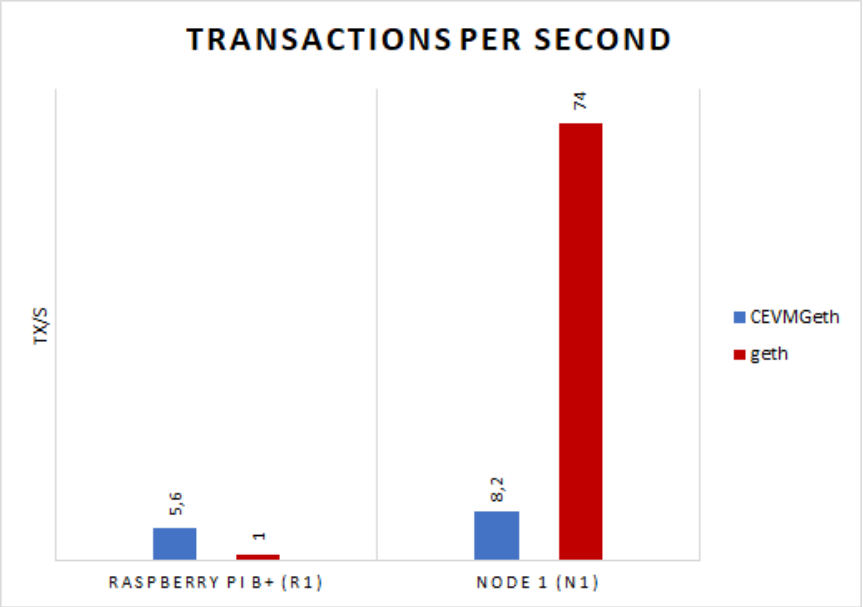


Figure 27: Transactions per second Raspberry Pi vs. BCSystem 1

Figure 27 shows the amount of transactions of the IoT device compared to the BCSystem 1. Here the IoT device creates 5,6 transactions per second whereas the server system has a rate of 8,2 tx/s.
Although the server system has a multiple of the hardware resources the IoT device has, it is only $\sim 32\%$ faster. On the other side measuring the amount of transactions with the geth network, the IoT device creates <1 tx/s and the server system 74 tx/s. To be more concrete the IoT device creates about 0,75 tx/s. The calculation is based on the execution time of one transaction which is 1.244 seconds. Only the server system is faster in this scenario because it runs the executions locally and has no extra communication and data-management overhead.
This test clearly shows the positive effect regarding the efficiency the CEVM implementation has on an IoT device.

6.3 Implication

Different experiments have been conducted to measure the behaviour of the presented blockchain approach with microservice based execution logic. The experiments are more adjusted to measure the data processing and calculation capabilities of the blockchain which are rather used in the area of IoT than in common blockchain applications like crypto currencies. The created tests are scenarios which an IoT device is more likely to be confronted with like executing an algorithm on a great amount of data or reading and writing big amounts of data.

The experimental results accurately represent the strengths and weaknesses of our redesigned blockchain. First of all the usability of the blockchain with outsourced execution logic depends on the area of application, the tasks executed and the hardware used. Applications that are dependent on fast response times of the blockchain system are less suitable for this approach because of the unavoidable overhead that originates from preparing data and communicating with the microservices. Even simple procedures take at least the overhead of time which is a great disadvantage. Especially low powered devices are suffering from extra overhead which has a grater impact on them. On the one hand there is more network data processing effort that leads to extra delays and on the other hand there is the extra code processing which is unavoidable by design. Hence data read and write operations are not the strengths of the redesigned blockchain.

The strengths of the system are revealed with the complexity of tasks and the amounts of data. Performing calculations in the cloud can speedup the execution times enormously and additionally effects the number of processed transactions. But the system must exceed a certain complexity threshold to be efficient. This threshold is the data complexity or effort respectively, needed to be faster than the same execution on the original Ethereum implementation. The threshold mainly depends on the speed of the network infrastructure which is usually faster on local area networks than on internet networks. The experiments have shown that with a standard internet network with a latency of 25ms, the CEVM execution with a workload of about ~ 36 input values running on an IoT device is already as fast as the original geth execution on the same device. Improving the network connection would lower the efficiency threshold which would speedup the CEVM system even more.

To sum up, although the system causes a lot of overhead it still runs very smoothly. Only because it becomes efficient when it reaches the efficiency threshold does not automatically mean that it is unusable slow when it is below this threshold. Here the execution times are still in the area of milliseconds. According to the experiments and the strengths of IoT devices, the CEVM system processes about 5 times more transactions per second and performs calculation tasks of 100000 values about 30 times faster than the geth implementation. With this performances the CEVM blockchain is believed to at least meet the requirements of current applications or might even accelerate established systems.

7 Discussion

The following section discusses the decentralization, some drawbacks as well as the advantages resulting from the approach presented above and its possible impacts on current and future technologies.

7.1 The centralized influence

A central question that arises is whether the presented approach influences the decentralization as a central principle of the blockchain. The strength of a blockchain is the fact that it can operate on a group of distributed nodes without any trusted third party. The approach presented above moves the execution engine to the cloud which is a centralized infrastructure. To answer the question how the centralized microservice influences the decentralized principle, the field of microservice functions must be examined in more detail.
First of all, as already discussed in section 2.7, microservices are autonomous components. The purpose of the microservices in the presented approach is solely the execution of smart contract bytecode. The network outsources the calculation processes to another environment so that the microservices neither undertake any coordinating, encrypting or transacting task nor any data management task. All four basic blockchain concepts such as distributed ledger, consensus, cryptography and smart contracts are preserved. Therefore, no third party takes over any central blockchain procedure.
Moreover, when applying the CAP theorem properties for distributed systems, as described in section 2.5, the CEVM approach meets the criteria by fulfilling at most two of the three CAP properties. Since the main blockchain operations like communication, data exchange and data management as well as interaction with external applications or users are still performed by the blockchain nodes themselves, the behavior regarding *Consistency*, *Availability* or *Partition tolerance* compared to the original Ethereum blockchain remains unaffected. To sum up, the usage of external computing capabilities in the CEVM blockchain has no influence on the decentralization of the blockchain network.

7.2 Problems with the presented approach

7.2.1 Sequential execution of transactions

The CEVM approach enables the blockchain-based execution of complex programs or algorithms. The execution environment is constructed to run as a container and can be deployed into the cloud, which enables performant parallel execution. But triggering multiple transactions which perform changes on the blockchain state is executed strictly sequentially. A blockchain node receives the transactions, queues them and executes one after another. Only after the first transaction has performed changes on the state, the second transaction can be triggered and so the data modification order is guaranteed, which is a critical point. But on the CEVM blockchain, which does not primarily limit the execution time, this might lead to extra idle time of the subsequent transactions until they are processed.
While working on the concept of the CEVM blockchain, there was also the plan of implementing a functionality for running multiple transactions in parallel. In theory, this would perfectly fit with the microservice-based execution logic and could make optimal use of the cloud resources, which in the following would increase the amount of processed transactions on the network. The idea was to create a new ordering service that operates based on modified transaction nonces and the storage access information it gets from the resolver module of every transaction. Also, basic practical experiments have been conducted but only 2 out of 10 blocks could be successfully created with parallel executed transactions. Much more research and investigation is needed to be able to successfully conduct blockchain transactions in parallel. This, however, would exceed the context of this thesis.

7.2.2 Uncertainess of resolver

The resolver is necessary for a pre-storage resolving. It is triggered at an unfortunate place by design. On the one hand, access to the StateDB at this point is possible but no stack-based execution engine is available. On the other hand, a stack-based execution is necessary to resolve all required storage locations. Executing the pure bytecode without data to get all storage locations does not work. Ethereum uses hashed storage locations for some datatypes and these hashes are again calculated during the stack-based execution. Storage resolving by searching for storage locations in the bytecode

is not enough but it is a necessary step for the *PreExec Microservice* to be able to resolve dynamic data types.
Thus, the resolver runs through the bytecode of the called function and resolves all `PUSH1` instructions as storage locations. Many `PUSH1` instructions, however, are only standard values pushed on the stack for operations which are not a specific storage location. The resolver cannot distinguish between standard values or storage locations, which gives the resolver a level of uncertainty.
As a consequence, the resolver passes a prepared *TransDB* to the *PreExec Microservice*, which beside correct storage locations might also contain unnecessary values. Yet the *PreExec Microservice* performs a stack-based execution and so determines only correct storage locations of the *TransDB* and ignores the others.
For smart contracts with simple storage constructs where variables and mappings are directly accessible, the resolver is only necessary for array datatypes. In programs with complex storage constructs like key-value mappings combined with arrays, the resolver needs to resolve the key position of the mapping first to receive the size of the array afterwards.

7.2.3 Data consistency during concurrent execution

Ethereum itself is often used by external applications, which apply the blockchain as a distributed and immutable database. Such applications interact with the Ethereum nodes either manually per RPC or with the help of external libraries like Web3 [30]. Web3 relays the whole communication between an end-user or software and the Ethereum blockchain.
When a user or a software wants to interact with a smart contract, gas must be provided. Usually the required gas is not known beforehand, which is why it must be estimated. The Ethereum client offers an interface for estimating gas consumption of a transaction by simulating the smart contract execution. The simulation is like a real execution without performing state changes on the blockchain. It can be executed at any time and is not dependent on consensus or block generation. The gas estimation performing multiple rounds of execution simulations, determines the gas consumption based on the OPCode instructions and the current storage state.

The CEVM blockchain with its microservice interactions has a larger execution circle. At the CEVM system, the gas estimation process is mainly

executed on the *PreExec microservice* since the system aborts when the first microservice throws an *out-of-gas* error. Since the overall execution of one simulation requires more time than on the standard *geth* implementation and often multiple rounds of estimations are necessary, this process takes up much more time for the end application.
Furthermore, it might lead to incorrectly estimated gas because of a greater probability of changed storage values in the meantime. Insufficient gas leads to *out-of-gas* abortion error, which has a greater probability at the CEVM system.

7.3 Possible impact on future technologies

The CEVM approach links the principles of decentralization and cloud computing. The blockchain technology provides the concept of a network where computer systems can autonomously exchange data and come to an agreement. Cloud computing is used for performing computations without resource limitations. Hence the technology enables the creation and running of calculationally intensive applications with data of a distributed database provided by a network of theoretically unlimited nodes. Furthermore, by outsourcing the execution process, the CEVM approach brings the blockchain technology much closer to IoT technology. Making IoT compatible with blockchain technology qualifies it for the application in current and future technologies like smart homes, smart cities or industry 4.0.

By outsourcing calculations into the cloud, the CEVM blockchain uses the sum of bundled cloud resources and in this way unloads local devices. It enables access to a centralized pool of high-performance systems instead of a network of only medium to low performance nodes.
The idea of outsourced calculations has already been applied in the areas of cloud computing and cloud gaming. Especially cloud gaming, which has only existed for the last few years, bundles high-performance resources and offers calculation time. All graphical processes are executed in the cloud and streamed to the end-user so that no expensive hardware is needed. With this technology, it is possible to use the highest graphical standards on every personal computer with a fast enough internet connection.
In cloud gaming, outsourced calculation reduces the necessity of investing in expensive hardware - the same holds true for future blockchain systems. With the CEVM blockchain, smart contract calculations and corresponding

results can be streamed to and from the cloud, which could have an impact on the development of future blockchain networks. In the future, this advantage might have an effect on the financial aspects behind the design of distributed systems and again also on Industry 4.0.

The latter, also called the fourth industrial revolution, describes a trend in manufacturing technologies towards automation and data exchange. Due to a better networking of IT systems, machines, controlling devices and sensors, they can be controlled in a decentralized way. The systems in such a highly integrated network cooperate by autonomously communicating and behaving as smart devices to achieve the common goal [1, p.2]. In Industry 4.0, the fields of application for IoT are wide-spread. The electricity industry, for example, has been transformed over the last few years with new sources of power generation on the one hand and new technologies ranging from smart phones to electric cars with different power demands on the other hand. Blockchain can be a tool to accelerate this global energy transformation by providing new ways of connecting components and devices and thus lowering the transaction costs and operating the grid in a more efficient way [1]. In fact, smart power grids already enable peer-to-peer electricity trading.
Another field of application that can be optimized by using a blockchain are supply chains and logistics. Exchanges and transactions are tracked in a decentralized database. This enables better traceability of products, better product management and helps solving logistic inefficiencies. It further makes the blockchain suitable for application in healthcare systems or the agricultural industry.

Furthermore, the CEVM approach may also have a positive impact on another problematic field of blockchain technology: energy consumption. Although the blockchain technology is a young technology, it is already the source of enormous energy consumption world-wide. The largest blockchains are running crypto currencies. Those currencies are powered by miners worldwide. Many of them are located in so-called mining farms, which are data centers for blockchains. The miners' energy consumption already influences the stability of power grids in certain regions of the world. The impact of crypto currencies on the stability of power grids is discussed in [27]. Each of the miners has a modest amount of energy consumption. The energy consumption of mining farms containing thousands of mining devices, however, add up to a large volume. Still, it is not primarily the amount of energy

consumed that causes problems, but possible abrupt changes in the volume. Power grids are designed to keep an equilibrium between energy supply and consumption. If sudden changes, like a software bug or an attack that forces the network to stop operating, appear in such enormous blockchain networks, the abrupt change in energy consumption can lead to a destabilization of the power grid and subsequently cause an emergency shutdown.
The experiments in section 6.2 have shown that the CEVM blockchain implementation runs smoothly even on a Raspberry Pi with single core CPU. The energy consumption of a Raspberry Pi B+ amounts to ~ 2 Watts whereas a middle-rate Ethereum mining hardware consumes $\sim 350 - 500$ Watts. Thus, by using the CEVM blockchain approach and adopting more efficient consensus mechanisms, future blockchain networks could be based on low-power devices that consume only a fraction of the energy as compared to current systems. This again could reduce the world-wide energy consumption of blockchain systems and avoid destabilization of power grids.

Another example and field of application where the CEVM approach may have positive impact are data tracking systems - projects that track unique identities of digital properties, to be more specific. The blockchain with its encryption and immutable storage enables a secure linkage of ownership and copyrights with the digital property. Such a solution for detecting tampered images is presented in [16]. In this project, unique descriptive metadata of images are linked with the ownership. Solutions like this are heavily reliant on external systems that perform the data operations and data management on the blockchain. The blockchain itself is often only the immutable database due to a lack in data processing capabilities. As shown in [6], current blockchains are not anywhere near conventional database systems regarding their data processing capabilities. This is mainly because of their restricted execution environments in time and gas to guarantee an even execution on all participating devices of different hardware resources. With the CEVM blockchain, the underlying hardware of the participating nodes in the network has no influence on the performance of the overall blockchain anymore. It is now possible to create smart contracts that provide data processing functionalities like search or sorting algorithms. This gives more power to the blockchain itself and reduces the dependency on external systems performing the data management for the blockchain. Related to tracking projects, this would allow blockchains to check themselves for already existing data

instead of external systems executing this task.

Coming back to IoT and sensor network devices, a wide field of application for the CEVM blockchain is offered by the the area of Smart City installations. This area comprises smart homes, intelligent transport and traffic systems, smart energy management in power girds, vehicular communication, smart health services and much more. With the current expansion of the 5G network and its advancements for IoT and machine-to-machine connectivity, the blockchain will play an important role in connecting machines, exchanging data and providing decentralized security services. A great deal of research has already been done in the areas of intelligent transportation systems, smart car communication and sensor technologies [17], [24] with the help of blockchain technology. The article [23] presents a framework for secure smart city services such as a sustainable IoT-enabled sharing economy based on blockchain technology. This research area increasingly integrates and combines the IoT and blockchains. The CEVM blockchain has the capabilities to improve the IoT and sensor networks' data processing capabilities. Furthermore, it can definitely solve the computation resource problems and prove to be an alternative approach aside from the network designs discussed in [17].

7.4 Limitations of the study

This thesis presents an approach that brings more execution capabilities to the blockchain and at the same time reduces the hardware requirements needed for running a blockchain node. Test scenarios with challenging programs have been created to demonstrate the system's behavior. The tests were performed on a test network that was set up for measuring the behavior of the CEVM blockchain as effectively as possible. The approach is expected to bring the blockchain technology even closer to the IoT. For demonstrating the behavior on IoT, a Raspberry Pi has been used because of its hardware similarity.

It has to be kept in mind that all of these experiments are only simulation scenarios of possible real world applications. The devices used as well as the test applications might not fully represent the behavior and challenges of real IoT devices. Especially devices of sensor networks are often only battery powered.

Furthermore, the presented approach and its implementation are only a pro-

totype for demonstrating and testing the functionality in a limited setting. It is not guaranteed that this software is able to fulfill the requirements of every blockchain application, but it provides evidence that it can perform and accelerate certain applications of this kind.

8 Conclusion

8.1 Future work

One possible focus in future research within the scope of blockchains based on microservice cloud architecture can be set on improving the blockchain execution logic. The CEVM approach is designed to have a loosely coupled execution engine, which means that the current EVM can be replaced with another stack-based engine. The successor of the current EVM, called EWASM [10], has already been announced as part of Ethereum 2.0 [8], which will be the next major upgrade of the core Ethereum protocol. As of summer 2020, Ethereum 2.0 as well as EWASM are still under development. EWASM (Ethereum flavored WebAssembly) is a virtual machine based on WebAssembly (Wasm) [31] that conforms to the Ethereum standards, which in particular guarantee deterministic execution and metering of gas. Just like EVM, it is a stack-based virtual machine that uses a binary instruction set. Wasm was originally designed for fast execution of high-level programming languages in the browser. Using the advantages of Wasm on the blockchain enables the use of high-level programming languages like Go, C, C++ and Rust for the programming of smart contracts and further accelerates their execution. The EWASM technology might also improve the CEVM blockchain by replacing the current EVM running as a microservice.

Apart form this, future research might concentrate on parallel transaction processing. The implementation of the CEVM client has opened up the possibility of pursuing a parallel transaction execution approach. Based on the experience during the implementation, as briefly mentioned in section 7.2.1, a concurrent execution is technically possible. But further research is necessary to elaborate on a design to guarantee a correct behavior of parallel transactions in case of storage access order and block generation. A parallel transaction execution in the context of the CEVM blockchain might have an enormous impact on the transaction throughput.

Another point in the context of the CEVM approach, which is also stated as a problem of the presented approach, is the uncertainness of the *StorageResolver*. There might be a better and more reliable way of preprocessing the bytecode for determining the storage accesses.

Regarding the system in a greater context, the CEVM approach is just one part of a blockchain technology based on a microservice cloud architecture.

The concept of a blockchain cloud architecture could be extended in further work. Such work could include new concepts of consensus mechanisms or a cloud-oriented blockchain database. Moreover, the presented approach also paves the way for integrating database-like functionalities on the blockchain. Future research could focus on applying database structures, query languages or realizing database algorithms with the cloud based execution logic.

Beside the possible future investigations on improving the blockchain approach itself, there are many fields of applications in present and future technologies. Section 7.3 discusses numerous technical directions on which the presented approach could have an impact.
The CEVM concept brings the blockchain closer to the IoT. This enables new secure and distributed communication channels. Consequently, the approach might find applications in the scope of a smart world by supporting machine-to-machine communication as well as machine-to-human interaction in domestic environments, in everyday life or in industry.

8.2 Concluding remarks

The above chapters have presented a comprehensive introduction into the blockchain technology and have introduced a new approach of a blockchain system with a cloud-based execution environment. The underlying principles and key concepts as well as detailed technical insights have been explained. With a focus on the properties that currently limit data processing capabilities, an approach has been introduced that improves the execution capabilities of the Ethereum blockchain: A blockchain with a cloud-based Ethereum virtual machine (CEVM). The design and concepts of the approach have been elaborated on and an experimental prototype has been presented. Furthermore, an experimental analysis based on the prototype has been conducted and its results have been analyzed. Subsequently open questions, drawbacks, possible research directions and impacts on future technologies have been provided as well. Finally, the thesis discusses possible approaches and open research problems regarding the stability and performance of the CEVM blockchain.

This thesis successfully shows how the calculation capabilities of a blockchain system can be improved by redesigning the execution process and re-configuring the network. The results of the experimental analysis partic-

ularly exhibit the positive impact of the approach on IoT and devices with reduced computation resources. Hopefully, this approach will have an impact on future research and helps improving the blockchain technology.

9 Appendix

9.1 Source code reference

The complete source code is available at the University of Vienna gitlab server:

`https://git01lab.cs.univie.ac.at/thesis/a01302703-florian-jaklitsch`

9.2 Install instructions

This following sections (9.3 - 9.6) are describing all steps necessary for setting up and running the implementation. The same documentation can be found at the `readme.md` file inside the gitlab repository.

Setting up a testnetwork requires the following components: At least one CEVMGeth node and the two microservices (PreExec-Microservice and CEVM-Microservice).
The CEVMGeth node requires the hostnames or IP-Adresses and ports of both microservices.

9.3 CEVMGeth install and configuration

The CEVMGeth software is located at the `/go-ethereum` directory. The implementation is based on the fork of Geth (go-ethereum) with ewasm and EVMC support (so it is ready to support the future ewasm virtual machine of the upcoming Ethereum 2.0). See `https://github.com/ewasm/testnet` for more information.

9.3.1 Building the source

Building the CEVMGeth works exactly like building geth. For prerequisites and detailed build instructions please read the **Installation Instructions**

(`https://github.com/ethereum/go-ethereum/wiki/Building-Ethereum`) on the wiki.
Building geth requires both a Go (version 1.7 or later) and a C compiler. Regarding the CEVM microservices, it is adviced to use Go version 1.13 or later.

Move the `/go-ethereum` folder into the correct directory inside the GOPATH directory. You get the current GOPATH by typing `go env`. In our case `GOPATH=/home/blockchain/go`. Inside the GOPATH directory, if it does not exist yet, create the new sub directory: `/src/github.com/ethereum`. Here we are placing the `/go-ethereum` folder from the repository. So the final path is `$GOPATH$/src/github.com/ethereum/go-ethereum`.

Once the dependencies are installed and everything is placed in the correct directory, run

```
make geth
```

or, to build the full suite of utilities:

```
make all
```

inside the `$GOPATH$/src/github.com/ethereum/go-ethereum` directory.

9.3.2 Setup

Setting up a CEVMGeth node works the same as setting up the original geth node.

1. **Create a new workspace directory.** For example:
   `mkdir /home/blockchain/devnet/`
2. **Create a blockchain account:**
   `./build/bin/geth --datadir /home/blockchain/devnet account new`
   The resulting testaccount only for this install instruction is
   `0x8a2e6d7de32721d9c8e2d8f3444ba1844523af56`.
   We store the password in a password file:
   /home/blockchain/devnet/password.txt
3. **Create a genesis file with PoA consensus:**

   We create the genesis file with the Geth tool `puppeth` (this is only available if you used `make all` as build instruction)

```
./build/bin/puppeth

What would you like to do? (default = stats)
 1. Show network stats
 2. Configure new genesis
 3. Track new remote server
 4. Deploy network components
> 2

Which consensus engine to use? (default = clique)
 1. Ethash - proof-of-work
 2. Clique - proof-of-authority
> 2

How many seconds should blocks take? (default = 15)
> 5

Which accounts are allowed to seal? (mandatory at least one)
> 0x8a2e6d7de32721d9c8e2d8f3444ba1844523af56
> 0x

Which accounts should be pre-funded? (advisable at least one)
> 0x8a2e6d7de32721d9c8e2d8f3444ba1844523af56
> 0x

Specify your chain/network ID if you want an explicit one
> 9222
```

The block generation time can be set to any value greater 0. CEVM-Geth even works with 1 second of block generation but it is adviced to use something ≥ 5 seconds. The network ID can be any number. Our testnetwork gets the id 9222. Finally the genesis file is exported with:

```
What would you like to do? (default = stats)
 1. Show network stats
 2. Manage existing genesis
 3. Track new remote server
```

```
 4. Deploy network components
> 2

 1. Modify existing fork rules
 2. Export genesis configuration
 3. Remove genesis configuration
> 2
```

The genesis file in this instruction is called `genesis.json`

Move the genesis file to `/home/blockchain/devnet/`.

4. **Initialize the blockchain:** `./build/bin/geth --datadir \ /home/blockchain/devnet init /home/blockchain/devnet/genesis.json`

5. **Configuring the microservice hostnames**

   Set the correct microservice hostnames/ip-adresses and ports in the `cevmconfig.json` file. This file is located in the root of the project folder. It must always be in the same location where the `geth` build file is executed. E.g.: By default, after building the project, the `geth` execution file is located at /build/bin/. In this case the `cevmconfig.json` must be moved to /build/bin/.

   Finally it must have the following structure:

```
|--go-ethereum
    |--build
        |--bin
            |--geth
            |--cevmconfig.json
```

6. **Running the CEVMGeth node:** `./build/bin/geth --datadir /home/blockchain/devnet/ --vm.ewasm="/home/blockchain/go/src/github.com/ethereum/go-ethereum/hera/build/src/libhera.so,metering=true,fallback=true" --syncmode 'full' --port 30311 --rpc --rpcaddr '0.0.0.0' --rpcport 8545 --rpccorsdomain "*" --rpcvhosts="*" --rpcapi 'personal,db,eth,net,web3,txpool,miner' --networkid 9222 --unlock '0x8a2e6d7de32721d9c8e2d8f3444ba1844523af56' --password /home/blockchain/devnet/password.txt --mine console`

argument	description
`--vm.ewasm`	Requires EWASM library /hera/build/src/libhera.so
`--datadir`	The defined workspace
`--password`	File containing the password of the account
`--unlock`	Unlocks the specified account

9.4 CEVM Microservice / PreExec Microservice

Now we use the 2 repositories `cevm-microservice` and `preexec-microservice`

```
|--a01302703-florian-jaklitsch
   |--cemv-microservice
   |--preexec-microservice
```

Depending on the environment, these two repositories have to be copied to a server, cloud instance etc. first. The microservices are both setup and deployed the same way.

Ports: By default the microservices are using:

port	microservice
8546	cevm-microservce
8547	preexec-microservice

If different ports are required, then they must be changed in the `config.json` file inside the `/cevm-microroservice/config` or `/preexec-microservice/config` directory respectivly.

9.4.1 Building

Running the microservices requires Go (version 1.13 or later). *This instruction applies to both microservices!*

- **Building the microservice directly**

  CEVM Microservice:

  Change directory to `/cevm-microservice` and run the command `go build .`

  PreExec Microservice:

Change directory to `/preexec-microservice` and run the command `go build .`

- **Build as docker container**

  CEVM Microservice:

  Change directory to `/cevm-microservice` and run the command `docker build -t cevm-microservice .`

  Start it with
  `docker run -d -p 8546:8546 cevm-microservice`

  Port must conform with the one defined in `config.json` file.

  PreExec Microservice:

  Change directory to `/preexec-microservice` and run the command `docker build -t preexec-microservice .`

  Start it with
  `docker run -d -p 8547:8547 preexec-microservice`

  Port must conform with the one defined in `config.json` file.

9.4.2 Setup

After building the microservices, they can be started either directly by running `./cevm-microservice` and `./preexec-microservice` or by deploying it as docker container.
During development, the microservices can also be started by using the basic Go command within a go environment `go run main.go` (only for development).
Finally we can call `http://hostname:8546` and `http://hostname:8547` to check if the microservices are responding.
In the following the microservice hostnames/ips and ports must be set in the `cevmconfig.json` of the CEVMGeth as descriped in section 9.4.2 CEVMGeth - Setup Step 5.

9.5 CEVMCommander

This tool helps deploying and running the test contracts: CPUHeavy, IO-Heavy and TXpS. It offers a command line interface which allows to deploy and execute the contracts with custom inputs.

9.5.1 Setup

All pre-known variables can be configured in the "GLOBAL VALUES" area of the `main.py` file. Otherwise all the variables can also be set via the command line interface.

value	description
`HOST_PORT`	IP Adress of CEVMGeth node
`ACCOUNT`	Blockchain account address that is allowed to transact.
`CPUHEAVY_ADDRESS`	Address of the deployed CPUHeavy contract
`IOHEAVY_ADDRESS`	Address of the deployed IOHeavy contract
`TXPS_ADDRESS`	Address of the deployed Txps
`BLOCK_TIME`	The block generation time configured in the genesis file

Start the CEVMCommander by running `python main.py`. The interface accepts the following inputs:

input	Description
[q]	Exit the program
[w]	Set CEVMGeth node Hostname / IP Address and Port of form: http://:8545
[e]	Set CEVMGeth node account (Address) of form 0x...
[1]	Deploy / Execute CPUHeavy benchmark contract
[2]	Deploy / Execute IOHeavy benchmark contract
[3]	Deploy / Execute Transactions per second contract

Before running any deployment / execution of a contract, make sure the host and account values are set correctly! The interface also outputs the currently set values. It is not necessary to set the contract addresses if they are not yet deployed. When calling a contract, the CEVMCommander will deploy the contract and set its address automatically.

9.5.2 Deploying the test contracts manually without CEVMCommander

The benchmarks are located at the directory `/contracts`. The best way for manually deploying the contracts is using Truffle. The benchmarks are already included in truffle projects. We are now deploying the `CPUHeavy` benchmark for demonstration.

1. Change directory to `/contracts/CPUHeavy`

2. In the `truffle-config.js` file, configure `host, port, from` with your values.

parameter	Description
`host`	IP Adress of CEVMGeth node
`port`	RPC Port of CEVMGeth node
`from`	Account that is sending the contract.

3. Run `truffle migrate` for compiling and deploying the contract to the blockchain.

4. After the contract has been deployed successfully, run `truffle console` and reference the deployed contract inside the truffle console with `CPUHeavy.deployed().then((instance)=>{app = instance;})`. Now the contract is referenced as `app` and we are able to interact with it. Lets try sorting 5 simple values: `app.sort([33,55,11,22,77])`

List of Figures

List of Listings

References

[1] Alladi, T., Chamola, V., Parizi, R. M., and Choo, K. R. Blockchain applications for industry 4.0 and industrial iot: A review. *IEEE Access 7* (2019), 176935–176951.

[2] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (2018), ACM, p. 30.

[3] Bitcoin. `https://bitcoin.org`. [Online; accessed 26.10.2020].

[4] Dai, H.-N., Zheng, Z., and Zhang, Y. Blockchain for internet of things: A survey. *arXiv preprint arXiv:1906.00245* (2019).

[5] De Angelis, S., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A., and Sassone, V. Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain. vol. 2058, CEUR-WS.

[6] Dinh, T. T. A., Liu, R., Zhang, M., Chen, G., Ooi, B. C., and Wang, J. Untangling blockchain: A data processing view of blockchain systems. *CoRR abs/1708.05665* (2017).

[7] Ethereum: A global open-source platform for decentralized applications. `https://ethereum.org`. [Online; accessed 24.10.2020].

[8] Ethereum 2.0. `https://ethereum.org/en/eth2/` [Online; accessed 26.10.2020].

[9] Official go implementation of the ethereum protocol. `https://github.com/ethereum/go-ethereum`. [Online; accessed 24.10.2020].

[10] Ethereum flavored webassembly (ewasm). `https://github.com/ewasm/design`. [Online; accessed 26.10.2020].

[11] Gin web framework. `https://github.com/gin-gonic/gin`. [Online; accessed 25.10.2020].

[12] Grishchenko, I., Maffei, M., and Schneidewind, C. A semantic framework for the security analysis of ethereum smart contracts. In

International Conference on Principles of Security and Trust (2018), Springer, pp. 243–269.

[13] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., and Rosu, G. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)* (2018), pp. 204–217.

[14] Hyperledger FABRIC Documentation: Introduction - Smart Contracts. `https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.html#smart-contracts`, 2019. [Online; accessed 26.10.2020].

[15] IO, E. Eos. io technical white paper. *EOS. IO (accessed 26.10.2020) https://github.com/EOSIO/Documentation* (2018).

[16] Jnoub, N., and Klas, W. Detection of tampered images using blockchain technology. In *IEEE International Conference on Blockchain and Cryptocurrency* (May 2019), pp. 2–4.

[17] Kushch, S., and Castrillo, F. P. A rolling blockchain for a dynamic wsns in a smart city. *CoRR abs/1806.11399* (2018).

[18] Maroufi, M., Abdolee, R., and Tazekand, B. M. On the convergence of blockchain and internet of things (iot) technologies. *arXiv preprint arXiv:1904.01936* (2019).

[19] Mukhopadhyay, M. *Ethereum Smart Contract Development.* Packt Publishing, 2018. ISBN: 9781788473040.

[20] Nagothu, D., Xu, R., Nikouei, S. Y., and Chen, Y. A microservice-enabled architecture for smart surveillance using blockchain technology. In *2018 IEEE International Smart Cities Conference (ISC2)* (2018), IEEE, pp. 1–4.

[21] Panarello, A., Tapas, N., Merlino, G., Longo, F., and Puliafito, A. Blockchain and iot integration: A systematic survey. *Sensors 18*, 8 (2018), 2575.

[22] Pongnumkul, S., Siripanpornchana, C., and Thajchayapong, S. Performance analysis of private blockchain platforms in varying work-

loads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)* (2017), IEEE, pp. 1–6.

[23] Rahman, M. A., Rashid, M. M., Hossain, M. S., Hassanain, E., Alhamid, M. F., and Guizani, M. Blockchain and iot-based cognitive edge framework for sharing economy services in a smart city. *IEEE Access 7* (2019), 18611–18621.

[24] Sharma, P. K., Moon, S. Y., and Park, J. H. Block-vn: A distributed blockchain based vehicular network architecture in smart city. *JIPS 13* (2017), 184–195.

[25] Sharma, S., RV, R., and Gonzalez, D. *Microservices: Building Scalable Software.* Packt Publishing, 2017. ISBN: 9781787285835.

[26] Sun, H., Hua, S., Zhou, E., Pi, B., Sun, J., and Yamashita, K. Using ethereum blockchain in internet of things: A solution for electric vehicle battery refueling. In *International Conference on Blockchain* (2018), Springer, pp. 3–17.

[27] Ullrich, J., Stifter, N., Judmayer, A., Dabrowski, A., and Weippl, E. Proof-of-blackouts? how proof-of-work cryptocurrencies could affect power grids. In *Research in Attacks, Intrusions, and Defenses* (Cham, 2018), M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., Springer International Publishing, pp. 184–203.

[28] Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R., and Wang, F.-Y. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)* (2018), IEEE, pp. 108–113.

[29] Wang, W., Hoang, D. T., Xiong, Z., Niyato, D., Wang, P., Hu, P., and Wen, Y. A survey on consensus mechanisms and mining management in blockchain networks. *CoRR abs/1805.02707* (2018).

[30] Web3. `https://github.com/ethereum/web3.js/`. [Online; accessed 26.10.2020].

[31] Webassembly specification: Release 1.1. `https://webassembly.github.io/spec/core/`. [Online; accessed 19.06.2020].

[32] Wood, G. Ethereum: A secure decentralised generalised transaction ledger (eip-150 revision). *Ethereum project yellow paper* (2014), 1–32.

[33] Xiao, Y., Zhang, N., Lou, W., and Hou, Y. T. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials* (2020).

[34] Xu, R., Nikouei, S. Y., Chen, Y., Blasch, E., and Aved, A. Blendmas: A blockchain-enabled decentralized microservices architecture for smart public safety. *arXiv preprint arXiv:1902.10567* (2019).

[35] Zhang, R., Xue, R., and Liu, L. Security and privacy on blockchain. *ACM Computing Surveys (CSUR) 52*, 3 (2019), 1–34.